IBM VisualAge C++ for OS/2

S25H-6961-00

IBM VisualAge C++ for OS/2 User's Guide

Version 3.0

IBM VisualAge C++ for OS/2

S25H-6961-00

IBM VisualAge C++ for OS/2 User's Guide

Version 3.0

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xxx.

Fifth Edition (May 1995)

This edition applies to Version 3.0 of IBM VisualAge C++ for OS/2 (Programs 30H1664, 30H1665, 30H1666) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. Please see "Communicating Your Comments to IBM" for a description of the methods. This page immediately precedes the Readers' Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM’s licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, USA.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Ave E., North York, ONT Canada M3C 1H7. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Programming Interface Information

This book is intended to help you create programs using VisualAge C++ product. It primarily documents General-Use Programming Interface and Associated Guidance Information provided by VisualAge C++ product.

General-Use programming interfaces allow the customer to write programs that obtain the services of VisualAge C++ compiler, debugger, browser, execution trace analyzer, and class libraries.

However, this book also documents Diagnosis, Modification, and Tuning Information. Diagnosis, Modification, and Tuning Information is provided to help you debug your programs.

Warning: Do not use this Diagnosis, Modification, and Tuning Information as a programming interface because it is subject to change.

Diagnosis, Modification, and Tuning Information is identified where it occurs by an introductory statement to a chapter or section.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

BookManager
C/2
C Set/2
C Set ++
Common User Access
CUA
IBM
LibraryReader
Open Class
Operating System/2
OS/2
OS/2 Warp
Personal System/2
PS/2
Presentation Manager
Systems Application Architecture
SAA
VisualAge
WorkFrame
WorkPlace Shell
System Object Model
SOM

Windows is a trademark of Microsoft Corporation.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks of others.

About This Guide

This guide tells you how to use the IBM VisualAge C++ Version 3.0 for OS/2 product (referred to throughout the book as VisualAge C++) to:

- develop and organize
- edit
- compile
- link
- debug
- analyze
- browse
- manage libraries for
- internationalize
- add application resources to

your C and C++ programs on the 32-bit Operating System/2* (OS/2 *) system. For information on miscellaneous tasks not in the above list, see Part 12, “Additional Utilities You May Find Useful” on page 811.

For information on using the Visual Builder component of VisualAge C++, see the *Visual Builder User's Guide*.

Who Should Read This Guide

This guide is written for application and systems programmers who want to use VisualAge C++ product to develop and run C or C++ applications. It assumes you have a working knowledge of the C or C++ programming language, the OS/2 operating system, and related products.

How to Read Syntax Diagrams

This book uses two methods to show syntax. One is for commands, preprocessor directives, and statements; the other is for options.

Syntax for Commands, Preprocessor Directives, and Statements

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►—— symbol indicates the beginning of a command, directive, or statement.

  The ——► symbol indicates that the command, directive, or statement syntax is continued on the next line.

The ►─── symbol indicates that a command, directive, or statement is continued from the previous line.

The ───►◄ symbol indicates the end of a command, directive, or statement.

Diagrams of syntactical units other than complete commands, directives, or statements start with the ►─── symbol and end with the ───► symbol.

Note: In the following diagrams, `STATEMENT` represents a C or C++ command, directive, or statement.

- Required items appear on the horizontal line (the main path).

```
►►──STATEMENT──required_item──────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──STATEMENT──┬───────────────┬──────────────────────────────────►◄
               └─optional_item─┘
```

- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──STATEMENT──┬─required_choice1─┬───────────────────────────────►◄
               └─required_choice2─┘
```

  If the items are optional, the entire stack appears below the main path.

```
►►──STATEMENT──┬──────────────────┬───────────────────────────────►◄
               ├─optional_choice1─┤
               └─optional_choice2─┘
```

  The item that is the default appears above the main path.

```
               ┌─default_item───┐
►►──STATEMENT──┴─alternate_item─┴─────────────────────────────────►◄
```

- An arrow returning to the left above the main line indicates an item that can be repeated.

```
               ┌────────────────┐
               ▼                │
►►──STATEMENT────repeatable_item┴─────────────────────────────────►◄
```

  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

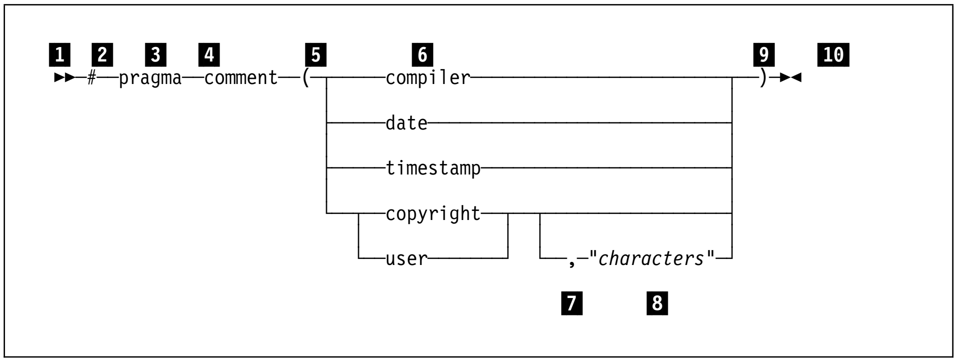
- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `pragma`).

  Variables appear in italicized lowercase letters (for example, *`identifier`*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Note: The white space is not always required between tokens, but it is recommended that you include at least one blank between tokens unless specified otherwise.

The following syntax diagram example shows the syntax for the `#pragma comment` directive. (See the *Language Reference* for information on the #**pragma** directive.)



The syntax diagram is interpreted in the following manner:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword `pragma` must appear following the # symbol.

4 The keyword `comment` must appear following the keyword `pragma`.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type.

8 A character string must follow the comma.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the #**pragma** `comment` directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Syntax for Compiler Options

- Optional elements are enclosed in square brackets [ ].
- When you have a list of items from which you can choose one, the logical `OR` symbol (|) separates the items.
- When you have a list of items from which you **must** choose one, the list of items is enclosed in angle brackets (< >).
- Variables appear in italicized lowercase letters (for example, *num*).

Examples

Syntax	**Possible Choices**

`/L[+|-]`	`/L`
	`/L+`
	`/L-`

`/Lt"`*`string`*`"`	`/Lt"Listing File for Program Test"`

Note that, for options that use a plus (`+`) or minus (`-`) sign, if you do not specify a sign, the plus is assumed. For example, the `/L` and `/L+` options are equivalent.

What's New

What's New with WorkFrame

WorkFrame Version 3.0 has many enhancements from its predecessor, Version 2.1, mostly having to do with usability. Immediately you'll notice that project containers have changed to please those who prefer the Version 1.1 Presentation Manager* menu-bar interface, but retains object-oriented characteristics for those partial to the Workplace Shell. There is also no separate WorkFrame folder; all WorkFrame objects are now located in the VisualAge C++ folder (although you can create WorkFrame projects anywhere on your system).

Here is a summary of what's new with WorkFrame this release:

- Project icon view containers now have a menu bar. The project container is split into two parts: the upper half holds project files and objects, and the lower half holds the project's **Monitor** where output from tools is displayed. By default, the monitor is hidden until a monitored action is started, or until you push the **Show** button on the project's tool bar. You push the same button to hide the monitor. 📖 See “Icon View” on page 15 for a more detailed description of the new project container.
- The project container has two toolbars, a configurable one for frequently used actions and another for monitor controls.
- Project-scoped actions are now available from the **Project** pull-down menu as well as pop-up menus, and the toolbar. File-scoped actions are available from the **Selected** pull-down menu, and pop-up menus on project files. 📖 See “Action Settings - Menus Page” on page 63 for more information on how to add your actions to these controls.
- A project's **Tools setup** consists of a tree view of actions, a list of environment variables, and a list of types that apply to the project. The **Tools setup** replaces the actions profile of Version 2.1. Actions, environment variables, and types are now part of the project and no longer exist in a separate actions profile object. 📖 See Chapter 3, “The Project Tools Setup” on page 45 for more information on the **Tools setup** window.
- Projects can share a **Tools setup** by inheriting another project's **Tools setup**. Any changes made to the base project's **Tools setup** are reflected in the projects that inherit from it. Inheritance can optionally be extended to include action options, so that any options changed in the base **Tools setup** are also changed in

the projects that inherit from it. See “Inheriting a Tools Setup” on page 33 for more information on project inheritance.

- Projects can now be nested, doing away with the separate concepts of base and composite projects in Version 2.1. In this version, *base project* refers to the parent project in a parent-child inheritance relationship.

- Consequently, projects can now contain any Workplace Shell objects, like a printer or OS/2 Window, along with project files and other WorkFrame projects. However, WorkFrame actions cannot be invoked on Workplace Shell objects in a project unless they are files, shadows of files, or other WorkFrame projects. See “Project Parts” on page 13 for a discussion about the objects a project can contain.

- Version 3.0 now has an powerful **Build** facility that removes the nuisance of creating and maintaining a make file by dynamically generating the information needed to build the project target each time a build is initiated. You can invoke the Build utility from the command line, and from the project toolbar and menus. The traditional **Make** facility which runs on a pre-generated makefile is still available. See Chapter 5, “Building Your Target” on page 91 to learn how to use the WorkFrame Build utility.

- MakeMake, WorkFrame's make file generation utility, can now be invoked from the command line with new command-line options.

- A project catalog called Project Smarts provides highly-customizable project templates that contain ready-to-use code skeletons for various kinds of applications, such as Presentation Manager and User Interface Class Library programming. You can extend this catalog or provide one of your own. Since REXX scripts control the instantiation of these projects, you have full power and flexibility to customize on the fly. See Chapter 6, “Project Smarts” on page 111 for more information on how to create projects from Project Smarts, and how to add Project Smarts templates of your own.

- WorkFrame now supports regular expressions, and other programmable types, along with file masks. Types are now classed into an extendible set of type classes that includes file masks, regular expressions, and logical and exclusion types. See “Types” on page 75 for a complete list of the type classes provided with WorkFrame, and an explanation of how to use them.

- Projects can now have multiple Project Access Methods (PAMs). This feature enables a project to contain objects from different file systems, remote locations, or library systems, and run remote or special-purpose actions supported by a registered PAM. This important feature enables programmers to develop code in the WorkFrame environment regardless of source, target platform, or language. Most users do not need to become familiar with the way PAMs work to use WorkFrame effectively. If you need to use PAMs from another solution-provider, work with objects other than OS/2 files, or if you are interested in writing a PAM of your own, you will want to read 🕮 Chapter 8, “Project Access Methods (PAMs)” on page 149 for more information about PAMs and using projects with multiple PAMs.

- A new project migration utility helps you migrate your WorkFrame Version 2.x or Version 1.x projects to Version 3.0 projects. 🕮 See Chapter 7, “Migrating Old Projects” on page 143 for more information on how the project migration utility works.

Compiler Changes

The following changes have been made to the compiler:

- You can now use multiple and nested response files with the compiler. Previously you could only specify one response file. See “Using Response Files” on page 203 for more information on using response files.
- Most options now have local scope; an option applies only to files that appear after it on the command line. See “Scope of Compiler Options” on page 257 for a complete description of compiler scope.
- You can no longer use any `/K` option. Use the equivalent `/W`*`grp`* option instead.
- You can no longer keep temporary files in shared memory with the `/Fd` option. Temporary files are now always saved to disk.
- You can now give directory names when you specify files with the following options:
  - `/Fa`
  - `/Fl`
  - `/Fo`
  - `/Fw`
- Direct to SOM support has been added to the compiler. The following options are now available:
  - `/Fr`
  - `/Fs`
  - `/Ga`
  - `/Gb`
  - `/Gz`
  - `/Xs`
- You can add line-number-only debugging information with the `/Tn` option.

- You can include additional levels of browse information with the /Fb option.
- You can optimize for size as well as speed with the /Oc option.
- Specify the /Gp option to support **__parmdwords** on system linkage. Previously, **__parmdwords** was supported by default.
- You can remove unreferenced functions by compiling and linking with the /Gl option.
- You can link in old object files (created by versions of the compiler before VisualAge C++ version 3.0) with the /Gk option.

Additional changes to the compiler that do not affect its options are documented in the *Programming Guide* cp 10

Linker Changes

LINK386 has been replaced by VisualAge C++ linker. VisualAge C++ linker has the following differences from LINK386:

- VisualAge C++ linker accepts object files compiled or assembled:
  - In 16- or 32-bit OMF format
  - In TIS OMF or IBM OMF format
  - For OS/2 version 1.0 or higher
  - For the 80286 (16-bit only), 80386, 80486, and Pentium microprocessors
- VisualAge C++ linker has a new, more flexible, command line syntax. See "Linking from the Command Line" on page 323 for more information. You can use a LINK386-compatible syntax instead by specifying /NOFREE. See "Using LINK386 Syntax" on page 325 for more information.
- You can use multiple response files with ILINK, but you cannot nest response files (you cannot use multiple response files with /NOFREE).
- Blank lines in a response file are treated as input (the linker will proceed to the next prompt). This includes blank lines at the beginning and end of the response file.
- You can specify options with either a slash (/) or a dash (-) as in /DEBUG or -DEBUG.
- You must separate each option from preceding text with a space or tab.
- ILINK generates a fatal error if you have unmatched quotes on the command line or in a response file.
- ILINK stops linking when it cannot find a file you specify.

- ILINK has a return code that is the sum of:

  4 For any warning messages
  8 For any error messages
  16 For any fatal errors

  You can force NMAKE to ignore return codes less than 8 by putting `-7` before the linker command in your makefile. See “Linker Return Codes” on page 342 for more information on return codes.

- VisualAge C++ linker can read the library format produced by VisualAge C++ Version 3.0, as well as the library format used by previous versions.

- VisualAge C++ linker does not allow other processes to open the .DEF file for update during a link.

- Constant segments are now in `CONST32_RO`, which is set to READONLY and SHARED. You can redefine these attributes in a .DEF file or with the `/SECTION` option.

- `/ALIGNMENT` cannot be set higher than 4096.

- VisualAge C++ linker supports browse information for VisualAge C++ with the `/BROWSE` option.

- `/MAP` has changed:

  - It has a *name* parameter, that allows you to name and direct the map file (this parameter is ignored if you specified `/NOFREE`).

  - It gives a complete map file; you no longer need to specify `/MAP:full`.

- `/NOIGNORECASE` is now the default, which you can override with `/IGNORECASE.`

- `/NOLOGO` now suppresses the echoing of text from a response file, as well as suppressing the product information at the start of the linking.

- VisualAge C++ linker has added options which reduce the need for a .DEF file:

  `/DLL`
  Produce a .DLL file

  `/EXEC`
  Produce an .EXE file

  `/PDD`
  Produce a physical device driver

  `/SECTION`
  Set section or segment attributes

  `/VDD`
  Produce a virtual device driver

- VisualAge C++ linker has added new optimization options:

  `/OPTFUNC`
  : Removes unreachable functions (for object file compiled with `/Gl`)

  `/EXEPACK:2`
  : Pack file for OS/2 version 3.0 or later

  `/DBGPACK`
  : Pack debugging information (use with `/DEBUG`)

- VisualAge C++ linker has also added the following new options:

  `/FORCE`
  : Create executable output file even if errors

  `/FREEFORMAT`
  : Use new command-line syntax (the default)

  `/NOBASE`
  : Emit internal fixups

  `/NOFORCE`
  : Do not create executable output file if errors (the default)

  `/NOFREEFORMAT`
  : Use LINK386-compatible command line syntax

  `/NOOLDCPP`
  : Resolve templates with linker (default)

  `/NOOPTFUNC`
  : Do not perform new optimization (the default)

  `/OLDCPP`
  : Resolve templates in old object code (created before VisualAge C++ version 3.0)

  `/OUT`
  : Name and direct output file (incompatible with `/NOFREE`)

- Most options now have a positive and negative setting. To accomplish this, ILINK has added the following options, which were previously LINK386 defaults you could not specify directly:

  – `/NOCODEVIEW`
  – `/NODEBUG`
  – `/DEFAULTLIBRARYSEARCH`
  – `/NOEXEPACK`
  – `/EXTDICTIONARY`
  – `/IGNORECASE`
  – `/NOINFORMATION`

  - /NOLINENUMBERS
  - /LOGO
  - /NOMAP
  - /PACKDATA

- Some LINK386 options are no longer supported. The following options are no longer available:

  /BATCH
  : There are no pauses to suppress.

  /DOSSEG
  : Segment ordering is on when you build .EXE or .DLL files, and off when you build device drivers. You no longer need to specify the option.

  /FARCALLTRANSLATION
  : No longer required.

  /NONULLSDOSSEG
  : No longer required.

  /PAUSE
  : No longer required.

  /RUNFROMVDM
  : No longer required.

  /WARNFIXUP
  : No longer required.

Performance Analyzer Changes

The following features are new with this release of VisualAge C++:

Performance Analyzer - Window Manager window

The **Performance Analyzer - Window Manager** window is the control window for the Performance Analyzer. From this window, you can start most Performance Analyzer functions. For instance, you can:

- Start creating a new trace file
- Start analyzing an existing trace file
- Open and close a diagram.

Trace On and Trace Off push buttons

These buttons, which appear on the **Application Monitor** window, let you start and stop the trace of your program.

WorkFrame integration

If you have started the Performance Analyzer from WorkFrame, you can:

- Display Help information for library functions, such as IBM Open Class functions, OS/2 system functions, and C runtime functions.

- Start the WorkFrame editor from a Performance Analyzer diagram and edit your source code.
- Start other programs from the Performance Analyzer.

Tracing capability in dynamic link libraries

In addition to tracing functions in the executable file, the Performance Analyzer can trace your program's activity in:

- Statically or dynamically linked Dynamic link libraries (DLLs).
- The following system libraries:
  - DOSCALL.DLL
  - PMGPI.DLL
  - PMWIN.DLL
- Dynamically linked load-on-call DLLs. If you only want to trace a load-on-call DLL, the **Trace Generation** window will not have any executables or DLLs listed in the window, and you will receive an informational message.

Tracing capability for up to 64 threads

The Performance Analyzer can trace up to 64 threads. The diagrams show activity on all or selected threads.

Pop-up menus

Clicking mouse button two in most diagrams displays pop-up menus that let you quickly access frequently used functions.

Time find capability in Call Nesting

The **Call Nesting** diagram has a search capability that lets you go to specific times in the trace file.

Time Line diagram

The Performance Analyzer can display user events in the **Time Line** diagram.

Status Area

Each diagram has a Status Area, which shows you detailed information about the trace file data.

Vertical Ruler

Many diagrams have a Vertical Ruler that shows your location in the trace file.

Debugger Changes

The following describes the new features that have been added to the debugger since the previous release.

Deferred breakpoints

Allows you to set a breakpoint in a DLL that is not currently loaded. If your application consists of DLLs that are dynamically loaded, use this feature to set breakpoints in the dynamically loaded DLLs that have not been loaded yet. These deferred breakpoints become active once the DLL is loaded.

Child process debugging
Supports debugging of processes started by a parent program.

Exception filtering
Allows you to select the exceptions that you want the debugger to recognize. An exception occurs when your application is unable to interpret specific requests.

Check heap when stopping
Helps to isolate memory management problems by checking for memory overwriting each time your program stops executing.

Hide debugger on Run
Hides the debugger windows while your application is running.

Color support
Allows you to change the color of the various window elements such as executable lines, non-executable lines, and the breakpoint prefix area.

SOM support
Allows you to debug SOM objects created with the compiler using Direct-to-SOM support or created with the SOM compiler. Support includes monitoring SOM classes in the monitor windows.

Scroll to line number
Allows you to scroll to a particular line number in the source code. This feature also provides the ability to set breakpoints.

Autosave window positions and sizes
Saves the window positions and sizes when the windows are moved or re-sized. Alternatively, you may save the window positions and sizes by position to the debugger windows on the desktop and selecting the **Save window positions and sizes** choice.

Integration
Provides quick and easy access to other tools such as an editor or the browser. This feature is available when the debugger is started from within the WorkFrame/2 environment.

Select include files
Allows you to select the include files you want to view. Include files are files that are included in your source file by a compiler directive and are considered program source files.

Windows menu
Displays a list of all the active debugger windows.

Hover help for title bar buttons

Displays the name of the title bar button when you place your mouse pointer on the button. If you drag the mouse pointer across the buttons, the name in the title bar area changes to reflect the button you are on.

The following describes the enhanced features that have been added to the debugger since the previous release.

Call stack window

Provides the option of displaying the remaining stack size, the stack frame size, the return address, the ESP value and the EBP value.

Breakpoint list window

Displays as a window allowing you to continuously monitor the breakpoint list. You can also display the source code where the breakpoint is set.

Storage window

Allows you to monitor expressions in a storage window. For example, if you are monitoring a pointer, as the pointer changes, the storage window changes to show the new location referenced by the pointer.

Change address breakpoint

Allows you to set a change address breakpoint by typing in an expression.

Enable program profiling

Allows you to enable or disable the use of program profiles which restores a program's breakpoints, source windows and monitors to the same state as when last debugged.

Browser Changes

This section summarizes the differences between the IBM VisualAge C++ Browser Version 3.0 and its predecessor, the IBM C Set ++ Browser Version 2.1. It also describes changes made to this document, from the previous version S61G-1367.

Changes to the Product

- The Browser Control window and the Text window have been removed. See Chapter 44, “Understanding and Using the Browser User Interface” on page 563 on how VisualAge C++ Browser functions are now handled.
- You can now identify System Object Model (SOM) classes and metaclasses with different shapes and colors.

- All Browser windows have an Information Area at the bottom of the window used to define menu items as you highlight them. As well, some processing messages are displayed here.
- You can now print the List and Graph window contents or save them to ASCII and OS/2 bitmap formats, respectively.
- When loading a large program, a window appears telling you the percentage of the program loaded.
- As part of the redesign of the Browser to increase the ease of use, all command line options from Version 2.1 have been removed. You can start the Browser with one easy to remember command - **icsbrs**.
- You no longer have to create complex queries in order to browse your programs. The Browser provides all possible queries via Object PopUp menus. For more information on these menus, see "PopUp Menus" on page 648.
- You can quickly browse the targets of your IBM WorkFrame projects, without having to recompile, by using the QuickBrowse facility. See "Browsing without Recompiling" on page 602 for more information.
- Quick access to the classes that make up the IBM VisualAge C++ Open Class Library through the **Load** ► and **Merge** ► Cascade menus.
- Quick access to all the IBM VisualAge C++ documentation for the classes and functions that make up the IBM VisualAge C++ Open Class Library.

Changes to this Publication

The changes made in this publication are:

- The format of the book has been changed.
- A fuller description of the user interface components has been added. See Chapter 44, "Understanding and Using the Browser User Interface" on page 563.
- Many section have been added to show you how to do some common Browser tasks. See Chapter 45, "Using the Browser" on page 601.
- A quick tour of the Browser has been added to show you some of the key features of the Browser. See Chapter 46, "A Tour of the Browser" on page 621.
- A fast-path keys and menu descriptions section has been added for quick reference. See Chapter 48, "Browser Fast-Path Keys and Menu Descriptions" on page 639.
- A trouble-shooting chapter to help you solve some problems you may encounter. See Chapter 47, "Trouble Shooting" on page 637.

How to Get Help

There are three kinds of online information available to you while you are using VisualAge C++:

Online documents
These are complete documents, like the one you are reading now, presented online. These documents contain detailed information on the different aspects of VisualAge C++. For your convenience, the online documents are presented in two formats:

- Standard format (`.INF` files). See “Getting Help Inside VisualAge C++” for instructions on opening standard format documents from inside VisualAge C++. See “Getting Help from the Command Line” on page xlviii for instructions on opening standard format documents from the command line. For a list of the VisualAge C++ documents that are available in standard format, see “Online Documents Available in VisualAge C++” on page xlix.
- BookManager format (on CD-ROM only). See “BookManager Books” on page xlix for details on how to access online documents in this format. For a list of the VisualAge C++ documents that are available in BookManager format, see “The IBM VisualAge C++ BookManager Library” on page 935.

Contextual help
Contextual help is available throughout VisualAge C++. This help tells you all about the elements that you see in the interface, including menus, entry fields, and pushbuttons.

How Do I **help**
Many of the common tasks that you want to perform with VisualAge C++ are described in *How Do I* help. The *How Do I* help for a task gives you step-by-step instructions for completing the task. There is overall *How Do I* help for VisualAge C++, as well as individual task lists for each of its components.

Getting Help Inside VisualAge C++

All three kinds of help are available directly within the VisualAge C++ interface:

- To get general contextual help for the component of VisualAge C++ that you are using, press **F1** anywhere in the window.
- To get contextual help on a particular menu, menu item, or button, highlight the element and press **F1**.

- To get access to all of the help information that is available to you in a particular window, click on **Help** in the menu bar at the top of the window. This menu includes the following selections:
  - **Help Index**, an alphabetical list of all of the help topics that are available from this window
  - **General Help**, overall help for the window
  - **Using Help**, general information about the help facility
  - **How Do I...**, the How Do I help for the component
  - **Product Information**, a dialog that shows the level of VisualAge C++ being used

  In addition, there are selections that let you open all of online documents that are available in VisualAge C++.
- To get detailed information, open the **Information** folder in the VisualAge C++ folder. In this folder you will find icons for a variety of online documents that describe, in detail, the different aspects of VisualAge C++. To open a particular online document, double click on its icon.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the `view` command. The installation routine stores the online document files in the `\IBMCPP\HELP` directory. To view the *Language Reference*, for example, make `C:\IBMCPP\HELP` your current directory (substituting the drive where you installed VisualAge C++ for `C:`) and enter the following command:

```
VIEW CPPLNG.INF
```

If you want to get information on a specific topic, you can specify a word or a series of words after the file name. If the words appear in an entry in the table of contents or the index, the online document is opened to the associated section. For example, if you want to read the section on operator precedence in the *Language Reference*, you can enter the following command:

```
VIEW CPPLNG.INF OPERATOR PRECEDENCE
```

Getting Help for a Keyword or Construct

If you are editing a file using Editor, you can get help for a keyword or construct by highlighting the word and pressing **F1**. In the other tools, you can get help for a keyword or construct by highlighting the word and pressing **Ctrl-H**.

BookManager Books

In addition to standard format, the online documents are also available in BookManager format in the CD-ROM version of VisualAge C++. You can read this information using either the IBM Library Reader/2 or IBM Library Reader/DOS. For details on installing and using the IBM Library Reader and BookManager books, see the `README.ENG` file in the root directory of the CD-ROM.

Online Documents Available in VisualAge C++

The following documents are available in standard format:

- *Building VisualAge C++ Parts for Fun and Profit*
- *C Library Reference*
- *Control Program Guide and Reference*
- *Graphics Programming Guide and Reference*
- *IPF Guide and Reference*
- *Kernel Debug Reference*
- *Language Reference*
- *Multimedia Application Programming Guide*
- *Multimedia Programming Reference*
- *Multimedia Subsystem Programming Guide*
- *Open Class Library Reference*
- *Open Class Library User's Guide*
- *OS/2 Bidirectional Language Support Development Guide*
- *OS/2 Tools Reference*
- *Presentation Manager Guide and Reference*
- *Programming Guide*
- *REXX Reference*
- *SOM Programming Guide*
- *SOM Programming Reference*
- *User's Guide*
- *Visual Builder User's Guide*
- *Visual Builder Parts Reference*
- *Editor Command Reference*
- *Welcome to VisualAge C++*
- *Workplace Shell Programming Guide*
- *Workplace Shell Programming Reference*

Part 1. Developing with WorkFrame

This part of the *User's Guide* explains how you can use the WorkFrame application development environment to manage your software projects.

Introducing WorkFrame

IBM WorkFrame is a unique and fully-customizable application development environment that puts all your tools at your fingertips. Fully-integrated with the OS/2 Workplace Shell, it offers an unobtrusive interface that simplifies the process of building and organizing software projects.

This chapter introduces you to Version 3.0 of WorkFrame. A brief overview is followed by a quick tutorial to give you a taste of working with WorkFrame. It also tells you how to get help as you work.

If you have used Version 2.1, you might want to read “What's New with WorkFrame” on page xxxvi for an overview of the new features in this release.

Overview

A natural extension to the OS/2 Workplace Shell Desktop, WorkFrame is an extendable collection of objects and actions that organize your code into *projects*.

The project is the core of the WorkFrame environment. It encapsulates all the objects you need to build a single target. It also has an associated set of *actions* (like Edit, Compile, and Debug) that you can easily access and customize.

You can organize complex applications into project hierarchies to give you a visual perspective of how your code is organized. You can perform a build from any point in a project hierarchy, giving you more control over the way you build your applications.

As you program, you will likely use other special tools, such as icon editors, visual builders and performance analyzers. WorkFrame makes these tools available to you as context-sensitive actions in pop-up and pulldown menus in projects, and from the **Project** pulldown of VisualAge C++ tools.

You can also customize projects with your own set of OS/2 (32- or 16-bit) tools, as well as any of your favorite DOS and Windows** tools. Any tools that you add to a project's set of action become just as accessible and seamlessly integrated as any VisualAge C++ action.

You set options for actions quickly and easily using graphical user interfaces. When you invoke a **Compile** action from a pop-up or pulldown menu, WorkFrame invokes the compiler with your preset options. A window called **Build Smarts** lets you quickly modify options to add debug and browse information to your preset options so that you can easily build your project for debugging and production scenarios.

The next section, "Working with Projects," takes you through a short Compile-Edit-Debug session with a sample WorkFrame project.

Working with Projects

Just to give you a flavor of what it's like working with WorkFrame projects, try this typical Compile-Edit-Debug scenario using the VisualAge C++ Touch sample project.

1. Open the **Sample Projects** folder in the VisualAge C++ folder. Open the **Compiler** samples folder. **Touch** appears among other sample projects.
2. Open the Touch project by double-clicking on it. The Touch project view shows you all the Touch source files. All the tools you need to work on your software project are accessible from pop-up menus in this view:

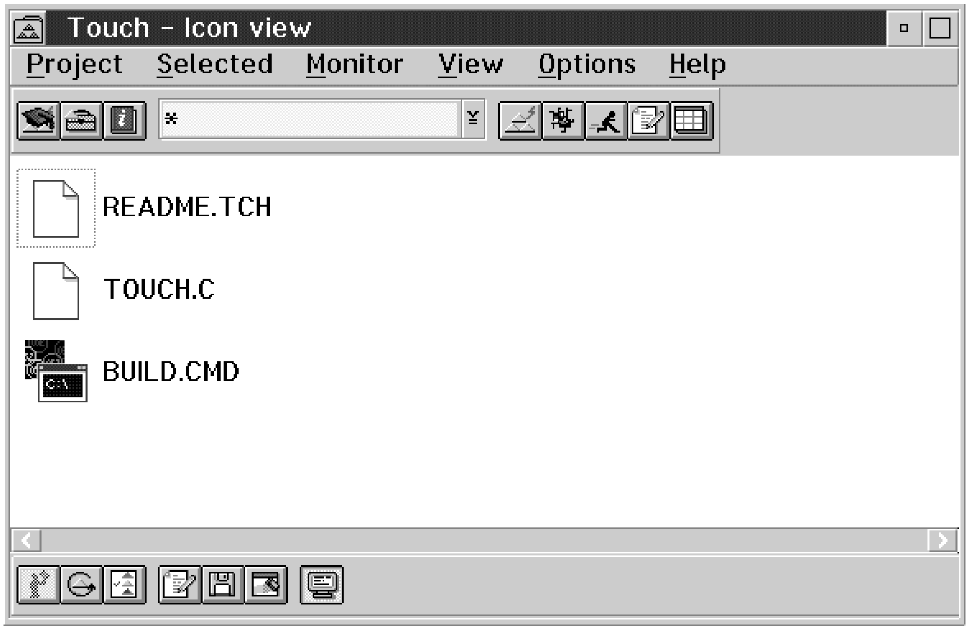


Figure 1. Touch Project - Icon View

3. Select the TOUCH.C file, and then press mouse button 2 to bring up its pop-up menu.

   You see a list of relevant actions for the TOUCH.C file, among them **Compile** and **Edit**. The actions that you can invoke are context-sensitive to the type of file you are invoking the action on. These actions are called *file-scoped*.

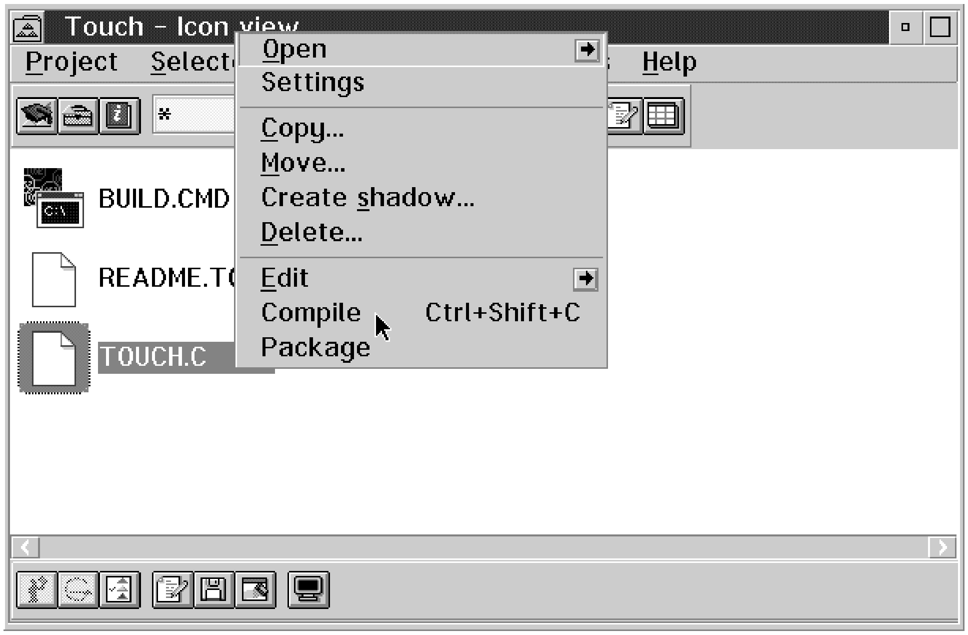


Figure 2. TOUCH.C pop-up menu

In the pop-up menu for the Touch README file, **Edit** and **Package** are the only applicable actions.

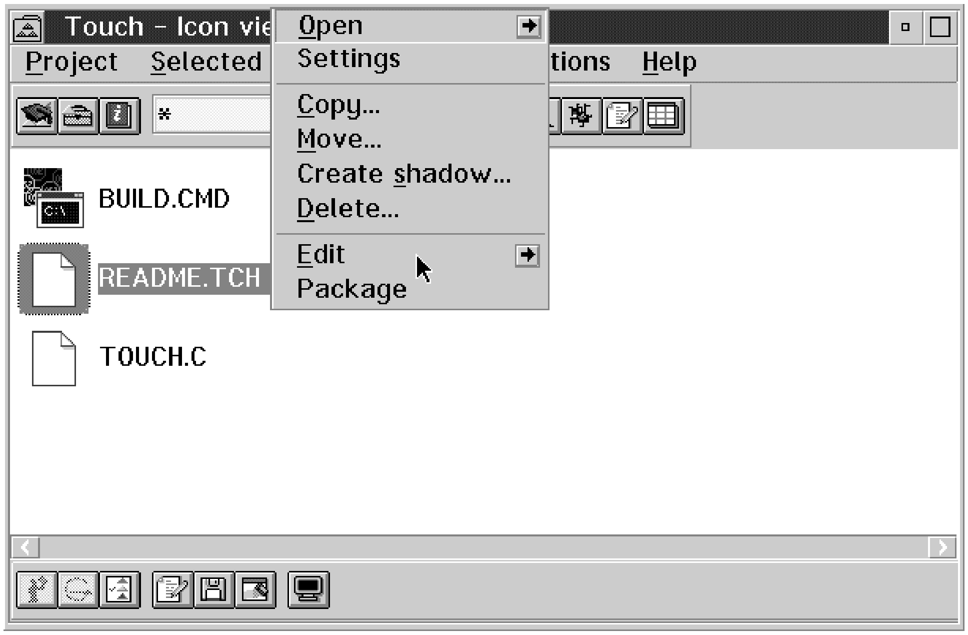


Figure 3. Touch README file pop-up menu

Actions that apply to the entire project are available from the **Project** pulldown or popup menu. They are called *project-scoped* actions.

4. Now, edit TOUCH.C by selecting **Edit** from the pop-up menu to use the default editor, or select a specific editor from the **Edit** cascaded menu.

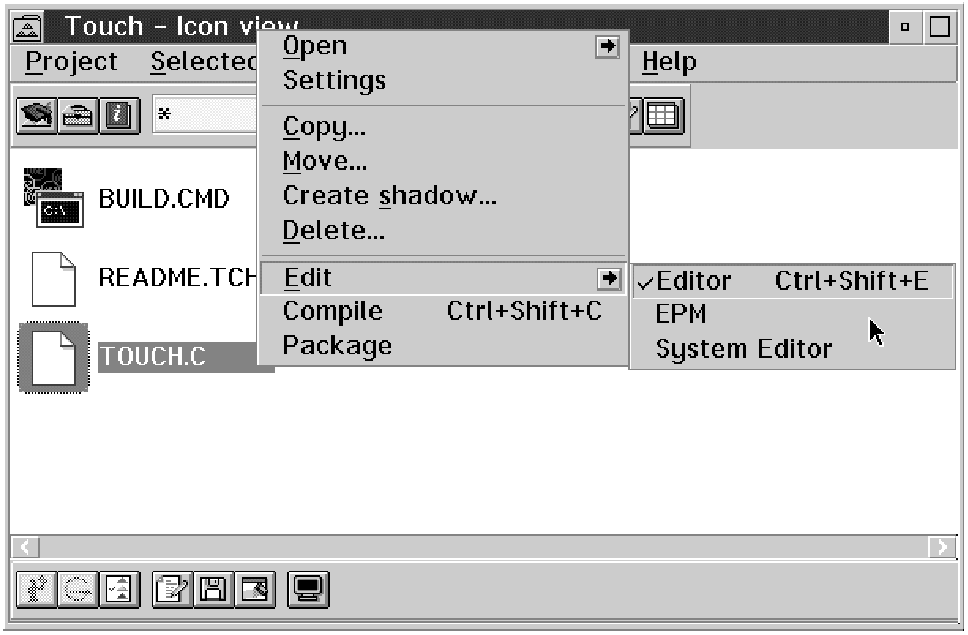
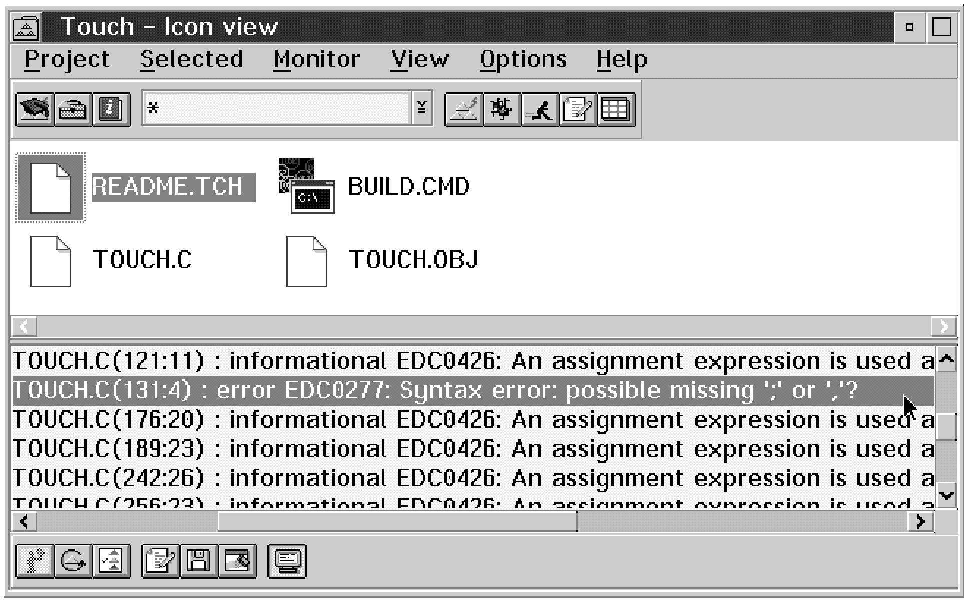


Figure 4. Edit cascading menu

The action classes on pop-up menus, like **Compile** and **Edit**, are further divided into more specific actions. From these menus, you could invoke one among the many compilers or editors that are available to your project. If you display the cascading choices for **Edit**, for example, you have a choice of editors. You can select one, or simply select the **Edit** menu item to invoke the default editor, the VisualAge Editor, in this project's initial configuration.

5. Scan through the file and insert a simple syntax error in the code (for instance, delete a semicolon or slightly change the name of a variable).

6. Save the file.
7. Now, return to the project and then compile TOUCH.C by selecting **Compile** from its pop-up menu. The output from the Compile action is displayed in the project **Monitor**, a list box that appears below the project container when an action like Compile or Link is started.

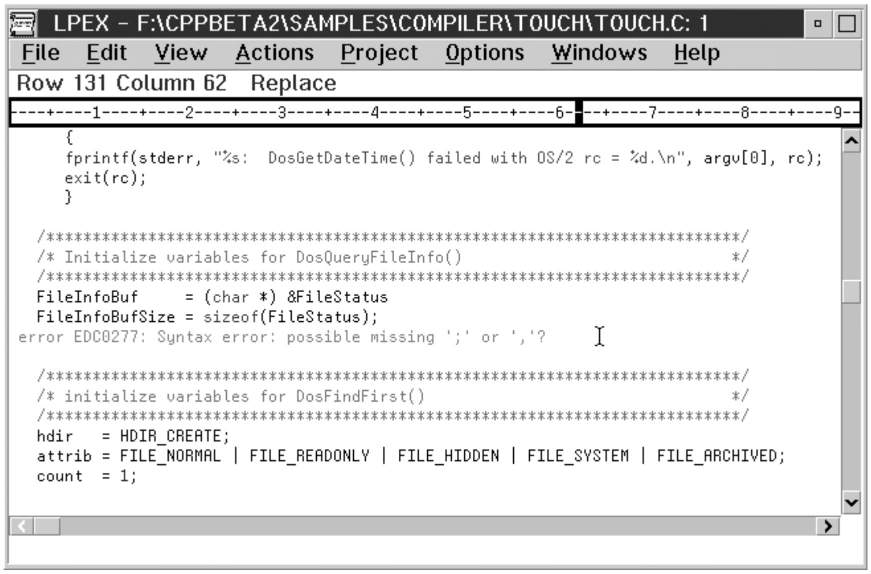


```
TOUCH.C(121:11) : informational EDC0426: An assignment expression is used a
TOUCH.C(131:4) : error EDC0277: Syntax error: possible missing ';' or ','?
TOUCH.C(176:20) : informational EDC0426: An assignment expression is used a
TOUCH.C(189:23) : informational EDC0426: An assignment expression is used a
TOUCH.C(242:26) : informational EDC0426: An assignment expression is used a
TOUCH.C(256:23) : informational EDC0426: An assignment expression is used a
```

Figure 5. WorkFrame project monitor

When the compilation is finished, you see an error message about the error you inserted into the code.

8. Double-click on the error in the WorkFrame **Monitor** listbox. The edit session is updated to show the line in the TOUCH.C source file where the error occurred.



Row 131 Column 62   Replace

```
      {
      fprintf(stderr, "%s:  DosGetDateTime() failed with OS/2 rc = %d.\n", argv[0], rc);
      exit(rc);
      }

   /**************************************************************************/
   /* Initialize variables for DosQueryFileInfo()                            */
   /**************************************************************************/
   FileInfoBuf     = (char *) &FileStatus
   FileInfoBufSize = sizeof(FileStatus);
error EDC0277: Syntax error: possible missing ';' or ','?

   /**************************************************************************/
   /* initialize variables for DosFindFirst()                                */
   /**************************************************************************/
   hdir   = HDIR_CREATE;
   attrib = FILE_NORMAL | FILE_READONLY | FILE_HIDDEN | FILE_SYSTEM | FILE_ARCHIVED;
   count  = 1;
```

Figure 6. Error line highlighted in editor

9. Correct the error, save the file, and then compile it again. There are three ways to do this:

   a. If you are using the VisualAge Editor, select **Compile** from the **Project** menu in the edit window.

   b. Select the **Repeat** pushbutton on the **Monitor**.

   c. Returning to the project container, invoke the **Compile** action as you did the first time.

   You will see the result of the second compilation in the same monitor window. The compilation should proceed without errors this time.

10. In the Touch project container, notice that TOUCH.OBJ has been added to the project's contents. Invoke the **Link** action from the TOUCH.OBJ pop-up menu.

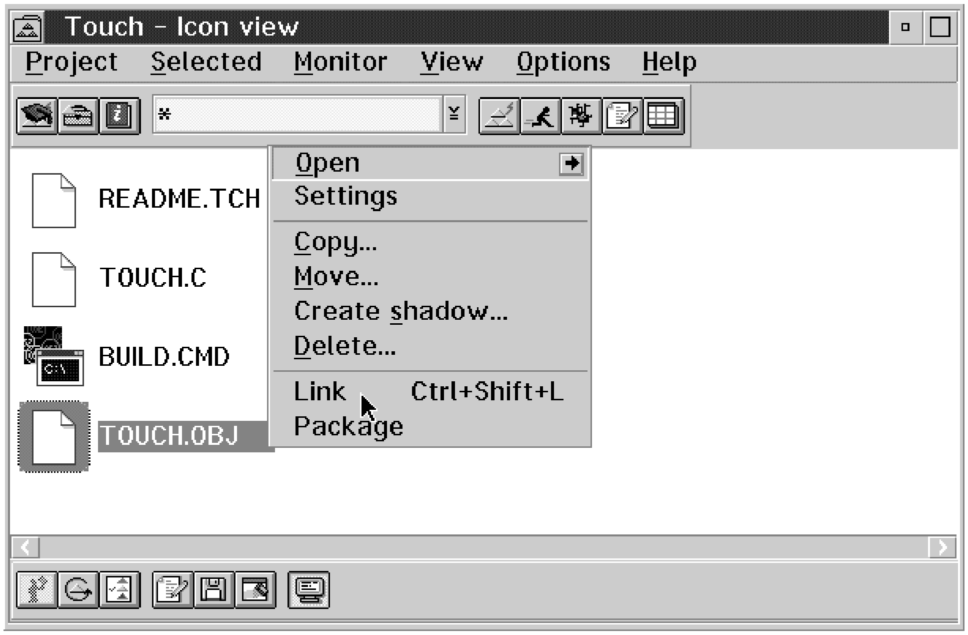


Figure 7. TOUCH.OBJ pop-up

Again, the result of the link is displayed in the **Monitor** window. After a successful link, TOUCH.EXE is added to the project's contents. From the TOUCH.EXE pop-up menu, you can select **Debug** to invoke the debugger on TOUCH.EXE, or **Analyze** to tune its performance.

The Build action is a way to compile and link a number of files in a single step to produce an executable. (📖 See “The Build Utility” on page 92 for more details about the WorkFrame Build utility.) This section took you through an Edit-Compile-Debug cycle the long way to illustrate the object-action paradigm of the WorkFrame interface. You will likely use the Build action more often to produce your project target.

WorkFrame Introduction

Now that you have a little taste of working with WorkFrame projects, you might want to know more. The rest of this guide will explain, in detail, everything you can configure in a project, how you can integrate your own tools, set options, organize project hierarchies, create make files, invoke builds, and more.

Understanding WorkFrame

"Working with Projects" on page 4 gave you a feel for the Workplace Shell-like interface of the WorkFrame project. Using WorkFrame's object-action approach to direct manipulation, you selected an object and invoked an appropriate action directly from its popup menu.

The basic object of interest in the WorkFrame development environment is the project. Any given project can contain objects of many different types. WorkFrame takes care of activating the appropriate action to process these types as desired. The difference is that you no longer need to think in terms of running different applications for each type. You can concentrate on the objects of the application itself since the available functionality is integrated in the WorkFrame environment, context-sensitive to each object type.

Settings for projects and other WorkFrame objects are accessible from a **Settings** notebook, as with any other Workplace Shell object. In addition to project files, pop-up menus also apply to actions, environment variables, and types. 📖 These are discussed fully in the next chapter, Chapter 2, "Managing Projects" on page 13.

Since no single integrated development environment is likely to satisfy all software developers, WorkFrame is not a tightly-integrated development environment. Rather, WorkFrame's loosely-integrated, open environment allows the plug-and-play addition of new or upgraded tools without changing the rest of the environment. Using WorkFrame, you can organize everything from your software projects, to your LaTeX documents.

The next few chapters will show you the many ways you can customize projects, integrate your own tools, and create ready-to-use application templates from Project Smarts, an extendible application wizard.

Getting Help

You can get two kinds of online help while using WorkFrame:

- Contextual online help, which gives you help from within WorkFrame
- **How Do I?** help, which gives you step-by-step instructions on how to perform several project-related tasks.

Using Contextual Help

Help on how to use any menu choice, window, or control is available through online context-sensitive help. You can access it in one of the following ways:

- Select an item from a **Help** pop-up or pulldown menu in any WorkFrame window. The Help menu in a project container gives you access to VisualAge C++ Manuals, **How Do I?** information, and context-sensitive help for WorkFrame.
- Press **F1** from any WorkFrame window to get help on the current field.
- Highlight a menu item and press **F1** to get help on the menu item.
- Highlight the name of an action on a pop-up or pull-down menu and press **F1** for help on the action, where supported.
- Click on the **Help** pushbutton, where available.

Using How Do I? Information

Refer to **How Do I?** help when you need to accomplish a specific task, or when you want to explore WorkFrame functions.

You can access the **How Do I?** information in a number of ways:

- From the **Help** pulldown menu, select **How Do I?** → **WorkFrame**.
- Open the **How Do I?** folder located in the main VisualAge C++ folder on your desktop, and select the **WorkFrame How Do I?** icon.

2 Managing Projects

This chapter explains how you can organize your own WorkFrame projects. WorkFrame projects are highly customizable, and you can easily tailor your development environment to suit your needs.

Introducing Projects

The *project* represents the complete set of data and actions you need to build a single *target*, such as a dynamic link library (DLL) or executable file (EXE). It consists of a set of *project parts* and a *Tools setup*.

Project Parts

Project parts are the data objects that make up the project. As well as a source file, a project part may also be a transient object, such as a target or intermediate object file created during the life of the project. A project part may also be another project. Including a project as a part of another project enables you to model hierarchies of projects.

A target is a specially designated project part. It is the result of a build action invoked on the project. 📖Builds are discussed in “The Build Utility” on page 92.

A project's parts are conceptually contained in the project; they are not physically stored in the project. The project only stores the information necessary to access the parts. The information is interpreted by a *Project Access Method*, (*PAM*). A PAM is a mechanism that gives WorkFrame access to the parts. Files on a local OS/2 system or OS/2-based local area network are accessed by the basic PAM, called IWFBPAM, that is shipped with WorkFrame. This is the default PAM that is used by VisualAge C++ projects.

Project parts can be accessed by more than one PAM, so a project can also contain files or objects from a foreign file system or repository. Unless you need to use another PAM in addition to the default PAM, you do not need to understand how PAMs work to use WorkFrame effectively. Otherwise, you might want to refer to 📖Chapter 8, “Project Access Methods (PAMs)” on page 149 for more information on PAMs.

Tools Setup

A project's **Tools setup** defines the actions, environment variables, and types that are available to the project. Examples of actions are Compile, Link, and Make. Types provide a way of grouping and filtering project parts so that you can refer to them with a single name. Examples of types are C++ Source, C Source, and Executables. Environment variables are system environment variables, such as PATH, and any environment variables that are defined using the OS/2 SET command. 📖Chapter 3, “The Project Tools Setup” on page 45 discusses the Tools setup of projects in detail.

Project Views

A project has five views:

- **Icon view**
- **Details view**
- **Tree view**
- **Tools setup**
- **Settings**

Icon view, **Details view**, **Tree view**, and project **Settings** are discussed in this chapter. 📖 Project **Tools setup** is discussed in Chapter 3, “The Project Tools Setup” on page 45.

You can open any one of the views from the project's system or pop-up menu. Select a view from the **Open** → cascading menu.

Note: By default, a project opens to its **Icon view**, but you can set the project's default view from the **Menu** page of the project's Settings notebook, as with any OS/2 folder.

Icon View

Figure 8 shows a fully expanded Icon view of a project.

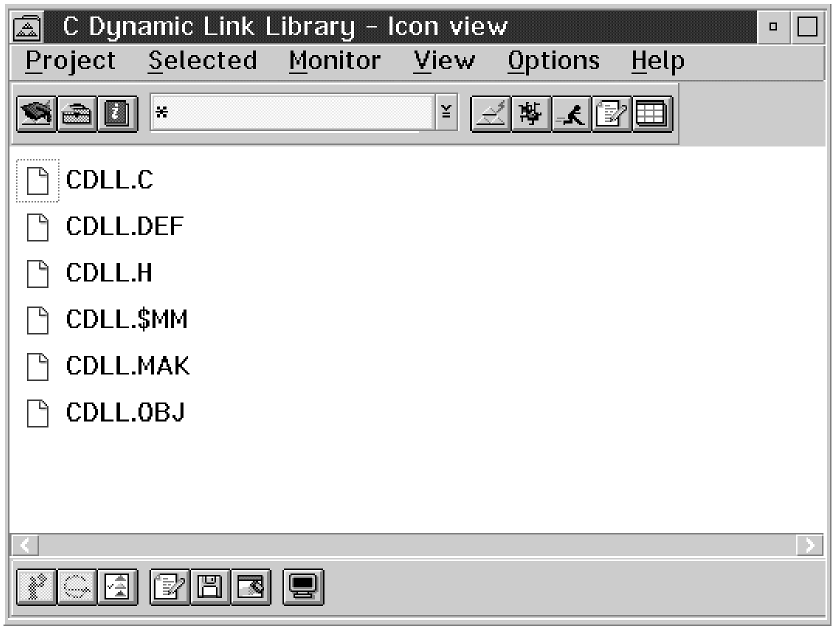

C Dynamic Link Library - Icon view
Project Selected Monitor View Options Help
*
CDLL.C
CDLL.DEF
CDLL.H
CDLL.$MM
CDLL.MAK
CDLL.OBJ

Figure 8. Project - Icon View

A project container has the following controls:

System menu

The contents of the system menu are the same as those of the pop-up menu from the project icon. There are menu items for project-scoped actions, for copying, deleting, shadowing, and moving the project, creating other projects, opening the project's Settings notebook, and opening different project views. You can also access this menu by pointing to the background of the project container and then pressing mouse button 2.

Project toolbar

The project toolbar has three fixed buttons on the left-hand side:

Build Smarts
Displays the **Build Smarts** window where you can set build features for your VisualAge C++ projects. See “Build Smarts” on page 67 for more information on how to use Build Smarts to aid you in your project builds.

Tools setup
Displays the **Tools setup** window that contains the actions, environment variables, and types associated with the project. See Chapter 3, “The Project Tools Setup” on page 45 for more information on the **Tools setup** window.

How Do I?
Displays the online **How Do I?** help for WorkFrame. It contains instructions for how to accomplish common WorkFrame tasks.

The right-hand side of the project toolbar also contains buttons for launching frequently-used, project-scoped actions like Build, Debug, and Run. Each button represents a single project-scoped action. You can customize the actions that appear on the toolbar by setting a control in an action's **Settings** notebook. 📖 See “Action Settings - Menus Page” on page 63 for more information on how to add an action to the project toolbar.

Performance Note: All WorkFrame toolbars have hover help. Hover help is a balloon that appears whenever you position the mouse pointer over a toolbar button. It contains help text about a button. You can turn off hover help when you become familiar with the toolbar buttons by deselecting the **Toolbars** → **Hover help** menu item from the **View** menu. You will see some performance gain when you do this.

Menu bar

The project menu bar contains menu items to launch project- and file-scoped actions, manipulate actions, variables and types, set action options, bring up the project's **Settings** notebook, and get help for WorkFrame projects.

The **Project** menu lists all the project-scoped actions available to the project.

The **Selected** menu contains the actions that apply to any selected project parts. Note that project- and file-scoped actions are also accessible from pop-up menus.

📖 Refer to “Actions on Menus” on page 73 for a list of rules that determine the way actions are listed on project menus.

Menu Note: Some actions that are only relevant when performing a Build on the project are excluded from project pull-down and pop-up menus by default. You can make any action visible or invisible on the project menus by configuring its menu settings. 📖 See “Action Settings - Menus Page” on page 63 to learn how.

The **Monitor** menu has controls for manipulating the project **Monitor**, where the output of monitored actions are displayed.

The **View** menu contains controls for opening different views on the project, including the project's **Settings** notebook and **Tools setup**. It also contains view options for the project toolbar and information area.

The **Options** menu is a list of actions whose options you change most frequently. When you select an action from this menu, the options dialog for the action appears. (Note that you can access the options for any action from the project's Tools setup. 📖 See “Action Options” on page 65 for more information on how to set options for all actions.) The **Options** menu also contains a menu item for the **Build Smarts** window that lets you quickly change options for actions used in a project build. 📖 See “Action Options” on page 65 and “Build Smarts” on page 67 for more information about setting action options.

The **Help** menu contains items for referencing WorkFrame online help, including WorkFrame **How Do I?** help and other online VisualAge C++ manuals.

You can control which actions appear in the **Project**, **Selected**, and **Options** menu bars by setting a control in each action's **Settings** notebook. 📖 See “Action Settings - Menus Page” on page 63 for more information on how to add an action to one of these menus.

Parts filter

The parts filter is an entry field on the project toolbar where you can enter a file mask or type name to filter the view of project parts. Only the parts that match the specified mask or type are displayed in the parts container. You can also use the drop-down list box to select from the types that are available to the project.

The current setting of the parts filter is only saved if you press the **Enter** key to activate it. It is saved for all the project views.

If you have a set of masks or types that you would like to use to filter the parts container with, define them as a single type so you can access them from the **Parts filter** drop-down list box. Refer to “Adding Types” on page 81 for instructions on how to define a new type for a project.

Parts container

This is the place where the project parts are displayed. Project parts are almost always files, although they can be Workplace Shell objects, and other kinds of objects as well. The parts of a project are specified in the **Location** page of the project's **Settings** notebook. See “Project Settings - Location Page” on page 25 for more information on how to specify a project's parts.

Project parts are displayed as icons in the project's **Icon view** container. You can control the layout and size of the icons from the **View** page of the project's **Settings** notebook.

You can customize the color and font of the project parts container by dragging a color and font from the OS/2 **Color Palette** and **Font Palette** as with any folder on your Desktop.

You can sort project parts on any column shown in the project's **Details view**. You specify the sort order from the **Sort** page of the project's **Settings** notebook. (This also works for **Icon view** and **Tree View**.) The project **Sort** page works the same way as the **Sort** page in any OS/2 folder.

Monitor

Output from actions like Compile and Link can be displayed in the **Monitor** window. By default, it is hidden when you open a project but appears when a monitored action is started. 📖 See Chapter 4, “The Project Monitor” on page 85 for information on how to use the **Monitor**.

You can change the color and font used in the **Monitor** list box by dragging a color and font from the OS/2 **Color Palette** and **Font Palette**.

Monitor toolbar

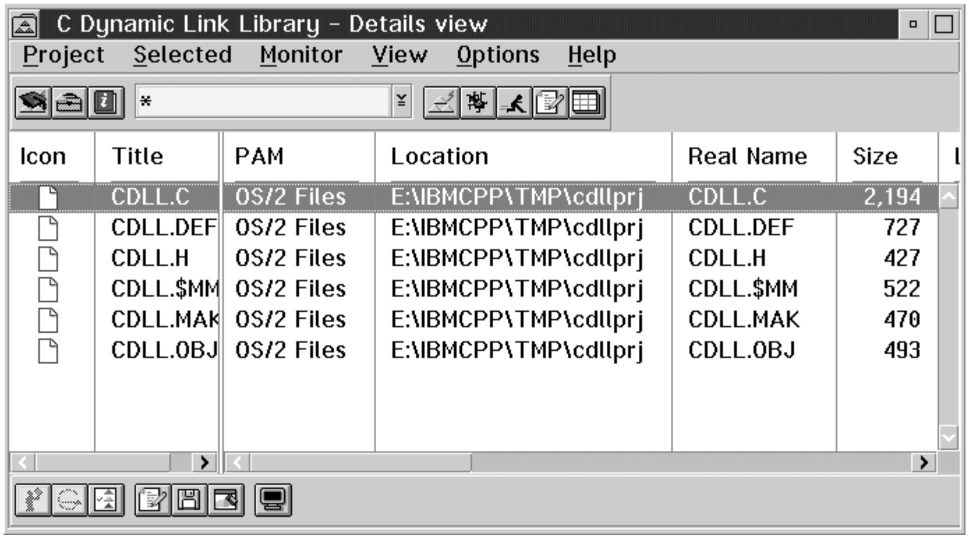
The Monitor toolbar contains buttons for controlling the project **Monitor**. An information line on the toolbar shows the name of the currently running action.

Split bar

The split bar divides the parts container from the project **Monitor**. The toggle setting and the position of the split bar when the project is closed is saved for each view of the project. The split bar can be saved at a different position for a tree view than for an icon view, for instance.

Details View

All the controls for a project's Details view are identical to those in its Icon view, except for the project parts container. Instead of just icons, project parts are represented by rows in a table. The columns in each row provide information about the part, such as its name, size, and creation date. Figure 9 shows an example of a project open to its Details view.



Icon	Title	PAM	Location	Real Name	Size
	CDLL.C	OS/2 Files	E:\IBMCPP\TMP\cdllprj	CDLL.C	2,194
	CDLL.DEF	OS/2 Files	E:\IBMCPP\TMP\cdllprj	CDLL.DEF	727
	CDLL.H	OS/2 Files	E:\IBMCPP\TMP\cdllprj	CDLL.H	427
	CDLL.$MM	OS/2 Files	E:\IBMCPP\TMP\cdllprj	CDLL.$MM	522
	CDLL.MAK	OS/2 Files	E:\IBMCPP\TMP\cdllprj	CDLL.MAK	470
	CDLL.OBJ	OS/2 Files	E:\IBMCPP\TMP\cdllprj	CDLL.OBJ	493

Figure 9. VisualAge C++ Project - Details View

The columns in the table are determined by the PAM that the project uses. If a project uses multiple PAMs, the columns displayed are the union of the columns from each PAM. Parts accessed by one PAM do not have entries in the columns that belong to another PAM.

You can set which columns to display in the parts container by selecting them from the project's **Settings** notebook, page 3 of 3 of the **View** tab.

Because VisualAge C++ projects use the basic OS/2 PAM, IWFBPAM, the columns in a VisualAge C++ project's Details view are, by default, identical to the Details view columns of a folder on the Desktop, with two additional columns: PAM name and source location.

Tree View

In a project **Tree view**, the project container is organized into a hierarchy of parts from each PAM. In a VisualAge C++ project, the only root, **OS/2 Files**, is that of the basic PAM, IWFBPAM. The basic PAM tree consists of the project's source directories. Each can be expanded to display the files they contain.

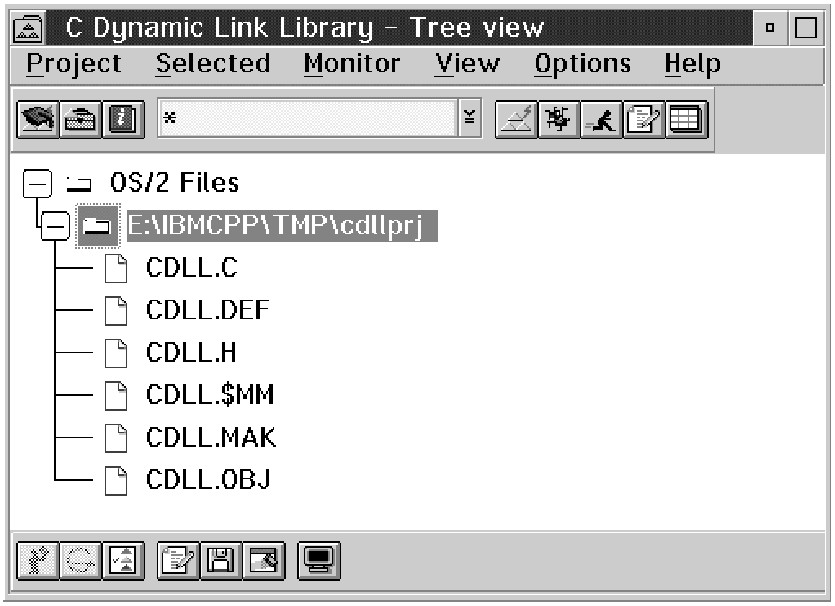


Figure 10. VisualAge C++ Project - Tree View

Note: Because a PAM can represent project parts that are not located in directories, a **Tree view** branch can potentially also represent any source location, such as a component in a version control tool, or a database table, if the project uses a PAM that supports it.

Creating a Project

There are many ways you can create a WorkFrame project:

- Drag the **VisualAge C++ Project** template in your VisualAge C++ folder. This will create a project with all the available VisualAge C++ actions and environment settings. To create a project with no actions or settings, drag the **WorkFrame V3 Project** from the **Templates** folder. You can set the project to inherit from the **VisualAge C++ Project** later, so that you have access to all the VisualAge C++ actions. 📖 See “Project Settings - Inheritance Page” on page 29 to learn how to set project inheritance.

- Select a template name from the **Create another →** cascaded menu on another project's pop-up menu. This has the same effect as dragging from a project template.

- Copy another project using the normal Workplace Shell copying techniques, like dragging or selecting **Copy** from the project's pop-up menu.

- Install a project from **Project Smarts**. The **Project Smarts** catalog contains a set of application templates that contain skeletal code for various kinds of applications, such as User Interface Class Library and Workplace Shell programming. For example, if you install the UI Class Library Application from **Project Smarts**, a project is created on your Desktop that has a completely configured project environment and template code for an UI Class Library program. 📖 See Chapter 6, “Project Smarts” on page 111 for more information on how to create projects from **Project Smarts**.

💡 If you have no existing source files to create a project with, and you have a specific kind of target in mind, then Project Smarts is the best way for you to create a project. If you already have source files to work with, you can either copy a project that is similar to your application, or create a new project from the **VisualAge C++ Project** template.

A project is created with a number of default settings, such the default PAM, view, and sort attributes. You may want to change some of these default project settings. The next section will take you through all the settings in a project **Settings** notebook.

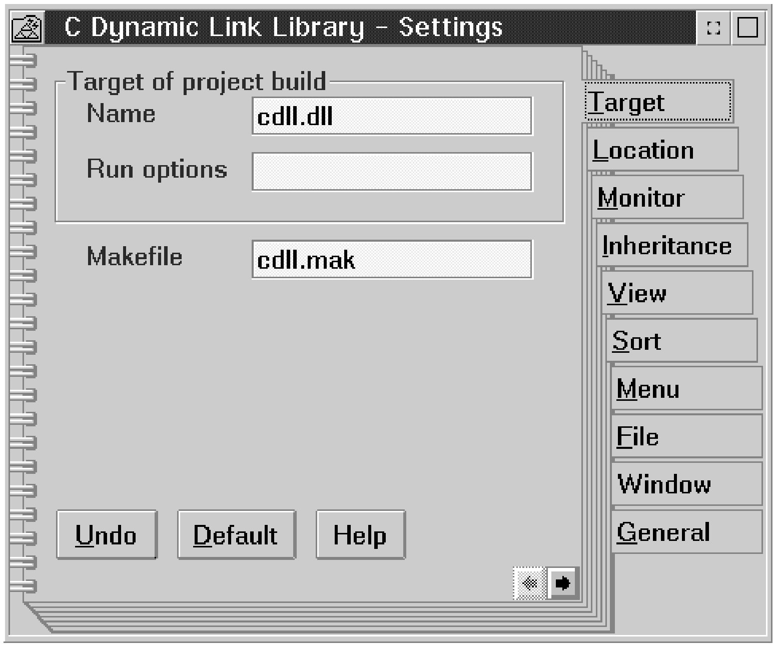
Project Settings

The **Settings** notebook contains settings for the project. Project settings are in the project's **Settings** notebook. To open the **Settings** notebook, you can:

- Select **Open** → **Settings** (or **Settings** if you are using OS/2 Warp) from the project icon's pop-up menu.
- Select **Open** → **Settings** (or **Settings** if you are using OS/2 Warp) from the project's system menu.
- Select a **Settings** notebook page from the project's **View** menu **Settings** cascade.

A project **Settings** notebook contains standard Workplace Shell pages, along with pages specific to a WorkFrame project. All the pages follow the Workplace Shell standard; that is, any data changed is immediately effective. Every page has an **Undo** button to return the page to the state it was in when the notebook was opened. The **Default** push button returns the page to the default values, and the **Help** push button provides help for the fields on the page.

Project Settings - Target Page


C Dynamic Link Library - Settings

Target of project build

Name: cdll.dll

Run options:

Makefile: cdll.mak

Figure 11. Project Settings - Target page

You can set the following information having to do with a project's designated target in the **Target** page of the project's Settings notebook:

Target of project build

Specify the target's name and parameters.

Name

The name of the project's target, for example, `PROGRAM.EXE`.

Target Notes:

1. A project can build more than one target, but WorkFrame recognizes only one project part as the designated target of the project. For example, a project can build both a .LIB and a .DLL, but the .LIB is the designated project target because it is produced from the .DLL file. See "The Build Utility" on page 92 for more information on building a project's target.
2. The object you specify as the project's designated target does not have to exist. It is understood to be the target of a **Build** action on the project.

Run options

The parameters entered here are passed to the target when it is executed. If the target is not an executable file, the contents of this field are ignored.

Make file

The name of the project's designated make file. This is the make file that is used when a project-scoped Build action is invoked. (See "The Build Utility" on page 92 for more information about builds.) A project can have more than one make file, for example, one that builds the target with debug information, and another that builds it optimized, but WorkFrame recognizes only one as the designated make file of the project.

You can enable prompting for your target so that when you run it, a dialog box appears where you can override the parameter string specified in the **Run options** entry field. To enable prompting when you run your target program through WorkFrame, open the options dialog of each Run action. Select the **Prompt** check box. Refer to "Action Options" on page 65 for more information about setting action options.

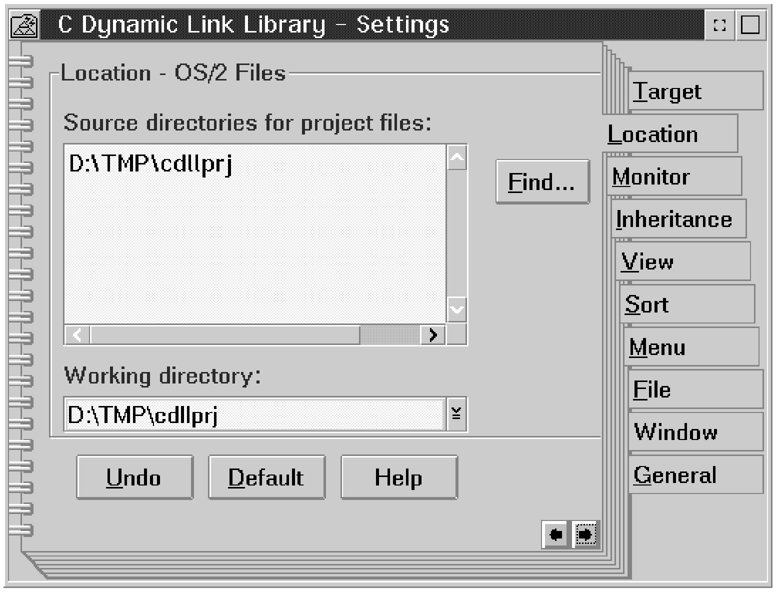
If your target program outputs to standard out, you can select the Run::Monitored action to run your target program in the **Monitor** window. You can also change the settings of the Run::Run Target action so that it always runs the target monitored. Chapter 4, “The Project Monitor” on page 85 tells you all about the WorkFrame project **Monitor**.

Project Settings - Location Page

You specify the source directories for your project in the **Location** page. All the files in the source directories are assumed to be the project's parts.

The information on this page is specific to the basic PAM, IWFBPAM, used by VisualAge C++ projects.

PAM Note: If your project uses multiple PAMs, the **Location** page is divided into a number of minor pages, one from each PAM. The basic PAM page described in this section appears with a minor tab called **OS/2 Files**, along with pages from the other registered PAMs. For an in-depth discussion of PAMs, see Chapter 8, “Project Access Methods (PAMs)” on page 149.



C Dynamic Link Library - Settings

Location - OS/2 Files

Source directories for project files:

D:\TMP\cdllprj

Working directory:

D:\TMP\cdllprj

Figure 12. Project Settings - Location page

On the **Location** page, you specify the following settings:

Source directories for project files

List the source directories where project files are to be stored, one path name per line. If the directories do not exist, you are prompted for permission to create them. If files already exist in one or more of the specified directories, they automatically become project parts. For example, you could enter the path names:

```
D:\MYAPP\DLL
D:\MYAPP\HEADERS
```

if the files are to be located in these directories.

You can also use the **Find...** push button to locate directories that you want to include, or drag a directory from the OS/2 **Drives** folder into the **Source directories** field.

Working directory

Select the working directory for your project from the directories you specified in the **Source directories** field. This directory is used:

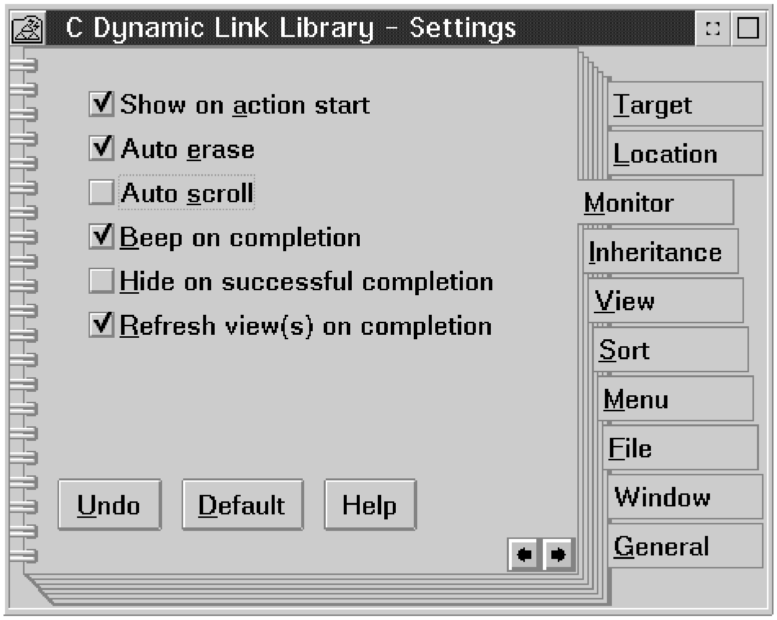
- To store any make files created by the MakeMake and Build utilities (📖 described in “The MakeMake Utility” on page 104 and “The Build Utility” on page 92).
- To store files copied or moved from other projects or folders.
- As the current directory from which actions are launched. Because many tools, such as the VisualAge C++ icc compiler, place their output in the current directory, tool output such as object files are often stored here.

  By default, the first directory in the list of **Source directories** is the project's working directory.

Project Settings - Monitor Page

You use the **Monitor** page to set options that have to do with the project **Monitor** (The project **Monitor** is described in Chapter 4, “The Project Monitor” on page 85).

The settings on this page are global, that is, they apply to *all* the projects on your system. These are the only global settings for WorkFrame projects.



C Dynamic Link Library - Settings

☑ Show on action start
☑ Auto erase
□ Auto scroll
☑ Beep on completion
□ Hide on successful completion
☑ Refresh view(s) on completion

Figure 13. Project Settings - Monitor page

You can specify the following monitor settings:

Monitor
Set these options to alter the Monitor behavior for the project.

Auto erase
Select this check box to clear the **Monitor** of its contents before a monitored action is invoked. If you do not select this option, the output of the invoked action is appended to the previous contents of the **Monitor** list box.

If you want to view the output of actions previously run in the **Monitor**, you can leave this option checked, and use the **History** window instead (select the **History** button on the **Monitor** toolbar). It shows you the output of any monitored action that ran in the current project session. This is faster than scrolling through the monitor. 📖 See “Action History” on page 87 for more information about the **Monitor History** window.

Auto scroll
Select this check box to turn on automatic scrolling for the **Monitor**.

Refresh views on completion
Select this option to automatically refresh the parts container each time an action has finished running in the monitor.

Show on action start
Select this check box to have the **Monitor** automatically show itself when a monitored action is started. If you do not select this option, the **Monitor** will stay hidden until you use the monitor **Show** toolbar button 🖥 to make it appear.

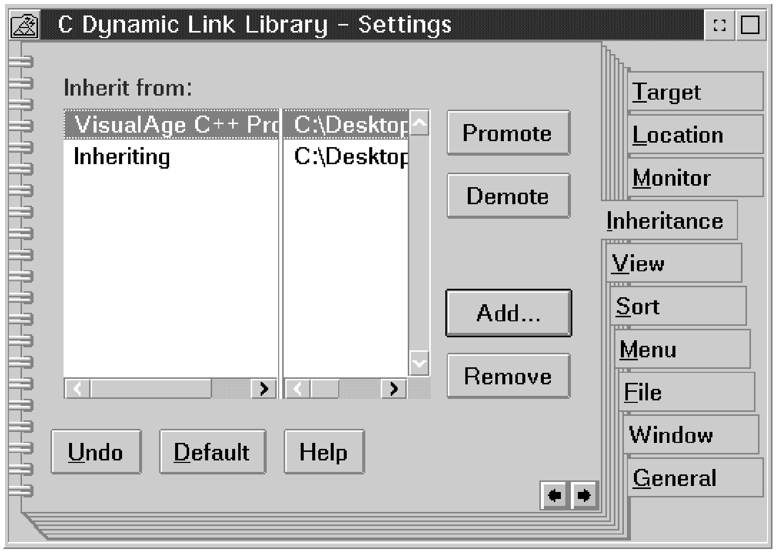
Hide on successful completion
Select this check box if you want the **Monitor** to hide itself when an action is successfully completed. If the action is not completed successfully, the **Monitor** remains open. If you do not select this check box, the **Monitor** stays open regardless of the result of the action until you hide it by clicking on the monitor toggle button.

Beep on completion
Select this check box to have the **Monitor** emit a high-pitched beep when an action has completed successfully, and a low-pitched beep if it finished with errors (that is, with a non-zero return code). If you do not select this option, the **Monitor** is silent.

Project Settings - Inheritance Page

On the **Inheritance** page, you can specify one or more projects whose **Tools setup** to inherit. See Chapter 3, “The Project Tools Setup” on page 45 for a complete discussion of the project **Tools setup**, and “Inheriting a Tools Setup” on page 33 for more detailed information about how project inheritance works.

The default for VisualAge C++ projects is to inherit from the **VisualAge C++ Project** which contains all the actions, environment variables, and types you need to develop applications using VisualAge C++ for OS/2.



Inherit from:

VisualAge C++ Pro	C:\Desktop
Inheriting	C:\Desktop

Figure 14. Project Settings - Inheritance page

The **Inheritance** page has the following controls:

Inherit from

The **Inherit from** list box lists, in reverse order of precedence, the titles of the projects whose actions, types, and environment variables to inherit.

Add

To add another project to the **Inherit from** list box, you can select the **Add** push button. The **Find Project** dialog appears where you can select a project to add to the inheritance list.

You select the physical file name of a project in the **Find Project** dialog. If the drive where the project is stored has a FAT (file allocation table) file system, the name may appear somewhat mangled because of the 8-character name limitation on those systems. Also, certain characters that may be part of the project's title are illegal in a FAT system, so OS/2 susbstitutes other characters. For example, the physical file name of the **VisualAge C++ Project** would be `VISUALAG` on a FAT file system. A project titled **C++ Program** would have the file name `C!!_PROG` on a FAT file system. You can determine the physical file name of any project by looking at the **File** page on the project's **Settings** notebook. Once you've selected a project file, its title is shown in the **Inherit from** list box.

Remove

To remove a project from the inheritance list, select its name, and then click on the **Remove** push button.

Promote

To move a project one position upwards in the **Inherit from** list, select its name, and then click on the **Promote** push button.

Demote

To move a project one position downwards in the list, select its name, and then click on the **Demote** push button.

Project Settings - View Pages

The **View** tab has three pages, one for each project view.

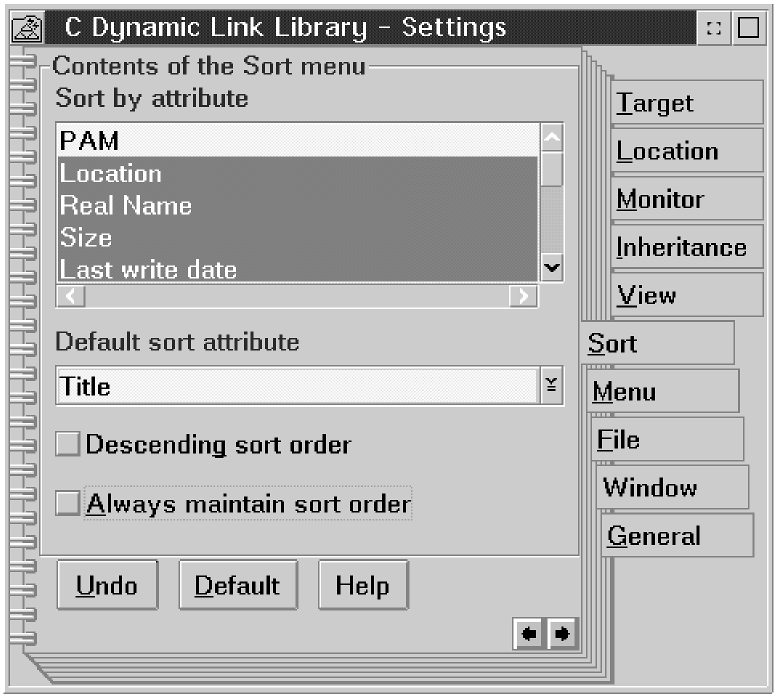
The first two pages, for Icon view and Tree view are very similar to those of a Folder on the Desktop. For icon view, you can select whether the icon format in the parts container should be **Flowed**, **Non-flowed**, or **Non-grid**. You can also select the icon display, **Normal size**, **Small size**, or **Invisible**.

For Tree view, you can select the **Lines** or **No lines** format. The icon display settings are the same as that for Icon view.

The page for Details view lists the columns that can be displayed. Deselect the columns you do not want displayed when your project is opened to Details view.

You set the font for the names of the project parts for any view in the parts container by selecting the **Change font** push button, which displays the standard font dialog. The list of selectable fonts are the same as those in your OS/2 **Font palette**.

Project Settings - Sort Page



C Dynamic Link Library - Settings

Contents of the Sort menu

Sort by attribute

PAM
Location
Real Name
Size
Last write date

Default sort attribute

Title

□ Descending sort order

□ Always maintain sort order

Figure 15. Project Settings - Sort page

To sort the parts in a project, select **Sort** from the project's pop-up or system menu. The **Sort** cascading menu contains a list of attributes by which to sort the parts. Select an attribute to sort the project container by.

Use the **Sort** page to add or remove attributes from the **Sort** cascading menu. The project settings **Sort** page is almost identical to that of a folder on the Desktop. It has the following controls:

Sort by attribute
This list contains the attributes for the project parts returned by each PAM. Select the attributes that you want to add to the **Sort** cascading menu on the project's pop-up menu. You specify the attributes by which the project parts are to be sorted by selecting them on the **Sort** menu.

Default sort attribute
Select the attribute that you want as the default sort attribute on the **Sort** cascading menu on the project's pop-up menu.

Descending sort order
Select this check box to sort the project parts in descending order. The default is to sort the parts in ascending order.

Always maintain sort order
Select this check box if you want the objects in the parts container sorted each time you open the project. If you add an object to the project, it will also be sorted as it is added. The project might take longer to open and refresh if you select this option.

Inheriting a Tools Setup

Projects can share actions, environment variables, and types by inheriting the **Tools setup** of one or more projects. (📖 See Chapter 3, “The Project Tools Setup” on page 45 for more information about the **Tools setup**). A project inherits from a list of *base projects* specified in the **Inheritance** page of the project's **Settings** notebook.

Any changes made to a project's **Tools setup**, such as adding, deleting and changing actions and their options, are reflected in the projects that inherit from it. Thus, you could organize your projects into a group of Presentation Manager projects that all inherit from a base Presentation Manager project, a group of SOM projects, and so on, so that any changes made to the base project are propagated down to related projects.

A project that inherits its **Tools setup** from other projects may also have locally defined actions, types, and environment variables that complement the inherited ones.

Another alternative to project inheritance is to copy one or more actions, types, and variables from one project to another using drag-and-drop, or by selecting **Copy...** from their pop-up menus. The copied objects would then be defined locally in the target project, rather than inherited.

You cannot add an action that has the same name and class as another action in the project, even if the other action is inherited.

Inheritance Precedence Rules

When projects inherit their actions, types, and variables from other projects, the following precedence rules apply:

Merging Actions

The set of actions available to a project is simply the union of all the actions from each base project in the inheritance list. If there are actions with the same name and class, the action from the project that appears later in the inheritance list is taken. If an action with the same name and class as another inherited action was defined in the project *before* the inheritance was established, the locally defined action is taken. (📖 See “Actions” on page 48 and “Action Classes” on page 49 for more information about actions.)

You cannot override an inherited action by adding another action with the same name and type after the inheritance has been established because adding an action with the same name and class as an existing action is not allowed.

For example, if project *Fred* inherits from projects *Wilma* and *Betty*, in that order, and the action Compile::C/C++ Compiler exists in both *Betty* and *Wilma*, then the Compile::C/C++ Compiler action from *Betty* is used in project *Fred*, because *Betty* follows *Wilma* in the inheritance list.

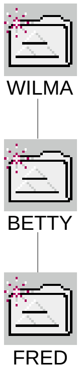


Figure 16. Project Fred inherits from Projects Wilma then Betty

Merging Types

The set of types available to a project is simply the union of all the types from each base project in the inheritance list. If there are types with the same name, the type from the project that appears later in the inheritance list is taken. A type already defined locally in the project before the inheritance was established overrides any other inherited type of the same name. You cannot override an inherited type by adding another type with the same name after the inheritance has already been established because adding another type with the same name as an existing one is not allowed.

Merging Environment Variables

Environment variables from multiple projects are merged as follows:

- The inherited projects are processed in the order that they are listed in the inheritance list.

  For example, if project *Fred* inherits from project *Wilma* and *Betty*, in that order (as illustrated in Figure 16), the variables from project *Wilma* are processed before those of project *Betty*. That means that if projects *Betty* and *Wilma* both define a `HELP` environment variable, the definition from *Betty* is used.

- Within each project, the variables are processed in the order that they are listed.
- As the variables are processed, any new variables encountered are added. If a processed variable is encountered again, the newer definition is kept.
- The variable values are interpreted before they are assigned to the variable. So, any environment variables in the value string are replaced with the current value of the environment variable. For instance, an environment variable in the form `%VARIABLE%` in the value string is replaced with the current value of the variable `VARIABLE`.
- You can combine and redefine environment variables dynamically in any form. New values can be prefixed or appended to existing variables or formed from some combination of text and existing variables, such as:

  ```
  PATH = %PATH%;D:\MYTOOLS;%DPATH%
  EPMPATH = %PATH%;%EPMPATH%
  ```

  For example, recall that project *Fred* inherits from projects *Wilma* and *Betty*, in that order. If project *Betty* defined the help variable as:

  ```
  HELP = D:\BETTY;%HELP%
  ```

  the `HELP` environment variable in project *Fred* would be defined as `D:\BETTY` with the value of `HELP` from project *Wilma* appended. This is true unless project *Fred* defines its own `HELP` environment variable, in which case, any environment variables in the value string would be replaced with the latest processed values from *Betty*.
- A variable can appear more than once in a **Tools setup**. The last one listed has the final value.

PAM Note: Although environment variables are interpreted only by a PAM, WorkFrame handles the merging of inherited environment variables without intervention from any PAM.

Projects Are Files

Projects are simply files to the operating system. Each project has a corresponding physical file in the OS/2 files system. These files are typically found in in the \DESKTOP subdirectory on your boot drive. Projects do not have to be stored on the Desktop; they can also be stored on other drives, including LAN drives. In that case, you would gain access to the projects using the OS/2 **Drives** folders.

You can determine the physical file name of a project by turning to the **File** page in the project's **Settings** notebook.

Because projects are simply files with extended attributes, you can back them up using normal OS/2 backup programs that also save and restore extended attributes. You can also compress them using OS/2 compression programs like ZIP and UNZIP that correctly save extended attributes on OS/2 files. Once compressed, projects can then be stored on diskette.

Note: When you copy a project's physical file, none of the project's parts, including any nested projects, are copied with it. To fully back up a project on a diskette, you need to save the compressed project file, and the project's parts.

Nested projects are not stored on the Desktop. They are stored in any of the source directories of their parent project. 🕮 See “Project Settings - Location Page” on page 25 for more information on a project's source directories.

The Default Project

WorkFrame recognizes a single default project in the system. In VisualAge C++, the WorkFrame default project is called **VisualAge C++ Project**. It contains all the actions available in the VisualAge C++. Many VisualAge C++ sample projects inherit from it.

The **VisualAge C++ Project** is also used as the default associated project when VisualAge C++ tools are started from the command line.

For example, the Edit::Editor action is defined as the default Edit action in the **VisualAge C++ Project**. This applies to any VisualAge C++ tools started from the command line that need to call the default Edit action, such as the VisualAge C++ Browser. (🕮 See “Default Actions” on page 71 for information on how to set the default action for a class of actions.)

If you start the VisualAge C++ Browser from an OS/2 command prompt by typing icsbrs, and request to view a source file in the Browser session, the Browser will invoke the default editor as defined in the default project. If the Browser was started from a specific project, the browser invokes the default editor as defined in the project from which it was invoked. You can change the default actions in the default project to customize the integrating behavior of tools started from the command line, but remember that projects inheriting from it will also be changed. See "Default Actions" on page 71 for information on how to set the default Edit action.

The identity of the WorkFrame default project is set in an environment variable called IWF.DEFAULT_PRJ. During installation, this variable was set in your CONFIG.SYS to the **VisualAge C++ Project** object identifier, CPPDFTPRJ. (True WPS object identifiers are surrounded by the <> characters, as in <CPPDFTPRJ> but these characters are not permitted in environment variable strings.) However, because WorkFrame projects are physically stored as files with extended attributes in the Desktop subdirectory of your boot drive, as well as in other drives and subdirectories, the IWF.DEFAULT_PRJ environment variable also accepts fully qualified path names. See "Projects Are Files" on page 36 for more information on how projects are physically stored on your system.

You can make any existing project the WorkFrame Default Project by specifying the project's fully qualified path, for example,

```
IWF.DEFAULT_PRJ = C:\DESKTOP\MY_DEFAULT_PROJECT
```

Changing the WorkFrame default project does not affect the projects that inherit from the original default project. It simply loses its special status and becomes like any other project.

Note: If you specify the default project with a fully qualified path name, do not move it to another location without updating the IWF.DEFAULT_PROJECT variable. Otherwise, the VisualAge C++ tools will not be able to work in an integrated fashion when they are started from the command line.

Organizing Projects

Organizing your projects well is key to using the WorkFrame environment effectively. The organization of your projects determines the way in which they are built, and the way in which make files are generated for your applications.

Most of your applications will most likely consist of a hierarchy of projects, rather than a single project, unless they are very simple. It is important that your project hierarchy reflects the project targets and dependencies between components for builds to be performed correctly. Follow these guidelines for a well-defined project hierarchy:

1. Create a separate project for each target or build path in your application. A *target* is defined as a single part or file that the project will build, such as an executable file, dynamic link library, or help file. A *build path* is a sequential processing of actions, with no conditional or alternative routes, that produces a single target file. Intermediate files may also be produced during the processing of a build.

   If your application consists of a two DLLs, a LIB, an EXE, and a HLP file, for example, you should create a separate project for each DLL, the EXE, the LIB, and the HELP file. If LIBs are also produced from the DLLs, the DLL and the LIB can be part of the same project. The LIB would be the designated target of the project, and the DLL is considered an intermediate file. In this example, you would create five projects in all.

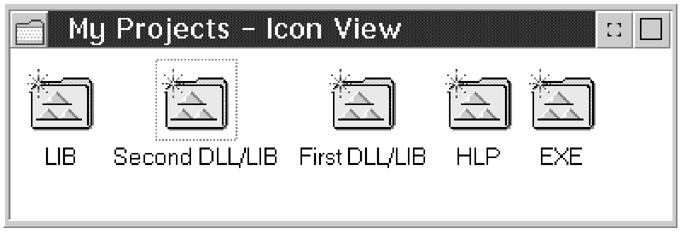


Figure 17. Five projects in a sample application

2. After you have created the projects, determine the dependencies that exist between them.

   For example, the DLLs may depend on the LIB, and the EXE depends on the DLLs. Also determine the project that is at the root of the dependency hierarchy, that is, the project that has no dependents. In this example, it is the EXE.

3. Nest the projects to reflect the dependency tree. That is, if a project *EXE* depends on another project *DLL*, place project *DLL* into project *EXE*. Project *DLL* is then said to be *nested* within project *EXE*. A project is nested in another project when its physical file is located in one of the source directories of the nesting project.

   You can move a project into another project by dragging it or by selecting **Move...** from its pop-up menu.

   Notes on Moving Projects:

   a. When you move a project into another project, you are actually moving the nested project's physical file into the nesting project's working directory. So if you move a project into another project using the **Move...** pop-up menu item, you will need to specify one of the parent project's source directories as the target in the **Move** dialog.

   b. When you move a project, you don't need to move its parts as well because the project does not actually contain its parts, but only a reference to their location.

   In a situation where two or more projects (say *First DLL/LIB* and *Second DLL/LIB*) depend on the same project *LIB*, nest one of the two depending projects within the other (say, nest *Second DLL/LIB* within *First DLL/LIB*), and then nest the mutual dependency project *LIB* within project *Second DLL/LIB*.

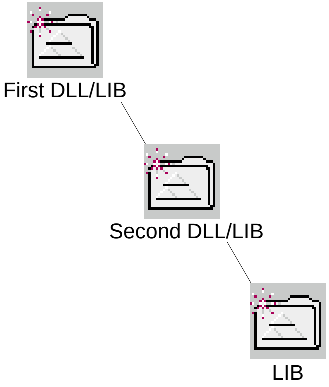


Figure 18. Projects First DLL/LIB and Second DLL/LIB depend on project LIB

The root project in the dependency tree will directly or indirectly contain all the other projects in the application.

Consider the example application that consists of an EXE, two DLLs, a LIB, and a HLP file. The two DLLs depend on the LIB, the EXE depends on the DLLs and the HLP file. The LIB and HLP files do not depend on any other targets. The project hierarchy would look like the one represented in Figure 19.

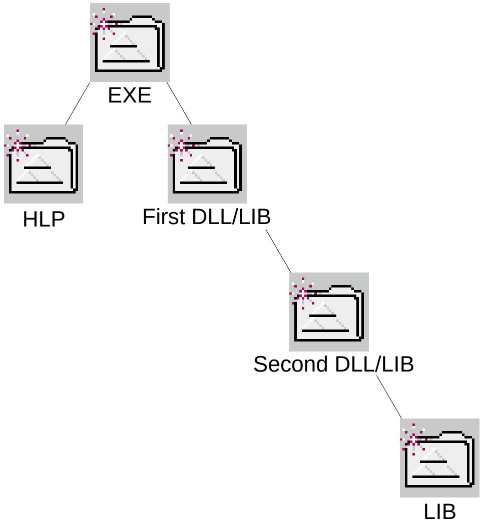


Figure 19. Project hierarchy of a sample application

The *EXE* project is the root project that contains all the other projects. *First DLL/LIB* and *Second DLL/LIB* both depend on the *LIB*, so one nests the other before nesting the *LIB* project. The two DLLs also produce a LIB file each, so the specified target of each DLL project is a LIB file, because it is the final file produced by a build.

You should nest projects this way because the WorkFrame Build utility infers the build sequence from the project hierarchy. When you use the Build utility to build a project, you have the option of having it build descendant projects first. Following this scheme ensures that a project's descendants contain the current project's dependencies. See “The Build Utility” on page 92 for more details.

4. All the files used by the projects in your application can be stored in one or more directories. For example, all the source files to build the DLLs and LIBs could be stored in one directory, all the source files for the EXE in another, and all the HLP files in a third or fourth directory.

   Nesting Note: If you store the files for each project in a different directory, the target of a nested project is not automatically considered a part of the nesting project. You must include the directory that contains the required target in the source directories list of the nesting project. As a better alternative, you could define an environment variable in the nesting project so that the required target is available. For example, the *EXE* project could define the `LIB` environment variable in its **Tools setup** to contain the working directory path of the *LIB* project whose target it depends on.

   For example, if the *First DLL/LIB* project's files are stored in a different directory from the *EXE* project, say in D:\MYAPP\LIB1, the *EXE* project must define the `LIB` environment variable to contain that directory so that the Link action is able to find the required library file:

   ```
   LIB=D:\MYAPP\DLL1;%LIB%
   ```

   Your project's directory structure does not have to mimic your project organization. A project can contain source files that are stored in multiple directories. For example, two projects that build two different targets from the same source files can share one or more source directories, but have separate working directories.

   If your project has any header files that are stored in directories other than the project's working directory, you should define the INCLUDE environment variable to reference the header files. Add the INCLUDE variable to the project's **Tools setup** so that it is set in any WorkFrame-generated make files.

Project Geometry

To put things into perspective for you, this section discusses three different, but simultaneous structures that a project can maintain: two conceptual, and one physical.

Inheritance

An inheritance graph of all the projects in your system would connect lines between the projects that inherit their **Tools setup** from each other. If you can think of a project's actions, types, and environment variables as its “behavior,” you can say that the projects in an inheritance graph borrow their behavior from one another.

Nesting

A nesting graph of all the projects in your system would be in the form of a tree, and would connect lines between projects and their subprojects. The organization of a project expresses the interdependency relationships that exist between itself and its descendants. This kind of project structure is very important for setting up project builds using the WorkFrame Build utility.

The Build utility allows its options to be passed to a project's descendants, or borrowed from a parent project. Although this resembles an inheritance relationship, it is distinct from it. Build settings can only be passed down to nested projects, or assumed from a containing project. The projects participating in an inheritance relationship with the current project are not involved.

Directories

The source directories of a project determine a project's parts. The source directories of a project can mimic its nesting structure, or serve a different organizational purpose, such as grouping files by type. For example, private and public header files could be stored in separate directories from the rest of the source.

Sharing Project Parts on a LAN

If you work with a team connected via a local area network, you can share WorkFrame projects by:

1. Sharing source code over the LAN. Because the basic OS/2 PAM supports directories physically located on a LAN, you can specify source directories using the Universal Naming Convention (UNC) or by naming logical drives that map to a remote location. Other PAMs may allow other notation depending on the environment they support.

2. You can also share the projects themselves if they are stored on a LAN drive. Projects are simply data files in the OS/2 file system, and can be accessed and stored in a remote OS/2 **Drives** folder. Projects to be shared over a LAN cannot reside directly on the Desktop because the Desktop is stored on a local drive. However, you can create a shadow of a project that exists on a LAN drive on your Desktop. This is an easy way of accessing shared projects.

   Note: The IBM Network File System (NFS) does not currently support extended attributes, so you cannot store projects on NFS drives.

3. Copying or moving projects over a LAN by dragging them into a remote OS/2 **Drives** folder. Then from the target system, drag the project from the remote OS/2 Drives folder onto the Desktop or other location.

   Notes on Moving Projects:

   - When you move a project into another project, you are actually moving the nested project's physical file into the nesting project's working directory. So if you move a project into another project using the **Move...** pop-up menu item, you will need to specify one of the parent project's source directories as the target in the **Move** dialog.

   - Copying or moving a project does not copy or move its parts or any other projects it might contain. “Projects Are Files” on page 36 tells you how you can fully back up a project and its parts.

4. Inheriting from a project that resides on the LAN. Because LAN drives are usually not attached when the OS/2 Desktop is loaded (when projects are initialized), you will need to refresh the inheriting project by selecting **Refresh now** from the project's popup menu. This tells WorkFrame to update the inheriting project with the information that is now available from the project on the LAN. The actions, environment variables, and types from the inherited project on the LAN are not available in the inheriting project until you refresh it once the LAN drive is attached,

Creating Project Templates

Once you have set up a project, you may want to create other projects like it. You can either copy your project, or create a project template of your own to place in the **Templates** folder on your Desktop, or any location you want. Then, whenever you select the **Create another** → menu item from a project's pop-up menu, your very own customized project template appears along with the default WorkFrame ones on the cascaded menu.

Here's how to create your own project template:

1. Make a copy of your project by selecting **Copy...** from the project's pop-up menu, or by pressing the **Ctrl** key while you drag your project's icon.
2. Open the new project's **Settings** notebook by selecting **Open** → **Settings...** (or **Settings** if you are using OS/2 Warp) from the project's pop-up or system menu).
3. Turn to the **General** page of the notebook. It is the last page.
4. Select the **Template** check box to turn the project into a template. The project's icon will change to a template icon. You can now move the template into the **Templates** folder by selecting **Move...** from its pop-up menu.

You can change any attributes of a project template (like its inheritance list, actions, options, colors and fonts) in the same way you can for any ordinary project.

As an alternative to creating a special project template, you can have all the projects you create inherit from your model project, so that any changes you make to the actions, options, environment variables, and types in your model project are also reflected in the new projects.

The Project Tools Setup

A project's **Tools setup** consists of the list of *actions* that can be used to manipulate the project and its parts, and the *types* and *environment variables* that support these actions.

You can see the list of actions, environment variables, and types that apply to a project by opening its **Tools setup** (select **Open** → **Tools setup** from the project's pop-up or system menu, or select **Tools setup** from the **View** menu).

The **Tools setup** window has three views, one each for actions, types, and variables. You can switch between the views by selecting **Actions**, **Variables**, or **Types** from the **View** menu in the **Tools setup** window, or using the buttons on the toolbar.

From the **Tools setup** window, you can:

- Set options for all the actions in your project. Action options are the parameters that are passed to the tool when its action is invoked.
- Add, delete, and change actions, types, and environment variables.
- Copy and move actions, types, and environment variables from one project to another.
- Find the project where an action, variable, or type is inherited from.

These functions are available from both pop-up menus and the menus on the menu bar. The **Actions**, **Types**, and **Variables** menus on the **Tools setup** menu bar have the same items as the pop-up menus on actions, types, and environment variable objects in the **Tools setup** window. The **View** menu contains menu items to configure the information area at the bottom of the **Tools setup** window, the toolbar, and the view that appears when you open the window.

An easy way to move actions, types and variables is to drag them to the target project. To copy, hold the **Ctrl** key down while you drag.

For more information on how to perform tasks related to actions, types, and environment variables, refer to the WorkFrame **How Do I?** help.

The Tools Setup Window

Figure 20 shows the **Actions** view of a **Tools setup** window. You can display the **Variables** view or the **Types** view by selecting from the **View** menu or by pushing the view buttons on the toolbar.

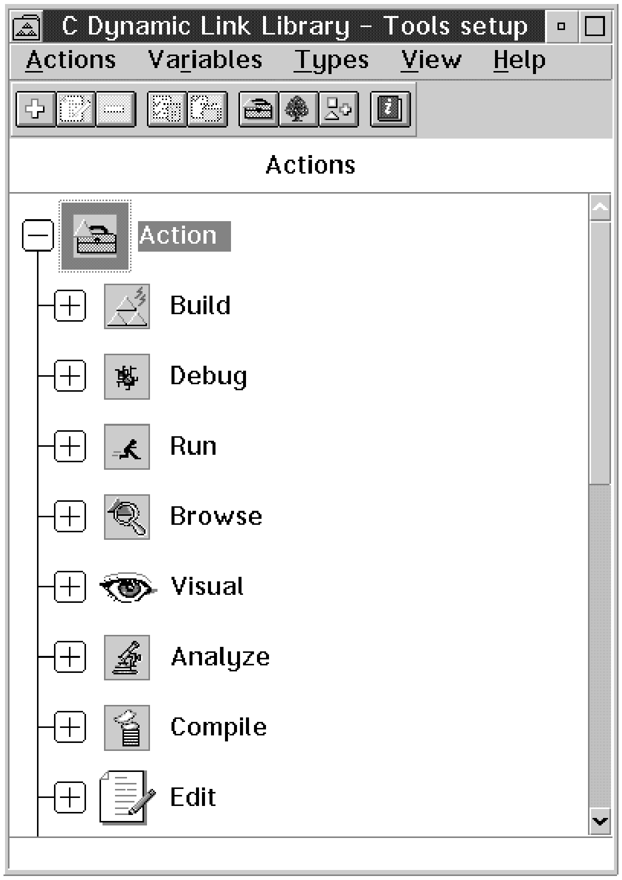


Figure 20. Tools Setup window - Actions View

The controls in the Actions view of the **Tools setup** window are identical in **Actions**, **Types**, and **Variables** views:

System menu
: Because the **Tools setup** is another view of a project, its system menu is identical to the system menu of any other project view.

Menu bar

The menu bar contains menu items to add, change, copy, move, and delete **Tools setup** objects. It also contains menu items to switch between views and control the way the window is displayed.

Tool bar

The tool bar contains the following buttons:

Add

Displays the **Add** window for the current view. For example, for Actions view, the **Add action** window is displayed.

Change

Opens the settings for the selected object. This button is not available when an inherited object is selected. You can only change **Tools setup** objects in the project where they are defined.

Delete

Deletes the selected **Tools setup** object. This button is not available when an inherited object is selected. You can only delete **Tools setup** objects from the project where they are defined.

File-scoped options

Displays the file-scoped options dialog of the selected action. Use this dialog to set options for the file-scoped action. Options are the parameters that are passed to the tool when its action is invoked. This button is disabled for **Variables view** and **Types view**.

Project-scoped options

Displays the project-scoped options dialog of the selected action. Use this dialog to set options for the project-scoped action. This button is disabled for the **Variables view** and **Types view**.

Actions view

Switches to **Actions** view.

Variables view

Switches to **Variables** view.

Types view

Switches to **Types** view.

How Do I?

Shows WorkFrame **How Do I?** help.

Project Tools Setup

Tools setup container
The **Tools setup** container displays the actions, variables, and types that are available to the project. Actions are displayed in expanded tree view, grouped by class and ordered by priority within each class. Variables and types are displayed in details view.

Actions

An *action* is a description of a tool or a function of a tool that can be used to manipulate a project's parts, or participate in a build. The tool can be any executable program or command file that the associated Project Access Method (PAM) recognizes. The basic OS/2 PAM, IWFBPAM, used by VisualAge C++ projects recognizes OS/2, DOS, and WIN-OS/2 executable programs, as well as .BAT and .CMD command files.

Actions are described in terms of their executable, name, class, input, and output. When tools are described in this manner, they can be integrated together without having prior knowledge of each other, and included in a WorkFrame-generated make file or build. They can also be substituted, added, or removed without affecting the rest of the project environment.

A tool can be described in terms of one or more actions in the project's **Tools setup**. For example, the VisualAge C++ Compiler, `icc.exe` can be described in terms of three different actions: Compile::C/C++ Compiler, Link::Linker, and Compile::Preprocessor. The actions vary only in the options used to execute the `icc.exe` executable. Similarly, the VisualAge C++ Editor can be described in terms of two different actions: Edit::Editor and Browse::Browser, again differing in the options with which they are invoked, and perhaps the types of files to which they apply.

A Compile::Preprocessor action doesn't actually come with the **VisualAge C++ Project**, but if you use the compiler's preprocessing function often, you can define one yourself that only invokes the preprocessor on the `icc.exe` executable. This is a good example of where an action does not map to a tool, in a one to one relationship. An action is more accurately a representation of a facet or behavior of a tool.

Action Classes

Actions are grouped into *classes* so that tools with similar function can be described and grouped together in project pop-up menus. For instance, the VisualAge Editor is invoked via the action named VisualAge Editor, that belongs to the Edit action class.

When tools are classified by their general function, integration between tools can occur on the class level. For example, the VisualAge C++ Browser calls the default Edit action to display source code for the objects being browsed. If the default Edit action is VisualAge Editor, the source code is displayed in an VisualAge Editor window. All this occurs without the Browser having to know the name of the editor, or how to invoke it. “Default Actions” on page 71 explains how you can set your default actions for Edit, Compile, and other action classes.

Action Settings

This section explains the pages of an action's **Settings** notebook so that you can understand how tools become actions in the WorkFrame environment.

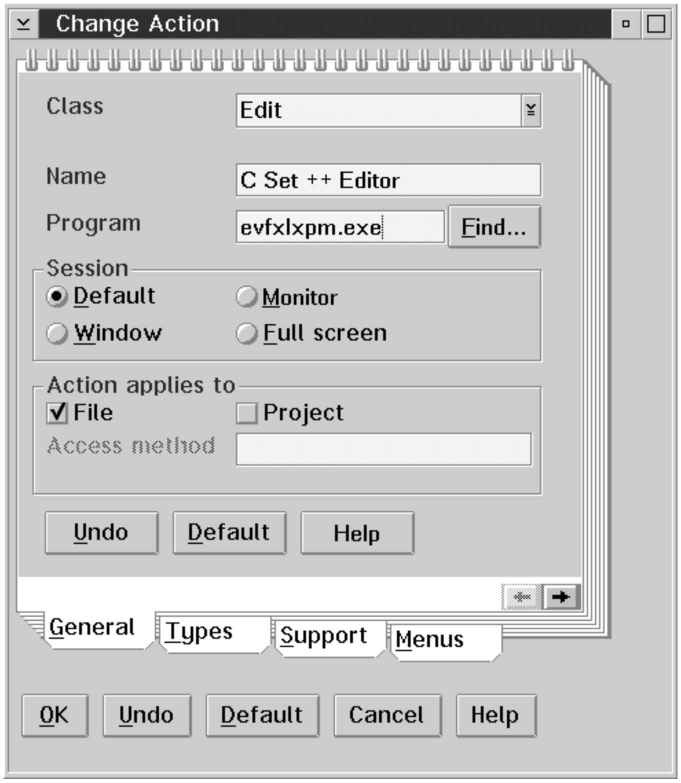
To bring up the **Settings** notebook for an action, select **Change...** from its pop-up menu.

To add an action to a project, select **Add...** from the pop-up menu of the Actions container. A Settings notebook appears where you can specify the necessary action settings.

If you highlight an action before selecting the **Add...** menu item, the fields in the Settings notebook are filled in with the settings of the highlighted action. This is a convenient way to copy the settings from a similar action.

Project Tools Setup

Action Settings - General Page



Change Action

Class: Edit

Name: C Set ++ Editor

Program: evfxlxpm.exe

Session
● Default ○ Monitor
○ Window ○ Full screen

Action applies to
☑ File □ Project
Access method

Figure 21. Action Settings - General page

You use the **General** page to specify the basic information about an action:

Class

The class of an action determines how the action is grouped in pop-up menus of WorkFrame projects and parts. It also determines the general nature of the action so that other tools can invoke it in a generic manner. WorkFrame predefines a number of action classes for you, such as Edit, Build, Preprocess, Compile, and Link. You can select one of the predefined names from the drop-down list box, or type in a new class name if none of the predefined classes is appropriate.

Note: The Run class is treated in a special manner by the basic OS/2 Project Access Method (PAM) supplied with WorkFrame. If you are adding or changing a Run class action, specify an asterisk (*) on the **Program** field to have the PAM execute the file specified in the Run action's options. For example, a project-scoped Run-action options string should contain the `%o` substitution variable to run the project's target. A file-scoped Run action should contain the `%f` substitution variable to run the selected file.

See “Substitution Variables” on page 69 for a list of WorkFrame substitution variables and their meaning. “Action Options” on page 65 discusses action options and how to set them.

Name

An action's name identifies it and sets it apart from the rest of the actions in the same class. The name is also used to list the action in project pop-up menus.

Program

This field contains the name of the program or command file to run when the action is invoked, for example, `icc.exe`, for a VisualAge C++ Compile action. The program name can be anything that the associated PAM can execute. The associated PAM is the PAM that controls access to the selected part or parts on which the action is being invoked. In the case of project-scoped actions, the associated PAM is the PAM explicitly named in the **General** page of the action's **Settings**.

The basic PAM used by VisualAge C++ projects can execute OS/2, DOS, and WIN-OS2 programs, as well as .CMD or .BAT command files. You do not need to configure any special WorkFrame settings to define actions for DOS and WIN-OS/2 programs or .BAT files. The basic OS/2 PAM automatically determines the kind of session the action should be run in so that it starts seamlessly when it is invoked.

Note: The basic OS/2 PAM has no capability to set DOS settings before invoking DOS programs.

If your project uses a PAM that can access project parts from a foreign file system, you can specify the name of an executable that runs on the file system. The PAM runs the action and communicates the results to WorkFrame. 📖 If you have projects that use more than one PAM, you may want to read Chapter 8, “Project Access Methods (PAMs)” on page 149 to understand more about how PAMs work.

To define an action that runs an OS/2 command like `copy` and `del`, enter `CMD.EXE` in the **Program** field. CMD.EXE is OS/2's command interpreter. The action should use the default Actions Support DLL and entrypoint, `IWFOPT` and `DEFAULT` (📖 see “Action Settings - Support Page” on page 58 for more information about actions support).

Then in the options dialog of your new action, type in the command string prefixed by a `/C`. The `/C` indicates to the command interpreter that a command follows.

For example, to define an action that copies the target program, which happens to be a DLL, to another directory on the LIBPATH, say `C:\OS2\DLL`, you could define these action options:

```
/C copy %o C:\OS2\DLL
```

`%o` is a WorkFrame substitution variable that represents the project's target. “Substitution Variables” on page 69 has a table of all the valid substitution variables you can use in command strings.

Session

Select one of these radio buttons to specify where the output from the action should be sent:

Default

Let the PAM decide what type of session the action should run in. If the action uses the basic PAM, OS/2 chooses the session type.

Monitor

Send the output to the project's **Monitor** window where you can click on error messages to bring up the editor on the source file. You can also save the output or print it. This option is only valid for actions, such as Compile and Link, that write their output to standard out.

Actions that run in the **Monitor** window are called *monitored actions*.

Note: Because the **Monitor** does not support user input, actions that prompt for input should be run in a **Window** or **Full screen** session instead.

Window

Output is sent to a text window.

Full screen

Output is sent to a full-screen window.

Scope

Actions may be *file-scoped* or *project-scoped*:

File-scoped

File-scoped actions apply to specific project parts, and can only be invoked from those parts. To run a file-scoped action, you must select one or more project parts and then invoke the action on the selected parts.

Note: Only file-scoped actions can be included in a WorkFrame-generated make file.

The types listed in the action's **Source types** list determine the project parts the action applies to. For example, the VisualAge C++ Compile action applies to parts that are of type `C/C++ Source`. This means you can invoke the VisualAge C++ Compile action by pointing to a .CPP file and selecting **COMPILE → VisualAge C++** from its pop-up menu. 📖 See “Types” on page 75 and “Action Settings - Types Page” on page 55 for more information on types and how they apply to actions.

Project-scoped

Project-scoped actions apply to a project as a whole, rather than to any selected set of project parts. These actions can be passed project information, such as the name of the project make file or target. Examples of project-scoped actions are Make, Build, Run, and Debug. The first two actions are invoked on the project's make file, and the last two on the project's target. Although make files and target files are also project parts, they have a special designation, that is, their names are explicitly recognized by the project.

Project-scoped actions are invoked from the project's pop-up menu. To bring up the project's pop-up menu, point to any background area in the project container and press mouse button 2.

Project-scoped actions cannot be included in a make file.

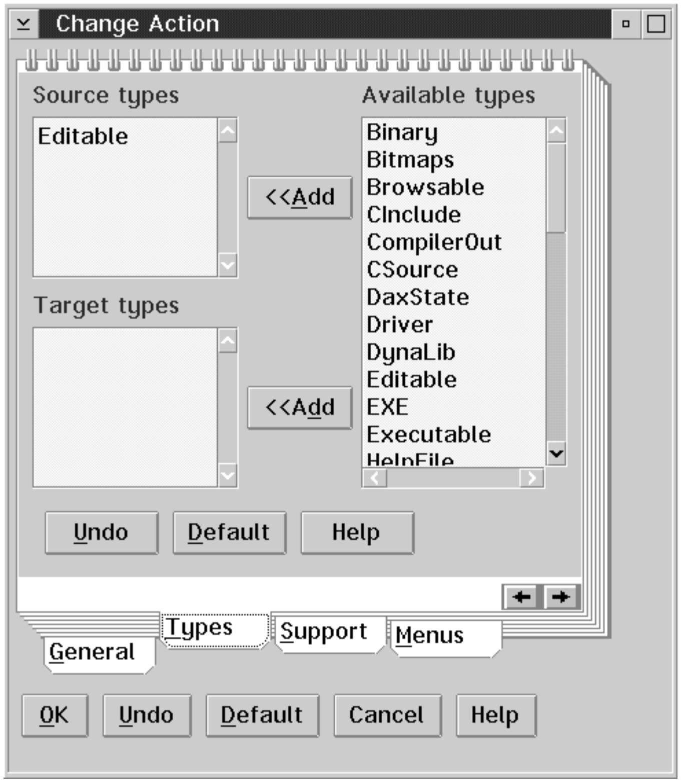
Access method

This field is only relevant for a project-scoped action. It identifies the PAM that executes the program that defines the action. The default is the basic PAM, IWFBPAM. If your projects only contain OS/2 files, you do not need to change this field.

If the action is file-scoped, the executing PAM is the PAM that can access the part or parts on which the action is invoked.

Action Settings - Types Page

The fields on this page are only relevant if the action is file-scoped.



Source types: Editable

Target types:

Figure 22. Action Settings - Types page

Use the **Types** page to specify the source and target types that apply to a file-scoped action. (📖 See “Types” on page 75 for a description of types.)

Source types

A list of file masks and named types that can apply to an action. This field is only required if the action is file-scoped. For example, the Edit::Editor action might have the following source types:

```
C/C++ Source
*.mak
*.rc
SOM Emissions
```

Target types

A list of file masks and named types that are produced by an invocation of the action. This field is only required if the action is file-scoped and could potentially be included in a WorkFrame-generated make file. Only file-scoped actions that have both source and target types can be included in a WorkFrame-generated make file.

For example, Edit::Editor would not have any target types listed (and so cannot be included in a WorkFrame-generated make file), but Compile::C/C++ Compiler might list the following:

```
Object Files
*.def
```

Build Notes:

- If the action can potentially be included in a project build (that is, it has both source and target types specified), you can only specify types of the classes "FileMask", "Logical OR", and "NOT in Logical OR" in the **Source types** and **Target types** list boxes. Types of other classes are ignored by the MakeMake utility. See “Type Classes” on page 77 for information about type classes, “The MakeMake Utility” on page 104 for information about the WorkFrame make file generation utility.
- All the types included with VisualAge C++ are of the allowed classes, "FileMask", "Logical OR", and "NOT in Logical OR". You can use the other type classes to specify the source types of actions that are not used in builds (such as Edit), and in the project **Parts filter**.

- WorkFrame uses the source and target types lists to infer the order that actions should be invoked to build the project's target when a Build action is invoked, or when you use the WorkFrame MakeMake tool to explicitly create a make file. Consequently, you should not list file types or masks that do not apply to the action in the **Source types** list. A Build action may produce unpredictable results otherwise.

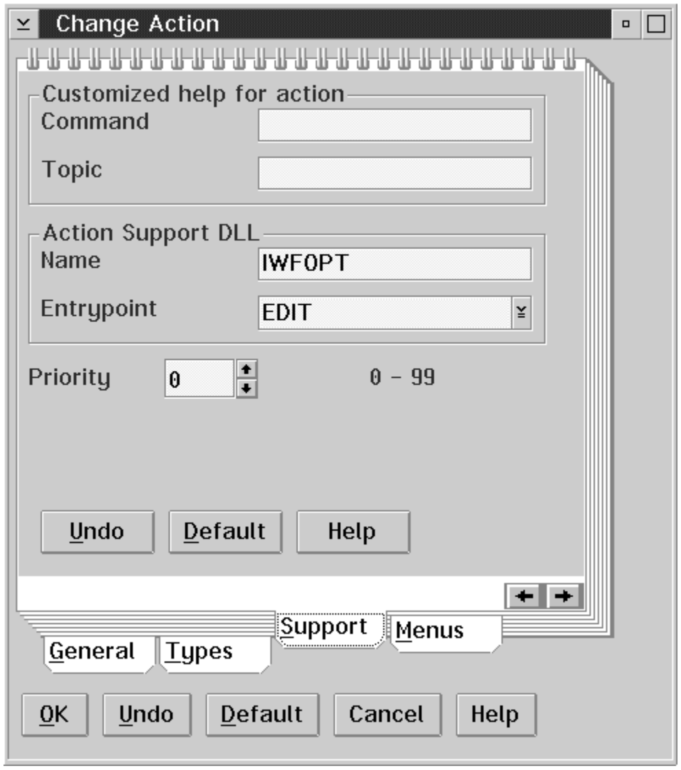
  📖 Chapter 5, “Building Your Target” on page 91 explains how you can set up the Build tool to generate and manage your project's make file.

Available types

This is the comprehensive list of types that are available to the project. To add a type to the **Source types** and **Target types** lists, select an available type and then click on the **<<Add** push button beside the **Source types** or **Target types** list box.

As you click on each available type in this list box, the information area at the bottom of the page shows the filter value of the type. Filters that are too long to fit in the information area are shown truncated.

Action Settings - Support Page



Name IWFOPT

Entrypoint EDIT

Priority 0 0 - 99

Figure 23. Action Settings - Support page

Use the **Support** page to specify the Actions Support DLL, any customized help for your action, and the priority for the action:

Customized help for action

You can specify context-sensitive help for an action using these entry fields. The help you specify here appears when you place focus on the menu item for the action and press the **F1** key. If you do not specify any customized help for an action, generic help for actions is displayed instead.

Command

This field contains the command that displays the help. For example, if the help is in an .INF file called HELP.INF, the command could be:

```
VIEW HELP.INF
```

Because, with the IPF View facility, you can specify the name of a panel in the .INF file to open the document to, the command could also be specified as:

```
VIEW HELP.INF Some Help Topic
```

Topic

You can specify the help topic separately from the command by entering the name of the topic in this field. WorkFrame appends the contents of this field to the text in the **Command** field, so you could also have specified the command in the previous example as:

```
VIEW HELP.INF
```

and specified the topic

```
Some Help Topic
```

in the **Topic** field.

If the topic needs to be specified in a position other than at the end of the command text, use the `%TOPIC%` substitution variable to place the text in the **Topic** field anywhere in the **Command** text.

For example, some viewing tool has the following command syntax:

```
SOMEVIEW TOPIC='Some Help Topic' FILE='HELP.FIL'
```

The **Command** text can then be specified as:

```
SOMEVIEW TOPIC=%TOPIC% FILE='HELP.FIL'
```

If the `%TOPIC%` substition variable occurs more than once in the **Command**, only the first instance is replaced.

Project Tools Setup

Action Support DLL

Each action has an associated Actions Support DLL that performs the following very important functions:

- Sets the default option settings for an action. (📖See "Action Options" on page 65 for more information about action options.)
- Displays a graphical user interface to gather the action options settings when you request to change the options for an action.
- Generates the command line that correctly invokes the action with the specified options.
- Processes the target and dependencies lists when the action is to be included in a make file.
- If the action is monitored, parses selected error messages to correctly invoke the default editor on the erroneous source file, and provides help for any parsed error messages.
- If the action is an editor, enables the Dynamic Data Exchange (DDE) communication with WorkFrame so that the editor displays source files and processes the error lines correctly.

The Actions Support DLL is specified by two fields:

Name
The name of the Actions Support DLL, for example, `IWFOPT`, the default Actions Support DLL.

Entrypoint
Because an Actions Support DLL can provide support for more than one action, you must also specify an entrypoint. The **Entrypoint** field is a drop-down list box that shows all the available entrypoints in the DLL.

Note: Actions Support DLLs are provided by tool developers who have integrated their tools into the WorkFrame environment. For tools that are not WorkFrame enabled, WorkFrame provides a default Actions Support DLL called IWFOPT.DLL, with entrypoints that apply to many action classes. The `DEFAULT` entrypoint is for actions whose class does not match any of the provided entrypoints.

For example, the Compile::Resource Compile action uses the COMPILE entrypoint in IWFOPT.DLL.

Some VisualAge C++ actions use the default Actions Support DLL, while others provide their own customized DLL. The following table shows the VisualAge C++ actions and their corresponding Actions Support DLLs and entrypoints:

Action	Actions Support DLL	Entrypoint
Analyze::Performance Analyzer	IWFOPT	DEFAULT
Bind::Parse and Bind Messages	IWFOPT	MSGBIND_PLUS
Bind::Resource Bind	IWFOPT	RESOURCE_BIND
BRSMON::Brsmon	IWFOPT	DEFAULT
Browse::Browser	IWFOPT	DEFAULT
Build::Build Normal	IWFBLDOP	BUILD
Build::Rebuild all	IWFBLDOP	BUILD
Compile::C/C++ Compiler	CPPICC30	COMPILE
Compile::IPF Compiler	IWFOPT	IPF_COMPILE
Compile::Message Compiler	IWFOPT	MESSAGE_COMPILE
Compile::Resource Compiler	IWFOPT	RESOURCE_COMPILE
Compile::SOM Compiler	IWFOPT	COMPILE
Database::Data Access Builder	IWFOPT	DEFAULT
Debug::Debugger	IWFOPT	DEBUG
Edit::Editor	IWFOPT	EDIT
Edit::Dialog Editor	IWFOPT	DEFAULT
Edit::EPM	IWFOPT	EDIT
Edit::Icon Editor	IWFOPT	DEFAULT
Edit::System Editor	IWFOPT	EDIT
Edit::Markexe	IWFOPT	DEFAULT
Inspect::View EXE Header	IWFOPT	DEFAULT
Lib::Import Lib	IWFOPT	IMPLIB
Lib::Import Lib (from Def)	IWFOPT	IMPLIB
Lib::Lib Tool	IWFOPT	IMPLIB
Link::Linker	CPPICL30	LINK
Make::Nmake	IWFOPT	MAKE
MakeMake::Makefile Generator	IWFOPT	DEFAULT
MapSym::Map Symbols	IWFOPT	DEFAULT
Package::Compress	IWFOPT	DEFAULT

Project Tools Setup

Action	Actions Support DLL	Entrypoint
Run::Foreground	IWFOPT	RUN
Run::Monitored	IWFOPT	RUN
Run::Run Target	IWFOPT	RUN
View::View Info	IWFOPT	DEFAULT
Visual::Visual Builder	IWFOPT	DEFAULT

For information on how to write your own Actions Support DLL, obtain the WorkFrame Version 3.0 Integration Kit[1].

Compatibility Note: Actions Support DLLs (or Options DLLs, as they were called in previous versions) that were written for Version 2.1 of WorkFrame will continue to be supported in this version. However, if an Actions Support DLL relies on the presence of the C/C++ Tools Version 2.01 runtime libraries, and you no longer have them on your system, it will no longer run. Contact the Actions Support DLL provider for an updated version. 📖 See Chapter 7, “Migrating Old Projects” on page 143 for information on how to migrate projects from previous versions of WorkFrame.

Priority

The priority of an action affects which action is launched when you double-click on a project part of a specific type. It also determines the default action of a class, and the ordering of actions on project menus. (📖 Refer to “Default Actions” on page 71 for more complete discussion about default actions, and “Types” on page 75 for information on types.)

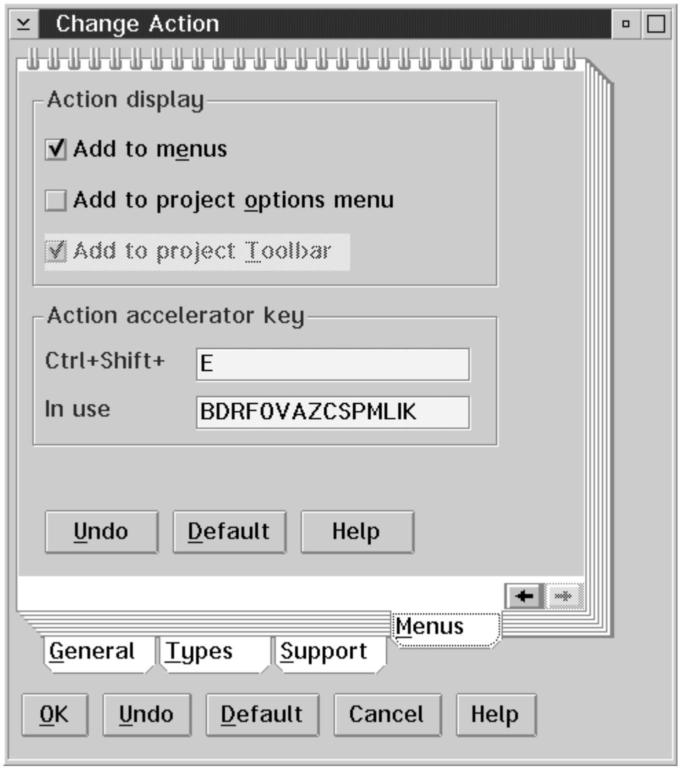
When you double-click on a project part, WorkFrame checks the **Priority** field of all the applicable actions, regardless of their class, and then invokes the action with the highest priority. If no actions apply to the part, the associated Workplace Shell behavior is invoked.

Set the **Priority** field to a number between 0 and 99. A higher number means a higher priority. Two actions can have the same priority value as long as they do not apply to the same type. If there are two actions with the highest priority for a selected type, the action that is actually invoked is undefined.

[1] To find out when and where this kit will be available, send a note to workframe@vnet.ibm.com, or call the VisualAge C++ automated help line 1-800-992-4777. Availability will also be announced on various networks where VisualAge C++ Service and Support is present.

The actions within each action class are listed in order of priority in the **Tools setup** actions view.

Action Settings - Menus Page



Action display

☑ Add to menus

□ Add to project options menu

☑ Add to project Toolbar

Action accelerator key

Ctrl+Shift+ E

In use BDRFOVAZCSPMLIK

Figure 24. Action Settings - Menus page

Use the **Menus** page to specify whether the action should appear in the project pull-down and pop-up menus, the **Options** menu, and the project toolbar. You can also use this page to define an accelerator key for your action.

Project Tools Setup

Action display

Use these check boxes to specify the menus that the action should appear in.

Add to menus

Select this check box to have the action appear in the **Selected** menu on the project menu bar if it is file-scoped, on the **Project** menu if it is project-scoped, or on both, if it is both file- and project-scoped. This setting also determines whether an action appears on pop-up menus.

Add to project Options menu

Select this check box to have the action appear in the **Options** menu on the project menu bar. It allows easy access to the options of frequently used actions. The **Options** menu lists all the project-scoped actions and all the file-scoped actions for which this setting is enabled. When you select an action name from this menu, WorkFrame displays its options dialog.

Note that options for all actions are always accessible from the **Actions** menu of the **Tools setup** window, or from the pop-up menu of an action in the **Tools setup** window.

Add to project Toolbar

If the action is project-scoped, select this check box to have the action appear on the project toolbar. The icon that is used for the action on the toolbar is that of the action class. All the WorkFrame predefined classes have unique icons bound to them.

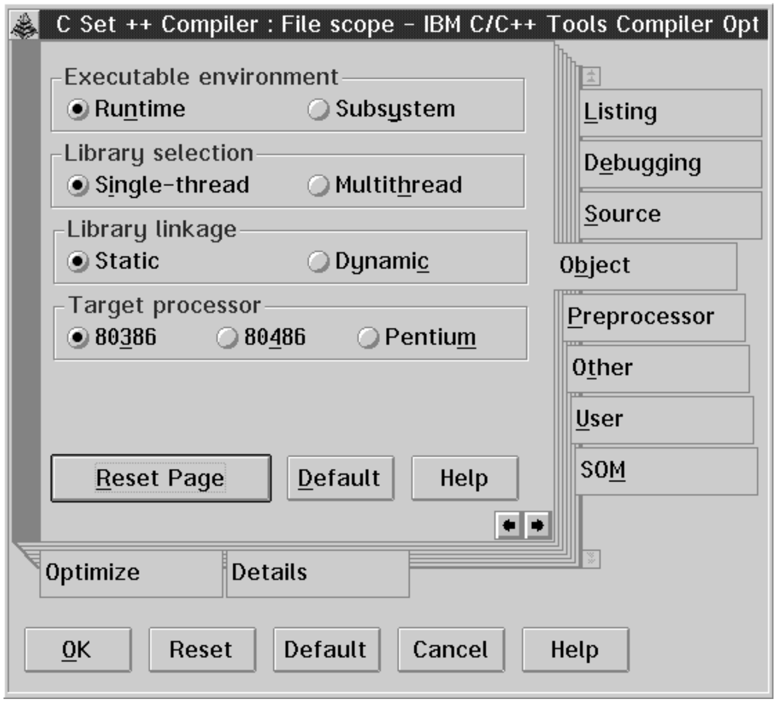
Action accelerator key

Action accelerators always start with the **Ctrl+Shift** keys. Type the letter of the key you want to associate with the action. The **In use** field shows you the letters that are already in use by other actions in the project. For example, a `C` in the **Ctrl+Shift+** field associates the accelerator key **Ctrl+Shift+C** with the Compile::C/C++ Compiler. This means you can use the **Ctrl+Shift+C** key stroke to invoke the Compile::C/C++ Compiler action on a selected project part.

Action Options

Each action has a set of options associated with it. The options represent the parameters passed to the tool to configure its behavior when its action is invoked. For example, to have the compiler generate Browser information, you need to invoke the compiler with the `/Fb`, or **Browser information**, option.

When you use WorkFrame projects to organize your tools and code, you can set these options once, and then when you invoke the action, by itself or as part of a Build, the options are always set for you. If the tool you are using provides customized WorkFrame support, you won't need to remember command-line options like `/Ge-` that tells the compiler that you're building a DLL, or `/St:<number>` to specify the stack size to the linker. Instead, you set options in a graphical user interface, like a dialog or notebook, with full access to online help. Figure 25 shows you an image of the Compile::C/C++ Compiler options dialog.



C Set ++ Compiler : File scope - IBM C/C++ Tools Compiler Opt

Executable environment: ● Runtime ○ Subsystem

Library selection: ● Single-thread ○ Multithread

Library linkage: ● Static ○ Dynamic

Target processor: ● 80386 ○ 80486 ○ Pentium

Figure 25. The VisualAge C++ Compiler options dialog

Project Tools Setup

To set options for an action in the **Tools setup** window, display the pop-up menu for an action and then select **File options** → **Change** (if the action is file-scoped) or **Project options** → **Change** (if the action is project-scoped). An options dialog where you can set options for the action appears.

You can also set an action's options by:

- Double-clicking on its icon. If the action is both file- and project-scoped, the file-scoped option dialog appears.
- Selecting its name and then selecting the appropriate (file-scoped or project-scoped) **Options** button on the **Tools setup** toolbar.
- Selecting its action name from the **Options** menu on the project menu bar. Only some actions are available on the **Options** menu. See "Action Settings - Menus Page" on page 63 for instructions on how to make an action available on this menu.

The dialog that appears is provided by the action's Support DLL. Actions that use WorkFrame's default Actions Support DLL (IWFOPT) do not have customized option dialogs. Instead, you enter options into an entry field in command line format.

Examples of VisualAge C++ actions that use the default Actions Support DLL are Database::Data Access Builder, and Make::NMake. Refer to the online help in the options dialog for instructions on how to use it.

Inheriting Options

If an action is inherited from another project, you can choose whether to:

1. Use the options stored with the defining project.

   This is the default. The action options from the inherited project are used in the inheriting project until you explicitly change the options in the inheriting project.

   As long as the options for an action in the inheriting project remain unchanged, any changes you make to the options in the inherited project are reflected in the inheriting project.

2. Set options locally.

   When you select **File options** → **Change** or **Project options** → **Change** from an inherited action's pop-up menu to change its options, the options are stored locally and are no longer affected by changes to the options in the inherited project. To revert to the inherited options, select **File options** → **Delete** or **Project options** → **Delete** from the action's pop-up menu to delete the locally stored options. You can also copy the options from the defining project by selecting **File options** → **Copy** or **Project options** → **Copy**. Copied options are stored locally.

Notes About Copying Options:

a. Copying options is a convenient way to set the options of an action in one project so that it is the same as the options in another. However, you can only copy options if the same action (that is, an action with the same name and class) exists in the destination project. If one doesn't exist, you should copy the entire action instead (select **Copy** from the action's pop-up menu, or drag the action while pressing the **Ctrl** key).

b. When you copy the options of an inherited action, the options of the action as it exists in the project where it is defined are copied. Note that this is not the same project from which you invoked the copy operation.

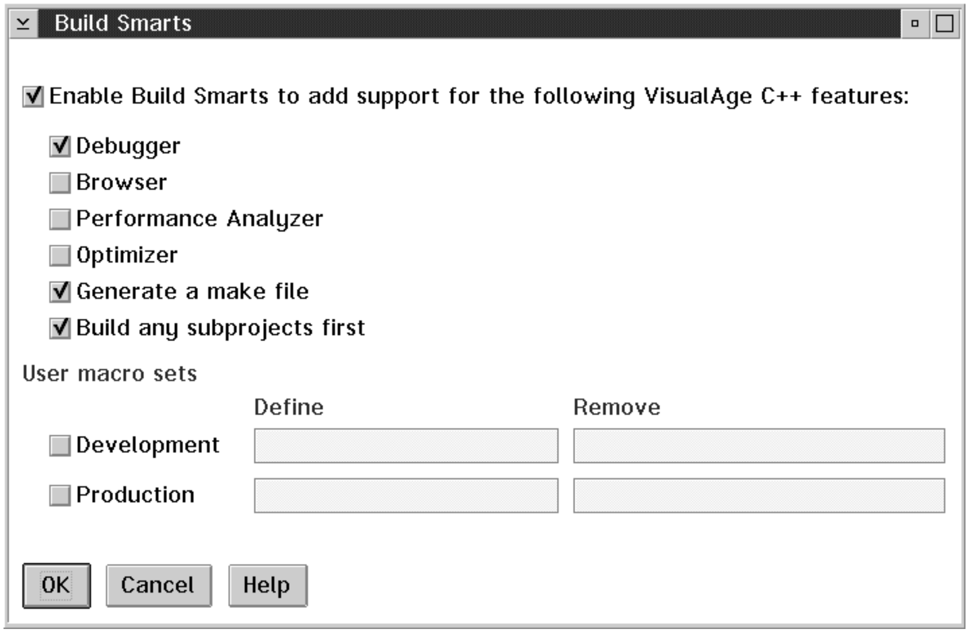
To determine the name of the project where an action is defined (or inherited from), select **Where defined** on the action's pop-up menu. **Where defined** is disabled when the action is locally defined.

Build Smarts

Build Smarts is a fast path for setting options for VisualAge C++ actions that affect the way your project target is built. The options that you set in the **Build Smarts** window work in combination with the options that are set for the individual actions themselves.

To display the **Build Smarts** window, select **Build Smarts** from the **Options** menu. The **Build Smarts** window is illustrated in Figure 26 on page 68.

Project Tools Setup



Build Smarts

☑ Enable Build Smarts to add support for the following VisualAge C++ features:

- ☑ Debugger
- □ Browser
- □ Performance Analyzer
- □ Optimizer
- ☑ Generate a make file
- ☑ Build any subprojects first

User macro sets

	Define	Remove
□ Development		
□ Production		

Figure 26. Build Smarts window

Each setting in the **Build Smarts** window can affect one or more VisualAge C++ actions. For instance, the **Debugger** check box affects the VisualAge C++ Compiler and Linker options. Selecting the **Debugger** check box effectively adds the `/Ti` option to the VisualAge C++ Compiler command line, and `/DE` to the VisualAge C++ Linker command line, when you initiate a build involving those actions.

The **Build Smarts** settings do not change the options already set for the individual actions in their options dialogs. The **Tools setup** settings are simply added to what is already set for the involved actions. When there are conflicts, the **Build Smarts** settings override those already set for the individual actions.

Two important build options are also included here: **Generate a make file** and **Build any subprojects first**. Select the first option to generate a new make file as part of running the build. Select the second option to build any child projects before building the current project so that all the dependencies are up to date.

At any time, you can disable the **Build Smarts** options by deselecting the **Enable Build Smarts** features check box if you only want to use the options set for the individual VisualAge C++ actions.

Substitution Variables

Substitution variables are place holders for items like file names and options in command strings. WorkFrame substitutes the variable with the appropriate value before passing the string on for processing. You can use them when you are specifying options for actions that use the default Actions Support DLL.

For example, you could write the command string

```
icc /Ti /C c:\dog\terrier.cpp
```

as

```
icc /Ti /C %f
```

where `%f` is the selected file

```
c:\dog\terrier.cpp.
```

The following table lists the substitution variables that you can use in WorkFrame/2 option strings:

Figure 27 (Page 1 of 2). WorkFrame substitution variables

%a %z	Is replaced by the names of all the selected project parts, each separated by a space. If the space between the **a** and the second **%** is replaced by a string, the names are separated by that string. For example, if the selected parts are
	`d:\cat.obj`
	`d:\dog.obj`
	`d:\bird.obj`
	the substitution variable `%a+%z` produces the string
	`d:\cat.obj+d:\dog.obj+d:\bird.obj`
	The only substitution variables allowed within the %a %z substitution variables are %% and %d.

%d	Is replaced with the project's working directory.
%e	Is replaced by the extension (including the period) of the first selected file.
%f	Is replaced with the fully qualified name of the first selected file. Specifying %f is the same as specifying %q%n%e.
%m	Is replaced by the make file name specified for the project in its **Settings** notebook.
%n	Is replaced by the file name (without an extension and path) of the first selected file.
%o	Is replaced by the target file name specified for the project in its **Settings** notebook.

Figure 27 (Page 2 of 2). WorkFrame substitution variables

%t	Is replaced by the file name (without an extension and path) of the project's target. **Settings** notebook.
%p	Is replaced by the fully qualified project file name.
%q	Is replaced by the path of the first selected file.
%r	Is replaced by the run options set for the project target file in the project's **Settings** notebook.
%TOPIC%	Is replaced by the contents of the Help **Topic** field in the action's **Settings** notebook. This field specified the help topic to be displayed when the user requests for help on the action.
%%	Is replaced by the % symbol.

If the first selected file is

```
d:\dogs\hounds\beagle.h
```

then:

%e is `.h`,

%f is `d:\dogs\hounds\beagle.h`

%n is `beagle`

%q is `d:\dogs\hounds\`

Note: If the file name includes spaces, as in `toy terrier.h`, then the file-name part of the substituted string will have quotation marks surrounding it. For example, %f would be replaced with `"toy terrier.h"`.

The substitution variables in the table below are used to specify error message format in error templates. WorkFrame uses error templates to parse error messages in a monitor window when you double-click on an error message to invoke the editor.

Figure 28. WorkFrame error template substitution variables

%f	Is replaced by the name of the file name where the error occurred. **Note:** If the error message does not emit a fully-qualified file name, the editor may not be able load the file if it is not located in the project's working directory.
%i	The line in the source file at which the error occurred.
%c	The column in the source file at which the error occurred.
%t	The text of the error message.

If an action emits error messages of the form

```
<file.c : 238,12 > Error message text.
```

then the error message template can be specified as:

```
<%f : %i,%c> %t
```

Default Actions

Default actions make it possible for you to invoke an action class, such as Edit, on a file without having to explicitly specify the appropriate Edit action.

For example, when you select **Edit** from the pop-up menu of a C++ source file (without following the cascading arrow) the VisualAge Editor is launched. But you could set up your Edit actions so that if you select the **Edit** menu item on the popup menu of an .IPF file, the EPM editor is launched instead.

Menu Note: If only one action in a class applies to the selected part, the class menu item appears without cascading choices.

This is possible because the project's **Tools setup** was configured so that the VisualAge Editor is the default Edit action for a C++ source file, and EPM is the default Edit action for a .IPF file.

Default actions can be defined for every action class that applies to a project file. The default action for a class is defined as the action that has the highest priority in the class. The default action for a class in the context of a specific file type is defined as the action in the class with the highest priority that applies to the type.

The priority of an action is set by a numeric field in the **Support** page of an action's **Settings** notebook. (📖 See “Action Settings - Support Page” on page 58 for information on how to set an action's priority.)

The type of files that an action applies to is determined by the **Source types** list specified for the action. The default action appears as the first action on the list of cascading choices off a class menu item on the pop-up menu of a file. In the example above, EPM was given a higher priority than VisualAge Editor, and the EPM action was set so that its list of source types does not include C++ source files. 📖 “Types” on page 75 has more information on types.

In the **Tools setup** Actions view, the actions within each class are listed in the order of their priority, so the default action for each class is always the first action in the list.

Again, take the Edit action class as an example. You could have three different editors, the VisualAge Editor, EPM, and the System Editor, that could apply to the same type of file, say “C Source.” But if the VisualAge Editor had the highest priority followed by EPM and the System Editor, then the VisualAge Editor would be the default Edit action for “C Source” files. However, if the Dialog Editor which applies to “Resource” files was also listed as an Edit action following the VisualAge Editor, EPM, and the System Editor, it would be considered the default Edit action for a “Resource” file if none of the three preceding editors also applies to “Resource” files.

In the case of project-scoped actions, the default project-scoped action for a class is defined as the first project-scoped action listed in the class grouping, that is, the project-scoped action that has the highest priority setting in the class.

The default action configurations also apply when a WorkFrame-aware tool invokes another action on its own behalf. For example, when the VisualAge C++ Browser needs to display the definition of a class, it invokes the default Edit action that applies to the source file that contains the definition. Because you can configure your default Edit actions for the different types of files in your project, the Browser would invoke the editor you choose as the default. Because the project **Tools setup** describes tools as actions, tools do not have to know the name of another tool or how to invoke it when they invoke the tool as a WorkFrame action class.

A project's default Edit action is very important. It determines which editor is launched when you double-click on error messages in the **Monitor**, and which editor is used when other tools, such as the Browser, need to use an editor.

Arrange the priority of the actions within a class so that the most restrictive actions (that is, the actions that apply to the fewest project parts) have the higher priority so that the more specialized actions become the default actions for the more specialized project parts.

Inheritance Note: When a project inherits another project's **Tools setup**, it also inherits the priority of the actions. Therefore, it could also inherit the base project's default actions. You cannot change the priority of an inherited action. For more information on how inherited actions are resolved, see "Inheritance Precedence Rules" on page 33.

Actions on Menus

Every action in your project's **Tools setup** is accessible from pop-up menus on projects or project parts, except for any actions that were configured not to appear there. (See "Action Settings - Menus Page" on page 63 for more information on how to make actions visible or invisible on project menus.) The way an action is set up and its priority within the list of actions in a class determines the pop-up menus in which the action can appear, and its location in the pop-up menu with respect to other actions of the same class.

Here is a short list of rules about action menus in WorkFrame projects:

1. Project-scoped actions appear on the project pop-up menu. You can bring up the project pop-up menu by pointing to the background of the project parts container and pressing mouse button 2. Project-scoped actions also appear on the project's system menu, as well as the **Project** pull-down menu.

2. File-scoped actions appear on pop-up menus on project parts. However, an action only appears on a project part's pop-up menu if the part matches one of the action's list of source types. Thus, the actions that appear on a project part's pop-up menu are only those that apply to the part. See "Types" on page 75 for more information about types. File-scoped actions also appear on the **Selected** menu when applicable parts are selected.

   See "Action Settings - Menus Page" on page 63 for more information on an action's settings, in particular, the **Menus** page where you can configure an action so that it appears in the **Project**, **Selected**, and pop-up menus.

3. Actions are categorized by their class in the menus. If there is more than one applicable action in a class, a cascading menu appears on the class name. If there is only one applicable action for the class, only the class name appears on the pop-up menu.
4. The actions on project menus appear in order of priority. You can change the order in which actions appear on pop-up menus by changing the priority of the actions in the **Tools setup**. "Action Settings - Support Page" on page 58 tells you how to set the priority of an action.
5. The first action in the cascading choices of the first action class menu item on the pop-up menu of a project part is the action that is invoked when you double-click on the part.
6. The first action of the cascading choices of an action class is the default action for the class that applies to the selected part or set of parts. It is the action that is invoked if the class name is selected on the pop-up menu without following the cascade.

Types

You may often need to refer to a group of project parts by name, to more easily specify the source and targets of an action, and more easily use the project **Parts filter**. (📖 See “Icon View” on page 15 for more information about the **Parts filter** filter on the project container.) You can name a group of project parts by specifying a *type*.

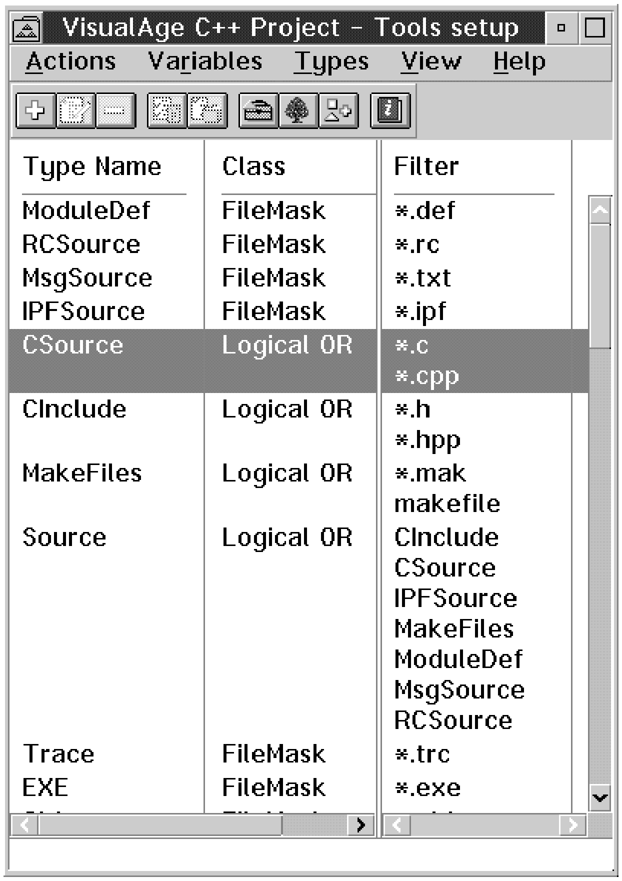
Types are used to categorize project parts. An example of a type is “C++ Source,” which can describe files whose names match the file masks *.CPP, *.HPP, *.H, *.DEF.

Types are used in three different ways:

- To determine the actions that apply to one or more selected project parts. For example, the Compile::C/C++ Compiler action applies to both “C++ Source” and “C Source.”
- To determine the targets that can be produced by an action. For example, the Compile::C/C++ Compiler action can produce “Object files.”
- To filter the parts shown in a project's container. The **Parts filter** entry field near the top of the container lets you specify the type of parts you want displayed. You can also enter a file mask in this field.

The **VisualAge C++ Project** has several predefined types, such as “Header Files,” “C Source,” “C++ Source,” “Object Files,” and “Executables.”

Project Tools Setup



Type Name	Class	Filter
ModuleDef	FileMask	*.def
RCSource	FileMask	*.rc
MsgSource	FileMask	*.txt
IPFSource	FileMask	*.ipf
CSource	Logical OR	*.c *.cpp
CInclude	Logical OR	*.h *.hpp
MakeFiles	Logical OR	*.mak makefile
Source	Logical OR	CInclude CSource IPFSource MakeFiles ModuleDef MsgSource RCSource
Trace	FileMask	*.trc
EXE	FileMask	*.exe

Figure 29. Types view of Tools Setup window

Types can be inherited from one or more projects, just like actions. The list of types available to a project are those that are defined locally in the project, and those that are inherited from other projects. “Inheritance Precedence Rules” on page 33 explains how inherited types are processed.

You can also add your own types to a project, such as “My C++ Source,” which could refer to files whose names match the file masks MY*.CPP, MY*.HPP, and so on. “Adding Types” on page 81 explains how you can add your own types.

PAM Note: The file name of the project part used to match the type filters is the file name returned by the PAM that provides access to the part. The basic OS/2 PAM returns regular OS/2 file names. See Chapter 8, “Project Access Methods (PAMs)” on page 149 for a more detailed discussion about PAMs.

Type Classes

Like actions, types are named and grouped into classes. A type's class provides the method in which a project part is determined to be a member of the type. All the type examples used so far belong to the "File Mask" type class. WorkFrame provides several predefined type classes, including "File Mask":

File Mask

This type class provides simple OS/2-style pattern matching on the file name of a project part. You specify filter patterns using wild cards.

Regular Expression

This type class is similar to "File Mask," but allows filter patterns to be specified using the full power of regular expressions. Unlike file masks, a regular expression cannot be used interchangeably with types when you use the project's **Parts filter**, or when specifying the Source and Target types of an action. You must define and use a Regular Expression type instead.

An example of a Regular Expression type is "VisualAge C++ DLLs," whose filter can be specified as: `^ICS.*\.DLL|^IWF.*\.DLL`. This identifies all the DLLs that are shipped with the VisualAge C++ product. Using file masks, you could specify them separately as `ICS*.DLL` and `IWF*.DLL`.

There are several different syntaxes for regular expressions. The one supported here is the Extended Regular Expressions (ERE) format supported by the VisualAge C Library. The syntax is described in the table below, where **a**, **b**, and **c** are regular expressions, and **n** and **m** are integers.

Figure 30 (Page 1 of 3). Regular expression syntax

a	Denotes an exact match.
.	Matches any single character. This is the same as the `?` wild card used in OS/2 file masks. To denote a literal dot character, precede the dot with a backslash, as in `\.`.
^a	Matches if `a` occurs at the beginning of the name.
a$	Matches if `a` occurs at the end of the name. For example, `^a$` matches only `a`.
a|b	Matches either **a** or **b**.

Figure 30 (Page 2 of 3). Regular expression syntax

[<list of characters>]	Matches any of the characters in the list. For example, `[abc]` would match the names `a`, `b`, and `c`. To match a dash (`-`) character, it must be placed first or last in the list (for example, `[-abc].c`). To match a caret (`^`) character, it must be placed somewhere other than in the first position. To match a close or open bracket character (`[` or `]`), it must be placed first in the list, or second after a caret.
[<range of characters>]	Matches any character in the range. For example, `[a-c]` would match the names `a`, `b`, and `c`. You could also write `[ab-c]` as an equivalent expression.
[^<characters>]	Matches any character other than those specified in the list or range. For example, `[^a-c]` will match any file not named `a`, `b`, or `c`.
a{n}	Matches **a** repeated exactly n times. For example, `a{3}` will match only the name `aaa`.
a{n,m}	Matches **a** repeated between **n** and **m** times, inclusively. For example, `a{2,3}` will match only the file names `aa` and `aaa`. If **m** is ommitted, then it assumes the value of infinity.
a?	Matches zero or one occurrences of **a**. This is shorthand for `a{0,1}`.
a+	Matches one or more repetitions of **a**. This is shorthand for `a{1,}`. For example, `a+b\.c` will match the file name `ab.c`, but not `b.c`.

Figure 30 (Page 3 of 3). Regular expression syntax

a*	Matches zero or more repetitions of **a**. For example `ba*` will match the names `b`, `ba`, and `baaaaaaaaa`. The regular expression `.*` is equivalent to the OS/2 `*` wildcard.
Notes: Because OS/2 files are case-insensitive, WorkFrame ignores case when matching regular expressions. Regular expressions can be grouped using parentheses. To match any literal regular expression character, precede it with a backslash (\).	

PAM

The filter for this type class is the name of a Project Access Method. Only parts returned by the specified PAM are members of the type.

For example, a version control tool might provide a PAM type called “Checked out” that identifies project parts that are already in use. A PAM that provides access to parts on a database might support its own types like “Locked” and “Unlocked.”

Typically, only a very specific list of types can be created from a PAM class type. The basic OS/2 PAM does not support any types.

Logical AND

The filter for this type is a list of other types or file masks. Parts that are members of *all* the listed types are also members of this type.

Logical OR

The filter for this type is a list of other types or file masks. Parts that are members of *any one* of the listed types are also members of this type.

NOT IN File Mask

This is the inverse of the **File Mask** type class. The filter for this type is a list of file masks. Parts that do *not* match any of the listed masks are members of this type.

NOT IN Regular Expression

This is the inverse of the **Regular Expression** type class. The filter for this type is a list of regular expressions. Parts that do *not* match any of the listed regular expressions are members of this type.

NOT IN Logical AND

This type class is the inverse of the **Logical-AND** type class. The filter for this type is a list of **Logical-AND** types. Parts that do *not* match any of the listed types are members of this type.

NOT IN Logical OR

This type class is the inverse of the **Logical-OR** type class. The filter for this type is a list of **Logical-OR** types. Parts that do *not* match any of the listed types are members of this type.

MakeMake Note: Only the "File Mask", "Logical OR", and "NOT IN Logical OR" classes are recognized by MakeMake in the source and target types of actions invoked during a project build. See “The MakeMake Utility” on page 104 for more information about WorkFrame's make file generation utility.

Type classes are added to the WorkFrame environment through a registration process. Registering new type classes is a specialized task. Usually only tool providers need to do. You register, change, and delete, type classes by selecting **Register...** from the Types container pop-up menu. When you register a new type class, you must provide the name of a DLL and an entrypoint that WorkFrame can call to query type membership. WorkFrame calls the type DLL entrypoint passing it a list of parts and the type filter. The entrypoint should then return a boolean value, true indicating that all the parts are valid members of the type, and false indicating that at least one of the parts is not. The predefined classes listed above are supported by the following WorkFrame-provided DLLs:

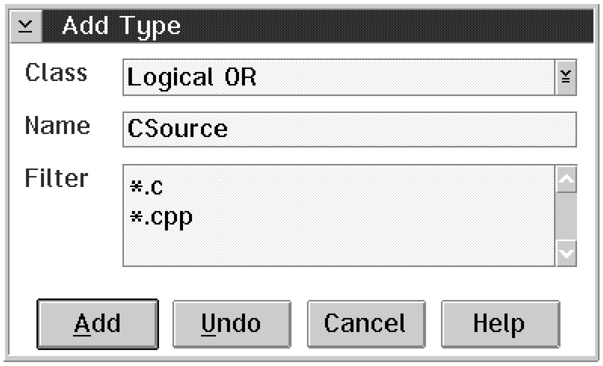
Figure 31. WorkFrame Predefined Type Classes

Type Class	**Module Name**	**Entrypoint**
File Mask	IWFTYPES	IWFFileMask
Regular Expression	IWFTYPES	IWFRegExp
PAM Name	IWFTYPES	IWFPAMName
Logical AND	IWFTYPES	IWFLogAND
Logical OR	IWFTYPES	IWFLogOR
NOT IN File Mask	IWFTYPES	IWFNotInFileMask
NOT IN Regular Expression	IWFTYPES	IWFNotInRegExp
NOT IN Logical AND	IWFTYPES	IWFNotInLogAND
NOT IN Logical OR	IWFTYPES	IWFNotInLogOR

For information on how to write a type class DLL, refer to the WorkFrame Version 3.0 Integration Kit[1].

Adding Types

To add a new type, select another similar type and bring up its pop-up menu. Select **Add...** to display the **Add Type** window.



Class: Logical OR

Name: CSource

Filter:
*.c
*.cpp

Figure 32. Adding a Type

The **Add Type** window has three fields that are filled in with the values of the type you selected when you selected the **Add...** menu item:

Name

Enter the name of your new type, for example, `My Files`.

Note: Type names cannot contain the OS/2 wildcard characters `*` and `?`. They are are interpreted as file masks otherwise.

Class

Select the class that you would like to use to evaluate your new type. For example, `Regular Expression`.

Filter

Enter the pattern or parameter that determines membership to your type. For example, `MYA*.DLL`.

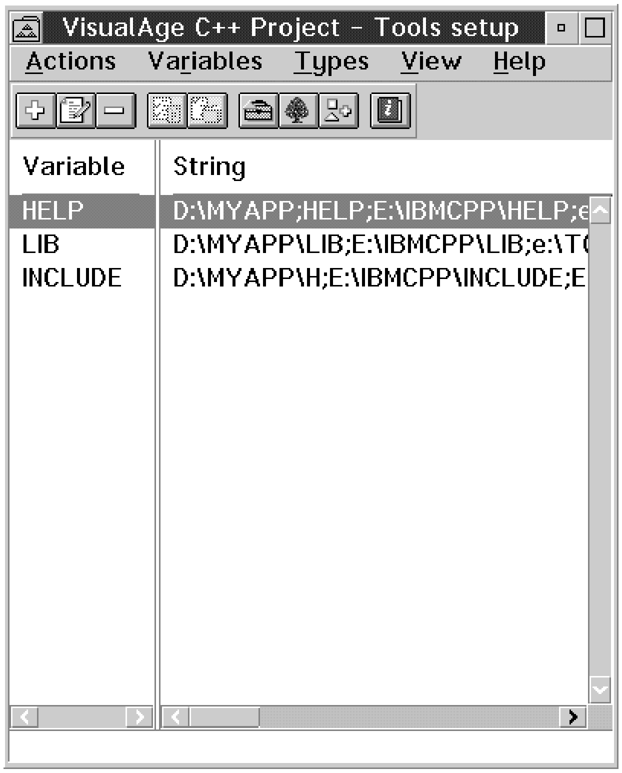
When you select the **Add** push button, your new type is added to the project.

[1] To find out when and where this kit will be available, send a note to workframe@vnet.ibm.com, or call the VisualAge C++ automated help line 1-800-992-4777. Availability will also be announced on various networks where VisualAge C++ Service and Support is present.

Environment Variables

In the project **Tools setup**, you can define environment variables that are active only for your project, without affecting variables in other sessions or those defined in your CONFIG.SYS.

Environment variables are the operating-system environment variables, like PATH and DPATH, and any other environment variables that are defined using the OS/2 SET command, such as TMP.



Variable	String
HELP	D:\MYAPP;HELP;E:\IBMCPP\HELP;e
LIB	D:\MYAPP\LIB;E:\IBMCPP\LIB;e:\T(
INCLUDE	D:\MYAPP\H;E:\IBMCPP\INCLUDE;E

Figure 33. Variables view of Tools Setup window

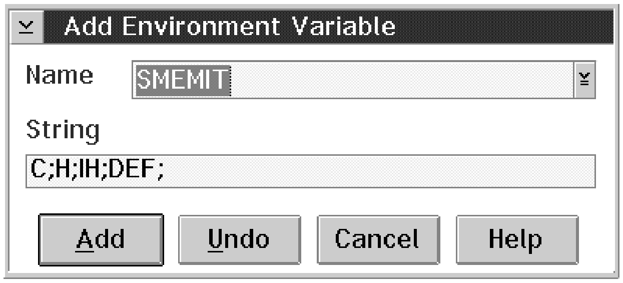
The environment variables listed in a project's environment are set for any tool launched as an action from a WorkFrame project.

Like actions and types, environment variables can be inherited from other projects. The list of variables available to a project include the variables that are defined locally in the project, and those that are inherited from other projects. “Inheritance Precedence Rules” on page 33 explains how environment variables are processed when inheritance is used.

Environment variables are stored as strings in the project environment; WorkFrame does not interpret them. Although environment variables are meant primarily for OS/2, PAMs that access non-OS/2 parts may also interpret their own environment variables. In this case, the PAM interprets and sets up the environment variables in a manner consistent with the environment it supports.

Adding Environment Variables

To define an environment variable for the project, select the **Add...** button from the Environment variables view toolbar in the **Tools setup** window. The **Add Environment Variable** window appears.



Name: SMEMIT

String: C;H;IH;DEF;

Figure 34. Adding an Environment Variable

It contains the following fields:

Name
: Enter the name of an environment variable, like PATH or LIB, or select from a list of currently defined variables. If you select a variable that has already been defined, its value is shown in the **String** field. You can redefine its value by editing the string.

String
: Enter or edit the value string of the environment variable.

When you select the **Add** push button, your environment variable is added to the project.

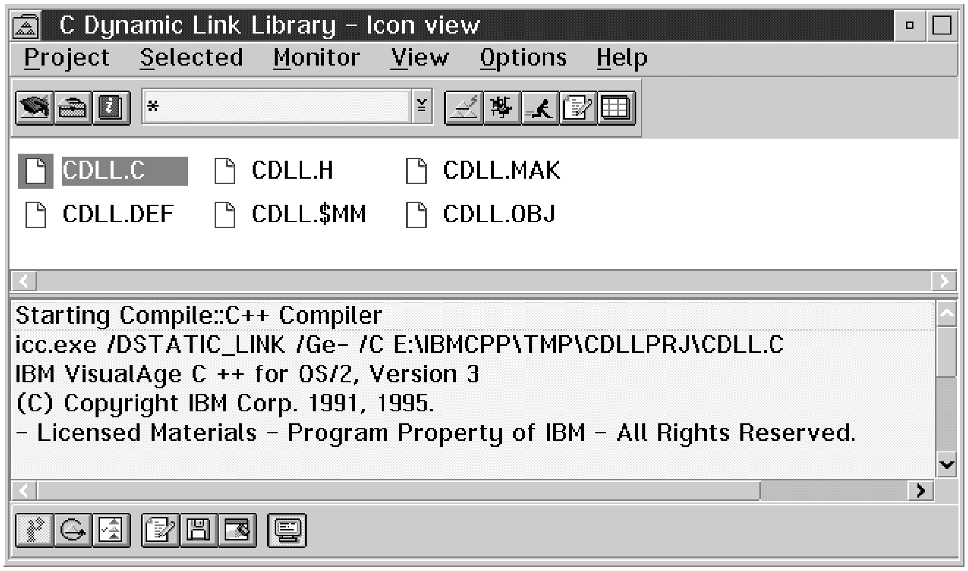
Environment variables are listed and processed in the order that they were added in the **Tools setup** window.

Project Tools Setup

4 The Project Monitor

The project **Monitor** makes it easy for you to monitor the output of the actions that you run from your project. In the **Monitor**, you can scroll through the output, save the output to a file, or copy it to the OS/2 clipboard. You can also use the actions history window to view the output of all the actions that have run in the current project session. From the **Monitor**, you can double-click on error messages to bring up the editor with the associated source file loaded. As you double-click on each error, the editor window is updated to show the line in the source file where the error occurred.

The project **Monitor** is an extension of the project view that appears below the project container when a *monitored action* is started. A monitored action is an action that has been set to run in the project's **Monitor**, and outputs to standard out. Actions can also run in full-screen or windowed sessions. Examples of monitored actions are Compile and Link.



```
Starting Compile::C++ Compiler
icc.exe /DSTATIC_LINK /Ge- /C E:\IBMCPP\TMP\CDLLPRJ\CDLL.C
IBM VisualAge C ++ for OS/2, Version 3
(C) Copyright IBM Corp. 1991, 1995.
- Licensed Materials - Program Property of IBM - All Rights Reserved.
```

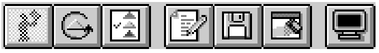
Figure 35. The project Monitor window.

After an action has finished running, the monitor shows the action's return code, and emits a high-pitched beep if the action completes successfully. If the action does not complete successfully (that is, has a non-zero return code), you will hear a low-pitched beep. You can turn off the beeps by deselecting the **Beep on completion** setting in the **Monitor** page of a project's **Settings** notebook. **Monitor** settings apply to all the projects on your system. Refer to "Project Settings - Monitor Page" on page 27 for more information about **Monitor** settings.

You can hide or show the project **Monitor** by clicking on the **Show** button on the monitor tool bar of the project window. Because the **Monitor** window and the project container are laid out on a split window, you can resize the **Monitor** relative to the project container.

Monitor Controls

The **Monitor** tool bar is located at the bottom of the project container. It has these buttons:



Stop

The **Stop** button stops an action that is currently running in the **Monitor**.

Note: Some processes may take a while to recognize the request to stop, while others may not accept any requests to stop.

The **Stop** button is only available when an action is running in the **Monitor**. It is disabled otherwise. All the other buttons in the **Monitor** are not available when an action is running.

Repeat

The **Repeat** button repeats the last action that ran in the **Monitor** during the current project session.

History

The **History** button displays the **Action History** window. It shows a list of all the actions that ran in the **Monitor** during the current project session. You can re-execute one or more actions from this list and view the results of each invocation.

Errors

The **Errors** button invokes the project's default editor on an error selected in the **Monitor** list box. If no error is selected, the editor is invoked for the first error that appears for the current action.

Save
The **Save** button saves the contents of the **Monitor** list box into a file that you name.

Erase
The **Erase** button deletes the contents of the **Monitor**.

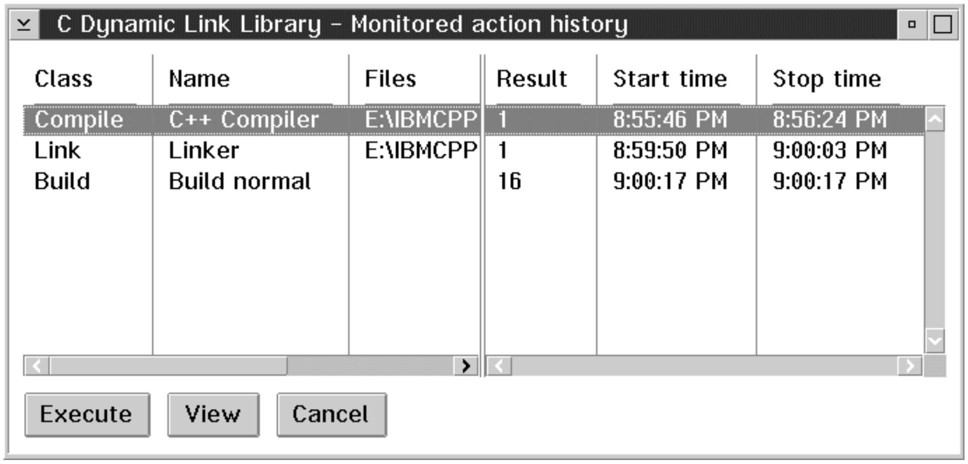
Show
The **Show** button toggles the project view to show or hide the **Monitor**.

These controls are also available from the **Monitor** menu on the project menu bar.

The toolbar also contains an information area where the name of the currently running action is shown.

Action History

The **Action History** window shows a chronological list of all the invocations of every action that ran in the **Monitor** during the current project session. It shows the start and stop times of each invocation, along with their results and output. It also shows you the files that were processed during each invocation.



Class	Name	Files	Result	Start time	Stop time
Compile	C++ Compiler	E:\IBMCPP	1	8:55:46 PM	8:56:24 PM
Link	Linker	E:\IBMCPP	1	8:59:50 PM	9:00:03 PM
Build	Build normal		16	9:00:17 PM	9:00:17 PM

Figure 36. Action History window

Select an action and then select the **Execute** push button to rerun the action. To view the output of the invocation, select the **View** push button. WorkFrame saves the contents of the **Monitor** after the action has run so that you can review the results of each invocation.

Note: You cannot edit a file from the **History** window by double-clicking on an error line.

Editor Interaction with the Monitor

Once an action has finished running in the **Monitor**, you can double click on an error message to invoke the default editor on the source file where the error occurred. The editor appears with the source file loaded. If your default editor is the VisualAge Editor, the error text is inserted below the relevant line in the source, and highlighted in a different color. If your default editor is the Enhanced Editor (EPM), all the lines that contain an error are highlighted, and the cursor is positioned at the first error line. Other WorkFrame-enabled editors may behave differently. See “Default Actions” on page 71 for more information about how to determine and set your default editor.

You can also use the **Errors** button on the **Monitor** toolbar to open an edit session on a selected error. If no error is selected, the editor is opened for the first error in the current action invocation.

If your default editor is not WorkFrame-enabled, the editor will appear with the source file loaded.

Note: Double-clicking on error messages from certain actions, like Link, will not invoke the editor if the error occurs in an file that is not editable, such as an object file.

As you double-click on other error messages in the **Monitor**, the editor updates itself by scrolling to the position in the source file where the error occurred. Both the VisualAge C++ Editor and EPM have controls that let you move to the next or previous error in the source file, and get help on the error message. For more information about the VisualAge C++ Editor, refer to the part about the VisualAge Editor called “Editing Files” in the *IBM VisualAge C++ User's Guide*. Information on EPM is available from its online help.

If you are using EPM Version 6.0, you can have EPM use an existing editor session when you double-click on an error in the **Monitor**. Change the Edit::EPM action options to include the `/R` option.

Monitor Notes

Here are some important facts about the **Monitor**.

Limitations

1. Double-clicking on an error line in the **Monitor** may not work for a Make action that runs on a hand-written make file or on an edited WorkFrame-generated make file. The **Monitor** parses the Make output to determine which action reported the error. It relies on the make file to produce a specific output format so that the errors can be parsed successfully.
2. Actions that run in a DOS session will always have a return code of zero, even if the action did not complete successfully.
3. Actions that run in a DOS session cannot have more than 128 characters in the argument string.

Troubleshooting

If double-clicking on an error in the monitor fails to invoke your default editor, try the following:

1. A previous DDE session between the **Monitor** and the editor may still be active, and must be terminated.

   In the VisualAge Editor, select **Delete messages** from the **Action** menu to delete the messages and reinitialize the edit session.

   In the version of EPM that ships with OS/2 (Version 5.52), you must end the current edit session on the source file before restarting the action. In Version 6.0 of the EPM compiler, you can select **End DDE session** from the **Compiler** pull-down.

2. Check whether the error message outputs an unqualified source-file name. If the full path of the source file is not present, the editor may be unable to find it if it is not in the project's working directory.
3. Ensure that the **Send errors to the monitor** check box is selected in the editor's options.

   To open the editor's options dialog, open the project's **Tools setup** window, and select **File options** → **Change** from the editor's pop-up menu.

4. If the action that emitted the error uses the default Actions Support DLL, ensure that the action's **Error template** correctly describes the error format. You specify the **Error template** in the action's options dialog. 📖 See “Substitution Variables” on page 69 for more information on how to use substitution variables to specify an error template.

   If the action uses its own customized Actions Support DLL, the error template is already built in.

Building Your Target

This chapter introduces you to two very useful WorkFrame utilities: Build and MakeMake. The Build utility dynamically builds your project's target and manages your make file for you. MakeMake is WorkFrame's make file creation utility.

Build and Make

A Build action and a Make action are very similar. Both are project-scoped actions used to build the target of a project. There are, however, very important differences between the two:

Make

The Make action runs a make utility, like NMAKE, against an existing make file. The make file is typically a project part that was generated by the WorkFrame make file generation utility, MakeMake, or written by hand. A make file is a static object that reflects the project settings at the time the make file was generated. Whenever changes are made to the project settings, such as the action options, you must update or regenerate the make file to reflect the most current project settings.

Make files are useful when you want to package the source files for distribution, and when you want to build the target in a constant and predictable manner. A project may contain multiple make files to build the same target in some variation, for debugging or profiling, and optimization, for example. However the project recognizes only one make file as the object of a project-scoped Make action. A file-scoped Make action can be invoked on any make file in the project.

Build

A Build action runs the WorkFrame Build utility, which also uses a make file and make utility to build the project target. However, the Build utility can dynamically generate the make file and dependencies file each time a Build is initiated. Therefore, if the project settings change, the build values are implicitly updated. You don't have to do anything to update the make file beyond changing the project settings.

The build utility also manages dependencies between projects in a project hierarchy. It can build the projects lowest in the hierarchy before building the ones higher up. You can initiate a build anywhere within a project hierarchy.

The Build utility also provides two additional conveniences: it can lock projects to prevent concurrent builds from colliding, and validate that the build target is not in use before the build is started.

The Build Utility

The WorkFrame Build utility eliminates the need for explicitly generating and maintaining make files. It uses the MakeMake utility to generate a new make file each time a Build is initiated. You can also set Build options to have the Build utility build all targets, even if they are not out of date with respect to their dependent files, or to effectively perform a Make action by only building the out-of-date files. See “Setting Build Options” on page 93 for more information on how to set up a build action for a project.

Build Prerequisites

The Build utility relies on a well-defined project setup to correctly generate the build rules and dependency information for your project:

- The project dependencies must be expressed within the project hierarchy. Follow the guidelines in “Organizing Projects” on page 38.
- Correct Build options, especially the build actions, must be set as described in “Setting Build Options” on page 93 .
- The actions must have the source and target types set up correctly. See “Action Settings” on page 49 for more information on how to set up actions.
- The Actions Support DLLs associated with each action involved in the Build must provide a correct list of dependencies to the MakeMake utility.

Note: The VisualAge C++ actions are already configured correctly for use in your builds. If you use only the VisualAge C++ actions in your builds, you only need to concern yourself with the first two prerequisites.

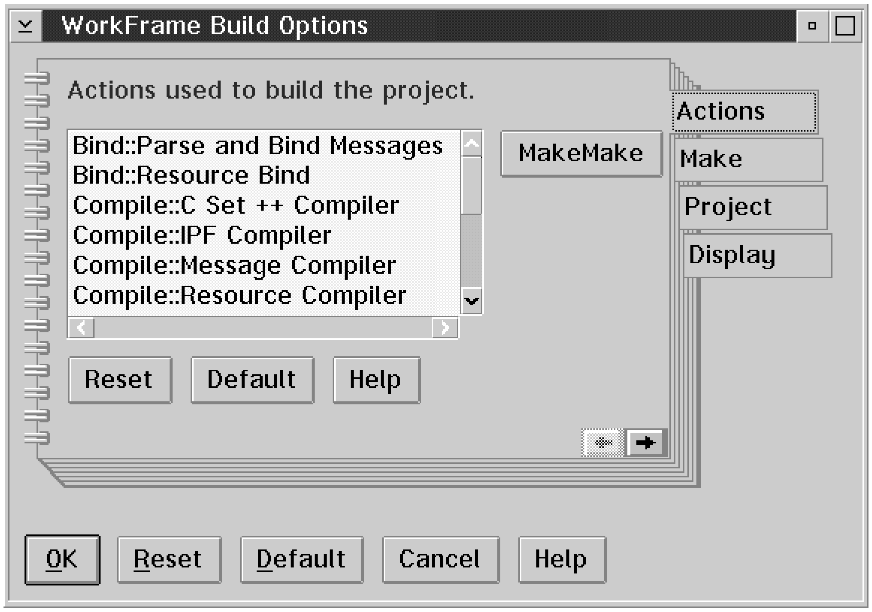
You start a build action on your project by selecting **Build** from the project's pop-up menu or toolbar.

Setting Build Options

You set Build options for a project the same way you set options for any other action. Open the project's **Tools setup** and then select **Project options** from the Build action's pop-up menu. The **Build** options notebook has the following pages:

- **Actions**
- **Make**
- **Project**
- **Display**

Build Options - Actions Page



WorkFrame Build Options

Actions used to build the project.

Bind::Parse and Bind Messages
Bind::Resource Bind
Compile::C Set ++ Compiler
Compile::IPF Compiler
Compile::Message Compiler
Compile::Resource Compiler

Figure 37. Build options - Actions page

The most important information needed to build a project's target is the set of actions needed to build it. You select these actions from the **Actions** page of the Build options notebook.

The **Actions** list box contains a list of file-scoped actions eligible for participation in a build. These actions have both source and target types specified in their settings. The Build utility attempts to formulate the build rules for your project by examining the source files and the source and target types of file-scoped actions in the project.

Building Your Target

You can select the build actions from the **Actions** list box on this page, or you can select the **MakeMake** button to invoke the MakeMake utility and select actions, and perhaps explicitly create a make file, from there. Where you select the Build actions depends on:

- Whether you want to have descendant projects use the same set of Build actions. If you do, you will need to:

  1. Select the Build actions from the **Actions** list box on this page.
  2. Select the **Pass Build settings to child projects** on the **Projects** page.
  3. In the Build options for the descendant projects select the **Use build settings from parent project** in the **Project** page (this is the default).

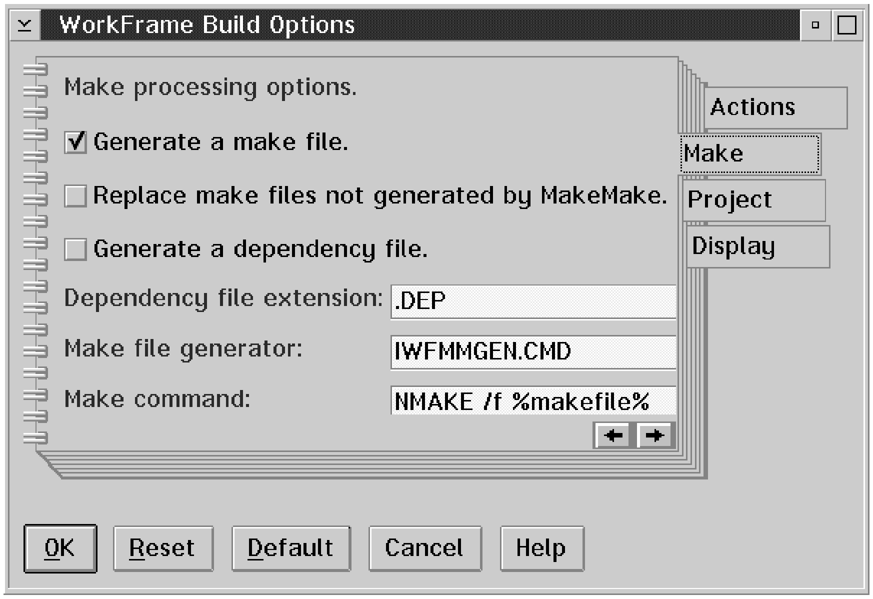
  The Build actions you select in the **Actions** list box apply to all the source files in the project. If you only want the Build actions to apply to some of the source files in the project, you must select the Build actions and applicable source files from MakeMake.

- Whether you want to explicitly select source files to which the Build actions should apply. If you do, you will need to select the Build actions from MakeMake. Descendant projects cannot use the current project's Build actions if they are specified from MakeMake.

  MakeMake saves the actions from your last successful make file generation in the extended attributes of the generated make file. If you already have a make file generated by MakeMake, you can use the previously saved actions by deselecting any actions selected in the **Actions** list box.

  Note: If any actions in the **Actions** list box are selected, those actions are used for the Build, even if Build actions were previously set from MakeMake.

Build Options - Make Page



Make processing options.

☑ Generate a make file.

☐ Replace make files not generated by MakeMake.

☐ Generate a dependency file.

Dependency file extension: .DEP

Make file generator: IWFMMGEN.CMD

Make command: NMAKE /f %makefile%

Figure 38. Build options - Make page

The options in the **Make** page relate to the make utility, and how the make file is managed and maintained.

Make processing options

Select whether or not the make file and separate dependency file are to be generated before each build. Generating these files before each build guarantees that the make file used to build the project target is up to date with the latest project and action settings.

Generate a make file

Select this option if you want the make file generated before each build. If you do not select this option, the existing make file is used.

Generate a dependency file

Select this option to create a separate dependencies file before every build. If you select the **Generate a make File** option without selecting this option, then the dependencies are stored with the make file.

Dependency file extension

Enter the file name extension for the dependency file, if one is to be generated. The default is `.DEP`.

Selecting both of the **Generate a make file** and **Generate a dependency file** options frees you from having to maintain the make and dependency files. If you do not select either of the options, the make file and dependency file are generated once and then never updated after that. You can then update the make file and dependency file manually by running MakeMake.

Make utility

Use these fields to identify the make utility and options you want to use to build the target.

Make file generator

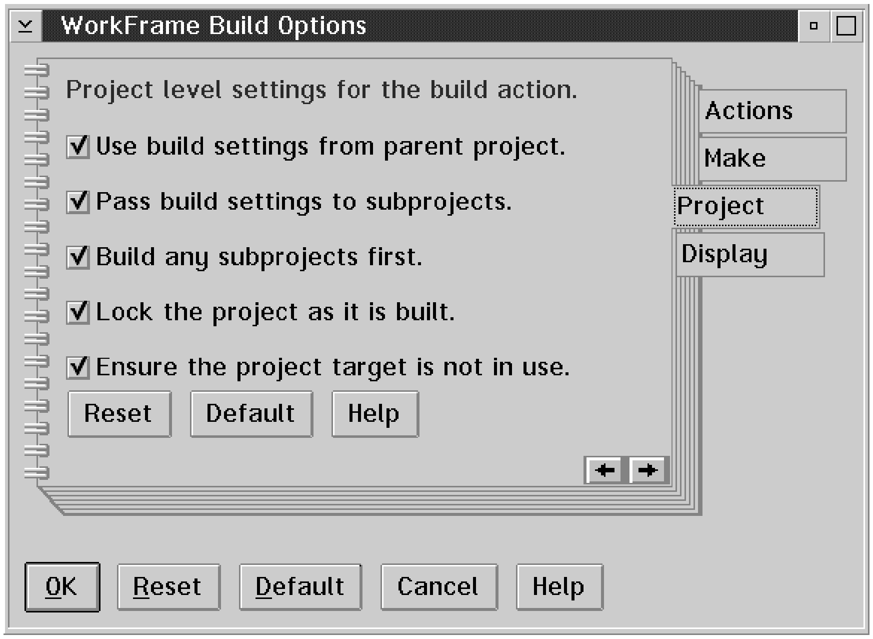
The make file generator is a script used by MakeMake to generate the make file in a format understood by the make utility. (🕮 See “Compatibility with Make Utilities” on page 108 for more information about make file generator scripts and how they can be used to generate make files that are compatible with any make utility.) The default script is `IWFMMGEN`, supplied by WorkFrame. This script enables MakeMake to generate make files with relative path names that are compatible with the NMAKE make utility included with VisualAge C++.

Make command

This field contains the command line to execute the make utility, including the required options. The default is to use the NMAKE utility from the IBM Developer's Toolkit with the command line `NMAKE /f %m`, which invokes the NMAKE utility on the project's designated make file. 🕮 See “Substitution Variables” on page 69 for a table of valid WorkFrame substitution variables.

💡 If you need a command run over the entire project hierarchy, you can specify any command in the **Make command** field, build-oriented or not. You should define a separate Build action for this. Copy an existing Build action by highlighting it, and then selecting the **Add** button on the toolbar.

Build Options - Project Page



Project level settings for the build action.

☑ Use build settings from parent project.

☑ Pass build settings to subprojects.

☑ Build any subprojects first.

☑ Lock the project as it is built.

☑ Ensure the project target is not in use.

Figure 39. Build options - Project page

The options on the **Project** page are only relevant if the project is part of a project hierarchy. See “Organizing Projects” on page 38 for more information on how to organize projects into hierarchies to establish dependency relationships between projects.

Use Build settings from parent project

Select this option to assume the Build settings from the parent project, if one exists. This feature enables a set of build actions to be specified at the root project level and have them be used by the root project and all its descendant projects, whether or not build actions have been configured for the subprojects. If the set of build actions for a descendant project needs to be different from those of its parent, you should explicitly set the build actions for the descendant project.

If the Build utility detects that a build action selected in the parent does not apply to the descendant project, or that the action does not exist in this project, it ignores the action when it formulates the build rules for the current project. If it cannot formulate the build rules from the build actions assumed from the parent project, the build will fail.

The build utility always uses the action options from the current project to build the target unless it has the **Use Build settings from parent project** setting selected.

Pass Build settings to subprojects

Select this option to allow any nested projects to assume this project's Build settings. If you do not select this option, and the subprojects are set to use the Build settings from their parent, they will assume the Build settings from this project's parent instead, if one exists. If this project is at the root of a project hierarchy, its subprojects must define Build settings of their own or their builds will fail.

Build any subprojects first

Select this option to have the Build utility build any subprojects before building this project's target. Select this option if your project has dependencies on the targets of the projects it nests.

Project hierarchies are built using a depth first search throughout the project tree. Any projects at the same level are built in an unspecified order. No dependencies should exist between projects on the same level in the project hierarchy. See “Organizing Projects” on page 38 for more information on project hierarchies.

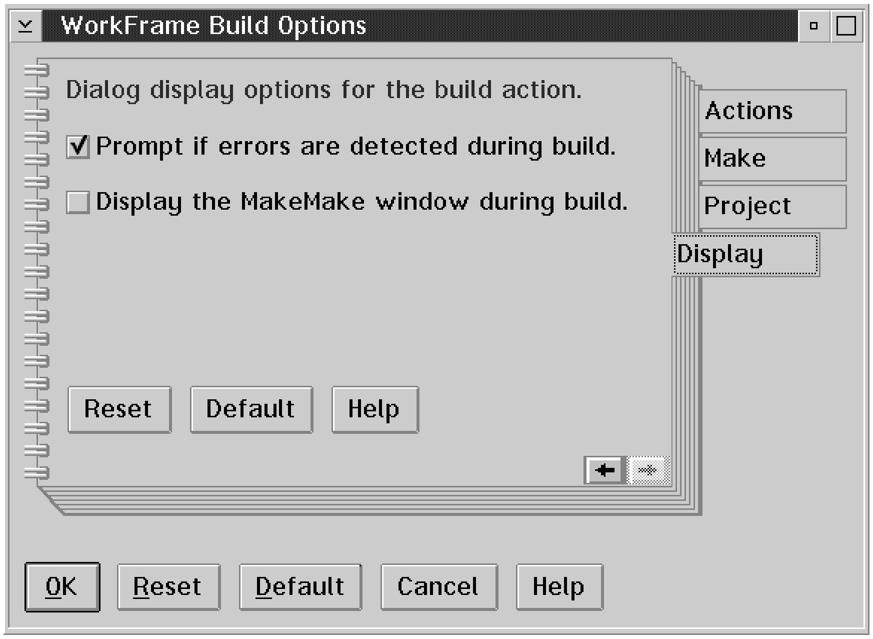
Lock the project as it is built

Select this option to have the build utility lock the project's target as it is being built. This ensures that the build will not fail because another process has gained access to the project's target while it is being built. For example, if the target program is running when the Build utility is trying to build it, the build will fail because the target file is locked.

Ensure the project target is not locked

Select this option to have the build utility check if the target can be accessed before starting the build. If the target is in use, the build will not start.

Build Options - Display Page



WorkFrame Build Options

Dialog display options for the build action.

☑ Prompt if errors are detected during build.

□ Display the MakeMake window during build.

Figure 40. Build options - Display page

Use the display page to set options for prompts and windows to be displayed during the build.

Prompt if errors are detected during build

Select this option to have the build utility display a message box to report an error. This is the default setting. If you do not select this option, any errors are written to standard out (viewable from the project monitor).

Display MakeMake window during build

Select this option if you want the build utility to display the MakeMake window during the build so that you can generate the make file explicitly before the build is run. The default is not to display the MakeMake window.

You will only need to select this option if all of the following are true:

- You selected the option to generate a make file as part of the build (**Make** page).
- Your build actions were selected through MakeMake (that is, no build actions were selected from the **Actions** page).
- You think that the way the make file is generated will change often between builds. For example, if you add another C++ file, you might want to add it to the source files selected in the **Make Make** window. (See “Generating Make Files” on page 104 for more information about the **MakeMake** window.) If you select your build actions from the **Actions** page, on the other hand, all applicable source files are processed automatically.

Running Build from the Command Line

You can also run the Build utility from the command line. To invoke the Build utility from the command line, you need to specify the name of the project whose target to build, along with any other options you might need.

Command Line Notes:

- When you run the Build utility from the command line, specifying only the project name, the default Build action for the project is launched with the options that were defined for the action. (📖 See "Default Actions" on page 71 for information about the default action within a class of actions.) You only need to specify the optional command-line parameters if you want to override the options already set for the action.
- If you do not specify any build actions on the command line, and no Build actions were set in the Build action options, the Build utility uses the Build actions from the last generated make file. Failing that, it displays the MakeMake window so that you can select the build actions from there.

The general syntax of the Build utility is:

```
iwfbuild /PROJ[ECT] <project> [options]
```

Required Options:

`/PROJ[ECT] <project>`

The name of the project to build. You can name the project with its path name (for example, `/PROJ D:\DESKTOP\MY_PROJECT`) or its persistent object identifier (for example, `/PROJ <MY_PROJECT>`).

Make file generation options:

`/A[CTION] <action>`
An action to be used in building the project's target, specified in the form `<action class>`::`<action name>`. You can repeat this option as many times as there are build actions to specify. If no actions are specified, the actions and source files from the last successful make file creation are used.

`/GENM[AKEFILE]`
Generate a make file as part of the build. This is the default.

`/NOGENM[AKEFILE]`
Do not generate a make file as part of the build; use the existing make file.

`/GEND[EPENDENCY] [<extension>]`
Generate the dependency information in a separate file. You can also specify the dependency file file name extension. The generated dependency file will have the same name as the make file, with a default extension of .DEP.

`/GENS[CRIPT] <script>`
The make file generation script to use. The default, `IWFMMGEN.CMD`, produces make files compatible with the NMAKE utility.

Make invocation options:

`/M[AKE] <invocation command>`
The make utility to run and its associated parameters. WorkFrame substitution variables can be used in the parameter string. See "Substitution Variables" on page 69 for a list of substitution variables you can use.

`/GENF[ORCE]`
Create the make file even if MakeMake detects that it has been user-modified.

`/NOGENF[ORCE]`
Prompt to determine whether or not to continue creating the make file if MakeMake detects that it has been user-modified. This is the default.

Project options:

/B[UILDSUBPROJECTS]
: Build any descendant projects first. This is the default.

/NOB[UILDSUBPROJECTS]
: Do not build any descendant projects.

/U[SEPARENTOPTIONS]
: Use the build settings from the parent project, if one exists. This is the default.

/NOU[SEPARENTOPTIONS]
: Do not use build settings from the parent project. Use this project's own build settings.

/PA[SSOPTIONSTOSUBPROJECTS]
: Pass build settings to any subprojects.

/NOP[ASSOPTIONSTOSUBPROJECTS]
: Do not pass build settings to subprojects. Subprojects should obtain their settings from this project's parent instead. If the project has no parent, and its subprojects have no Build settings defined, the build will fail on the subprojects.

Miscellaneous options:

/C[HECKTARGETS]
: Check if the target can be accessed before starting the build. If the target is locked, the build will not start.

/P[ROMPT]
: Display a message box to report an error. This is the default.

/NOP[ROMPT]
: Do not display a message box to report an error. Write the messages to standard out.

/OPTIONC[LASS]
: The name of the build action class to invoke. The default is `Build`.

/OPTIONN[AME]
: The name of the Build action to invoke. The default is to invoke the default action of the project's build class.

The MakeMake Utility

Use the MakeMake utility to generate make files for your project. It can generate a make file with the dependencies built in, or it can generate a separate dependencies file.

MakeMake creates a make file for your project by examining the actions and source files associated with your project and then trying to determine the correct sequence of commands to build the project's target. Typically, in a hierarchy of projects, one make file is generated per project. The Build utility handles the dependencies between projects and determines the order in which each project's make file should be processed to build the target of the current-level project.

Generating Make Files

The **MakeMake** window has two list boxes:

Actions
All the file-scoped actions in your project that have both source and target types specified in their settings are listed here. Select the actions necessary for building the target of your project. For example, if your project builds a simple DLL, you might select the Compile::C/C++ Compiler and Link::Linker actions from the **Actions** list.

Source Files
All the project files or parts that match the source types for the selected actions are listed here. Select the source files that are to be processed by the make file.

For example, if you have a project that builds a C DLL with the Compile::C/C++ Compiler and Link::Linker actions, select the .C source files. You do not have to select header files because the Compile::C/C++ Compiler Actions Support DLL, CPPICC30, parses C and C++ source files to detect any dependencies on included header files.

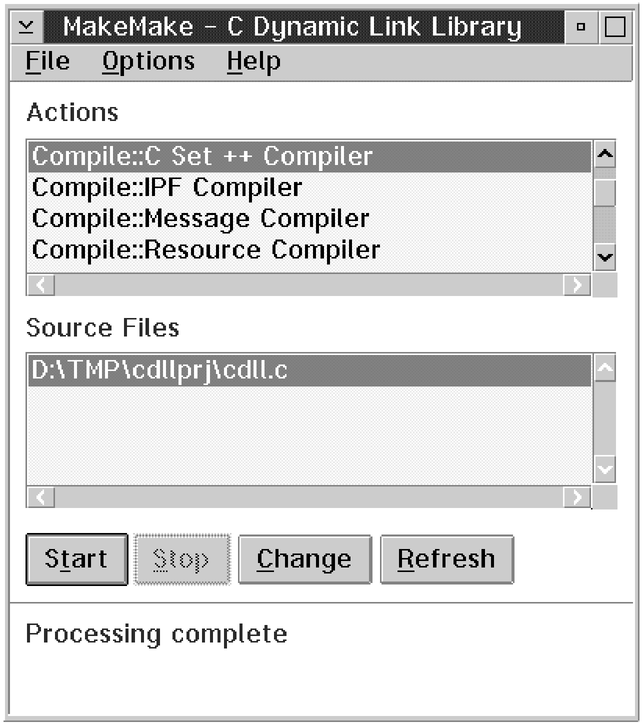
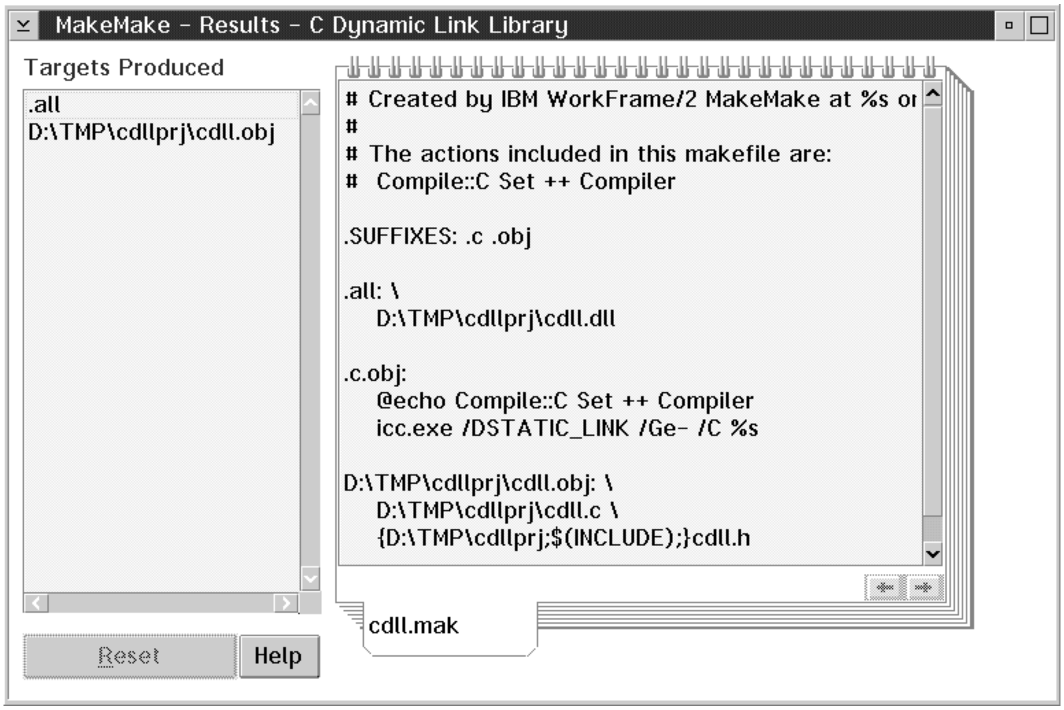


Figure 41. MakeMake window

When you have selected the actions and source files to build your target with, select the **Start** push button to start the make file generation. After a short while, the MakeMake **Results** window appears. It displays the generated make file. The targets that are produced by each action in the make appear on a list box on the left side of the **Results** window. You can edit the make file from this window. If a separate dependency file was generated, the make file and dependency file are displayed in a notebook format. Use the notebook tabs to view the dependency file.



```
# Created by IBM WorkFrame/2 MakeMake at %s or
#
# The actions included in this makefile are:
#  Compile::C Set ++ Compiler

.SUFFIXES: .c .obj

.all: \
   D:\TMP\cdllprj\cdll.dll

.c.obj:
   @echo Compile::C Set ++ Compiler
   icc.exe /DSTATIC_LINK /Ge- /C %s

D:\TMP\cdllprj\cdll.obj: \
   D:\TMP\cdllprj\cdll.c \
   {D:\TMP\cdllprj;$(INCLUDE);}cdll.h
```

Figure 42. MakeMake Results window

Note: The MakeMake **Results** window will not appear if the **Always show make file** option on the MakeMake **Options** menu is not selected. In that case, you can display the **Results** window to view the generated make file by selecting the **Change** push button from the MakeMake window.

To save the make file, close the make file **Results** window and return to the main MakeMake window. Press **F4** to save the make file and exit MakeMake.

MakeMake generates make files using build rules determined by examining the selected actions and source files. It does not use information from any existing make files in your project to generate a new make file. If you have a make file that you want to keep, rename it so that it is not overwritten by the newly generated make file.

Limitations

The MakeMake utility creates make files that build a single target by invoking a series of actions. Since projects are typically organized as a hierarchies of projects, with each project in the hierarchy representing a single target, MakeMake works very well with WorkFrame projects. However, there are some limitations that you should be aware of.

Note: These limitations would only concern you if you need to:

- Work with projects that use multiple PAMs
- Build with actions that use Actions Support DLLs not provided with VisualAge C++.
- Use actions that specify types that are not provided with VisualAge C++

Here are four important MakeMake limitations:

- The source and target types of each action involved in the make must be specified correctly so that MakeMake can infer the order in which the actions are to be executed.
- The types specified in each action's settings for **Source types** and **Target types** must be of the type classes "FileMask", "Logical OR", and "NOT in Logical OR" or MakeMake will ignore the type. (See “Type Classes” on page 77 for information about type classes).

  All the types included with VisualAge C++ are of the allowed classes, "FileMask", "Logical OR", and "NOT in Logical OR". The other type classes can be used to specify the source types of actions that are not used in builds (such as Edit), and for the project **Parts filter**.
- MakeMake only works with project parts that are accessed by the basic PAM, IWFBPAM and other PAMs derived from it.
- MakeMake calls application programming interfaces (APIs) in the Actions Support DLLs of every action involved in the make to determine the list of project parts or files the source is dependent on. (See “Action Settings - Support Page” on page 58 for more information about Actions Support DLLs). For example, if the Link::Linker action is involved in a build, MakeMake calls its Actions Support DLL to determine the list of Link dependencies, which might include one or more .OBJ files and a .MAP file. The Actions Support DLL for a Compile action returns a list of dependencies that includes the header files that were included by the C or C++ source.

  The accuracy of the dependencies and targets list is dependent upon the Actions Support DLL for each action, not the MakeMake utility. If MakeMake fails to generate the correct list of dependencies or targets for an action external to VisualAge C++, contact the supplier of the Actions Support DLL.

Compatibility with Make Utilities

The make files produced by MakeMake are, by default, compatible with NMAKE, the IBM Developer's Toolkit make utility. MakeMake can also produce make files that are compatible with other make utilities if the make utility, or some other third party, provides a generation script that transforms the intermediate make file produced by MakeMake to its own format. WorkFrame provides a default generation script, called IWFMMGEN.CMD, that produces make files compatible with the NMAKE utility.

The intermediate make file produced by MakeMake has the same name as the final make file, with a `.$mm` file name extension.

Vendors of other make utilities can supply their own script to enable MakeMake to generate make files in their required format. For more information on how to write a make file generation script for MakeMake, obtain the WorkFrame Version 3.0 Integration Kit[1].

Using MakeMake from the Command Line

The MakeMake utility can be invoked from the command line as well as from the project pop-up menu. To invoke MakeMake from the command line, optionally specify the name of the project to process, along with any other options you might want. If you do not specify a project name on the command line, the **MakeMake** window appears empty. You can load a project from the **MakeMake** window by selecting **Open project** from the **File** menu.

The general syntax of the MakeMake utility is:

```
iwfmmake [options]
```

The options are:

`/PROJ[ECT] <project>` The name of the project to be processed. You can name the project with its path name (for example, `/PROJ D:\DESKTOP\MY_PROJECT`) or its persistent object ID (for example, `/PROJ <MY_PROJECT>`).

[1] To find out when and where this kit will be available, send a note to workframe@vnet.ibm.com, or call the VisualAge C++ automated help line 1-800-992-4777. Availability will also be announced on various networks where VisualAge C++ Service and Support is present.

Make file generation options:

`/A[CTION] <action>`
: An action to be used in building the project's target, specified in the form `<action class>`::`<action name>`. You can repeat this option as many times as there are make actions to specify. If no actions are specified, the actions from the project's default Build action are used. There are no Build actions specified, the actions and source files from the last successful make file creation are used.

`/GEND[EPENDENCY] [<extension>]`
: Generate the dependency information in a separate file. You can also specify the dependency file's file name extension. The generated dependency file will have the same name as the make file, with a default extension of .DEP.

`/GENS[CRIPT] <script>`
: The make file generation script to use. The default, `IWFMMGEN.CMD`, produces make files compatible with the NMAKE utility.

`/GENF[ORCE]`
: Create the make file even if MakeMake detects that it has been user-modified.

`/NOGENF[ORCE]`
: Prompt to determine whether or not to continue creating the make file if MakeMake detects that it has been user-modified. This is the default.

Miscellaneous options:

`/P[ROMPT]`
: Display a message box to report an error. This is the default. If you also specify the `/NODISPLAY` option, prompts that require a response are displayed in an error box (these only occur in exceptional situations). Errors are sent to standard out.

`/NOP[ROMPT]`
: Do not display a message box to report an error. Send messages to standard out. This option is only valid if `/NODISPLAY` is also specified.

`/D[ISPLAY]`
: Display the **MakeMake** window. This is the default.

`/NOD[ISPLAY]`
: Do not display the **MakeMake** window (execute the make file generation in batch mode). `/NOPROMPT` implies `/NODISPLAY`.

The MakeMake utility fails and returns these error codes under the following conditions:

88665 The access method (PAM) failed.

88666 One or more actions are not related to any other actions.

88667 No files that match the source types of <action> were found.

88668 An action creates a file not listed in the action's list of target types.

88669 An action's support DLL did not return any source files to be processed.

88671 The <action> action does not produce any files.

88672 An action returned an invalid command line string.

88673 Help could not be initialized.

88675 No valid actions were defined.

88682 A file was selected as a source file for the <action> action, but is also listed as a dependency for the <action> action. You will be prompted whether or not to continue.

88683 A loop was detected in the list of actions to process. You will be prompted whether or not to continue.

88684 Error loading project files.

88685 Bad parameter.

88686 Error allocating memory.

88687 Project load cancelled by user.

88689 Unable to invoke the specified make file generation script.

88690 Unable to write file <file> to disk.

88691 The make file has been modified since the last time it was generated. You will be prompted whether or not to continue generating the make file.

88692 The project has no source directories specified, and thus has no source files.

Note: MakeMake returns error codes only if it was invoked with the `/NOP` or `/NOD` options.

6 Project Smarts

Project Smarts is a powerful tool to help you quickly get started writing VisualAge C++ applications. It is a catalog of skeleton applications you can use as a base with which to write your own applications.

To use Project Smarts, find the Project Smarts icon in the VisualAge C++ folder and double-click on it to open the VisualAge C++ Project Smarts catalog. It contains projects of common applications including:

- UI Class Library Application
- Presentation Manager Application
- Workplace Shell Application
- Direct-to-SOM Application
- Data Access Application
- Visual Builder Application
- Resource Dynamic Link Library
- C++ Dynamic Link Library
- C Dynamic Link Library
- IPF Context-Sensitive Help
- IPF Document

When you instantiate one of these Project Smarts applications, a fully-configured, development-ready project is created on your desktop. All the actions, options, and environment variables you need to develop a similar application are preconfigured for you. Each project is created with template source files to help you get started quickly on the real work, without having to set up the basics every time. Project Smarts applications are skeleton programs that you can actually build and run.

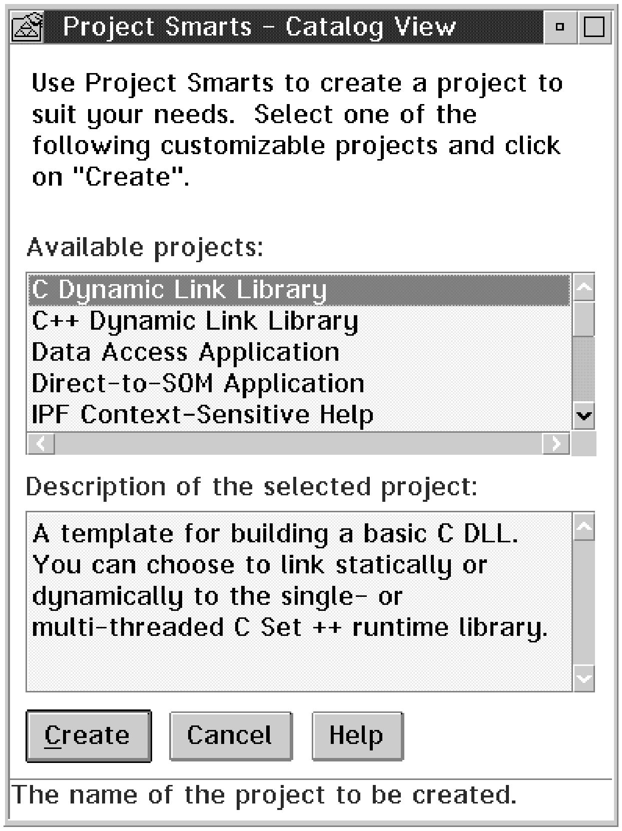
A Project Smarts application is distinct from a sample project; it does not teach you programming techniques or concepts. The code provides a starting point for you to build on when developing your own applications from the code templates.

The VisualAge C++ **Samples** folder is another resource for creating fully-configured projects. You could create a project and have it inherit from a similar sample project so that your new project uses the sample project's **Tools setup**. (See “Project Settings - Inheritance Page” on page 29 for more information on project inheritance.)

Creating Projects from Project Smarts

Project Smarts is one of many ways you can create a WorkFrame project. You create projects from Project Smarts when you want to develop an application similar to the application skeletons in the VisualAge C++ Project Smarts catalog.

To open the VisualAge C++ Project Smarts catalog, double-click on its icon. The **Project** list box in the catalog contains a list of VisualAge C++ applications as shown in Figure 43. As you select an project title, the **Description** field is updated with a short description of it.



Project Smarts - Catalog View

Use Project Smarts to create a project to suit your needs. Select one of the following customizable projects and click on "Create".

Available projects:

C Dynamic Link Library
C++ Dynamic Link Library
Data Access Application
Direct-to-SOM Application
IPF Context-Sensitive Help

Description of the selected project:

A template for building a basic C DLL. You can choose to link statically or dynamically to the single- or multi-threaded C Set ++ runtime library.

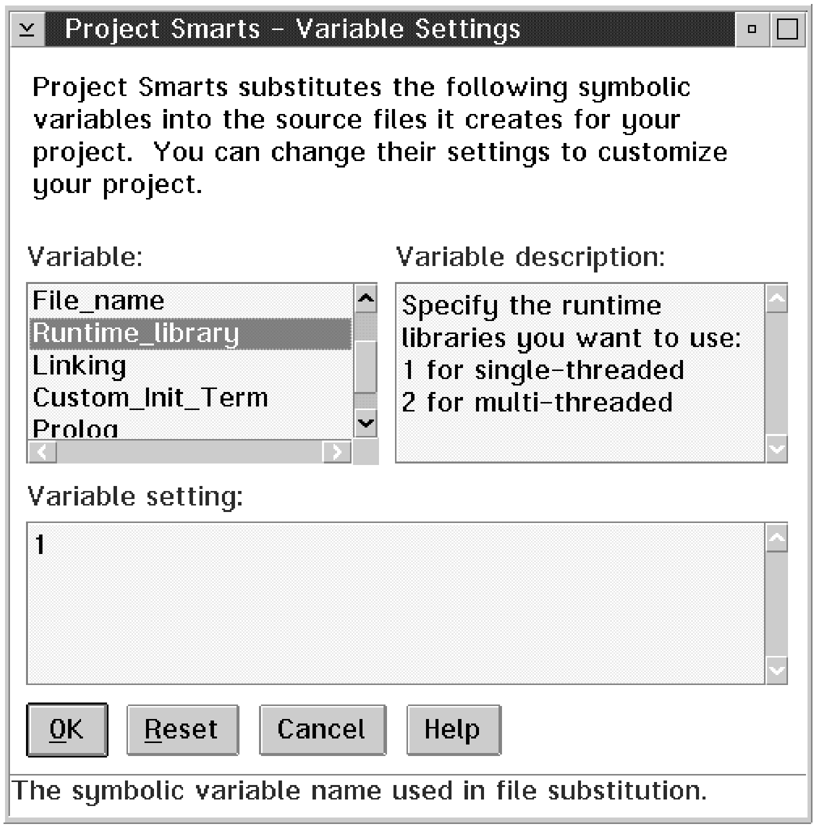
The name of the project to be created.

Figure 43. VisualAge C++ Project Smarts catalog

To create one of the Project Smarts projects, select its name and then click on the **Create** push button. Because the creation is controlled by a REXX script that is customized for each project template, the installation process may differ slightly between projects. In general, a Project Smarts console window appears where you can see the installation progressing. During the installation, a **Location** window appears asking you for the directory to install the source files in, and the folder to create the project in.

In most cases, a **Variable Settings** window, as shown in Figure 44, will also appear. Use it to customize the generated project by setting values for certain substitution variables, such as the source-file prolog text, current date, and user name. As you select each variable name in the **Variable** list box, the **Variable description** list box is updated with a short description about the variable. Verify the defaults and set values of your own by editing the **Variable setting** field. When you have verified all the variable settings, select the **OK** push button to continue with the installation.

The values that you set are substituted when the application is installed. They are also saved for the next time you install the same Project Smarts application.



Project Smarts - Variable Settings

Project Smarts substitutes the following symbolic variables into the source files it creates for your project. You can change their settings to customize your project.

Variable:

- File_name
- Runtime_library
- Linking
- Custom_Init_Term
- Prolog

Variable description:

Specify the runtime libraries you want to use:
1 for single-threaded
2 for multi-threaded

Variable setting:

1

The symbolic variable name used in file substitution.

Figure 44. Project Smarts Variable Settings window

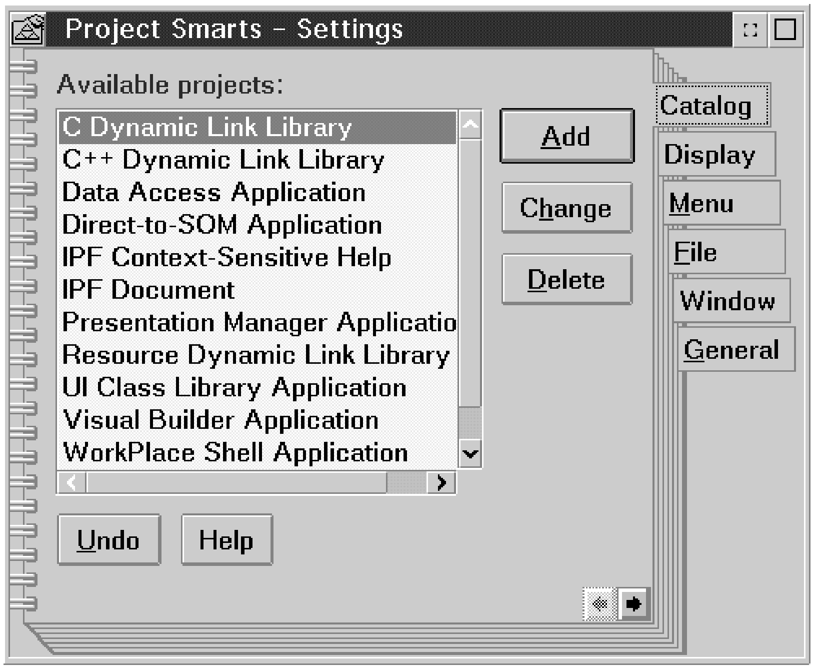
When the installation is complete, the created project appears on your desktop or in the folder you specified. You are now ready to begin working on developing the specifics of your application.

Adding Your Own Project Smarts Application

If you have code skeletons of your own, you can add them to the VisualAge C++ Project Smarts catalog, or you can create your own catalog. You can create your own Project Smarts catalog by dragging the **Project Smarts** template from the **Templates** folder on your desktop, or by selecting **Create another** from the pop-up menu of the VisualAge C++ Project Smarts catalog. REXX utilities for creating Project Smarts catalogs are also available, and are described in “Writing An Installation Script” on page 118 .

You will need to specify a REXX installation script for every Project Smarts application you add. You can use the default install script, IWFSMART.CMD, or write your own customized script. Project Smarts REXX programming interfaces are available to make writing the script an easy task. They provide the mechanisms for displaying and updating the **Project Smarts installation console**, as well as **Variable Settings** and **Project Location** dialogs. Refer to “Writing An Installation Script” on page 118 for instructions on how to use the Project Smarts utilities to write your own customized script.

To add your own Project Smarts application, open the **Settings** notebook of the Project Smarts catalog. The **Catalog** page shows a list of Project Smarts projects.

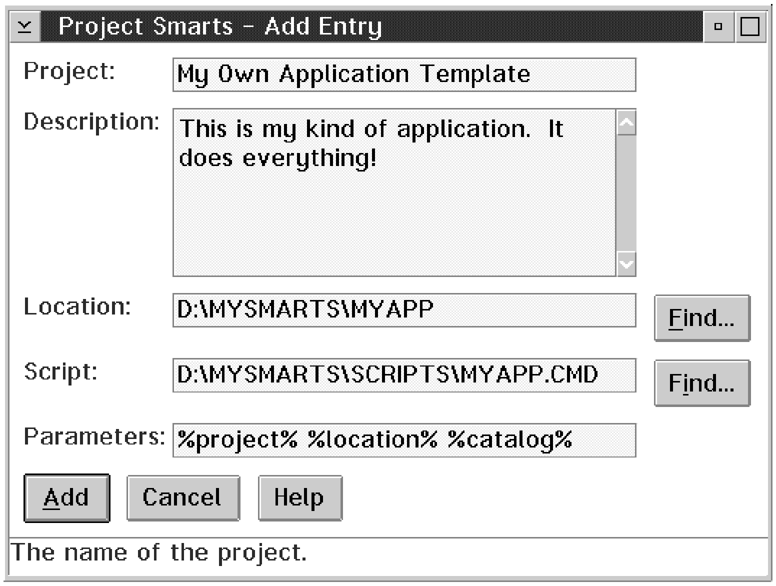


C Dynamic Link Library
C++ Dynamic Link Library
Data Access Application
Direct-to-SOM Application
IPF Context-Sensitive Help
IPF Document
Presentation Manager Applicatio
Resource Dynamic Link Library
UI Class Library Application
Visual Builder Application
WorkPlace Shell Application

Figure 45. VisualAge C++ Project Smarts **Settings** *notebook*

Select the **Add...** push button. The Project Smarts **Catalog Entry** window appears where you can enter all the information for your new application. The **Catalog Entry** window is shown in Figure 46 on page 116.

An easy way to create a Project Smarts application from an existing project or folder is to drag the project or folder into the Project Smarts catalog. When you drop it, the **Catalog entry** window is displayed. The fields are updated with information from the dropped object.



Project Smarts - Add Entry

Project: My Own Application Template

Description: This is my kind of application. It does everything!

Location: D:\MYSMARTS\MYAPP

Script: D:\MYSMARTS\SCRIPTS\MYAPP.CMD

Parameters: %project% %location% %catalog%

The name of the project.

Figure 46. VisualAge C++ Project Smarts Catalog Entry window

Fill in the fields in the **Catalog Entry** window to describe your new Project Smarts application:

Project
: Enter the project's title. Your application is listed under this title in the catalog.

Description
: Enter a description of your application. This description also appears in the catalog.

Source
: Enter the location of the source file skeletons for your application. The location can be a directory or an existing WorkFrame project. The install script copies the source from this location.

Your application's source files can contain substitution variables whose default values are set by your installation script (the default installation script assumes that there are no substitution variables). Substitution variables imbedded in the source files must be of the form `%VARIABLE%`. A REXX Project Smarts utility substitutes the variables in the source files with their specified values in the script when the files are copied over to the new location during the install.

Script

Enter the name of the Project Smarts REXX install script that creates your Project Smarts application. You can enter a fully qualified path name or a command file found on the PATH.

You can specify your own customized script, or use the default script, IWFSMART.CMD. See "Writing An Installation Script" on page 118 for more information on how to write a custom installation script.

Parameters

List any parameters that your install script may require. Project Smarts will pass these parameters to your script when it invokes it during the installation. Five substitution variables are also available to be passed as parameters to your script:

`%project%`

The name of the project, as specified in the **Project** field of the **Catalog Entry** window when the Project Smarts application was added to the catalog. Your install script can use this name as the default name of the project it creates. See "Project Creation Utilities" on page 129 for more information on project creation utilities.

`%location%`

The directory where the base source files are located, as specified in the **Source** field of the **Catalog Entry** window. Your install script can use this information to copy the files from the source directory, or do some other file processing. See "File Substitution Utility" on page 129 for more information on file copying and substitution utilities

`%description%`

The description of the application, as specified in the **Description** field of the **Catalog Entry** window. You can use this variable in the prolog of the source files.

`%catalog%`

The fully-qualified path name of the Project Smarts catalog. The path name of the catalog is a required parameter for the **IwfSaveVariables** and **IwfRestoreVariables** utilities because the variable settings are stored with the Project Smarts catalog. See "Variable Settings Utilities" on page 127 for more information about the Variable Settings utilities.

`%script%`

The name of the installation script, as specified in the **Script** field of the **Catalog Entry** window.

Note: If you specified the default Project Smarts installation script, IWFSMART.CMD, on the **Script** field, you must enter these required parameters:

```
%project% %location%
```

When you click on the **OK** push button, your new Project Smarts application is added to the catalog.

Writing An Installation Script

Every Project Smarts application needs an associated installation script. Because the install script is a REXX program, you can easily write highly customized scripts to install any kind of application. As an alternative, you can use the generic, default Project Smarts script IWFSMART.CMD, to install your application. No customization is done if you use the default install script to install your Project Smarts application. This script will ask for the following information during the installation:

Project
The name to give the created project.

Directory
The directory where the source files are to be copied to.

Folder
The Workplace Shell folder where the project is to be created.

The default install script then creates a project based on this information.

If you created your Project Smarts application by dragging a WorkFrame project into the Project Smarts catalog, the default script creates a project from the source files in the project source directories. The created project inherits its environment from the VisualAge C++ Default Project. See “The Default Project” on page 36 for more information about default projects.

The full power of REXX is available to customize your install script any way you want, including the ability to:

- Display a window to gather information from the end user. Any information an install script needs from the end user for customization or special processing can be obtained using substitution variables. Project Smarts programming interfaces provide a generic window, called **Variable Settings**, where end-users can set the values for the variables during the installation. See “File Substitution Utility” on page 129 for more information on file copying and substitution utilities, and “Variable Settings Utilities” on page 127 for more information on the **Variable Settings** window utilities.

- Manipulate the contents of the source files in different ways, such as changing titles, removing or adding functionality, depending on the preferences of the end user. This can be accomplished by expressing some of the source file contents as substitution variables.
- Install files based on options selected by the end user. For instance, a project might provide a number of selectable features, like context-sensitive help and SOM support. Files of the selected type can then be created for these features.
- Set up projects based on local coding standards.

By writing your own installation script, you have unlimited customization opportunities as the application provider, and as the end-user.

The next few sections of this chapter assume a working knowledge of the REXX command language. Refer to the *OS/2 Procedures Language 2/REXX* online document to learn more about REXX. To view it, type `view rexx` from an OS/2 command line.

Project Smarts Installation Script Utilities

A number of Project Smarts REXX utilities are available to help you write an installation script:

The Initialization Utilities
The Project Smarts utilities need to be initialized before they can be used. You need to call the initialization utilities to do this.

Project Smarts Catalog utilities
Use these utilities to create and update new Project Smarts catalogs on a system.

The Progress Console utilities
The **Progress Console** communicates the status of the installation process to the end-user. It has three areas:

1. A progress indicator
2. A status area
3. A log area that shows all the status lines displayed so far.

All REXX output from REXX `SAY` and `TRACE` commands are displayed in the console status and log areas. Interfaces that control updates to the progress indicator are also provided.

Location dialog utility
The **Location** dialog queries the end-user for the title, target directory and folder to place the created project in.

Variable Settings window utilities

The end user uses this window to specify the values for any substitution variables used by the script. It lists the variables with their associated descriptions and default values. The end user can change the value of any variable (for example, a `%USER_NAME%` variable) to customize the created project.

The script can use these variables any way it needs to, including imbedding them in source files to customize the application, or performing further processing. The *File substitution* utility transforms any imbedded substitution variables in the source files before it copies them to the target directory.

The values the end user enters can be saved for the next time the application is installed.

The File Substution utility

This utility copies the base files specified in the **Source** entry field in the **Catalog Entry** window, and transforms their contents according to the variable settings received from the Variable Settings window. This enables you to customize the source file contents using substitution variables.

Project Creation utilities

These utilities create and set up a WorkFrame project from a set of files in a directory, or another existing WorkFrame project. You can also copy and then customize existing projects with these utilities.

Project Setup utilities

These utilities configure a WorkFrame project's **Tools setup**. You can use these utilities to add and delete actions, environment variables, and types.

Note: Your installation script does not need to use any of these utilities. You can choose to display your own customized windows instead. The utilities are provided to make writing installation scripts an easy task.

A sample installation script is shown in “Project Smarts Sample Installation Script” on page 139.

Initialization

To use the Project Smarts utilities in your script, you first need a few statements to initialize them:

- Declare a global stem variable. It is a structure that you initialize and pass to the Project Smarts utilities:

  ```
  /* Initialize - use a global stem variable.       */
  /* This is required to use any of the Project Smarts    */
  /* REXX programming interfaces.                   */
  stem = "stem"
  ```

- You might want to initialize some return code constants. All the Project Smarts REXX utilities return 0 for sucess, and 95 for cancel:

  ```
  RC_OK     = 0
  RC_CANCEL = 95
  ```

- If you are using the Project Smarts **Progress Console** window, one of the first things you should do in your script is call the **IwfOpenConsole** utility to open it. Call it with the stem global variable:

  ```
  /* Open the installation Console. */
  rc = IwfOpenConsole(stem);
  ```

- You need to load the REXX utility functions. They are used by the Project Smarts utilities. RexxUtil functions provide OS/2 system commands, user or text screen input and output, and OS/2 INI file input and output. For more information on RexxUtil functions, refer to the *OS/2 Procedures Language/2 REXX* document (type view rexx from an OS/2 command prompt).

  ```
  /* Load the REXX utility functions.   */
  rc = RxFuncAdd('SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs');
  rc = SysLoadFuncs();
  ```

- The functions for all the Project Smarts utilities are loaded automatically for you when your script is started, with the exception of the Project Setup utilities. You must load these utilities explicitly if you intend to use them in your script.

  ```
  /* Load the Project Setup functions */
  rc = RxFuncAdd('IwfEnvPrfLoadFuncs','IWFPAPI','IwfRxEnvPrfLoadFuncs');
  rc = IwfEnvPrfLoadFuncs();
  ```

  Note: At the end of your script, issue this call to unload the Project Smarts Project Setup utilities:

  ```
  rc = IwfEnvPrfDropFuncs()
  ```

- If your script uses one or more Project Setup utilities repeatedly, you can optimize performance by optionally calling an initialization function before calling any Project Setup utilities, and a deinitialization function afterwards, so that the initialization and deinitialization processing is done only once for the entire script. If you do not call these utilities, the initialization and deinitialization processing is performed each time a Project Setup utility is called.

```
/* Initialize the Project Setup utilities */
rc = IwfInitEnvPrfAPIs();
         :
/* Calls to Project Setup utilities */
         :
/* De-initialize the Project Setup utilities */
rc = IwfTermEnvPrfAPIs();
```

- The Project Smarts catalog can pass arguments to your script. They are passed in the order that they are listed in the **Catalog Entry** window. For example, if your application expects three arguments, then you need to extract them:

```
/* Extract the passed arguments (substitution variables).
 * This script expects 3 substitution variables:
 * Project name, description, and location.  You can use
 * your own internal variable names.
*/
if (Arg() <> 3) Then
   call Abort("Error in parameter list.");
Parse Arg Proj,Desc,Locn;
```

Catalog Utilities

Use these utilities to create and tailor your own Project Smarts catalog.

The **IwfOpenCatalog**, **IwfCloseCatalog**, and **IwfUpdateCatalog** utilities all return the catalog's object handle in the *stem.hCatalog* variable.

- **SysCreateObject**

  Use the RexxUtil function, **SysCreateObject**, to create a new Project Smarts catalog object.

```
rc = SysCreateObject
        (objectClass,  /* Workplace Shell object class name */
         title,        /* The object title */
         location,     /* Object location e.g. <WP_DESKTOP> */
         setupString,  /* Project Smarts object setup string */
         options);     /* What to do if object exists */
```

objectClass Specify `'IWFQuickStart'` to create a Project Smarts catalog.

title The Project Smarts catalog title.

location The object location. You can specify this parameter as either an object ID, as in `'<WP_DESKTOP>'`, or a file-system path, as in `'C:\DESKTOP\MYFOLDER'`.

setupString The Project Smarts catalog setup string. You can use the following key names in your setup string:

`ICONFILE` The icon you want to associate with your Project Smarts catalog. If you do not specify this key name, the default Project Smarts icon is used.

`OBJECTID` Use this key name if you want to assign a persistent object ID to your new Project Smarts catalog.

For example, your setup string could look like this:

```
stpString = 'ICONFILE=mysmarts.ico;OBJECTID=<MY_SMARTS_APPS>;'
```

options This parameter is optional. If you want, you can specify the action to be taken if the object already exists. The allowed options are:

`'fail'` Abort the operation.

`'replace'` Delete the existing object, and create a new one.

`'update'` Update the settings of the existing object.

rc The return code. **SysCreateObject** returns 1 (TRUE) if the object was successfully created, and 0 (FALSE) if the object was not created.

For example, the following statement creates a Project Smarts catalog called **My Applications** on the Desktop. The object uses the default icon, and is assigned the object ID <QS_MYAPPS>:

```
rc = SysCreateObject ('IWFQuickStart',
                      'My Project Smarts Applications',
                      '<WP_DESKTOP>',
                      'OBJECTID=<QS_MYAPPS>',
                      'Replace');
```

For more information about the **SysCreateObject** RexxUtil utility, refer to the *OS/2 Procedures Language 2/REXX* online document (type `view rexx` from an OS/2 command line).

- **IwfOpenCatalog**

  This utility opens the Project Smarts catalog named in the `stem` variable. You must open the catalog using **IwfOpenCatalog** before you update it, and close it using **IwfCloseCatalog** after you are finished. You can name a catalog by its object ID, as in <`CSetQSCatalog`> for the VisualAge C++ Project Smarts catalog, or by a file-system path, as in `C:\Desktop\Project Smarts`.

```
/* Initialize the stem variable with the name of the catalog to open */
stem.pszCatalog = "<MySmartsCatalog>"
rc = IwfOpenCatalog(stem);
if (rc <> 0) then
   call Abort("Unable to open the Project Smarts catalog");
```

- **IwfUpdateCatalog**

  This utility adds a catalog entry to the named catalog. Initialize the `stem` variable to contain the description of the Project Smarts application you are adding. The `stem` fields for this utility correspond to the entry fields in the **Catalog Entry** window.

```
/* Initialize stem variable for the catalog entry to be added */
stem.pszCatalog           = "<MySmartsCatalog>"
stem.pszName              = "My Enterprise Application"
stem.pszDescription       = "My kind of application.\nIt does everything!"
stem.pszSourceLocation    = "D:\MYAPP"
stem.pszInstallScript     = "D:\MYAPP\MYSCRIPT.CMD"
stem.pszInstallParameters = "%project% %location%"

rc = IwfUpdateCatalog(stem);
if (rc <> 0) then
   call Abort("Unable to update the catalog");
```

- **IwfCloseCatalog**

  Use this utility to close the catalog after you have finished making updates to it.

```
stem.pszCatalog = "<MySmartsCatalog>"
rc = IwfCloseCatalog(stem);
if (rc <> 0) then
   call Abort("Unable to close the catalog");
```

Progress Console Utilities

These utilities open, update, and close the **Progress Console** window. They all return the window handle of the **Progress Console** window in the *stem.hwndConsole* variable.

- **IwfOpenConsole**

  This utility opens the Project Smarts **Progress Console**. Call it with the `stem` variable.

```
IwfOpenConsole(stem);
```

- **IwfCloseConsole**

  This utility closes the Project Smarts **Progress Console**. The end user can also close the it by selecting **Close** from the system menu.

  Call this utility by passing it the stem variable:

```
IwfCloseConsole(stem);
```

- **IwfUpdateConsoleProgress**

  Use this utility to update the console progress indicator. Set the stem variable to a percentage figure before calling the utility:

  ```
  stem.usPercent = 10   /* Set indicator to 10% completion */

  rc = IwfUpdateConsoleProgress(stem);
  ```

- **IwfUpdateConsoleStatus**

  Use this utility to update the console status area. Set the stem variable to a text string before passing it:

  ```
  stem.pszStatusText = "Initializing..."   /* Set the status text */

  rc = IwfUpdateConsoleProgress(stem);
  ```

- All SAY and TRACE output is automatically displayed in the log message area.

Location Dialog Utility

- **IwfQueryLocation**

  This utility displays the **Location** dialog that queries the location where the project files are to be copied. Initialize the stem variable to specify the initial values for the fields in the dialog. For example:

  ```
  /* Set the default values for the Location dialog, and display it  */
  stem.pszTargetProject   = Proj      /* Title as shown in the catalog */
  stem.pszTargetDirectory = "C:\TMP"  /* Default location */
  stem.pszTargetFolder    = "Desktop" /* Create project on Desktop */
  do until (rc = RC_OK)
     rc = IwfQueryLocation(stem);
     if (rc  = RC_CANCEL) then call Cancel;
     if (rc <> RC_OK)     then call Abort("Error querying target information.");
  end
  ```

  The **IwfQueryLocation** utility updates its three parameters with the values the end-user set in the **Location** dialog.

Variable Settings Utilities

- **IwfQueryVariables**

  This utility displays the **Variable Settings** window the end user uses to verify substitution variable values. Initialize the stem variable to contain the substitution variables and their default values before calling this utility.

  For example, consider a Project Smarts application with two substitution variables, `%USER_NAME%` and `%COMPANY%`:

```
stem.usVariableCount = 2                             /* Only two variables */

stem.pszVariableName.1        = "%USER_NAME%"         /* Variable name */
stem.pszVariableDescription.1 = "Your name"           /* Description */
stem.pszVariableValue.1       = "Type your name here" /* Default value */

stem.pszVariableName.2        = "%COMPANY%"
stem.pszVariableDescription.2 = "Your company name"
stem.pszVariableValue.2       = "Type your company name here"

do until (rc = RC_OK)
   rc = IwfQueryVariables(stem);
   if (rc  = RC_CANCEL) then call Cancel;
   if (rc <> RC_OK)     then call Abort("Error querying variable settings.");
end
```

  Note that the stem variable fields have an index that corresponds with the variable number.

  You can place C escape sequences, such as the `\n` (newline) character, in the description and value fields.

  The new variable values set by the end-user are returned in the parameters used by the **IwfQueryVariables** utility. It also returns the window handle of the project console in the *stem.hwndConsole* variable.

- **IwfSaveVariables**

  Call this utility after calling **IwfQueryVariables** to save the current variable settings to disk so that it is available the next time the script is run. This utility takes the same parameters as the **IwfQueryVariables** utility, plus two more:

  stem.pszCatalog The Project Smarts catalog path name. The variable settings are stored with the catalog file. This path name can be passed in a substitution variable as an argument to the installation script. See “Adding Your Own Project Smarts Application” on page 114 for more information about the substitution variables you can use to pass arguments to your installation script.

  stem.pszApplication An application key name used to store the data with. Ensure that this key name is unique, and that there is little chance that the key name might already exist or be reused within the catalog. The data may be corrupted otherwise.

```
/* Specify the catalog and application name */
stem.pszCatalog = CatName             /* Catalog name from argument */
stem.pszApplication = "myApp"         /* Any application key name */

rc = IwfSaveVariables(stem);
if (rc <> RC_OK)     then call Abort("Error saving variable settings.");
```

- **IwfRestoreVariables**

  Call this utility just before calling **IwfQueryVariables** to restore the saved settings from disk so that they are shown in the **Variable Settings** window as defaults. This utility takes the same parameters as the **IwfQueryVariables** utility, plus two more.

```
/* Variable values have already been initialized to the script defaults */
/* Now just specify the catalog and application name.                   */
stem.pszCatalog = 'D:\Desktop\Project Smarts' /* Project Smarts catalog */
stem.pszApplication = "myApp"       /* Application key name given    */
                                    /* on last IwfSaveVariables call */

rc = IwfRestoreVariables(stem);
if (rc <> RC_OK)     then call Abort("Error restoring variable settings.");
```

The restored variable values are returned in the variables used by the **IwfQueryVariables** utility.

File Substitution Utility

- **IwfCopyWithSubstitution**

This utility copies all the source files to the target directory (as queried by the **Location** dialog), substituting any substitution variables in the source with their set values.

Call this utility with the *stem* variable. Since this utility requires the same parameters as the **IwfQueryVariables** utility, the variables used for the substitution are those specified by that utility.

The parameters are the same as those returned by the **IwfQueryVariables** utility, plus two more parameters returned by the **IwfQueryLocation** utility.

```
/* These parameters have already been returned by the IwfQueryLocation */
/* and IwfQueryVariables utilities:                                      */
/*   stem.pszSourceFileMask       (Path or file mask to copy from)      */
/*   stem.pszTargetDirectory      (Directory to copy to)                */
/*   stem.usVariableCount         (Number of variables used by script) */
/*   stem.pszVariableName.1                                             */
/*   stem.pszVariableDescription.1                                      */
/*   stem.pszVariableValue.1                                            */
/*          :                                                           */
/*   stem.pszVariableName.n       (Where n is stem.usVariableCount)     */
/*   stem.pszVariableDescription.n                                      */
/*   stem.pszVariableValue.n                                            */

rc = IwfCopyWithSubstitution(stem);

if (rc <> RC_OK) then call Abort("Error performing copy and substitution.");
```

Project Creation Utilities

- **SysCreateObject**

You can use the RexxUtil function, **SysCreateObject**, to create a new WorkFrame project.

```
rc = SysCreateObject (objectClass,  /* Workplace Shell object class name */
                      title,        /* The object title */
                      location,     /* Object location. e.g. <WP_DESKTOP> */
                      setupString,  /* Project Smarts object setup string */
                      options);     /* Action if object already exists */
```

objectClass Specify `'IWFProject'` for a WorkFrame project.

title The project title. You may want to specify the project title that was queried by the **Location** dialog. The **IwfQueryLocation** utility returns this value in the `stem.pszTargetProject` variable.

Project Smarts

location The object location. Specify the location obtained via the **IwfQueryLocation** utility, and returned in the `stem.pszTargetFolder` variable. Its location can be specified as an object ID, as in `'<WP_DESKTOP>'`, or as a file system path, as in `'C:\DESKTOP\MYFOLDER'`.

setupString The Project Smarts catalog setup string. You can use the following key names in your setup string (these key names correspond to the fields of a project's **Settings** notebook):

- `TARGETNAME` The project's target file name. If you do not specify this key name, it defaults to `'target.exe'`.
- `MAKEFILENAME` The project's make file name. If you do not specify this key name, it defaults to `'makefile'`.
- `RUNPARAMETERS` The parameters to use when running the project's target program.
- `RUNPROMPT` Prompt for parameters before running the project's target program. Specify `'TRUE'` to prompt, `'FALSE'` for no prompt.
- `RUNMONITORED` Run the project's target program in the **Monitor** window. Specify `'TRUE'` for yes; `'FALSE'` for no. If you do not specify this key name, the default is not to run the target program in the **Monitor**.
- `PAMORDER` An ordered, newline-delimited list of PAMs to use. If you do not specify this key name, the default is the basic PAM, IWFBPAM.

PAMLOCATION:name Where name is the name of the PAM DLL, with no path or extension, in uppercase. If your project uses the basic PAM, the correct name for this key is PAMLOCATION:IWFBPAM. The data is a list of source directories for the project, separated by a newline character. For example, 'C:\PROJECT\HEADERS\nC:\PROJECT\SOURCE' You can specify the target directory obtained via the **IwfQueryLocation** utility as the value of this key name. If you specify this key name, you must also specify the PAMDEFAULT:name key name. If you do not specify this key name, the default is a subdirectory of the TMP environment variable directory.

PAMDEFAULT:name Where name is the name of the PAM DLL, with no path or extension, in uppercase. If your project uses the basic PAM, then the correct name for this key is PAMDEFAULT:IWFBPAM. The data is the working directory path for the project, and must be a path specified by the PAMLOCATION:name, for example 'C:\PROJECT\SOURCE'. You must specify this key name if you specify the PAMLOCATION:NAME key name.

FILTER A list of types or file masks to filter the project container. The default is no filter.

INHERITLIST The list of projects to inherit from, separated by a newline character. Each project name can be an file system path name, as in 'D:\Desktop\Some Project', or a persistent object ID, in the form '<OBJECT_ID>'. The default is not to inherit from any projects.

To add projects to any projects already in the inheritance list of a project, you can use the INHERITLIST+ key name instead. To remove any inheritance already defined in a project, specify an empty string for the value of the INHERITLIST key name.

`TITLE` Use this key name to specify the project's name or title.

`OBJECTID` Use this key name to assign a persistent object ID to your new project. The object ID should be in the format `<OBJECT_ID>`.

`MONAUTOSCROLL` Set the project's **Monitor** to scroll automatically while displaying output from actions. Specify `'TRUE'` for automatic scrolling, `'FALSE'` for none. The default is `'TRUE'`.

`MONAUTOERASE` Set the project's **Monitor** to erase its contents before displaying the output from an action. Specify `'TRUE'` to erase monitor contents, `'FALSE'` otherwise. The default is `'TRUE'`.

`MONDISPLAYONSTART` Set the project's **Monitor** to automatically show itself when a monitored action is started. Specify `'TRUE'` to have the monitor show itself when a monitored action is started, `'FALSE'` otherwise. The default is `'TRUE'`.

`MONHIDEONCOMPLETION` Set the project's **Monitor** to hide itself on the successful completion of an action. Specify `'TRUE'` to hide the **Monitor** upon successful completion, `'FALSE'` otherwise. The default is `'FALSE'`.

options This optional parameter specifies the action to be taken if the object already exists. The allowed options are:

`'fail'` Abort the operation.

`'replace'` Delete the existing object and create a new one.

`'update'` Update the settings of the existing object.

rc The return code. **SysCreateObject** returns 1 (TRUE) if the object was successfully created, and 0 (FALSE) if the object was not created.

For example, the following statement creates a WorkFrame project called **My Project** on the Desktop. The object uses the basic PAM, has the source directory D:\MYSOURCE, a target, PROGRAM.EXE, a make file, PROGRAM.MAK, inherits from a project called Base Project on the Desktop, and is assigned the object ID <MY_PRJ>:

```
Str = 'PAMLOCATION:IWFBPAM=d:\mysource; ,
       PAMDEFAULT:IWFBPAM=d:\mysource; ,
       TARGETNAME=program.exe;MAKEFILENAME=program.mak; ,
       USEPROJECTS=d:\desktop\base project; ,
       OBJECTID=<MY_PRJ>',

rc = SysCreateObject ('IWFProject',   /* Project object class name */
                      'My Project',   /* Project title */
                      '<WP_DESKTOP>', /* Create project on Desktop */
                      Str,            /* Setup string */
                      'Replace');     /* Replace if exists */
```

- **IwfCreateProjectFromFiles**

  A simpler alternative to **SysCreateObject**, this utility creates a project, and then sets the newly created project's source directory to the directory where the files were copied to. You can specify a directory, file mask, or project path for the *stem.pszTargetFile* parameter. If a directory or file mask is specified, the utility searches the copied files to find an executable and make file, and sets these as the project's target and make file name. If a project is specified, the project's target, make file, and other settings are copied to the new project.

  This utility takes the same parameters as the **IwfQueryLocation**, plus one more:

```
/* These parameters were set by the IwfQueryLoction utility:  */
/*        stem.pszTargetProject                               */
/*        stem.pszTargetDirectory                             */
/*        stem.pszTargetFolder                                */
/*                                                            */
/* Specify the project or directory specified in the "Source" */
/* field of the Catalog Entry dialog.  The "Src" variable     */
/* was passed as an argument to this script.                  */
stem.pszTargetFile = Src;

rc = IwfCreateProjectFromFiles(stem);
```

- **CreateProjectFromProject**

  This utility copies an existing project, and modifies it with the values you specify in the argument setup string. (See **SysCreateObject** above for the valid key name-value pairs). The advantage of using this utility over **SysCreateObject** and **IwfCreateProjectFromFiles** is that action options are copied, along with other project settings. You can create a model project for your Project Smarts application with the appropriate action options already set. Then the installation script can use this utility to copy the model project and customize certain settings like the project source directory, target name, and make file name.

  Initialize the *stem* variable with the following parameters:

  pszSourceProject The project to copy, specified as an object ID (of the form `<OBJECT_ID>`), or a fully-qualified path name (for example, `D:\DESKTOP\My Project`).

  pszTargetProject The name of the project to be created (for example, `New C++ Project`). You can use the project name returned by the **IwfQueryLocation** utility.

  pszTargetProjectSetup An optional setup string to update the newly created project with. See the discussion on creating an object with the **SysCreateObject** REXX utility for more information about project setup strings.

  pszTargetDirectory If you do not specify a setup string in the *pszTargetProjectSetup* parameter, you must specify the path name for the new project's source directory in this parameter. If you specify a setup string, this parameter is ignored.

  pszTargetFolder The folder in which to create the new project, specified as a fully-qualified path name, for example `D:\DESKTOP\My Projects`. You can use the target folder path returned by the **IwfQueryLocation** utility.

  For example, these statements create a project by copying a project identified by `<MY_MODEL_PROJECT>` to the Desktop, and setting it with values indicated in the parameter setup string:

```
stem.pszSourceProject = '<MY_MODEL_PROJECT>' /* Can specify as a path */
stem.pszTargetProject = 'My New Project'     /* New project name */
stem.pszTargetProjectSetup = 'PAMLOCATION:IWFBPAM=d:\mysource;,
                              TARGETNAME=program.exe;,
                              MAKEFILENAME=program.mak;' /* Setup string */
stem.pszTargetDirectory = ''                 /* No need if using setup string */
stem.pszTargetFolder = 'D:\DESKTOP'          /* Copy project to Desktop */

rc = IwfCreateProjectFromProject(stem);
```

Project Setup Utilities

- **IwfAddAction**

  Use this utility to add an action to the project. Initialize the stem variable with the following parameters:

  pszProject The fully-qualified path name of the project to which you want to add the action.

  pszActionName The name of the action.

  pszActionClass The class of the action, for example `'Edit'`.

  pszCommand The command to run the action, for example `'EPM.EXE'`.

  pszSrcMask A newline-delimited list of source types or masks for the action, for example:

  ```
  C Source\nC++ Source\nText Files'
  ```

  pszTgtMask A newline-delimited list of target types or masks for the action, if any apply. For example `'Object Files'`.

  pszDllName The name of the action's support DLL, for example `'IWFOPT'`.

  pszDllEntryName The support DLL entrypoint to use, for example `'Edit'`.

  pszHelpCmd The help command for the action-specific help, if any. For example, `'VIEW.EXE'`.

  pszHelpTopic The help topic for the action-specific help, if any. For example, `'EPM.INF'`.

  pszucActionScope The scope of the action, `'P'` for project-scoped, `'F'` for file-scoped, and `'B'` for both. The default is file-scoped.

  pszucRunMode The type of session the action should run in, `'F'` for full-screen, `'W'` for windowed, `'M'` for monitored, and `'D'` for default.

  pszucAccelKey The accelerator key for the action, that is used with the **Ctrl+Shift** keystroke, for example `'E'`. Ensure that no other action in the project uses the same accelerator key, or you will get undefined behavior.

  pszPam If the action is project-scoped, specify the name of the PAM responsible for invoking the action, for example `'IWFBPAM'`.

  pszfPrjMenu Specify `'T'` if you want the action added to the project menus, or `'F'` otherwise.

  pszfOptMenu Specify `'T'` to add the action to the project **Options** pulldown menu, or `'F'` otherwise.

  pszfTBMenu If the action is project-scoped, specify `'T'` to add the action to the project toolbar, `'F'` otherwise. Specify `'F'`, the default, if the action is file-scoped.

There are safe defaults for all of the parameters except for *pszProject*, *pszActionName*, *pszActionClass*, *pszCommand*, and *pszSrcMask*, which are required.

The following example adds the Edit::EPM action to the project D:\DESKTOP\MY PROJECT:

```
/* Add an 'Edit::EPM' action to the project D:\DESKTOP\MY PROJECT */
stem.pszProject = 'D:\DESKTOP\MY PROJECT'
stem.pszActionName = 'EPM'
stem.pszActionClass = 'Edit'
stem.pszCommand = 'EPM.EXE'
stem.pszSrcMask = 'Editable'
stem.pszTgtMask = ''
stem.pszDllName = 'IWFOPT'
stem.pszDllEntryName = 'Edit'
stem.pszHelpCmd = ''
stem.pszHelpTopic = ''
stem.pszucActionScope = 'F'         /* File scoped */
stem.pszucRunMode = 'W'             /* Windowed session */
stem.pszucAccelKey = 'E'
stem.pszPam = 'IWFBPAM'
stem.pszfPrjMenu = 'T'              /* Add to menus */
stem.pszfOptMenu = 'T'              /* Add to Options menu */
stem.pszfTBMenu = 'F'               /* Do not add to project toolbar */

rc = IwfAddAction(stem)
```

- **IwfCopyAction**

  Use this utility to copy an action from one project to another. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Copy the Edit::EPM action from D:\DESKTOP\MODEL to
D:\DESKTOP\MY PROJECT */
stem.pszProject = 'D:\DESKTOP\MODEL'  /* Source project path name */
stem.pszActionClass = 'Edit'                /* Class of the action to copy */
stem.pszActionName = 'EPM'                  /* Name of action to copy */
stem.pszDestProject = 'D:\DESKTOP\MY PROJECT' /* Destination project */
stem.pszDestActionClass = 'Edit'            /* Destination action class */
stem.pszDestActionName = 'Enhanced Editor'  /* Destination action name */

rc = IwfCopyAction(stem);
```

- **IwfDestroyAction**

  Use this utility to delete an action from a project. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Delete the Edit::EPM action from D:\DESKTOP\MY PROJECT */
stem.pszProject = 'D:\DESKTOP\MY PROJECT' /* Project path name */
stem.pszActionClass = 'Edit'                /* Class of action to delete */
stem.pszActionName = 'EPM'                  /* Name of action to delete */
stem.pszDeleteAny = 'TRUE '   /* Set to 'TRUE' to delete the action      */
                              /* in the inherited project, if the action */
                              /* is inherited.                           */
rc = IwfDestroyAction(stem)
```

- **IwfAddVariable**

  Use this utility to add an environment variable to a project. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Add the HELP = D:\SOURCE\HELP;%HELP% environment variable */
stem.pszProject = 'D:\DESKTOP\MY PROJECT' /* Project path name */
stem.pszName = 'HELP'                           /* Variable name */
stem.pszValue = 'D:\SOURCE\HELP;%HELP%'   /* Variable value */

rc = IwfAddVariable(stem);
```

- **IwfDestroyVariable**

  Use this utility to delete an environment variable from a project. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Delete the HELP = D:\SOURCE\HELP;%HELP% environment variable */
stem.pszProject = 'D:\DESKTOP\MY PROJECT' /* Project path name */
stem.pszName = 'HELP'                           /* Variable name */

rc = IwfDestroyVariable(stem)
```

- **IwfAddType**

  Use this utility to add a type to a project. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Define the 'C Source' type in my project */
stem.pszProject = 'D:\DESKTOP\MY PROJECT' /* Project path name */
stem.pszName = 'C Source'          /* Type name */
stem.pszClass = 'FileMask'         /* The type's class */
stem.pszValue = '*.c\n*.h'         /* The list of file masks or filters */

rc = IwfAddType(stem);
```

- **IwfDestroyType**

  Use this utility to delete a type from a project. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Delete the 'C Source' type from my project */
stem.pszProject = 'D:\DESKTOP\MY PROJECT' /* Project path name */
stem.pszName = 'C Source'                       /* Variable name */

rc = IwfDestroyType(stem)
```

- **IwfRegisterTypeClass**

  Use this utility to register a new type class to the project. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Register a 'State' type class to my project */
stem.pszProject = 'D:\DESKTOP\MY PROJECT' /* Project path name */
stem.pszClass = 'State'                    /* Type class name */
stem.pszDllName = 'STATETYP'               /* Type DLL name */
stem.pszEntryPoint = 'QueryState'          /* Type DLL entrypoint name */

rc = IwfRegisterTypeClass(stem);
```

- **IwfDeregisterTypeClass**

  Use this utility to deregister a new type class from the project. Initialize the *stem* variable to contain the required parameters before calling this utility.

```
/* Deregister the 'State' type class from my project */
stem.pszProject = 'D:\DESKTOP\MY PROJECT' /* Project path name */
stem.pszClass = 'State'                    /* Type class name */
```

Project Smarts Sample Installation Script

Following is the source for the default Project Smarts installation script. It uses many of the Project Smarts install utilities. You can also look at the installation scripts for the Project Smarts applications that come with VisualAge C++. They are located in the \SMARTS\SCRIPTS subdirectory under your VisualAge C++ installation directory.

```
/*REXX*/


/*   IWFSMART.CMD  Project Smarts default install script
 *
 *   (c) Copyright International Business Machines Corporation 1995.
 *   All rights reserved.
 *
 */


/* Initialize - use a global stem variable.  This is required to use */
/* any of the Project Smarts REXX utilities.                    */
stem = "stem"

/* Initialize some return code constants */
RC_OK     = 0
RC_CANCEL = 95


/* Open the installation console. */
rc = IwfOpenConsole(stem);


/* Load the REXX utility functions. This is required if your script */
/* uses any RexxUtil functions, such as SysCreateObject.            */
rc = RxFuncAdd('SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs');
rc = SysLoadFuncs();


/* Extract the passed parameters. This script expects 3 arguments: */
/* Project name, description, and location.                        */
if (Arg() <> 2) Then
   call Abort("Error in parameter list.");
Parse Arg Proj,Src;
```

```
/* Initialize the progress, status, and log via the 'stem' */
/* variable. Use the stem.pszStatusText parameter to write */
/* text to the Status area of the console.                 */
stem.usPercent     = 10
stem.pszStatusText = "Initializing..."
rc = IwfUpdateConsoleProgress(stem);
rc = IwfUpdateConsoleStatus(stem);
SAY "Initializing the Project Smarts install..."

/* Display the Location dialog to query the target */
/* location for the install.                       */
stem.usPercent     = 30
stem.pszStatusText = "Querying the install location..."
rc = IwfUpdateConsoleProgress(stem);
rc = IwfUpdateConsoleStatus(stem);
SAY "Enter the project name, target directory, and ,
     folder for the installation."

/* Set defaults for Location dialog */
env='OS2ENVIRONMENT';            /* Get Default target dir */
Locn = VALUE('tmp',,env)'\myprj'; /* from TMP env variable */
stem.pszTargetProject   = Proj
stem.pszTargetDirectory = Locn
stem.pszTargetFolder    = "Desktop"
do until (rc = RC_OK)
 rc = IwfQueryLocation(stem);
 if (rc = RC_CANCEL) then call Cancel
 if (rc <> RC_OK) then call Abort("Error querying location.")
end

/* Copy the files over to the target location.           */
/* IwfCopyWithSubstitution substitutes any substitution  */
/* variables in the files with their current settings.   */
stem.usPercent     = 60
stem.pszStatusText = "Copying files."
rc = IwfUpdateConsoleProgress(stem);
rc = IwfUpdateConsoleStatus(stem);
SAY "Copying files in "Src" to "stem.pszTargetDirectory"."
stem.pszSourceFileMask = Src;
rc = IwfCopyWithSubstitution(stem);
if (rc <> RC_OK) then call Abort("Error performing copy.");
```

```
/* Create the WorkFrame project.  The IwfCreateProjectFromFiles API */
/* creates a project for the copied files.                          */
stem.usPercent     = 90;
stem.pszStatusText = "Creating the Workframe/2 project.";
rc = IwfUpdateConsoleProgress(stem);
rc = IwfUpdateConsoleStatus(stem);
SAY "The project will be created in "stem.pszTargetFolder;
stem.pszTargetFile = Src;
rc = IwfCreateProjectFromFiles(stem)
if (rc <> RC_OK) then call Abort("Error creating project.");

/* Done!  Update the progress indicator to 100%, and display a */
/* message box informing the user.                             */
stem.usPercent     = 100
stem.pszStatusText = "Installation successful."
rc = IwfUpdateConsoleProgress(stem);
rc = IwfUpdateConsoleStatus(stem);
SAY "Done!"
SAY "The installation is complete and was successful."
rc = RxMessageBox("The WorkFrame project has been created.",,
                  "Done!", "OK", "Information");

/* Perform uninitialization. */
call Done(0);

/* Subroutines */
Cancel:
rcCancel = RxMessageBox("Cancel installation?", , "YesNo", "Query")
if (rcCancel = 6) Then
   call Done(8)
return

Abort:
arg abortMessage
rcAbort = RxMessageBox(abortMessage, , "OK", "Error")
call Done(16)

Done:
arg exitRc
/* Close the installation console. */
rcDone = IwfCloseConsole(stem);
exit(exitRc)
```

Project Smarts

Migrating Old Projects

If you have projects from previous versions of the WorkFrame product, you will need to migrate them to this version. WorkFrame Version 3.0 comes with a migration utility that migrates WorkFrame Version 1.x and Version 2.x projects and profiles to new WorkFrame Version 3.0 projects.

Start the WorkFrame migration utility by double-clicking on the **Migrate WorkFrame Projects** icon in the VisualAge C++ **Tools** folder, or by typing `iwfmig2` on an OS/2 command prompt. The **Project Migration** window appears as shown in Figure 47.

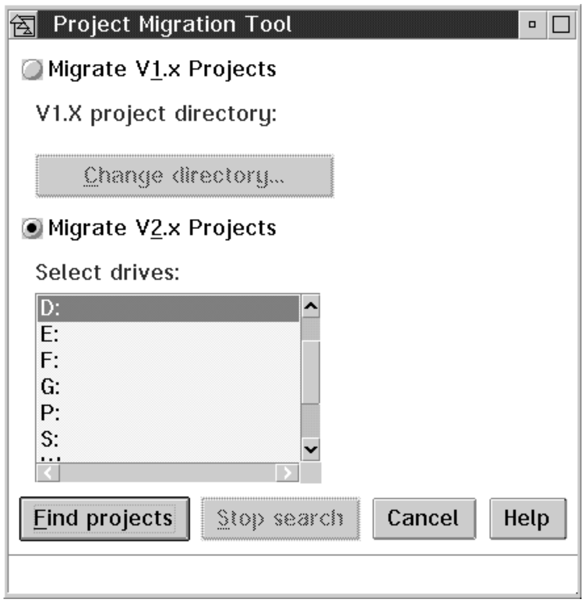


Figure 47. Project Migration Window

Select the **Migrate V1.X Projects** radio button to migrate WorkFrame Version 1.1 or Version 1.0 projects on your system. Select **Migrate V2.X Projects** to migrate your Version 2.1 and Version 2.5 projects.

Migrating Version 2.x Projects

Projects from WorkFrame Version 2.1 or Version 2.5 projects can reside anywhere in your system. As Workplace Shell objects, they are usually stored in folders on the Desktop. The project migration utility can scan the drives on your system to search for Version 2.x projects. If you have projects stored on a LAN drive that you want to migrate, ensure that you are attached to these drives before you start the project migration tool so that they are listed in the **Select drives** list box.

To start the search for Version 2.x projects, select the drives you want scanned then click on the **Find projects** push button. The information area at the bottom of the window informs you of the search progress. After a few seconds, another window appears listing the found projects. Select the projects you want to migrate from the **Projects** list box in the **Select V2.X Projects to Migrate** window. You can also select the **More projects...** push button from this window to manually add other projects to the list.

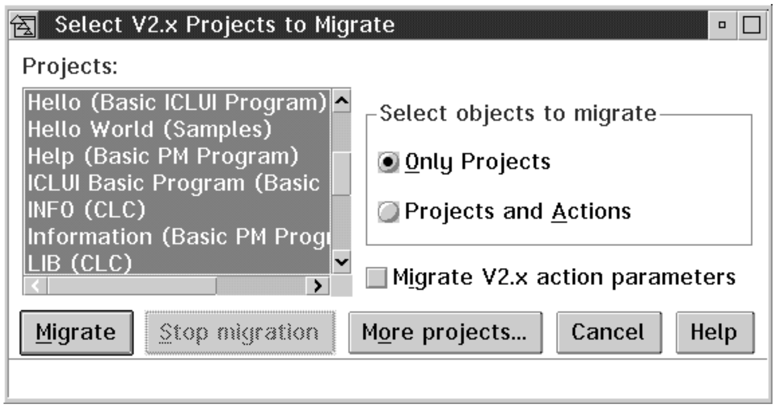

Select V2.x Projects to Migrate
Projects:
Hello (Basic ICLUI Program)
Hello World (Samples)
Help (Basic PM Program)
ICLUI Basic Program (Basic
INFO (CLC)
Information (Basic PM Progi
LIB (CLC)
Select objects to migrate
● Only Projects
○ Projects and Actions
□ Migrate V2.x action parameters
Migrate
Stop migration
More projects...
Cancel
Help

Figure 48. Window listing found Version 2.x projects

Note: The project migration tool does not scan for project templates. If you want to migrate a project template, you must first transform it into a regular project by opening its **Settings** notebook to the **General** page, and then deselecting the **Template** check box.

You must also choose whether to migrate:

Only projects

This is recommended if you are also migrating your project code to use Version 3.0 of the VisualAge C++ tools. If you select this option, the project migration tool creates a Version 3.0 project that emulates your Version 2.x project. It inherits from the VisualAge C++ default project and so all the VisualAge C++ actions will be available to your migrated project. See “The Default Project” on page 36 for more information about the VisualAge C++ default project.

Note: Since the actions are not migrated, neither are their options. You will need to reset the options for the new actions in your project.

Projects and actions

If you want to continue using actions, like Compile and Link, from the previous version of VisualAge C++, you will need to migrate the actions as well. If you select this option, the actions, variables, and types from the old project's actions profile and the default actions profile are copied over to a new project that contains only the information from the profiles. The migration utility calls these projects *actions projects*.

Next, the migration tool creates another project, based on the Version 2.x project, that inherits from the newly created actions project. If the actions profile cannot be found, it is not migrated. In this case, the new project will inherit from the default project, the **VisualAge C++ Project**.

Migrate V2.x action parameters

You can also decide whether or not you want action options migrated as well. The migration utility only migrates the options of actions that use the Version 2.1 default actions support DLL, DDE3DEF2. It cannot migrate the C Set C++ Compiler, Linker, Debugger, EXTRA, and Browser options because they use a custom actions support DLL. Furthermore, some options are not valid for this version of VisualAge C++.

Note: It is recommended that you do not migrate your old action options because the default options of the VisualAge C++ actions are more appropriate than those from the previous version. Some some may even be invalid in this version.

When you push the **Migrate** push button, the migration process starts. The migration tool creates a folder on your desktop called **WorkFrame V3.0 Migration** which contains two folders called **WorkFrame V3.0 Projects** and **WorkFrame V3.0 Action Projects**. If you chose not to migrate the actions from your old projects, the latter folder is empty. The **WorkFrame V3.0 Projects** folder contains the migrated projects.

Migrating Old Projects

What Information is Migrated?

Here is a list of the Version 2.x project information that is migrated by the project migration tool.

Information migrated from projects:

- Target name
- Run parameters
- Run prompt
- Run monitored
- Make file name
- Source directories
- Working directory
- File name filter
  - The migration utility creates a new type called “Migrated file masks” and adds it to the project's **Parts filter** entry field.
- Auto-scroll monitor
- PAMs are migrated as follows:
  - DDE3BPAM to IWFBPAM
  - EVFP370 to IWFP370
  - EVFPADM to IWFPADM
  - EVFP400 to IWFP400
  - All others to IWFBPAM.

The following information is migrated from action profiles into Version 3.0 projects:

- Action scope (If an action is project-scoped, its controlling PAM defaults to IWFBPAM.)
- Action source and target types
- Action run mode
- Options DLLs are mapped to new Actions Support DLLs as follows:
  - DDE3DEF2 to IWFOPT
  - DDE4ICC2 to CPPICC30
  - DDE4ICL2 to CPPICL30
  - All others are migrated as is.
- Environment variables
- Types (All types are migrated to types of class FileMask.)

The following information are not migrated:

- Since there is no concept of a composite project in WorkFrame Version 3.0, composite projects are not migrated. However, any base projects in a composite project can be migrated. The names of the migrated base projects include the name of their composite project in parentheses.

- Action options are not migrated.

Migrating Version 1.x Projects

Projects from WorkFrame Version 1.x can only reside in a single directory, usually the directory where Version 1.x of WorkFrame was installed. To start the search for Version 1.x projects, enter the directory to search and select the **Find projects** push button. The search progress is shown in the information area at the bottom of the window.

After a few seconds, the **Select V1.X Projects to Migrate** window appears. Select the projects you want to migrate, and then select the **Migrate** push button to start the migration.

The migration tool creates a folder on your desktop called **WorkFrame V3.0 Migration** in which a folder called **WorkFrame V3.0 Projects** is created. **WorkFrame V3.0 Projects** folder contains your migrated projects. Projects migrated from Version 1.x inherit their actions from the VisualAge C++ default project.

What Information is Migrated?

The project migration tool migrates the following information from Version 1.x projects:

- Target name
- Make file name
- Source directories
- Working directory
- File name filter

Actions are not migrated.

Migrating Old Projects

Project Access Methods (PAMs)

Reader's Note: Unless you need to use projects that have parts from more than one PAM, or are interested in providing a specialized PAM, you do not need to know the information in this chapter to use WorkFrame effectively.

A Project Access Method (PAM) is a set of functions through which a simple abstraction of a file system or repository is provided to WorkFrame. These functions are packaged in a dynamic link library. PAMs allow a WorkFrame project to contain any kind of object that a PAM can support, for example a version of a file in a source code control library, a foreign file system like MVS or AIX, or an object-oriented database. PAMs also allow WorkFrame to abstract tools from any of these supported locations into actions, so that they can be executed from a WorkFrame project just like any VisualAge C++ action.

Special-purpose PAMs are provided by tool or solution providers who want to use WorkFrame as an integrating framework. WorkFrame includes a basic OS/2 PAM, called IWFBPAM, that provides access to the OS/2 file system, including access to OS/2 files stored on a LAN.

By default, every project uses the basic OS/2 PAM. If your project uses other PAMs besides the basic PAM, the **Locations** tab in the project **Settings** notebook will contain have one page for every available PAM.

Note: If your projects use a PAM other than the basic OS/2 PAM, refer to the information provided by the PAM provider to find out more about how it enhances the behavior of your projects.

The Role of a PAM

WorkFrame has no knowledge of how to access the parts of a project. It relies entirely on the PAM to return the list of parts in a source location and to execute any actions that run on the environment it supports. A PAM may provide specialized support for its environment, like storage and retrieval of database objects, or managing the communication with a remote file server. However, all PAMs must provide the basic support for parts maintenance like returning the list of parts in a source location, copying, moving, or deleting parts. PAMs must also launch actions and pass the output of the action back to WorkFrame.

Support for Multiple PAMs

PAMs are registered globally with WorkFrame, and every registered PAM is available to every project. Because a project can use more than one PAM at a time, a project can contain a mix of project parts supported by different PAMs. This means that a project can contain local files, remote objects, and any other parts supported by the PAMs used by the project. The interface for manipulating project parts is consistent, regardless of the physical location of the part.

Note: The basic OS/2 PAM must be used by all projects unless another PAM that provides the same level of OS/2 support is used.

A PAM is registered with WorkFrame if its DLL name is listed on the IWFPAM environment variable in the CONFIG.SYS file, space-delimited. The basic PAM is always registered by default, even if the IWF.PAM environment variable is not set. To register a PAM with WorkFrame, set the IWF.PAM environment variable like this:

```
   IWFPAM = YOURPAM MYPAM
```

The above statement registers two PAMs with WorkFrame, called YOURPAM and MYPAM. The basic PAM, IWFBPAM, is implicitly registered.

Each PAM adds a page to the **Settings** notebook of every project. This page is for configuration that is specific to a PAM. A project is using a PAM if it has information entered on the page provided by the PAM. For example, the basic OS/2 PAM provides the **OS/2 Files** page where you can enter the source directories and working directory of the project. It is called the **Location** page when there is only one PAM. If your project uses more than one PAM, the **OS/2 Files** page appears as a minor page within the **Location** tab. The **OS/2 Files** page is then accessible using its minor tab within the **Location** page, as are the pages provided by the other PAMs.

In all cases, only the PAM that has access to a given part is responsible for interpreting the part. If you launch an action against multiple parts managed by multiple PAMs, the action is started once for each involved PAM, only for the parts managed by the PAM.

Support for copying and moving parts to and from locations managed by different PAMs, either within a project or between projects, depends on the support provided by each PAM. If a PAM supports the copying of its parts to a location managed by another PAM, both PAMs must either maintain an OS/2 representation of its parts, or be able to copy parts to and from OS/2 as files. Copying and moving between remote PAMs is completed by the source PAM copying the part to an OS/2 location that WorkFrame specifies, and the target PAM copying the part to the destination. Each PAM can specify whether its parts are always available locally, as OS/2 files.

If they are, WorkFrame can optimize copying files by bypassing the intermediate copy step.

Project parts from different PAMs can be sorted together, or by PAMs. You can specify the sort criteria for your project's parts in the the **Sort** page of the project's **Settings** notebook. 📖 See “Project Settings - Sort Page” on page 31 for more information on how to sort project parts.

For more information on how to write your own PAM, obtain the WorkFrame Version 3.0 Integration Kit[1].

Compatibility

PAMs that were written for Version 2.1 of the WorkFrame product are no longer supported in Version 3.0. This is because PAMs are now required to provide a custom **Location** page for the project **Settings** notebook. Contact your PAM provider for an updated version of a Version 2.1 PAM.

Different versions of PAMs with the same name cannot coexist on the same system.

[1] To find out when and where this kit will be available, send a note to workframe@vnet.ibm.com, or call the VisualAge C++ automated help line 1-800-992-4777. Availability will also be announced on various networks where VisualAge C++ Service and Support is present.

Project Access Methods (PAMs)

Part 2. Editing Files

The following tasks will familiarize you with the VisualAge C++ editor. You can use this editor to edit C and C++ files. You can also use the editor to start other tasks, by selecting choices from the VisualAge C++ menu. From the editor, you can compile, link, debug, and browse your program. For more help with using editor commands, see the *Editor Command Reference*.

Introduction to the Editor

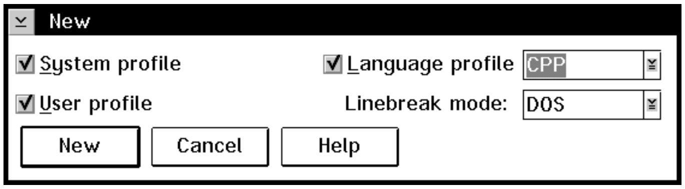
This chapter introduces you to some of the basic features of the editor. Examples in this section often build on each other and should be performed in the order given.

As you perform the suggested exercises in this section, pay close attention to the various command menus you see. Many actions on these menus can also be accessed directly from the keyboard with special keystroke combinations. Where applicable, such keystroke combinations appear to the right of the command name in the menus.

Creating a New File

You can use the editor to create a new file in many different ways. Some are:

- If the editor is not already running, start the editor by doing one of the following:
  - Double-click on the editor icon in an application folder. An empty editor window similar to that shown in Figure 50 on page 156 will appear.
  - Type **lxpm** at an OS/2 Command Line prompt.
- If the editor is already running, do the following:
  1. Select the **File** menu choice from the editor menu-bar.
  2. Select the **New...** choice from the resulting pulldown menu. The window shown in Figure 49 will appear.



New

☑ System profile

☑ User profile

☑ Language profile CPP

Linebreak mode: DOS

Figure 49. New File Window

  3. Use the check boxes and selection lists to select the options you want. For example, in Figure 49, the CPP language profile has been selected.
  4. Click on the **New** button in this window. A new, empty editor window similar to that shown in Figure 50 on page 156 will appear.
  5. You can start entering the text for your new file into this window.

Editor Introduction

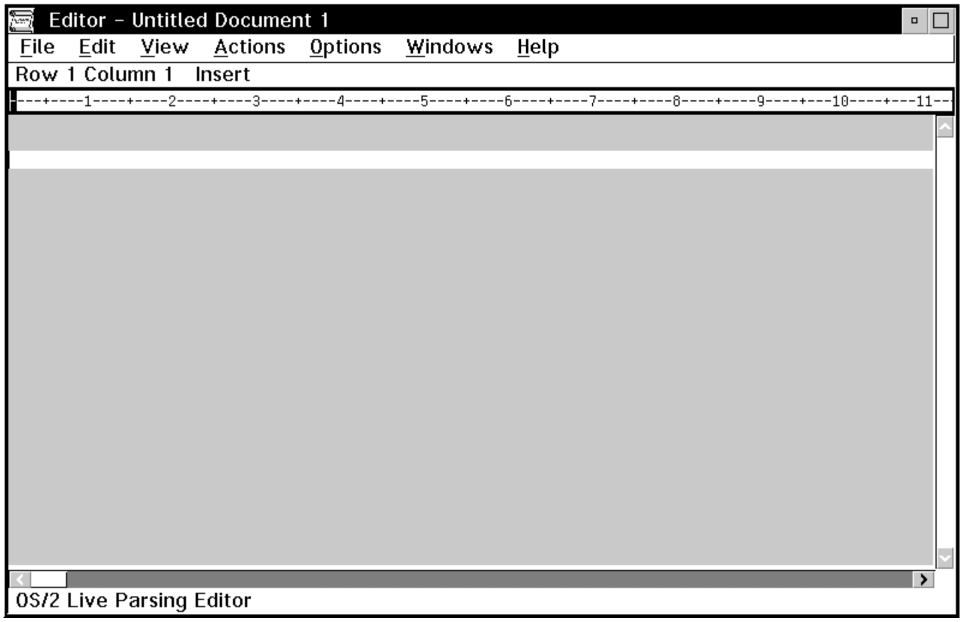


Figure 50. Editor Window

Entering and Editing Text

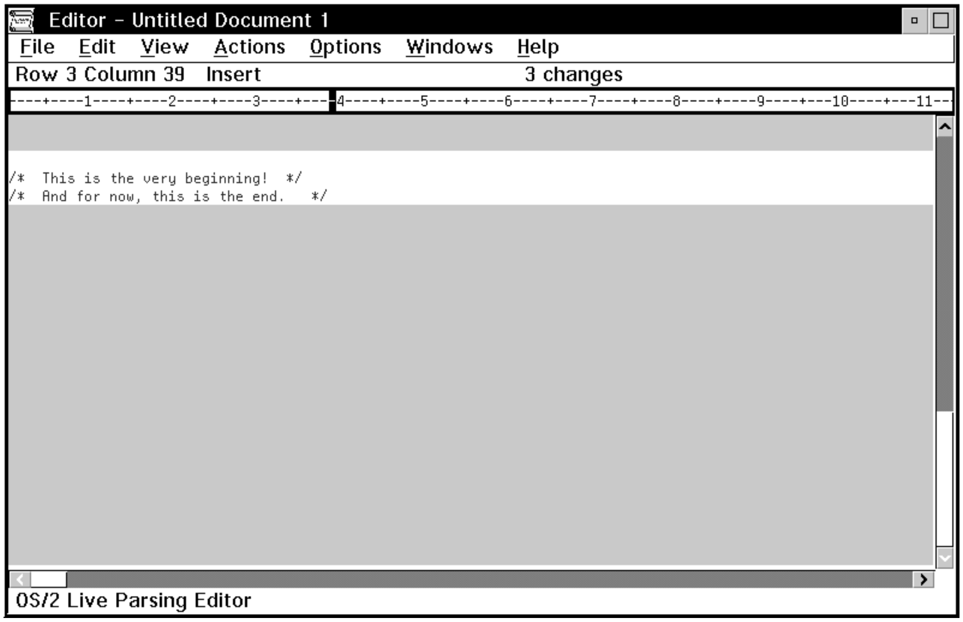
This section shows you how to enter text in the editor window. This section also introduces you to some of the special keys you can use to edit text. For more information on the keys available for a window, select the **Help** choice from the menu-bar, then select the **Keys help** choice from the resulting pulldown menu.

Entering Text

To enter text, just begin typing. Type the following text in the editor:

```
/*  This is only the beginning!   */
/*  And for now, this is the end.   */
```

If you make a typing error, use the **backspace** key to go back and correct the error. When you are done, your editor window will look similar to Figure 51 on page 157.



```
/*  This is the very beginning!  */
/*  And for now, this is the end.   */
```

Figure 51. Editor Window with New Text

Moving the Cursor

Use any of the following methods to move the cursor in the editor window:

- Position the mouse pointer where you want the cursor to be, and click mouse button 1.

  Tip! If you inadvertently select text by dragging the mouse, deselect the text by pressing Alt+U or by clicking mouse button 2.

- Press the Up, Down, Left, or Right arrow keys.
- Press the Home key to move the cursor to the beginning of a line, or the End key to move it to the end of a line.
- Press Alt+Left arrow to move the cursor one word left, or Alt+Right arrow to move it one word right.
- Press the Page Up or Page Down keys to move the cursor up or down one window at a time.
- Press Ctrl+Home to move the cursor to the beginning of a file, or Ctrl+End to move it to the end of a file.
- Press Ctrl+J to move the cursor to the place where you last entered text in the file.

Editor Introduction

Inserting a Blank Line

To create a new line in the editor window, do one of the following:

- Move the cursor to any point on a line. Press Ctrl+Enter.
- Move the cursor to the end of a line. Press the Enter key.

A blank line is created after the current line and the cursor moves to the first column position of this new line.

Replacing and Inserting Text

The status area below the menu-bar indicates which mode the editor is in. In *Replace* mode, the shape of the cursor is a solid block one character width in size. Text under this block is replaced by any new text you type. In *Insert* mode, the shape of the cursor is a thin vertical line. Any new text you type is inserted into the file at the cursor position, and existing text right of the cursor shifts to the right.

To toggle back and forth between the two modes, press the Insert key. For example, place the cursor at about the middle of any line and do the following:

1. Ensure that the editor is in *Insert* mode and type a few words. New text is inserted at the cursor position without overwriting existing text.
2. Put the editor in *Replace* mode and type a few words. New text overwrites existing text at the cursor position.

Splitting and Joining Lines of Text

To split a line so part of it forms a new line below the current line, do the following:

1. Move the cursor to where you want to split the line. When you split a line, all text right of the cursor moves to a new line created below the current line.
2. Press Alt+S to split the line and leave the cursor at its current position, or press the Enter key to split the line and move the cursor to beginning of the new line.

To rejoin the lines, do one of the following:

- If the cursor is at the end of a line, press the Delete key to join the current line and the next line together.
- If the cursor is at the beginning of a line, press the Backspace key to join the current line and the previous line together.
- You can also place the cursor anywhere on a line and press Alt+J to join the current line and next line together.

Deleting Text

There are many ways to delete unwanted text. Some are:

Deleting Single Characters with the Delete Key - In *Insert* mode, the character right of the cursor is deleted. In *Replace* mode, the character under the cursor is deleted. In either mode, text right of the cursor moves left to fill the resulting gap.

Deleting Single Characters with the Backspace Key - The character left of the cursor is deleted. Text right of or under the cursor moves left to fill the resulting gap.

Deleting from the Cursor to the End of the Line - To delete all text between the cursor position and the end of the current line, press Ctrl+Delete.

Deleting an Entire Line - To delete a complete line, move the cursor to the unwanted line and press Ctrl+Backspace.

Exercise

Before continuing with the next exercise, edit your file so it contains only the following lines:

```
/*  This is only the very beginning!   */
/*  And for now, this is the end.   */
```

Undoing Changes

The editor records each change you make to a file in the editor window. The number of changes made since the last file save is displayed on the status line below the editor menu-bar. If you want to undo a change, do the following:

1. Select the **Edit** choice from the editor menu-bar.
2. Select the **Undo** choice from the resulting pulldown menu. The last change to the file is undone.
3. Repeat the above steps until all unwanted changes have been undone.

Exercise

Try this with the text you entered in the previous exercise:

1. Move the cursor to the left of the word `very` on the first line in your file.
2. Delete text from the cursor to the end of the line by pressing Ctrl+Delete.
3. Undo this change as described above.
4. Press the backspace key a few times.
5. Undo the changes until the original line is restored.
6. Type some text anywhere in the file. Move the cursor to a different line and type some more text.
7. Undo changes until the original lines are restored.

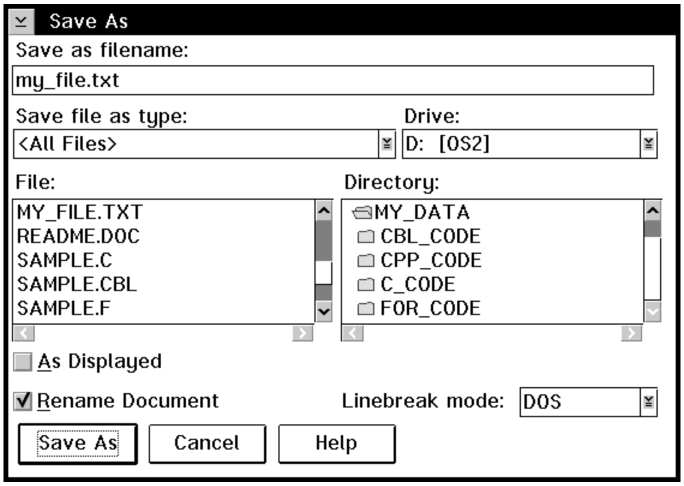
Before continuing with the next exercise, restore your file so it contains only the following lines:

```
/*  This is only the very beginning!   */
/*  And for now, this is the end.   */
```

Saving a File

To save your file, do the following:

1. If you have more than one editor window open, select the window that contains the work you want to save.
2. Select the **File** menu choice from the editor menu-bar.
3. Select the **Save** choice from the resulting pulldown menu.
4. If you are saving an existing file, the file will be saved under its current name. If you are saving a new file, the **Save As** window will appear, as shown in Figure 52. Enter the new name of the file and click on the **Save As** button.



Save as filename: my_file.txt

Save file as type: <All Files>

Drive: D: [OS2]

File:

MY_FILE.TXT
README.DOC
SAMPLE.C
SAMPLE.CBL
SAMPLE.F

Directory:

MY_DATA
CBL_CODE
CPP_CODE
C_CODE
FOR_CODE

□ As Displayed

☑ Rename Document

Linebreak mode: DOS

Figure 52. Save As Window

Before continuing with the next exercise, do the following:

1. Edit your file so it contains only the following lines:

```
/*  This is only the very beginning!   */
/*  And for now, this is the end.   */
```

2. Save your file under the name `my_file.txt`, as described above.

Closing an Editor Window

The editor lets you have more than one view of a document open at a time, each within its own editor window. This may affect the method you choose to close an editor window.

If you have more than one view of a document open, and:

- you want to close all views of that document, do the following:
  1. Select the **File** menu choice from the editor menu-bar.
  2. Select the **Close all views** choice from the resulting pulldown menu.
  3. All windows containing views of that document are closed. The editor will warn you if an editor window has unsaved changes.
- you want to close only one view of a document, do the following:
  1. Open the system menu of the editor window you want to close.
  2. Select the **Close** choice from the resulting pulldown menu.
  3. The selected editor window is closed. The editor will warn you if the editor window has unsaved changes.

At this time, close all views before continuing with the next exercise.

Opening an Existing File

To open an existing file, do one of the following:

- Drag a file icon from a File Manager window onto an Editor icon.
- If you are in the WorkFrame/2 environment, you can:
  – Double-click on an icon representing a source file, or,
  – Select an icon representing a source file, then select **Edit** from the pop-up menu for that icon. To raise an icon's pop-up menu, move your mouse pointer to the desired icon and press mouse button 2.
- At an OS/2 Command Line prompt, type

  ```
  lxpm filename.ext
  ```

  where `filename.ext` is the name of the file you want to edit.

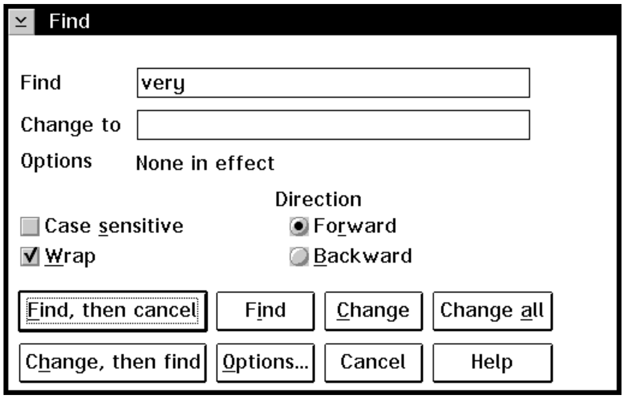
An editor window opens showing the chosen file. You can begin editing this file, as described in "Entering and Editing Text" on page 156.

For this and the following exercises, open the `my_file.txt` file that you saved in "Saving a File" on page 160.

Finding Text

To find a word in your document, do the following:

1. Select the **Edit** choice from the editor menu-bar.
2. Select the **Find...** choice from the resulting pulldown menu. A window similar to that shown in Figure 53 appears.



Find: very

Change to:

Options: None in effect

☐ Case sensitive

☑ Wrap

Direction

● Forward

○ Backward

Figure 53. Finding Text

3. Type the word you want to find in the **Find** text entry area.
4. Select any desired options.
5. Click on the **Find, then cancel** button. The **Find** window disappears and the cursor moves to the next occurrence of the chosen word.

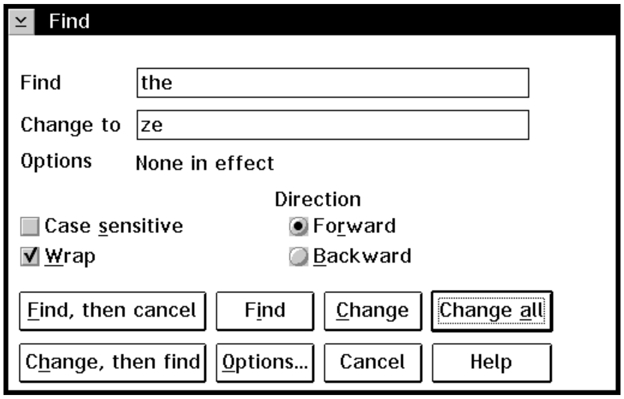
Exercise Try this with the `my_file.txt` file:

1. Move the cursor to the beginning of the first line.
2. Find the word `very`.

Finding and Replacing Text

To search for and replace words with other words of your choice, do the following:

1. Bring up the **Find** window as described above.
2. Type the word you want to find in the **Find** text entry field.
3. Type the word that will replace the found word in the **Change to** text entry field. Figure 54 shows an example.



Find: the

Change to: ze

Options: None in effect

□ Case sensitive

☑ Wrap

Direction

● Forward

○ Backward

Figure 54. Finding Text

4. Click on the **Change** button to find and replace only the next word found, or click on **Change all** to replace all occurrences of the found word in the file.

Exercise

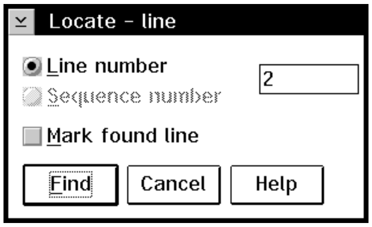
Try this with the `my_file.txt` file:

1. Bring up the **Find** window. Fill in the entries required to replace the word `the` with `ze`.
2. Click on the **Change all** button to change all occurrences of `the` to `ze`.
3. Look at the resulting file to see the changes.
4. Bring up the **Find** window. Fill in the entries required to replace the word `ze` back to the word `the`.
5. Click on the **Change, then find** button. The first occurrence of `ze` is changed, then the cursor is moved to the next occurrence of `ze`.
6. Continue clicking on the **Change, then find** button until the message line at the bottom of the window indicates that there are no more occurrences of the word `ze`. Then click on the **Cancel** button.

Finding Lines in the File

A quick way to find a specific line in a file is to use the **Locate Line** window.

1. Select the **Edit** choice from the editor menu-bar.
2. Select the **Locate** choice from the resulting pulldown menu.
3. Select the **Line...** choice from the next pulldown menu. The **Locate Line** window appears, as shown in Figure 55. The **Line Number** text entry field displays the line number of the current cursor position.



Locate - line

● Line number 2

○ Sequence number

□ Mark found line

Figure 55. Locate Line Window

4. Replace the current line number with another number, and click on the **Find** button. The **Locate Line** window disappears and the cursor moves to the beginning of the chosen line.

   Note: If you try to go to a line number that is greater than the number of lines in the file, the cursor will move to the beginning of the last line in the file.

Creating and Finding Marks

Marks function like bookmarks in a file. You can use marks to easily move around between *marked* positions in your file.

Marks have the following characteristics:

- Marks are not attached to a specific character, but to a specific cursor position. Changes to text at a marked position does not alter the mark.
- If text is inserted or deleted to the left of a marked position, the mark will shift accordingly.
- If lines are inserted or deleted before the line that a marked position is on, the mark will shift accordingly.
- If the marked position itself is deleted, the mark is lost.
- When you close a file, all marks are lost.

Naming Marks

To create, or name, a mark, do the following:

1. Place the cursor at the position to be marked.
2. Select the **Edit** choice from the editor menu-bar.
3. Select the **Name a mark...** choice from the resulting pulldown menu. A window similar to that shown in Figure 56 appears.

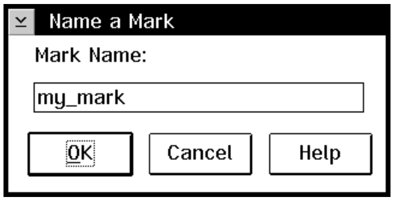


Figure 56. Name a Mark Window

4. Enter the name of the new mark in the text entry field.
5. Click on the **OK** button.

Finding Marks

To move the cursor to a named mark, do the following:

1. Select the **Edit** choice from the editor menu-bar.
2. Select the **Locate** choice from the resulting pulldown menu. A window similar to that shown in Figure 57 appears, listing all marks currently defined in the file.

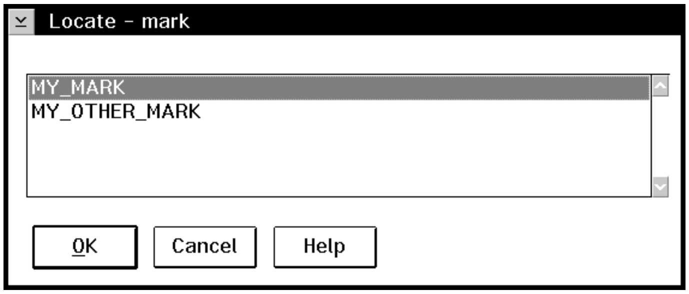


Figure 57. Find a Mark Window

3. Select the name of the mark you want to move to.
4. Click on the **OK** button.

Exercise

Try this with the `my_file.txt` file:

1. Move the cursor anywhere on the first line of your file.
2. Create a mark called `my_mark`, as described in "Naming Marks."
3. Move the cursor anywhere near the middle of the second line.
4. Create a mark called `my_other_mark`.
5. Find the mark called `my_mark`, as described above.
6. Move the cursor to the start of the second line, and insert some text. Then find the mark called `my_other_mark`.
7. Find the mark called `my_mark`. Press the Delete key to remove the marked position, then try to find the mark again.

Using a Quick Mark

To mark one cursor position quickly, you can use the **Quick Mark** feature. To do so, do the following:

1. Move the cursor to the position where you want to set a mark.
2. Select the **Edit** choice from the editor menu-bar.
3. Select the **Set quick mark** choice from the resulting pulldown menu. A quick mark is set at the chosen position in your file.

To find a quick mark, do the following:

1. Select the **Edit** choice from the editor menu-bar.
2. Select the **Locate** choice from the resulting pulldown menu.
3. Select the **Quick mark** choice from the next pulldown menu. The cursor is moved to the position marked by the quick mark.

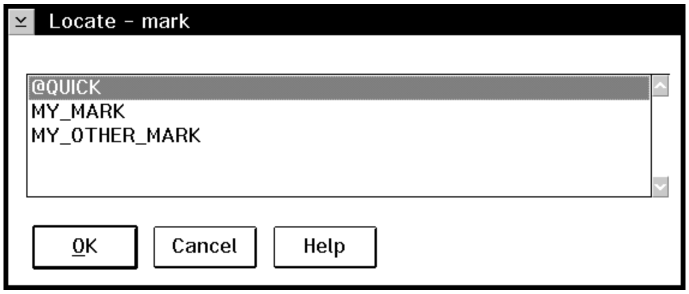
You can have only one quick mark per editor window. Setting a new quick mark will cause a previous quick mark to be lost.

Exercise

Try this with the `my_file.txt` file:

1. Move the cursor to any position within your file.
2. Create a quick mark.
3. Move the cursor elsewhere in the file.
4. Find the quick mark using the method described above.
5. Repeat steps 1 to 4 at a different cursor position in your file.

You can also find a quick mark using the procedure described in "Finding Marks" on page 165. The quick mark will appear in the list of named marks as **@QUICK**.



Locate - mark

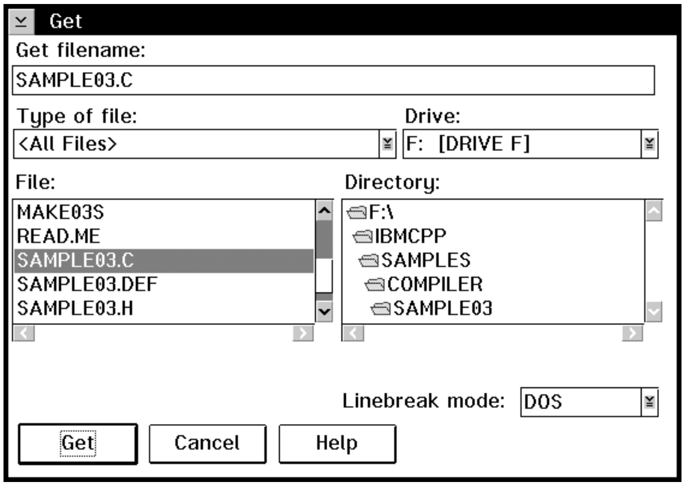
@QUICK
MY_MARK
MY_OTHER_MARK

Figure 58. Find a Mark Window

Inserting Text From Other Files

To read existing text from another file into your file, do the following:

1. Move the cursor anywhere on the line above where the inserted text is to appear.
2. Select the **File** choice from the editor menu-bar.
3. Select the **Get file...** choice from the resulting pulldown menu. A window similar to that shown in Figure 59 appears.



Get

Get filename: SAMPLE03.C

Type of file: <All Files>

Drive: F: [DRIVE F]

File:
MAKE03S
READ.ME
SAMPLE03.C
SAMPLE03.DEF
SAMPLE03.H

Directory:
F:\
IBMCPP
SAMPLES
COMPILER
SAMPLE03

Linebreak mode: DOS

Figure 59. Get File Window

4. Select the desired file and click on the **OK** button. The inserted text will appear on the line(s) after the cursor position.

Exercise

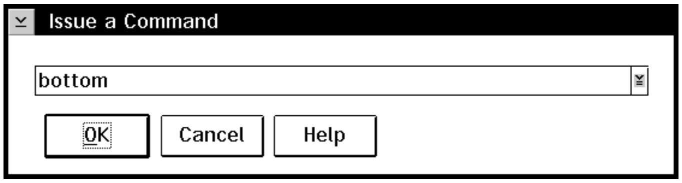
Try this with the `my_file.txt` file:

1. Position the cursor anywhere on the first line in your file.
2. Insert the file from `\IBMCPP\SAMPLES\COMPILER\SAMPLE03\SAMPLE03`**.C** into your file.
3. Look at the resulting file, then undo the change.

Issuing Commands

The editor is fully programmable. The following exercises describe how to issue commands to it:

1. Select the **Actions** choice from the menu-bar.
2. Select **Issue edit command...** choice from the resulting pulldown menu. The **Issue a Command** window appears. See Figure 60.



bottom

Figure 60. Issue a Command Window

3. In the text entry field, type:

   ```
   bottom
   ```

   and select the **OK** push button (or press Enter). The cursor moves to the last character in the file.
4. Bring up the **Issue a Command** window again.
5. In the entry field, type:

   ```
   add 5
   ```

   Make sure to leave a space between the command and the number.
6. Click on the **OK** push button (or press Enter). Five lines are added at the end of the file after the cursor.

Issuing Multiple Commands

The `mult` command lets you issue multiple commands on one command line. This exercise shows how to use it to perform the two steps of the previous exercise in one step.

1. Bring up the **Issue a Command** window and enter the following command:

   ```
   mult ;bottom ;add 5
   ```

   This `mult` command includes two editor commands: `bottom` moves the cursor to the last character in the file, and `add 5` adds five lines to the end of the file.

2. Select the **OK** push button (or press Enter). Five new lines are added to the end of the file.

There are many other commands you can use in the **Issue a Command** window. For a complete listing of editor commands, select the **Help** menu-bar choice in the Editor window, and then select **Editor reference**.

Tips!

- *To use a command again, press the down arrow in the Issue a Command window to display previously used commands, and select one.*
- *You can use command synonyms when entering commands. For example,* **;** *is the editor default synonym for* **mult ;** *. In step 1 above, you could have instead entered the command:*

  ```
  ; bottom ;add 5
  ```

  Note: *Synonyms are used in the standard profiles that are supplied with the editor.*

- *You can view a log of the messages produced by commands and macros you have used. Try this. Bring up the* **Issue a command** *window and enter the following command:*

  ```
  query list.synonym
  ```

  Then select the **Windows** *choice from the menu-bar, and the* **Macro log** *choice from the resulting pulldown menu. The synonyms currently in use are listed at the bottom of the log.*

- *To get help for a command or determine what its return codes are, type* `HELP command` *in the* **Issue a Command** *window, where* `command` *is the name of the command for which you want help.*

Blocking and Manipulating Text

This section describes how you can mark and manipulate selected blocks of text.

Editor Block Manipulation Facilities

You can use the standard OS/2 clipboard to cut, copy, and paste selected text both within the editor and between the editor and other OS/2 applications. The editor also offers you other ways to quickly manipulate blocks of text within the editor.

Default Editor Marking Mode

The editor uses *stream marking* as its default marking mode. In this mode, the cursor is tied to the marked text selection. If you move the cursor, the marked text selection area will either change or be deselected. Also, if a text block is marked, the marked text will be replaced by the next character you type.

Other Editor Marking Modes

Stream marking, while probably most familiar to users, is of limited use to programmers. The editor offers other marking modes that may be more appropriate for programmers.

In these modes, the cursor is not coupled to the marked text selection. You can mark a text block in a document or view, and copy that text block directly to the same or another document or view without using the OS/2 clipboard. Also, this lack of coupling means that typing within a marked text block does not cause the entire marked text block to be replaced. Instead, depending on whether the editor is in *insert* or *replace* mode, each typed character is either inserted before or replaces the character at the current cursor position.

You can change the default marking mode to one of the following choices:

stream Default. The cursor is coupled to the marked text selection.

character Similar to stream, except that the cursor is *not* coupled to the marked text selection.

element Whole elements are marked. The cursor is not coupled to the marked text selection.

rectangle Rectangular areas can be marked starting at any character position on a line, and extend over as many lines as needed. Rectangular blocking is useful for copying or deleting columns of text. The cursor is not coupled to the marked text selection.

Changing the Default Marking Mode

To change the default block marking mode for the current editing session, do the following:

1. Open the **Issue a Command** window by pressing `Shift+F9`.
2. Enter the following command,

   ```
   set blockdefaulttype markmode
   ```

   where `markmode` is one of the modes listed above.

Unmarking a Block of Text

The following section shows you several ways to mark blocks of text. During and between exercises, you will also have to unmark these blocks.

To do so, use one of the following methods:

- Select the **Edit** menu-bar choice, then select **Block**, and then select **Unmark**.
- Press Alt+U.
- Click mouse button 2.

Marking Blocks of Text

Some of the many ways to mark blocks of text are described below.

Marking a Block of Text with the Mouse

To select a block of text with the mouse, do the following:

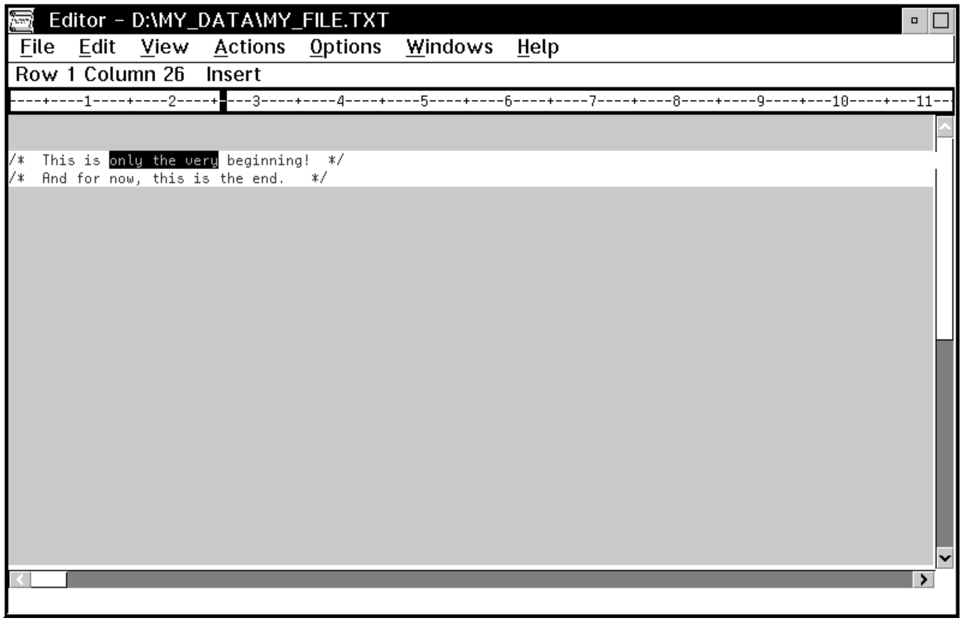
1. Position the mouse pointer over the character that will start the block.
2. Press and hold mouse button 1.
3. Drag the mouse pointer to the to the end of the block of text you want and release the mouse button. The block is now marked, as shown in Figure 61 on page 172.

   Note: The default block marking mode is used when marking text with the mouse.

Exercise

Try this with the `my_file.txt` file:

1. Mark the block `is only the very` on the first line of your file.
2. Unmark the block using one of the methods described in “Unmarking a Block of Text.”



```
/*  This is only the very beginning!  */
/*  And for now, this is the end.   */
```

Figure 61. Marking a Block of Text

Marking a Block of Text without the Mouse

You can mark a block of text with or without a mouse by doing the following:

1. Use the mouse or the cursor keys to move the cursor to the character that will start the block, and press Alt+B.
2. Move the cursor to the end of the block of text and press Alt+B again. Text between the two points are marked.

Marking Complete Lines of Text

Place the cursor anywhere on a line and press Alt+L. The entire line is marked. To mark additional lines of text, move the cursor to the last line to be marked and press Alt+L. All lines between the two selected lines are marked.

Marking a Rectangular Block

To mark a section of text occupying certain columns and lines:

1. Move the cursor to one corner of the desired block, and press Alt+R.
2. Move the cursor to the opposite corner of the desired block, and press Alt+R again. The entire block between corner points is selected.

Alternatively, double-click with mouse button 2 where you want the rectangular block to start, click and hold mouse button 1, and drag to where you want the block to end.

Manipulating Marked Blocks

Once you have marked a block of text, you can manipulate the marked block with either OS/2's standard clipboard-based facilities or the editor's own block manipulation facilities. The following examples illustrate the editor's own block manipulation facilities and the power of marking modes in which the cursor is not coupled to the marked text selection.

Before continuing, do the following:

1. Open a file of your choice for editing.
2. Bring up the **Issue a Command** window by pressing Shift+F9, and issue the following command to change the default marking mode:

   ```
   set blockdefaulttype rectangle
   ```
3. Mark a rectangular block of text.

Copying a Marked Block

1. Move the cursor to the location in the file where the blocked text will be copied.
2. Select the **Edit** choice from the editor menu bar.
3. Select the **Block** choice from the resulting pulldown menu.
4. Select the **Copy** choice from the next pulldown menu. A copy of the marked block is inserted at the cursor position.

Alternatively, position the cursor and then press Alt+C to copy a selected block.

Deleting a Marked Block

1. Select the **Edit** choice from the editor menu bar.
2. Select the **Block** choice from the resulting pulldown menu.
3. Select **Delete** from the next pulldown menu to delete the marked block.

Alternatively, press Alt+D to delete a marked block.

Moving a Marked Block

To move a marked block to a different location in the file, do the following:

1. Move the cursor to the desired location.
2. Select the **Edit** choice from the editor menu bar.
3. Select the **Block** choice from the resulting pulldown menu.
4. Select the **Move** choice from the next pulldown menu. The block is placed below the cursor.

Alternatively, position the cursor and then press Alt+M, or use Alt+Z if you want to place the block over existing text.

Exercise Try this with your file:

1. Mark any word on the first line, and copy it to the end of the second line.
2. Mark a rectangular block of text. Move the cursor anywhere *outside* the marked block. Move the marked block to the new cursor location.
3. Delete the rectangular block that you have just moved.
4. Mark a complete line. Copy the marked line to a different line.

Opening, Closing, and Moving between Views

Displaying multiple views of a source file can help you carry out functions such as cutting and pasting from one section to another. Any changes you make to one view are automatically updated in all other views.

Exercise Try this with the `my_file.txt` file:

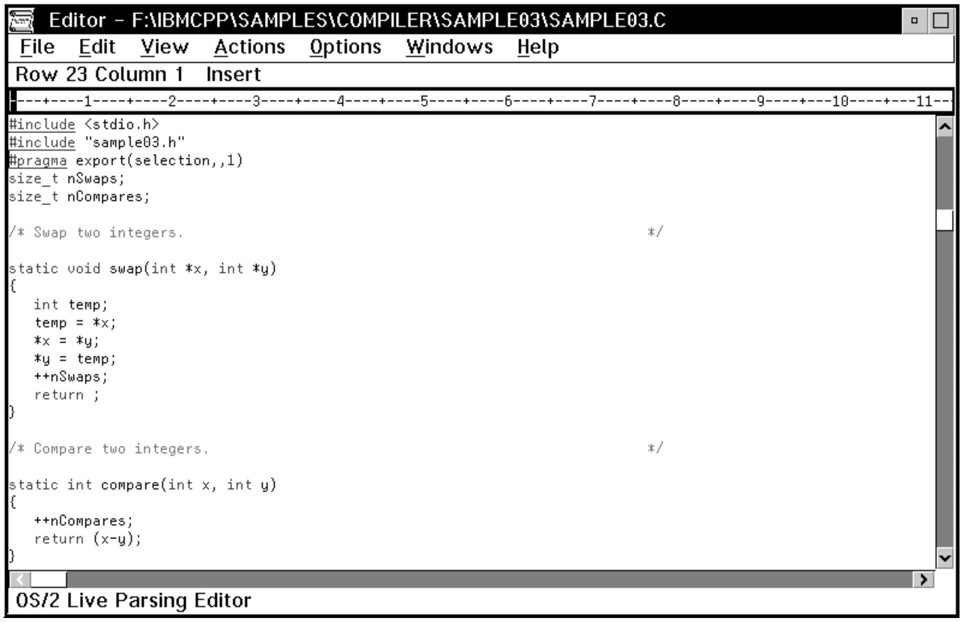
1. Open a new view. Select the **File** menu bar choice, then select **Open new view**. A second Editor window appears, containing exactly the same text as the first.
2. Arrange the two windows so you can see as much of them as possible.
3. Press Ctrl+End in both windows to go to the end of the file. In one window, position your cursor to the start of the phrase `the end` in the last line.
4. Change `the end` to `the new end`. The change occurs in both windows.
5. Move to another view. Select the **Window** menu bar choice, then select **Next view**.
6. Close one of the views by selecting its system menu, and then selecting **Close**.

Using a Parser

The Editor supplies parsers for several programming languages such as C, C++, COBOL, and REXX. An emphasis parser uses colors and fonts to highlight different items in the program, but maintains the original indentation of the program.

To see how the parser works, do the following:

1. Open the `\IBMCPP\SAMPLES\COMPILER\SAMPLE03\SAMPLE03`**.C** file for editing.



```
#include <stdio.h>
#include "sample03.h"
#pragma export(selection,,1)
size_t nSwaps;
size_t nCompares;

/* Swap two integers.                                                     */

static void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
   ++nSwaps;
   return ;
}

/* Compare two integers.                                                  */

static int compare(int x, int y)
{
   ++nCompares;
   return (x-y);
}
```

Figure 62. C Program in Editor

When a file is opened, the editor uses the file name's extension to select the appropriate load macro. A load macro contains a line that specifies which parser to load. When you opened the `SAMPLE03.C` file, the C parser was automatically invoked. A parser and its load macro also:

- Set default fonts and colors
- Set up special actions for keys
- Color and emphasize the data being displayed
- Set up additions to the **View** menu that contain selections specific to the chosen parser

2. Scroll the document to see the different colored fonts used to identify the different parts of a C language document.

Using Elements, Fonts, and Classes

Each element or line in a document may contain more than one class. Classes are used to describe the type of data the element contains. The element displayed in Figure 63, for instance, is made up of a CODE class and a COMMENT class.

A variety of colored fonts distinguish the different classes. Figure 63 shows how the fonts and the classes work to emphasize a line of C language code:

	CODE class	COMMENT class
Lines of C language code in document.	`lowkey = array[i];`	`/* array indices */`
Each letter in this font string represents a font type for each character in the line. Each font type could be assigned a different color.	`IIIIII_O_IIIIIPIPP`	`CCCCCCCCCCCCCCCCCCC`

Figure 63. Classes and Fonts in a C Language Element

To display the fonts in a line of code:

1. Put the cursor on a line of code.
2. Select the **Issue edit command...** choice from the **Actions** menu.
3. Type `query fonts` in the text entry box and click on **OK**. The font names in the selected element are shown on the message line below the edit window.

To display the classes in a line of code:

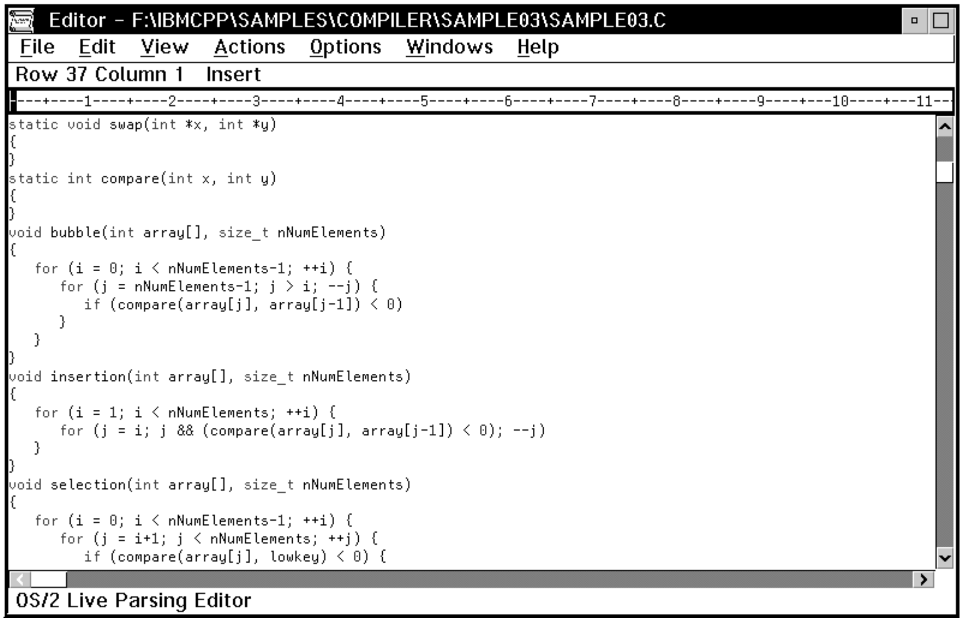
1. Select the **Issue edit command...** choice from the **Options** menu again.
2. Type `query class` in the text entry box and click on **OK**. The class names in the selected element are shown on the message line below the edit window.

Displaying Different Classes

To display different classes of the C language document:

1. Select the **Outline logic** choice from the **View** menu to display only the logic structure of the program sample. See Figure 64 on page 177.
2. Select the **Functions** choice from the **View** menu to display only the names of functions in the program sample.
3. Select **Include all** from the **View** menu to display all the lines.

Note: Available **View** menu choices differ according to the type of file being edited.



```
static void swap(int *x, int *y)
{
}
static int compare(int x, int y)
{
}
void bubble(int array[], size_t nNumElements)
{
   for (i = 0; i < nNumElements-1; ++i) {
      for (j = nNumElements-1; j > i; --j) {
         if (compare(array[j], array[j-1]) < 0)
      }
   }
}
void insertion(int array[], size_t nNumElements)
{
   for (i = 1; i < nNumElements; ++i) {
      for (j = i; j && (compare(array[j], array[j-1]) < 0); --j)
   }
}
void selection(int array[], size_t nNumElements)
{
   for (i = 0; i < nNumElements-1; ++i) {
      for (j = i+1; j < nNumElements; ++j) {
         if (compare(array[j], lowkey) < 0) {
```

Figure 64. C Program in Editor with only Logic Structures Displayed

The **View** menu choices are based on the **include** and **exclude** commands that allow you to see only those elements that have specified classes associated with them.

To use the **include** command to display only the elements that contain items from the COMMENT class:

1. Select the **Issue a command...** choice from the **Actions** menu.
2. In the text entry box, type: `set include comments`
3. Click on **OK**. Only lines with COMMENT class elements are displayed.
4. Select the **Include all** choice from the **View** menu to display all classes again.

Entering Some Code

The editor is a live parsing editor because it monitors and records all the changes made to the document as you work. New data is colored or formatted as you add to its contents.

To type some lines of code into the `sample03.c` document:

1. Position the cursor in the program code, and press Enter several times to insert some blank lines.
2. Type in your own program code. When you have completed a line of code, either press the Enter key or click mouse button 1. The line of code is parsed by the C parser.
3. Do not save changes to the `sample03.c` file.

You are now finished the introduction to the editor.

Customizing the Editor

You can customize the editor so that you can design anything from key assignments to more complex functions such as external commands and parsers.

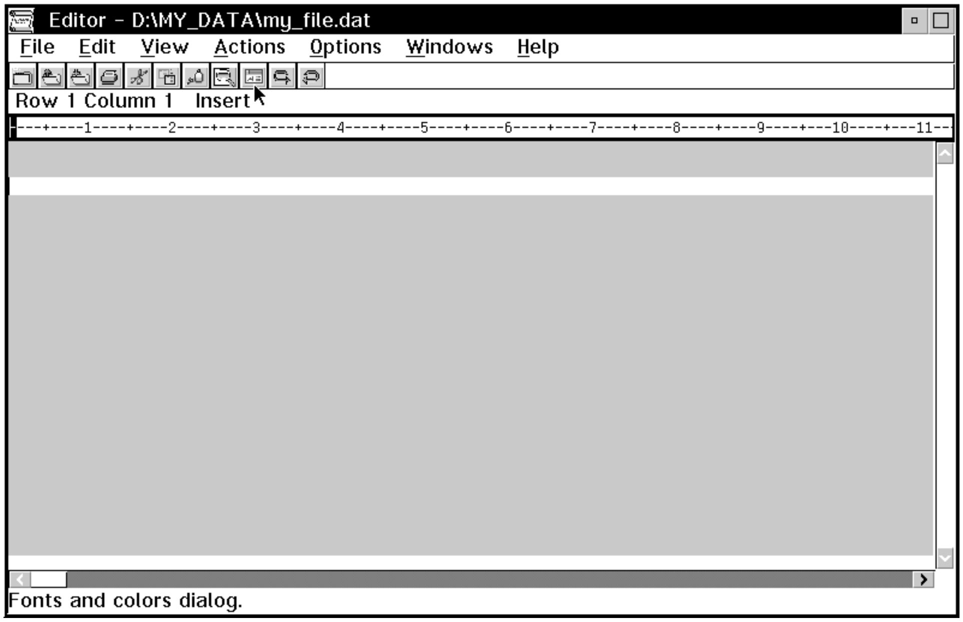
This section covers some simple modifications you can perform. More complicated procedures, such as creating external commands or parsers, require an extensive knowledge of the editor commands and programming.

Using the Editor Tool Bar

The editor has an optional Tool Bar that you can use to invoke commonly used commands with a single mouse click.

If the Tool Bar is not already enabled, you can enable it by doing the following:

1. Select the **View** choice from the editor menu-bar.
2. Select the **Tool bar** choice from the resulting pulldown menu. The Tool Bar appears below the menu-bar, as shown in Figure 65.



Row 1 Column 1 Insert

Fonts and colors dialog.

Figure 65. Editing Window with Tool Bar Enabled

To invoke an action, click on a Tool Bar item. As you move the mouse over a Tool Bar item, a description of that item's action appears in the message line area at the bottom of the editing area.

Customizing the Tool Bar

You can add your own items to the tool bar. The following exercise shows how to add a simple text button called `my button` to the tool bar to the right of the third item already in the tool bar. Pressing the button invokes the `mult ;bottom ;add 5` command, which adds 5 lines to the bottom of the file in the current editor window.

1. Enable the tool bar as described in "Using the Editor Tool Bar" on page 179.
2. Press `Shift+F9` to open the **Issue a Command** window.
3. Enter the following command to define your new tool bar entry. Press `Enter` or click on the **OK** button when you are done.

   ```
   set toolbar.my_button 3 mult ;bottom ;add 5
   ```
4. Open the **Issue a Command** window again. To add a message line description to your new tool bar entry, issue the following command:

   ```
   set help. This button adds 5 lines to the bottom of the file.
   ```

A new entry called `my button` will appear on the tool bar. If you place your mouse pointer over this new button, the button's description will appear in the message line area below the edit window. Selecting the new button will run the `mult ;bottom ;add 5` command, which adds 5 lines to the end of your file.

You can also add icons to the tool bar. For more information, refer to *toolbar Parameter* in the Editor reference.

Customizing the Menus on the Menu Bar

You may want to customize the menus on the menu bar rather than create action keys because:

- Menus are visible and accessible immediately to the new user
- Some menu items can be grayed out and disabled under certain conditions
- Menus can be selected with the mouse
- If desired, a mnemonic key can be assigned to a menu selection.
- If desired, an accelerator key can be assigned to a menu selection.
- You can create context-sensitive help for menus.

To create a new menu and to assign key sequences to each of its selections, use the **set actionbar** command. With the **set actionbar** command, you can add, modify, and delete menus on the editor window action bar.

The menus and key sequences you create are lost when you close the file. To make them available each time you start the editor, see "Modifying Editor Behavior Permanently" on page 185.

Exercise

This exercise creates a new menu bar selection called **Queries**. Pulldown choices associated with this menu bar selection will query various editor parameters. You can substitute the command or commands sequence of your choice. Some of the newly created pulldown choices can be accessed by mnemonic and/or accelerator keys.

1. Press `Shift+F9` to open the **Issue a Command** window.
2. Enter the following command to define the menu bar selection and its first pulldown choice.

   ```
   set actionbar.~Queries.Class 5 query class
   ```

   The ~ before the `Q` designates the `Q` as being the mnemonic for the menu bar selection. There is no mnemonic or accelerator key associated with the `Class` pulldown menu choice. The new menubar selection is placed to the right of the fifth selection already on the menu bar.
3. Enter the following command to define another pulldown choice.

   ```
   set actionbar.~Queries.F~onts\tCtrl+Z 5
   query fonts
   ```

   The new pulldown choice uses the letter `o` as a mnemonic, and displays but does not define Ctrl+Z as an accelerator key. Use the `set action` command to define the accelerator key action. For example:

   ```
   set action.c-Z query fonts
   ```
4. Define one more pulldown menu entry.

   ```
   set actionbar.~Queries.~Block_default 5 query blockdefaulttype'
   ```
5. Select the new menu bar choice and its pulldown choices using the mouse, mnemonic, and accelerator keys as available.

Customizing the Keyboard

The editor provides two mechanisms for customizing the keyboard:

- The **set key** command lets you remap the connections between the physical keyboard and the logical keyboard. This remapping is global and affects all files.
- The **set action** command lets you program the logical keyboard, and assign a command or set of commands to a key. This programming is file related, so the same key can have a different effect for different files.

Remapping the Keyboard Using the Set Key Command

Each physical key on the keyboard is mapped to a key on a logical keyboard. By default, each physical key is mapped onto the corresponding logical key, but this mapping can be altered with the **set key** command.

You can use **set key** to use the editor with a restricted keyboard that does not have all of the required keys. For example, if your keyboard does not have the F11 and F12 keys, you can remap the Alt+1 and Alt+2 key combinations to have the same affect with the following:

```
set key.A-1 F11
set key.A-2 F12
```

Similarly, you can use various accented characters that are not on the keyboard by assigning them to keys:

```
set key.A-A 132     lowercase a umlaut
set key.C-A 142     uppercase A umlaut
```

Note: For both Ctrl and Alt keys, you can use uppercase and lowercase. Alt+a and Alt+A represent the same key combination.

The **set key** command is sometimes simpler than using **set action** because the remapping is global. For example, you could define the Ctrl+H key as the help key in all files using the following:

```
set key.C-H F1
```

Remapping the Keyboard Using the Set Action Command

The editor performs an action associated with a key when it determines which logical key has been pressed. Many keys have default actions built in, while others are set by the **PROFSYS** macro. Any key can be assigned an action using the **set action** command. For example, the following command makes the F7 key scroll the text in a window up one screen:

```
set action.F7 scroll screen up
```

Presentation Manager reserves some keys, such as F1 and F10.

Customizing the Autosave Facility for the Editor

The autosave facility saves a file automatically after a specified length of time has elapsed *or* a specified number of changes have occurred. You may want to customize the autosave facility for each file.

An autosave *triggered by time* occurs when a specified number of minutes has elapsed and at least one change has occurred during the elapsed time. The number of minutes elapsed must be greater than or equal to the duration set by the **autotime** parameter.

An autosave *triggered by changes* occurs when a specified number of changes have occurred. The minimum number of changes is specified by the **autocount** parameter, while the **autochanges** parameter keeps track of the number of changes made. When the total number of changes as recorded by **autochanges** is greater than or equal to **autocount**, an autosave occurs.

In both cases, the keyboard has to be idle for a number of seconds as set by the **idletime** parameter. This ensures that you are not typing data while the editor is trying to autosave.

Example

A file is opened with the following autosave variables:

- autotime 5
- autocount 50
- idletime 2

This means an autosave is triggered every 5 minutes, provided at least one change has been made. An autosave is also triggered after every 50 changes made to the file, regardless of the elapsed time. However, no autosave actually takes place unless you stop typing data for at least two seconds.

Procedure

To customize the autosave settings for a file:

1. Find out the current autosave settings with the following commands:

   query autocount Gives the number of changes that must occur before each autosave. This is set by **set autocount**.

   query autonext Gives the number of seconds before the next autosave will occur.

   query autotime Gives the number of minutes that must elapse before each autosave. This is set by **set autotime**.

   query idletime Gives the number of seconds of no data entry before an autosave will occur. This is set by **set idletime**.

query autochanges
Gives the number of changes since the last autosave. This can be changed by **set autochanges**.

2. Use the appropriate parameters (**autotime**, **autochanges**, **autocount**, and **idletime**) to alter the autosave settings.

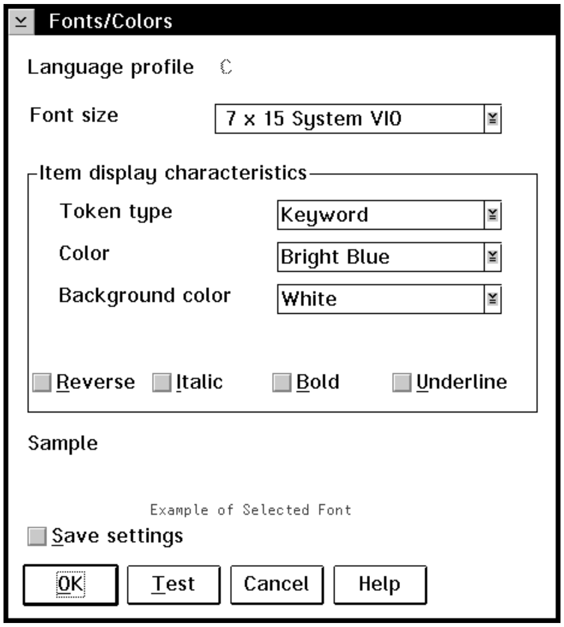
   For example:

   ```
   set autotime 15
   ```

Customizing Editor Fonts and Colors

To customize the colors and fonts used to emphasize different parts of a program source file, do the following:

1. Select the **Options** choice from the editor menu-bar.
2. Select the **Fonts/colors...** choice from the resulting pulldown menu. A window similar to that shown in Figure 66 appears.



Language profile C

Font size 7 x 15 System VIO

Item display characteristics

Token type Keyword

Color Bright Blue

Background color White

□ Reverse □ Italic □ Bold □ Underline

Sample

Example of Selected Font

□ Save settings

Figure 66. Fonts/Colors Window

3. Use the selection lists and check boxes to:
   - Select a base font for the editor window
   - Select a color scheme for each type of text element.
   - Select a character emphasis for each type of text element.

     For example, Figure 66 on page 184 shows that the color coding for a C Keyword token is Bright Blue on a White background, with no special character emphasis.
4. If you want your font and color selections saved for future edit sessions, select the **Save Settings** check box. Otherwise, color and font settings will stay in effect only for the current edit session.
5. When done, click on the **OK.** button.

Modifying Editor Behavior Permanently

Most customizations remain in effect only for the current edit session. To use your customizations in future edit sessions, you must save those customizations to a profile.

Standard Editor Profiles

Much of the editor's flexibility comes from **profiles**, which are text files containing editor commands. Some of the profiles supplied with the editor provide specific editing features, and run automatically at specified times.

The standard editor profiles are kept in the `\IBMCPP\MACROS` directory. You can edit these profiles directly, but **always** keep a backup. If you change them incorrectly, the behavior of the editor may be affected.

The following types of profiles are provided:

PROFINIT Sets global settings when the editor is initialized. This profile is only run once during initialization.

PROFSYS.LX Sets up the editor for each document. It contains commands such as menu and key setup commands that need to be run only once and are for all types of files. This profile is run once per document, before the xxx.LXL and PROFILE.LX profiles. Its contents should not be altered directly. To customize editor behavior, use the **PROFILE.LX** file instead.

xxx.LXS This profile runs whenever a file with filename type `xxx` is saved, including autosaves. Use of this profile is optional. For example, you could create an CPP.LXS profile to be run when files with a CPP source type are saved.

xxx.LXL	This is a load profile containing commands (such as parser selection) specific to files with a certain source type. For example, the C.LXL load profile is run if a file with a filename extension of C is opened.
PROFILE.LX	This is a profile in which you can save your own customization commands. It is the last standard profile that is run before a file is opened, and its commands will modify the commands used in previous profiles.

User-Defined Load Profiles

You can directly modify the IBM-supplied editor load profiles to your own personal preferences, but your modifications may be lost if you later reinstall the editor. To avoid losing your load profile modifications, we suggest you do the following:

1. Do not directly modify the xxx.LXL standard load profiles.
2. Instead, create an xxx.LXU user load profile in the \IBMCPP\MACROS directory, where xxx is the file type to which this profile will apply.

   For example, if your personalized load profile is to apply to .CPP files, your personalized load macro will be called CPP.LXU.
3. In the xxx.LXU, include only the editor commands needed to modify the standard load profile behavior to your preferences.
4. When you load an editor with file type xxx, the editor will load your xxx.LXU user load profile immediately after the xxx.LXL standard load profile.

Note: Personalized load profiles can be stored in the \IBMCPP\MACROS directory, but we recommend that you store them in your own directory. See “Storing Personalized Profiles” for more information.

Storing Personalized Profiles

Personalized profiles should be stored in your own directory. This helps to ensure that your profiles will not be lost if you later reinstall the editor. It also simplifies profile management when the editor is used in a network environment.

To store your profiles in your own directory, do the following:

1. Create a directory in which to store your personal profiles, for example:

   ```
   mkdir \my_prof
   ```
2. Create your personalized profiles and store them in this directory.
3. Add the new directory to the LPATH environment variable in your config.sys file. If this environment variable does not already exist, create it. For example:

   ```
   set LPATH=d:\my_prof;d:\lpex\macros;
   ```
4. Reboot your OS/2 system.

Sample Personalized Profile

This exercise saves some of the customizations performed in this chapter to the **PROFILE.LX** file.

Perform the following steps:

1. Select the **Options** choice from the menu bar.
2. Select the **Profiles** choice from the resulting pulldown menu.
3. Select the **User preferences** choice from the next menu. The editor will load the `PROFILE.LX` file for editing in a new edit window.

   Note: To open all active profiles for editing, select the **All active** choice from the menu. Be sure you have made backup copies of the standard editor profiles before you try to edit them.

4. Modify the `PROFILE.LX` file to contain the following lines. The first line of the profile must be a comment, as shown. Do not forget the quotation marks around the each command line.

   ```
   /* profile.lx */
   'set blockdefaulttype rectangle'
   'set toolbar.my_button 3 mult ;bottom ;add 5'
   'set help. This button adds 5 lines to the bottom of the file.'
   'set actionbar.~Queries.Class 5 query class'
   'set actionbar.~Queries.F~onts\tCtrl+Z 5 query fonts'
   'set action.c-Z query fonts'
   ```

5. Save the file and quit the editor. Your new editor customizations will take effect the next time you start the editor.
6. To cancel your customizations, edit the `PROFILE.LX` file to remove the unwanted editor commands. Save the modifed file.

Note: If you later reinstall the editor software, you will lose your customizations. See “Storing Personalized Profiles” on page 186 for information on how to safely store your personalized profiles.

Customising the Editor

Part 3. Using the Data Access Builder

This part of the *User's Guide* describes the Data Access Builder, which you can use to create database access classes.

11

Overview

Data Access Builder is an application development tool and classes you use to create database access classes. You then use these methods with the database access classes to access a DB2/2 relational database.

Using the Data Access Builder tool, you map your existing relational database tables to object interfaces. Relational database tables map to a class, and columns of the table map to class attributes. Foreign keys are mapped as attributes with the value of the foreign key identifier. Once you've defined your mapping, you can generate code based on it.

This approach is fast and efficient where you have existing data and want to do a simple mapping between the data and an object interface.

The database access methods that Data Access Builder generates perform data manipulation. These methods use static SQL for efficient access. In an application they allow you to:

- Add data
- Delete data
- Update data
- Retrieve data.

In addition, Data Access Builder also gives you a set of pre-defined C++ and SOM classes to:

- Connect to the DB2/2 database
- Access database tables to manage transactions
- Disconnect from the DB2/2 database.

Collectively these are the database access classes.

Using the Actions Profile

Data Access Builder provides an actions profile that you can use for WorkFrame projects that use the Data Access Builder tool. To use the Data Access Builder actions profile with a project:

1. Open the **VisualAge C++ 3.0 Tools** folder.
2. Copy the **DAXSAMP Database DLL** project from the **Database** folder in the Samples folder of **VisualAge C++**. The project should be copied with a new project name.

3. Open the new project.
4. Choose **Settings** from the **View** menu and fill in the appropriate target and location (directories) information for the new project.
5. The SQLPrep, compile, link and import library actions are set appropriately to create the DLL. You need to set the LIB and INCLUDE environment variable to reference this new project target for any projects that use the DLL. You also need to copy the DLL produced by this project to a directory on your LIBPATH. If you generate more than one class for the DLL, you need to create a .def file. Use the ones generated as an example.

Starting Data Access Builder

You can start Data Access Builder using any of the following methods:

- Double click on the **Data Access Builder** icon in the **VisualAge C++ 3.0 Tools** folder.
- Choose **Data Access Builder** from the:
  - **DATABASE** cascade of the **Project** menu of any WorkFrame-integrated tool that uses the Data Access Builder actions profile.
  - **Actions** cascade in the system menu of any WorkFrame project that uses the Data Access Builder actions profile.
- Type `icsdata` from an OS/2 command line, and press Enter.
- Select **Database** from the pop-up menu of any *.dax file in a project.

Saving a Data Access Builder Session

At any time, you can save your Data Access Builder session. All information about the session is saved.

1. Choose **Save** from the **File** menu. If this is the first time you are saving this session, the **Save as** window displays.
   a. Type the name you want to assign to the session in the **Filename** entry field. For example, it is recommended that the file extension be **.DAX**.
   b. Indicate the directory you want the information stored in by selecting one of the drives in the **Drives** list box.
   c. Choose **OK** to save the session.

   On subsequent saves of the same session, the **Save** window does not appear. The session is saved in the name shown in the title bar.

Saving a Data Access Builder Session under Another Name

At any time, you can save the current Data Access Builder session under a different name.

1. Choose **Save as** from the **File** menu.
2. Type the name you want to assign to this session in the **Filename** entry field. For example, it is recommended that the file extension be **.DAX**.
3. Indicate the directory you want the information stored in by selecting one of the drives in the **Drives** list box.
4. Choose **OK** to save the new session name.

Opening a Previously Saved Data Access Builder Session

1. Choose **Open** from the **File** menu.
2. Type the name of the session you want to restart in the **Filename** entry field.
3. Choose **OK** to open the session.

Displaying Pop-Up Menus in Data Access Builder

In Data Access Builder, table icons and class icons have pop-up menus.

To display a pop-up menu:

- Position the mouse pointer over the icon and click mouse button 2.

Opening the Settings for a Class or Table

- Choose **Open settings** from the pop-up menu, or
- Double click on the icon.

Creating Classes from Existing Tables or Views

If you have a table or a view that you want to create a corresponding class for:

1. Create a class following the steps described in “Creating Classes” on page 194.
2. Change the default mapping information following the steps in “Changing the Mapping between a Table and a Class” on page 195. By default, all columns in the table are chosen. This step is only necessary if you want to change the default.
3. Change any settings you want for the class following the steps described in “Changing the Class Name” on page 194. This step is only necessary if you want to change the default.
4. Generate the class following the steps described in “Generating Code Using Data Access Builder” on page 197.

Creating Classes

By default, the **Startup** window displays.

1. Choose the **Create classes...** push button. The **Create classes** window displays.
2. Select the database you want to work with from the **Databases** list box. Choose the **Connect** push button. The **Tables** list box fills with all tables in that database.
3. Select the table, or tables, you want to create a class for.
4. Choose **Create classes**. An icon for each table selected and the associated class is created in Data Access Builder.

Viewing a Table

You can view the table by looking at its settings notebook. The settings notebook contains:

- Table name and the database it resides in
- Column names and their definitions
- Constraints

To open a table's settings notebook, follow the steps described in “Opening the Settings for a Class or Table” on page 193.

Deleting the Table Mapping

Data Access Builder automatically provides the mapping between a table and a class. You may decide later to remove the mapping. To delete a mapping:

1. Choose **Delete** from the class icon's pop-up menu. The class is removed from Data Access Builder.

Deleting a Table

Once you have selected a table from the database to use in Data Access Builder, you may decide not to use it.

To delete a table from Data Access Builder

1. Choose **Delete** from the table icon's pop-up menu.

Changing the Class Name

1. Choose the **Open settings** from the class icon pop-up menu.
2. Choose the **Name** page of the class notebook.
3. Type the name you want to assign to the class in the **Class name** entry field.

You can also change the File name on the same **Name** page.

1. Type the name you want to assign to the files in the **File name** entry field. The file name should not exceed seven characters. One character is reserved for Data Access Builder to distinguish generated file contents.

Changing the Mapping between a Table and a Class

By default, each column in the table is automatically mapped to one attribute in the class. Changing this default mapping is optional.

Deleting a Mapping

1. Choose **Open settings** from the class icon's pop-up menu.
2. Choose the **Attributes** page of the class notebook.
3. Select the mapping you want to delete from the list.
4. Select the **Delete** push button.

Adding a Mapping

If you previously deleted a mapping, you can add that mapping back.

1. Choose **Open settings** from the class icon's pop-up menu.
2. Choose the **Attributes** page of the class notebook.
3. Select the mapping you want to add from the list.
4. Select the **Add** pushbutton to map the table column to the class attribute.

SQL data types are mapped to default C++ data types and IDL data types as follows:

SQL Data Type	C++ Data Type	IDL Data Type
char	*IString*	*string <n>*
date	*IString*	*string <10>*
decimal	*double*	*double*
double	*double*	*double*
float	*double*	*double*
integer	*long*	*long*
long varchar	*IString*	*string*
smallint	*short*	*short*
time	*IString*	*string <8>*
timestamp	*IString*	*string <26>*
varchar	*IString*	*string <n>*

Warning: Data type *double* is an inexact representation of a decimal field. Errors could occur due to arithmetic rounding.

Mapping Tables without Primary Keys

The **Attributes** page of the class notebook shows the details of how the class maps to the table.

A gold (right-pointing) key indicates the primary key. A grey (left-pointing) key indicates the foreign key. By default, the gold key is also indicated on the **Attributes** page as the *data identifier*. This is because there must be a value in the primary key column (not nullable) and the value kept in the column must be unique.

Data identifiers are used to identify a row uniquely. Before the update, delete, and retrieve operations, assign the values to the data identifiers to indicate the row. An error will occur if the row indicated by the data identifier does not exist. For the add operation assign values to the data identifier.

If the table does not have a primary key, the first attribute is selected as the default data identifier. Your application *must* ensure this attribute contains unique values. If values kept in the data identifier identify more than one row, errors occur for the retrieve operation and multiple rows are affected for update and delete operations.

To change the data identifier from its default assignment to your own specified value:

1. Choose **Open settings** from the class icon pop-up menu.
2. Choose the **Attributes** page of the class notebook.
3. Select the mapping that contains the attribute to be the data identifier.
4. Click on the **Data identifier** check box. A check mark appears in the check box. An icon appears in the **Data Id** column, indicating that this attribute is the data identifier.

To remove the data identifier:

1. Choose **Open settings** from the class icon pop-up menu.
2. Choose the **Attributes** page of the class notebook.
3. Select the mapping that contains the attribute to be the data identifier.
4. Click on the **Data identifier** check box. A check mark disappears from the check box. An icon disappears from the **Data Id** column.

At least one attribute **must** be identified as the data identifier.

Changing Attribute Names

1. Choose the **Open settings** from the class icon pop-up menu.
2. Choose the **Attributes** page of the class notebook.
3. Select the mapping that contains the attribute to be changed.
4. Type the name you want to assign to the attribute in the **Attribute Name** entry field.

Generating Code Using Data Access Builder

You can use Data Access Builder to generate Visual Builder parts, IDL, or both.

Once the source code has been generated it should not be modified. Any manual changes are lost when the source code is re-generated.

Generating Visual Builder Parts Using Data Access Builder

Generating the Part

1. Create a class following the steps described in “Creating Classes from Existing Tables or Views” on page 193.
2. Choose **Generate** from the class icon's pop-up menu.

   The following files are created:

`filename`**y.cpp**	C++ source code
`filename`**v.vbe**	Visual Builder part information file
`filename`**v.def**	Module definition file for creating DLL, if you are running from the command line
`filename`**v.hpp**	C++ header file
`filename`**v.mak**	Makefile for the generated files, if you are running Data Access Builder from the command line.
`filename`**v.sqc**	SQL source file

   The files are created in the current directory. If files with the same `filename` exist in the directory, you are asked whether you want to overwrite them.

Generating IDL Using Data Access Builder

1. Create a class following the steps described in “Creating Classes from Existing Tables or Views” on page 193.
2. Choose **Generate** from the class icon's pop-up menu.

   The following files are created in the directory that you started Data Access Builder from:

`filename`**x.cpp**	C++ source code
`filename`**i.idl**	IDL source code
`filename`**i.mak**	Makefile for the generated files if you are running Data Access Builder from the command line.
`filename`**i.sqc**	SQL source file

   Note: `filename` is the name assigned to the class as described in “Changing the Class Name” on page 194.

Viewing Files Generated by Data Access Builder

1. Choose **View source** from the pop-up menu of the class icon. The **View source** window displays.
2. Choose the files you want to view.
3. Choose **VIEW**. The default WorkFrame editor displays the source file.

Part 4. Compiling Your Program

This part of the *User's Guide* describes the input to the compiler, how to compile and link programs, how to set compiler options, and how to use the compiler listing. It also describes static and dynamic linking of programs.

Starting the Compiler

The `icc` command invokes the VisualAge C++ compiler, which takes your C or C++ source code as input and produces an intermediate code file, a preprocessed file, or an object file. The command also invokes the VisualAge C++ linker to link the object file into an executable module or a dynamic link library (DLL).

You can invoke the compiler from:

- WorkFrame
- A make file
- An OS/2 command line or using a `.CMD` file

You can also invoke it from within a program by using the `system` function. For example:

```
system("icc myprog.c");
```

📖 See the *C Library Reference* for information about the `system` function.

To compile without linking, use the `icc` command with the `/C+` option. Then you can link your program yourself, using either the VisualAge C++ linker or any other linker that processes IBM 32-bit object files. 📖 See Part 5, “Linking Your Program” on page 319 for more information on linking.

Compiling within WorkFrame

To use the compiler through WorkFrame, do the following:

1. Open the VisualAge C++ folder.
2. Double click on the **Project Smarts** icon to open the Project Smarts Catalog.
3. In the Project Smarts - Catalog window, select the type of project you want to build.

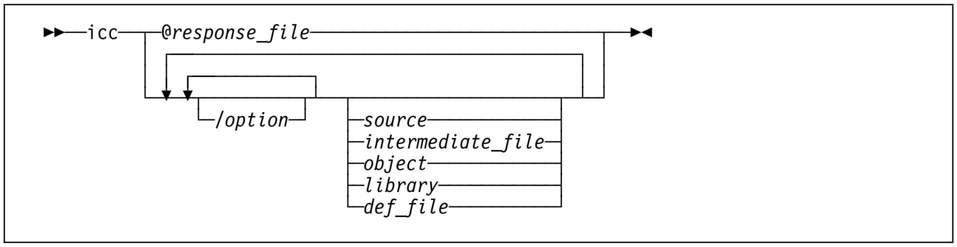
   If you want to build a type of project that is not listed, you can either pick a similar project type and then customise its settings, or create one from the **Project** template without project-specific defaults, and customise its settings extensively.
4. Select **Create**. WorkFrame begins creating the project. The Console window shows the status of the process. Other windows may appear, for you to provide additional information about your project.

5. In these other windows, provide specific information in fields where the default information is unacceptable.
6. Select **OK** when you are done, in each window.
7. Your project becomes an icon on the desktop, or in a folder if you specified a different destination. Find your project.
8. Double click on your project icon. The Project Window appears.

   At this point you can customise settings for the project, if the default settings for the project type are unacceptable. The **Options** menu contains choices that allow you to specify the actions available to the project, and compiler and linker options. Use **Build Smarts** to set options for a standard task. Use the Compiler and Linker Options dialogs to set options on an individual basis.
9. Select **Build** from the **Actions** menu. Your project is created, with the compiler and linker invoked as required.

Compiling from the Command Line

The syntax for the `icc` command is as follows:



Depending on how you want to compile your files and how you have set up the ICC environment variable, many of the parameters used with the `icc` command are optional when you issue the command from a command line.

For example, to compile and link the program `bob.c`, you would enter the following:

```
icc bob.c
```

An object code file `bob.obj`, and an executable file `bob.exe` are created.

For a list of all the VisualAge C++ compiler options see “Compiler Options Summary” on page 264. This summary is also available in the online version of the *User's Guide*. You can jump directly to the summary (or any other topic) from the command line with the `view` command, followed by the book name (`cppug.inf`) and the topic name. For example:

```
view cppug compiler options
```

You can also access an options summary with the `/?` option. Go to an OS/2 command line and type:

```
icc /?
```

This listing is printed to **stderr**, but you can use the OS/2 redirection symbols to redirect it to **stdout** or to a file.

Note: The listing generated by this command is not intended to be used as a programming interface.

Using Response Files

Instead of specifying compiler options and source files on the command line, you can use a *response file*. A response file is a text file that contains a string of options and file names to be passed to `icc`. (The string does not specify `icc` itself.) For example, a response file that contains the single line:

```
/Sa /Fl catherine.c
```

would give the following command line:

```
icc /Sa /Fl catherine.c
```

A response file can have any valid file name and extension. To use the response file, specify it on the `icc` command line preceded by the at sign (@). For example:

```
icc @d:\response.fil
```

No space can appear between the at sign and the file name. You can use multiple response files, and even call another response file from within a response file. You can mix response files with other input on the command line. Options and file names set in the ICC environment variable are still used.

The command string in a resource file can span multiple lines. No continuation character is required. The string can also be longer than the limit imposed by the OS/2 command line. In some situations you may need to use a response file to accommodate a long command line, such as when you use the intermediate code linker or compile C++ code containing templates.

Because the compiler appends a space to the end of each line in the response file, be careful where you end a line. If you end a line in the middle of an option or file name, the compiler may not interpret the file as you intended. For example, given the following response file:

```
/Sa /F
l catherine.c
```

the compiler would construct the command line:

```
icc /Sa /F l catherine.c
```

The compiler would then generate an error that the `/F` option is not valid, and would try to compile and link the files `l.obj` and `catherine.c`.

Compiling from a Make File

Use a make file to organize the sequence of actions (such as compiling and linking) required to build your project. You can then invoke all the actions in one step. The NMAKE utility can save you time by performing actions on only the files that have changed, and on the files that incorporate or depend on the changed files. 📖 See Chapter 58, “Program Maintenance Utility (NMAKE)” on page 815 for more information.

You can write the make file yourself, or you can use WorkFrame to manage the make file. When you build through WorkFrame, a make file is created and maintained automatically.

Controlling Compiler Input

This section describes the methods you can use to control input to the compiler.

Compiling Programs with Multiple Source Files

To compile programs that use more than one source file, specify all the file names on the command line. For example, to compile a program with three source files (`mainprog.c`, `subs1.c`, and `subs2.c`), type:

```
icc mainprog.c subs1.c subs2.c
```

The source file containing the main module can be anywhere in the list. The executable output file will have the same name as the first source file name but with the extension `.exe`. In the example above, the executable file will be `mainprog.exe`.

You can compile a combination of C and C++ files. For example:

```
icc cprog.c cppprog.cpp cxxprog.cxx othprog.oth
```

The file extension determines whether the file is compiled as a C file (`.c` or any other unrecognized extension) or as a C++ file (`.cpp` or `.cxx`). In the example above, `cprog.c` and `othprog.oth` are compiled as C files, and `cppprog.cpp` and `cxxprog.cxx` are compiled as C++ files.

You can also use the `/Tc`, `/Tp`, and `/Td` options to specify whether a file is a C or C++ file, regardless of its extension. The `/Tc` and `/Tp` options apply only to the file name immediately following the option, and specify that the file is a C file (`/Tc`) or a C++ file (`/Tp`). For example, given the following command line:

```
icc /Tc cprog.cpp cppprog.cpp /Tp cxxprog.c
```

`cprog.cpp` is compiled as a C file, and `cppprog.cpp` and `cxxprog.c` are compiled as C++ files.

The `/Td` option applies to all files that follow it on the command line. Use `/Tdc` to specify that all source and unrecognized files that follow are to be treated as C files, or `/Tdp` to specify that they are to be treated as C++ files. (You can specify `/Td` to return to the default handling of files.)

File Types

Option	Behavior
/Tc	Compile next file as C file.
/Tp	Compile next file as C++ file.
/Tdc	Compile all subsequent source files and unrecognized files as C files.
/Tdp	Compile all subsequent source files and unrecognized files as C++ files.
/Td	Compile *.c and unrecognized files as C files. Compile *.cpp and *.cxx as C++ files.

For example, given the following command line:

```
icc /Tdp cxxprog.c othprog.oth /Td newprog.new
```

cxxprog.c and othprog.oth are compiled as C++ files, and newprog.new is compiled as a C file because /Td reset the default handling of files (files with unrecognized extensions are treated as C files).

File Types

The VisualAge C++ compiler uses file extensions to distinguish between the different types of file it uses. The default file extensions are:

.asm	assembler listing file
.c	C source file
.cpp	C++ source file
.cxx	C++ source file
.ctn	temporary file
.def	definition file
.dll	dynamic link library
.exe	executable file
.h	C header file
.hcp	precompiled header file
.hpp	C++ header file
.i	preprocessor output file
.l	temporary file
.lst	listing file
.lib	library file
.m	temporary file
.map	map file
.obj	object file
.pdb	browser file
.w	intermediate file
.wh	intermediate file
.wi	intermediate file
.wit	temporary file
.wli	temporary file
.ws	intermediate file

For example, when you are using VisualAge C++ defaults, the command:

```
   icc  module1.c  module2.obj  mylib.lib  mydef.def
```

compiles the source code file `module1.c` and produces the object file `module1.obj`. When the linker is invoked, the object files `module1.obj` (created during this invocation of the compiler) and `module2.obj` (created previously), the library file `mylib.lib`, and the definition file `mydef.def` are passed to the linker. The result is an executable file called `module1.exe`.

Using Wildcards in File Names

You can use wildcards (`*` or `?`) in the name for any input file. Use `*` to stand for zero or more unknown characters. Use `?` to stand for exactly one unknown character.

For example,

```
   icc  module?.c  my*.lib  mydef.def
```

compiles all source code files that begin with `module` plus one additional character (such as `module1.c` and `module2.c`) and generates object files, with names derived from the source files (for example, `module1.obj` and `module2.obj`). When the linker is invoked, the object files are linked with all libraries that begin with `my` (such as `mylib.lib` and `myothr.lib`).

You cannot use wildcards in the name for an output file. Either accept the default output filenames, or specify an output filename in full.

OS/2 Environment Variables for Compiling

The VisualAge C++ compiler checks the OS/2 environment variables for path information and for the default values of compiler options. If the VisualAge C++ installation program updated your CONFIG.SYS file, many of the environment variables already have values for the compiler to use. If the program did not update CONFIG.SYS, you can set these values by running the CSETENV.CMD file in your OS/2 session before using the compiler.

Some environment variables, for example ICC, are optional. They are not added to your CONFIG.SYS file or to CSETENV.CMD for you.

The environment variables described in this section are called the **compiler** environment variables. A number of environment variables are also used at run time. 🕮 See the *Programming Guide* for more information on the runtime environment variables.

OS/2 Environment Variables

The following OS/2 environment variables affect the operation of the VisualAge C++ compiler during compilation.

DPATH Lists the directories that the compiler searches for help and message files.

ICC Sets compiler options and file names. See “Specifying Compiler Options” on page 253 for a detailed description of the ICC variable.

ILINK Sets options that the VisualAge C++ linker uses when it links the object files that the compiler generates. The options in this variable are processed after the options on the command line, but before any options in the ICC environment variable. If some options conflict, the option that was processed last takes effect.

Note: You cannot specify file names in the ILINK environment variable. If you invoke the linker through the compiler, you can specify file names for the linker in ICC.

INCLUDE Lists directories that the compiler searches for header files.

LIB Lists directories that the linker searches for library (`.LIB`) files. This should include the directory that contains the VisualAge C++ libraries, and the directory that contains the Toolkit's OS2386.LIB library.

LIBPATH Lists directories that executables (including the compiler and linker) search for DLLs that they use to run.

LOCPATH Lists directories that the **`setlocale( )`** function uses to locate locale data. See the *Programming Guide* for more information on creating locales and using the **`setlocale( )`** function. See Chapter 53, “LOCALDEF Utility” on page 689 for information on using the LOCALDEF utility.

PATH Lists the directory (or directories, separated by semicolons) to be searched for executable files when the compiler is invoked. This variable should include the directories containing VisualAge C++ executables, for example the VisualAge C++ compiler (`icc.exe`) and linker (`ilink.exe`) executable files.

TMP Sets the directory where the VisualAge C++ compiler places all its temporary work files. This directory might also be used by other applications that generate temporary files. If this variable is undefined, the compiler uses the current directory. If you installed the compiler on a LAN, temporary files are stored in your local directory. The work files created by the compiler are normally erased at the end of compilation; however, if an interruption occurs during compiling, these work files may still exist after the compilation ends. If you set the `TMP` variable, you eliminate the possibility of work files being scattered around your file system.

You may be able to reduce compile time by setting TMP to point to a virtual disk (also called a RAM disk). See the OS/2 documentation for information on using the VDISK device driver to create a virtual disk.

Setting Environment Variables

Use the OS/2 SET command to give values to environment variables. Set the LIBPATH variable, and all DEVICE statements, in CONFIG.SYS. You can set other variables in any of three places:

CONFIG.SYS file

Add a line that sets the environment variable to the value you want. For example,

```
SET TMP=C:\IBMCPP\TMP
```

If the environment variable already exists in CONFIG.SYS, add the VisualAge C++ values to the existing variable. You can also have the VisualAge C++ installation program update CONFIG.SYS for you.

Environment variables specified in CONFIG.SYS are in effect for every session you start. This is a good place to specify variables that you want to apply each time you compile.

CSETENV.CMD file

The VisualAge C++ installation program creates this OS/2 command file. You must invoke this file in each session where you will use VisualAge C++. The variables set by CSETENV.CMD are in effect only for the session or sessions in which it is invoked.

If the installation program updated your CONFIG.SYS file, CSETENV.CMD contains commands to set the TMP and TZ variables only. If the installation did not update CONFIG.SYS, CSETENV.CMD contains statements to set up all the VisualAge C++ environment variables.

You can add environment variables of your choice to this file, to specify variables that you always use without having to type them individually on the command line. The variables in this file override environment variables in the CONFIG.SYS file. To avoid overriding values in CONFIG.SYS, append the original value of the variable using *%variable%*, as shown in this PATH statement:

```
SET PATH=C:\IBMCPP\BIN;%PATH%
```

command line

When the `SET` command is used on the OS/2 command line, the values you specify are in effect only for that OS/2 session. They override values previously specified in CONFIG.SYS or CSETENV.CMD. You can append the original value of the variable using *`%variable%`*.

Example The following example could be used in the CSETENV.CMD file or on the command line. If the executable files that make up the compiler are in `C:\IBMCPP\BIN`, the following command adds this directory to the `PATH` variable:

```
SET PATH=C:\IBMCPP\BIN;%PATH%
```

This command makes `C:\IBMCPP\BIN` the first directory searched by the OS/2 operating system (after the current directory). To put the new directory at the end of the search sequence, put `%PATH%` before the new directory name.

File Names in ICC

In addition to compiler options, you can also put file names into the ICC variable. For example, if you specify:

```
SET ICC=test.c check.c
```

the command

```
icc main.c
```

causes `test.c`, `check.c`, and `main.c` to be compiled and linked, in that order. You can also specify intermediate files (created with the `/Fw` option) in ICC. They are treated like source files.

If you specify library (`.LIB`), object (`.OBJ`), or module definition (`.DEF`) files in ICC, they are passed to the linker when the compiler invokes it.

Controlling #include Search Paths

The #**include** preprocessor directive allows you to retrieve source statements from secondary input files and incorporate them into your program.

You can nest #**include** directives in an included file. You can have up to 127 levels of nesting in a C file (128 including the main level), and up to 255 levels of nesting in a C++ file (256 including the main level), when using the VisualAge C++ compiler.

Compiler options and environment variables let you choose the disk directories searched by the compiler when it looks for #**include** files.

This section describes how to specify #**include** file names and how to set up search paths for these files.

#include Syntax

```
►►──#include──┬─<filename>─┬──►◄
              └─"filename"─┘
```

In the above figure, angle brackets indicate a **system #include** file, and quotation marks indicate a **user #include** file.

#include File Name Syntax

You can specify any valid OS/2 file name in a #**include** directive. The file name must be sufficiently qualified (have enough of a path) for the compiler to be able to locate the file. In some cases, an unqualified or partially qualified file name may be sufficient, for example:

```
#include "..\HEADERS\myheader.h"
```

In other cases, you may have to include the entire path name.

If a path name is too long to fit on one line, you can place a backslash (\) as a continuation character at the end of the unfinished line to indicate that the current line continues onto the next line. The backslash can follow or precede a directory separator, or divide a name. For example, to include the following file as a user #**include** file:

```
c:\cset\include\mystuff\subdir1\subdir2\subdir3\myfile.h
```

You could insert one of the following #**include** directives in your program:

```
#include "c:\cset\include\mystuff\subdir1\sub\
dir2\subdir3\myfile.h"
```

or

```
#include "c:\cset\include\mystuff\subdir1\\
subdir2\subdir3\myfile.h"
```

Notes:

1. The continuation character (\)must be the last non-white-space character on the line. (White space includes any of the space, tab, new-line, or form-feed characters.) The line cannot contain a comment.
2. The continuation character (\), although the same character as the directory separator, does not take the place of a directory separator or imply a new directory.

Ways to Control the #include Search Paths

You can control the #**include** search paths in three ways:

- Use the `/I,` `/Xc`, and `/Xi` compiler options on the command line when invoking the compiler.
- Use the `/I,` `/Xc`, and `/Xi` compiler options in the ICC environment variable.
- Specify the search paths in the INCLUDE environment variable.

For more information on the compiler options `/I,` `/Xc`, and `/Xi`, see "#**include** File Search Options" on page 274.

#include Search Order

When the compiler encounters either a user or system #**include** file statement with a fully qualified file name (full path and file name), it looks only in the directory specified by the name.

User #include Files

When the compiler encounters a user #**include** file specification that is not fully qualified, it searches for the file in the following places, in order:

1. The directory of the original top-level file.
2. Any directories specified using `/I` that have not been removed with `/Xc`. Directories specified in the ICC environment variable are searched before those specified on the command line.
3. Any directories listed in the INCLUDE environment variable, unless you specified the `/Xi` option.

System #include Files

When the compiler encounters a system #**include** file specification that is not fully qualified, it searches for the file in the following places, in order:

1. Any directories specified using `/I` that have not been removed with `/Xc`. Directories specified in the ICC environment variable are searched before those specified in the command line.
2. Any directories listed in the INCLUDE environment variable, unless you specified the `/Xi` option.

Accumulation of Options

The #**include** search options are cumulative between the ICC and INCLUDE environment variables and the command line. For example, given the following ICC and INCLUDE environment variables:

```
ICC=/I\rosanne
INCLUDE=c:\kent;\alan
```

and the following command line:

```
icc /Xi+ /Ic:\connie test.c /Xi- /Xc /Id:\dal f:\moe\marko\jay.c
```

The #**include** files are processed as follows:

User #include files

Any user #**include** files referenced in `test.c` are searched for first in the current directory, then in the directory `\rosanne`, and then in `c:\connie`. Because of the `/Xi` option, none of the directories in INCLUDE are searched.

Any user #**include** files referenced in `jay.c` will be searched for in the following directories, in the given order: `f:\moe\marko`, `d:\dal`, `c:\kent`, and `\alan`. The directories in INCLUDE are searched because the `/Xi-` option overrides the `/Xi+` option specified previously. The `/Xc` option removes the directories `\rosanne` and `c:\connie` from the current search path.

System #include files

Any system #**include** files referenced in the file `test.c` are searched for first in the directory `\rosanne` and then in the directory `c:\connie`. Because of the `/Xi+` option, none of the directories in INCLUDE are searched.

Any system #**include** files referenced in the file `f:\moe\marko\jay.c` will be searched for first in the `d:\dal` directory, then in the `c:\kent` directory, and finally the `\alan` directory. The directories in INCLUDE are searched because the `/Xi-` option overrides the `/Xi+` option specified previously. The `/Xc` option removes the directories `\rosanne` and `c:\connie` from the current search path.

Setting the Source Code Language Level

You can set the language level of your source code to one of the following levels:

ANSI Compile according to a strict interpretation of the ANSI standard. Use this level when you want your code to be portable to other compilers.

C **SAA Level 2** Compile according to the SAA Level 2 standard. Use this level when you want your code to be portable to other IBM compilers.

Extended Allow extended language features, allow non-standard language usage. Use this level when you are compiling code ported from another compiler, or when you are compiling code that does not need to be portable. If you will be compiling and running your code primarily on the personal computer platform, you should use the extended language level.

C++ **Compatible** Allow older versions of the C++ language. Use this level when you are compiling older code.

The levels are described in detail below. You can set the level using compiler options either on the command line or in ICC, or by using a **`#pragma langlvl`** directive. Note that a **`#pragma langlvl`** directive in your source code overrides any conflicting compiler options. When you set the language level, you also define the macro associated with that level. The SAA C standards conform to the ANSI standards, but also feature some additional elements.

ANSI

Allow only language constructs that conform to ANSI C standards or, for C++ code, that conform to the standards in the ANSI working paper on C++ standards. All non-ANSI constructs cause compiler errors.

Use this language level to write code that is portable across ANSI-conforming systems. If your code is error-free at this level, it should be error-free with any other compiler.

Note: Because VisualAge C++ has a strict interpretation of the ANSI standard, it can find errors in code that compiles cleanly with other compilers. If you are porting code **`into`** VisualAge C++, compile with the `Extended` language level.

To set this language level, use either the `/Sa` option or **`#pragma langlvl(ansi)`**, which define the macro `__ANSI__`.

SAA Level 2

C

Allow only language constructs that conform to SAA Level 2 C standards. This language level is valid for C code only, because there is no SAA standard for C++. SAA constructs include all those allowed under the ANSI language level, because the SAA C standard conforms to the ANSI standard. All non-SAA constructs cause compiler errors. This language level supports some additional library functions, and specifies some behaviors that are left as implementation-defined by the ANSI standard. 📖 See the *Language Reference* for a full description of the SAA C standard.

Use this language level to write code that is portable across SAA systems. If your code is error-free at this level, it should be error-free with any other IBM compiler.

To set this language level, use either the `/S2` option or **`#pragma langlvl(saal2)`**, which define the macro `__SAA_L2__`.

Note: You can also use **`#pragma langlvl(saa)`**, which defines the macro `__SAA__`. This level allows constructs that conform to the most recent SAA C definition. Because Level 2 is currently the most recent definition, the `__SAA__` and `__SAA_L2__` macros are equivalent at this time.

Extended

Allow all VisualAge C++ language constructs. These include all constructs that fall under the ANSI and SAA Level 2 language levels and the VisualAge C++ extensions to those standards. This level also allows certain OS/2 library functions. 📖 See the *Programming Guide* for more information.

Use the Extended language level when you are creating code that does not need to be portable, or when you are porting code into VisualAge C++ from another compiler or platform. Extended is the default language level.

To explicitly state this default (for example, on the command line to override a setting in ICC), use the /Se option or **#pragma langlvl(extended)**, which define the macro __EXTENDED__.

Compatible

C++ Allow constructs and expressions that were allowed by earlier levels of the C++ language. This language level is valid for C++ code only.

When the language level is set to compatible:

- Classes declared or defined within classes or declared within argument lists are given the scope of the closest non-class.
- typedefs and enumerated types declared within a class are given the scope of the closest non-class.
- The overload keyword is recognized and ignored.
- An expression showing the dimension in a delete expression is parsed and ignored. For example, given:

  ```
  delete [20] p;
  ```

  20 is ignored.
- Conversions from const void* and volatile void* to void* are allowed. At other language levels, these conversions would require an explicit cast.
- Where a conversion to a reference type uses a compiler temporary type, the reference need not be to a const type.
- You can bypass initializations as long as they are not constructor initializations.
- You can return a void expression from a function that returns void.
- operator++ and operator-- without the second zero argument are matched with both prefix and postfix ++ and --.
- You can use the $ character in identifiers. Note that you can also use $ in C++ files when the language level is set to extended.
- In a cast expression, the type to which you are casting can include a storage class specifier, function-type specifier (inline or virtual), template specifier, or typedef. At other language levels, the type must be a data type, class, or enumerated type.

- You can have a trailing comma in a list of enumerators, for example, `enum E {e , };`.
- Given the expression `class A *a = new(x) A[100];`, the compiler looks for a member operator `new` because the placement syntax (`new(x)`) is used. The member operators are not typically used to allocate arrays.
- You can use the comma operator in a constant expression. This allows comma expressions to be used in places like `case` labels and array bounds, where they are normally prohibited.
- You can declare a member function using both the `inline` and `static` keywords, for example, `inline static void sandra :: pete(void);`. The `static` keyword is ignored.
- No error is generated if a function declared to return a non-void type does not contain at least one return statement. Such a function can also contain return statements with no value without generating an error.
- If two pointers to functions differ only in their linkage types, they are considered to be compatible types.

Use this language level to write code that is portable to systems with older implementations of C++, or to port older code to the VisualAge C++ product.

To allow older C++ constructs, use the `/Sc` option or **`#pragma langlvl(compat)`**, which define the macro `__COMPAT__`.

Setting the Language Level

Compiler Output

14 Controlling Compiler Output

The VisualAge C++ compiler can generate the following output:

- An object module for each C/C++ source file input.
- One executable module (or dynamic link library).
- A listing file for each C/C++ source file that contains information about the compilation.
- Preprocessed header files.
- Template-include files. 📖 See the chapter on generating template-include files in the *IBM VisualAge C++ for OS/2 Programming Guide* for more information about these files.
- A linker map file.
- A preprocessor output file for each C/C++ source file. You can use this output file for debugging information.

  Note: This information is not intended to be used as a programming interface.
- An assembler listing file for each C/C++ source file. The format of the listing is in the style of the MASM 5.1 assembler input. The C/C++ source is annotated in the listing. Assembler listings will not always compile, especially if reserved MASM keywords are used as external variables or functions.

  Note: This listing is not intended to be used as a programming interface.
- A browser listing file for use by the VisualAge C++ browser.
- Intermediate code files. Four files (`.w`, `.wh`, `.wi`, `.ws`) are produced per source file.

  Note: These files are not intended to be used as a programming interface.
- Temporary files.

  Note: These files are not intended to be used as a programming interface.
- Diagnostic information about possible programming errors.

  Note: This information is not intended to be used as a programming interface.
- Messages (for example, the IBM logo and help messages).
- A return code (0 for a compile without errors).

Compiler Output

Object Files

The object files that are produced by VisualAge C++ compiler can be linked to create either executable (`.EXE`) files or dynamic link libraries (`.DLL` files). Use the `/Ge+` option to create an executable file or `/Ge-` to create a DLL. See "Code Generation Options" on page 299 for more information on using compiler options to specify the type of object file to be created.

Optimizing Object Code: VisualAge C++ compiler can perform many optimizations, such as local and global optimizations, function inlining, and instruction scheduling on object code. Use the `/O+` option to generate code that executes as fast as possible. By default, optimization is turned off (`/O-`). When you specify `/O`, you can also specify:

`/Oc` Optimize code for size as well as speed.

`/Os` Invoke the instruction scheduler. Turned on by default when you specify `/O`.

By default, `/O` also sets `/Oi` to inline user functions qualified with the **_Inline** or `inline` keywords.

The compiler can perform more complete optimization when you specify `/Ol`, to invoke the intermediate code linker.

Specify `/Gl` to perform additional optimization during the linking step, by removing unreferenced functions. See Chapter 17, "Optimized Linking" on page 333 for more information on linker optimizations.

See "Code Generation Options" on page 299 for more information on using compiler options to control optimization. For more information on how you can optimize your code, see the chapter on optimizing code in the *Programming Guide*.

Generating Debugger Information: The information necessary for running the VisualAge C++ Debugger can be placed in the object file produced by the compiler using the `/Ti+` option. To include the debugger information in the executable file or DLL, use the `/DE` linker option. If you use `icc` to invoke the linker and specify `/Ti+`, the `/DE` option is automatically passed to the linker.

When you use `/Ti+`, do **not** turn on optimization (`/O+`, `/Oc+`, `/Oi+`, or `/Os+`), unless you are using the information with the performance analyzer, and not with the debugger. Because the compiler produces debugging information as if the code were not optimized, the information may not accurately describe an optimized program being debugged, which makes debugging difficult. Accurate symbol and type information is not always available.

If you cannot avoid debugging an optimized program, turn the scheduler off (`/Os-`), and step through the program at the assembly level, using the Register and Storage windows for information.

To make full use of the VisualAge C++ Debugger, set optimization off and use the `/G3` option. (Note that these are the defaults.)

See “Debugging and Diagnostic Information Options” on page 281 for more information on using compiler options to control the generation of debugging information.

See Part 6, “IBM VisualAge C ++ Debugger” on page 393 for more information on the VisualAge C++ debugger.

Generating Performance Analyzer Information: To include the information required by Performance Analyzer in the object file, use both the `/Ti+` and `/Gh+` options. To include the Performance Analyzer information in the executable file or DLL, use the `/DE` linker option. If you use `icc` to invoke the linker and specify `/Ti+`, the `/DE` option is automatically passed to the linker.

When you specify `/Gh+`, the compiler generates a call to a profiling hook function as the first instruction in the prolog of each function. There are two profiling hook functions:

`_ProfileHook32` Profile hook for all 32-bit functions.

`_ProfileHook16` Profile hook for all 16-bit callback functions. These functions are defined with either the **_Far16 __cdecl** or **_Far16 _Pascal** linkage keywords.

Other profiler vendors who plan to support the VisualAge C++ product must provide their own profiling hook functions to gather all necessary runtime information.

Generating Browser Information: To create browser information, use the `/Fb+` option to produce .PDB files that the browser can use to display information about your program.

If you use `icc` to invoke the linker and specify `/Fb`, the `/BROWSE` option is automatically passed to the linker.

If you are compiling only, you must specify the `/BROWSE` option directly, when you link, to

See “Creating Files to Use with the Browser” on page 559 for more information on generating browser files, and the differences between `/Fb` and `/Fb*`.

Compiler Output

Executable Files

By default, the compiler generates one executable file for each compiler invocation. If you specify `/C+`, the compiler generates only object files, which you can then link separately to create an executable file.

There are two types of executable files:

- Those that run in the VisualAge C++ runtime environment.

  This is the default, and most C and C++ applications run under this environment. It supports all the VisualAge C++ runtime functions and automatically provides initialization, exception management, and termination routines for C and C++.

- Those that run as subsystems.

  Programs developed as subsystems can only make use of a subset of the VisualAge C++ runtime library. You have to take care of initialization, exception management, and termination using OS/2 services and APIs.

  Subsystems are intended for developing applications that cannot have a resident environment, such as PM display and printer drivers. If your application does not require the VisualAge C++ runtime environment, you can also use the subsystem library to reduce your program's size and improve its performance. To compile a subsystem executable file, use the `/Rn` option.

  For more information on subsystems and their uses, see the chapter on developing subsystems in the *Programming Guide*. For information on the compiler options used to produce subsystems, see “Code Generation Options” on page 299.

You can use several compiler options to change the executable file created by the compiler (see “Code Generation Options” on page 299 for more information).

Compiler Listings

When you compile a program, you can produce a listing file that contains information about the source program and the compilation. You can use this listing to help you debug your programs.

Note: The compiler listing file is not intended to be used as a programming interface.

At the very minimum, the listing will show the options used by the compiler, any error messages, and a standard header that shows:

- The product number
- The compiler version and release number
- The date and time compilation commenced
- A list of the compiler options in effect.

For information on how to use compiler options to specify the information and format of this file, see “Listing File Options” on page 276.

Temporary Files

The VisualAge C++ compiler creates and uses temporary files during compilation. Temporary files are usually erased at the end of a successful compilation; however, if the compilation is interrupted, these files may be left on the disk. They are located in the path specified by the TMP environment variable. If you use memory files and they overflow to the disk, they will also be located in the path specified by TMP. If this variable is undefined, the compiler uses the current directory. For more information on the TMP variable, see “OS/2 Environment Variables for Compiling” on page 207 and the chapter on run-time environment variables in the *IBM VisualAge C++ for OS/2 Programming Guide*.

Compilation time may be improved if you specify a virtual disk as the location for the temporary files.

Note: Do **not** copy the compiler executables onto a virtual disk. They are already preloaded.

See the OS/2 documentation for information on using the `VDISK` device driver to create a virtual disk.

Messages

You can use compiler options to control:

- The level of error message that the compiler outputs and that increments the error count maintained by the compiler (with the `/W`*`n`* option).
- How many errors are allowed before the compiler stops compiling (with the `/N`*`n`* option).
- The diagnostics run against the code (with the `/W`*`grp`* option).

See the online *User's Guide* for a list of compiler error messages.

See “Debugging and Diagnostic Information Options” on page 281 for more information on using the compiler options to control messages.

Return Codes

VisualAge C++ compiler returns the highest return code it receives from executing the various phases of compilation. These codes are:

Code **Meaning**

0 The compilation was completed, and no errors were detected. Any warnings have been written to **stdout**. Your executable file should run successfully.

12 Error detected; compilation may have been completed; successful execution impossible.

16 Severe error detected; compilation terminated abnormally; successful execution impossible.

20 Unrecoverable error detected; compilation terminated abnormally and abruptly; successful execution impossible.

If the error code is greater than 20, contact your IBM service representative.

For every compilation, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved.

Precompiled Header Files

You can use the `/Fi+` compiler option to create or recreate precompiled versions of header files used during that compilation.

To use the precompiled header files, specify the `/Si+` option. You can specify a name for the precompiled header object and a directory. If you do not specify a name or directory, the precompiled header files are stored in the current working directory, with the name `csetc.pch` (for C files) or `csetcpp.pch` (for C++ files).

For more information on generating and using precompiled headers, see "Using Precompiled Headers" on page 239.

Note: In the previous version of C Set ++, it was possible to use the same precompiled header file for both C and C++ files. This is no longer possible with VisualAge C++ version 3.0.

When you use precompiled header files, the following restrictions apply:

- You cannot use the same precompiled header file for C and C++ programs.
- To create a precompiled header file, the compiler process must have write permission to the directories you specify, or to the current working directories if none are specified. To use a precompiled header, the compiler process must have read permission for that file.
- Precompiled header files do not appear in any listing files.
- If you specify `/P+` to run the preprocessor only, the `/Fi` and `/Si` options are ignored.

Using the Intermediate Code Linker

The intermediate code linker combines the information in all `.w`, `.wh`, `.wi`, and `.ws` intermediate code files into one set of files which is then used by the compiler to optimize the code and generate a single object file.

In addition to the optimizations performed by the intermediate linker itself, using this linker exposes more of your program to the optimizer at a time. The optimizer can then generate more efficient code. Using the intermediate linker can result in improved code optimization, especially where inlining is used, and better program performance. Note that using the intermediate linker on code being compiled into an .EXE file results in better performance improvements than if the same code were being compiled into a .DLL file.

The intermediate linker also performs more thorough error checking than the compiler can perform on its own. See “Error Checking” on page 228 for more information.

To use the intermediate linker, specify the `/Ol+` option on the `icc` command line. For best results, use the `/Gu` option, as described in “Using the `/Gu` Option” on page 227, and specify `/O+` to turn optimization on.

Note: Because optimization limits the generation of debugging information, use `/O-` if you want to debug your program. The `/Ol` option does not affect debugging information.

Given the following command:

```
icc /O+ /Ol+ vij.c thomas.c tim.c
```

the compiler:

1. Compiles each source file into a set of intermediate code files (`.w, .wh`, `.wi`, and `.ws` files).

2. Invokes the intermediate code linker to link the intermediate code files of all three source files.
3. Optimizes the code.
4. Creates **one** object module for all three files and names it after the first file specified on the command line (vij.obj). (You can change the name of the object file using the /Fo option.)
5. Invokes the linker to create an executable module (vij.exe). (You can change the name of the executable file using the /Fe option.) If you want to link your object files separately, use the /C+ option on the icc command line. You can then invoke the linker as you would for any other object file.

Intermediate code files

Instead of creating an object file directly, you can use the /Fw+ option to create and save the intermediate code files to be linked by the intermediate linker at a later time. When you use /Fw+, compilation stops when the intermediate files are created. For example:

```
icc /Fw+ brian.c jim.c
```

creates only the intermediate files for brian.c and jim.c. No object or executable modules are created.

The /Fw option also takes an optional file-name parameter that lets you specify the file name for the intermediate files. For example:

```
icc /Fwtony jeff.c
```

names the resulting intermediate files for jeff.c to tony.w, tony.wh, tony.wi, and tony.ws. Note that there is no space allowed between /Fw and the file-name parameter.

You can specify existing intermediate files on the icc command line to run the intermediate linker and complete the compilation. You need only specify the name of the .w file; the .wh, .wi, and .ws files are included automatically. No option is required. For example, the command:

```
icc brian.w jim.w
```

links all intermediate files for brian.c and jim.c, creates an object file, and invokes the linker to create an executable module.

Note: You cannot use compiler options related to source files with intermediate files because the source has already been partially compiled. For example, you cannot produce a listing file from intermediate files or set the language level for the program.

You can also combine intermediate and source files on the command line to run the intermediate linker on all the files and complete the compilation. No option is required. For example:

```
icc brian.w jim.c
```

Restrictions

Consistent Options

When you use the intermediate linker, some options must be consistent across all files. Without the intermediate linker, these options can be inconsistent, allowing you to compile different object files with different options. Because the intermediate linker creates only one object file, you can no longer have these different options in effect. The following options must be consistent across all source files, when you use the intermediate code linker:

```
/G3                 /Gr                /O
/G4                 /Gs                /Oc
/G5                 /Gt                /Oi
/Ge                 /Gv                /Om
/Gf                 /Gw                /Op
/Gh                 /Re                /Os
/Gi                 /Rn                /Ti
/Gp                 /Nd                /Tn
```

System Requirements

If you use the intermediate code linker on a large application, you will require more system resources than if you were simply compiling. For example, compiling and intermediate linking a 40 000-line application requires a working set of approximately 25M. If your executable module or DLL contains more than 100 000 lines of code, using the intermediate code linker is not recommended.

Using the /Gu Option

One of the optimizations performed by the intermediate linker is to discard any defined data or functions that are:

- Not referenced in the files included in the link.
- Not defined as exports either by the **_Export** keyword, by #**pragma** export, or in the .DEF file. (Note: If you define exports in the .DEF file, you must include the file name in the icc command line.) Export functions when you are creating a DLL, and want its functions to be available to other DLLs or .EXE files.

If you call functions in files not included in the intermediate link, such as library functions or OS/2 APIs, this optimization cannot be performed because the data and functions could possibly be used by one of these external functions. Because library functions and APIs rarely use data defined in user code, the result is often poorly optimized code.

To ensure that all unreferenced data and functions are discarded, use the `/Gu+` option. This option tells the intermediate linker that any external functions that are referenced will not use anything defined in the files being linked. Use the **_Export** keyword to mark any definitions that will be used in a separate compilation unit (by another DLL, or .EXE file).

In addition, `/Gu+` causes all external functions and data that are not exported to be defined as `static`, which can result in better optimization.

Error Checking

Another benefit of using the intermediate code linker is enhanced error checking of all files included in the intermediate link step. The intermediate linker can find errors that would otherwise generate linker errors or unexpected runtime behavior, such as:

- Redefinition of variables and functions
- Inconsistent declarations or definitions of the same function (including differences in return type, linkage, number of arguments, and argument properties)
- Type mismatches between different declarations or definitions of the same variable, with the exception of:
  - Differences in integer type of the same length (`int` and `long`)
  - Some mismatches within structures and unions
  - Mismatches between array declarations where one of the declarations is an external reference

The intermediate linker also checks for inconsistent compiler options that could cause conflicts. If the intermediate linker warns you of conflicting options, either change the options to make them consistent or, if it is necessary to use the options inconsistently, turn off the intermediate code linker (`/Ol`). 📖 See "Restrictions" on page 227 for a list of options that must be consistent across source files.

The intermediate linker generates compiler errors `EDC6004` through `EDC6024` 📖 See the online *User's Guide* for explanations of error messages.

Inlining User Code

By default, the compiler inlines certain library functions, meaning that it replaces the function call with the actual code for the function at the point where the call was made. These library functions are called intrinsic or built-in functions.

You can also request that the compiler inline the code for your own functions. There are two ways to inline user code:

1. Use the **_Inline** keyword to specify which functions you want to have inlined. You must specify the `/Oi` option to turn inlining on.

   C++ The C++ language provides the function specifier `inline` that you can use in the same manner as **_Inline**. The **_Inline** keyword is not supported for use in C++ programs.

2. Use the `/Oi` option with a *`value`* parameter to automatically inline functions smaller than the value specified.

Note: Requesting that a function be inlined makes it a candidate for inlining but does not necessarily mean that the function will be inlined. In all cases, whether a function is actually inlined is up to the compiler.

Using Keywords

C For C files, use the **_Inline** keyword to qualify either the prototype or definition of the functions you want to have inlined. For example:

```
_Inline int james(int a);
```

specifies that you want `james` to be considered for inlining.

C++ In C++ files, use the `inline` function specifier in the same way as **_Inline**. For example:

```
inline int angelique(char c);
```

specifies that you want `angelique` to be considered for inlining.

The **_Inline** and `inline` keywords have the same meaning and syntax as the storage class `static`. When you turn inlining on, the keywords also cause the functions they qualify to be considered for inlining. In addition, C++ member functions that are defined in a class declaration are considered candidates for inlining by the compiler.

Inlining User Code

Using the /Oi Option

The /Oi option controls whether user functions are inlined or invoked through a function call:

/Oi- Do not inline user code. This is the default.

/Oi+ Inline functions qualified with the **_Inline** or inline keyword. When optimization is turned on (/O+), /Oi+ becomes the default.

/Oi*value* Inline functions qualified with the **_Inline** or inline keyword, as well as other functions that are smaller than or the same size as *value* in abstract code units (ACUs) as measured by the compiler. This option is called auto-inlining. In general, choosing the functions you want inlined yields better results than auto-inlining.

The /Oi option only affects user code, and does not affect the inlining of intrinsic VisualAge C++ library functions. To disable the inlining of library functions, parenthesize the function call. For example:

```
   (strcpy)(str1, str2);
```

In this way, you can selectively disable inlining of VisualAge C++ functions.

Some library functions are implemented as built-in functions, meaning there is no backing code in the library. You cannot parenthesize calls to these functions.

See the *C Library Reference* for a list of all the intrinsic and built-in library functions.

You cannot selectively disable inlining for user functions: you can request that a function be inlined, but you cannot turn inlining on and then request that a specific user function **not** be inlined.

If you use auto-inlining, *value* has a range between 0 and 65 535 ACUs (abstract code units). The number of ACUs that constitute a function is proportional to the size and complexity of the function. Because the compiler calculates ACUs based on internal algorithms, you can only estimate the number of ACUs for a given function. The following code samples provide some examples on which you can base your estimates.

The following function is 33 ACUs:

```
int florence(char a, int b)
{
   if(a != 10)
      b++;
   else
      b += 10;
   return(a);
}
```

The next function is 51 ACUs:

```
int sanjay(long par1, long par2)
{
   while(par1)
   {
      if(par2)
         test3();
      par1--;
   }

   if(par1)
      testing();
   par1 += par2;
}
```

When you compile, the compiler generates a message for each function in inlines based on the *value* you specified. Messages are not generated for functions qualified with **_Inline** or `inline`, or for C++ functions defined in a class declaration. For most applications, the most effective *value* for auto-inlining is between 5 and 20.

Note: The *value* required to inline a specific function may be slightly larger when `/O+` is specified than when `/O-` is specified.

When you turn inlining on for C programs, `static` functions that are called only once and are relatively small (16 ACUs or less) are also inlined. For this type of function, there is always a greater benefit in inlining. You can use `/Oi`*value* with a very small value to display the names of these functions. They are not inlined for C++ programs.

Inlining User Code

Benefits of Inlining

Inlining user code eliminates the overhead of the function call and linkage, and also exposes the function's code to the optimizer, resulting in faster code performance. Inlining produces the best results when:

- The overhead for the function is significant; for example, when functions are called within nested loops.
- The inlined function provides additional opportunities for optimization, such as when constant arguments are used.

For example, given the following function:

```
void glen(int a, int b)
{
   if (a == 10)
   {
      switch(b)
      {
         case 1:  .
                  :
         case 20: puts("b is 20");
                  break;
         case 30: .
                  :
         default: .
                  :
      }
   }
}
```

and assuming your program calls `glen` several times with constant arguments, for example, `glen(10, 20);`, each call to `glen` causes the `if` and `switch` expressions to be evaluated. If `glen` is inlined, the compiler can then optimize the function. The evaluation of the `if` and `switch` statements can be done at compile time, and the function code can then be reduced to only the `puts` statement from `case 20`.

The best candidates for inlining are small functions that are called often. Use Performance Analyzer or a profiler to determine which functions to inline to obtain the best results.

To improve performance further:

- Use constant arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- If you have a function that is called many times from a few functions, but infrequently from others, create a copy of the function with a different name and inline it only in the functions that call it often.
- Turn optimization on.

Drawbacks of Inlining

Inlining user code usually results in a larger executable module because the code for the function is included at each call site. Because of the extra optimizations that can be performed, the difference in size may be less than the size of the function multiplied by the number of calls.

Inlining can also result in slower program performance, especially if you use auto-inlining. Because auto-inlining looks only at the number of ACUs for a function, the functions that are inlined are not always the best candidates for inlining. As much as possible, use the **_Inline** or `inline` keyword to choose the functions to be inlined.

When you use inlining, you need more stack space. When a function is called, its local storage is allocated at the time of the call and freed when it returns to the calling function. If that same function is inlined, its storage is allocated when the function that calls it is entered, and is not freed until that calling function ends. Ensure that you have enough stack space for the local storage of the inlined functions.

Restrictions on Inlining

The following restrictions apply to inlining:

- You cannot inline functions that use a variable number of arguments.
- You cannot inline functions with **_System** linkage that make use of the `__parmdwords` function.
- C++ For C++, you cannot declare a function as `inline` after it has been called.

Inlining User Code

- To use **_Inline** or `inline`, the code for the function to be inlined must be in the same source file as the call to the function. To inline across source files you must either:

  1. Place the function definition (qualified with **_Inline**) in a header file that is included by all source files where the function is to be inlined.
  2. Use the intermediate code linker (with the `/Ol+` option) and auto-inlining. The intermediate code linker is described in “Using the Intermediate Code Linker” on page 225.

- Turn off inlining (`/Oi-`) if you plan to debug your executable module. Inlining can make debugging difficult; for example, if you set an entry breakpoint for a function call but the function is inlined, the breakpoint will not work.

- Performance Analyzer treats an inlined function as part of the function in which it is inlined.

- A function is not inlined during an inline expansion of itself. For a function that is directly recursive, the call to the function from within itself is not inlined. For example, given three functions to be inlined, `A`, `B`, and `C`, where:

  – `A` calls `B`
  – `B` calls `C`
  – `C` calls back to `B`

  the following inlining takes place:

  – The call to `B` from `A` is inlined.
  – The call to `C` from `B` is inlined.
  – The call to `B` from `C` is not inlined because it is made from within an inline expansion of `B` itself.

Setting the Calling Convention

The VisualAge C++ compiler supports six 32-bit calling conventions, and three 16-bit conventions:

32-bit: **_Optlink**
_System
__cdecl
__stdcall
_Pascal
_Far32 _Pascal

16-bit: **_Far16 __cdecl**
_Far16 _Pascal
_Far16 _Fastcall

C

The **_Far32 _Pascal** convention can only be used in C programs and only when the /Gr+ option is specified.

The default is **_Optlink** for calls to 32-bit code. You must explicitly specify a calling convention for all 16-bit calls. If you specify only **_Far16**, the convention defaults to **_Far16 __cdecl**. You can change the default for 32-bit code with the /M option:

Option	**Calling Convention**
/Ms	**_System** calling convention
/Mc	**__cdecl** calling convention
/Mt	**__stdcall** calling convention
/Mp	**_Optlink** calling convention (the default)

See “Code Generation Options” on page 299 for more information on these compiler options.

You can also set the calling convention for individual functions using linkage keywords.

For example, to declare kathryn as a function with the **_System** calling convention, you could use the following statement:

```
int _System kathryn(int i);
```

You can also use **#pragma** directives to set the calling convention for C programs, but this is obsolete and may not be supported in future versions of the compiler.

For example:

```
#pragma linkage(kathryn, system)
```

Note that, when using the #**pragma** `linkage` directive, you must declare the function separately. Using linkage keywords is generally quicker and easier than using #**pragma** `linkage` directives.

Both the keywords and the #**pragma** `linkage` directive take precedence over a conflicting compiler option. If you use both methods and specify different conventions for the same function, an error message is generated.

Note: You cannot change the calling convention for C++ member functions. Member functions always use the **_Optlink** convention.

The linkage keywords and #**pragma** `linkage` directive are described in more detail in the *Language Reference*. For more information on the calling conventions and how they work, see the *Programming Guide*.

Choosing Your Runtime Libraries

When you compile, the compiler defines default VisualAge C++ runtime libraries for the linker to use. You can use compiler options to control the linking process by changing the type of runtime library you link to. If you do not specify any options, the compiler uses the library that produces single-thread executable modules that are statically linked. You can link to another library by specifying the appropriate options. You would link to another library to:

- Dynamically link your program (discussed in the following section).
- Create a multithread executable module. (📖See the *Programming Guide* for more detailed information.)
- Develop a subsystem. (📖See the *Programming Guide* for more detailed information.)
- Create a DLL for use with another executable module. (📖See the *Programming Guide* for more detailed information.)

The naming conventions used for the libraries are intended to help identify their function. The libraries are named as follows:

Figure 67. VisualAge C++ Library Naming Conventions

Character Position				**Significance**
1 - 4	**5**	**6-7**	**8**	
CPPO				Product prefix
	S M N			Single-thread library Multithread library Subsystem library (no runtime environment)
		30		Version of VisualAge C++
			I O	Import library Object library (contains initialization routines) Statically bound library (no eighth letter)

For example, the library CPPO30S.LIB is the standard single-thread library for building both executable modules and DLLs, while CPPON30I.LIB is the standard import library for creating a subsystem.

For a list of all libraries and files shipped with the VisualAge C++ product, see the appendixes of the *Programming Guide*.

Static and Dynamic Linking

Static linking means that code for all the VisualAge C++ runtime functions called in your program is copied from a .LIB file into your output .EXE or .DLL file. The .EXE or .DLL files will be larger because there is a copy of the runtime functions in each file. These programs will take up more storage, and if you run them at the same time, there will also be a copy of the library functions in memory for each program. Statically linked programs, however, are easier to distribute because the library functions are part of your executable file. See Note 1 below.

Dynamic linking means that code for the VisualAge C++ runtime functions called in your program is **`not`** copied into your output .EXE or .DLL file. Instead, the function code stays in a separate VisualAge C++ DLL file, and your calls to the function are resolved at load time. The amount of disk space required by your .EXE or .DLL file is reduced, and there is only one copy of the library functions in memory for all programs that use them. Dynamically linked programs can be harder to distribute, since the separate DLL file must be distributed along with your executable file.

Use the `/Gd` compiler option to control whether your executable file links to the runtime library statically or dynamically.

Choosing Runtime Libraries

The default is `/Gd-`, which statically links with the .LIB version of the runtime library.

Specify `/Gd+` to dynamically link with the DLL version of the runtime library.

The compiler option you choose causes the corresponding library to be linked in by default. If you override the default libraries with the `/NOD` linker option, you must explicitly give the name of all libraries you are using on the linker command line.

You can also statically or dynamically link to other libraries. For more information, see Chapter 19, "Linking with Library Files" on page 343.

Under the VisualAge C++ licensing agreement, you cannot ship the VisualAge C++ DLLs as they are with a product that you develop. If you want to dynamically link to the VisualAge C++ library, you can create your own version of the VisualAge C++ runtime DLLs, as described in the *Programming Guide*, or you can use the DLLRNAME utility, described on page 675, to rename the VisualAge C++ DLLs before you ship them.

Notes:

1. When you use static linking, all external names beginning with `Dos`, `Vio`, or `Kbd` (in the case shown) become reserved external identifiers. They are not reserved if you use dynamic linking.
2. You can also link dynamically to your own DLLs. Creating and using your own DLLs is discussed in "Producing a Dynamic Link Library" on page 339, "Linking to Dynamic Link Libraries" on page 345, and in the *Programming Guide*.

Using the Multithread Library

More than one thread may use the same runtime functions. To avoid contention for internal resources, the library ensures that only one thread at a time is active in the critical section of a function. Although this support is mandatory in a multithread program, it is unnecessary in a single-thread program.

This section describes only the compiler options you use to choose the single-thread or multithread version of the library. There is more information on creating a multithread program in the *Programming Guide*.

If you want to create an executable file with multithread capabilities:

1. Specify the `/Gm+` option when you compile.
2. Use the multithread library when you link the object files.

If you want to create an executable file designed for a single thread only:

1. Use the default option `/Gm-` when you compile.
2. Use the single-thread library when you link the object files.

The compiler option you choose causes the corresponding library to be linked in by default. If you override the default libraries with the `/NOD` linker option, you must explicitly give the name of all libraries you are using on the linker command line.

Enabling Subsystem Development

If you are creating a subsystem (for example, a PM display or printer driver), specify the `/Rn` option to select the subsystem libraries. See page 312 for a description of the option.

Functions in the subsystem libraries are intended for use in single-thread applications only. No multithread support is provided. If you want to use the subsystem libraries in multithread programs, you must provide your own protection and serialization using OS/2 semaphores. You must also provide your own buffering for input and output.

See the *Programming Guide* for information on developing subsystems.

Using Precompiled Headers

You can improve your compile time by using precompiled headers. Use the options `/Fi+` and `/Si+` together to automatically create and maintain precompiled header files for your application.

Note: In the previous version of C Set ++, it was possible to use the same precompiled header file for both C and C++ files. This is no longer possible with VisualAge C++ version 3.0.

If you use the options consistently, precompiled header files are created if they do not exist, and used if they do. When a source file is changed, the precompiled version is automatically regenerated.

In previous versions of C Set ++, each header was precompiled separately. This meant that the precompiled headers still had to be interpreted by the compiler, to allow for the context in which they were being included (for example, different **`#define`** directives that might be in effect).

The compiler now generates a single precompiled object for the first **initial sequence** of **`#include`** directives. The next time you compile, this single object can be used wherever that initial sequence appears. Since the precompiled object is only used in cases where the context is the same (same language, same beginning sequence of

#include directives, compatible options and macro definitions), the precompiled object does not have to be re-interpreted every time it is included.

To get the most benefit from this new method, use the same initial sequence of headers wherever possible. The more files that share the same initial sequence, the greater the improvement in your compile time. 📖 See "Organizing Your Source Files" on page 245 for tips on getting the most improvement.

You can specify different names or directories for precompiled header files, with the `/Fi` and `/Si` options, or using **`#pragma hdrfile`**. This allows you to create more than one initial sequence, and further improve your compile time. If you do not specify a name or directory, the precompiled headers are stored in the current working directory, with the name `csetc.pch` (for C files), or `csetcpp.pch` (for C++ files).

Note: In the previous version of C Set ++, it was possible to use the same precompiled header file for both C and C++ files. This is no longer possible with VisualAge C++ version 3.0.

Determining the Initial Sequence

The initial sequence of headers can consist of the following:

- **#include** directives
- comments
- **#error** directives
- null directives
- false conditional compilation blocks beginning with **#elif** or **#else**.
- **#endif** directives

The first **#include** directive can be preceded only by comments and preprocessing directives. If it is preceded by anything else, then the compiler does not create or attempt to use precompiled headers with that source file.

The initial sequence is ended by any construct not in the above list. You can also stop the initial sequence with **#pragma hdrstop**. If you use **#pragma hdrstop** before the first **#include** directive, there is no initial sequence.

Any **#include** directives after the initial sequence are not precompiled: they will be compiled every time you compile the source file.

Note: When a header contains conditional compilation directives to prevent it from being included a second time, it is only counted once in the initial sequence, even if it appears multiple times.

Example

Given the following code:

```
main.c                       h1.h
------------------------     -----------------
/* Comments are OK */        int h1;
#define M 1                  #include "h3.h"
#undef  N
#line 10
#if F
  int f(int);                h2.h
#endif                       -----------------
#if STDIO                    int h2;
  #include <stdio.h>
#endif
#include "h1.h"
/* Comments are OK */        h3.h
#                            -----------------
#include "h2.h"              #ifndef H3_H
#include "h3.h"              #define H3_H
main() {                       int h3;
}                            #endif
```

The initial sequence can vary, depending on whether any macros are defined on the command line.

Macros defined	Resulting initial sequence
None	`"h1.h", "h2.h", "h3.h"`
STDIO	`<stdio.h>, "h1.h", "h2.h", "h3.h"`
F	No initial sequence (because `int f(int)` occurs before any **`#include`** directives)

Although `h3.h` is included twice (once in `main.c` and once in `h1.c`), only the first **`#include`** is considered in the initial sequence, because the second **`#include`** does not take effect.

Matching the Initial Sequence

Once the precompiled initial sequence is created, it can be used by other compilation units (in the next compile, and in subsequent compiles). Other compilation units can use the precompiled initial sequence under the following conditions:

- The compilation unit has a matching initial sequence of **`#include`** directives. The compilation unit can have a longer initial sequence, as long as the first part of the sequence matches. Any **#include** directives beyond the initial matching portion are compiled normally.

- The files that make up the precompiled header object have not changed. The compiler checks the modification date on each file.
- Any macros that were expanded or tested while generating the precompiled header object are defined with the same replacement tokens. The compiler checks macro names that are:
  - Defined before the start of the initial sequence, using **#define** or the `/D` option.
  - Undefined before the start of the initial sequence, using **#undef** or the `/U` option.
  - Predefined by the compiler.

  If the macro was not expanded or tested during the precompile, then the status of the macro does not matter, and does not have to match.
- No additional macros have been defined.
- The same compiler options are in effect.

Example

Given two compilation units, `prog1.c` and `prog2.c`:

```
prog1.c                        prog2.c
-----------------------        -------------------------
#undef X                       #define X 1
#include "h1.h"                #include "h1.h"
#include "h2.h"                #include "h2.h"
func1() {}                     func2() {}

h1.h
----------------------
#if TEST
  int h1;
#endif

h2.h
----------------------
char h2 = M;
```

The file prog2.c can use the precompiled header object from prog1.c when:

- TEST has the same definition in both prog1.c and prog2.c, or is not defined in both.
- M has the same definition in both prog1.c and prog2.c, or is not defined in both.
- No additional macros have been defined in prog2.c (whether they are used or not).

The different definitions of X in prog1.c and prog2.c do not matter, since X is never tested or expanded.

Using Multiple Initial Sequences

Because of the restrictions on reusing precompiled headers (same sequence of headers, same context in terms of macro names and options), you may want to use more than one precompiled header object.

You can specify the name of an alternate precompiled header file to use, or an alternate directory to search, with either of two methods:

- With the options /Fi+ and /Si+ on the command line. If the options specify different file names, the compiler uses the last one specified for both options.
- With **#pragma hdrfile** in the source file, before the first **#include** directive. The pragma only takes effect if you specify at least one of /Fi or /Si.

If you specify a file name with both the options and the **#pragma**, the **#pragma** file name is used.

The default file names are:

- csetc.pch for C compiles
- csetcpp.pch for C++ compiles

The default directory is the current working directory.

Examples The following examples show the interaction of options and **`#pragma hdrfile`** directives:

Example 1

```
#pragma hdrfile "fred.pch"
#include "h1.h"
#include "h2.h"
main () {}
```

Options Specified	**Behavior**
`/Fi+ /Si+`	The headers "h1.h" and "h2.h" are precompiled using the file "fred.pch"
`/Fidave.pch`	The compiler ignores the name specified with the option, because it is in conflict with the **`#pragma hdrfile`** directive. The compiler generates new headers in "fred.pch", but does not use them.

Example 2

```
#pragma hdrfile "fred.pch"
#include "h1.h"
#pragma hdrstop
#include "h2.h"
main () {}
```

Options Specified	**Behavior**
`/Fi+ /Si+`	Only the header "h1.h" is precompiled, using the file "fred.pch"
`/Sidave.pch`	The compiler ignores the name specified with the option, because it is in conflict with the **`#pragma hdrfile`** directive. The compiler looks for the precompiled headers in "fred.pch", but does not generate new headers.

Example 3

```
#include "h1.h"
#pragma hdrstop
#include "h2.h"
main () {}
```

Options Specified	Behavior
`/Fi+ /Si+`	Only the header "h1.h" is precompiled, using the file "csetc.pch" (for a C file) or "csetcpp.pch" (for a C++ file)
`/Sidave.pch /Fijohn.pch`	The compiler ignores the name specified with /Si, and uses the file "john.pch", which is specified later. The compiler looks for precompiled headers in "john.pch" and regenerates them if they are not found or are out of date.

Example 4

```
#pragma hdrstop
#include "h1.h"
main () {}
```

Options Specified	Behavior
`Any options`	No headers are precompiled, because there is no initial sequence: **`#pragma hdrstop`** occurrs before the first **`#include`** directive.

Organizing Your Source Files

To take full advantage of the precompiled header improvements, you may need to reorganize your source files. Using precompiled headers without organizing your source files can actually **slow** your compile.

There are several strategies you can use to organize your files:

A precompiled header for each compilation unit

Use **`#pragma hdrfile`** in each primary source file to specify a distinct precompiled header object for each compilation unit.

Benefits
Each compilation unit has its own precompiled headers, so you can create the longest possible initial sequence for each compilation unit. You do not need to match the initial sequences in other compilation units, because you are not sharing the precompiled header object.

Drawbacks
If you change one header file, the compiler regenerates every precompiled header object that includes that header. This method can also require large amounts of disk space, since many precompiled headers are generated: a common header is precompiled separately for every compilation unit that includes it.

Single header file

Create a single header file which has **`#include`** directives for every header in your application, and include it in each primary source file.

Benefits
You get the maximum possible benefit from using precompiled header files, and the maximum possible improvement in compile time. You have an exact match of the initial sequence for every compilation unit.

Drawbacks
If you are using a program maintenance utility such as NMAKE, you will end up recompiling the entire application every time you change even a single header file. For larger applications, this is probably unacceptable.

Global header file

Create a single header file which has **`#include`** directives for those header files that are shared by many different compilation units. **`#include`** this global header file as the first step of the initial sequence in each primary source file, followed immediately by **`#pragma hdrstop`**.

Benefits
You get the benefit from precompiling common and shared headers, without having to recompile when you change less common headers.

Drawbacks
There is only one group of precompiled headers. Any headers outside of that group are compiled normally, and the precompiled header object must be regenerated every time you change even one of the common headers.

Grouping headers

Do the following:

1. Identify headers that are common throughout your source, and divide them into smaller groups of associated headers. A header can be included in more than one group.

2. For each group, create a header that contains **`#include`** directives for each header in the group.

3. In each primary source file, identify the precompiled header file name with **#pragma hdrfile**, **#include** the appropriate group header, and end the initial sequence with **#pragma hdrstop**. If a source file does not use any of the precompiled header groups, put **#pragma hdrstop** as the first directive in the file.

Benefits
Common headers are precompiled. Changing a header only affects the groups it belongs to, rather than requiring all precompiled headers to be regenerated. This method does not use as much disk space as having a precompiled header object for each compilation unit, and is more maintainable than a single global header file.

Drawbacks
Requires additional work to organize headers into groups.

Controlling the Logo Display on Compiler Invocation

By default, the VisualAge C++ logo appears on **stderr** when the compiler is invoked. You can stop the logo from appearing on icc invocation by specifying the /Q+ option. To request explicitly that the logo appear, specify the /Q- option.

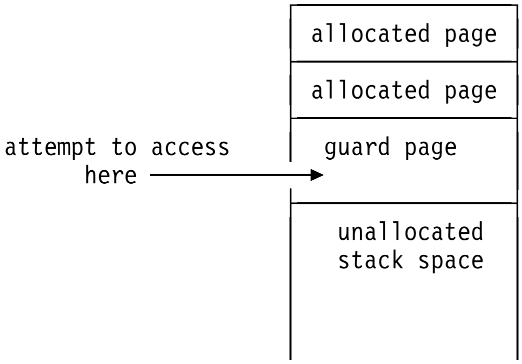
Controlling Stack Allocation and Stack Probes

Under the OS/2 operating system, the stack is fully allocated for the first thread of a process. For all threads other than the first, the operating system allocates the stack as a sparse object. The total stack size is rounded up to the nearest multiple of 4K from the size you specified. The page with the largest address is committed, and the page below it is set up as a *guard page*. No other pages are committed.

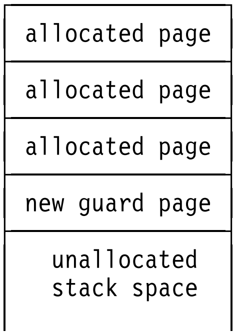
When the guard page is accessed, an *out of stack* exception (XCPT_GUARD_PAGE_VIOLATION) is generated. The system responds by attempting to get another guard page below the one previously allocated:[1]

[1] For the purposes of this discussion, the stack grows down.

Controlling Stack Allocation and Stack Probes



If this attempt is successful, the original guard page becomes a normal stack page and the next uncommitted page becomes the new guard page.



This process continues until a new guard page can no longer be allocated.

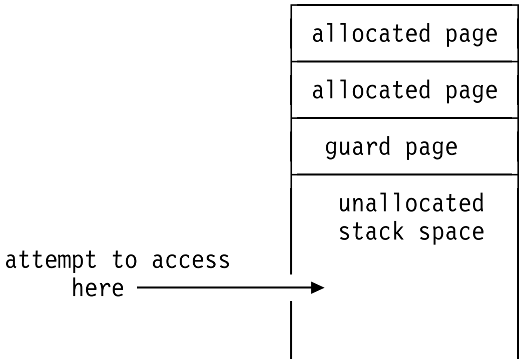
If the system cannot set a new guard page because it has reached the size limit of the stack (📖 see "Setting the Stack Size" on page 250), a *guard page allocation failure* exception (`XCPT_UNABLE_TO_GROW_STACK`) is generated. (The same exception is generated when the `_alloca` function runs out of memory.)

Note: The last 4K of the stack (the final guard page) is reserved to allow handling of exception conditions. If a guard page exception occurs and not enough stack remains to handle the exception, the program is terminated. For more information about exceptions and error handling, 📖 see the *Programming Guide*.

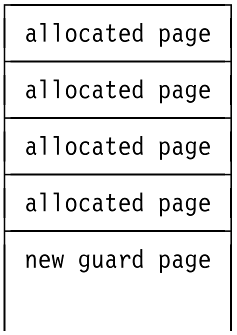
Using Stack Probes

For the stack growth mechanism to work correctly, each 4K page must be accessed in the correct order. To ensure the correct access, VisualAge C++ compiler generates one or more stack probes in the prolog of each procedure that has automatic storage greater than 2K. (Stack probes start after 2K because exception handling may require up to 2K of stack storage.)

When a guard-page exception occurs, the stack probe instructions allow the exception mechanism to enlarge the stack if necessary. If an attempt is made to access the stack below the guard page:



stack probes cause the operating system to allocate each page of the stack up to that access point and to create a new guard page:



Without stack probes, accessing the stack below the guard page is an access violation (you cannot access uncommitted pages). The process terminates. The compiler ensures that structures greater than 4K that are passed by value are placed on the stack to allow this mechanism to work.

Support for automatic stack growth is provided by default as needed.

Setting Stack Size

Note: The _alloca function allocates storage on the stack. Unless you specify the /Gs+ option, the compiler generates stack probes to allocate the required memory.

You do not need to use stack probes if:

- Your program has only one thread. The stack is fully allocated for the first thread.
- You can guarantee that the stack will always be allocated. For example, you could write a guard routine to run once at the beginning of each thread and serially access each page up to the last page, leaving that page as a guard page.
- Your local variables require less than 2K of storage on the stack.

To turn off stack-probe generation, specify the /Gs+ compiler option. (📖 See page 305 for the option description.) Because stack probes go into the prolog of every function with more than 2K of stack storage, your program will run faster with the stack probes turned off. However, it is only safe to turn off stack probes if you can meet one or more of the above criteria.

Setting the Stack Size

You can set the stack size in one of three ways:

1. Specify the /B"/STACK:*size*" compiler option.
2. Specify the /STACK:*size* linker option on the linker command line.
3. Specify the STACKSIZE statement in a module definition (.DEF) file for the first thread of an application; use the _beginthread function call for threads created later.

📖 See Chapter 21, “Creating Module Definition Files” on page 369 for more information on .DEF files. See the *C Library Reference* for a description of the _beginthread function.

The default stack size is 32K for the first thread. Setting the stack size using one of the options listed above overrides the default value. For example, specifying the linker option

```
/STACK:65536
```

sets the stack size to be 64K.

Setting Stack Size

If your program calls 16-bit code, you can set the stack for the 16-bit code using the #**pragma** `stack16` directive, 📖 described in the *Language Reference*. Because the 16-bit stack is allocated from the 32-bit stack, you must ensure that the 32-bit stack is large enough for both your 32-bit and 16-bit code.

Setting Stack Size

15

Setting Compiler Options

You can use compiler options to alter the compilation and linking of your program. This chapter describes these options and tells you how to use them.

Specifying Compiler Options

Compiler options are not case-sensitive, so you can specify them in lower-, upper-, or mixed case. You can also substitute a dash (`-`) for the slash (`/`) preceding the option. For example, `-Rn` is equivalent to `/Rn`. Lower- and uppercase, dashes, and slashes can all be used on one command line, as in:

```
icc  /ls  -RN  -gD  /Li  prog.c
```

Some options have parameters. See “Using Parameters with Compiler Options” on page 255 for information.

You can specify compiler options in the following ways:

- On the command line
- In the ICC environment variable
- In the WorkFrame environment

Options specified on the command line override the options in the ICC variable.

Setting Options on the Command Line

Compiler options specified on the command line override any previously specified in the ICC environment variable (as described below and in “OS/2 Environment Variables for Compiling” on page 207).

For example, to compile a source file with the multithread option, enter:

```
icc /Gm myprog.c
```

Setting Options in ICC

Frequently used command-line options can be stored in the ICC environment variable. This method is useful if you find yourself repeating the same command-line options every time you compile. You can also specify source file names in ICC.

Specifying Compiler Options

The ICC environment variable can be set either from the command line, in a command (.CMD) file, or in the CONFIG.SYS file. If it is set on the command line or by running a command file, the options will only be in effect for the current session. If it is set in the CONFIG.SYS file, the options will be in effect every time you use `icc` unless you override them using a .CMD file or by specifying options on the command line.

For example, to specify that a source listing be generated for all compilations and that the macro `DEBUG` be defined to be 1, use the following command at the OS/2 prompt (or in your CONFIG.SYS file if you want these options every time you use the compiler):

```
   SET ICC=/Ls+  /DDEBUG::1
```

Use a double colon (::) instead of the `"="` sign, because the `"="` sign is not allowed in OS/2 environment variables.

Now, type `icc prog1.C` to compile `prog1.C`. The macro `DEBUG` is defined as 1, and a source listing is produced.

Options you specify on the command line override options in the ICC variable. For example, the following compiler invocation voids the effect of the ICC setting in the last example:

```
   icc /Ls- /UDEBUG fred.c
```

See “OS/2 Environment Variables for Compiling” on page 207 for more information about using ICC and other environment variables.

Setting Options in the WorkFrame Environment

If you have installed the WorkFrame product, you can set options as follows:

1. Double-click on your project icon to open the **Project** window.
2. From the **Options** menu, select **Build Smarts**. The **Build Smarts** window appears, in which you can select a standard task you want to build for. The options appropriate for that task are set automatically when you select **OK**.

   If you want to set options on an individual basis, as well as by general task, then do the following:

   a. Select **Compiler** from the **Options** menu to display the **Compiler Options** notebook.

   b. Select options in the notebook. Turn the pages to see all the options (there can be several pages under one tab).

If you prefer to set options in the command-line form, turn to the **User** tab of the notebook, and use the entry field there to enter options as you would on the command line.

See the online version of the *User's Guide*, and the online help for the notebook, for a mapping of command-line options to WorkFrame options.

c. Select **OK** when you are done.

The next time you build your project, the options you selected are used.

For more information on compiling with WorkFrame, 📖 see “Compiling within WorkFrame” on page 201.

Using Parameters with Compiler Options

For all compiler options that take parameters, the following rules apply:

- If a parameter is required, you can put zero or more spaces between the option and the parameter.
  For example, both `/FeMyexe.exe` and `/Fe  Myexe.exe` are valid.
- If a parameter is optional, do not put spaces between the option and parameter. For example, `/FlMylist.lst` is valid, but `/Fl Mylist.lst` is not.

The syntax of the compiler options varies according to the type of parameter that is used with the option. There are four types of parameters:

- Strings
- File names
- Switches
- Numbers

Strings

If the option has a string parameter, and the string contains spaces, enclose the string with a pair of double quotation marks. For example, `/V"Version 1.0"` is correct. If there are no spaces in the string, the quotation marks are not necessary. For example, both `/VNew` and `/V"New"` are valid.

If the string itself contains double quotation marks, precede these with the backslash (\) character. For example, if the string is `abc"def`, specify it on the command line as `"abc\"def"`. This combination is the only escape sequence allowed within string options. Do not end a string with a backslash, as in `"abc\"`.

If the string is optional, do not put a space between the option and the string.

Compiler Option Parameters

File Names

If you want to use a file that is in the current directory, specify only the file name. If the file you want to use is not in the current directory, specify the path and file name. For example, if your current directory is E:\, your source file is E:\myprog.c, and you compile using the defaults, your executable file will be called myprog.exe. If you want to put your executable file into the F:\ directory and call it newprog.exe, use the following command:

```
icc /FeF:\newprog.exe myprog.c
```

If you do not specify an extension for the executable file, .EXE is assumed.

If your file name contains spaces (as permitted by the High Performance File System (HPFS)) or any elements of the HPFS extended character set, it must be enclosed in double quotation marks. In such a case, do not put a space between the option and a file name or directory.

Switches

Some options are used with plus (+) or minus (-) signs. If you do not use a sign, the compiler processes the option as if you had used the + sign. When you use an option that uses switches, you can combine the switches. For example, the following two option specifications have the same result:

```
/La+ /Le+ /Ls+ /Lx-
/Laesx-
```

Note that the - sign applies only to the switch immediately preceding it.

Numbers

When an option uses a number as a parameter, do not put a space between the option and the number. When an option uses two numbers as parameters, separate the numbers with a comma. Do not leave a space between the numbers and the comma. For example:

```
/Sg10,132
```

is correct.

Scope of Compiler Options

Options apply only to the source files that follow the option. The last, or rightmost, occurrence of these options is the one that is in effect for the source file or files that follow it.

In the following example, the file `module1.c` is compiled with the option `/Fa-` because this option follows `/Fa+`:

```
icc /Fa+ /Fa- module1.c
```

In the next example, the file `module1.c` is compiled with the `/Fa+` option, while `module2.c` is compiled with `/Fa-`:

```
icc /Fa+ module1.c /Fa- module2.c
```

Exceptions The following options behave differently:

/D
: Defines a preprocessor macro. `/D` is different from other options in that the **first** definition of a macro is the one that is used. If a preprocessor macro is defined more than once, a warning appears.

/I
: Sets search paths for **#include** files. This option is cumulative. If you specify the option more than once, the parameters you specify are appended to the parameters previously stated. For example, the command

```
icc /Ia: /Ib:\cde /Ic:\fgh prog.c
```

causes the following search path to be built:

```
a:;b:\cde;c:\fgh
```

/B
: Passes options to the linker. Like `/I`, this option is cumulative. If you specify the option more than once, the parameters you specify are appended to the parameters previously stated. All options on the command line, and in environment variables, are accumulated **before** the object files are linked. The options apply to all object files linked.

Specifying Options with Multiple Source Files

When you compile programs with multiple source files, an option applies to all the source files that follow it. For example, if you enter the following command:

```
icc /Oi+ main.c /Fa sub1.c /Lx /Oi- sub2.c
```

- The file `main.c` will be compiled with the option `/Oi+`
- The file `sub1.c` will be compiled with the options `/Oi+` and `/Fa+`
- The file `sub2.c` will be compiled with the options `/Oi-,` `/Fa+` and `/Lx`

Scope of Compiler Options

The name of the executable module will be the same as the name of the first source file (`main`) but with the extension `.EXE`.

ICC Combined with Options Entered on the Command Line

When you specify compiler options both in the ICC environment variable and on the command line, the compiler evaluates both sets of options. When the compiler is invoked:

1. The string associated with ICC is retrieved.
2. The command line is retrieved.
3. The command line is appended to the ICC string, combining the two into a single command line.
4. This combined command line is read from left to right, and the compiler option precedence rules are applied.
5. The files are compiled and linked using the options as interpreted in the previous step.

As a result, values in ICC are processed before the command line, and options on the command line override any conflicting options in ICC.

Related Options

Some options are required with other options:

- If you specify the listing file option `/Le` (expand macros), or one of `/Li` or `/Lj` (expand **#include** files), you must also specify the `/Ls` option to include the source code.
- If you specify the `/Gr` option to generate code to run at ring zero, you must also specify the `/Rn` (no runtime environment) option.
- If you specify any of the listing options `/Lp` (set page length), `/Lt` (set title), or `/Lu` (set subtitle), you must also specify `/L` (produce listing file).
- If you specify `/Xs` (exclude specified files), you must also specify `/Ga` (turn on implicit SOM mode).

To use Performance Analyzer, you must specify both the `/Gh` and `/Ti` options.

Conflicting Options

Some options are incompatible with other options. If options specified on the command line are in conflict, the following rules apply:

- The syntax check option (/Fc) takes precedence over the output file generation (/Fa, /Fb, /Fe, /Fm, /Fo, and /Ft), intermediate code linker (/Fw and /Ol), and preprocessor (/P, /Pc, /Pd, and /Pe) options.
- The preprocessor options (/P, /Pc, /Pd, and /Pe) take precedence over the output file generation (/Fa, /Fb, /Fe, /Fl, /Fm, /Fo, and /Ft), intermediate code linker (/Fw, /Gu, and /Ol) precompiled header file (/Fi and /Si), and all listing file (/L) options.
- The option for no runtime environment (/Rn) takes precedence over the multithreading (/Gm), enable variables for 16-bit (/Gt), ddnames (/Sh), memory file (/Sv), and machine-state dump (/Tx) options.
- The option to not create an object file (/Fo-) takes precedence over the option to include debug information in the object (/Ti).
- The compile-only option (/C) takes precedence over the name executable module (/Fe) and generate linker map (/Fm) options.
- The no-optimization option (/O-) takes precedence over the instruction scheduler option (/Os+).
- The options to expand #**include** files in the listing (/Li and /Lj) take precedence over the precompiled header file options (/Fi and /Si).
- The option to expand user and system #**include** files (/Lj+) takes precedence over the option to expand user #**include** files only (/Li).
- The option to use the intermediate code linker (/Ol) requires some options to be consistently defined for all input files. The following options must have the same setting for all source files when you specify /Ol:

/G3
/G4
/G5
/Ge
/Gf
/Gh
/Gi
/Gp

/Gr
/Gs
/Gt
/Gv
/Gw
/Re
/Rn
/Nd

/O
/Oc
/Oi
/Om
/Op
/Os
/Ti
/Tn

See “Using the Intermediate Code Linker” on page 225 for more information.

Scope of Compiler Options

Language-Dependent Options

Some VisualAge C++ options are only valid when compiling C programs, while others only apply to C++ programs.

C **C Programs Only**

/Gv Control handling of DS and ES registers for virtual device driver development. VDD support is provided for C only.

/Sg Set margins for input files. This option is provided primarily for compatibility with IBM C/370. C++ does not require any such compatibility.

/Sq Set sequence numbers for input files. This option is provided primarily for compatibility with IBM C/370. C++ does not require any such compatibility.

/Sr Set type conversion rules. The C++ language only supports the new type conversion rules defined by the ANSI standard.

/Ss Allow use of double slashes for comments. C++ allows double slashes to indicate comments as part of the language.

/S2 Allow only SAA Level 2 C constructs. There is no SAA definition of the C++ language.

C++ **C++ Programs Only**

/Fb Control generation of browser files.

/Fr Give release order of class (SOM)

/Fs Create and direct IDL file (SOM)

/Ft Control generation of files for template resolution. The C language does not support templates.

/Ga Turn on implicit SOM mode.

/Gk Resolve templates in old C++ object files

/Gx Control inclusion of C++ exception handling information. The C language does not include specific constructs for exception handling.

/Gz Initialize SOM classes during static initialization

/Sc Allows constructs compatible with earlier versions of the C++ language. These constructs are not allowed in C.

/Nx Set names of exception-handling segments

/Xs Exclude files in specific directory when implicit SOM mode on.

Compiler Options for Presentation Manager Programming

If you are using the VisualAge C++ product to develop PM applications, you may need the following options:

Option	Description
/Se	Allow all VisualAge C++ language extensions. (This is the default.)
/Gm	Use the multithread libraries.
/Gs-	Do not remove stack probes. (This is the default.)
/Wpro	Produce diagnostic messages about unprototyped functions. (These are generated by default).

Examples of Compiler Options for Choosing Libraries

Figure 68 on page 262 shows the combinations of compiler options you use to create a particular type of module, according to:

- Static or dynamic linking
- Threading level:
  - Single-thread (/Gm-)
  - Multithread (/Gm+)
- Library being used:
  - Standard (/Re)
  - Subsystem (/Rn)
- Module being built:
  - .EXE file (/Ge+)
  - .DLL file (/Ge-)

The defaults used by the compiler are:

- /Gd- (Use static linking)
- /Gm- (Use the single-thread library)
- /Re (Use the standard library)
- /Ge+ (Build an .EXE file).

Examples of Compiler Options

Figure 68. Combinations of Compiler Options for Specifying Libraries

Linking Type	Threading	Library used	Module Type	Options required in addition to defaults
Static	Single	Standard	EXE	None
Static	Single	Standard	DLL	`/Ge-`
Static	Multi	Standard	EXE	`/Gm+`
Static	Multi	Standard	DLL	`/Gm+ /Ge-`
Static	N/A	Subsystem	EXE	`/Rn`
Static	N/A	Subsystem	DLL	`/Rn /Ge-`
Dynamic	Single	Standard	EXE	`/Gd+`
Dynamic	Single	Standard	DLL	`/Gd+ /Ge-`
Dynamic	Multi	Standard	EXE	`/Gd+ /Gm+`
Dynamic	Multi	Standard	DLL	`/Gd+ /Gm+ /Ge-`
Dynamic	N/A	Subsystem	EXE	`/Gd+ /Rn`
Dynamic	N/A	Subsystem	DLL	`/Gd+ /Rn /Ge-`

Compiler Option Classification

The compiler options are divided into groups by function. The following list tells you which options are in each group.

- "Output File Management Options" on page 268

  `/F`

- "**#include** File Search Options" on page 274

  `/I  /Xc /Xi`

- "Listing File Options" on page 276

  `/L`

- "Debugging and Diagnostic Information Options" on page 281

  `/N  /W  /Ti /Tm /Tn /Tx`

- "Source Code Options" on page 289

  `/S  /Tc /Td /Tp`

- "Preprocessor Options" on page 296

  `/D  /P  /U`

- "Code Generation Options" on page 299

  `/G  /M  /Nd /Nt /O  /R`

- "System Object Model (SOM) Options" on page 313

  `/Fr /Fs /Ga  /Gb  /Xs`

- "Other Options" on page 316

  `/B  /C  /H  /J  /Q  /Tl /V`

The table that follows gives all options, in all groups, in alphabetical order. The options are described in more detail in the sections following the table.

Compiler Options Summary

Figure 69 (Page 1 of 4). Compiler Options Summary

Option	Description	Default	Page
/?	Display list of compiler options with descriptions.	None.	316
/B"*options*"	Pass options to linker, in addtion to default options.	/B""	316
/C[+|-]	Perform compile without linking, instead of compiling and linking.	/C-	317
/D*name*[::*n*] /D*name*[=*n*]	Define preprocessor macros.	None.	296
/Fa[+|-][*dir*][*name*]	Produce, name, and direct assembler listing file.	/Fa-	269
/Fb[+|-|*]	Produce a browser file.	/Fb-	269
/Fc[+|-]	Perform syntax check only, instead of a full compile.	/Fc-	270
/Fe*name*	Specify name of executable output file.	Name of first source file	270
/Fi[+|-][*dir*][*name*]	Produce, name, and direct precompiled header file.	/Fi-	271
/Fl[+|-][*dir*][*name*]	Produce, name, and direct listing file.	/Fl-	271
/Fm[+|-] /Fm*name*	Produce and name linker map file.	/Fm-	272
/Fo[+|-][*dir*][*name*]	Control and name object file.	/Fo[+]	272
/Fr<*classname*>	Give release order of class	None	315
/Fs[+|-][*name*][*dir*]	Create and direct IDL file	/Fs-	315
/Ft[+|-] /Ft*dir*	Control and direct files for template resolution.	/Ft[+]	273
/Fw[+|-][*dir*][*name*]	Create intermediate code files only, instead of a full compilation. Specify name and directory for files.	/Fw-	273
/G[3|4|5]	Specify type of processor.	/G3	299
/Ga[+|-]	Turn on implicit SOM mode.	/Ga-	313
/Gb[+|-]	Disable direct access to attributes for DSOM.	/Gb-	314
/Gd[+|-]	Dynamically link to the runtime library, instead of linking statically.	/Gd-	300
/Ge[+|-]	Build .EXE or .DLL file.	/Ge[+]	300
/Gf[+|-]	Use fast floating-point execution.	/Gf-	301
/Gh[+|-]	Enable code for performance analysis.	/Gh-	301
/Gi[+|-]	Use fast integer execution.	/Gi-	302
/Gk[+|-]	Link with old C++ libraries	/Gk-	302
/Gl[+|-]	Remove unreferenced functions.	/Gl-	303

Figure 69 (Page 2 of 4). Compiler Options Summary

Option	Description	Default	Page
/Gm[+|-]	Link with the multithread library, instead of the single-thread library.	/Gm-	303
/Gn[+|-]	Hide default library information from linker.	/Gn-	304
/Gp[+|-]	Support _parmdwords in **_System** linkage.	/Gp-	304
/Gr[+|-]	Allow object code to run at ring 0.	/Gr-	304
/Gs[+|-]	Remove stack probes.	/Gs-	305
/Gt[+|-]	Enable variables for passing to 16-bit functions.	/Gt-	305
/Gu[+|-]	Stop external functions from using data defined in intermediate files.	/Gu-	305
/Gv[+|-]	Handle DS and ES registers for virtual device driver development.	/Gv-	306
/Gw[+|-]	Generate FWAIT instruction after each floating-point load instruction.	/Gw-	306
/Gx[+|-]	Remove C++ exception handling information.	/Gx-	307
/Gz[+|-]	Do not prebuild SOM classes at static initialization time.	/Gz-	314
/H*num*	Set significant length of external names.	/H255	317
/I*path*[;*path*]	Specify #**include** search paths, in addition to directory of source file and paths in INCLUDE.	No additional paths.	275
/J[+|-]	Treat unspecified char variables as signed char, instead of unsigned char.	/J[+]	317
/L[+|-]	Produce a minimal listing file.	/L-	277
/La[+|-]	Include a minimal layout in the listing file.	/La-	277
/Lb[+|-]	Include a layout in the listing file.	/Lb-	278
/Le[+|-]	Expand macros in listing file.	/Le-	278
/Lf[+|-]	Set all listing options on.	/Lf-	278
/Li[+|-]	Expand user #**include** files in the listing file.	/Li-	279
/Lj[+|-]	Expand user and system #**include** files in the listing file.	/Lj-	279
/Lp*num*	Set page length of listing file.	/Lp66	279
/Ls[+|-]	Include the source code in the listing file.	/Ls-	280
/Lt"*string*"	Set title string for listing file.	Name of first source file.	280
/Lu"*string*"	Set subtitle string in listing file.	/Lu""	280
/Lx[+|-]	Generate a minimal cross-reference table in the listing file.	/Lx-	281
/Ly[+|-]	Generate a cross-reference table in the listing file.	/Ly-	281
/M[p|s|c|t]	Set default calling convention.	/Mp	307
/N*n*	End compilation when error count reaches *n*.	No limit.	282

Compiler Options Summary

Figure 69 (Page 3 of 4). Compiler Options Summary

Option	Description	Default	Page
/Nd*name*	Set names of default data, uninitialized data, and constant segments.	Use DATA32, BSS32, and CONST32_RO.	308
/Nt*name*	Set names of default code or text segment.	Use CODE32.	308
/Nx*name*	Set names of exception-handling segments	EH_CODE and EH_DATA	309
/O[+|-]	Optimize code.	/O-	309
/Oc[+|-]	Optimize code for size.	/Oc-	310
/Oi[+|-] /Oi*value*	Inline specified user functions.	/Oi- /Oi+ when /O+	310
/Ol[+|-]	Use intermediate linker.	/Ol-	311
/Om[+|-]	Limit working set size.	/Om-	311
/Op[+|-]	Do not perform optimizations that involve the stack pointer.	/Op+	311
/Os[+|-]	Invoke the instruction scheduler.	/Os- /Os+ when /O+	312
/P[+|-]	Run the preprocessor only, instead of a full compile.	/P-	297
/Pc[+|-]	Preserve source code comments in preprocessor output.	/P-	297
/Pd[+|-]	Redirect preprocessor output.	/P-	297
/Pe[+|-]	Suppress `#line` directives in preprocessor output.	/P-	298
/Q[+|-]	Display compiler logo when invoking compiler.	/Q-	318
/R[e|n]	Generate code that can be used as a subsystem without a runtime environment.	/Re	312
/S[a|c|e|2]	Set language level.	/Se	289
/Sd[+|-]	Set the default file extension for source files to `.c`, instead of `.obj`.	/Sd-	289
/Sg[*l*][,<*r*|*>] /Sg-	Set left and right margins for the input file, and ignore text outside these margins.	/Sg-	290
/Sh[+|-]	Allow use of ddnames.	/Sh-	291
/Si[+|-][*dir*][*name*]	Use precompiled header files, if they exist and are current.	/Si-	291
/Sm[+|-]	Ignore unsupported 16-bit keywords, instead of treating them like any other identifier.	/Sm-	292
/Sn[+|-]	Allow use of DBCS.	/Sn-	292
/Sp[1|2|4]	Specify alignment or packing of data items within structures and unions.	/Sp4	292
/Sq[*l*][,<*r*|*>] /Sq-	Ignore text in specified columns, instead of processing all the contents of the input file.	/Sq-	293

Figure 69 (Page 4 of 4). Compiler Options Summary

Option	Description	Default	Page
/Sr[+|-]	Use old-style rules for type conversion, instead of new-style rules.	/Sr-	293
/Ss[+|-]	Allow double slashes to indicate comments.	/Ss-	293
/Su[+|-|1|2|4]	Control size of `enum` variables, instead of using the SAA rules.	/Su-	294
/Sv[+|-]	Allow use of memory files.	/Sv-	294
/Tc	Compile the following file as a C source file, regardless of its extension.	Compile based on file extension.	294
/Td[c|p]	Specify the default language (C or C++) for files, instead of compiling according to the file extension.	/Td	295
/Ti[+|-]	Generate debugger information.	/Ti-	282
/Tl[+|-|*value*]	Control preloading of the compiler.	/Tl[+]	318
/Tm[+|-]	Enable debug version of memory management functions.	/Tm-	283
/Tn[+|-]	Generate partial debugger information.	/Tn-	283
/Tp	Compile the following file as a C++ source file, regardless of its extension.	Compile based on file extension.	295
/Tx[+|-]	Provide a complete machine-state dump when an exception occurs, instead of providing only the exception message and address.	/Tx-	284
/U<*name*|*>	Undefine macros.	Retain macros.	298
/V"*string*"	Include a version string in the object and executable files.	No string.	318
/W[0|1|2|3]	Set severity level of messages the compiler produces and counts.	/W3	284
/W*grp*[+|-][*grp*]	Generate or suppress messages in the *grp* group.	/Wall-pro+ret+cnd+	284
/Xc[+|-]	Do not search paths specified using /I.	/Xc-	275
/Xi[+|-]	Do not search paths specified in INCLUDE.	/Xi-	275
/Xs[*dir*|-]	Exclude files in directory *dir* when `/Ga` is on (implict SOM mode).	/Xs-	314

Output File Management Options

Use these options to control the files that the compiler produces.

Note: You do not need the plus symbol (+) when specifying an option: the forms /Fa+ and /Fa are equivalent.

File Names and Extensions

If you do not specify an extension for the file management options that take a file name as a parameter, the default extension is used. For example, if you specify /Flcome, the listing file will be called come.lst. Although you can specify an extension of your own choosing, you should use the default extensions. See “File Types” on page 206 for more information on default extensions.

If you use an option without using an optional *name* parameter, the name of the following source file and the default extension is used, with the exception of the /Fm option. If you do not specify a name with /Fm, the name of the first file given on the command line is used, with the default extension .map.

Note: If you use the /Fe option, you **must** specify a name or a path for the file. If you specify only a path, the file will have the same name as the first source file on the command line, with the path specified.

See “File Names” on page 256 for more information on using file names as parameters with options.

Examples

- Perform syntax check only:

  ```
  icc /Fc+ myprog.c
  ```

- Name the object file:

  ```
  icc /Fobarney.obj fred.c
  ```

  This names the object file barney.obj instead of the default, fred.obj.

- Name the executable file:

  ```
  icc /Febarney.exe fred.c
  ```

  This names the object file barney.exe instead of the default, fred.exe.

- Name the listing file:

  ```
  icc /Floutput.my /L fred.c
  ```

  This creates a listing output called output.my instead of fred.lst.

- Name the linker map file:

  ```
  icc /Fmoutput.map fred.c
  ```

  This creates a linker map file called `output.map` instead of `fred.map`.

- Name the assembler listing file:

  ```
  icc /Fabarney fred.c
  ```

  This names the output `barney.asm`. instead of `fred.asm`.

/Fa

Syntax:	**Default:**

`/Fa[+|-]`
`/Fa[`*`dir`*`][`*`name`*`]`
	`/Fa-`

Use `/Fa` to produce, name, and direct an assembler listing file that has the source code as comments. The listing file will be *name*`.asm` and will be placed in directory *dir*.

The compiler produces a listing file for each source file that follows the option on the command line. The name you provide applies only to the first listing file.

If you do not specify a name or directory, then the listing takes the same name as the source file, with the extension `.asm`, and is put in the current directory.

If the directory you specify is not valid, the compiler does not generate a listing file: it generates a warning message and the option does not take effect.

Note: The listing is not guaranteed to compile.

By default, the compiler does not create an assembler listing file.

/Fb

Syntax:	**Default:**
`/Fb[+|-|*]`	`/Fb-`

C++ Use `/Fb` to produce a browser file, for use by the VisualAge C++ Browser. The file has the same name as the next source file with the extension `.pdb`. You can include maximum information in the browser file by specifying `/Fb*`. You do not need to specify `/Fb*` unless the compiler issues a message that tells you to use the option.

🕮 See “Creating Files to Use with the Browser” on page 559 for more information on the differences between `/Fb` and `/Fb*`.

If you are compiling and linking in one step, the compiler passes the `/BROWSE` option to the linker. If you are compiling only, you must specify the `/BROWSE` option directly, when you link, to preserve the browse information.

The browser file allows the VisualAge C++ browser to browse your program. 🕮 See Part 8, “Browsing Programs and Libraries” on page 551 for more information on the browser.

Note: This option is valid for C++ files only.

By default, the compiler does not produce a browser file.

/Fc

Syntax:	**Default:**
`/Fc[+|-]`	`/Fc-`

Use `/Fc` to perform only a syntax check. The only output files you can produce when this option is in effect are listing (`.lst`) files.

By default, the compiler compiles and produces output files according to any other options in effect.

/Fe

Syntax:	**Default:**
`/Fe`*`name`*	Use name of first source file, and add the .EXE or .DLL extension.

Use `/Fe` to specify the name of the .EXE or .DLL file you are producing. The executable output file will be *`name`*`.exe` or *`name`*`.dll`.

If you do not provide a name, the file takes the same name as the first source file, with the extension `.exe` or `.dll`.

/Fi

Syntax:	**Default:**
/Fi[+|-]	/Fi-
/Fi[*dir*][*name*]	

Use /Fi to control creation of precompiled header files. The compiler creates a precompiled header file if none exists or if the existing one is out-of-date.

If you specify a *name* or *dir*ectory with the option, then the precompiled headers are placed in a file with the name and in the directory you specify.

You can also use the **#pragma hdrfile** directive to tell the compiler what file to generate. You must still specify /Fi.

If you do not specify a name or directory, the file is named csetc.pch (if the next source file is a C file) or csetcpp.pch (if the next source file is a C++ file), and placed in the current working directory.

Use the /Si option to use the precompiled header files. Use /Fi and /Si in combination to ensure that your precompiled header files are always up to date.

Note: The file you generate (/Fi) must be the same file you use (/Si). If you specify different file names or directories with the two options, the name or directory specified last is used with both options. If you specify a file name or directory with **#pragma hdrfile**, it overrides the name or directory specified with the options.

See “Using Precompiled Headers” on page 239 for more information.

By default, the compiler does not create a precompiled header file.

/Fl

Syntax:	**Default:**
/Fl[+|-]	/Fl-
/Fl[*dir*][*name*]	

Use /Fl to produce, name, and direct a listing file. The listing file will be *name*.lst, and will be placed in directory *dir*.

The compiler produces a separate listing file for each source file that follows the option on the command line. The name you provide applies only to the first listing file.

📖 See “Compiler Listings” on page 222 for more information.

If you do not specify a name or directory, the listing takes the same file name as the source file, with the extension `.lst`, and is put in the current directory.

If the directory you specify is not valid, the compiler does not generate a listing file: it generates a warning message and the option does not take effect.

By default, the compiler does not produce a listing file.

/Fm

Syntax:	**Default:**
`/Fm[+|-]` `/Fm`*name*	`/Fm-`

Use `/Fm` to produce and name a linker map file. The map file will be *name*`.map`.

If you do not provide a name, the map file takes the same file name as the source file, with the extension `.map`.

📖 See “Generating a Map File” on page 341 for more information.

By default, the compiler does not produce a map file.

Note: Specify `/B"/MAP:full"` for a more detailed map file.

/Fo

Syntax:	**Default:**

`/Fo[+|-]`
`/Fo[`*dir*`][`*name*`]`
	`/Fo[+]`

Use `/Fo` to produce, name, and direct an object file. The object file will be *name*`.obj`, and will be placed in directory *dir*.

The compiler produces a separate object file for each source file that follows the option on the command line. The name you provide applies only to the first object file.

📖 See “Object Files” on page 220 for more information.

If you do not specify a name or a directory, the object file takes the same file name as the source file, with the extension `.obj`, and is put in the current directory.

If the directory you specify is not valid, the compiler generates a warning message, and places the object file in the current directory, with its default name.

By default, the compiler produces an object file with the same name as the source file and the extension `.obj`.

Specify `/Fo-` if you do not want the compiler to create an object file.

/Ft

Syntax:	**Default:**

`/Ft[+|-]`
`/Ft`*`dir`*
	`/Ft[+]`

C++ Use `/Ft` to control generation of files for template resolution.

Note: This option is valid for C++ files only. The C language does not support the use of templates.

Specify /Ft- to suppress generation of files for template resolution.

Specify /Ft*dir* to generate the files for template resolution and place them in the *dir* directory.

By default, files for template resolution are generated and stored in the TEMPINC subdirectory under the current directory.

/Fw

Syntax:	**Default:**

`/Fw[+|-]`
`/Fw[`*`dir`*`][`*`name`*`]`
	`/Fw-`

Use `/Fw` to produce, name, and direct intermediate code files, without completing compilation. The intermediate code files will be *name*`.w`, *name*`.wh`, *name*`.wi`, and *name*`.ws`, and will be placed in directory *dir*.

The compiler produces a separate set of intermediate code files for each source file that follows the option on the command line. The name you provide applies only to the first set of intermediate code files.

Intermediate code files are used by the intermediate code linker. For more information, see “Using the Intermediate Code Linker” on page 225.

If you do not specify a name, the intermediate code files take the same file name as the source file, with the extensions `.w`, `.wh`, `.wi`, and `.ws`.

If the directory you specify is not valid, the compiler does not save the intermediate code files: it generates a warning message and the option does not take effect.

By default, the compiler performs regular compilation, without saving intermediate code files.

#include File Search Options

Use these options to control which paths are searched when the compiler looks for **#include** files. The paths that are searched are the result of the information in the INCLUDE and ICC environment variables, combined with how you use the following compiler options.

Using the #include File Search Options

The `/I` option must be followed by one or more directory names. A space may be included between `/I` and the directory name. If you specify more than one directory, separate the directory names with a semicolon.

If you use the `/I` option more than once, the directories you specify are appended to the directories you previously specified. For example:

```
/Id:\hdr;e:\   /I f:\
```

is equivalent to

```
/Id:\hdr\;e:\;f:\;
```

If you specify search paths using `/I` in both the ICC environment variable and on the command line, **all** the paths are searched. The paths specified in ICC are searched before those specified on the command line.

Once you use the `/Xc` option, the paths previously specified by using `/I` cannot be recovered. You have to use the `/I` option again if you want to reuse the paths canceled by `/Xc`.

The `/Xi` option has no effect on the `/Xc` and `/I` options. For further information on **#include** files and search paths, see “Controlling **#include** Search Paths” on page 211.

/I

Syntax:	Default:
`/I`*`path`*`[;`*`path`*`]`	Directory of source file, paths in INCLUDE environment variable

Use `/I` to specify #**include** search path(s). The compiler will search *`path[;path]`*. Note that for user include files, the directory of the source file is always searched first.

By default, the compiler searches the directory of the source file (for user files only), and then search paths given in the INCLUDE environment variable.

/Xc

Syntax:	Default:
`/Xc[+|-]`	`/Xc-`

Use `/Xc` to stop the compiler from searching paths specified using `/I`.

By default, the compiler searches paths specified using `/I`.

/Xi

Syntax:	Default:
`/Xi[+|-]`	`/Xi-`

Use `/Xi` to stop the compiler from searching paths specified by the INCLUDE environment variable.

By default, the compiler searches paths specified in the INCLUDE environment variable.

Listing File Options

Use these options to control whether or not a listing file is produced, the type of information in the listing, and the appearance of the file.

Note: The following options only modify the appearance of a listing; they do not cause a listing to be produced. Use them with one of the other listing file options, or the /Fl option, to produce a listing:

```
/Le /Li /Lj /Lp /Lt /Lu
```

If you specify any of the /Le, /Li, or /Lj options, you must also specify the /L, /Lf, or /Ls option.

Including Information about Your Source Program

You can use three options to include information about your source program in the listing file:

/Ls[+] Includes your source program in the listing file.

/Li[+] Shows the included text after the user #**include** directives.

/Lj[+] Shows the included text after both user and system #**include** directives.

See the option descriptions for additional information.

Note: If you specify the /Lj option, /Li[+] and /Li- have no effect.

Including Information about Variables

The options that produce information about the variables used in your program provide the following amount of detail:

/La[+] Includes a table of all the referenced struct and union variables in the source program.

/Lb[+] Includes a table of all struct and union variables in the program.

/Le[+] Includes all expanded macros in the listing file.

/Lx[+] Includes a cross-reference table that contains a list of the referenced identifiers in the source file together with the numbers of the lines in which they appear.

`/Ly[+]` Includes a cross-reference table that contains a list of all identifiers referenced by the user and all external identifiers, together with the numbers of the lines in which they appear.

See the option descriptions for additional information.

/L

Syntax:	**Default:**
`/L[+|-]`	`/L-`

Use `/L` to produce a listing file. The listing file contains only a prolog and error messages. You can modify the contents of the listing using other listing file options.

See “Compiler Listings” on page 222 for more information.

By default, the compiler does not produce a listing file.

/La

Syntax:	**Default:**
`/La[+|-]`	`/La-`

Use `/La` to include a table in the listing file that shows the referenced `struct` and `union` variables in the source program. The table shows how each structure and union in the program is mapped. It contains the following information:

- The name of the structure or union and the elements within each.
- The byte offset of each element from the beginning of the structure or union. The bit offset for unaligned bit data is also given.
- The length of each element.
- The total length of each structure, union, and substructure in both packed and unpacked formats.

By default, the listing file does not include a layout.

/Lb

Syntax:	Default:
`/Lb[+|-]`	`/Lb-`

Use `/Lb` to include a table in the listing file that shows all the `struct` and `union` variables in the source program. The table shows how each structure and union in the program is mapped. It contains the following information:

- The name of the structure or union and the elements within each.
- The byte offset of each element from the beginning of the structure or union. The bit offset for unaligned bit data is also given.
- The length of each element.
- The total length of each structure, union, and substructure in both packed and unpacked formats.

By default, the listing file does not include a layout.

/Le

Syntax:	Default:
`/Le[+|-]`	`/Le-`

Use `/Le` to expand all macros in the listing file.

Note: To use `/Le`, you must also specify either `/L` or `/Ls`.

By default, the listing file does not show macros expanded.

/Lf

Syntax:	Default:
`/Lf[+|-]`	`/Lf-`

Use `/Lf` to set all listing options on, and generate a listing file.

By default, all listing options are off.

/Li

Syntax:	**Default:**
`/Li[+|-]`	`/Li-`

Use `/Li` to expand user #**include** files in the listing file.

Note: To use `/Li`, you must also specify either `/L` or `/Ls`.

By default, the listing file does not show user #**include** files expanded.

/Lj

Syntax:	**Default:**
`/Lj[+|-]`	`/Lj-`

Use `/Lj` to expand user and system #**include** files in the listing file.

If you use HPFS and have very long file names, there may not be enough room for the file names on the lines showing the included code. Counters are used in the INCLUDE column of the listing output, and the file name corresponding to each number is given at the bottom of the source listing.

Note: To use `/Lj`, you must also specify either `/L` or `/Ls`.

By default, the listing file does not show user and system #**include** files expanded.

/Lp

Syntax:	**Default:**
`/Lp`*`num`*	`/Lp66`

Use `/Lp` to set the page length in the listing file. Each page will be *`num`* lines long. You can set *`num`* to any number from 15 to 65535.

By default, the listing file has 66 lines per page.

/Ls

Syntax:	**Default:**
`/Ls[+|-]`	`/Ls-`

Use `/Ls` to include the source code in the listing file.

By default, the listing file does not include the source code.

/Lt

Syntax:	**Default:**
`/Lt"`*`string`*`"`	Use name of source file

Use `/Lt` to set the title string of the listing file to *`string`*. Maximum string length is 256 characters.

By default, the title string is set to the name of the source file.

Note: You can also specify a title using the **`#pragma title`** directive, but this title does not appear on the first page of the listing output.

/Lu

Syntax:	**Default:**
`/Lu"`*`string`*`"`	`/Lu""`

Use `/Lu` to set the subtitle string in the listing file to *`string`*. Maximum string length is 256 characters.

By default, no subtitle is set (null string).

Note: You can also specify a subtitle using the `subtitle` directive, but this subtitle does not appear on the first page of the listing output.

/Lx

Syntax:	**Default:**
`/Lx[+|-]`	`/Lx-`

Use `/Lx` to generate a cross-reference table in the listing file for referenced variable, structure, and function names, that shows line numbers where names are declared.

By default, the listing file does not include the cross-reference table.

/Ly

Syntax:	**Default:**
`/Ly[+|-]`	`/Ly-`

Use `/Ly` to generate a cross-reference table in the listing file of all variable, structure, and function names, plus all local variables referenced by the user.

By default, the listing file does not include the cross-reference table.

Debugging and Diagnostic Information Options

Use these options to help debug your programs.

Use `/Ti` to prepare your output for debugging with the VisualAge C++ debugger.

Use `/W`*`grp`* to control what types of diagnostic messages are produced.

Changed with this version: The `/K`*`n`* options are no longer valid. See Figure 70 on page 286 for a list of equivalent `W`*`grp`* options you can use intead.

Note: The information generated by the VisualAge C++ Debugger and `/W`*`grp`* options is provided to help you diagnose problems in your code. Do not use the diagnostic information as a programming interface.

/N

Syntax:	**Default:**
`/N`*n*	No limit

Use `/N` to set the maximum number of errors before compilation aborts. Compilation ends when the error count reaches *n*.

By default, the compiler sets no limit on the number of errors.

/Ti

Syntax:	**Default:**
`/Ti[+|-]`	`/Ti-`

Use `/Ti` to generate information for the VisualAge C++ Debugger and for Performance Analyzer.

By default, the compiler does not generate debug information. When you use `/Ti+`, do not turn on optimization (`/O+`, `/Oc+`, `/Oi+`, or `/Os+`), unless you are using the information with the performance analyzer, and not with the debugger. Because the compiler produces debugging information as if the code were not optimized, the information may not accurately describe an optimized program being debugged, which makes debugging difficult. Accurate symbol and type information is not always available.

If you cannot avoid debugging an optimized program, turn the scheduler off (`/Os-`), and step through the program at the assembly level, using the Register and Storage windows for information.

To make full use of the VisualAge C++ Debugger, set optimization off and use the `/G3` option. (Note that these are the defaults.)

For more information on the VisualAge C++ Debugger, see Part 6, "IBM VisualAge C ++ Debugger" on page 393.

For more information on Performance Analyzer, see Part 7, "Performance Execution Trace Analyzer" on page 475.

/Tm

Syntax:	**Default:**
`/Tm[+|-]`	`/Tm-`

Use `/Tm` to enable debug versions of memory management functions. The debug memory management functions (`_debug_calloc`, `_debug_malloc`, `new`, and so on) are then used in place of the regular memory management functions. This option defines the `__DEBUG_ALLOC__` macro. See the *C Library Reference* for information on the C debug memory management functions and the *Language Reference* for information on the debug versions of `new` and `delete`.

When you specify `/Tm`, the compiler generates additional code at the beginning of every function, that pre-initializes the local variables for the function. This makes it easier to find uninitialized local variables.

By default, the compiler uses the regular memory management functions (`calloc`, `malloc`, `new`, and so on), and does not pre-initialize their local storage.

/Tn

Syntax:	**Default:**
`/Tn[+|-]`	`/Tn-`

Use `/Tn` to generate abbreviated information for the debugger. You can then use the debugger to single-step through the source view of the files, but cannot view variables.

Specify `/Ti` to generate more complete debugger information.

If you specify both `/Tn` and `/Ti`, the compiler will generate the more complete debugging information indicated by `/Ti`.

By default, the compiler does not generate line number information.

/Tx

Syntax:	**Default:**
/Tx[+|-]	/Tx-

Use /Tx to provide a complete machine-state dump when an exception occurs.

By default, the compiler provides only the exception message and address when an exception occurs, and does not provide a complete machine-state dump.

/W

Syntax:	**Default:**
/W[0|1|2|3]	/W3

Use /W to set the severity level of messages the compiler produces and that causes the error count to increment. See the online *User's Guide* for a description of error messages and severity levels.

By default (/W3)), the compiler produces and counts all message types (severe error, error, warning, and informational).

You can set the following severity levels:

/W0 Produce and count only severe errors.

/W1 Produce and count severe errors and errors.

/W2 Produce and count severe errors, errors, and warnings.

/W3 Produce and count all message types (severe error, error, warning, and informational).

/Wgrp

Syntax:	**Default:**
/W*grp*[+|-] [*grp*]	/Wall-pro+ret+cnd+

Use /W*grp* to generate messages in the *grp* group. You can specify more than one group.

The /W*grp* options control informational messages that warn of possible programming errors, weak programming style, and other information about the structure of your programs. Similar messages are in groups, or suboptions, to give you greater control over which types of messages you want to generate. You can also specify these groups in your source code, with **#pragma info**.

By default, the compiler generates diagnostic messages in the pro, ret, and cnd groups.

When you specify /Wall[+], all suboptions are turned on and all possible diagnostic messages are reported. Because even a simple program that contains no errors can produce many informational messages, you may not want to use /Wall very often. You can use the suboptions alone or in combination to specify the type of messages that you want the compiler to report. Suboptions can be separated by an optional + sign. To turn off a suboption, you must place a - sign after it.

You can also combine the /W[0|1|2|3] options with the /W*grp* options.

/Wgrp Option

The following table lists the message groups and the message numbers that each controls, as well as the /K*n* option that formerly controlled each message. Messages generated for C files begin with EDC0, while messages for C++ files begin with EDC3.

Figure 70 (Page 1 of 2). /Wgrp Options

grp	/K*n* **Option**	**Controls Messages About**	**Messages**
/Wall	/Kf	All diagnostics.	All message numbers listed in this table.
/Wcls	(none)	Use of classes.	EDC3110, EDC3253, EDC3266
/Wcmp	(none)	Possible redundancies in unsigned comparisons.	EDC3138
/Wcnd	/Kb	Possible redundancies or problems in conditional expressions.	EDC0424, EDC0425, EDC0426, EDC0427, EDC0420, EDC0421, EDC0422, EDC0423, EDC3107, EDC3130, EDC3388, EDC3389, EDC3390, EDC3391, EDC3392, EDC3393
/Wcns	/Kb	Operations involving constants.	EDC0475, EDC0476, EDC0477, EDC3131, EDC3219, EDC3220
/Wcnv	/Kb	Conversions.	EDC3313, EDC3528
/Wcpy	(none)	Problems generating copy constructors.	EDC3199, EDC3200
/Weff	/Kb	Statements with no effect.	EDC0509, EDC0435, EDC0436, EDC0437, EDC0473, EDC0474, EDC0478, EDC0479, EDC0483, EDC3165, EDC3215
/Wenu	/Ke	Consistency of enum variables.	EDC0439, EDC0440, EDC0471, EDC3137, EDC3366
/Wext	/Kb and /Kx	Unused external definitions.	EDC0415, EDC0493, EDC0494, EDC3127
/Wgen	/Kb	General diagnostics.	EDC0438, EDC0448, EDC0466, EDC0480, EDC0489, EDC0492, EDC3101
/Wgnr	(none)	Generation of temporary variables.	EDC3151
/Wgot	/Kg	Usage of goto statements.	EDC0413
/Wini	/Ki	Possible problems with initialization.	EDC0444, EDC0445, EDC0446, EDC0447, EDC0482
/Winl	(none)	Functions not inlined.	EDC3542

Figure 70 (Page 2 of 2). /Wgrp Options

grp	/K*n* **Option**	**Controls Messages About**	**Messages**
/Wlan	(none)	Effects of the language level.	EDC3116
/Wobs	/Kb	Features that are obsolete.	EDC0450, EDC0470
/Word	/Kb	Unspecified order of evaluation.	EDC0428, EDC0429, EDC0430, EDC0431, EDC0432
/Wpar	/Kp	Unused parameters.	EDC0414, EDC3126
/Wpor	/Ko, /Kb	Nonportable language constructs.	EDC0433, EDC0434 EDC3108, EDC3133, EDC3135, EDC3136, EDC3307
/Wppc	/Kc	Possible problems with using the preprocessor.	EDC0076, EDC0290, EDC0293, EDC0311, EDC0312, EDC0313, EDC0389, EDC0441, EDC0442, EDC0443, EDC0457, EDC0468
/Wppt	/Kt	Trace of preprocessor actions.	EDC0467
/Wpro	/Kb	Missing function prototypes.	EDC0304
/Wrea	/Kb	Code that cannot be reached.	EDC0472, EDC0520, EDC3119
/Wret	/Kb	Consistency of `return` statements.	EDC0449, EDC0481
/Wtrd	/Ka	Possible truncation or loss of data or precision.	EDC0374, EDC0416, EDC0418, EDC0419, EDC0451, EDC0452, EDC0453, EDC0495, EDC3108, EDC3135, EDC3136
/Wtru	/Kr	Variable names truncated by the compiler.	EDC0484
/Wund	(none)	Casting of pointers to or from an undefined class.	EDC3098, EDC3397, EDC3405, EDC3406
/Wuni	/Ki	Uninitialized variables.	EDC0412
/Wuse	/Kb, /Kx	Unused `auto` and `static` variables.	EDC0409, EDC0410, EDC0469, EDC0490, EDC0491, EDC3002, EDC3099, EDC3100, EDC3101
/Wvft	(none)	Generation of virtual function tables.	EDC3280, EDC3281, EDC3282

More information about the messages generated by the /W*grp* options is available 📖 in the online version of the *User's Guide*.

Examples

- Produce all diagnostic messages:

  ```
  icc /Wall blue.c
  icc /Wall+ blue.c
  ```

- Produce diagnostic messages about:

  – Unreferenced parameters
  – Missing function prototypes
  – Uninitialized variables

  by turning on the appropriate suboptions:

  ```
  icc /Wpar+pro+uni blue.c
  icc /Wparprouni blue.c
  ```

- Produce all diagnostic messages except:

  – Warnings about assignments that can cause a loss of precision
  – Preprocessor trace messages
  – External variable warnings

  by turning **on** all options, and then turning **off** the ones you do not want:

  ```
  icc /Wall+trd-ppt-ext- blue.c
  ```

- Produce only basic diagnostics, with all other suboptions turned off:

  ```
  icc /Wgen+ blue.c
  ```

- Produce only basic diagnostics and messages about severe errors, errors, or warnings (`/W2`):

  ```
  icc /Wgen2 blue.c
  ```

Source Code Options

Use these options to control how the VisualAge C++ compiler interprets your source file. This control is especially useful, for example, if you are migrating code or ensuring consistency with a particular language standard.

/S

Syntax:	**Default:**
`/S[a|c|e|2]`	`/Se`

Use `/S` to set the language level.

You can set the following language levels:

`/Sa` Conform to ANSI standards.

C++ `/Sc` Allow constructs compatible with older levels of the C++ language.

Note: This option is valid only for C++ files.

`/Se` Allow all VisualAge C++ language constructs. This is the default.

C `/S2` Conform to SAA Level 2 standards.

Note: This option is valid only for C files.

🕮 See "Setting the Source Code Language Level" on page 214 for more information.

By default, the compiler allows all VisualAge C++ language extensions.

/Sd

Syntax:	**Default:**
`/Sd[+|-]`	`/Sd-`

C Use `/Sd` to set the default file extension to `.c`. Any file without an extension is then assumed to be a C source file, and will be compiled and linked.

By default, you must specify the extension for a source file:

```
icc anthony.c
icc efrem.cpp
```

If you omit the extension, VisualAge C++ compiler assumes that the file is an object file (`.obj`) and does not compile it, but only invokes the linker. The following commands are equivalent (assuming that `/Sd+` has not been specified elsewhere, such as in ICC):

```
icc dale
icc dale.obj
icc /Sd- dale
```

If you want the default file extension to be the default source file extension, use the `/Sd+` option. For example, the following two commands are equivalent:

```
icc alistair.c
icc /Sd+ alistair
```

Note: The `/Tc` and `/Tp` options override the setting of `/Sd`. If you specify either `/Tc` or `/Tp` followed by a file name without an extension, the compiler looks for the name specified, **without an extension**, and treats the file as a C file (if `/Tc` was specified) or a C++ file (if `/Tp` was specified). For example, given the following command:

```
icc /Tp xiaohu
```

the compiler searches for the file `xiaohu` and compiles it as a C++ file.

/Sg

Syntax:
`/Sg[l] [,<r|*>]`
`/Sg-`

Default:
`/Sg-`

Use `/Sg` to set left and right margins of the input file and ignore text outside these margins. This option is useful when you use source files that contain columns of characters you want to ignore.

Note: This option is only valid for C files.

The left margin is set to *l*. The right margin can be the value *r*, or you can use an asterisk to denote no right margin. *l* and *r* must be from 1 to 65535, and *r* must be greater than or equal to *l*.

By default, no margins are set, and the compiler uses the entire input file.

/Sh

Syntax:	Default:
`/Sh[+|-]`	`/Sh-`

Use `/Sh` to allow the use of data definition names (ddnames).

A ddname is the part of the data definition before the equal sign. Use a ddname in a call to `fopen` or `freopen` to refer to the data definition stored in the environment.

For more information on using ddnames, see the *Programming Guide*.

By default, the compiler does not allow ddnames.

/Si

Syntax:	Default:

`/Si[+|-]`
`/Si[`*dir*`][`*name*`]`
	`/Si-`

Use `/Si` to use precompiled header files, if they exist and are current.

If you specify a *name* or *dir*ectory with the option, then the compiler looks for a precompiled header file with the name and in the directory you specify.

You can also use the **#pragma hdrfile** directive to tell the compiler what file to look for. You must still specify `/Si`.

If you do not specify a name or directory, the compiler looks for a file named `csetc.pch` (if the next source file is a C file) or `csetcpp.pch` (if the next source file is a C++ file), in the current working directory.

Use the `/Fi` option to create or update the precompiled header files. Use `/Si` and `/Fi` in combination to ensure that your precompiled header files are always up to date.

Note: The file you generate (`/Fi`) must be the same file you use (`/Si`). If you specify different file names or directories with the two options, the name or directory specified last is used with both options. If you specify a file name or directory with **#pragma hdrfile**, it overrides the name or directory specified with the options.

See “Using Precompiled Headers” on page 239 for more information.

By default, the compiler does not use precompiled header files.

/Sm

Syntax:	**Default:**
`/Sm[+|-]`	`/Sm-`

Use `/Sm` to ignore unsupported 16-bit keywords, such as `near` and `far`.

By default, the compiler treats unsupported 16-bit keywords like any other identifier.

/Sn

Syntax:	**Default:**
`/Sn[+|-]`	`/Sn-`

Use `/Sn` to allow use of double-byte character set (DBCS). Compile with this option if your source files contain DBCS characters. Using `/Sn` increases your compile time.

By default, the compiler does not perform the checking needed to handle DBCS characters correctly.

/Sp

Syntax:	**Default:**
`/Sp[1|2|4]`	`/Sp4`

Use `/Sp` to specify alignment or packing of data items within structures and unions.

By default, structures and unions are aligned along 4-byte boundaries (normal alignment).

You can align structures and unions along 1-byte, 2-byte, or 4-byte boundaries. `/Sp` is equivalent to `/Sp1`.

/Sq

Syntax:	**Default:**

`/Sq[`*l*`] [,<`*r*`|*>]`
`/Sq-`
	`/Sq-`

C

Use `/Sq` to specify columns in which sequence numbers appear, and ignore text in those columns. This option can be used when importing source files from other systems, and is provided primarily for compatibility with IBM C/370.

Note: This option is only valid for C files.

Sequence numbers appear between columns *l* and *r* of each line in the input source code. *l* and *r* must be from 1 to 65535, and *r* must be greater than or equal to *l*. If you do not want to specify a right column, use an asterisk for *r*.

By default, the compiler does not use sequence numbers.

/Sr

Syntax:	**Default:**
`/Sr[+|-]`	`/Sr-`

C

Use `/Sr` to use old-style rules for type conversion. Old-style rules preserve the sign. They do not conform to ANSI standards.

By default, the compiler uses ANSI standard rules for type conversion. These rules preserve accuracy.

Note: This option is valid for C files only. C++ files must use the ANSI standard type conversion rules.

/Ss

Syntax:	**Default:**
`/Ss[+|-]`	`/Ss-`

C

Use `/Ss` to allow the use of double slashes (`//`) for comments.

By default, the compiler does not allow double slashes to indicate comments in a C file.

Note: This option is only valid for C files. C++ allows double slashes to indicate comments as part of the language.

/Su

Syntax:	**Default:**
`/Su[+|-|1|2|4]`	`/Su-`

Use `/Su` to control the size of `enum` variables. If you do not provide a size, all `enum` variables are made 4 bytes.

By default, the compilers uses the SAA rules: make all `enum` variables the size of the smallest integral type that can contain all variables.

You can specify the following sizes:

`/Su[+]` Make all `enum` variables 4 bytes.

`/Su1` Make all `enum` variables 1 byte.

`/Su2` Make all `enum` variables 2 bytes.

`/Su4` Make all `enum` variables 4 bytes.

/Sv

Syntax:	**Default:**
`/Sv[+|-]`	`/Sv-`

Use `/Sv` to allow the use of memory files.

For more information on using memory files, 📖 see the *Programming Guide*.

By default, the compiler does not allow memory files.

/Tc

Syntax:	**Default:**
`/Tc` *`filename`*	Compile `*.cpp` and `*.cxx` as C++, compile `*.c` and unrecognized files as C.

C Use `/Tc` to specify that the following file is a C file, regardless of its extension.

Important: The `/Tc` option **must** be immediately followed by a file name, and applies only to that file.

By default, the compiler compiles `.cpp` and `.cxx` files as C++ files, and `.c` and all other unrecognized files as C files.

/Td

Syntax:	Default:
`/Td[c|p]`	`/Td`

Use `/Td` to specify a default language (C or C++) for files.

By default, the compiler compiles `.cpp` and `.cxx` files as C++ files, and `.c` and all other unrecognized files as C files.

You can change the default as follows:

`/Tdc` Compile all source and unrecognized files that follow on the command line as C files.

`/Tdp` Compile all source and unrecognized files that follow on the command line as C++ files.

See “File Types” on page 206 for a list of file types the compiler recognizes.

Note: You can specify `/Td` anywhere on the command line to return to the default rules for the files that follow it.

/Tp

Syntax:	Default:
`/Tp` *`filename`*	Compile `*.cpp` and `*.cxx` as C++, compile `*.c` and unrecognized files as C.

C++

Use `/Tp` to specify that the following file is a C++ file, regardless of its extension.

Important: The `/Tp` option **must** be immediately followed by a file name, and applies only to that file.

By default, the compiler compiles `.cpp` and `.cxx` files as C++ files, and `.c` and all other unrecognized files as C files.

Preprocessor Options

Use these options to control the use of the preprocessor.

Note that the `/Pc`, `/Pd`, and `/Pe` options are actually suboptions of `/P`. Specifying `/Pc-` is the same as specifying `/P+c-` and causes only the preprocessor to be run.

Using the Preprocessor

Preprocessor directives, such as **#include**, allow you to include C or C++ code from another source file into yours, to define macros, and to expand macros. See the *Language Reference* for a list of preprocessor directives and information on how to use them.

If you run only the preprocessor, you can use the preprocessor output (which has all the preprocessor directives executed, but no code compiled) to debug your program. For example, all macros are expanded, and the code for all files included by **#include** directives appears in your program.

By default, comments in the source code are not included in the preprocessor output. To preserve the comments, use the `/Pc` option. For C programs, if you use `//` to begin your comments, you must also specify the `/Ss` option to include those comments in the preprocessor output.

The `/P`, `/Pc`, `/Pd`, and `/Pe` options can be used in combination with each other. For example, specify `/Pcde` to preserve comments, suppress `#line` directives, and redirect the preprocessor output to **stdout**.

/D

Syntax:	**Default:**

`/D`*name*`[::`*n*`]`
`/D`*name*`[=`*n*`]`
	Define no macros on the command line.

Use `/D` to define preprocessor macro *name* to the value *n*. If *n* is omitted, the macro is set to a null string. Macros defined on the command line override macros defined in the source code.

If the value *n* is more than one word, delimit it with double quotes:

```
/Dname="a b c"
```

To define *n* to a string literal, delimit the string literal with `/"` at either end, and enclose the whole in double quotes:

```
/Dname=" /"Some text/" "
```

Use the /U option to undefine macros on the command line.

By default, no macros are defined on the command line.

/P

Syntax:	Default:
/P[+|-]	/P-

Use /P to run the preprocessor only, and create a preprocessor output file that has the same name as the source file, with the extension .i.

By default, both the preprocessor and the compiler run, and no preprocessor output is generated.

/Pc

Syntax:	Default:
/Pc[+|-]	/P-

Use /Pc to run the preprocessor only, and create a preprocessor output file that includes the comments from the source code. The output file has the same name as the source file, with the extension .i.

Specify /Pc- to run the preprocessor only, and create a preprocessor output file with the comments stripped out. The output file has the same name as the source file, with the extension .i. /Pc- is equivalent to /P[+].

By default, both the compiler and preprocessor run, and no preprocessor output is generated.

/Pd

Syntax:	Default:
/Pd[+|-]	/P-

Use /Pd to run the preprocessor only, and send the preprocessor output to **stdout**.

Specify /Pd- to run the preprocessor only, and not redirect preprocessor output. Preprocessor output is written to a file that has the same name as the source file, with the extension .i. /Pd- is equivalent to /P[+].

By default, both the compiler and preprocessor run, and no preprocessor output is generated.

/Pe

Syntax:	**Default:**
/Pe[+|-]	/P-

Use /Pe to run the preprocessor only, and suppress generation of #line directives in the preprocessor output. The output file has the same name as the source file, with the extension .i.

Specify /Pe- to run the preprocessor only, and generate #line directives in the preprocessor output. The output file has the same name as the source file, with the extension .i. /Pe- is equivalent to /P[+].

By default, both the compiler and preprocessor run, and no preprocessor output is generated.

/U

Syntax:	**Default:**
/U<*name*|*>	Retain macros.

Use /U to undefine macros.

Specify /U*name* to undefine macro *name*.

Specify /U* to undefine all macros.

Note: /U does not affect some macros, such as __DATE__, __TIME__, __TIMESTAMP__, __FILE__, and __FUNCTION__, nor does it undefine macros defined in source code.

Use the /D option to define or redefine macros on the command line. Macros defined on the command line override macros defined in the source code.

By default, the preprocessor retains all macros.

Code Generation Options

Use these options to specify the type of code that the compiler will produce. The types of code include:

- Dynamically linked runtime libraries
- Statically linked runtime libraries
- Single-thread programs
- Multithread programs
- Subsystems

See the *Programming Guide* for more information.

Notes:

1. The `/Oi[+]` option is more effective when `/O[+]` is also specified.
2. Using optimization (`/O[+]`) limits your use of the VisualAge C++ Debugger to debug your code. The `/Ti` option is not recommended for use with optimization, when you are debugging. You can still use the `/Ti` option for analysis with Performance Analyzer.

/G

Syntax:	**Default:**
`/G[3|4|5]`	`/G3`

Use `/G` to specify the type of processor your code will run on.

By default, the compiler optimizes the code for a 386 processor (`/G3`).

You can specify the following processors:

`/G3` Optimize code for use with a 386 processor. The code will run on a 486 or Pentium microprocessor. The compiler includes any 486 or Pentium microprocessor optimizations that do not detract from the performance on the 386 processor. If you do not know what processor your application will be run on, use this option.

`/G4` Optimize code for use with a 486 processor. The code will run on a 386 or Pentium microprocessor. The compiler includes any Pentium microprocessor optimizations that do not detract from the performance on the 486 processor.

`/G5` Optimize code for use with a Pentium Microprocessor. The code will run on a 386 or 486 processor.

/Gd

Syntax:	**Default:**
`/Gd[+|-]`	`/Gd-`

Use `/Gd` to dynamically link to the runtime library. Your .EXE or .DLL file will call functions from VisualAge C++ DLLs, and must have access to these DLLs to run.

See “Static and Dynamic Linking” on page 237 for more information.

By default, the runtime library is statically linked. VisualAge C++ functions are copied into your .EXE or .DLL file from VisualAge C++ .LIB files, and does not need access to the VisualAge C++ DLLs. When you use the default, all external names beginning with the letters `Dos`, `Kbd`, and `Vio` are reserved. This restriction does not apply when compiling with `/Gd[+]`.

/Ge

Syntax:	**Default:**
`/Ge[+|-]`	`/Ge[+]`

Use `/Ge-` to build a `.DLL` file.

By default (`/Ge[+]`), the compiler builds an `.EXE` file.

The VisualAge C++ libraries provide two initialization routines, one for executable modules and one for DLLs. For each object file, the compiler must include a reference to the appropriate initialization routine. The name of this routine is then passed to the linker when the file is linked. Use the `/Ge` option at compile time to tell the compiler which routine to reference.

The `/Ge-` option causes the compiler to generate a reference to `_dllentry` for every module compiled. The `/Ge+` option generates a reference to `_exeentry` only if a `main` function is found in the source. If no `main` function is included, no linking reference is generated.

If you want to create a library of objects that can be linked into either an executable file or a DLL, use the `/Ge+` option when you compile. Typically, none of these objects would contain a definition of to `main`.

If one of the objects **does** contain a definition of `main`, you can override the `/Ge` option when you link your files, as follows:

1. Create a source file that defines `_exeentry`
2. Compile it into an .OBJ file, with the options `/C+` (create .OBJ file only, no linking) and `/Ge-` (compile for DLL)
3. Link the resulting object file with your other object files

When you link, the definition of `_exeentry` resolves any references in your other object files. Because you compiled the file with `/Ge-`, it contains a reference to `_dllentry`, and the linker links in the correct initialization routines.

/Gf

Syntax:	**Default:**
`/Gf[+|-]`	`/Gf-`

Use `/Gf` to specify fast floating-point execution.

If your program does not need to abide by ANSI rules regarding the processing of `double` and `float` types, you can use this option to increase your program's performance. Because the fast floating-point method does not perform all the conversions specified by the ANSI standards, the results obtained may differ from results obtained using ANSI methods, but are often more precise.

By default, the compiler does not use fast floating-point execution.

/Gh

Syntax:	**Default:**
`/Gh[+|-]`	`/Gh-`

Use `/Gh` to enable code to be run by Performance Analyzer and other profiling tools by generated profiler hooks in function prologs.

For more information on Performance Analyzer, see Part 7, “Performance Execution Trace Analyzer” on page 475.

Note: To enable code for Performance Analyzer, you must also specify `/Ti`.

By default, code is not enabled for Performance Analyzer.

/Gi

Syntax:	Default:
`/Gi[+|-]`	`/Gi-`

Use `/Gi` to specify fast integer execution.

If you are shifting bits by a variable amount, you can use fast integer execution to ensure that for values greater than 31, the bits are shifted by the result of a modulo 32 of the value. Otherwise, the result of the shift is 0.

Note: If your shift value is a constant greater than 32, the result will always be 0.

By default, the compiler does not use fast integer execution.

/Gk

Syntax:	Default:
`/Gk[+|-]`	`/Gk-`

C++ Use `/Gk` when you are linking object files created by versions of the compiler before version 3.0 that contain C++ templates. You must compile and link in one step (`/C-`, which is the default).

Note: This option is only valid for C++ files.

If you compile and link old object files without the `/Gk` compiler option, the linker cannot resolve the templates in the old files, and stops linking.

When you specify `/Gk`, the compiler invokes the prelinker from version 2.1 to resolve the templates in old object files, and passes the `/OLDCPP` option to the linker, so that it continues linking when it encounters old object files that contain templates. For more information on this linker option, see “/OLDCPP, /NOOLDCPP” on page 360.

By default, the compiler lets the linker handle template resolution in C++ files.

/Gl

Syntax:	**Default:**
`/Gl[+|-]`	`/Gl-`

Use `/Gl` to remove unreachable functions. The compiler passes the `/FUNCTIONOPT` option to the linker. The linker removes functions that are:

- Not referenced anywhere in the object code
- Referenced, but unreachable
- Not exported for use in other files

See “EXPORTS” on page 379 for more information on exporting functions.

When the function is removed, any additional functions that were required only by that function are also removed. Removing the functions and code reduces the size of your .EXE or .DLL output file.

Since the functions are removed during the linking stage, `/Gl` only takes effect if you compile and link in one step (`/C-`, which is the default). If you want to link separately from the compiler (`/C+`), you can invoke this optimization with the linker option `/FUNCTIONOPT`.

By default, the linker does not remove unreachable functions.

Performance Consideration: Optimized linking generally takes longer than regular linking, because of the extra processing that the linker performs. However, if the optimization is effective enough, it can actually speed up the linking process, because there is less information to write to file. Generally, you may want to compile without the `/Gl` option, until your code is tested and stable.

/Gm

Syntax:	**Default:**
`/Gm[+|-]`	`/Gm-`

Use `/Gm` to link with the multithread version of the library.

See “Choosing Your Runtime Libraries” on page 236 and “Using the Multithread Library” on page 238 for more information.

By default, object files are linked with the single-thread version of the library.

/Gn

Syntax:	Default:
/Gn[+|-]	/Gn-

Use /Gn to suppress linker information about the default libraries defined by the /Gd, Gm, /Ge, and /Rn options. All libraries must then be explicitly identified at link time.

By default, the compiler embeds the names of the default libraries in the object files, for use by the linker.

/Gp

Syntax:	Default:
/Gp[+|-]	/Gp-

Use the /Gp option to support the _parmdwords function with the **_System** linkage convention.

Previously, _parmdwords was always supported for **_System** linkage.

By default, _parmdwords is now not supported.

/Gr

Syntax:	Default:
/Gr[+|-]	/Gr-

Use /Gr to generate object code that runs at ring 0. Use this option if you are writing code, such as device drivers or operating systems, that will run at ring 0 instead of ring 3.

Note: To use /Gr, you must also specify /Rn.

By default, object code is not allowed to run at ring 0.

/Gs

Syntax:	**Default:**
`/Gs[+|-]`	`/Gs-`

Use `/Gs` to remove stack probes from the generated code.

For more information on stack probes, see “Controlling Stack Allocation and Stack Probes” on page 247.

By default, stack probes are not removed.

/Gt

Syntax:	**Default:**
`/Gt[+|-]`	`/Gt-`

Use `/Gt` to enable tiled memory and store all variables such that they may be passed to 16-bit functions. Static and external variables are mapped into 16-bit segments. Variables larger than 64K will be aligned on, but will still cross, 64K boundaries. When this option is specified, the memory management functions `calloc`, `free`, `malloc`, and `realloc` are mapped to the tiled versions `_tcalloc`, `_tfree`, `_tmalloc`, and `_trealloc`.

By default, variables are not enabled to be passed to 16-bit functions.

/Gu

Syntax:	**Default:**
`/Gu[+|-]`	`/Gu-`

Use `/Gu` to tell the intermediate linker that data defined in the intermediate link is not used by external functions. The data is used only within the intermediate files being linked, with the exception of data that is exported using `_Export`, **`#pragma export`**, or a `.DEF` file.

When you specify `/Gu`, the intermediate code linker can optimize code more effectively. See “Using the Intermediate Code Linker” on page 225 for more information about the intermediate code linker.

By default, external functions may use data defined in the intermediate files being linked.

/Gv

Syntax:	Default:
`/Gv[+|-]`	`/Gv-`

C

Use `/Gv` to perform special handling of the DS and ES registers, for virtual device driver development. The DS and ES registers are:

1. Saved, on entry to an external function
2. Set to the selector for DGROUP
3. Restored, on exit from the function

Note: This option is valid for C files only. Virtual device driver development is not supported for C++ programs.

For more information on developing virtual device drivers, see the *Programming Guide*.

By default, the DS and ES registers are handled in a normal manner.

/Gw

Syntax:	Default:
`/Gw[+|-]`	`/Gw-`

Use `/Gw` to generate an `FWAIT` instruction after each floating-point load instruction. This allows the program to take a floating-point stack overflow exception immediately after the load instruction that caused it.

Note: This option is not recommended because it increases the size of your executable file and greatly decreases its performance. You do not need this option unless you call assembler code that leaves a different number of values on the floating point stack.

By default, `FWAIT` instructions are not generated after each floating-point load instruction.

/Gx

Syntax:	Default:
`/Gx[+|-]`	`/Gx-`

C++ Use `/Gx` to remove C++ exception-handling information.

For more information on C++ exception handling see the *Programming Guide* and the *Language Reference*.

By default, C++ exception-handling information is not removed.

Note: This option is valid for C++ files only.

/M

Syntax:	Default:
`/M[p|s|c|t]`	`/Mp`

Use `/M` to set the calling convention, as follows:

Option	Calling Convention
`/Ms`	**_System** calling convention
`/Mc`	**__cdecl** calling convention
`/Mt`	**__stdcall** calling convention
`/Mp`	**_Optlink** calling convention

The default is the **_Optlink** calling convention (`/Mp`).

You must include the header files for libraries that use a different calling convention from the one you specify. The libraries using the following calling conventions:

VisualAge C++ libraries
Functions use **_Optlink** calling convention. Include the VisualAge C++ library header files to call VisualAge C++ functions when you set `/Ms`, `/Mc`, or `/Mt`.

Toolkit libraries
APIs use **_System** calling convention. Include the Toolkit library header files to call OS/2 APIs when you set `/Mp` (the default), `/Mc`, or `/Mt`.

If you do not include the header files, then your code will attempt to call functions with the calling convention you set, rather than with the calling convention the function requires.

/Nd Option •/Nt Option

See “Setting the Calling Convention” on page 235 for more information on calling conventions.

/Nd

Syntax:	Default:
`/Nd`*`name`*	Use DATA32, BSS32, and CONST32_RO

Use `/Nd` to specify the names of default data, uninitialized data, and constant segments as *name*DATA32, *name*BSS32, and *name*CONST32_RO. You can then give the segments special attributes, with the linker option `/SECTION`, or the module statement `SEGMENTS`. The renamed segments are not placed in the default data group (DGROUP)

You can also use #**pragma `dataseg`** to name these segments.

Notes:

1. CONST32_RO is never in the default data group. It replaces CONST32, which in previous versions was part of the default data group. While the `/Nd` option allows you to rename CONST32_RO, you do not need to rename it in order to assign it special attributes.
2. CONST32_RO is READONLY and SHARED by default.

If you do not use the `/Nd` option, the default names are DATA32, BSS32, and CONST32_RO.

/Nt

Syntax:	Default:
`/Nt`*`name`*	Use CODE32

Use `/Nt` to specify the name of default code or text segments as *name*CODE32. You can then give the segments special attributes, with the linker option `/SECTION`, or the module statement `SEGMENTS`. You can also use #**pragma `alloc_text`** to name these segments.

If you do not use the `/Nt` option, the default name is CODE32.

/Nx

Syntax:	Default:
/Nx*name*	EH_CODE and EH_DATA

C++ Use /Nx to specify the names of one code and one data segment, that contain information relating to C++ exception handling. This information is only used during the processing of a throw statement. Separating the segments from the default code and data segments improves the paging behavior of your program.

The code segment is named *name*_CODE of class CODE. The data segment is named *name*_DATA of class DATA.

Note: This option is valid for C++ files only.

By default, the segments are named EH_CODE and EH_DATA.

/O

Syntax:	Default:
/O[+|-]	/O-

Use /O to optimize code for speed.

Note: Do not optimize code if you want to use debugging or diagnostic options. The debugger may operate unpredictably with optimized code. 🕮 See “Generating Debugger Information” on page 220 for more information.

When you specify /O, the following optimization options are turned on by default:

/Oi Turn on inlining.
/Os Invoke the instruction scheduler.

By default, the code is not optimized.

/Oc

Syntax:	**Default:**
`/Oc[+|-]`	`/Oc-`

Use `/Oc` to optimize code for size as well as speed. You must also specify `/O`.

`/Oc` performs the same set of optimizations as `/O`, except for those that increase the size of the code. Code optimized with `/Oc` is **not** slower than unoptimized code, and is likely to be faster, though not as fast as code optimized with `/O` on its own.

For example, `/Oc` stops loops from being unrolled, and stops most inlining (unless you specified a conflicting `/Oi` value).

Note: Do not optimize code if you want to use debugging or diagnostic options. The debugger may operate unpredictably with optimized code. See “Generating Debugger Information” on page 220 for more information.

By default, the code is not optimized.

/Oi

Syntax:	**Default:**
`/Oi[+|-]`	`/Oi-`
`/Oi`*`value`*	`/Oi+` when `/O+`

Use `/Oi` to control inlining of user code.

By default, the compiler does not inline user code, unless you specify `/O[+]`. When you specify `/O[+]`, `/Oi[+]` becomes the default.

You can specify the following types of inlining:

`/Oi+` Inline all user functions that are qualified with the `_Inline` or `inline` keyword.

`/Oi-` Do not inline any user code.

`/Oi`*`value`* Inline all user functions qualified with the `_Inline` or `inline` keyword or that are smaller than *`value`* in abstract code units.

See “Inlining User Code” on page 229 for more information.

/Ol

Syntax:	Default:
`/Ol[+|-]`	`/Ol-`

Use `/Ol` to pass code through the intermediate linker before generating an object file. 📖 See "Using the Intermediate Code Linker" on page 225 for more information.

By default, code is not passed through the intermediate linker.

/Om

Syntax:	Default:
`/Om[+|-]`	`/Om-`

Use `/Om` to limit the working set size for the compiler to approximately 35M.

The compiler may use a large amount of memory when inlining user code, especially when performing automatic inlining at large thresholds (`/Oi50` and higher).

By default, there is no limit to the compiler's working set size.

Note: Because `/Om[+]` can cause the compiler to disregard some inlining opportunities, code generated with `/Om-` (the default) may be more efficient.

/Op

Syntax:	Default:
`/Op[+|-]`	`/Op+`

Use `/Op` to perform optimizations involving the stack pointer. To use this option, you must also specify `/O[+]`.

By default, optimizations involving the stack pointer are always performed.

Specify `/Op-` when you optimize code that directly manipulates the stack pointer. Using `/Op-` decreases the performance of your .EXE or .DLL file.

/Os

Syntax:	**Default:**

`/Os[+|-]`	`/Os-`
	`/Os+` when `/O+`

Use `/Os` to invoke the instruction scheduler. To use this option, you must also specify `/O[+]`.

By default, the instruction scheduler is invoked when you have specified `/O[+]`. Otherwise, the instruction scheduler cannot be invoked.

/R

Syntax:	**Default:**
`/R[e|n]`	`/Re`

Use `/R` to control the executable runtime environment.

Specify `/Rn` to generate executable code that can be used as a subsystem without a runtime environment.

For more information on developing subsystems, see “Enabling Subsystem Development” on page 239.

By default (`/Re`), the compiler generates executable code that runs in a VisualAge C++ environment.

System Object Model (SOM) Options

This section describes the compiler options available for SOM support in VisualAge C++. See the *Programming Guide* for background information on the reasons for these options and on their uses.

SOM options that affect the same settings as SOM pragmas are effective except when overridden by those pragmas. For example, the /Ga compiler option, which causes all classes to implicitly derive from SOMObject, turns the **SOMAsDefault** pragma on at the start of the translation unit. This pragma remains in effect until a **#pragma SOMAsDefault(off|pop)** is encountered in the translation unit. See the *Programming Guide* for more information on the relationship between SOM pragma settings and SOM options.

In addition to the compiler options, the compiler defines a macro, **__SOM_ENABLED__**, whose value corresponds to the level of SOM support provided by the compiler. If SOM support is not provided for a particular release of the compiler, **__SOM_ENABLED__** is not predefined.

The macro's value is a positive integer constant. For the first SOM-supporting release of VisualAge C++, the level of SOM supported is SOM 2.1, so the macro has the value 210.

/Ga

Syntax:	**Default:**
/Ga[+|-]	/Ga-

This option turns on implicit SOM mode, and also causes the file som.hh to be included. It is equivalent to placing **#pragma SOMAsDefault(on)** at the start of the translation unit.

All classes are implicitly derived from SOMObject until a **#pragma SOMAsDefault(off)** is encountered.

For further details, see the *Programming Guide*.

SOM Options

/Gb

Syntax:	Default:
`/Gb[+|-]`	`/Gb-`

This option instructs the compiler to disable direct access to attributes. Instead, the get and set methods are used. This is equivalent to specifying **`#pragma SOMNoDataDirect(on)`** as the first line of the translation unit.

For further details, see the *Programming Guide*.

/Gz

Syntax:	Default:
`/Gz[+|-]`	`/Gz-`

Use this option to initialize SOM classes at their point of first use during the execution of your program.

By default, all SOM classes statically used in your program are initialized at static initialization time. This makes your program smaller, but may result in the initialization of classes that are not dynamically used.

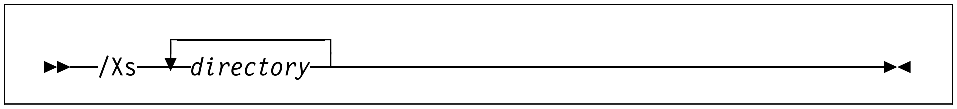
With any setting of this option, any reference to a static member of a SOM class will cause the class to be initialized.

For further details, see the *Programming Guide*.

/Xs

Syntax:	Default:
`/Xs<`*`directory`*`|->`	`/Xs-`

Use this option to exclude files in the specified directories when implicit SOM mode is turned on (when classes are implicitly derived from SOM). The syntax of this option is:



where *directory* is the name of the directory or directories you want to exclude. Directory names are separated with a semicolon (;).

This option is useful for mixing implicit SOM mode with existing include files that include declarations of classes you do not want to be implicit SOM classes.

/Fr

Syntax:	**Default:**
/Fr<*classname*>	None

Use this option to have the compiler write the release order of the specified class to standard output. The release order is written in the form of a **SOMReleaseOrder** pragma. You can capture the output from this option when developing new SOM classes, and include the pragma in the class definition. The syntax of the option is:

```
►►──/Fr──C++ClassName──────────────────────────────────►◄
```

For further details, see the *Programming Guide*.

/Fs

Syntax:	**Default:**
/Fs[+|-|*file*| *directory*]	/Fs-

Use this option to have the compiler generate an IDL file if a file with an .hh extension is explicitly specified on the command line. The syntax of the option is:

```
►►──/Fs──┬───────────┬──────────────────────────────────►◄
         ├─+─────────┤
         ├─-─────────┤
         ├─filename──┤
         └─directory─┘
```

where:

/Fs<+> specifies that an IDL file will be created for every .hh file that is specified on the command line and is in the current directory. This is the default.

/Fs *filename.ext* is like /Fs +, but the IDL file that is created will have the specified filename. If you do not specify an ext, the extension will be idl.

`/Fs directory_name` is like `/Fs +`, but the IDL file that is created will be put in the directory *`directory_name`* rather than the current directory. *`directory_name`* must end with a backslash `"\"`.

`/Fs-` specifies that no IDL file should be created.

Other Options

Use these options to control linker parameters, logo display, default `char` type, and other VisualAge C++ settings.

Examples

- Passing a parameter to the linker:

  ```
  icc /B"/NOI" fred.c
  ```

  The `/NOI` option tells the linker to preserve the case of external names in `fred.obj`.

- Imbedding a version string or copyright:

  ```
  icc /V"Version 1.0" fred.c
  ```

  This imbeds the version notice in `fred.obj`.

/?

Syntax:	**Default:**
`/?`	None

Use `/?` to display a list of compiler options with descriptions.

/B

Syntax:	**Default:**
`/B"`*`options`*`"`	`/B""`

Use `/B` to pass the *`options`* string to the linker as parameters. The `icc` default parameters are also passed. 📖 See “Summary of Linker Options” on page 350 for information about the options you can pass to the VisualAge C++ linker.

By default, only the `icc` default parameters are passed to the linker. 📖 See “Linking through the Compiler” on page 329 for a description of the options passed to the linker by default.

/C

Syntax:	**Default:**
`/C[+|-]`	`/C-`

Use `/C` to peform a compile only, without linking.

When you link separately, you need to specify options that the compiler normally passes to the linker. See “Linking through the Compiler” on page 329 for a description of these options.

By default, code is both compiled and linked.

/H

Syntax:	**Default:**
`/H`*`num`*	`/H255`

Use `/H` to set the significant length of external names. The first *`num`* characters of an external name are set as significant. The value of *`num`* must be between 6 and 255 inclusive.

By default, the first 255 characters of external names are significant.

/J

Syntax:	**Default:**
`/J[+|-]`	`/J+`

Use `/J-` to treat unspecified `char` variables as `signed char`, for arithmetic and compare operations.

This option does **not** change the definition of the variable. In C++, `char`, `signed char`, and `unsigned char` are three distinct types. Unspecified `char` variables are considered `signed char` or `unsigned char` for arithmetic and compare operations only. You cannot convert one type to another type without casting.

By default (`/J[+]`), unspecified `char` variables are treated as `unsigned char`.

/Q

Syntax:	Default:
`/Q[+|-]`	`/Q-`

Use `/Q` to stop the compiler logo from appearing when you invoke the compiler.

By default, the compiler logo appears on **stderr**.

/Tl

Syntax:	Default:
`/Tl[+|-|`*value*`]`	`/Tl[+]`

Use `/Tl` to control preloading of the compiler.

By default, each compiler component is preloaded as required, and kept in memory for ten minutes from the time it was last referenced.

You can set the following preloading options:

`/Tl[+]` Preload the compiler components as required. A component remains in memory for 10 minutes. If it is referenced in that time, the timer starts again. Each compiler component has its own timer.

Note: This option is not available if you are using the OS/2 2.0 operating system without the Service Pack.

`/Tl-` Do not preload the compiler. You can specify this option without a file name to unload any components that are loaded.

`/Tl`*value* Preload the compiler components as required and keep the files in memory for *value* minutes.

/V

Syntax:	Default:
`/V"`*string*`"`	No string.

Use `/V` to include a version string in the object and executable output files. The version string is set to *string*. The length of the string can be from 1 to 256 characters.

By default, no version string is set.

Part 5. Linking Your Program

This part of the *User's Guide* describes the VisualAge C++ linker, which links the object files produced by the compiler into executable files, for example .EXE files or .DLL files. By default, the compiler invokes the linker for you.

Starting the Linker

Once the compiler has created object modules out of your source files, use the linker to link them together with the VisualAge C++ runtime libraries to create an .EXE file, .DLL file, or device driver. By default, `icc` invokes the linker for you.

There is a new linker with VisualAge C++ version 3.0. Earlier versions of C Set ++ used LINK386 as the linker. LINK386 is still available with OS/2, but the VisualAge C++ compiler is designed to work with the new linker. LINK386 cannot link object files created by the VisualAge C++ compiler. The VisualAge C++ linker is now faster, and has improved linking strategies that can significantly reduce the size of your code. 📖 See Chapter 17, “Optimized Linking” on page 333 for a description of these linking strategies.

The linker also handles C++ template resolution independently. In earlier versions of C Set ++, you had to link through the compiler using the `/Tdp` option to ensure templates resolved correctly. With VisualAge C++ Version 3.0, you can invoke the linker as a separate step.

Note: The object files must be created by the Version 3.0 compiler. The linker does not resolve template functions in object code created by old versions of the compiler. If you are linking old object files that contain template functions, invoke the linker through the compiler and specify the `/Gk` compiler option.

For a complete list of the differences between the VisualAge C++ linker and LINK386, 📖 see “Linker Changes” on page xxxix.

There are several ways you can start the linker:

- From the popup menu of an object file in a WorkFrame project, or from the project popup menu as part of the make or build process.
- From the command line.
- Through the compiler, which automatically invokes the linker.
- Through a make file, which invokes both the compiler and the linker.

Linking within WorkFrame

To use the linker through WorkFrame, do the following:

1. Open the VisualAge C++ folder.
2. Double click on the **Project Smarts** icon to open the Project Smarts Catalog.
3. In the Project Smarts - Catalog window, select the type of project you want to build.

   If you want to build a type of project that is not listed, you can either pick a similar project type and then customise its settings, or create one from the **Project** template without project-specific defaults, and customise its settings extensively.
4. Select **Create**. WorkFrame begins creating the project. The Console window shows the status of the process. Other windows may appear, for you to provide additional information about your project.
5. In these other windows, provide specific information in fields where the default information is unacceptable.
6. Select **OK** when you are done, in each window.
7. Your project becomes an icon on the desktop, or in a folder if you specified a different destination. Find your project.
8. Double click on your project icon. The Project Window appears.

   At this point you can customise settings for the project, if the default settings for the project type are unacceptable. The **Options** menu contains choices that allow you to specify the actions available to the project, and compiler and linker options. Use **Build Smarts** to set options for a standard task. Use the Compiler and Linker Options dialogs to set options on an individual basis.
9. Select **Build** from the **Actions** menu. Your project is created, with the compiler and linker invoked as required.

Linking from the Command Line

Specify the `ilink` command followed by any sequence of options, file names, or directories, separated by space or tab characters.

```
►►──ILINK──┬─/option───┬──┬──────────────────────────────┬──►◄
           ├─filename──┤  └─┬──────────┬──@responsefile─┘
           └─directory\─┘    ├─/LOGO────┤
                             └─/NOLOGO──┘
```

The linker recognizes the input as follows:

Options Start with a / or - character.

Directories End with a / or \ character.

Response files Start with the `@` character.

Definition files End with the `.def` extension.

Library files End with one of these extensions:

- `.a`
- `.imp`
- `.lib`

Object files Any other input. You must enter at least one object file.

You can specify the name of the output file with the `/OUT` option. You can specify the name of a map file with the `/MAP` option.

In addition to the libraries you specify, by default the linker searches the VisualAge C++ runtime libraries defined in your object files at compile time. See "Choosing Your Runtime Libraries" on page 236 for more information on setting the default libraries.

The directories you specify become part of the linker's search path, before any directories set in the LIB environment variable. See "Search Rules" on page 335 and "Specifying Directories" on page 336 for more information.

You can specify a definition file to describe the characteristics of an application or library, including imports and exports. See Chapter 21, "Creating Module Definition Files" on page 369 for more information.

Note: If the linker cannot find a file, it stops linking. The linker does not assume the .OBJ extension for a file: if you specify a file with no extension, then the linker looks for that file with no extension.

Starting the Linker

You can still use the use the LINK386 command-line syntax, if you prefer. To use the old-style syntax, specify the /NOFREE option at the start of the command line, or in the ILINK environment variable. 📖 See “Using LINK386 Syntax” on page 325 for more information on using the old syntax.

Examples

The following command links the object files fun.obj, text.obj, table.obj, and care.obj. The linker searches for unresolved external references in the library file xlib.lib and in the default libraries. Since there is no name provided for the executable file, it is named fun.exe, taking the file name of the first object file and the default extension .exe. The linker also produces a map file, funlist.map.

```
ILINK /MAP:funlist fun.obj text.obj table.obj care.obj xlib.lib
```

The following command links the object file fun.obj and produces the executable file fun.exe, as well as the map file fun.map.

```
ILINK /MAP fun.obj
```

The following command links the files main.obj, getdata.obj, and printit.obj into an executable file named main.exe, and produces a map file named main.map.

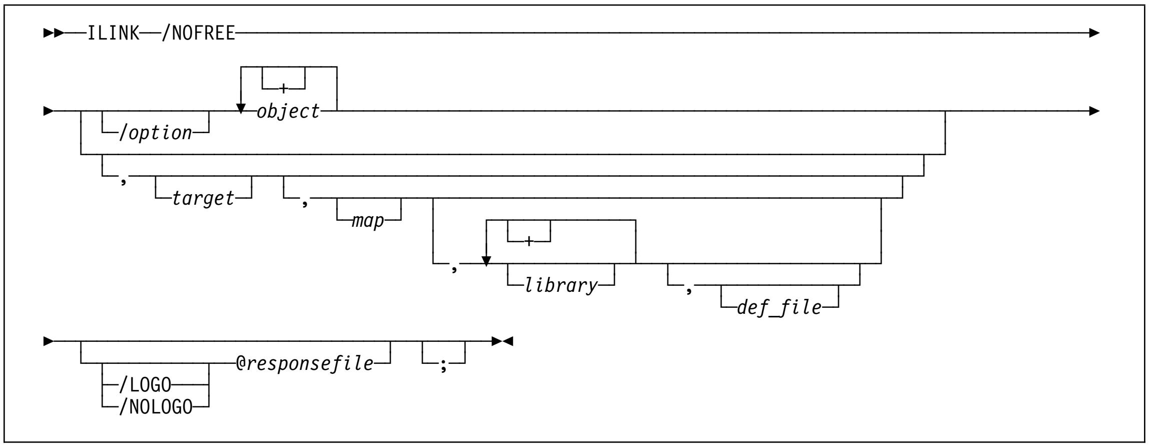
```
ILINK /MAP main.obj getdata.obj printit.obj
```

The following command links getdata.obj and printit.obj into a DLL named getdata.dll. (This example assumes that exports are declared in the source files, using **#pragma export** or the **_Export** keyword; otherwise there would have to be a module definition file, to define the exports.)

```
ILINK getdata.obj printit.obj /OUT:getdata.dll /DLL
```

Using LINK386 Syntax

If you specify `/NOFREE` at the start of the command line (or in the ILINK environment variable), then you can use the LINK386-style syntax to specify input to the linker. The syntax for the `ilink` command then becomes:



You must specify at least one object file. Leave any other parameter blank to accept the default entry for it. You can end the command line at any point with a semicolon (;), to accept the default for all remaining parameters. If you do not specify the semicolon, the linker prompts you for any remaining parameters.

If you specify a response file, anything after the response file is ignored.

You can specify the following parameters:

option Options that modify the behavior of the linker. Each option must begin with a forward slash (/) or dash (-). You can specify linker options anywhere on the command line, except for `/LOGO` or `/NOLOGO`, which must appear before any response file, and `/FREEFORMAT` or `/NOFREEFORMAT`, which must appear before any other input. Separate options with a space or tab character. See Chapter 20, "Setting Linker Options" on page 347 and "Linker Options" on page 351 for more information.

object Object files to be linked. Separate multiple file names with a plus sign (+), space, or tab character. You must specify at least one object file. If you don't include an extension, the linker assumes `.obj`. Instead of entering a list of object files, you can enter a library file. The linker then links all the objects in the library into the *target* file. See "Entering Library Files As Object Files" on page 338 for more information.

target
: The file you want to create. The linker can produce either an executable file (extension `.exe`), a dynamic link library (extension `.dll`), or a device driver (extension `.sys` or `.vdd`). By default, the linker produces a .EXE file.

map
: Map file. Specify `/MAP` to generate a map file that lists the object files included in *target*. Specify `/LINENUMBERS` to include linenumber information in the map file. By default, the linker does not generate a map file. See "Generating a Map File" on page 341 for more information.

library
: Static or import libraries (.LIB files) that the linker searches for functions referenced by your object files.

Do not enter DLLs in this field (use import libraries, that represent the DLLs, instead).

Separate multiple file names with a plus sign (+), space, or tab character. By default, the linker searches the VisualAge C++ runtime libraries defined in your object files at compile time. See "Choosing Your Runtime Libraries" on page 236 for more information on setting the default libraries.

You can also specify a directory path without a file name to add that directory to the linker's search path, before any directories set in the LIB environment variable. See "Search Rules" on page 335 for more information on specifying directories.

See "Linking with .LIB Files" on page 344 and "Linking to Dynamic Link Libraries" on page 345 for more information on linking with libraries.

def_file
: The module definition file that describes the characteristics of an application or library, including imports and exports. By default, the linker does not use a .DEF file. See Chapter 21, "Creating Module Definition Files" on page 369 for more information.

Note: If the linker cannot find a file, it stops linking.

For information on the default entries for these fields, see "File Name Defaults" on page 337 .

Examples

The following command links the object files `fun.obj`, `text.obj`, `table.obj`, and `care.obj`. The linker searches for unresolved external references in the library file `xlib.lib` and in the default libraries. Since there is no name provided for the executable file, it is named `fun.exe`, taking the file name of the first object file and the default extension `.exe`. The linker also produces a map file, `funlist.map`.

```
ILINK /NOFREE /MAP fun+text+table+care, ,funlist, xlib.lib;
```

The following command links the object file fun.obj and produces the executable file `fun.exe`, as well as the map file `fun.map`.

```
ILINK /NOFREE /MAP fun;
```

The following command links the files `main.obj`, `getdata.obj`, and `printit.obj` into an executable file named `main.exe`, and produces a map file named `main.map`.

```
ILINK /NOFREE /MAP main+getdata+printit, , main;
```

The following command links `getdata.obj` and `printit.obj` into a DLL named `getdata.dll`. The module definition file `moddef.def` contains a LIBRARY statement, which tells the linker to produce a DLL.

```
ILINK /NOFREE getdata+printit,getdata.dll, , moddef
```

Responding to Linker Prompts

If you do not end the command line or response file with a semicolon (;), and not all parameters are filled in, the linker prompts you for the missing information. For each prompt, simply enter the same input that you would enter on the command line, and press **Enter**. These are the linker prompts:

```
Object Modules [.obj]:

Run File [basename.*]:

List File [targetname.map]:

Libraries [.lib]:

Definitions File [nul.def]:
```

basename defaults to the name of the first object file you specify. *targetname* defaults to the name of the run file. The prompt displays the default response in square brackets ([ ]). Press Enter to accept the default response to a prompt and continue to the next prompt. To accept default responses for all remaining prompts, enter a semicolon (;).

Starting the Linker

Notes:

1. To extend input to a new line, type a plus sign (+) as the last character on the current line. When the same prompt appears on a new line, you can continue. You cannot split a file name across lines.
2. You must enter at least one object file.
3. You can specify options anywhere on any response line. Each option must begin with a forward slash (/), or a dash (-).
4. At any prompt, you can specify responses both for that prompt and for subsequent prompts, by separating the responses with commas (,).

Using Response Files

Instead of specifying linker input on the command line, you can put options and file name parameters in a response file. You can combine the response file with options and parameters on the command line, but the command line options and parameters must appear **before** the response file; the linker ignores anything on the command line **after** the response file.

In the response file, separate responses with commas, or put each response on a separate line. Then, when you invoke the linker, use the following syntax:

```
ILINK @responsefile
```

where *responsefile* is the name of the response file. The @ symbol indicates that the file is a response file. If the file is not in the working directory, specify the path for the file as well as the file name. If the linker cannot find a file, it stops linking.

&Keyconcept.You can begin using a response file at any point on the linker command line or at any linker prompt. You can only specify one response file. Anything on the command line after the response file is ignored.

Options can appear anywhere in the response file. If an option is not valid, the linker generates an error message and stops linking.

To keep the linker from echoing the responses from the response file, specify the `/NOLOGO` option **before** the response file (or in the ILINK environment variable). You cannot turn `/NOLOGO` on or off within or after the response file.

If the file does not contain responses for all the prompts, the linker displays the appropriate prompt and waits for you to supply a response. If you end the response file with a semicolon, the linker provides default responses for the remaining prompts. However, you must provide the file name for at least one object file.

You can use special characters in the response file the same way you would use them in responses entered at the keyboard. For example, you can extend input to a new line by using the plus sign (+) and choose default responses for all remaining prompts by using a semicolon (;).

Example

Given a response file named `fun.lnk` containing the following:

```
/DEBUG /MAP +
fun text table care
fun
funlist
graf.lib;
```

When you enter:

```
ILINK /NOFREE @fun.lnk
```

the linker does the following:

- Links the four object modules `fun.obj`, `text.obj`, `table.obj`, and `care.obj` into an .EXE file named `fun.exe`.

  The linker takes the second line to specify object files (the first line clearly contains options). Because no output type is specified, the linker defaults to .EXE.
- Generates the map file `funlist.map` (assuming the extension .map).
- Preserves debugging information (because of the `/DEBUG` option).
- Links any needed routines from the library file `graf.lib`, and from the default VisualAge C++ libraries specified in the object files.
- Assumes the default for any remaining prompts (because of the semicolon after `graf.lib`), and so does not look for a module definition file.

Linking through the Compiler

When you invoke the VisualAge C++ compiler, it compiles the object files from your source code and then automatically starts the linker, to link the object files into an .EXE or .DLL file. Use the `/B` compiler option to pass options to the linker. By default, the compiler invokes the linker with the following options:

`/NOFREEFORMAT` Use the LINK386-compatible syntax, instead of the free-format syntax. See “Linking from the Command Line” on page 323 for more information on the free-format syntax. See “Using

LINK386 Syntax” on page 325 for more information on the LINK386-compatible syntax.

`/BASE:65536` Specify the starting address of the program. For .DLL files, this results in a smaller and potentially faster executable, if the specified address is free when the .DLL is loaded. For .EXE files, the OS/2 operating system always loads executable programs at 64K. You can give the linker the address 65536 (or `0x10000`) to let the linker know where the program will be loaded, so it can resolve relocation information at link time, resulting in a smaller .EXE file.

`/PMTYPE:VIO` Create program that is compatible with Presentation Manager.

In addition, some compiler options generate equivalent linker options:

`/Fb` Generate browser information. Passes `/BROWSE` to the linker.

`/Fm` Generate linker map file. Passes `/MAP` to the linker.

`/Gk` Resolve template functions in old object files. Passes `/OLDCPP` to the linker.

`/Gl` Remove unreferenced functions. Passes `/OPTFUNC` to the linker.

`/Gn` Hide default library information from the linker. Passes `/NODEFAULTLIBRARYSEARCH` to the linker.

`/Q` Suppress product information at start of compile. Passes `/NOLOGO` to the linker.

`/Ti` Generate debugger information. Passes `/DEBUG` to the linker.

See “Linker Options” on page 351 for more information on these linker options.

Passing Additional Options to the Linker

You can override these options, and pass additional options to the linker, using the `/B` compiler option. For example, to generate a map file and override the default alignment, specify

```
icc /B"/AL:256 /MAP"
```

See Chapter 15, “Setting Compiler Options” on page 253 for more information.

If you do not want the compiler to start the linker, specify the `/C` compiler option. You can then invoke the linker in a separate step.

Linking from a Make File

Use a make file to organize the sequence of actions (such as compiling and linking) required to build your project. You can then invoke all the actions in one step. The NMAKE utility saves you time by performing actions on only the files that have changed, and on the files that incorporate or depend on the changed files. 📖 See Chapter 58, "Program Maintenance Utility (NMAKE)" on page 815 for more information.

You can write the make file yourself, or you can use WorkFrame to manage the make file. When you build through WorkFrame, a make file is created and maintained automatically.

Starting the Linker

17 Optimized Linking

Removing Unreachable Functions

Just as the compiler can optimize your source code by removing or replacing instructions, the linker can optimize your object code, including code in libraries you are linking in, by removing unreferenced functions. When the function is removed, any code that was required only by that function is also removed, including any other functions that were referenced only by that function. This reduces the size of your output file.

Link with the option `/OPTFUNC` to remove functions that are:

- Not referenced in any input file
- Rendered unreferenced by the removal of other functions
- Not exported for use in other files

See "EXPORTS" on page 379 for more information on exporting functions and data.

If you are compiling and linking in one step, you can use the `/Gl` compiler option to invoke this optimisation.

Performance Consideration

Optimized linking generally takes longer than regular linking, because of the extra processing that the linker performs. However, if the optimization is effective enough, it can actually speed up the linking process, because there is less information to write to file. Generally, you may want to link without the `/OPTFUNC` option, until your code is tested and stable.

Packing Executables

Specify `/EXEPACK` to reduce the size of the executable by compressing pages in the file. The operating system automatically decompresses the pages when the program is loaded. If your program is intended to run only on OS/2 version 3.0 or later, then specify `/EXEPACK:2` for best results. If your program is also intended to run on older versions of OS/2, specify `/EXEPACK:1`.

Specify `/PACKCODE` to produce slightly faster and more compact code by grouping neighboring code segments that have similar attributes.

Specify `/PACKDATA` to produce more compact files by grouping neighboring data segments that have similar attributes.

Optimized Linking

Specify /DBGPACK when you are debugging, to reduce the size of the executable file and potentially improve debugger performance.

📖 See “Linker Options” on page 351 for more information on these and other linker options.

18

Input and Output

The linker takes object files, links them with each other and with any library files you specify, and produces an executable output file. The executable output can be either an executable program (extension `.exe`) file, a dynamic link library (extension `.dll`), or a device driver (extension `.sys` or `.vdd`).

The linker optionally produces a map file, which provides information about the contents of the executable output.

Input	**Output**
options	*executable file* (.EXE, .DLL, .SYS, or .VDD)
object files (*.OBJ)	*map file* (.MAP)
library files (*.LIB)	*return code*
import libraries (*.LIB)	
module definition file (.DEF)	

Search Rules

When searching for an object (.OBJ), library (.LIB), or module definition (.DEF) file, the linker looks in the following locations in this order:

1. The directory you specified for the file, or the current directory, if you did not give a path. Default libraries do not include path specifications.

   Note: If you specify a path with the file, the linker searches only that path, and stops linking if the file cannot be found there.

2. Any directories entered by themselves on the command line (they must end with a slash (/) or backslash (\) character and, if you specified `/NOFREE`, they must be in the *libraries* parameter). See "Specifying Directories" on page 336 for more information.

3. Any directories listed in the LIB environment variable.

If the linker cannot locate a file, it generates a fatal error message and stops linking.

Linker Input and Output

Example

If you respond to linker prompts as follows:

```
ILINK /NOFREE
Object Modules [.obj]: FUN TEXT TABLE CARE
Run File [fun.*]:
List File [fun.map]:
Libraries [.lib]: NEWLIBV3 C:\TESTLIB\
Definitions File [nul.def]:
```

The linker links four object files to create an executable file named FUN.EXE. The linker searches NEWLIBV3.LIB before searching the default libraries to resolve references.

To locate NEWLIBV3.LIB and the default libraries, the linker searches the following locations in this order:

1. The current directory (because NEWLIBV3.LIB was entered without a path)
2. The C:\TESTLIB\ directory
3. The directories listed in the LIB environment variable

Specifying Directories

To have the linker search additional directories for input files, specify a drive or directory by itself on the command line. Specify the drive or directory with a slash (/) or backslash (\) character at the end for the linker to recognize it as a path.

Note: If you specified `/NOFREE`, then you can only specify directories in the *`library`* parameter at the command line, or in response to the `Libraries [.LIB]:` prompt. You must still end each directory with a slash (/) or or backslash (\) character.

The paths you specify are searched before the paths in the LIB environment variable. See “Search Rules” on page 335 for more information.

File Name Defaults

If you do not enter a file name, the linker assumes the defaults shown below. If you specify /NOFREE, the linker also assumes default file extensions for files without extensions.

Figure 71. Linker File Name Defaults

File	Default File Name	Default Extension
Object files	None. You must enter at least one object file name.	.OBJ
Output file	The base name of the first object file.	.EXE
Map file	The base name of the output file.	.MAP
Library files	The default libraries defined in the object files. Use compiler options to define the default libraries. See “Choosing Your Runtime Libraries” on page 236 for more information. Any additional libraries you specify are searched **before** the default libraries.	.LIB
Module definition file	None. The linker assumes you accept the default for all module statements.	.DEF

Specifying Object Files

When you invoke the linker from the command line, the linker assumes that any input it cannot recognize as other files, options, or directories must be a object file. Use a space or tab character to separate files. See “Linking from the Command Line” on page 323 for more information on how the linker interprets input.

If you specified /NOFREE to use the LINK386-compatible syntax, then the first set of file names you give it are taken as object files, up to the first comma. Use a plus (+), space, or tab as a separator between the file names. If you do not specify an extension, the linker assumes the .OBJ extension.

When you invoke the linker through the compiler, the compiler automatically passes the object files it creates to the linker, as well as passing any object files you specify on the compiler command line or in the ICC environment variable.

The linker accepts object files compiled or assembled:

- In 16- or 32-bit OMF format
- For OS/2 version 1.0 or higher
- For the 80286 (16-bit only), 80386, 80486, and Pentium microprocessors

You must enter at least one object file.

Linker Input and Output

Note: If you are linking with the LINK386-compatible syntax (`/NOFREE`), then you can also pass the linker a library file, in place of an object file. See “Using LINK386 Syntax” on page 325 and “Entering Library Files As Object Files” for more information.

Entering Library Files As Object Files

If you specify `/NOFREE` to use LINK386-compatible syntax, then you can enter library files in place of object files in the *`object`* parameter on the command line or at the `Object Modules [.OBJ]:`. prompt. Be sure to include the .LIB file name extension; otherwise, the linker assumes a `.obj` extension.

When you enter a library as an object file, all the modules in the library are added to your output file, just as if you had entered all of the library's modules as object files in the *`object`* parameter.

In contrast, when you enter a library in the *`library`* parameter, The linker links only to those modules needed to resolve external references.

If you are linking with the `/FREEFORMAT` option (the default), then you cannot enter library files as object files.

Specifying Executable Output Type

You can use the linker to produce executable modules (with the extension .EXE), dynamic link libraries (with the extension .DLL), and device drivers (with the extension .SYS or .VDD). The linker produces .EXE files by default.

Use options, or statements in the module definition (.DEF) file, to specify what kind of output you want:

- To produce an .EXE, specify the `/EXEC` option, or include the module statement NAME.
- To produce a .DLL, specify the `/DLL` option, or include the module statement LIBRARY.
- To produce a device driver, specify the `/PDD` or `/VDD` option, or include the module statement PHYSICAL DEVICE or VIRTUAL DEVICE.

For more information on using .DEF files, see Chapter 21, “Creating Module Definition Files” on page 369.

Producing an .EXE File

The linker produces .EXE files by default. Use the `/EXEC` option, or the `NAME` module statement, to explicitly identify the output file as an .EXE file.

An .EXE file is one that can be executed directly: you can run the program by typing the name of the file. In contrast, DLL and device driver programs execute when they are called by other processes, and cannot be run independently.

To reduce the size of the .EXE file and improve its performance, use the following options:

- `/ALIGNMENT:value` to set the alignment factor in the output file. Set *`value`* to smaller factors to reduce the size of the executable, and to larger factors to reduce load time for the executable. By default, the alignment is set to `512`.
- `/BASE:0x00010000` to specify the load address for the executable. The load address must be `0x00010000`. Any other value will produce a warning, and will not be used. Specifying this value explicitly allows the linker to omit relocation records, which can result in a smaller executable.
- `/EXEPACK` to compress the file.

  Note: Set `/EXEPACK:2` for executables that will run only on OS/2 v3.0 and later.
- `/OPTFUNC` (once your code is tested and stable) to remove unreachable functions.

If you do not specify an extension for the output file name, the linker automatically adds the extension `.exe` to the name you provide. If you do not specify an output file name at all, the linker generates an .EXE file with the same file name as the first .OBJ file it linked.

If you are using a module definition (.DEF) file, you can include a NAME statement to provide a name for the application. The .DEF file should **not** contain a LIBRARY, VIRTUAL DEVICE, or PHYSICAL DEVICE statement. For more information on using .DEF files, see Chapter 21, “Creating Module Definition Files” on page 369.

Producing a Dynamic Link Library

A dynamic link library (.DLL) file contains executable code for common functions, just as an ordinary library (.LIB) file does. However, when you link with a DLL (using an import library), the code in the DLL is **not** copied into the executable (.EXE) file. Instead, only the import definitions for DLL functions are copied. At run time, the dynamic link library is loaded into memory, along with the .EXE file.

Linker Input and Output

To produce a DLL as output, compile the object files with the /Ge- compiler option, and link them with the /DLL linker option. If you are using a module definition (.DEF) file, specify the LIBRARY statement in the .DEF file. For more information on using .DEF files, see Chapter 21, “Creating Module Definition Files” on page 369.

To reduce the size of the DLL and improve its performance, use the following options:

- /ALIGNMENT:value to set the alignment factor in the output file. Set *value* to smaller factors to reduce the size of the DLL, and to larger factors to reduce load time for the DLL. By default, the alignment is set to 512.
- /EXEPACK to compress the file.

  Note: Set /EXEPACK:2 for DLLs that will run only on OS/2 v3.0 and later.
- /OPTFUNC (once your code is tested and stable) to remove unreachable functions.

If you use the /BASE option, set /BASE:0x12000000 (or a lesser value), and provide a separate value for each DLL. For DLLs, setting a /BASE value can save load time when the given load address is available. If the load address is not available, the /BASE value is ignored, and there is no load time benefit.

Use the **_Export** keyword, **#pragma export**, or the EXPORTS statement in a module definition file, to specify the functions and data you want to make available to other executables. See “EXPORTS” on page 379 for more information.

Once you have produced the DLL, you can produce an executable that links to the DLL. This process can be done in two ways:

Using a .DEF file
Provide a .DEF file when you create the executable. In the .DEF file, use the IMPORTS statement to specify which of the DLL's functions your object files need. See “Linking to a DLL Using a .DEF File” on page 346.

Using an import library.
Use the IMPLIB utility to create an import library. When you use an import library, you no longer need to use the IMPORTS statement. The linker determines which functions your object files need during the linking process. See “Linking to a DLL Using an Import Library” on page 346.

Producing a Device Driver

You can produce both physical device drivers and virtual device drivers:

- A physical device driver (.SYS) allows the operating system to interact with a system peripheral, such as a monitor or printer.
- A virtual device driver (.VDD) allows the operating system to handle input and output with multiple DOS or WIN-OS/2 sessions. Each session can then act as if it has complete control of the input or output device, while actually sharing the control with other sessions. See OS/2 System help (accessible from the desktop popup menu) for more information on virtual device drivers.

To produce a device driver (.SYS or .VDD) file as output, specify the `/PDD` or `/VDD` options, or specify the PHYSICAL DEVICE or VIRTUAL DEVICE statement in your module definition (.DEF) file.

If you are creating a physical device driver, use the SEGMENTS statement in your .DEF file to specify which segments have I/O privilege.

If you are creating a virtual device driver, compile with the options `/Gr+` and `Rn+`, and use the subsystem libraries. See the *Programming Guide* for more information on creating virtual device drivers.

For more information on using .DEF files, see Chapter 21, "Creating Module Definition Files" on page 369.

Generating a Map File

Specify `/MAP` to generate a map file, which lists the object modules in your output file; segment names, addresses, and sizes; and symbol information. If you do not specify a name for the map file, the map file takes the name of the executable output file, with the extension `.map`.

Note: If you are linking with the `/NOFREE` option, you can specify a name for the map file in the *`map`* parameter. Any name you specify with `/MAP` option, is ignored. See "Using LINK386 Syntax" on page 325 for more information on using `/NOFREE`.

To prevent the map file from being generated, specify the /NOMAP option.

Specify `/LINENUMBERS` to include source file line numbers and associated addresses in the map file.

See "Linker Options" on page 351 for more information on these and other options.

Linker Return Codes

The linker has the following return codes:

Code	Meaning
0	The link was completed successfully. The linker detected no errors, and issued no warnings.
4	**Warnings** issued. There may be problems with the output file.
8	**Errors** detected. The linking might have completed, but the output file cannot be run successfully.
12	Both warnings issued and errors detected (see return codes 4 and 8)
16	**Severe errors** detected. Linking ended abnormally, and the output file cannot be run successfully.
20	Both warnings issued and severe errors detected (see return codes 4 and 16)
24	Both errors and severe errors issued (see return codes 8 and 16)
28	The linker issued warnings, detected errors, and detected severe errors (see return codes 4, 8, and 16)

If you invoke the linker through the compiler, then no return code is issued for warnings. If you invoke the linker through a make file, you can force NMAKE to ignore warnings by putting `-7` before the `ILINK` command.

19 Linking with Library Files

The linker searches library files to resolve references in your object files to functions outside the object files. The library files contain object modules, which contain the code for functions your object files can reference. Some libraries are searched by default: the compiler embeds the names of default libraries into object files. See "Choosing Your Runtime Libraries" on page 236 for more information on choosing your default libraries.

There are two kinds of library files: static libraries, and dynamic link libraries (DLLs) (which you link to through import libraries). The default libraries are static library files, unless you specify the compiler option `/Gd` so you can link to the DLL versions of the default libraries. In static linking, you link to .LIB files that contain the object modules you need. In dynamic linking, you link to import libraries (also .LIB files) that contain import definitions for functions in the object modules; the object modules themselves are in .DLL files.

Static linking means that code for all the VisualAge C++ runtime functions called in your program is copied from a .LIB file into your output .EXE or .DLL file. The .EXE or .DLL files will be larger because there is a copy of the runtime functions in each file. These programs will take up more storage, and if you run them at the same time, there will also be a copy of the library functions in memory for each program. Statically linked programs, however, are easier to distribute because the library functions are part of your executable file.

Dynamic linking means that code for the VisualAge C++ runtime functions called in your program is **not** copied into your output .EXE or .DLL file. Instead, the function code stays in a separate VisualAge C++ DLL file, and your calls to the function are resolved at load time. The amount of disk space required by your .EXE or .DLL file is reduced, and there is only one copy of the library functions in memory for all programs that use them. Dynamically linked programs can be harder to distribute, since the separate DLL file must be distributed along with your executable file. They can also be easier to maintain, because you do not necessarily need to relink whenever there is a change in one of the DLLs you use.

Use the `/Gd` compiler option to control whether your executable file links to the runtime library statically or dynamically.

The default is `/Gd-`, which statically links with the .LIB version of the runtime library.

Specify /Gd+ to dynamically link with the DLL version of the runtime library.

The compiler option you choose causes the corresponding library to be linked in by default. If you override the default libraries with the /NOD linker option, you must explicitly give the name of all libraries you are using on the linker command line.

If you are linking with libraries other than the default libraries, you can specify the libraries when you link. The libraries you specify at the linking step are searched before the default libraries.

Linking with .LIB Files

The linker uses library (.LIB) files to resolve external references from code in the object (.OBJ) files. When the linker finds a function or data reference in a .OBJ file that requires a module from a library, the linker links the module from the library to the output file.

The compiler embeds the names of needed libraries (called default libraries) in object files. You do not need to specify these libraries: the linker searches them automatically.

Specify library files only when:

- You want to use libraries other than the default libraries. Libraries you specify are searched ahead of the default libraries.
- You want to specify the default library with its full path, because the default library is not in the current directory, and not in a directory specified in the LIB environment variable (see "Search Rules" on page 335 for more information). You can also specify a path to a directory, without specifying a file name, to have the linker search that directory before the directories in the LIB environment variable (see "Specifying Directories" on page 336 for more information).

To ignore the default libraries, use the /NODEFAULTLIBRARYSEARCH option. Use the option with care, because most compilers expect their object files to be linked with default libraries.

If you want to include all of a library's objects in the output file, instead of only the required ones, you must link with the /NOFREE option, to use the LINK386-compatible syntax. You can then enter the library name as an object file. See "Entering Library Files As Object Files" on page 338 for more information.

Linking to Dynamic Link Libraries

A dynamic link library (DLL) file contains executable code for common functions, just as an ordinary library (.LIB) file does. However, when you link with a DLL, the code in the DLL is **not** copied into the executable (.EXE) file. Instead, only the import definitions for DLL functions are copied. At run time, the dynamic link library that contains the functions is loaded into memory, along with the .EXE file that references them.

There are two ways to link with a DLL:

- **Using a .DEF file** to define functions your object files import from the DLL. 📖 See “Linking to a DLL Using a .DEF File” on page 346.
- **Using an import library**, created with the IMPLIB utility, that allows the linker to determine which functions your object files need to import from the DLL. 📖 See “Linking to a DLL Using an Import Library” on page 346.

Advantages of Using DLLs

Size
Applications require less disk space. With dynamic linking, applications that use the same function can share a single copy of the function, rather than each application having its own copy of the function.

Independence
Dynamic link libraries and applications stay independent. You can update dynamic link libraries without relinking the applications that use them (as long as the function names and parameters are preserved). If you use third-party DLLs, this is particularly convenient, because you can update the third-party DLL without recompiling or relinking your programs. At run time, your applications automatically call the updated function code from the new DLL.

Sharing
When appropriate, code and data segments loaded from a dynamic link library can be shared by the different applications that access the DLL. Without dynamic linking, such sharing is not possible because each file has its own copy of all the code and data it uses. By sharing segments with dynamic linking, applications can use memory more efficiently.

Linking Speed
Your applications link more quickly, because the linker does not have to copy code from the library into your program.

Linking to DLLs

Linking to a DLL Using a .DEF File

To use functions and data from a DLL, complete the following steps:

1. Create a module definition (.DEF) file for your executable output file.
2. In the .DEF file, use the IMPORTS statement to specify the DLL your executable will link to, and what functions and data in the DLL your executable references.

Linking using a .DEF file requires you to define the functions and data you want to import.

If you link using an import library, the linker determines the functions and data you need to import. The linker imports only the functions and data necessary to resolve references.

Linking to a DLL Using an Import Library

An import library contains information on all the functions and data exported by a given DLL. Use the IMPLIB utility to create an import library, as follows:

- Use IMPLIB directly on the DLL.
- Use IMPLIB on the .DEF file for the DLL. If the DLL exports were defined in a .DEF file (with the EXPORTS statement), IMPLIB can create an import library from the .DEF file.

Once you have created the import library, convert the library to the new library format:

```
ILIB /CONV /NOE /NOBR mylib.lib
```

This improves linking performance.

You can then use the import library with the linker, when you generate executables that reference functions in the DLL. Enter the import library in the *library* parameter on the command line.

The advantage of using an import library is that you do not need to define which functions are imported from the DLL. The linker imports only those functions your application needs. See the IMPLIB User's Guide for more information on creating import libraries.

Setting Linker Options

Linker options are not case sensitive, so you can specify them in lower-, upper-, or mixed case. You can also substitute a dash (-) for the slash (/) preceding the option. For example, -DEBUG is equivalent to /DEBUG. You can specify options in either a short or long form. For example, /DE, /DEB, and /DEBU are all equivalent to /DEBUG. See “Summary of Linker Options” on page 350 for the shortest acceptable form for each option. Lower- and uppercase, short and long forms, dashes, and slashes can all be used on one command line, as in:

```
ilink /de -DBGPACK -Map /NOI prog.obj
```

Separate options with a space or tab character. You can specify compiler options in the following ways:

- On the command line
- In the ILINK environment variable
- In WorkFrame

Options specified on the command line override the options in the ILINK environment variable.

Some linker options take numeric arguments. You can enter numbers in decimal, octal, or hexadecimal format using standard C-language syntax. See “Specifying Numeric Arguments” on page 349 for more information.

Setting Options on the Command Line

Linker options specified on the command line override any previously specified in the ILINK environment variable (as described in “Setting Options in the ILINK Environment Variable” on page 348).

You can specify options anywhere on the command line. Separate options with a space or tab character.

For example, to link an object file with the /MAP option, enter:

```
ilink /M myprog.obj
```

Setting Linker Options

Setting Options in the ILINK Environment Variable

Store frequently used options in the ILINK environment variable. This method is useful if you find yourself repeating the same command-line options every time you link. You cannot specify file names in the environment variable, only linker options.

The ILINK environment variable can be set either from the command line, in a command (`.CMD`) file, or in the CONFIG.SYS file. If it is set on the command line or by running a command file, the options will only be in effect for the current session (until you reboot your computer). If it is set in the CONFIG.SYS file, the options are set when you boot your computer, and are in effect every time you use the linker unless you override them using a `.CMD` file or by specifying options on the command line.

See “OS/2 Environment Variables for Compiling” on page 207 for more information about using ILINK and other environment variables.

Example

In the following example, options on the command line override options in the environment variable. If you enter the following commands:

```
SET ILINK=/NOI /AL:256 /DE
ILINK test
ILINK /NODEF /NODEB prog
```

The first command sets the environment variable to the options /NOIGNORECASE, /ALIGNMENT:256, and /DEBUG

The second command links the file `test.obj`, using the options specified in the environment variable, to produce `test.exe`

The last command links the file `prog.obj` to produce `prog.exe`, using the option /NODEFAULTLIBRARYSEARCH, in addition to the options /NOIGNORECASE and /ALIGNMENT:256. The /NODEBUG option on the command line overrides the /DEBUG option in the environment variable, and the linker links without the /DEBUG option.

Setting Options in the WorkFrame Environment

If you have installed the WorkFrame product, you can set linker options using the options dialogs. You can use the dialogs when you create or modify a project.

Options you select while creating or changing a project are saved with that project.

For more information on setting options and linking with WorkFrame, see “Linking within WorkFrame” on page 322.

Specifying Numeric Arguments

Some linker options and module statements take numeric arguments. The linker accepts numeric arguments in C-language syntax. You can specify numbers in any of the following forms:

Decimal Any number **not** prefixed with 0 or 0x is a decimal number. For example, 1234 is a decimal number.

Octal Any number prefixed with 0 (but not 0x) is an octal number. For example, 01234 is an octal number.

Hexadecimal Any number prefixed with 0x is a hexadecimal number. For example, 0x1234 is a hexadecimal number.

Summary of Linker Options

Figure 72 (Page 1 of 2). Linker Options Summary

Syntax	Description	Default	Page
/?	Display help	None	351
/A[LIGNMENT]:*factor*	Set alignment factor	/A:512	351
/BAS[E]:*address* /NOBAS[E]	Set preferred loading address	/BAS:0x00010000	352
/BR[OWSE] /NOBR[OWSE]	Add browse information	/NOBR	352
/C[ODEVIEW] /NOC[ODEVIEW]	Include debugging information	/NOC	353
/DB[GPACK] /NODB[GPACK]	Pack debugging information	/NODB	353
/DE[BUG] /NODEB[UG]	Include debugging information	/NODEB	354
/DEF[AULTLIBRARYSEARCH] /NOD[EFAULTLIBRARYSEARCH][:*lib*]	Search default libraries	/DEF	354
/DLL	Generate DLL	/EXEC	355
/EXEC	Generate .EXE file	/EXEC	355
/E[XEPACK][:1|:2] /NOEXE[PACK]	Compress data	/NOEXE	356
/EXT[DICTIONARY] /NOE[XTDICTIONARY]	Use extended dictionary to search libraries	/EXT	357
/Fo[RCE] /NOFO[RCE]	Create executable output file even if errors	/NOFO	357
/FR[EEFORMAT] /NOFR[EEFORMAT]	Use free format command line syntax	/FR	357
/H[ELP]	Display help	None	358
/IG[NORECASE] /NOI[GNORECASE]	Ignore capitalization in identifiers	/NOI	358
/I[NFORMATION] /NOIN[FORMATION]	Display status of linking process	/NOIN	358
/L[INENUMBERS] /NOLI[NENUMBERS]	Include line numbers in map file	/NOLI	359
/LO[GO] /NOL[OGO]	Display logo, echo response file	/LO	359
/M[AP][:[*dir*][*name*]] /NOM[AP]	Generate map file	/NOM	360
/OLD[CPP] /NOOLD[CPP]	Ignore template resolution directives in old object files	/NOOLD	360

Figure 72 (Page 2 of 2). Linker Options Summary

Syntax	Description	Default	Page
/OPTF[UNC] /NOOPTF[UNC]	Remove unreferenced functions	/NOOPTF	361
/O[UT][:[*dir*]*name*	Name output file	Name of first .OBJ file	361
/PACKC[ODE][:*number*] /NOP[ACKCODE]	Pack neighboring code segments with similar attributes	/PACKC:0xFfffFfff	362
/PACKD[ATA][:*number*] /NOPACKD[ATA]	Pack neighboring data segments with similar attributes	/PACKD:0xFfffFfff	363
/PDD	Generate physical device driver	/EXEC	363
/PM[TYPE]:*type*	Specify application type	None	363
/SEC[TION]:*name*,*attribs*	Set attributes for segment	Accept default attributes	364
/SE[GMENTS]:*number*	Set maximum number of segments	/SE:128	365
/ST[ACK]:*size*	Set stack size of application	None	366
/VDD	Generate virtual device driver	/EXEC	367

Linker Options

/?

Syntax:	Default:
/?	None

Use /? to display a list of valid linker options. This option is equivalent to /HELP.

/ALIGNMENT

Syntax:	Default:
/A[LIGNMENT]:*factor*	/ALIGNMENT:512

Use /ALIGNMENT to set the alignment factor in the .EXE or .DLL file.

The alignment factor determines where pages in the .EXE or .DLL file start. From the beginning of the file, the start of each page is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 1 to 4096.

The default alignment is 512 bytes.

/BASE, /NOBASE

Syntax:	**Default:**
`/BAS[E]:`*`address`*	`/BASE:0x00010000`
`/NOBASE`	

Use `/BASE` to specify the preferred load address for the first load segment of a .DLL file. The number you specify in *`address`* is rounded up to the nearest multiple of 64K. The second load segment is then loaded at the next available multiple of 64K, and so on.

If the file's load segments cannot be loaded beginning at this preferred address, then the preferred address is ignored and the objects are loaded according to the internal relocation records retained in the file data.

For .EXE files, use the default base address of 64K (`/BASE:0x00010000`). Specifying this address explicitly can slightly reduce the size of the executable. Any other address will result in a warning, and 64K will be used anyway.

This option has the same effect as the BASE module definition file statement. If you specify both the BASE statement and the `/BASE` option, the statement value overrides the option value.

Specify `/NOBASE` to retain relocation records and emit internal fixups, when you generate an .EXE file. This does not affect the actual base address, or interfere with any value you specified with `/BASE`. You can specify both options.

/BROWSE, NOBROWSE

Syntax:	**Default:**
`/BR[OWSE]`	`/NOBROWSE`
`/NOBR[OWSE]`	

Use `/BROWSE` to add browse information to the load module the linker generates. `/BROWSE` is automatically turned on when you specify `/DEBUG`. This option is only effective if the object files are compiled with the `/Fb` or `/Fb*` compiler option.

The browse information is used by the VisualAge C++ Browser when you browse your program.

See “Creating Files to Use with the Browser” on page 559 for more information.

If you are compiling and linking in one step, specifying /Fb automatically passes /BROWSE to the linker.

/CODEVIEW, NOCODEVIEW

Syntax:	**Default:**
/C[ODEVIEW] /NOC[ODEVIEW]	/NOCODEVIEW

Obsolete: These options will not be available in future releases of the linker. Use /DEBUG, /NODEBUG instead.

Use /CODEVIEW to include debug information in the output file, so you can debug the file with the debugger, or trace its execution with the Performance Analyzer. The linker will embed symbolic data and line number information in the output file.

For debugging, compile the object files with /Ti.

For Performance Analyzer, compile the object files with /Ti and /Gh.

When you specify /CODEVIEW, /BROWSE is turned on by default.

/CODEVIEW provides the same functionality as /DEBUG, and is provided only for purposes of compatibility.

Note: Linking with /CODEVIEW or /DEBUG increases the size of the executable output file.

📖 See Part 6, “IBM VisualAge C ++ Debugger” on page 393 for more information.

/DBGPACK, /NODBGPACK

Syntax:	**Default:**
/DB[GPACK] /NODB[GPACK]	/NODBGPACK

Use /DBGPACK to eliminate redundant debug type information. The linker takes the debug type information from all object files and needed library components, and reduces the information to one entry per type. This results in a smaller executable output file, and can improve debugger performance.

Performance Consideration: Generally, linking with /DBGPACK slows the linking process, because it takes time to pack the information. However, if there is enough redundant debug type information, /DBGPACK can actually speed up your linking, because there is less information to write to file.

You can only pack debug information in objects created with version 3.0 of the compiler or later. If you use /DBGPACK with older object files, the linker generates a warning and does not pack the debug information.

When you specify /DBGPACK, /DEBUG and /BROWSE are turned on by default.

/DEBUG, /NODEBUG

Syntax:	**Default:**
/DE[BUG]	/NODEBUG
/NODEB[UG]	

Use /DEBUG to include debug information in the output file, so you can debug the file with the debugger, or analyze its performance with the performance analyzer. The linker will embed symbolic data and line number information in the output file.

For debugging, compile the object files with /Ti.

For Performance Analyzer, compile the object files with /Ti and /Gh.

When you specify /DEBUG, /BROWSE is turned on by default.

Note: Linking with /DEBUG increases the size of the executable output file.

See Part 6, “IBM VisualAge C ++ Debugger” on page 393 for more information.

/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH

Syntax:	**Default:**
/DEF[AULTLIBRARYSEARCH	/DEFAULTLIBRARYSEARCH
/NOD[EFAULTLIBRARYSEARCH][:*library*]	

Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references. The default libraries for an object file are defined at compile time, and embedded in the object file. The linker searches the default libraries by default.

Use /NODEFAULTLIBRARYSEARCH to tell the linker to ignore default libraries when it resolves external references. If you specify a *library* with the option, the linker ignores that default library, but searches any others that are defined in the object files.

If you specify /NODEFAULTLIBRARYSEARCH, then you must explicitly specify all the libraries you want to use, including VisualAge C++ runtime libraries and any OS/2 libraries you need.

📖 See “Choosing Your Runtime Libraries” on page 236 for more information on defining the default libraries when you compile, and 📖 see “Linking with .LIB Files” on page 344 for more information on explicitly specifying libraries when you link.

If you are compiling and linking in one step, specifying /Gn automatically passes /NODEFAULTLIBRARYSEARCH to the linker.

/DLL

Syntax:	**Default:**
/DLL	/EXEC

Use /DLL to identify the output file as a dynamic link library (.DLL file). You can also identify the output file as a DLL with the LIBRARY statement in a module definition file. For more information on generating a DLL, see 📖 “Producing a Dynamic Link Library” on page 339 . The object files should be compiled with /Ge-

If you specify /DLL with any of /EXEC, /PDD, or /VDD, then only the last specified of the options takes effect.

If you do not specify /DLL, or any of the other options above, then by default the linker produces an .EXE file (/EXEC).

/EXEC

Syntax:	**Default:**
/EXEC	/EXEC

Use /EXEC to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default. You can also identify the output as an .EXE file with the NAME statement in a module definition file. For more information on generating an .EXE file, 📖 see “Producing an .EXE File” on page 339. The object files should be compiled with /Ge+.

If you specify /EXEC with any of /DLL, /PDD, or /VDD, then only the last specified of the options takes effect.

/E, /NOEXE Options

If you do not specify /EXEC, or any of the other options above, then the linker produces an .EXE file by default.

/EXEPACK, /NOEXEPACK

Syntax:
/E[XEPACK][:1|:2]
/NOEXE[PACK]

Default:
/NOEXEPACK

Use /EXEPACK to reduce the size of the executable by compressing pages in the file. The operating system automatically decompresses the pages when the program runs.

Specify /EXEPACK[:1] to compress data segments in your output file, using run-length encoding compression. If compression does not reduce the size of the segment, the linker does not compress that segment.

Specify /EXEPACK:2 to compress both data and code segments, as follows:

- For data segments, the linker tries both LZW compression and run-length encoding compression, and uses the method with the more efficient result.
- For code segments, the linker uses LZW compression.

Segments are evaluated one page at a time. If compression does not reduce the size of the page, the page is not compressed.

OS/2 v3.0 only!: Only set /EXEPACK:2 if you are developing for OS/2 version 3.0 or later. OS/2 version 2.1 or earlier cannot run programs that have been compressed with /EXEPACK:2.

Linking and compressing generally takes longer than linking alone, because of the extra time spent compressing. However, if the compression is effective enough, it can actually speed up the linking process, because there is less information to write to file.

By default, the linker does not compress the output file.

/EXTDICTIONARY, /NOEXTDICTIONARY

Syntax:	**Default:**
/EXT[DICTIONARY] /NOE[XTDICTIONARY]	/EXTDICTIONARY

Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is an internal list of symbol locations included with libraries. The linker searches this list by default, to speed up the linking process.

Use /NOEXTDICTIONARY if you are linking to an import library, or if you have redefined a symbol used in the extended dictionary. Otherwise the linker issues error L2044 because you have defined the same symbol in two different places. When you link with /NOEXTDICTIONARY, the linker searches the dictionary directly, instead of searching the extended dictionary. This results in slower linking, because references must be resolved individually.

/FORCE

Syntax:	**Default:**
/FO[RCE] /NOFO[RCE]	/NOFO

Use /FORCE to produce an executable output file even if there are errors during the linking process.

By default, the linker does not produce an executable output file if it encounters an error.

/FREEFORMAT, /NOFREEFORMAT

Syntax:	**Default:**
/FR[EEFORMAT] /NOFR[EEFORMAT]	/FR

Use /FREEFORMAT to allow free placement of files, options, and directories on the command line, separated by space or tab characters. Use the /OUT option to name the executable output file. Use the /MAP option to name the map file. Library and definition files are identified by their extension.

/FREEFORMAT is in effect by default. For more information on the /FREEFORMAT syntax, see “Linking from the Command Line” on page 323.

Use /NOFREEFORMAT to allow a LINK386-compatible command line syntax, in which different types of file are grouped and separated by commas. If you specify /NOFREEFORMAT, then you cannot specify /OUT. Instead, specify a name for the executable output file in the appropriate place in the command line syntax. See “Using LINK386 Syntax” on page 325 for more information.

/HELP

Syntax:	**Default:**
/H[ELP]	None

Use /HELP to display a list of valid linker options. This option is equivalent to /?.

/IGNORECASE, /NOIGNORECASE

Syntax:	**Default:**
/IG[NORECASE] /NOI[GNORECASE]	/NOIGNORECASE

Use /IGNORECASE to turn off case sensitivity, ignoring capitalization in identifiers. For example, with this option on, the linker treats ABC, abc, and Abc as equivalent.

By default, the linker is case sensitive, and would treat ABC, abc, and Abc as unique names.

/INFORMATION, /NOINFORMATION

Syntax:	**Default:**
/I[NFORMATION] /NOIN[FORMATION]	/NOINFORMATION

Use /INFORMATION to have the linker display information about the linking process as it occurs, including the phase of linking and the names and paths of the object files being linked.

If you are having trouble linking because the linker is finding the wrong files or finding them in the wrong order, use /INFORMATION to determine the locations of the object files being linked and the order in which they are linked.

The output from this option is sent to **stdout**. You can redirect the output to a file using OS/2 redirection symbols.

/LINENUMBERS, /NOLINENUMBERS

Syntax:	**Default:**

`/L[INENUMBERS]`
`/NOLI[NENUMBERS]`
	`/NOLINENUMBERS`

Use `/LINENUMBERS` to include source file line numbers and associated addresses in the map file. For this option to take effect, there must already be line number information in the object files you are linking. When you compile, use the `/Tn` option to include line numbers in the object file (or the `/Ti` option, to include all debugging information). If you give the linker an object file without line number information, the `/LINENUMBERS` option has no effect.

The `/LINENUMBERS` option forces the linker to create a map file, even if you specified `/NOMAP`.

For more information on map files, 📖 see "Generating a Map File" on page 341.

By default, the map file is given the same name as the output file, plus the extension `.map`. You can override the default name by specifying a map file name.

/LOGO, /NOLOGO

Syntax:	**Default:**

`/LO[GO]`
`/NOL[OGO]`
	`/LOGO`

Use `/NOLOGO` to suppress the product information that appears when the linker starts. `/NOLOGO` also stops the contents of the response file from being echoed to the screen.

Specify `/NOLOGO` before the response file on the command line, or in the ILINK environment variable. If the option appears in or after the response file, it is ignored.

If you are compiling and linking in one step, you can use the `/Q` compiler option to suppress the product information, and stop the contents of the response file from being echoed to the screen.

By default, the linker displays product information at the start of the linking process, and displays the contents of the response file as it reads the file.

/MAP, /NOMAP

Syntax:	**Default:**

`/M[AP][:[`*`dir`*`][`*`name`*`]]`
`/NOM[AP]`
	`/NOM`

Use `/MAP` to generate a map file with the name *`name`*, and in the directory *`dir`*, that lists the composition of each segment, and the public (global) symbols defined in the object files. The symbols are listed twice: in order of name, and in order of address.

If you do not specify *`dir`*, the map file is generated into the current working directory. If you do not specify *`name`*, the map file has the same name as the executable output file, with the extension `.map`.

For compatibility with LINK386, you can specify `/MAP:full`. With the VisualAge C++ linker, this is the same as specifying `/MAP`.

Note: If you are linking with the `/NOFREE` option, you can specify a name for the map file in the *`map`* parameter. Any name you specify with the `/MAP` option is ignored. See “Using LINK386 Syntax” on page 325 for more information on using `/NOFREE`.

For more information on map files, see “Generating a Map File” on page 341.

If you are compiling and linking in one step, you can use the `/Fm` compiler option to generate a linker map file.

By default, the linker does not produce a map file.

/OLDCPP, /NOOLDCPP

Syntax:	**Default:**

`/OLD[CPP]`
`/NOOLD[CPP]`
	`/NOOLDCPP`

The compiler passes the `/OLDCPP` option to the linker when you compile and link in one step with the `/Gk` option.

Compile and link with `/Gk` when you are linking old object files or libraries, created with version 2.1 of the compiler or earlier, that use templates.

The linker resolves templates for object files created by version 3.0 of the compiler. Since the linker cannot resolve templates in old object files, it normally generates an error message and stops linking when it encounters old object files that require template resolution. The /OLDCPP option informs the linker that the compiler has handled the template resolution already, by calling the muncher from the previous version of the compiler.

/OPTFUNC, /NOOPTFUNC

Syntax:	**Default:**
/OPTF[UNC] /NOOPTF[UNC]	/NOOPTF

Use /OPTFUNC to remove unreachable functions. The linker removes functions that are:

- Not referenced anywhere in the object code
- Rendered unreferenced by the removal of other functions
- Not exported for use in other files

📖 See "EXPORTS" on page 379 for more information on exporting functions.

When the function is removed, any additional functions that were required only by that function are also removed. Removing the functions and code reduces the size of your .EXE or .DLL output file.

By default, the linker does not remove unreachable functions.

If you are compiling and linking in one step, you can use the /Gl compiler option to invoke this optimisation.

Performance Consideration: Optimized linking generally takes longer than regular linking, because of the extra processing that the linker performs. However, if the optimization is effective enough, it can actually speed up the linking process, because there is less information to write to file. Generally, you may want to link without the /OPTFUNC option, until your code is tested and stable.

/OUT

Syntax:	**Default:**
/O[UT]:*name*	Name of first .OBJ file with appropriate extension

Use /OUT to specify a name for the executable output file. To use /OUT, you must be using the default command line syntax (/FREEFORMAT). If you are using the /NOFREE (LINK386-compatible) format, then you cannot use the /OUT option. 📖 See "Using

LINK386 Syntax” on page 325 for information on naming the output file when /NOFREE is specified.

If you do not provide an extension with *name*, then the linker provides an extension based on the type of file you are producing:

File produced	Default extension
Executable program	.exe
Dynamic link library	.dll
Physical device driver	.sys
Virtual device driver	.vdd

If you do not use the /OUT option, then the linker uses the file name of the first object file you specified, with the appropriate extension.

/PACKCODE, /NOPACKCODE

packing code

Syntax:	Default:
/PACKC[ODE][:*number*] /NOP[ACKCODE]	/PACKCODE:0xFffffFfff

Use /PACKCODE to produce slightly faster and more compact code. The linker groups neighboring code segments that have similar attributes, and assigns them to the same load segment. The linker adjusts offsets to each routine upward as required.

Specify *number* to set the maximum size for a load segment. The linker will start new load segments as necessary to avoid exceeding the maximum.

For 16-bit segments, *number* is ignored, and 65500 is used instead.

By default, the linker sets a maximum of 0xFffffFfff.

Use /NOPACKCODE to turn off code segment packing.

Note: If you are compiling old object files that contain **#pragma alloc_text** directives, or were compiled with the /Nt option, then use /NOPACKCODE for debugging. This restriction does not apply to object files created with VisualAge C++ v3.0 and later.

Use the /OPTFUNC option to reduce the size of your output files even further.

/PACKDATA, /NOPACKDATA

Syntax:	Default:

`/PACKD[ATA][:`*number*`]`
`/NOPACKD[ATA]`
	`/NOPACKDATA`

Use `/PACKDATA` to produce more compact files by grouping neighboring data segments that have similar attributes, and assigning them to the same load segment.

Specify *number* to set the maximum size for a load segment. The linker will start new load segments as necessary to avoid exceeding the maximum. By default, the linker sets a maximum of `0xFffffFfff`.

By default, the linker does not pack data segments.

/PDD

Syntax:	Default:
`/PDD`	`/EXEC`

Use `/PDD` to identify the output file as a physical device driver (.SYS file). You can also identify the output file as a .SYS file with the `PHYSICAL DEVICE` statement in a module definition file. For more information on generating a device driver, see 🕮 "Producing a Device Driver" on page 341 .

If you specify `/PDD` with any of `/EXEC`, `/DLL`, or `/VDD`, then only the last specified of the options takes effect.

If you do not specify `/PDD`, or any of the other options above, then by default the linker produces an .EXE file (`/EXEC`).

/PMTYPE

Syntax:	Default:
`/PM[TYPE]:`*type*	None

Use `/PMTYPE` to specify the type of .EXE file that the linker generates. Do not use this option when generating dynamic link libraries (DLLs) or device drivers. The option is equivalent to the NAME module statement, but uses different type names.

/SEC Option

Figure 73. /PMTYPE Parameters

Type	Description	Equivalent NAME Statement Parameter
PM	Presentation Manager application. The application uses the API provided by the Presentation Manager, and must run in the Presentation Manager environment.	WINDOWAPI
VIO	Application compatible with Presentation Manager. The application can run inside the Presentation Manager, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications.	WINDOWCOMPAT
NOVIO	Application that is not compatible with the Presentation Manager and must run in a separate screen group from the Presentation Manager.	NOTWINDOWCOMPAT

/SECTION

Syntax:	Default:
/SEC[TION]:*name*,*attributes*	Depends on segment type

Use /SECTION to specify memory-protection attributes for the *name* segment. You can specify the following attributes:

Letter	Sets Attribute
E	EXECUTE
R	READ
S	SHARED
W	WRITE

For example,

```
/SEC:dseg1,RS
```

sets the READ and SHARED attributes, but not the EXECUTE, or WRITE attributes, for the segment dseg1 in an .EXE file.

Defaults

Segments are assigned attributes by default, as follows:

Segment	Default Attributes
Code segments	EXECUTE, READ (ER) Correspond to the `SEGMENTS` attribute `EXECUTEREAD`.
Data segments (in .EXE file)	READ, WRITE (RW) Correspond to the `SEGMENTS` attribute `READWRITE`.
Data segments (in .DLL file)	READ, WRITE, SHARED (RWS) Correspond to the `SEGMENTS` attributes `READWRITE` and `SHARED`.
CONST32_RO segment	READ, SHARED (RS) Correspond to the `SEGMENTS` attributes `READONLY` and `SHARED`.

You can also set these attributes, and other attributes, to segments using statements in a module definition file:

`CODE`	Sets default attributes for CODE segments
`DATA`	Sets default attributes for DATA segments
`SEGMENTS`	Sets attributes for specific segments

Assignments given in a module definition file override any assignments made with `/SECTION`. See Chapter 21, "Creating Module Definition Files" on page 369 and "Summary of Module Statements" on page 372 for more information on module definition files.

/SEGMENTS

Syntax:	**Default:**
`/SE[GMENTS]:`*`number`*	`/SE:256`

Use `/SEGMENTS` to set the number of logical segments a program can have. You can set *`number`* to any value in the range 1 to 3072. See "Specifying Numeric Arguments" on page 349.

For each logical segment, the linker must allocate space to keep track of segment information. By using a relatively low segment limit as a default (256), the linker is able to link faster and allocate less storage space.

/ST Option

When you set the segment limit higher than 256, the linker allocates more space for segment information. This results in slower linking, but allows you to link programs with a large number of segments.

For programs with fewer than 256 segments, you can improve link time and reduce linker storage requirements by setting `number` to the actual number of segments in the program.

/STACK

Syntax:	**Default:**
`/ST[ACK]:`*`size`*	None

Use `/STACK` to set the stack size (in bytes) of your program. The size must be an even number from `0` to `0xFfffFffe`. If you specify an odd number, it is rounded up to the next even number.

You cannot specify a stack size in which the second most significant byte is either 02 or 04 (in hex), because of a restriction in OS/2 2.0. The linker issues a warning, and adds 64k to the specified stack size to avoid this restriction.

For example, if you specify `/STACK:0x00020000` the linker adds 64k, which results in `/STACK:0x00030000`

Similarly, if you specify `/STACK:0x11041111` the linker adds 64k, which results in `/STACK:0x11051111`

If your program generates a stack-overflow message, use `/STACK` to increase the size of the stack.

If your program uses very little stack space, you can save space by decreasing the stack size.

See “Controlling Stack Allocation and Stack Probes” on page 247 for more information.

Note: Once the executable is produced, you can still change its stack size, using the EXEHDR utility in the toolkit. See the EXEHDR section for more information.

`/STACK` is equivalent to the STACKSIZE statement in a module definition (.DEF) file. If you specify both the statement and the option, the statement value overrides the option value.

/VDD

Syntax:	**Default:**
`/VDD`	`/EXEC`

Use `/VDD` to identify the output file as a virtual device driver (.VDD file). You can also identify the output file as a .VDD file with the `VIRTUAL DEVICE` statement in a module definition file. For more information on generating a DLL, 📖 see “Producing a Device Driver” on page 341.

If you specify `/VDD` with any of `/EXEC`, `/DLL`, or `/PDD`, then only the last specified of the options takes effect.

If you do not specify `/VDD`, or any of the other options above, then by default the linker produces an .EXE file (`/EXEC`).

/VDD Option

21 Creating Module Definition Files

A module definition file contains one or more module statements. These statements:

- Define various attributes of your executable output file
- Define attributes of code and data segments in the file
- Identify data and functions that are imported into or exported from your file

Use module definition files when:

- You are creating a DLL, and did not define exports in your source files (using **`#pragma export`**, or the **`_Export`** keyword). You can use the `EXPORTS` module statement to define exports, instead of defining exports in the source files.

- You are linking with a DLL, and are not using an import library to do so. You can use the `IMPORTS` module statement to define imports, instead of linking to an import library to resolve references to a DLL.

- You are creating a device driver.

- You need to define attributes of the executable output file more precisely than you can with options alone (for example, you want to define library initialization and termination behavior, with the `LIBRARY` statement).

- You need to define segment attributes more precisely than you can with options alone (for example, you are creating a device driver, and need to give some segments I/O access with the `SEGMENTS` statement).

When creating a module definition file, follow these rules:

- Use a NAME, LIBRARY, VIRTUAL DEVICE, or PHYSICAL DEVICE statement to define the type of executable output you want. You can only use one of these statements, and it must precede all other statements in the module definition file.

- Begin comment lines with a semicolon (;). The linker ignores any line in the file that begins with a semicolon.

- Enter all module definition keywords (for example, NAME, LIBRARY, and IOPL) in uppercase letters.

- Do not use module definition keywords, or reserved words, as a text parameter to a statement (for example, you cannot use the LIBRARY statement to name a library `SHARED`, because SHARED is a keyword). See “Reserved Words” on page 370 for a list of keywords and reserved words.

See “Linker Module Statements” on page 373 for detailed descriptions of all statements.

Example

```
;This is a module definition (.DEF) file for a
;dynamic link library (DLL).

LIBRARY
;Identifies the output as a DLL

DESCRIPTION
 'Sample DLL'
;Embeds a description in the DLL

CODE      EXECUTEONLY
;All CODE segments by default can be executed but not read

SEGMENTS
   dseg1 CLASS 'DATA' READONLY
;sets data segment dseg1 to be read only

STACKSIZE 1024
;Sets stack size to 1024

EXPORTS
;Makes data and functions defined inside the DLL available
;to other runtime modules
 Init    @1
 Begin   @2
 Finish  @3
 Load    @4
 Print   @5
;The functions Init, Begin, Finish, Load, and Print can be called either
;by their entry name or by their ordinal positions (1, 2, 3, 4, or 5)
```

Reserved Words

The following words cannot be used as text parameters to a module statement. For example, you cannot use these words as the names of functions defined with the EXPORTS statement, or to name a stub file with the STUB statement.

The words are either module definition keywords, or reserved by the linker.

Reserved Words

Note: Although module definition keywords should always be entered in uppercase letters, and only the uppercase forms are shown below, the mixed- and lower-case forms of these words are also reserved. For example, `CONTIGUOUS`, `ContiGuous`, and `contiguous` are all reserved.

ALIAS
BASE
CODE
CONFORMING
CONTIGUOUS
DATA
DESCRIPTION
DEV386
DISCARDABLE
DOS4
DYNAMIC
EXECUTEONLY
EXECUTEREAD
EXETYPE
EXPANDDOWN
EXPORTS
FIXED
HEAPSIZE
HUGE
IOPL
IMPORTS
IMPURE
INCLUDE
INITGLOBAL
INITINSTANCE
INVALID
LIBRARY
LOADONCALL
LONGNAMES
MAXVAL
MIXED1632
MOVABLE
MOVEABLE
MULTIPLES
NAME
NEWFILES
NODATA
NOEXPANDDOWN
NOIOPL
NONAME
NONCONFORMING
NONDISCARDABLE
NONE
NONPERMANENT
NONSHARED
NOTWINDOWCOMPAT
OBJECTS
OLD
ORDER
OS2
PERMANENT
PHYSICAL DEVICE
PRELOAD
PRIVATE
PROTECT
PURE
READONLY
READWRITE
REALMODE
RESIDENT
RESIDENTNAME
ROBASE
SEGMENTS
SHARED
SINGLE
STACKSIZE
STUB
SWAPPABLE
SYSBASE
TERMGLOBAL
TERMINSTANCE
UNKNOWN
VIRTUAL DEVICE
WINDOWAPI
WINDOWCOMPAT
WINDOWS

Summary of Module Statements

Figure 74 (Page 1 of 2). Linker Module Statements Summary. Default parameters are underlined. The defaults for NONE|SINGLE|MULTIPLE, SHARED|NONSHARED, INITGLOBAL|INITINSTANCE, and TERMGLOBAL|TERMINSTANCE are described in the detailed description of the option.

Statement	Description	Parameters	Page
BASE=*address*	Set preferred loading address.	Loading address	373
CODE *attributes*	Give default attributes for code segments.	CONFORMING|NONCONFORMING EXECUTEONLY|EXECUTEREAD IOPL|NOIOPL PRELOAD|LOADONCALL	374
DATA *attributes*	Give default attributes for data segments. See detailed description for defaults of NONE|SINGLE|MULTIPLE, SHARED|NONSHARED.	IOPL|NOIOPL NONE|SINGLE|MULTIPLES PRELOAD|LOADONCALL READONLY|READWRITE SHARED|NONSHARED	376
DESCRIPTION '*text*'	Describe the executable.	Descriptive text	378
EXETYPE *opsystem*	Identify operating system.	OS2|WINDOWS|UNKNOWN	378
EXPORTS *e*[=*i*] [@*o*[*keywrd*]] [*p*]	Define exported functions and data.	Entry name Internal name Ordinal position RESIDENTNAME|NONAME Parameter size	379
HEAPSIZE *size*	Specify local heap size.	*bytes*|MAXVAL	381
IMPORTS [*intname*=]*dllname*.*entry*	Define imported functions.	Internal name Name of exporting module Entry name or ordinal value	381
LIBRARY [*lib*] [*init*] [*term*]	Identify output as dynamic link library (DLL). See detailed description for defaults of parameters.	Library name INITGLOBAL|INITINSTANCE TERMGLOBAL|TERMINSTANCE	383
NAME [*appname*] [*apptype*]	Identify output as executable (EXE).	Application name WINDOWAPI|WINDOWCOMPAT|NOTWINDOWCOMPAT	384
OLD '[*dir*]*name*'	Preserve ordinal values from old DLL.	Name of old DLL	386
PHYSICAL DEVICE [*drivername*]	Identify output as physical device driver.	Name of driver	386

Figure 74 (Page 2 of 2). Linker Module Statements Summary. Default parameters are underlined. The defaults for NONE|SINGLE|MULTIPLE, SHARED|NONSHARED, INITGLOBAL|INITINSTANCE, and TERMGLOBAL|TERMINSTANCE are described in the detailed description of the option.

Statement	**Description**	**Parameters**	**Page**
SEGMENTS [']*s*['] [CLASS '*c*'] [*a*]	Give attributes for specific segments. See detailed description for default of SHARED|NONSHARED.	Segment name Class of the segment ALIAS CONFORMING|NONCOMFORMING EXECUTEONLY|EXECUTEREAD IOPL|NOIOPL MIXED1632 PRELOAD|LOADONCALL READONLY|READWRITE SHARED|NONSHARED	387
STACKSIZE `size`	Specify local stack size.	Stack size (in bytes)	390
STUB '`file name`'	Add DOS executable file to module.	File name to add	391
VIRTUAL DEVICE [`drivername`]	Identify output as virtual device driver.	Name of driver	391

Linker Module Statements

BASE

Syntax:
`BASE=`*`address`*

Parameters:
Loading address

Use the BASE statement to specify the preferred load address for the first load segment of the module. The number you give for the option is rounded up to the nearest multiple of 64K. The second load segment is then loaded at the next available multiple of 64K, and so on.

If the module's load segments cannot be loaded beginning at this preferred address, then the preferred address is ignored and the load segments are loaded according to the internal relocation records retained in the file data.

For .EXE files, accept the default base address of 64K (BASE=0x00010000). Any other address will result in a warning, and 64K will be used anyway.

This statement has the same effect as the `/BASE` linker option. If you specify both the statement and the option, the statement value overrides the option value.

CODE

Syntax:
`CODE` *`attributes`*

Parameters:
CONFORMING|NONCONFORMING
EXECUTEONLY|EXECUTEREAD
IOPL|NOIOPL
PRELOAD|LOADONCALL

Use the CODE statement to define default attributes for code segments within the executable you are creating. You can override the default attributes with the SEGMENTS statement (described on page 387), or the `/SECTION` option (described on page 364), which define attributes for specific segments.

Attribute Rules

- You can only specify one attribute from each pair. If you specify neither attribute, ILINK uses the default. See the description of the parameter for its default.
- Attributes can appear in any order.

CONFORMING|NONCONFORMING

Use these attributes to specify whether a code segment is a 286-conforming segment. These attributes are relevant for device drivers, or system-level code. They apply to code segments only.

- CONFORMING specifies that the segment is conforming, and uses a range of instructions that can be executed by a 286 (16-bit) processor. A CONFORMING segment can be called from either Ring 2 or Ring 3, and executes at the privilege level of the caller.
- NONCONFORMING specifies that the segment is nonconforming, and uses instructions that require a 386 processor or higher. The segment is not guaranteed to be executable by a 286 processor.

The default is NONCONFORMING.

EXECUTEONLY|EXECUTEREAD

Use these attributes to specify whether a code segment can be read as well as executed. These attributes apply to code segments only.

- EXECUTEONLY specifies that the segment can only be executed.
- EXECUTEREAD specifies that the segment can be both executed and read.

The default is EXECUTEREAD.

IOPL|NOIOPL

Use these attributes to determine whether a segment has I/O privilege, that is, whether it can access the hardware directly.

- IOPL specifies that the segment has I/O privilege.
- NOIOPL specifies that the segment does not have I/O privilege.

The default is NOIOPL.

Note: 32-bit segments must be NOIOPL. You cannot specify a 32-bit segment as IOPL.

PRELOAD|LOADONCALL

Use these attributes to specify when the segment is loaded.

Note: These attributes are ignored on OS/2 version 2.0 and later.

- PRELOAD specifies that the segment will be loaded automatically when the program starts.
- LOADONCALL specifies that the segment will not be loaded until accessed.

The default is LOADONCALL.

Example

Given the following line in a .DEF file,

```
CODE LOADONCALL IOPL
```

CODE segments are not loaded until accessed (LOADONCALL), and have I/O hardware prvilege (IOPL). In addition, the linker assumes the following defaults:

- EXECUTEREAD (can be read as well as executed)
- NONCONFORMING (not guaranteed to run on a machine based on the 80286 microprocessor)

These attributes apply to all CODE segments, except when you override them with the SEGMENTS statement

DATA

Syntax:
`DATA` *`attributes`*

Parameters:
IOPL|NOIOPL
NONE|SINGLE|MULTIPLE
PRELOAD|LOADONCALL
READONLY|READWRITE
SHARED|NONSHARED

Use the DATA statement to define default attributes for data segments within the executable you are creating. You can override the default attributes with the SEGMENTS statement (described on 📖page 387), or the `/SECTION` option (described on 📖page 364), which define attributes for specific segments.

Attribute Rules

- You can only specify one attribute from each group. If you specify none of the attributes in a group, the linker uses the default. See the description of the parameter for its default.
- Attributes can appear in any order.

IOPL|NOIOPL

Use these attributes to determine whether a segment has I/O privilege, that is, whether it can access the hardware directly.

- IOPL specifies that the segment has I/O privilege.
- NOIOPL specifies that the segment does not have I/O privilege.

The default is NOIOPL.

Note: 32-bit segments must be NOIOPL. You cannot specify a 32-bit segment as IOPL.

NONE|SINGLE|MULTIPLE

Use these attributes to specify how the automatic data segment can be shared. The automatic data segment is the physical segment represented by the group name DGROUP. This segment group makes up the physical segment that contains the local stack and heap of the application.

- NONE specifies that no automatic data segment is created.
- SINGLE specifies that a single automatic data segment is shared by all instances of the module. In this case, the module is said to have solo data. SINGLE is the default for .DLL files.

- MULTIPLE specifies that the automatic data segment is copied for each instance of the module. In this case, the module is said to have instance data. MULTIPLE is the default for .EXE files.

PRELOAD|LOADONCALL

Use these attributes to specify when the segment is loaded.

Note: These attributes are ignored on OS/2 version 2.0 and later.

- PRELOAD specifies that the segment will be loaded automatically when the program starts.
- LOADONCALL specifies that the segment will not be loaded until accessed.

The default is LOADONCALL.

READONLY|READWRITE

Use these attributes to set the access rights to a data segment. These attributes apply to data segments only.

- READONLY specifies that the segment can only be read.
- READWRITE specifies that the segment can be both read and written to.

The default is READWRITE.

SHARED|NONSHARED

Use these attributes to specify whether the segment can be shared by other processes. These attributes apply to data segments only.

- SHARED specifies that one copy of the segment is loaded and shared among all processes accessing the module. SHARED is the default for dynamic link library (.DLL) files.
- NONSHARED specifies that the segment cannot be shared, and must be loaded separately for each process. NONSHARED is the default for executable program (.EXE) files.

Example

Given the following line in a .DEF file,

```
DATA LOADONCALL NONSHARED
```

DATA segments are not loaded until they are accessed (LOADONCALL), and cannot be shared between multiple copies of the program (NONSHARED). In addition, the linker assumes the following defaults:

- READWRITE (DATA segments can be read and written to)
- MULTIPLE (the automatic data segment is copied for each instance of the module)
- NOIO (DATA segments have no I/O hardware privilege)

These attributes apply to all DATA segments, except when you override them with the SEGMENTS statement

DESCRIPTION

Syntax:	**Parameters:**
`DESCRIPTION 'text'`	Descriptive text

Use the DESCRIPTION statement to insert the specified text into the .EXE or .DLL file you are creating. The DESCRIPTION statement is useful for embedding source control or copyright information into your program or DLL.

The inserted text must be a one-line string enclosed in single quotation marks.

Example

Given the following line in a .DEF file,

```
DESCRIPTION 'Template Program'
```

the linker inserts the text `Template Program` into the .EXE or .DLL file.

EXETYPE

Syntax:
`EXETYPE` *opsystem*

Parameters:
OS2|WINDOWS|UNKNOWN

Use the EXETYPE statement to specify the operating system the .EXE or .DLL will run under. This statement is optional, and can provide an additional degree of protection against the program being run in an incorrect operating system.

For *opsystem*, specify one of the following:

OS2	OS/2 .EXE and .DLL files (this is the default)
WINDOWS	Microsoft Windows applications
UNKNOWN	Other applications

When you use EXETYPE, the linker sets bits in the header that identify operating-system type. Operating-system loaders can then check these bits before running the application.

EXPORTS

Syntax:

```
EXPORTS
  enm [=inm] [@ord[keywrd]] [pwrds]
```

Parameters:
Entry name
Internal name for function
Ordinal position of function
RESIDENTNAME|NONAME
Size of function's parameters

Use the EXPORT statement when you are creating a dynamic link library (DLL) to define the names and attributes of data and functions exported from the DLL, and of functions that run with I/O hardware privilege.

You can also specify exports in your source code, using the **`_Export`** keyword, or the **`#pragma export`** directive.

Note: Exported data and functions are those available to other .EXE or .DLL files. Data and functions that are **not** exported can only be accessed within your DLL, and cannot be accessed by other .EXE or .DLL files.

Give export definitions for functions and data in your DLL that you want to make available to other .EXE or .DLL files.

The EXPORTS keyword marks the beginning of the export definitions. Enter each definition on a separate line. You can provide the following information for each export:

enm The entry name of the data construct or function , which is the name other files use to access it. Always provide an entry name for each export.

inm The internal name of the data construct or function, which is its actual name as it appears **within** the DLL. If you do not specify an internal name, the linker assumes it is the same as *enm*.

ord The data construct or function's ordinal position in the module definition table. If you provide the ordinal position, the data construct or function can be referenced either by its entry name or by the ordinal. It is faster to access by ordinal positions, and may save space.

EXPORTS

keywrd You can specify one of two values:

RESIDENTNAME Indicates that you want the data construct or function's name kept resident in memory at all times. You only need to specify RESIDENTNAME if you gave an ordinal position in *ord*. When a data construct or function does not have an ordinal position defined, OS/2 keeps the name of the exported data construct or function resident in memory by default.

NONAME Indicates that you want the data construct or function to always be referenced by its ordinal number. If you specify NONAME, the data construct or function cannot be referenced by name: it can only be referenced by ordinal number.

You cannot specify both values.

pwrds The total size of the function's parameters, as measured in words (bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults *pwrds* to determine how many words to copy from the caller's stack to the stack of the I/O-privileged function.

Example

The following example defines three exported functions:

- SampleRead
- StringIn
- CharTest

```
EXPORTS
  SampleRead = read2bin @8
  StringIn = str1       @4 RESIDENTNAME
  CharTest  6
```

The first two functions can be accessed either by their exported names or by an ordinal number. Note that in the module's own source code, these functions are actually defined as read2bin and str1, respectively. The last function runs with I/O privilege, and so has *pwrds* (the total size of the parameters) defined for it: six words.

HEAPSIZE

Syntax:	**Parameters:**
`HEAPSIZE` *`size`*	*`bytes`*|MAXVAL

Use the HEAPSIZE statement to define the size of the application's local heap in bytes. This value affects the size of the automatic data segment (DGROUP), which contains the local stack and heap of the application.

You can enter any positive integer for the heap size. 📖 See “Specifying Numeric Arguments” on page 349.

Instead of entering the number of bytes, you can enter the keyword MAXVAL. This increases the size of DGROUP to 64K, if it is smaller than 64K. MAXVAL is useful in bound applications, when you want to force a 64K requirement for DGROUP. MAXVAL is not generally useful for 32-bit programs.

Example

Given the following line in a .DEF file,

```
HEAPSIZE 4000
```

the linker sets the local heap to 4000 bytes.

IMPORTS

Syntax:	**Parameters:**
`IMPORTS`	Internal name for function
`[`*`intname`*`= ]`*`dllname`*`.`*`entry`*	Name of exporting module
	Function's entry name

Use the IMPORT statement to define the names of the functions imported from a DLL for your .EXE or .DLL file to use.

If your file references functions that are defined in a DLL, you must import the functions from the DLL before your file can use them. You can qualify imported functions with the **`_Import`** keyword, but you must still define them in the module definition file, to give the name of the DLL the functions are defined in. 📖 See “Linking to a DLL Using a .DEF File” on page 346 for more information.

IMPORTS

Note: Instead of using the IMPORTS statement, you can use an import library (created by the IMPLIB utility) to resolve external references to DLL symbols. See “Linking to a DLL Using an Import Library” on page 346 for more information.

The IMPORTS keyword marks the beginning of the import definitions. Enter each definition on a separate line. Each import definition corresponds to a particular function. The only limit on the number of import definitions is that the total amount of space required for their names must be less than 64K.

You can provide the following information for each import definition:

intname
The internal name of the function, that is used within your module to call the function. This is the name used by the importing module, although the function can have a different name in the module where it is defined (the exporting module). If *entry* contains a name, then by default, *intname* uses the same name.

dllname
The name of the DLL that contains the function. You must provide this information for each import you define.

entry
The function to be imported, identified either by entry name or by ordinal value.

You can only use an ordinal value if one is defined for the function in its export definition (see “EXPORTS” on page 379). If you use the ordinal value, then you must also define an *intname* for your module to use.

The entry name for the function is always defined in its export definition.

By default, the exporting module and importing module both call the function by its entry name. However, each module can provide its own internal name for the function. It is possible for the function to have up to three distinct names:

- The exporting module's internal name for the function (associated with the entry name by the export statement)
- The function's entry name (and an optional ordinal value)
- The importing module's internal name for the function (associated with either the entry name or the ordinal value by the import statement)

Example

The following example defines three functions to be imported:

- SampleRead

- SampleWrite
- A function that has been assigned an ordinal value of 1

```
IMPORTS
  Sample.SampleRead
  SampleA.SampleWrite
  ReadChar = Read.1
```

The functions are found in the modules Sample, SampleA, and Read, respectively. The SampleRead and SampleWrite functions are called by their entry names. The function from the Read module is called by the internal name ReadChar, which maps to the ordinal value 1. Its actual entry name is not shown, because it is called by ordinal value instead of by its entry name.

LIBRARY

Syntax:	**Parameters:**
`LIBRARY [`*`libname`*`] [`*`init`*`] [`*`term`*`]`	Library name INITGLOBAL|INITINSTANCE TERMGLOBAL|TERMINSTANCE

Use the LIBRARY statement to identify the output file as a dynamic link library (DLL), and optionally define the name, library module initialization, and library module termination.

You can also identify the output file as a DLL with the `/DLL`option.

The following table shows defaults for the fields, depending on whether the DLL has 16-bit entry points, or 32-bit entry points:

Figure 75. LIBRARY Default Values

Field	Default for DLLs with 16-bit Entry Points	Default for DLLs with 32-bit Entry Points
libname	Name of output file with .DLL extension removed	Name of output file with .DLL extension removed
init	INITGLOBAL	Matches *term*, if termination given. Otherwise INITGLOBAL.
term	None (applies only to DLLs with 32-bit entry points)	Matches *init*, if initialization given. Otherwise TERMGLOBAL.

If you use the LIBRARY statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the NAME, PHYSICAL DEVICE, or VIRTUAL DEVICE statements.

If you provide a name in *libname*, it becomes the name of the library as it is known by OS/2. The name can be any valid file name.

Use the *init* field to define the type of library initialization you want:

INITGLOBAL The library initialization routine is called only when the library module is initially loaded into memory.

INITINSTANCE The library initialization routine is called each time a new process gains access to the library.

If you are generating a DLL with 32-bit entry points, you can set the type of library termination you want:

TERMGLOBAL The library termination routine is called only when the library module is unloaded from memory.

TERMINSTANCE The library termination routine is called each time a process gives up access to the library.

Example

The following example assigns the name `calendar` to the dynamic link library (DLL), and specifies that library initialization be performed each time a new process gains access. If `calendar` has 32-bit entry points, the linker will assume TERMINSTANCE.

```
LIBRARY calendar INITINSTANCE
```

NAME

Syntax:	**Parameters:**
`NAME [`*appname*`] [`*apptype*`]`	Application name WINDOWAPI|WINDOWCOMPAT |NOTWINDOWCOMPAT

Use the NAME statement to identify the output file as an executable program (.EXE file), and optionally define the name and type of the .EXE file.

You can also identify the output file as an .EXE file with the `/EXEC` option.

NAME

If you use the NAME statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the LIBRARY, PHYSICAL DEVICE, or VIRTUAL DEVICE statements.

If you specify *appname*, it becomes the name of the .EXE as it is known by OS/2. The name can be any valid file name. If you do not provide a name, the name of the executable program is the same as the name of the output file, with the .EXE extension removed.

The NAME statement also allows you to define the type of the program:

Figure 76. NAME Statement Parameters

Type	**Description**	**/PMTYPE option equivalent**
WINDOWAPI	Presentation Manager application. The application uses the API provided by the Presentation Manager, and must run in the Presentation Manager environment.	PM
WINDOWCOMPAT	Application compatible with Presentation Manager. The application can run inside the Presentation Manager, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications.	VIO
NOTWINDOWCOMPAT	Application that is not compatible with the Presentation Manager and must run in a separate screen group from the Presentation Manager.	NOVIO

You can also use the `/PMTYPE` option to set the type. If conflicting types are defined by the option and in the NAME statement, the type defined by the NAME statement overrides the option value.

Example

The following example assigns the name `calendar` to the executable program, and specifies it as compatible with PM.

```
NAME calendar WINDOWCOMPAT
```

OLD

Syntax:	**Parameters:**
`OLD '[dir]name'`	Name of DLL

Use the OLD statement when you create a dynamic link library (DLL) to preserve compatibility with an older version of the DLL. When you provide the *name* of the old DLL, specify the directory it is in as well, unless it is in the current working directory.

The linker compares exported data constructs or functions in the old DLL with exported data constructs or functions in the current DLL. If the old data construct or function has an ordinal value assigned to it, the linker assigns the ordinal value to the equivalent data construct or function in the new DLL.

If another run-time module called functions or referenced data from the old DLL by ordinal value, it can continue calling functions and referencing data from the new DLL using the same ordinal values.

The linker will only assign the old ordinal value to a data construct or function when:

- The data construct or function name in the old DLL matches the data construct or function name in the new DLL exactly
- The old data construct or function has an ordinal value assigned to it
- The new data construct or function does not already have an ordinal value assigned to it.

See “EXPORTS” on page 379 for more information on assigning ordinal values.

PHYSICAL DEVICE

Syntax:	**Parameters:**
`PHYSICAL DEVICE [drivername]`	Name of driver

Use the PHYSICAL DEVICE statement to identify the output file as a physical device driver (.SYS), and optionally provide a name for the driver.

You can also identify the output file as a .SYS file with the /PDD option.

If you use the PHYSICAL DEVICE statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the LIBRARY, NAME, or VIRTUAL DEVICE statements.

If you provide a name in *drivername*, it becomes the name of the driver as it is known by OS/2. The name can be any valid file name. If you do not provide a name, the name of the device driver is the same as the name of the output file, with the .SYS extension removed.

SEGMENTS

Syntax:

```
SEGMENTS
  [']segname['] [CLASS 'class'] [attribs]
```

Parameters:
Segment name
Class of the segment
ALIAS
CONFORMING|NONCOMFORMING
EXECUTEONLY|EXECUTEREAD
IOPL|NOIOPL
MIXED1632
PRELOAD|LOADONCALL
READONLY|READWRITE
SHARED|NONSHARED

Use the SEGMENTS statement to define the attributes of one or more segments in the .EXE or .DLL file on a segment-by-segment basis. The attributes you specify in this statement override any defaults set in the CODE and DATA statements.

You can also set some segment attributes with the `/SECTION` option. See "/SECTION" on page 364 for more information.

The SEGMENTS keyword marks the beginning of the segment definitions. Enter each definition on a separate line. You can enter up to 256 separate definitions.

Each segment definition begins with its name (*segname*). If *segname* is the same as a module statement or keyword (such as DATA or IOPL), then you must enclose *segname* in single quotation marks (').

You can specify the class of the segment with the CLASS keyword, followed by the class of the segment, enclosed in single quotation marks ('). If you do not specify a class for the segment, the linker assumes the segment is of class CODE.

After the name and class, you can set attributes for the segment. If you do not specify attributes for a segment, the linker assumes a default set of attributes, as underlined in the parameters list above, or as set by the CODE and DATA statements. The default for SHARED|NONSHARED varies, depending on the type of output file.

SEGMENTS

Attribute Rules

- You can only specify one attribute from each pair. If you specify neither attribute, ILINK uses the default. See the description of the parameter for its default.
- Attributes can appear in any order.

ALIAS

Use ALIAS to allow the segment to be addressed using both the 16-bit segmented method (_far16), and the 32-bit linear method. When you specify ALIAS, the loader prepares an additional segment selector for the segment that allows for 16-bit addressing of the segment. The segment can then be called using 16-bit far calls and 32-bit near calls.

By default, segments are addressable only by the 32-bit linear method.

CONFORMING|NONCONFORMING

Use these attributes to specify whether a code segment is a 286-conforming segment. These attributes are relevant for device drivers, or system-level code. They apply to code segments only.

- CONFORMING specifies that the segment is conforming, and uses a range of instructions that can be executed by a 286 (16-bit) processor. A CONFORMING segment can be called from either Ring 2 or Ring 3, and executes at the privilege level of the caller.
- NONCONFORMING specifies that the segment is nonconforming, and uses instructions that require a 386 processor or higher. The segment is not guaranteed to be executable by a 286 processor.

The default is NONCONFORMING.

EXECUTEONLY|EXECUTEREAD

Use these attributes to specify whether a code segment can be read as well as executed. These attributes apply to code segments only.

- EXECUTEONLY specifies that the segment can only be executed.
- EXECUTEREAD specifies that the segment can be both executed and read.

The default is EXECUTEREAD.

IOPL|NOIOPL

Use these attributes to determine whether a segment has I/O privilege, that is, whether it can access the hardware directly.

- IOPL specifies that the segment has I/O privilege.
- NOIOPL specifies that the segment does not have I/O privilege.

The default is NOIOPL.

Note: 32-bit segments must be NOIOPL. You cannot specify a 32-bit segment as IOPL.

MIXED1632

Use MIXED1632 to specify that the segment is part of a group that allows a mix of 16-bit and 32-bit code. If you create a group that allows such mixing, you must declare each segment in the group as MIXED1632.

PRELOAD|LOADONCALL

Use these attributes to specify when the segment is loaded.

Note: These attributes are ignored on OS/2 version 2.0 and later.

- PRELOAD specifies that the segment will be loaded automatically when the program starts.
- LOADONCALL specifies that the segment will not be loaded until accessed.

The default is LOADONCALL.

READONLY|READWRITE

Use these attributes to set the access rights to a data segment. These attributes apply to data segments only.

- READONLY specifies that the segment can only be read.
- READWRITE specifies that the segment can be both read and written to.

The default is READWRITE.

SHARED|NONSHARED

Use these attributes to specify whether the segment can be shared by other processes. These attributes apply to data segments only.

- SHARED specifies that one copy of the segment is loaded and shared among all processes accessing the module. SHARED is the default for dynamic link library (.DLL) files.
- NONSHARED specifies that the segment cannot be shared, and must be loaded separately for each process. NONSHARED is the default for executable program (.EXE) files.

Example

The following example specifies segments named cseg1, cseg2, and dseg. The first segment is assigned class `mycode`, and the third segment is assigned class `data`. The second segment is assigned class `CODE` by default. Each segment is given different attributes.

```
SEGMENTS
  cseg1 CLASS 'mycode' IOPL
  cseg2 EXECUTEONLY PRELOAD CONFORMING
  dseg CLASS 'data' LOADONCALL READONLY
```

STACKSIZE

Syntax:	**Parameters:**
`STACKSIZE` *size*	Stack size in bytes

Use STACKSIZE to set the stack size (in bytes) of your program. The size must be an even number, from 0 to 0xFfffFffe. If you specify an odd number, it is rounded up to the next even number.

You cannot specify a stack size in which the second most significant byte is either 02 or 04 (in hex), because of a restriction in OS/2 2.0. The linker issues a warning, and adds 64k to the stack size to avoid the restriction.

For example, if you specify `STACKSIZE 0x00020000` the linker adds 64k, which results in `STACKSIZE 0x00030000`

Similarly, if you specify `STACKSIZE 0x11041111` the linker adds 64k, which results in `STACKSIZE 0x11051111`

If your program generates a stack-overflow message, use the STACKSIZE statement to increase the size of the stack.

If your program uses the stack very little, you can save some space by decreasing the stack size. See “Controlling Stack Allocation and Stack Probes” on page 247 for more information.

Note: Once the output file is produced, you can still change its stack size, using the EXEHDR utility. See the EXEHDR User's Guide for more information.

The STACKSIZE statement is equivalent to the `/STACK` linker option. If you specify both the statement and the option, the statement value overrides the option value.

Example

The following example allocates 4 K of local-stack space:

```
STACKSIZE 4096
```

STUB

Syntax:	**Parameters:**
`STUB '`*`filename`*`'`	Name of file to add

Use the STUB statement to add a DOS .EXE file to the beginning of your .EXE or .DLL. The stub function is then invoked whenever your .EXE or .DLL file is run under DOS. Typically, the stub displays the message that the program cannot run in DOS mode, and ends the program.

If you do not use the STUB statement, the linker adds its own standard stub for this purpose.

The linker searches for the file name you specify as the stub as follows:

1. In the directory you specify, or in the current directory if you did not give a path
2. In the directories listed in the PATH environment variable

Example

The following example adds the DOS .EXE file STOPIT.EXE to the beginning of the file you are creating. STOPIT.EXE runs whenever your file is run under DOS.

```
STUB 'STOPIT.EXE'
```

VIRTUAL DEVICE

Syntax:	**Parameters:**
`VIRTUAL DEVICE [`*`drivername`*`]`	Name of driver

Use the VIRTUAL DEVICE statement to identify the output file as a virtual device driver (.VDD), and optionally provide a name for the driver.

You can also identify the output file as a .VDD file with the `/VDD` option.

If you use the VIRTUAL DEVICE statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the LIBRARY, NAME, or PHYSICAL DEVICE statements.

VIRTUAL DEVICE

If you provide a name in *drivername*, it becomes the name of the driver as it is known by OS/2. The name can be any valid file name. If you do not provide a name, the name of the device driver is the same as the name of the output file, with the .VDD extension removed.

Part 6. IBM VisualAge C ++ Debugger

This part of the *User's Guide* describes the VisualAge C++ Debugger, which you can use to debug your programs once you have compiled and linked them.

Debugging Your Program

22 Introduction

The IBM VisualAge C++ for OS/2 Debugger (hereafter referred to as the debugger) uses Presentation Manager (PM) window services to help detect and diagnose errors in code developed in IBM 32-bit C/C++.

Use the debugger to debug your code at source level, set breakpoints, and examine message queues. You can also monitor variables, registers, storage, and the call stack.

Understanding the New and Enhanced Features

The following describes the new features that have been added to the debugger since the previous release.

Deferred breakpoints
Allows you to set a breakpoint in a DLL that is not currently loaded. If your application consists of DLLs that are dynamically loaded, use this feature to set breakpoints in the dynamically loaded DLLs that have not been loaded yet. These deferred breakpoints become active once the DLL is loaded.

Child process debugging
Supports debugging of processes started by a parent program.

Exception filtering
Allows you to select the exceptions that you want the debugger to recognize. An exception occurs when your application is unable to interpret specific requests.

Check heap when stopping
Helps to isolate memory management problems by checking for memory overwriting each time your program stops executing.

Hide debugger on Run
Hides the debugger windows while your application is running.

Color support
Allows you to change the color of the various window elements such as executable lines, non-executable lines, and the breakpoint prefix area.

SOM support
Allows you to debug SOM objects created with the compiler using Direct-to-SOM support or created with the SOM compiler. Support includes monitoring SOM classes in the monitor windows.

Scroll to line number
Allows you to scroll to a particular line number in the source code. This feature also provides the ability to set breakpoints.

Autosave window positions and sizes
Saves the window positions and sizes when the windows are moved or re-sized. Alternatively, you may save the window positions and sizes by position to the debugger windows on the desktop and selecting the **Save window positions and sizes** choice.

Integration
Provides quick and easy access to other tools such as an editor or the browser. This feature is available when the debugger is started from within the WorkFrame environment.

Select include files
Allows you to select the include files you want to view. Include files are files that are included in your source file by a compiler directive and are considered program source files.

Windows menu
Displays a list of all the active debugger windows.

Hover help for title bar buttons
Displays the name of the title bar button when you place your mouse pointer on the button. If you drag the mouse pointer across the buttons, the name in the title bar area changes to reflect the button you are on.

The following describes the enhanced features that have been added to the debugger since the previous release.

Call stack window
Provides the option of displaying the remaining stack size, the stack frame size, the return address, the ESP value and the EBP value.

Breakpoint list window
Displays as a window allowing you to continuously monitor the breakpoint list. You can also display the source code where the breakpoint is set.

Storage window
Allows you to monitor expressions in a storage window. For example, if you are monitoring a pointer, as the pointer changes, the storage window changes to show the new location referenced by the pointer.

Change address breakpoint
Allows you to set a change address breakpoint by typing in an expression.

Enable program profiling

Allows you to enable or disable the use of program profiles which restores a program's breakpoints, source windows and monitors to the same state as when last debugged.

Debugging Your Program

23 Before You Begin

This section lists the considerations you need to be aware of and preparatory items you should complete before you run the debugger on your program.

Writing Code that the Debugger Supports

Using C and C++, you can write your program code with stylistic features that are not supported by the debugger. Multiple statements on the same line are difficult to debug. None of the individual statements can be accessed separately when you set breakpoints[2] or when you use step commands.

Compiling and Linking Your Program

Before using the debugger, compile and link your program with the following options:

`/Ti+` Compiles your program to produce an object file that includes line number information and a symbol table, in addition to the source code.

`/O-` Compiles your program with optimization off. This is the default.

`/Oi-` Compiles your program with inlining off. This is the default.

`/DEbug` Links your program to produce an executable file that includes line number information and a symbol table, in addition to the executable code.

Note: When you specify the /Ti+ option with the /DEbug option, icc passes this option to the linker automatically, so you only need to use it if you link separately from the compile.

The following option is only required if you want to use the heap check feature which is 📖 described in “Check heap when stopping” on page 421.

`/Tm+` Compiles your program to allow the heap to be checked.

For more information about compiling and linking your program, refer to *IBM C++ Tools Programming Guide*.

[2] Breakpoint is a defined location or condition in a program that, when it is met, stops the execution of the program.

Setting Environment Variables

There are several environment variables that you can use with the debugger. They affect the search order and number of tab spaces between your source code.

The search path tells the debugger where to find the source file used in the source windows. The debugger searches for the source files in the following order:

1. The path defined by the PMDOVERRIDE environment variable, if specified.
2. The path where the object file was compiled.
3. The path where the executable file is located.
4. The path defined by the PMDPATH environment variable, if specified.
5. The current path.
6. The path defined in the INCLUDE environment variable.

To override the normal search order, use the PMDOVERRIDE environment variable. To set the PMDOVERRIDE environment variable, type the following at the command prompt:

```
Set PMDOVERRIDE=path;path
```

where, *path* is the location of your source files. If the source file is not found in the defined override path, the debugger uses the normal search order.

To set the PMDPATH environment variable, type the following at the command prompt:

```
Set PMDPATH=path;path
```

where, *path* is the location of your source files.

To set the number of spaces between tab stops in your source code, type the following at the command prompt:

```
Set PMDTABGRID=n
```

where, *n* is the increment number of spaces between tab stops. For example, if *n* is 5, tab stops would be 5, 10, 15, and so on.

24

Getting Started

This section describes how to start a debugging session from either the OS/2 command prompt or the WorkFrame environment, explains the WorkFrame integration process, and describes how to debug REXX and WorkPlace Shell objects. It also describes how to end a debugging session.

Starting the Debugger from OS/2

To start the debugger from the OS/2 command prompt, enter the command `icsdebug` and the following parameters, in the order they are listed:

1. Any debugger parameters that you want to use.
2. Name of the program you want to debug.
3. Any input parameters that you want to pass to the program.

For example, type the following:

```
icsdebug /x myprog xyz
```

where */x* represents a debugger parameter, *myprog* represents your program name, and *xyz* represents the program parameter you want to pass to the program.

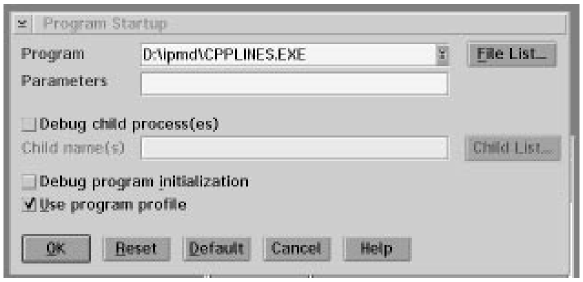
The debugger parameters are:

/p+ Use program profile information.

/p- Do not use any program profile information.

/i Start the debugger in the system initialization routine so that you can debug initialization code.

If you type `icsdebug` and press Enter, the **Program Startup** window displays.



Program Startup

Program D:\ipmd\CPPLINES.EXE

Parameters

☐ Debug child process(es)

Child name(s)

☐ Debug program initialization

☑ Use program profile

Figure 77. Program Startup Window

Getting Started

Use the **Program Startup** window to specify the program you want to debug.

- Type the name of the program you want to debug in the **Program** entry field. You can also select the **File List** push button.

  If you select **File List**, the **Select Program** window displays. From this window, select the program you want and select **OK**. The **Program Startup** window displays again with the program name you selected displayed in the **Program** entry field.

  You can also select the list button to display a drop-down combination box. The drop-down combination box contains a list of up to five previously debugged programs.

- In the **Parameters** field, type any parameters that you want to pass to your program. You must separate multiple parameters with spaces.
- Enable the **Debug child process(es)** check box to debug processes that are started by a parent program. When you enable this check box, the **Child name(s)** entry field becomes active.
- In the **Child name(s)** entry field, type the name of the child process you want to debug. You can also select the **Child List** push button.

  If you select **Child List**, the **Select Child Process** window displays. From this window, select the child process you want to debug. The **Program Startup** window displays again with the program name you selected displayed in the **Child name(s)** entry field.
- Enable the **Debug program initialization** check box to start the debugger in the system initialization routine so that you can debug initialization code.
- Enable the **Use program profile** check box to restore the debugger windows and breakpoints when debugging a program more than once. It is stored separately for each program debugged.
- Select the **OK** push button to start the debugging session.

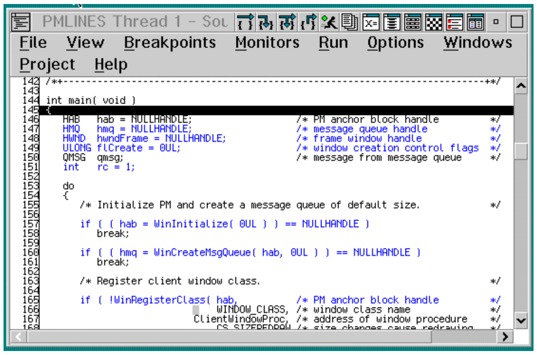
Starting the Debugger from WorkFrame.

Before you start the debugger from the WorkFrame environment, you must create a project for the program you want to debug. Before you can compile and link a target program with debugging information, you must set the debugger options that the WorkFrame environment uses for creating a project. For information on creating a project, setting options, and starting the debugger, refer to *IBM WorkFrame* section.

Understanding Integration

Integration is a new feature that allows you to access other tools from the debugger. To make this feature available, you have to start the debugger from within the WorkFrame environment. For information on starting the debugger from WorkFrame, refer to the *IBM WorkFrame* section.

After you start the debugger, the **Source** window displays with a new menu bar item called **Project** as shown in Figure 78.



```
/**-----------------------------------------------------------------**/

int main( void )
{
   HAB   hab = NULLHANDLE;                    /* PM anchor block handle      */
   HMQ   hmq = NULLHANDLE;                    /* message queue handle        */
   HWND  hwndFrame = NULLHANDLE;              /* frame window handle         */
   ULONG flCreate = 0UL;                      /* window creation control flags */
   QMSG  qmsg;                                /* message from message queue  */
   int   rc = 1;

   do
   {
      /* Initialize PM and create a message queue of default size.         */

      if ( ( hab = WinInitialize( 0UL ) ) == NULLHANDLE )
         break;

      if ( ( hmq = WinCreateMsgQueue( hab, 0UL ) ) == NULLHANDLE )
         break;

      /* Register client window class.                                      */

      if ( !WinRegisterClass( hab,            /* PM anchor block handle      */
                         WINDOW_CLASS,        /* window class name           */
                        ClientWindowProc,     /* address of window procedure */
                         CS_SIZEREDRAW        /* size changes cause redrawing */
```

Figure 78. Integration Source Window

Note: The **Source** window is the only debugger window that contains this feature.

Select choices from the **Project** menu to open other tools such as an editor, browser, or the compiler.

This feature also provides context sensitive help for items in the **Source** window. This help becomes available when you select a string, using the mouse pointer, and press and hold down **Ctrl+H**. If you place the mouse pointer on a space in the **Source** window and double-click, an editor opens to the corresponding source line that you selected.

Debugging REXX and WorkPlace Shell Objects

To debug REXX DLLs, type the following at an OS/2 prompt:

```
icsdebug cmd.exe /K <your rexx.cmd>
```

When ICSDEBUG.EXE displays the code for CMD.EXE, do the following:

1. Set a deferred breakpoint to stop when the DLL is loaded.
2. Run the program.

3. When the breakpoint is encountered, open the desired component in the DLL and set breakpoints.

To debug WorkPlace Shell objects, replace the RUNWORKPLACE line in your config.sys with the following line:

```
SET RUNWORKPLACE=C:/OS2/CMD.EXE
```

After rebooting, type the following at an OS/2 prompt:

```
icsdebug c:/os2/pmshell
```

1. Set a deferred breakpoint for the DLL containing the WPS program.
2. Run the program.
3. When the breakpoint is encountered, open the desired component in the DLL and set breakpoints.

Ending the Debugging Session

To end the debugging session, select **Close debugger** from the **File** menu in a debugger window. The **Close Debugger** window displays. Select one of the following choices:

- Select **Yes** to end your debugging session.
- Select **No** to return to the previous screen without exiting the debugger.

You can also end the debugging session by pressing F3 in any of the debugger windows.

25

Frequently Used Features

This section introduces the title bar buttons, ways to execute your program, and how to set breakpoints.

Using the Title Bar Buttons

Buttons have been provided for easier access to frequently used features. The following buttons are located in the title bar of the source windows:

Step over executes the current line in the program. If the current line is a call, execution is halted when the call is completed.

Step into executes the current line in the program. If the current line is a call, execution is halted at the first statement in the called function.

Step debug executes the current line in the program. The debugger steps over any function for which debugging information is not available (for example, library and system routines), and steps into any function for which debugging information is available.

Step return automatically executes the lines of code up to, and including, the return statement of the current function.

Run allows you to start and stop the program.

When the debugger is running, the **Run** button changes to the **Halt** button. You can click on the **Halt** button to halt the program execution. You can also interrupt the program you are debugging by selecting the **Halt** choice from the **Run** menu or by pressing SysRq (Alt+PrintScreen).

Frequently Used Features of the Debugger

View changes the current source window to one of the other source windows. For example, you can change from the **Disassembly** window to the **Mixed** window.

Monitor Expression displays the **Monitor Expression** window, which allows you to type in the name of the expression you want to monitor.

Call Stack displays the **Call Stack** window, which allows you to view all of the active functions for a particular thread including the PM calls. The functions are displayed in the order that they were called.

Registers displays the **Registers** window, which allows you to view all the processor and coprocessor registers for a particular thread.

Storage displays the **Storage** window, which shows the storage contents and the address of the storage.

Breakpoints displays the **Breakpoints List** window, which allows you to view all the breakpoints that have been set.

Control displays the **Control** window.

Note: If you place your mouse pointer on a button in the title bar, the name of that button displays in the window title area. As you drag the mouse pointer across the buttons, the name in the window title area changes to reflect the button you are on.

Executing a Program

You can execute your program by using step commands or the **Run** command.

Step commands Step commands control the execution of the program. The execution of the line of code is reflected in all open views, and is performed in the thread specific to the view.

The step commands are located in the title bar of the source windows and under the **Run** menu of the source windows.

To single step your program, click mouse button two. This executes the current line in the program.

Run command The **Run** command runs the program until a breakpoint is encountered, the program is halted, or the program ends.

You can start the **Run** command from the **Run** button in title bar or the **Run** menu of the source windows.

When you execute your program, a clock icon displays to indicate that the program is running and might require input to continue to the next breakpoint or termination of the program.

Setting Breakpoints

You can control how your program executes by setting breakpoints. A breakpoint stops the execution of your program at a specific location or when a specific event occurs.

To set breakpoints, select the **Breakpoints** menu from the **Control** window or from any of the source windows and select the the appropriate choice for the type of breakpoint you want to set. You can also set a simple line breakpoint by double-clicking in the *prefix area* of an executable statement in any of the source windows. The prefix area is the area to the left of the source code where line numbers or addresses are displayed. The prefix area turns *red* indicating that the breakpoint has been set.

📖 Refer to “Breakpoints Menu Choices” on page 413 for more information on breakpoints.

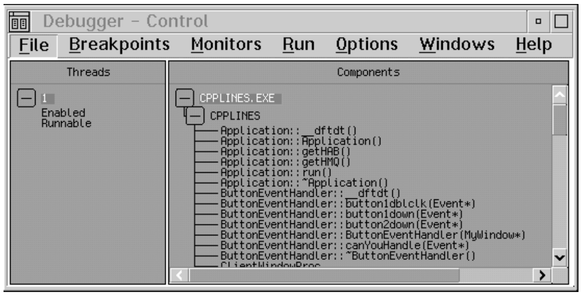
Frequently Used Features of the Debugger

Introducing the Main Debugger Windows

This section introduces the **Control** window and the three source windows which offer different views of your source code. It describes the menu items and choices that are located in each of these windows.

Using the Control Window

The **Control** window, as shown in Figure 79, is the control window of the debugger and displays during the entire debugging session. This window is divided in two panes. One pane shows the threads for the program you are debugging and the other pane shows the components for the program you are debugging.



```
1
Enabled
Runnable
```

```
CPPLINES.EXE
 CPPLINES
  Application::__dftdt()
  Application::Application()
  Application::getHAB()
  Application::getHMQ()
  Application::run()
  Application::~Application()
  ButtonEventHandler::__dftdt()
  ButtonEventHandler::button1dblclk(Event*)
  ButtonEventHandler::button1down(Event*)
  ButtonEventHandler::button2down(Event*)
  ButtonEventHandler::ButtonEventHandler(MyWindow*)
  ButtonEventHandler::canYouHandle(Event*)
  ButtonEventHandler::~ButtonEventHandler()
  ClientWindowProc
```

Figure 79. Control Window

The **Threads** box contains the threads and the state of the threads started by your program. The following states are possible for the threads listed in the **Threads** box:

- Enabled and runnable
- Disabled and runnable
- Critical and runnable
- Suspended and runnable
- Enabled and blocked
- Disabled and blocked
- Critical and blocked
- Suspended and blocked.

To display the state of a thread, select the plus icon to the left of the thread. To enable or disable the thread listed in the **Threads** pane, double-click on the word **Enabled** or **Disabled** depending on the state of the thread. You can also toggle the **Thread enabled** choice from the **Run** menu.

When a check mark symbol displays beside the **Thread enabled** choice, threads are enabled and the debugger allows the highlighted thread to execute. When the check mark symbol does not display, threads are disabled and do not execute.

The **Components** box shows the executable files that are associated with the program you are debugging.

To display a list of object files contained within an executable file, select the plus icon to the left of the executable file name. To open a source window of an object file, double-click on the object file name.

To display a list of functions for a specific object file, select the plus icon to the left of the object file name. To open a source window of a specific function, double-click on the function name.

You can display any object or function by double-clicking on the name in the **Components** box or by highlighting the component name and selecting a view from the **View** menu in one of the source windows.

You specify which components display in the **Components** list by selecting **Options → Window Settings → Only components with debugging data**. When this choice is enabled, only components compiled and linked with debugging data are listed. Otherwise, all components are listed.

File Menu Choices

Select choices from the **File** menu of the **Control** window to open a new source file, locate a particular function, open a source window that contains the next line to be executed, start a new debugging session, or end a debugging session.

Open new source...: Displays the **Open New Source** window which allows you to open a new source file. If you have multiple source files in your program, only one source file is initially displayed. Use the **Open New Source** window to open additional source files.

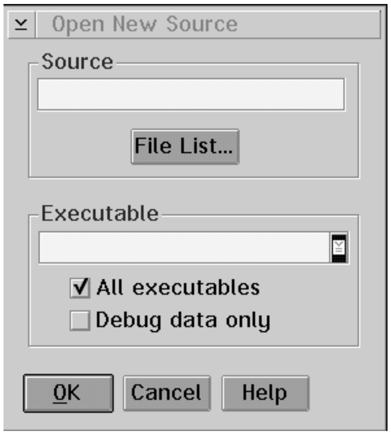


Figure 80. Open New Source Window

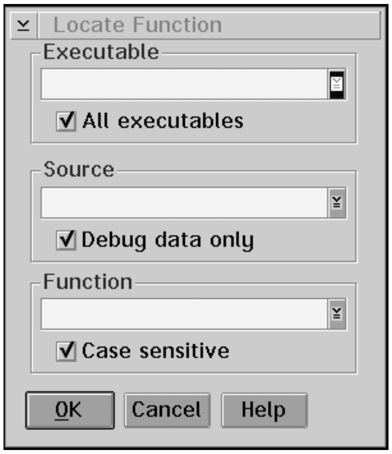
To use the **Open New Source** window:

- Type the name of the object file you want to open the source for in the **Source** entry field. For example, to look for the source used to compile A123.OBJ, type the following:

  ```
  A123.
  ```

  If you are unsure of the file name, select the **File List** push button to view a list of the files that you can select.
- Type the name of the executable file in the **Executable** entry field. The source files for the executable file display in the **Source** entry field.
- Enable the **All executables** check box if you want to search all the executable files. Disable the **All executables** check box to search for a particular executable file.
- Enable the **Debug data only** check box if you want to search only the source files that contain debugging information.
- Select the **OK** push button.

Locate function...: Displays the **Locate Function** window, shown in Figure 81, which allows you to open a source window to a particular function.



☑ All executables

☑ Debug data only

☑ Case sensitive

Figure 81. Locate Function Window

To use the **Locate Function** window:

- Type the name of the function you want to search for in the **Function** entry field.

  If the function that you specify is not found, the following message displays:

  ```
  No matching function found
  ```

  This means it may be a static function or the function you specified does not exist.
- Enable the **All executables** check box to search all the executable files for the function.

  The debugger searches each object file for global functions that match the function name specified. If an object file contains the global function that was specified, then it will also search for any static function with the same name.

  Note: To search for a function in a specific executable file, disable this check box and type the name of the executable file in the **Executable** entry field and type the name of the source file in the **Source** entry field.
- Enable the **Debug data only** check box if you want to search only the object files that contain debugging information.
- Enable the **Case sensitive** check box if you want to search for the string exactly as typed. Disable this check box if you want to search for both uppercase and lowercase characters.
- Select the **OK** push button.

Where is execution point: Opens a source window containing the next line to be executed.

Program startup...: Displays the **Program Startup** window, which allows you to start another debugging session. 🕮 Refer to "Starting the Debugger from OS/2" on page 401 for detailed information.

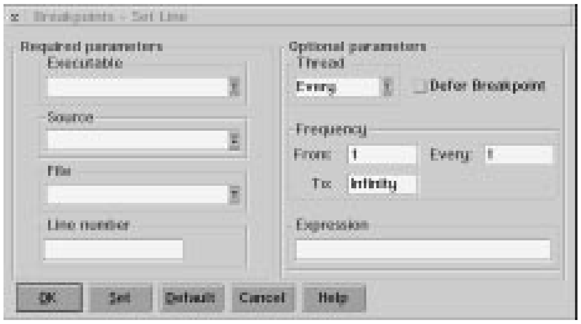
Close debugger: Ends the debugging session. When you select the **Close debugger** choice, the **Close Debugger** message box prompts you to confirm that you want to end the debugging session.

Breakpoints Menu Choices

Select choices from the **Breakpoints** menu to set breakpoints and to stop the execution of your program at any point. You can set as many breakpoints as you want.

Breakpoints can be set from the **Control** window or from the source windows. When you set a breakpoint in one source window, it is reflected in the other source windows.

Set line...: Displays the **Set Line** window which allows you to set a line breakpoint to stop the execution of your program at a specific line number.



Thread: Every

□ Defer Breakpoint

Frequency

From: 1

Every: 1

To: Infinity

Figure 82. Set Line Window

The **Set Line** window is divided into two group headings: **Required parameters** and **Optional parameters**.

The **Required parameters** group heading contains the following:

- **Executable Entry Field**

  To select a component from the **Executable** list:

1. Type the executable name in the entry field *or* open the **Executable** list by selecting the list button. This displays a drop-down combination box.
2. Highlight the executable where you want to set the breakpoint

- **Source Entry Field**

  To select a component from the **Source** list:
  1. Type the source name in the entry field *or* open the **Source** list by selecting the list button. This displays a drop-down combination box.
  2. Highlight the source where you want to set the breakpoint.
- **File Entry Field**

  If the source you selected has include files with executable statements, then the **File** list displays all the file names that contain executable lines. If the source you selected does **not** have include files, the **File** entry field does not display in this window.
  1. Type the name of the file in the entry field *or* open the **File** list by selecting the list button. This displays a drop-down combination box.
  2. Highlight the file where you want to set the breakpoint.
- **Line number Entry Field**

  To set a line breakpoint, type the line number in the **Line number** entry field. The breakpoint is set on the line number.

The **Optional parameters** group heading contains the following:

- **Thread Entry Field**

  To select a thread ID from the **Thread** list:
  1. Open the **Thread** list by selecting the list button. This displays a drop-down combination box.
  2. Highlight the thread where you want to set the breakpoint.

  Select EVERY, the default, to set a breakpoint in all of the active threads in your program. The **Every** choice is thread independent. Select one of the individual threads to set a breakpoint in one thread only. Threads are added to the **Thread** list as new threads are activated.
- **From Entry Field**

  This field is used for location breakpoints and load occurrence breakpoints. Type in a whole integer number to start activating the breakpoint the *nth* time the location is encountered.
- **To Entry Field**

  This field is used for location breakpoints and load occurrence breakpoints. Type in a whole integer number to stop activating the breakpoint after the *nth* time the location is encountered.
- **Every Entry Field**

This field is used for location breakpoints and load occurrence breakpoints. Type in a whole integer number to indicate how often the breakpoint should be activated within the **From** and **To** range.

- **Expression Entry Field**

  If you are setting an address, function, or line breakpoint, you can also type in an expression. The execution of the program stops only if this condition tests true. For example, you could type the following:

  ```
  (i==1) ¦¦ (j==k) && (k!=5)
  ```

  Note: Variables in a conditional expression associated with a FUNCTION breakpoint are limited to any static or global variables that are known to the called function when the function is called. Local variables and automatic variables cannot be used.

  The maximum length of the condition is 256 characters.

- **Defer breakpoint Check Box**

  Enable the **Defer breakpoint** check box if you want to set a breakpoint in a DLL that is not currently loaded.

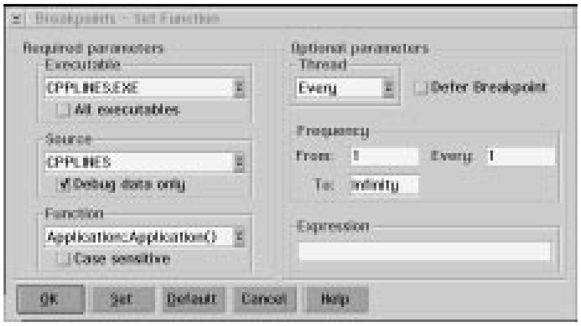
  Note: If your application consists of an EXE or preloaded DLLs, do not use this choice. If your application consists of DLLs that are dynamically loaded, use this choice to set breakpoints in DLLs which have not been loaded yet.

  If you set a deferred line breakpoint and the line is located in a template, the debugger sets the line breakpoint in all of the templates when the DLL is loaded.

  When a DLL is loaded and a deferred breakpoint has been set in the DLL, the state of the breakpoint changes from deferred to active. When a DLL is freed, any breakpoints that were set in the DLL change from the active state to deferred state.

  If you enter an invalid source, file or line number, the debugger will be unable to activate the breakpoint when the DLL is loaded. Therefore, the invalid breakpoint will remain in the deferred state even after the DLL is loaded.

Set function...: Displays the **Set Function** window which allows you to set a function breakpoint to stop the execution of your program when the first instruction of the function is encountered where the breakpoint has been set.



Required parameters

Executable: CPPLINES.EXE
□ All executables

Source: CPPLINES
☑ Debug data only

Function: Application::Application()
□ Case sensitive

Optional parameters

Thread: Every
□ Defer Breakpoint

Frequency
From: 1 Every: 1
To: infinity

Expression:

Figure 83. Set Function Window

The entry fields in this window are the same as in the **Set Line** window except for the following:

- **Function Entry Field**

  Type the name of the function where you want to set the breakpoint or select a function from the **Function** list. To select a function, do the following:
  1. Open the **Function** list by selecting the list button. This displays a drop-down combination box.
  2. Highlight the function where you want to set the breakpoint.

  If a function is overloaded, then a window displays with a list of all the overloaded function names. Select the appropriate function from the list.
- **Defer breakpoint Check Box**

  Enable the **Defer breakpoint** check box if you want to set a breakpoint in DLLs which have not been loaded yet.

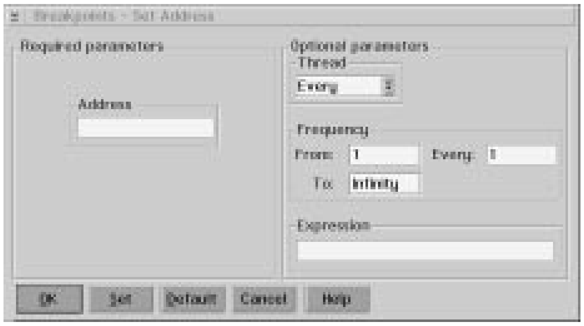
  If you set a deferred breakpoint in a function and that function is overloaded, the debugger sets the breakpoint in all of the overloaded functions when the DLL is loaded.

  When a DLL is loaded and a deferred breakpoint has been set in the DLL, the state of the breakpoint changes from deferred to active. When a DLL is freed, any breakpoints that were set in the DLL change from the active state to deferred state.

  If you enter an invalid source file or invalid function, the debugger will be unable to activate the breakpoint when the DLL is loaded. Therefore, the invalid breakpoint will remain in the deferred state even after the DLL is loaded.

For a description of the types of data you can enter in the entry fields under the **Optional parameters** group heading, 🕮 refer to “Set line...” on page 413.

Set address...: Displays the **Set Address** window, which allows you to set an address breakpoint to stop the execution of your program at a specific address.



Thread: Every
From: 1
Every: 1
To: Infinity

Figure 84. Set Address Window

The entry fields in this window are the same as in the **Set Line** window except for the following:

- **Address Entry Field**

  Type the name of the address in the **Address** entry field.

  Note: The address can be either segmented or flat format.

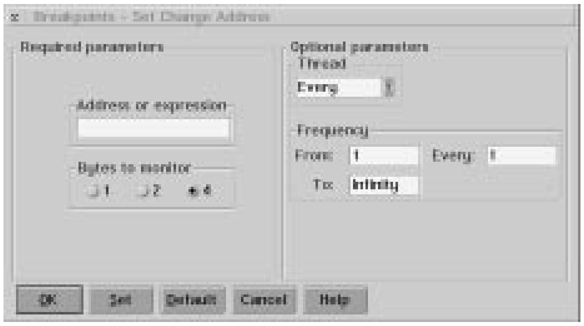
  For example, to set an address breakpoint for the address *0x000A1FCC*, you would type one of the following in the **Address** entry field.

  ```
  0x000A1FCC or A1FCC
  ```

  The 0x is optional.

For a description of the types of data you can enter in the entry fields under the **Optional parameters** group heading, 🕮 refer to “Set line...” on page 413.

Set change address...: Displays the **Change Address Breakpoint** window, which allows you to sets a change address breakpoint to stop the execution of your program when contents of memory at a given address changes where the breakpoint has been set.



Address or expression:

Bytes to monitor: ○ 1 ○ 2 ● 4

Thread: Every

Frequency
From: 1 Every: 1
To: Infinity

Figure 85. Change Address Breakpoint Window

Use the **Change Address Breakpoint** window to set a change address breakpoint. To do so, type a hexadecimal address or an expression and select the range of bytes.

Note: The debugger supports up to 4 enabled change address breakpoints. However, you can set as many disabled change address breakpoints as you want.

- **Address or expression Entry Field**

  Type a hexadecimal address or an expression that can be evaluated to a hexadecimal address.

  Note: If you type *ABC* in the **Address or expression** entry field, and there is a variable named ABC, it uses the value of the variable instead of the hex value ABC. Also, you can type *&a* in the **Address or expression** entry field to set the breakpoint on the address of a variable *a*.

  For example, type the following in the **Address or expression** entry field to set a change address breakpoint for the address *A1FCC*.

  ```
  A1FCC
  ```

  Type the following in the **Address or expression** entry field to set a change address breakpoint for the expression *&variable*.

  ```
  &variable
  ```

Warning: If you set a change address breakpoint that is on the call stack, you should remove the breakpoint before leaving the routine associated with the breakpoint. Otherwise, when you return from the routine, the routine's stack frame will be removed from the stack leaving the breakpoint intact. Any other routine that gets loaded on the stack will then contain the breakpoint.

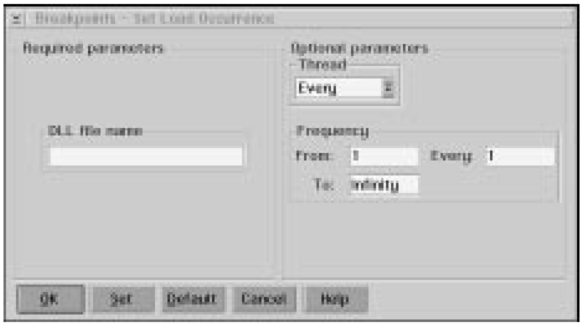
- **Bytes to monitor Radio Buttons**

Select one of the radio buttons to specify the range of bytes. The 2-byte range must be aligned on a word boundary and the 4-byte range must be aligned on a double-word boundary.

Execution stops when the specified range of memory changes.

For a description of the types of data you can enter in the entry fields under the **Optional parameters** group heading, 🕮 refer to "Set line..." on page 413.

Set load occurrence...: Displays the **Load Occurrence Breakpoint** window, which allows you to set a load occurrence breakpoint to stop the execution of your program when the DLL is encountered where the breakpoint has been set.



Thread: Every
Frequency
From: 1
Every: 1
To: infinity

Figure 86. Load Occurrence Breakpoint Window

To use the **Load Occurrence Breakpoint** window, type the name of the DLL in the **DLL file name** entry field. Execution stops when the DLL is loaded.

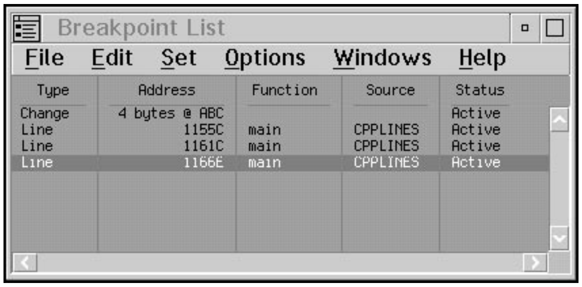
- **DLL file name Entry Field**

To set a load occurrence breakpoint when MY.DLL is loaded, you would type one of the following in the **DLL file name** entry field:

```
MY or MY.DLL
```

For a description of the types of data you can enter in the entry fields under the **Optional parameters** group heading, refer to “Set line...” on page 413.

List: Displays the **Breakpoint List** window, which lists all the breakpoints that have been set. It also shows the state of each breakpoints.



Type	Address	Function	Source	Status
Change	4 bytes @ ABC			Active
Line	1155C	main	CPPLINES	Active
Line	1161C	main	CPPLINES	Active
Line	1166E	main	CPPLINES	Active

Figure 87. Breakpoint List Window

Use the **Breakpoints List** window to display a list of the breakpoints that have been set. The following information is also provided for each breakpoint.

- The enablement state
- The type of breakpoint
- The position of the breakpoint
- The conditions under which the breakpoint is activated.

For more information on the **Breakpoint List** window, refer to “Using the Breakpoint List Window” on page 453.

Delete all: Deletes all the breakpoints that have set.

Monitors Menu Choices

Select choices from the **Monitors** menu of the **Control** window to open other debugging windows such as monitors, call stack, registers, and storage.

The first three choices listed under the **Monitors** menu are also accessible from the title bar buttons of the source windows.

Call stack Displays the **Call Stack** window, which allows you to monitor the call stack for a particular thread. This window is described in “Using the Call Stack Window” on page 443.

Registers Displays the **Registers** window, which allows you to monitor registers and flags for a particular component or thread. This window is described in “Using the Registers Window” on page 445.

Storage Displays the **Storage** window, which allows you to monitor the storage in your program. This window is described in “Using the Storage Window” on page 447.

Local variables Displays the **Local Variables** window, which allows you to display the local variables for the program's current function. This window is described in “Using the Local Variables Window” on page 450.

Window analysis Displays the **Window Analysis** window, which allows you to display the windows of the program in a three dimensional view. This window is described in “Using the Window Analysis Window” on page 459.

Message queue Displays the **Message Queue** window, which allows you to display the PM messages associated with a PM application. This window is described in “Using the Message Queue Window” on page 462.

Run Menu Choices

Select choices from the **Run** menu to execute your program, stop execution, or enable or disable threads.

Run: Executes the program from the current line until a breakpoint is encountered or the program ends.

Halt: Interrupts the program you are debugging. You can access this choice by pressing *SysRq (Alt+PrintScreen)*.

Program restart: Allows you to restart the debugging session.

Hide debugger on Run: Minimizes the debugger windows while your application is running.

Check heap when stopping: Checks all memory blocks allocated or freed by the compiler debug memory management functions to make sure that overwriting has not occurred outside the bounds of allocated blocks and free memory blocks have not been overwritten. When **Check heap when stopping** choice is enabled, each time the program stops, the heap is checked. For example, stopping at a breakpoint or at the end of a step command would cause the heap check to be performed. If a heap error is detected, your application terminates. The **Termination** window displays showing the source line number where the application stopped and the heap check was performed.

Notes

- For the **Check heap when stopping** choice to work, you have to compile your application using the *Tm+* compiler option.
- If you enable the **Check heap when stopping** choice and you run your application to termination, the heap check is not made. To check the heap just before termination, set a breakpoint on the last line of your application.
- If you are debugging a multiple thread program and a thread stops while running in compiler memory management code which is holding a memory semaphore, the heap check will not be performed.
- If the stopping thread is running in 16-bit code, the heap check will not be performed.

Thread enabled: Enables or disables threads.

When a thread is enabled, a check mark symbol displays beside the **Thread enabled** choice and the thread is executed when you run your program. When a thread is not enabled, a check mark symbol does not display and the highlighted thread is not executed when you run your program.

Options Menu Choices

Select choices from the **Options** menu to control how the debugger windows display.

Window settings→: Use the **Window settings** cascading choices to modify how characteristics are displayed in the **Control** window.

Only components with debug data
If enabled, only the components containing debugging information are displayed in the **Control** window.

Sort threads
If enabled, threads are sorted numerically in the **Control** window.

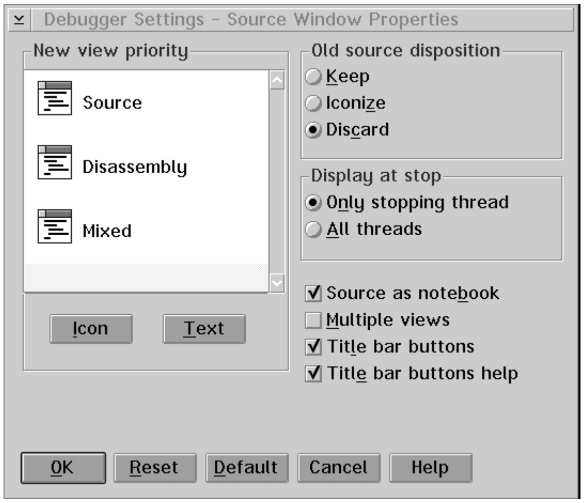
Sort components
If enabled, components are sorted alphabetically in the **Control** window.

Titles on
If enabled, the heading titles are displayed in the **Control** window.

Fonts...
Displays the system **Font Selection** window.

Debugger settings→: Use the **Debugger settings** cascading choice to set various debugger options that control how the debugger windows display. These settings affect the behavior of the debugger and remain in effect for the duration of the debugging session.

Source window properties...: Displays the **Source Window Properties** window, which allows you to select how the threads and source files initially display.



Debugger Settings – Source Window Properties

New view priority

- Source
- Disassembly
- Mixed

Old source disposition

○ Keep
○ Iconize
● Discard

Display at stop

● Only stopping thread
○ All threads

☑ Source as notebook
☐ Multiple views
☑ Title bar buttons
☑ Title bar buttons help

Figure 88. Source Window Properties Window

Use the **Source Window Properties** window to define the following:

- Which source window displays when the debugger starts.
- When a source window first displays during a debugging session.
- How to process a source window from which execution has just left. The window can remain displayed, be turned into an icon, or be discarded.

New view priority Group Heading

Source Displays the source code for the thread or component.

Disassembly Displays your source code as assembler instructions without symbolic information.

Mixed Displays a line of source code followed by the assembler instructions for that line of source code.

When a source window opens, the **New view priority** indicates which source window displays, subject to the availability of the source code. You can select the order in which the source windows display.

To change the order:

1. Press and hold mouse button two on the view icon you want to rearrange.
2. Drag the icon and release mouse button two when you have the icon in its new location.

You can display the source windows in the **New view priority** group heading as icons or text. Select the appropriate push button to set the display mode.

Old source disposition Group Heading

In the course of debugging, these selections allow you to control the behavior of source windows from which execution has just left. The **Old source disposition radio buttons** control the behavior of source windows within a thread.

The dispositions that the views can take are:

Keep Leaves open the source windows that contain the components and threads that you select with **Display at stop**.

Iconize Changes into icons the views that contain the components and threads that you select with **Display at stop**.

Discard Disposes of the views that contain the components and threads that you select with **Display at stop**.

Display at stop Group Heading

You can control how many source windows are displayed using the following radio buttons:

The choices are:

Only stopping thread Keeps, iconizes, or discards all views that are not the stopping thread.

All threads Keeps, iconizes, or discards the views for old components within each thread.

For example, if you select **Only stopping thread**, the **Old source disposition** applies to all of the source windows except the current view of the stopping thread. If you select **All threads**, the **Old source disposition** applies only to the source windows for the components from which execution has just left within a thread.

Source as notebook Check Box

You can display your source window in a notebook format if there are include files in the source file.

Multiple views Check Box

You can choose to display more than one source window for a particular source file.

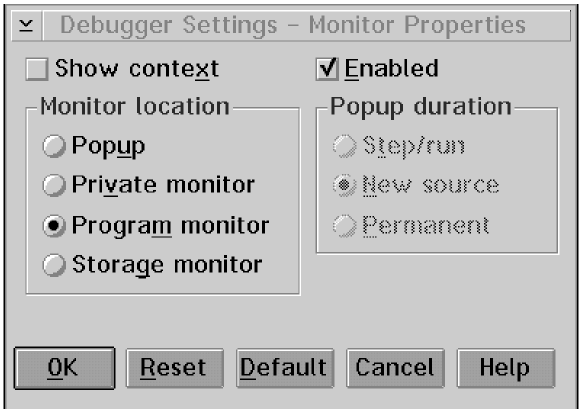
Title bar buttons Check Box

You can choose to display or not display the title bar buttons. The default is to display the title bar buttons.

Title bar buttons help Check Box

You can choose to display or not display the title bar buttons hover help. The default is to display the hover help.

Monitor properties...: Displays the **Monitor Properties** window, which allows you to select the settings for monitoring variables or expressions.



Debugger Settings – Monitor Properties

□ Show context ☑ Enabled

Monitor location
- ○ Popup
- ○ Private monitor
- ● Program monitor
- ○ Storage monitor

Popup duration
- ○ Step/run
- ● New source
- ○ Permanent

.

Figure 89. Monitor Properties Window

Use the **Monitor Properties** window to set the following:

- Whether the context for variables or expressions displays in the monitor windows.
- The window into which the variable or expression being monitored is placed.

- Whether the displayed contents of the variable or expression are updated as the state of the program changes.
- For popup expression windows, how long the monitor windows display.

Show context Check Box

Select the **Show context** check box to display the context for variables or expressions when they are selected for monitoring.

Monitor location Group Heading

Choose one of the following radio buttons to select the monitor window that opens when you select a variable or expression to monitor. The selections you can make, and the corresponding windows, are:

Popup
Display the variable or expression in a popup expression window.

Private monitor
Display the variable or expression in the **Private Monitor** window.

Program monitor
Display the variable or expression in the **Program Monitor** window.

Storage monitor
Display the variable or expression in the **Storage** window.

Enabled Check Box

Select the **Enabled** check box to update the displayed contents of variables when they are selected for monitoring.

Popup duration Group Heading

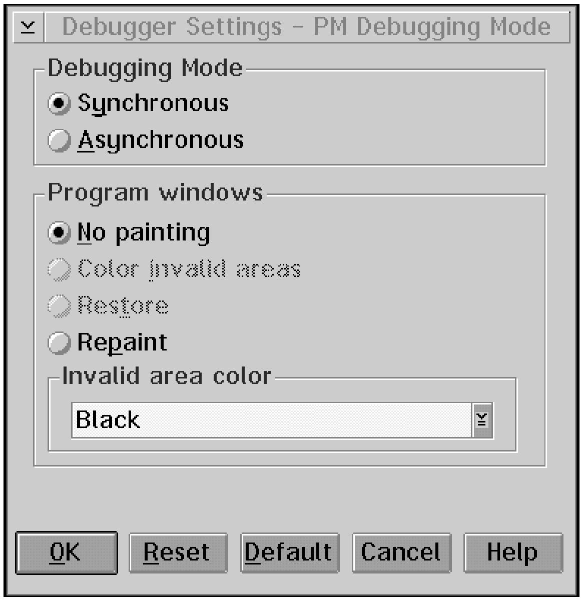
If you select **Popup** from the **Monitor location** group heading, select one of the following radio buttons to specify how long the popup expression window displays:

Step/run The monitor window closes when the next step command or **Run** is executed.

New source The monitor window closes when execution stops in a new source.

Permanent This monitor window is associated with a specific source window and closes when the associated source window closes.

PM debugging mode...: Displays the **PM Debugging Mode** window, which allows you to set the debugging mode and control the interaction between the program windows and PM.



Debugger Settings - PM Debugging Mode

Debugging Mode
● Synchronous
○ Asynchronous

Program windows
● No painting
○ Color invalid areas
○ Restore
○ Repaint

Invalid area color
Black

Figure 90. PM Debugging Mode Window

Use the **PM Debugging Mode** window to set the debugging mode to *asynchronous* or *synchronous*.

When the debugger is operating in asynchronous mode and the program you are debugging is stopped, the debugger immediately responds to messages that have been sent to the program being debugged on the program's behalf. The debugger answers the messages with a simple default response, freeing up other processes to operate while the debugger has control. When you are running the debugger in asynchronous mode, other PM applications running in the system are not blocked when the program being debugged stops.

Warning:

Do not operate the debugger in asynchronous mode if the PM application that you are debugging requires the appropriate response to its messages. For example, a dynamic data exchange (DDE) message would require the appropriate response.

When the debugger is operating in synchronous mode, the messages that are passed between PM applications are answered by their target applications in the order that they were created. The messages that are passed within the debugger take priority over any other messages that are passed in the system.

When the program being debugged is stopped and the debugger is in synchronous mode, other PM applications are locked, leaving the debugger free to operate. In synchronous mode, you will not be able to use any other PM applications that are running.

The PM system is a message-based system. As program events are encountered by PM programs, the programs communicate with each other by passing messages and by receiving user input through input messages. When a PM program encounters an enabled breakpoint, the input queue can become blocked and dependent program events, or processes, can also become blocked as a result. For example, the input queue can become blocked when your program stops at a breakpoint that has been triggered by an input event.

Debugging mode Group Heading

Select one of the following radio buttons to set the debugging mode:

- Synchronous
- Asynchronous

Program windows Group Heading

No painting radio button

Select this radio button if you want none of the invalid areas of the window to be repainted.

Color invalid areas radio button

The **Color invalid areas** option works only in asynchronous mode. This option paints the invalid areas in a solid fill color. The color can be changed by selecting a different color from the **Invalid area color** combination box.

Restore radio button

The **Restore** option works only in asynchronous mode. This option restores the application window with the last available image of the window. The image that you can regain consists of the last available image when a step or run command ended, minus any parts of the window that were covered when the step or run command ended. The parts of the window that were covered are filled with the solid color you chose from the **Invalid area color** combination box.

Repaint radio button

Restores that application window with the last available image. The image that you can regain consists of the last available image when a step or run command ended minus any parts of the window that were covered when the step or run command ended. The parts of the window that were covered are filled with the solid color you chose from the **Invalid area color** combination box.

The **Repaint** option differs from the **Restore** option. It interrupts the normal debugging process of the window as follows:

- The program windows will not receive any screen interaction messages while the application is stopped. For example, the application will not receive any of the WM_MOUSEMOVE or WM_PAINT messages that were generated while the application was stopped.
- An extra WM_PAINT message is generated for the program windows when execution resumes.

  Note: The program windows might not process the WM_PAINT message depending on where the breakpoints are set or on which step or run command was selected.

Invalid area color Group Heading

Select the color that is to be used to repaint the invalid area of an application window. Depending on the original color of the application window, certain colors will be more appropriate for repainting. The color you choose is used when you select the **Color invalid areas**, **Restore**, or **Repaint** options.

Default data representation →: Use the **Default data representation** cascading choices to change the representation for a data type in a specific language.

Select a language to change the default representation of the selected data type. For example, you can change the default representation for an integer in the C language from decimal to hexadecimal. Select the **System** choice to change the default representation of the math coprocessor registers. This choice is language independent.

Program profiles→: Use the **Program profiling** cascading choices to enable program profiling, delete program profiles, or change the location where the program profiles are stored,

Program profiles are used to restore the debugger windows and breakpoints when debugging a program more than once. They are stored separately for each program debugged. The file extension for the files that contain this information is @2R.

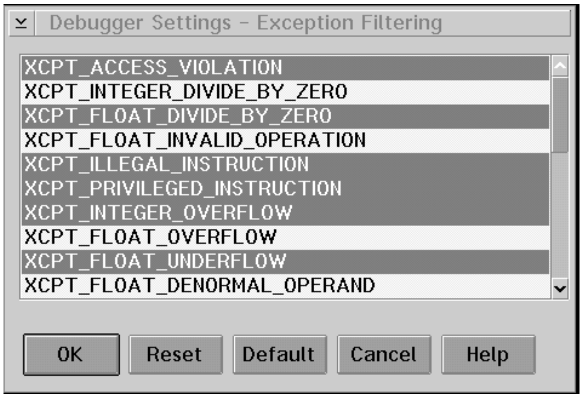
Note: Only information for executable files and preloaded DLLs relating to the primary thread is restored.

Enable program profiling
Enables program profiling so that program profiles are saved in a file for use when debugging the program again.

Delete
Delete program profiles for a program that you have debugged.

Change location
Change the location of the file that holds the program profiles. This also moves any existing profiles to the new directory.

Exception filtering...: Displays the **Exception Filtering** window, shown in Figure 91, which allows you to select which exceptions you want the debugger to recognize.



XCPT_ACCESS_VIOLATION
XCPT_INTEGER_DIVIDE_BY_ZERO
XCPT_FLOAT_DIVIDE_BY_ZERO
XCPT_FLOAT_INVALID_OPERATION
XCPT_ILLEGAL_INSTRUCTION
XCPT_PRIVILEGED_INSTRUCTION
XCPT_INTEGER_OVERFLOW
XCPT_FLOAT_OVERFLOW
XCPT_FLOAT_UNDERFLOW
XCPT_FLOAT_DENORMAL_OPERAND

Figure 91. Exception Filtering Window

To highlight an exception, do the following:

1. Select the exception by clicking on the name. It becomes highlighted.
2. Select the **OK** push button.

If a highlighted exception is encountered during the execution of your program, the **Application Exception** window is displayed. Any other exceptions that are encountered are ignored.

Using the Application Exception Window: During execution of the program by the debugger, it is possible that the program can generate an exception. If this happens, the debugger suspends execution of the program and indicates the location within the code where the exception occurred.

The **OS/2 Application Exception** window displays with the following choices:

Examine/Retry	You can investigate the cause of the exception and retry execution of the line that caused the fault.
Step Exception	The debugger steps into the first registered exception handler, which is tracked by OS/2. Execution stops at the first executable line of code in that exception handler.
Run Exception	The debugger runs the exception handlers.

Save window positions and sizes: When you select this choice, the window positions and sizes are saved for each type of debugger window currently open.

Alternatively, you can select the **Autosave window positions and sizes** choice to enable automaic saving of window positions and sizes.

Autosave window positions and sizes: Select the **Autosave window positions and sizes** choice to enable automatic saving of window positions and sizes for each *type* of debugger window currently open. Type refers to the different windows such as **Registers**, **Storage**, and so on.

When you enable this choice, the window positions and sizes are automatically saved when you move or resize a window. When you close the debugging session and start another session the positions and sizes that you selected are displayed in the new session.

Alternatively, you may save the window positions and sizes by positioning the debugger windows on the desktop and selecting the **Save window positions and sizes** choice.

Windows Menu Choices

Select the **Windows** menu of the **Control** window to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

Help Menu Choices

Select choices from the **Help** menu of the **Control** window to display the various types of help information.

Help index Displays an alphabetical index of all available debugger help topics.

General help Displays help information for the active window.

Using help Describes how to use the IBM C/C++ Debugger help facility.

How do I Displays the debugger task help.

Product information Displays product information.

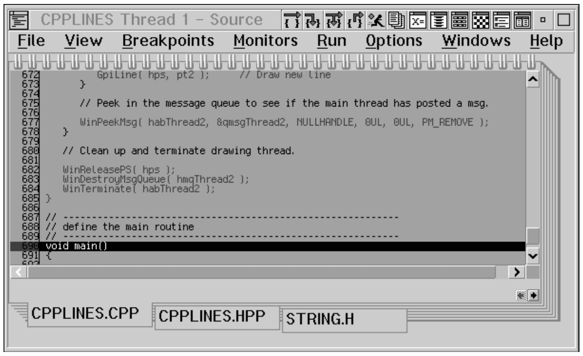
Using the Source Windows

A source window allows you to view the program you are debugging. You can look at your source in one of the following windows:

- **Source**
- **Disassembly**
- **Mixed**.

A source window is thread-specific. Executable lines initially display in blue and non-executable lines initially display in black.

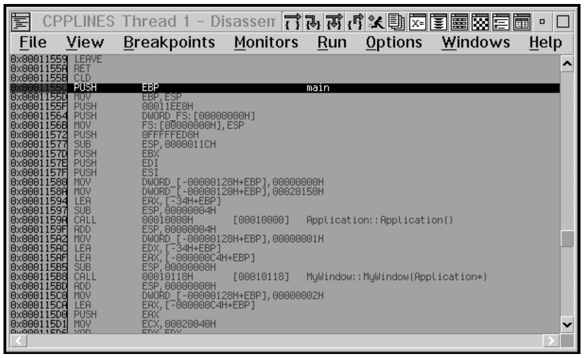
The **Source** window, as shown in Figure 92, displays the source code for the object that contains the main function to the program being debugged. If it is available, the **Source** window displays with the **Control** window when the debugging session starts. Otherwise, the **Disassembly** window displays.



```
            GpiLine( hps, pt2 );     // Draw new line
        }

        // Peek in the message queue to see if the main thread has posted a msg.

        WinPeekMsg( habThread2, &qmsgThread2, NULLHANDLE, 0UL, 0UL, PM_REMOVE );
    }

    // Clean up and terminate drawing thread.

    WinReleasePS( hps );
    WinDestroyMsgQueue( hmqThread2 );
    WinTerminate( habThread2 );
}

// ------------------------------------------------------------
// define the main routine
// ------------------------------------------------------------
void main()
{
```

Figure 92. Source Window

The **Disassembly** window, as shown in Figure 93 on page 433, displays the assembler instructions for your program, without symbolic information.



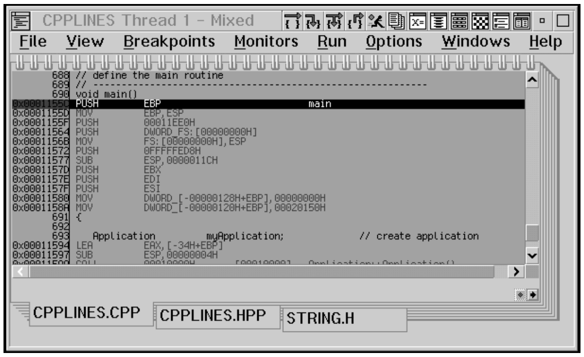
```
0x00011559 LEAVE
0x0001155A RET
0x0001155B CLD
0x0001155C PUSH        EBP                        main
0x0001155D MOV         EBP,ESP
0x0001155F PUSH        00011EE0H
0x00011564 PUSH        DWORD_FS:[00000000H]
0x0001156B MOV         FS:[00000000H],ESP
0x00011572 PUSH        0FFFFFED8H
0x00011577 SUB         ESP,0000011CH
0x0001157D PUSH        EBX
0x0001157E PUSH        EDI
0x0001157F PUSH        ESI
0x00011580 MOV         DWORD_[-00000128H+EBP],00000000H
0x0001158A MOV         DWORD_[-00000120H+EBP],00020150H
0x00011594 LEA         EAX,[-34H+EBP]
0x00011597 SUB         ESP,00000004H
0x0001159A CALL        00010000H       [00010000]   Application::Application()
0x0001159F ADD         ESP,00000004H
0x000115A2 MOV         DWORD_[-00000128H+EBP],00000001H
0x000115AC LEA         EDX,[-34H+EBP]
0x000115AF LEA         EAX,[-000000C4H+EBP]
0x000115B5 SUB         ESP,00000008H
0x000115B8 CALL        00010118H       [00010118]   MyWindow::MyWindow(Application*)
0x000115BD ADD         ESP,00000008H
0x000115C0 MOV         DWORD_[-00000128H+EBP],00000002H
0x000115CA LEA         EAX,[-000000C4H+EBP]
0x000115D0 PUSH        EAX
0x000115D1 MOV         ECX,00020040H
```

Figure 93. Disassembly Window

The **Mixed** window, as shown in Figure 94, displays your program as follows:

- Each lines of source code is prefixed by its line number, as in the **Source** window.
- Each disassembled line is prefixed by an address, as in the **Disassembly** window.
- Source comment lines also display.
- The lines of source code are treated as comments within the lines of disassembly code. You can only set breakpoints or run your program on lines of disassembly code.

Note: The **Mixed** window cannot be opened if the source code is not available.



```
       688 // define the main routine
       689 // -----------------------------------------------------------
       690 void main()
0x0001155C PUSH        EBP                        main
0x0001155D MOV         EBP,ESP
0x0001155F PUSH        00011EE0H
0x00011564 PUSH        DWORD_FS:[00000000H]
0x0001156B MOV         FS:[00000000H],ESP
0x00011572 PUSH        0FFFFFED8H
0x00011577 SUB         ESP,0000011CH
0x0001157D PUSH        EBX
0x0001157E PUSH        EDI
0x0001157F PUSH        ESI
0x00011580 MOV         DWORD_[-00000128H+EBP],00000000H
0x0001158A MOV         DWORD_[-00000120H+EBP],00020150H
       691 {
       692
       693    Application        myApplication;            // create application
0x00011594 LEA         EAX,[-34H+EBP]
0x00011597 SUB         ESP,00000004H
```

Figure 94. Mixed Window

File Menu Choices

The choices in the **File** menu are the same as those listed under the **File** menu of the **Control** window. Refer to “File Menu Choices” on page 410 for a description of the choices.

View Menu Choices

Select choices from the **View** menu to locate strings of text, scroll to a particular line, view include files, change the current window to a notebook format, or select a different view of your program.

Find...: Displays the **Find** window which allows you to search for a text string.

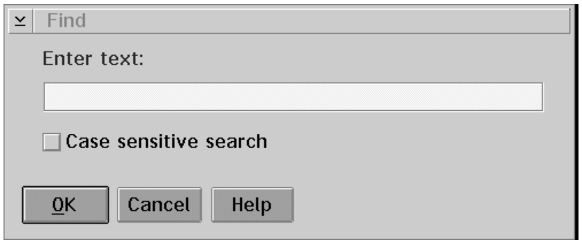


Figure 95. Find Window

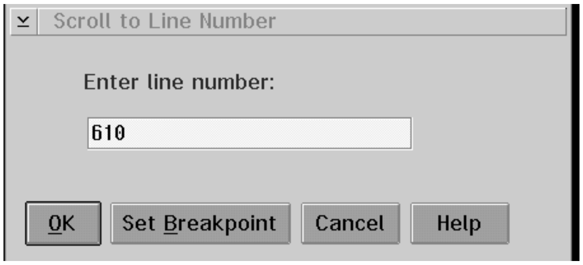
To use the **Find** window to search for a text string:

1. Type the text string you want to search for in the **Enter text** entry field.
2. Enable the **Case sensitive** check box if you want to search for the string exactly as typed. Disable this check box to search for uppercase and lowercase characters.
3. Select the **OK** push button.

The search string can have alphabetic and numeric characters, a maximum of 256 characters, and uppercase and lowercase characters.

Find next: Allows you to search for the next occurrence of the text string that you typed in the **Find** window.

Scroll to line number...: Displays the **Scroll to Line Number** window, which allows you to go to a particular line in your program or set a line breakpoint.



Enter line number: 610

Figure 96. Scroll to Line Number Window

To use the **Scroll to Line Number** window to scroll to a specific line:

1. Type the line number you want to scroll to in the **Enter line number** entry field.
2. Select the **OK** push button to scroll to that line.

Note: If the **Source** window is active, just type a number and the **Scroll to Line Number** window automatically displays.

To use the **Scroll to Line Number** window to set a breakpoint:

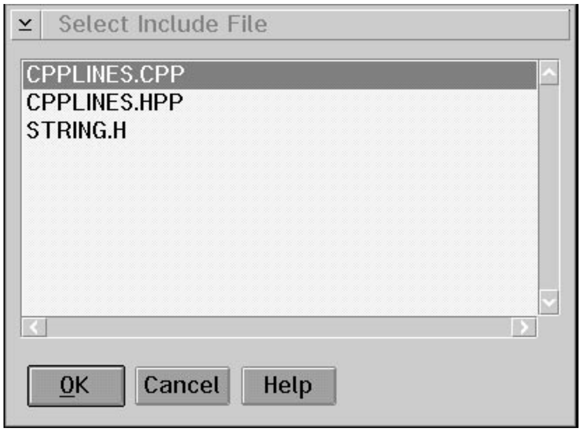
1. Type the line number you want to set the breakpoint on in the **Enter line number** entry field.
2. Select the **Set Breakpoint** push button to set the breakpoint on the specified line number.

Notebook: Enable the **Notebook** choice to display the source windows in notebook format.

Note: If you have include files in your program, the **Notebook** choice enables by default.

The **Source** and **Mixed** windows can be displayed in a notebook format. The **Disassembly** window cannot be displayed in a notebook format.

Select include...: Displays the **Select Include File** window that allows you to view the files that are included in your program.



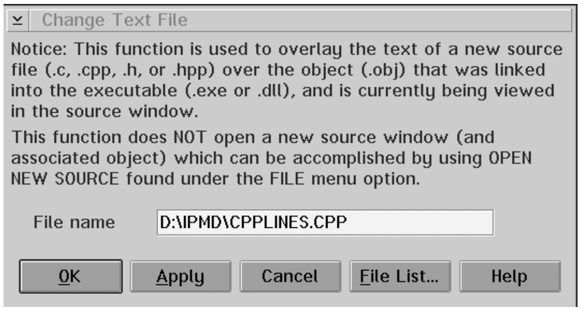
CPPLINES.CPP
CPPLINES.HPP
STRING.H

Figure 97. Select Include File Window

To use the **Select Include File** window:

1. Select the include file. The include file name is highlighted.
2. Select the **OK** push button. The selected include file view displays.

Change text file...: Displays the **Change Text File** window, which allows you to specify a file name to be used as the source in the current view. This is useful if the debugger found the incorrect source file for your program, so that you can specify the use of a different source file from a different directory.



Notice: This function is used to overlay the text of a new source file (.c, .cpp, .h, or .hpp) over the object (.obj) that was linked into the executable (.exe or .dll), and is currently being viewed in the source window.

This function does NOT open a new source window (and associated object) which can be accomplished by using OPEN NEW SOURCE found under the FILE menu option.

File name D:\IPMD\CPPLINES.CPP

Figure 98. Change Text File Window

Use the **Change Text File** window to replace the path name or file name of the program you are debugging with a new path name or file name.

To replace the file name:

1. Type the new path name or file name in the **File name** entry field.
2. Select the appropriate push button.

Source: Displays the **Source** window, which displays the source code for the object that contains the main function to the program being debugged.

📖 Refer to “Using the Source Windows” on page 432 for more information.

Disassembly: Displays the **Disassembly** window, which displays the assembler instructions for your program, without symbolic information.

📖 Refer to “Using the Source Windows” on page 432 for more information.

Mixed: Displays the **Mixed** window, which is a combination of the **Source** and **Disassembly** windows.

📖 Refer to “Using the Source Windows” on page 432 for more information.

Breakpoints Menu Choices

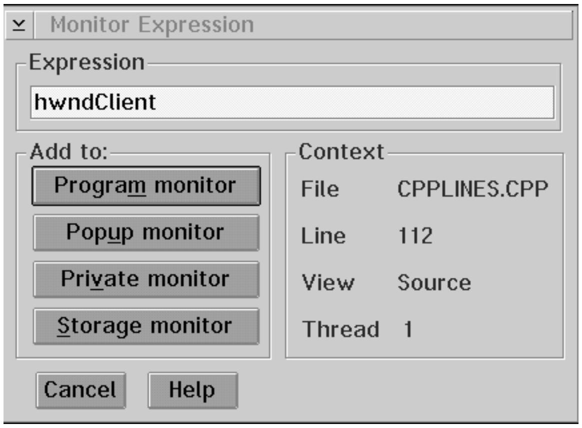
The choices listed under the **Breakpoints** menu are the same as those listed for the **Control** window. 📖 Refer to “Breakpoints Menu Choices” on page 413 for a description of the **Breakpoints** menu choices.

Monitors Menu Choices

Select choices from the **Monitors** menu of the source windows to monitor expressions or variables and view the other debugger windows such as call stack, registers, and so on.

The first four choices listed under the **Monitors** menu are also accessible from the title bar buttons in the source windows.

Monitor expression...: Displays the **Monitor Expression** window, shown in Figure 99, which allows you to monitor expressions or variables and add them to various monitor windows.



Expression: hwndClient

File CPPLINES.CPP
Line 112
View Source
Thread 1

Figure 99. Monitor Expression Window

Use the **Monitor Expression** window to type the name of the expression you want to monitor. This window lists the following contextual information:

- The component you are in.
- The view of the program that is active.
- The active line of the source code, which is highlighted.
- The thread you are in.

To specify an expression to be monitored:

1. Type the name of the variable or expression you want to monitor in the **Expression** entry field.
2. Select the appropriate push button from the **Add to** group heading for the location from where you want to monitor your expression.

   Note: The expression displays as specified in the **Monitor Properties** window. To change the default location, select **Monitor properties** from the **Debugger settings** choice from the **Options** menu in the source windows or the **Control** window.

Call Stack: Displays the **Call Stack** window, which allows you to monitor the call stack stack for a particular thread. This window is described in "Using the Call Stack Window" on page 443.

Registers: Displays the **Registers** window, which allows you to monitors registers and flags for a particular component or thread. This window is 📖 described in "Using the Registers Window" on page 445.

Storage: Displays the **Storage** window, which allows you to monitors the storage in your program. This window is 📖 described in "Using the Storage Window" on page 447.

Local variables: Displays the **Local Variables** window, which allows you to display the local variables for the program's current function. This window is 📖 described in "Using the Local Variables Window" on page 450.

Window analysis: Displays the **Window Analysis** window, which allows you to display the windows of the program in a three dimensional view. This window is 📖 described in "Using the Window Analysis Window" on page 459.

Message queue: Displays the **Message Queue** window, which allows you to display the PM messages associated with a PM application. This window is 📖 described in "Using the Message Queue Window" on page 462.

Run Menu Choices

Select choices from the **Run** menu to perform step commands, run your program, restart the debugging session, hide debugger windows, enable heap check, and enable or disable threads.

Step over: Executes the current, highlighted line in the program, but does not enter any called function.

Step into: Executes the current, highlighted line in the program and enters any called program or function.

Step debug: Executes the current, highlighted line in the program. The debugger steps over any function for which debugging information is not available (for example, library and system routines), and steps into any function for which debugging information is available.

Step return: Automatically executes the lines of code up to, and including, the return statement of the current function.

Run: Runs the program, executing all enabled threads. Control returns to the debugger when the program ends or execution stops at an enabled breakpoint.

Halt: Interrupts the program you are debugging. You can also access this choice by pressing *SysRq (Alt+PrintScreen)*.

Program restart: Select the **Program restart** choice to start the debugging session again. **Program restart** allows you to restart the current debugging session on the existing program, while **Program startup** allows you to debug another program.

Run to location: Executes your program from the current line up to the line that is highlighted or gray in the prefix area.

To use the **Run to location** choice:

1. Single-click in the prefix area of the line you want to become the current line. The prefix area turns gray.
2. Select the **Run to location** choice. The program runs up to the line that you marked.

The **Run to location** choice stops only on executable lines. If a highlighted line is not executable, the run is not performed.

Jump to Location: Select the **Jump to Location** choice to change the current line in your program without executing the lines between the present current line and the new current line.

To use the **Jump to location** choice:

1. Single-click in the prefix area of the line you want to become the current line. The prefix area turns gray.
2. Select the **Jump to location** choice. The current line is changed and the lines between are not executed.

The **Jump to location** choice stops only on executable lines. If a highlighted line is not executable, the jump is not performed.

Warning: Jumping out of the current function may corrupt the call stack and cause unpredictable results.

Hide debugger on Run: For a description of this choice, 📖 refer to "Hide debugger on Run" on page 421.

Check heap when stopping: For a description of this choice, 📖 refer to "Check heap when stopping" on page 421

Thread enabled: For a description of this choice, 📖 refer to "Thread enabled" on page 422.

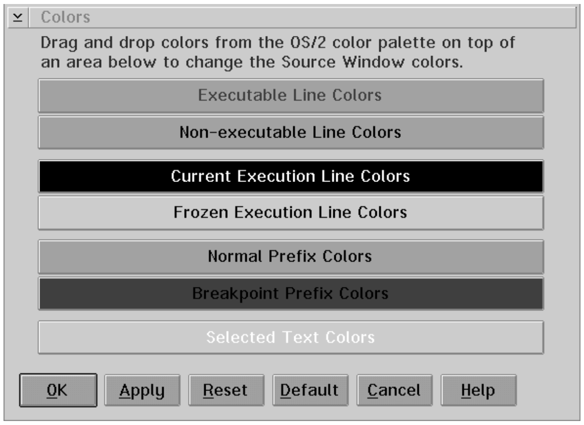
Options Menu Choices

Select choices from the **Options** menu to control how the debugger windows display.

Window settings→: Use the **Window settings** cascading choices to modify the colors and the font of the source windows.

Colors...

Displays the **Colors** window, shown in Figure 100, which allows you to change the color of the various window elements.



Drag and drop colors from the OS/2 color palette on top of an area below to change the Source Window colors.

Figure 100. Colors Window

Use the **Colors** window to change the color of the background and foreground (text) in the source windows.

To change the color of the background in the source window:

- Open the **OS/2 Color Palette** window.
- Using the mouse pointer, select a color from the color palette.
- Hold down mouse button two and drag the selected color into the **Colors** window.
- Release mouse button two over the text line that represents the source window area that you want to change.
- Select the **Apply** push button.

To change the color of the foreground in the source window:

- Open the **OS/2 Color Palette** window.
- Using the mouse pointer, select a color from the color palette.

- Hold down the **Ctrl** key and mouse button two and drag the selected color into the **Colors** window.
- Release the **Ctrl** key and mouse button two over the text line that represents the source window area that you want to change.
- Select the **Apply** push button.

Note: To change the colors in the other debugger windows, simply drag and drop the selected colors directly on the window that you want to change.

Fonts...

Displays the **Font Selection** window.

Debugger settings→: Use the **Debugger settings** cascaded choices to set various debugger options that control how the debugger windows display. These settings affect the behavior of the debugger and remain in effect for the duration of the debugging session.

🕮 Refer to "Debugger settings→" on page 423 for a description of the choices.

Windows Menu Choices

Select the **Windows** menu of the source windows to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

Help Menu Choices

Select choices from the **Help** menu of the source windows to display the various types of help information.

🕮 Refer to "Help Menu Choices" on page 432 for a description of the help choices.

27 Introducing the Basic Debugging Windows

This section introduces the basic debugging windows. These are the windows that are normally located in the **Monitors** menu of the **Control** and the source windows. The windows include call stack, registers, storage, monitor, and breakpoints.

Using the Call Stack Window

The **Call Stack** window as shown in Figure 101, lists all of the active functions for a particular thread including the PM calls. The functions are displayed in the order that they were called.

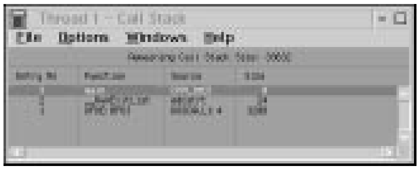


Figure 101. Call Stack Window

Each **Call Stack** window displays call stack information for one thread only. When the state of the program changes, such as when you execute the program or you update displayed data, the **Call Stack** window changes to reflect the current state. You can double-click on any call stack entry to display the source code for that entry. The line that calls the next stack entry is highlighted. The remaining stack size shows the bytes left in the stack for the thread.

To display the **Call Stack** window, select **Call Stack** from the **Monitors** menu or select the **Call Stack** button from the title bar .

File Menu Choice

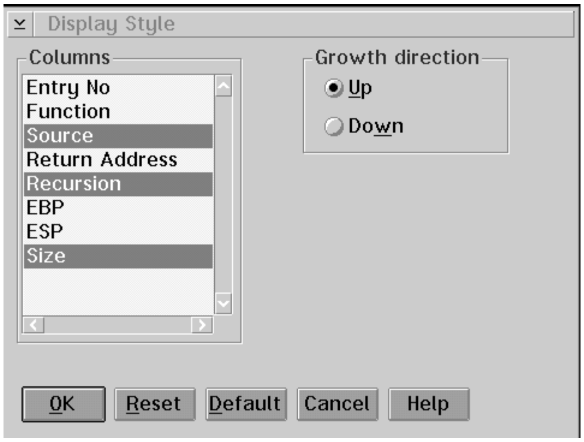
Use the choice from the **File** menu to end the debugging session.

The **Close debugger** choice allows you to end the current debugging session. When you select **Close debugger**, the **Close Debugger** message box prompts you to confirm that you want to end the debugging session.

Options Menu Choices

Use choices from the **Options** menu to control how the items on the call stack display and select the font you want for the **Call Stack** window.

Display style...: Displays the **Display Style** window, shown in Figure 102, which allows you to select the type of information you want displayed in the call stack and choose how the items are to be displayed.



Growth direction: ● Up ○ Down

Figure 102. Display Style Window

To use the **Display Style** window:

1. Select one or more of the items under the **Columns Group Heading** to display for each call stack entry. Each item causes a new column to be added to the **Call Stack** window.

   The following items are available:

   Entry No. Represents the position of the call stack item in the list. Entry level 1 is the first function invoked.

   Function Lists program name or the address of the function call that created the new call stack entry.

   Source Lists the component name that contains the function. The name displayed corresponds with a name listed in the **Components** list box in the **Control** window.

   Return Address The address represents where execution will return in that function.

   Recursion Lists the recursion level. 0 is the first invocation.

EBP Start of the call stack frame for that function.

ESP End of the call stack frame for that function.

Size Size of the call stack frame for that function.

2. Select one of the following **Growth direction** radio buttons to determine how new items are displayed on the call stack.

   Up Displays new items at the top of the **Call Stack** window.

   Down Displays new items at the bottom of the **Call Stack** window.

Fonts...: Displays the **Fonts Selection** window, which allows you to select the type of font you want to use for the **Call Stack** window.

Windows Menu Choices

Select the **Windows** menu of the **Call Stack** window to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

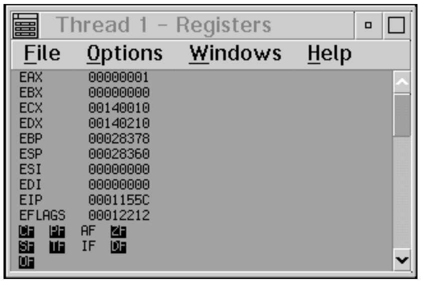
Help Menu Choices

Select choices from the **Help** menu of the **Call Stack** window to display the various types of help information.

Refer to “Help Menu Choices” on page 432 for a description of the help choices.

Using the Registers Window

The **Register** window, as shown in Figure 103 on page 446, lists all the processor and coprocessor registers for a particular thread. The contents of all of the registers except ST0 through ST7 are displayed in hexadecimal. To update a register, type over the contents that are displayed in the register. To toggle the value of a 1-bit flag, double-click on it or place the cursor on it and press Enter.



```
EAX       00000001
EBX       00000000
ECX       00140010
EDX       00140210
EBP       00028378
ESP       00028360
ESI       00000000
EDI       00000000
EIP       0001155C
EFLAGS    00012212
CF   PF   AF   ZF
SF   TF   IF   DF
OF
```

Figure 103. Registers Window

In the **Registers** window, floating-point registers display as floating-point decimal numbers. They can be updated with a floating-point decimal number or with a hexadecimal string that represents a floating-point number.

To display the processor registers and flags, including the math coprocessor information, select **Registers** from the **Monitors** menu or select the **Register** button from the title bar .

File Menu Choice

Use the choice from the **File** menu to end the debugging session.

The **Close debugger** choice allows you to end the current debugging session. When you select **Close debugger**, the **Close Debugger** message box prompts you to confirm that you want to end the debugging session.

Options Menu Choice

Use the choice from the **Options** menu to select the font you want for the **Registers** window.

When you select **Fonts...**, the **Font Selection** window displays.

Windows Menu Choices

Select the **Windows** menu from the **Registers** window to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

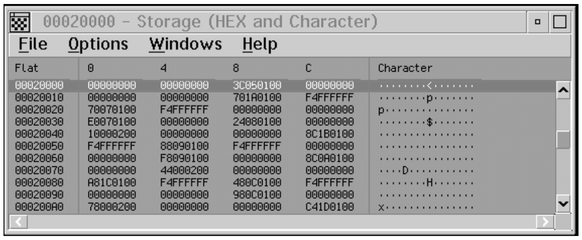
Help Menu Choices

Select choices from the **Help** menu of the **Registers** window to display the various types of help information.

📖 Refer to “Help Menu Choices” on page 432 for a description of the help choices.

Using the Storage Window

The **Storage** window, as shown in Figure 104, shows the storage contents and the address of the storage.



```
Flat      0         4         8         C         Character
00020000  00000000  00000000  3C050100  00000000  ········<·······
00020010  00000000  00000000  701A0100  F4FFFFFF  ········p·······
00020020  70070100  F4FFFFFF  00000000  00000000  p···············
00020030  E0070100  00000000  24080100  00000000  ········$·······
00020040  10000200  00000000  00000000  8C1B0100  ················
00020050  F4FFFFFF  88090100  F4FFFFFF  00000000  ················
00020060  00000000  F8090100  00000000  8C0A0100  ················
00020070  00000000  44000200  00000000  00000000  ····D···········
00020080  A81C0100  F4FFFFFF  480C0100  F4FFFFFF  ········H·······
00020090  00000000  00000000  980C0100  00000000  ················
000200A0  78000200  00000000  00000000  C41D0100  x···············
```

Figure 104. Storage Window

Multiple storage windows can display the same storage. When you run a program or update displayed data, the **Storage** window is updated to reflect the change.

To update the storage contents and all affected windows, type over the contents of the field in the **Storage** window.

To specify a new address location, type over the address field in the **Storage** window. The window scrolls to the appropriate storage location.

To display the **Storage** window, select **Storage** from the **Monitors** menu or select the **Storage** button from the title bar ▦ .

File Menu Choice

Use the choice from the **File** menu to end the debugging session.

The **Close debugger** choice allows you to end the current debugging session. When you select **Close debugger**, the **Close Debugger** message box prompts you to confirm that you want to end the debugging session.

Options Menu Choices

Use choices from the **Options** menu to monitor expressions, control how the items in the storage window display, and select the font you want for the **Storage** window.

Monitor expression...: Displays the **Monitor Expression in Storage** window, as shown in Figure 105, which allows you to type in the name of the expression you want to monitor.

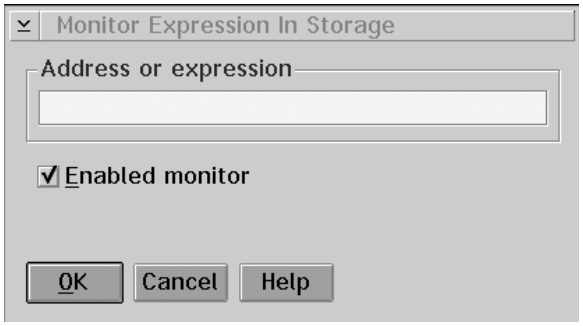


Figure 105. Monitor Expression in Storage Window

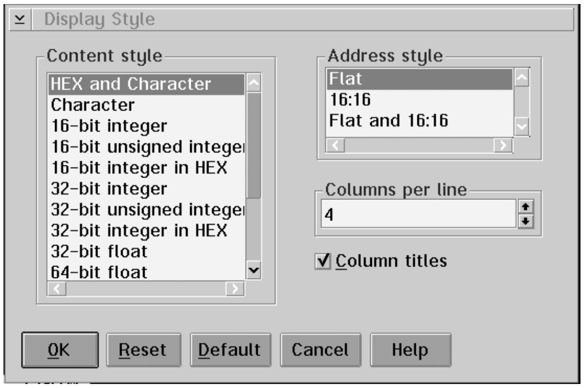
To specify an expression, type the name or address of the variable or expression you want to monitor in the **Address or expression** entry field.

The expression evaluator used is based on the context. For example if you display the **Storage** window by selecting the **Monitor expression...** choice from the **Monitors** menu, the evaluator used is based on the context in the **Monitor Expression** window. However, if you display the **Storage** window first and then select the **Monitor expression...** choice from the **Options** menu of the **Storage** window, the evaluator used is based on the context of the stopping thread.

Note: You cannot look at variables that have been defined using the DEFINE preprocessor directive. If the variable is not in scope when the monitor is opened, the default address is displayed. If the variable goes out of scope, the address is changed to a hex constant.

If you enable the **Enabled monitor** check box, the monitor updates the stop value of the program to the actual value in storage. However, a disabled monitor suspends this updating and reflects the stop value or the value held when the monitor was disabled.

Display style...: Displays the **Display Style** window, shown in Figure 106, which allows you to select the format for the storage contents and storage addresses and change the columns per line that display.



Content style: HEX and Character

Address style: Flat

Columns per line: 4

☑ Column titles

Figure 106. Display Style Window

Use the **Storage Display Style** window to select the parameters that control how the storage contents display and set how the storage addresses display.

Content style Group Heading

Select how you want the storage contents displayed. You can select from several storage display styles.

To select the storage content style:

1. Scroll to the content style you want.
2. Select the content style.
3. The style becomes highlighted.

Address style Group Heading

Select how you want the address style displayed.

To select an address style:

1. Scroll to the address style you want.
2. Select the address style.
3. The address style becomes highlighted.

Columns per line Entry Field

Select the number of columns per line you want displayed in the **Storage** window.

Use the Up or Down arrow keys to select the number of columns you want displayed in the **Storage** window. The available number of columns per line are 1-16.

Enable the **Column titles** check box if you want to display the titles of the columns in the **Storage** window.

Fonts...: Displays the **Font Selection** window, which allows you to select the type of font you want to use for the **Storage** window.

Windows Menu Choices

Select the **Windows** menu from the **Storage** window to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

Help Menu Choices

Select choices from the **Help** menu of the **Storage** window to display the various types of help information.

Refer to "Help Menu Choices" on page 432 for a description of the help choices.

Using the Local Variables Window

The **Local Variables** window, as shown in Figure 107, monitors the local variables (static, automatic, and parameters) for the current execution point in the program. The contents of the **Local Variables** window change each time your program enters or leaves a function.

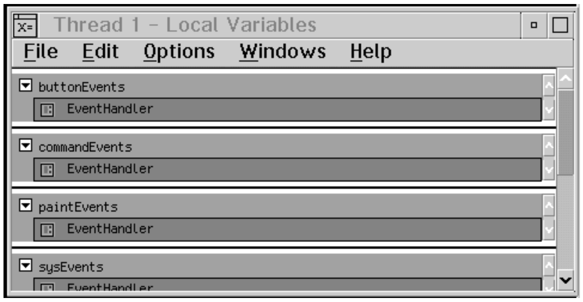


Figure 107. Local Variables Window

File Menu Choice

Use the choice from the **File** menu to end the debugging session.

The **Close debugger** choice allows you to end the current debugging session. When you select **Close debugger**, the **Close Debugger** message box prompts you to confirm that you want to end the debugging session.

Edit Menu Choices

Select the choices from the **Edit** menu of the **Local Variables** window to delete, select, or deselect variables.

Delete: Select the **Delete** choice to delete variables or expressions that are being monitored from a monitor window.

To delete a variable or expression from a monitor window:

1. Select the variable or expression using your mouse pointer. The variable or expression becomes highlighted.
2. Select the **Delete** choice from the **Options** menu.

Select all: Select the **Select all** choice to select all the expressions in the window.

Deselect all: Select the **Deselect all** choice to cancel the selection of all the expressions in the window,

Options Menu Choices

Select choices from the **Options** menu to control how the contents of variables display and to set debugger options.

Representaion→: Use the **Representation** cascading choices to display the contents of the variable in a new representation. The types of representation that display on the menu depend on the data type of the variable you are monitoring.

The following are possible representations:

Hexadecimal
Displays the contents of the monitored variable in hexadecimal notation.

Decimal
Displays the contents of the monitored variable in decimal notation.

String
Displays the contents of the monitored variable as a character string.

Hexadecimal pointer
Displays the contents of the monitored variable as a hexadecimal pointer.

Decimal pointer
Displays the contents of the monitored variable as a decimal pointer.

Array
Displays the contents of the monitored variable as an array.

Floating point
Displays the contents of the monitored variable in floating-point notation.

Character
Displays the contents of the monitored variable in character form.

Note: Floating point registers or variables display as either a floating-point decimal number or as a hexadecimal string. However, they cannot be updated with a hexadecimal string that represents a floating-point number. If you need to update a floating-point variable with a hexadecimal representation of a floating-point number, you must step through the **Disassembly** window to see when the variable loads into a register and then change the value in the **Registers** window.

Show context: Select the **Show context** choice to display the contextual information for the variable you are monitoring. The following information displays:

- Source
- File
- Line
- Thread.

Hide context: Select the **Hide context** choice to hide the contextual information for the variable you are monitoring.

Fonts...: Displays the **Font Selection** window, which allows you to select the type of font you want to use for the **Local Variables** window.

Windows Menu Choices

Select the **Windows** menu from the **Local Variables** window to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

Help Menu Choices

Select choices from the **Help** menu of the **Local Variables** window to display the various types of help information.

🕮 Refer to “Help Menu Choices” on page 432 for a description of the help choices.

Using the Monitor Windows

The debugger has three other windows that allow you to monitor variables and expressions. These windows are as follows:

- **Data Popup**
- **Program Monitor**
- **Private Monitor**.

A **Data Popup** window monitors single variables or expressions. This window is associated with a specific source window and closes when the associated window closes.

The variables or expressions can be transferred either to the **Program Monitor** window or the **Private Monitor** window.

The **Program Monitor** and the **Private Monitor** windows are used as collectors for individual variables or expressions you might be interested in. Variables and expressions may be created in these monitors or may be transferred to them from a **Data Popup** window.

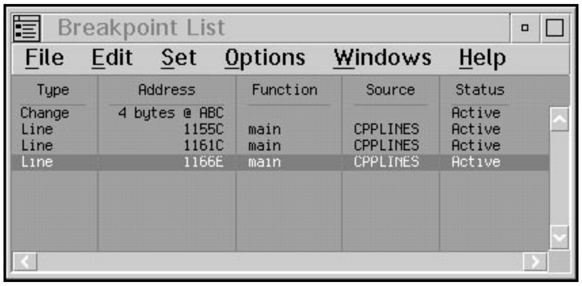
The difference between the **Private Monitor** window and the **Program Monitor** window is the length of time that they remain open. The **Program Monitor** window remains open for the entire debugging session. The **Private Monitor** window is associated with the source window from which it was opened and closes when its associated view is closed.

Using the Breakpoint List Window

Use the **Breakpoint List** window to display a list of the breakpoints that have been set. The following information is provided for each breakpoint that has been set:

- The enablement state
- The type of breakpoint
- The position of the breakpoint
- The conditions under which the breakpoint is activated.

To display the **Breakpoint List** window, as shown in Figure 108 on page 454, select **List** from the **Breakpoints** menu or select the **Breakpoints** button in the title bar ☰.



Type	Address	Function	Source	Status
Change	4 bytes @ ABC			Active
Line	1155C	main	CPPLINES	Active
Line	1161C	main	CPPLINES	Active
Line	1166E	main	CPPLINES	Active

Figure 108. Breakpoint List Window

File Menu Choice

Use the choice from the **File** menu to end the debugging session.

The **Close debugger** choice allows you to end the current debugging session. When you select **Close debugger**, the **Close Debugger** message box prompts you to confirm that you want to end the debugging session.

Edit Menu Choices

Select the choices from the **Edit** menu of the **Breakpoint List** window to delete, disable, modify, or delete breakpoints.

Delete: Deletes any breakpoints that are highlighted in the **Breakpoint List** window.

To delete a breakpoint:

1. Highlight the breakpoint you want to delete.
2. Select the **Delete** choice.

Disable: Disables any highlighted breakpoints. The breakpoint remains set but not active. This allows you to run your program and not stop when the breakpoint is encountered.

To disable a breakpoint:

1. Highlight the breakpoint you want to disable.
2. Select the **Disable** choice.

Modify: Use the **Modify** choice to change the breakpoints that have been set in your program.

To modify a breakpoint:

1. Highlight the breakpoint you want to change.
2. Select the **Modify** choice. The breakpoint window that represents the type of breakpoint displays.
3. Make the appropriate changes to the entry fields.
4. Select the **OK** push button to accept your changes and close the window. If you want to make other changes, select the **Set** push button to accept the changes and keep the window open.

Delete all: Deletes all the breakpoints that have been set.

To delete all the breakpoint:

1. Select the **Delete all** choice. The **Delete All Breakpoints** window displays.
2. Select **Yes** from the **Delete All Breakpoints** window.

Set Menu Choices

📖 Refer to “Breakpoints Menu Choices” on page 413 for a description of the **Set** menu choices.

Options Menu Choices

Select choices from the **Options** menu of the **Breakpoint List** window to sort the breakpoints, change the style, and change the font for the window.

Sort...: Displays the **Sort Breakpoints** window, shown in Figure 109, which allows you to sort the breakpoints by the characteristics of the breakpoint.

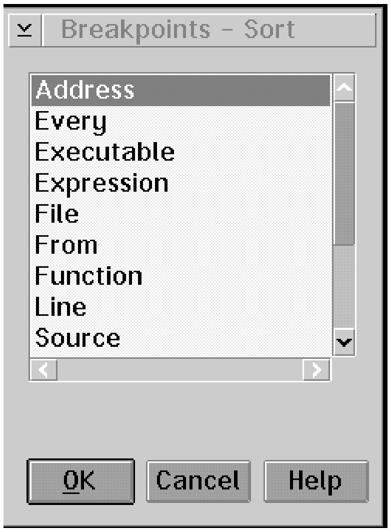


Figure 109. Sort Breakpoints Window

Use the **Sort Breakpoints** window to sort the breakpoints that have been set in your program.

Breakpoints can be sorted according to the following categories:

- Type
- Executable
- Source
- File
- Function
- Line number
- Address
- Status
- Thread
- Condition
- From
- To
- Every.

Select the category you want and select the **OK** push button.

Display style...: Displays the **Display Style** window, which allows you to control how the items in the breakpoint list display.

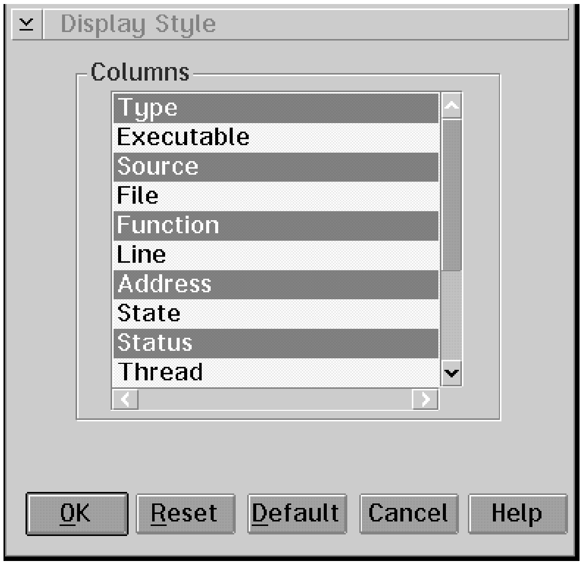


Figure 110. Breakpoints - Display Style Window

To change how the items in the breakpoint list display:

1. Select one or more of the items under the **Columns** group heading. Each item you select causes a new column to be added to the **Breakpoint List** window.
2. Select the **OK** push button.

Fonts...: Displays the **Font Selection** window that allows you to select the font you want to use for the text in the **Breakpoint List** window.

Windows Menu Choices

Select the **Windows** menu from the **Breakpoint List** window to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

Help Menu Choices

Select choices from the **Help** menu of the **Breakpoint List** window to display the various types of help information.

📖 Refer to “Help Menu Choices” on page 432 for a description of the help choices.

28

Introducing the PM Debugging Windows

Because the debugger runs in the OS/2 environment, and more specifically the PM environment, it offers some windows that allow you to debug programs written for PM.

Using the Window Analysis Window

Window analysis provides you with an understanding of PM application windows. It presents both graphical and textual information about the windows of your application and lets you observe the relationships between windows.

It allows you to view a three-dimensional image of your application's windows, characteristics of windows, and parent-child relationships between the windows.

Your application creates many windows, directly or indirectly, to perform tasks. All of these windows are children or descendents of desktop and desktop-object windows. Windows created by the application being debugged are referred to as *debuggee windows*.

Window analysis consists of the following windows:

- The **Window Analysis** window, which is the primary window.
- The **Parent and Z-Order Tree** window, which is a secondary window.
- The **Window Characteristics** window, which is a secondary window.

The secondary windows provide information pertaining to the **Window Analysis** window. When you select an item in any of the three windows, it is reflected in the other two windows. If you close the **Window Analysis** window, the secondary windows closes.

The **Window Analysis** window presents an image of your debuggee windows. When this image displays, you can rotate the image to visually separate the windows, select a window on the image, and look at the detailed information pertaining to that window.

The **Window Analysis** window is represented in a notebook format. The notebook is divided in two sections; desktop and desktop-object. The notebook has major and minor tabs. Major tabs correspond to the two major sections that are located at the bottom of the notebook. Major tab pages define the beginning of major sections and are called primary pages. Pages within major sections are called *regular pages*. They have minor tabs that are located at the right of the notebook.

Introducing the Presentation Manager (PM) Debugging Windows

The images on the desktop and desktop-object primary pages represent children of the desktop window and desktop-object window, respectively. The images on the regular pages represent the child of the desktop or desktop-object window as a parent and all its descendents.

To display the pages and tabs, you can use the notebook standard keyboard selection technique. Another method of displaying the pages is to double-click on the window on the primary page.

You can select any window on a page by selecting the window. By using the Tab and BackTab keys, you can move the selection from window to another window. When you select an item in the **Window Analysis** window, it is reflected in the two secondary windows.

Each page has a status line that is used to display sizes of a window that is selected on the page. Use the vertical slider or vertical arrows to rotate the page image vertically (around the x-axis). Use the horizontal slider to rotate the image horizontally (around the y-axis).

The borders of the windows are drawn as follows:

Screen	Solid thick line (only on a primary page)
Visible window	Solid thin line
Invisible window	Dashed thin line

File Menu Choice

Use the choice from the **File** menu to end the debugging session.

The **Close debugger** choice allows you to end the current debugging session. When you select **Close debugger**, the **Close Debugger** message box prompts you to confirm that you want to end the debugging session.

Monitors Menu Choices

Select choices from the **Monitors** menu of the **Window Analysis** window to view the secondary windows of **Window Analysis**.

Parent and z-order tree: Displays the **Parent and Z-order Tree** window which allows you to see the relationships between the following windws:

- Debuggee
- Non-debuggee
- Debuggee and non-debuggee.

The non-debuggee windows shown in the **Parent and Z-Order Tree** window are the desktop and desktop-object windows and their children which are not debuggee windows.

Window characteristics: Displays the **Window Characteristics** window which allows you to display the characteristics of the debuggee windows.

The **Window Characteristics** window displays in a table format with each row representing a different debuggee window and each column representing a different characteristic. The rows listed in the **Window Characteristics** window reflect the debuggee windows on the current page of the **Window Analysis** window.

Options Menu Choices

Select choices from the **Options** menu of the **Window Analysis** window to control the display of bitmaps and desktop-object windows and rotate the image to the center or to the default position.

Bitmaps: Enable the **Bitmaps** choice to display bitmaps on the tabs of the notebook in the **Window Analysis** window. When bitmaps are enabled, a bitmap displays on the tab if the window is shown on the screen. If the window is hidden, text displays on the tab. (For desktop-object section, this is always the case). The text displays on the tabs when bitmaps are disabled. The text is the letter “D” (Desktop) or letter “O” (Desktop-object) concatenated with the number of a regular page in the section.

Desktop-object windows: You can enable or disable the **Desktop-object windows** choice to include the desktop-object windows in the **Window Analysis** window.

Rotate to center: Select the **Rotate to center** choice to center the image on the **Window Analysis** window.

Rotate to default: Select the **Rotate to default** choice to rotate the image to the default position in the **Window Analysis** window

Windows Menu Choices

Select the **Windows** menu to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

Help Menu Choices

Select choices from the **Help** menu to display the various types of help information.

Using the Message Queue Window

The **Message Queue** window, as shown in Figure 111, displays PM messages associated with a PM application. It presents formatted messages in a list as they occur. Using the **Message Queue** window, you can control:

- How the information displays for each message.
- How message parameters are formatted.
- Which messages are monitored.
- Which windows have their messages monitored.
- Which message queues have their messages monitored.
- How the user generated messages display.

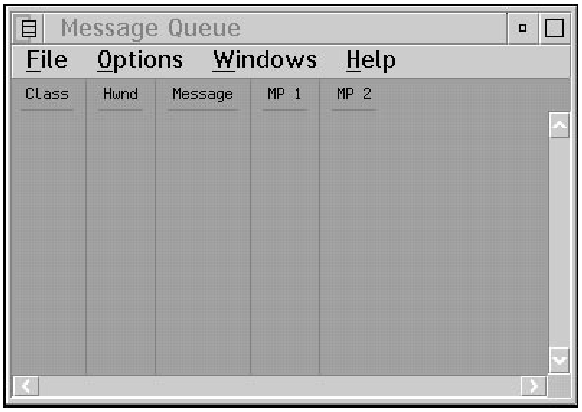


Figure 111. Message Queue Window

File Menu Choice

Select the choice from the **File** menu of the **Message Queue** window to end a debugging session.

Options Menu Choices

Select choices from the **Options** menu to suspend messages, clear messages, resize the columns, and select what you want to monitor.

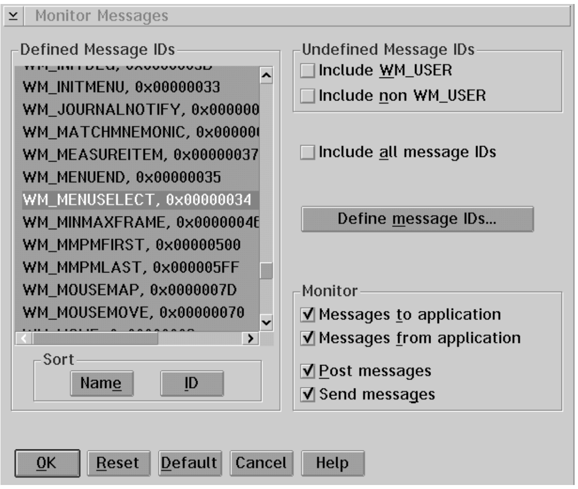
Suspend: Select the **Suspend** choice to stop any new messages from being added to the **Message Queue** window.

Clear: Select the **Clear** choice to clear all the messages in the **Message Queue** window.

Resize column width: Select the **Resize column width** choice to recalculate the widths of the columns in the **Message Queue** window.

Note: This option is available only when the **automatic column resizing** choice is not selected in the **Display Style** window.

Monitor messages...: Displays the **Monitor Messages** window, shown in Figure 112, which allows you to specify the messages you want monitored.



Monitor Messages

Defined Message IDs

WM_INITMENU, 0x00000033
WM_JOURNALNOTIFY, 0x000000
WM_MATCHMNEMONIC, 0x00000
WM_MEASUREITEM, 0x00000037
WM_MENUEND, 0x00000035
WM_MENUSELECT, 0x00000034
WM_MINMAXFRAME, 0x0000004
WM_MMPMFIRST, 0x00000500
WM_MMPMLAST, 0x000005FF
WM_MOUSEMAP, 0x0000007D
WM_MOUSEMOVE, 0x00000070

Undefined Message IDs

☐ Include WM_USER
☐ Include non WM_USER

☐ Include all message IDs

Monitor

☑ Messages to application
☑ Messages from application
☑ Post messages
☑ Send messages

Figure 112. Monitor Messages Window

The **Monitor Messages** window consists of the following:

Defined Message IDs Group Heading

These are the pre-defined and user-defined messages. Each message displays as a name and hex number. You can select multiple messages for monitoring.

Sort Group Heading

- Select the **Name** button to sort the message names alphabetically.
- Select the **ID** button to sort the message IDs numerically.

Undefined Messages IDs Group Heading

- Select the **Include WM_USER** check box to include all undefined messages that are in the range WM_USER and above.
- Select the **Include non WM_USER** check box to include all undefined messages that are in the range less than WM_USER.

Include all message IDs Check Box

Select the **Include all message IDs** check box to include all messages. When this check box is enabled, the defined and undefined message selections are disabled.

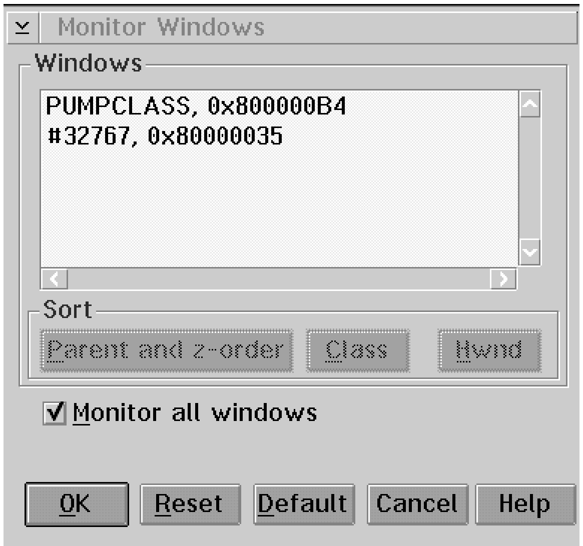
Define Messages IDs Push Button

Select this button and the **Define Messages** window displays.

Monitor Group Heading

- Select the **Messages to application** check box to monitor messages that are received by an application.
- Select the **Messages from application** check box to monitor messages that are dispatched by an application.
- Select the **Post messages** check box to monitor post messages.
- Select the **Send messages** check box to monitor send messages.

Monitor windows...: Displays the **Monitor Windows** window, shown in Figure 113, which allows you to specify the windows you want monitored.



PUMPCLASS, 0x800000B4
#32767, 0x80000035

☑ Monitor all windows

Figure 113. Monitor Windows Window

You may identify specific windows you want to monitor by using the **Window Analysis** window.

The **Monitor Windows** window consists of the following:

Windows Group Heading

Each window displays with the class and handle. You can select multiple windows for monitoring.

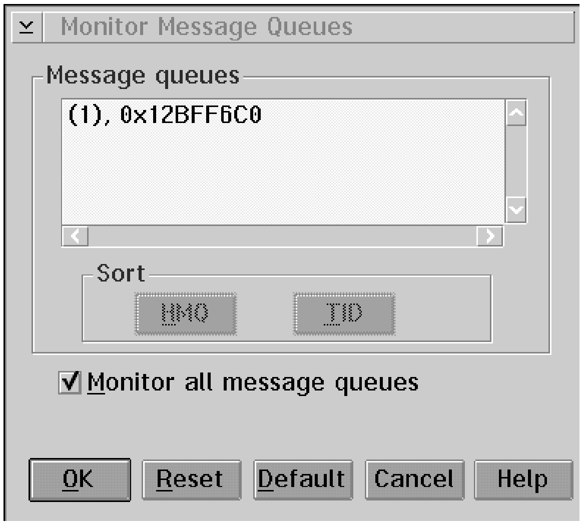
Sort Group Heading

- Select the **Class** button to sort the window names alphabetically.
- Select the **Hwnd** button to sort the window handles in numerical order.
- Select the **Parent and z-order** button to sort the window handles to show the parent and z-order relationship of application windows.

Monitor all windows Check Box

Select the **Monitor all windows** check box to monitor all the windows. This disables individual selection.

Monitor queues...: Displays the **Monitor Queues** window, shown in Figure 114, which allows you to specify the message queues you want monitored.



Monitor Message Queues

Message queues

(1), 0x12BFF6C0

Sort

☑ Monitor all message queues

Figure 114. Monitor Queues Window

The **Monitor Message Queues** window consists of the following:

Message queues Group Heading

This section contains a list of the message queues. You can select multiple queues.

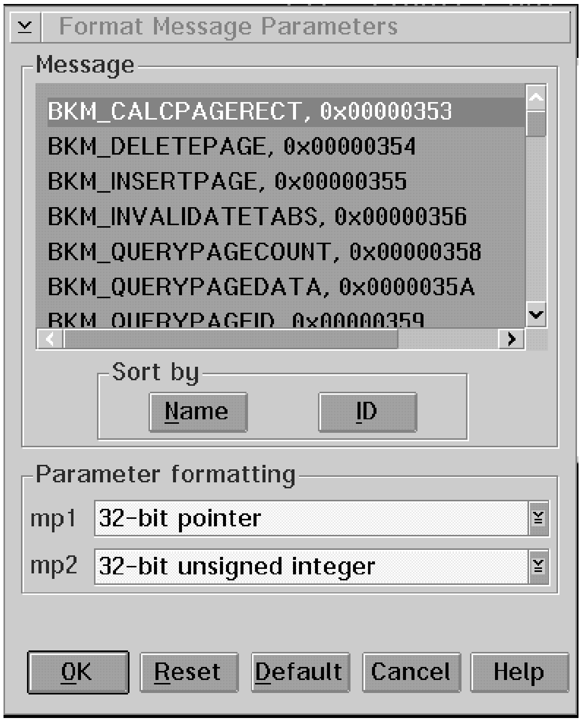
Sort Group Heading

- Select the **HMQ** button to sort the message queues by message queue handles.
- Select the **TID** button to sort the message queues by thread id.

Monitor all message queues Check Box

Select the **Monitor all message queues** check box to monitor all the message queues. This disables individual selections.

Format parameters...: Displays the **Format Parameters** window, shown in Figure 115, which allows you to specify how message parameters display.



BKM_CALCPAGERECT, 0x00000353
BKM_DELETEPAGE, 0x00000354
BKM_INSERTPAGE, 0x00000355
BKM_INVALIDATETABS, 0x00000356
BKM_QUERYPAGECOUNT, 0x00000358
BKM_QUERYPAGEDATA, 0x0000035A
BKM_QUERYPAGEID, 0x00000359

mp1 32-bit pointer
mp2 32-bit unsigned integer

Figure 115. Format Parameters Window

The **Format Message Parameters** window consists of the following:

Message Group Heading

This section contains a list of the all the defined messages.

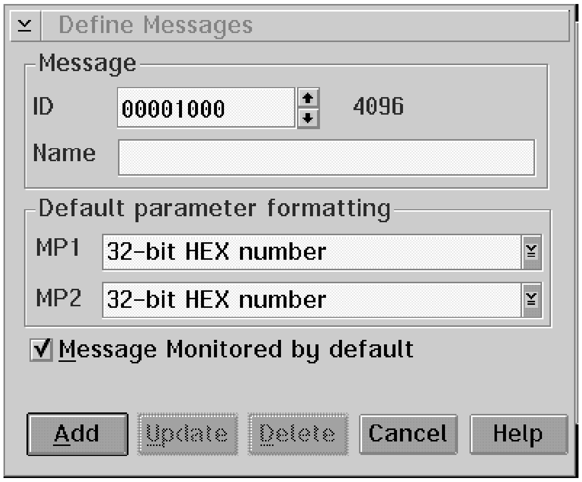
Sort by Group Heading

- Select the **Name** button to sort the messages alphabetically by name.
- Select the **ID** button to sort the messages numerically by ID.

Parameter formatting Group Heading

- MP1 - Message parameter 1 Select the arrow to open the **MP1** list. Select the type of formatting you want from the parameter list. The formatting selections are saved in program profiles.
- MP2 - Message parameter 2 Select the arrow to open the **MP2** list. Select the type of formatting you want from the parameter list. The formatting selections are saved in program profiles.

Define messages...: Displays the **Define Messages** window, shown in Figure 116, which allows you to define messages.



Define Messages

Message

ID 00001000 4096

Name

Default parameter formatting

MP1 32-bit HEX number

MP2 32-bit HEX number

☑ Message Monitored by default

Figure 116. Define Messages Window

Message Group Heading

In the **Name** entry field, type in the name of the message that you want to define.

In the **ID** entry field, type the hex number for the message or use the spin button to locate the number of the message.

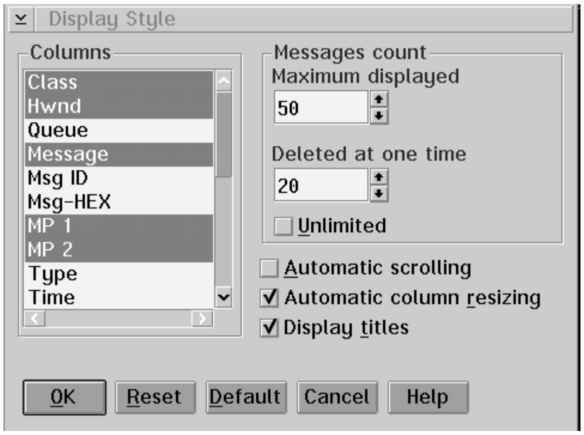
Default parameter formatting Group Heading

- MP1 - Message parameter 1 Select the arrow to open the **MP1** list. Select the default type of formatting you want from the parameter list. The formatting selections are saved in program profiles.
- MP2 - Message parameter 2 Select the arrow to open the **MP2** list. Select the default type of formatting you want from the parameter list. The formatting selections are saved in program profiles.

Message monitored by default Check Box

Select the **Message monitored by default** check box if you want to monitor the messages by default.

Display style...: Displays the **Display Style** window, shown in Figure 117, which allows you to specify which columns you want displayed in the **Message Queue** window.



Display Style

Columns

Class
Hwnd
Queue
Message
Msg ID
Msg-HEX
MP 1
MP 2
Type
Time

Messages count

Maximum displayed 50

Deleted at one time 20

□ Unlimited

□ Automatic scrolling

☑ Automatic column resizing

☑ Display titles

Figure 117. Display Style Window

The items in this window affect system performance. You may want to experiment with different settings to see what works best for your system.

Columns Group Heading

Select the columns you want to display in the **Message Queue Monitor** window. These are described in the **column list**.

Messages count Group Heading

The following choices affect system performance:

Maximum displayed Entry Field

Select the number of messages you want to display from the **Maximum displayed** entry field. You can type in a number or use the spin button to select the number you want. A higher maximum may cause the system to respond faster. However, too high a maximum, may cause the system may respond slower.

Note: You may need to experiment with this setting to see what works best for your system.

Deleted at one time Entry Field

Select the number of messages you want deleted at one time from the **Deleted at one time** entry field. You can type in a number or use the spin button to enter the number you want. The more messages deleted at once causes the system to respond faster.

Unlimited Check Box

Select **Unlimited** to have no maximum on the number of messages to be displayed. When you select **unlimited**, the **Maximum displayed** and **Deleted at one time** entry fields are disabled. This choice sets no limit on messages to have displayed and the system may respond slower.

Automatic scrolling Check Box

Select **Automatic scrolling** if you want the window to automatically scroll forward when the screen is filled with messages. Selecting this decreases system performance.

Automatic column resizing Check Box

Select **Automatic column resizing** if you want the columns to automatically resize. Selecting this decreases system performance.

Note: When this check box is selected, the **Resize column width** choice is not available.

Display titles Check Box

Select **Display titles** if you want titles to be displayed in the window. This does not affect system performance.

Windows Menu Choices

Select the **Windows** menu to view a list of all the open debugger windows. By selecting a window from the **Windows** menu, it is brought into focus and made the active window. Also, if the window is minimized, it is restored.

Help Menu Choices

Select choices from the **Help** menu to display the various types of help information.

🕮 Refer to “Help Menu Choices” on page 432 for a description of the help choices.

Expressions Supported

This section describes the expression language supported by the debugger, which is a subset of C/C++. This includes the operands, operators, and data types.

Note: You can display and update bit fields for C/C++ code only. You cannot look at variables that have been defined using the `define` preprocessor directive.

Supported Expression Operands

You can monitor an expression that uses the following types of operands only:

Operand — **Definition**

Variable A variable used in your program.

Constant The constant can be one of the following types:

- Fixed or floating-point constant.

  Note: The largest floating-point constant is 1.8E308. The smallest floating-point is 2.23E-308.
- A string constant, enclosed in quotation marks (“ ”)
- A character constant, enclosed in single quote marks (’ ’)
- Segment:Offset address specification (0000:0000)

  When you are specifying a segment offset address for monitoring in a variable monitor window, specify the offset address in the following format:

  ```
  0x0000:0
  ```

Registers One of the following register names:
AX, BX, CX, DX, SP, BP, SI, DI, AL, BL, CL, DL, AH, BH, CH, DH, EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP, CS, DS, ES, FS, GS, SS, EFLAGS.
One of the following floating-point register names:
ST0 through ST7, FPCW, FPSW, FPTW, FPEIP, FPCS, FPEDP, FPDS.

In the case of conflicting names, the program variable names take precedence over the register names. For conversions that are done automatically when the registers display in mixed-mode expressions, general purpose registers are treated as unsigned arithmetic items with a length appropriate to the register. For example, EAX is 32-bits, AX is 16-bits, and AL is 8-bits.

If you monitor an enumerated variable, a comment displays to the right of the value. If the value of the variable matches one of the enumerated types, the comment contains the name of the first enumerated type that matches the value of the variable. If the length of the enumerated name does not fit in the monitor, the contents display as an empty entry field.

The comment (empty or not) lets you distinguish between a valid enumerated value and an invalid value. An invalid value does not have a comment to the right of the value.

You can *not* update an enumerated variable by entering an enumerated type. You must enter a value or expression. If the value is a valid enumerated value, the comment to the right of the value updates.

Bit fields are supported for C/C++ compiled code only. You can display and update bit fields, but you cannot use them in expressions. You cannot look at variables that have been defined using the DEFINE preprocessor directive.

Supported Expression Operators

You can monitor an expression that uses the following operators only:

Figure 118 (Page 1 of 2). Supported Expression Operators

Operator	Coded as
Global scope resolution	::*a*
Class scope resolution	*a*::*b*
Subscripting	*a*[b]
Member selection	*a.b* or *a->b*
Size	*sizeof a* or *sizeof (type)*
Logical not	!*a*
One's complement	˜*a*
Unary minus	-*a*
Unary plus	+*a*
Dereference	**a*
Type cast	(*type*) *a*
Multiply	*a* * *b*
Divide	*a* / *b*

Figure 118 (Page 2 of 2). Supported Expression Operators

Operator	**Coded as**
Modulo	*a* % *b*
Add	*a* + *b*
Subtract	*a* - *b*
Left shift	*a* << *b*
Right shift	*a* >> *b*
Less than	*a* < *b*
Greater than	*a* > *b*
Less than or equal to	*a* <= *b*
Greater than or equal to	*a* >= *b*
Equal	*a* == *b*
Not equal	*a* != *b*
Bitwise AND	*a* & *b*
Bitwise OR	*a* | *b*
Bitwise exclusive OR	*a* ^ *b*
Logical AND	*a* && *b*
Logical OR	*a* || *b*

Supported Data Types

You can monitor an expression that uses the following typecasting operations:

- 8-bit signed byte
- 8-bit unsigned byte
- 16-bit signed integer
- 16-bit unsigned integer
- 32-bit signed integer
- 32-bit unsigned integer
- 32-bit floating-point
- 64-bit floating-point
- 80-bit floating-point
- Pointers
- User-defined types.

Expressions Supported

Part 7. Performance Execution Trace Analyzer

This part of the *User's Guide* describes the VisualAge C++ Performance Analyzer, which you can use to help you understand and improve the performance of your programs.

Note: Additional information is available in the Performance Analyzer online Help facility. To access this information, select **General help** from the **Help** menu on any Performance Analyzer window. After the Help facility starts, you can view the table of contents by selecting **Contents** from the **Options** menu.

For an index of Performance Analyzer topics, select **Help index** from the **Help** menu on any Performance Analyzer window.

You may find that the hypertext links in the online Help are easier to use.

30

Introducing the Performance Analyzer

The IBM VisualAge C++ for OS/2, V3.0 Performance Analyzer is an application that helps you understand and improve the behavior of IBM C and C++ programs.

The Performance Analyzer traces the execution of a program and creates a trace file. The trace file contains trace analysis data that can be displayed in diagrams. Using these diagrams, you can improve program performance, examine occurrences that produce faults, and in general, understand what happens when a program runs.

The Performance Analyzer does not replace static analyzers or debuggers, but it can complement them by helping you understand aspects of the program that would otherwise be difficult or impossible to see.

For instance, with the Performance Analyzer you can:

Time and tune programs

The Performance Analyzer time stamps each trace event using a high resolution clock (about 838 nanoseconds per clock tick). As a result, the trace file contains a detailed record of when each traced function was called and when it returned.

The trace data also shows how long each function runs, which helps you find hot spots.

Locate program hangs and deadlocks

The Performance Analyzer provides a complete history of events leading up to the point where a program stops. You can view the function call stack from anywhere in the program.

Trace multithreaded interactions

When multithreaded programs are traced, you can look at the sequencing of functions across threads in some of the diagrams. This highlights problems within critical areas of the program.

New and Enhanced Features

Performance Analyzer - Window Manager window

The **Performance Analyzer - Window Manager** window is the control window for the Performance Analyzer. From this window, you can start most Performance Analyzer functions. For instance, you can:

- Start creating a new trace file
- Start analyzing an existing trace file
- Open and close a diagram.

Trace On and Trace Off push buttons
These buttons, which appear on the **Application Monitor** window, let you start and stop the trace of your program.

WorkFrame integration
If you have started the Performance Analyzer from WorkFrame, you can:

- Get index Help for functions other than those that are part of your product.
- Start the WorkFrame editor from a Performance Analyzer diagram and edit your source code.
- Start other programs from the Performance Analyzer.

Tracing capability in dynamic link libraries
In addition to tracing functions in the executable file, the Performance Analyzer can trace your program's activity in:

- Statically or dynamically linked dynamic link libraries (DLLs).
- The following system libraries:
  - DOSCALL.DLL
  - PMGPI.DLL
  - PMWIN.DLL
- Dynamically linked load-on-call DLLs. If you only want to trace a load-on-call DLL, the **Trace Generation** window will not have any executables or DLLs listed in the window, and you will receive an informational message.

64 threads
The Performance Analyzer can trace up to 64 threads. The diagrams show activity on all or selected threads.

Pop-up menus
Clicking mouse button two in most diagrams displays pop-up menus that let you quickly access frequently used functions.

Time find capability in Call Nesting
The **Call Nesting** diagram has a search capability that lets you go to specific times in the trace file.

Time Line diagram
The Performance Analyzer can display user events in the **Time Line** diagram.

Status area
Each diagram has a status area, which shows you detailed information about the trace file data.

Vertical ruler
Many diagrams have a Vertical Ruler that shows your location in the trace file.

31 Preparing Your Program

Before you create a trace file and begin using the Performance Execution Trace Analyzer, you ***must*** compile and link your program with the proper options. This is described in "Compiling and Linking Your Program."

The Performance Analyzer provides several ways to customize trace files. If you want to customize a trace file, you may have to complete some steps *before you compile and link your program.* For more information and instructions, see the following topics:

- "Tracing Dynamic Link Libraries (DLLs)" on page 480
- "Tracing System Calls" on page 480
- "Creating User Events in Your Program" on page 481
- "Starting and Stopping the Performance Analyzer from Your Program" on page 482.

Compiling and Linking Your Program

You ***must*** compile and link your program with the proper options before you create a trace file and analyze it with the Performance Analyzer.

Compiling

When compiling your program, use the following options:

/Gh Includes the profile hooks that allow the Performance Analyzer to monitor your executable.

/Ti Includes debugging information in the compiled object file.

Linking

You must link the CPPOPA3.OBJ object file into your program.

When linking your program, use the following options:

/DE Instructs the linker to include debug information in the executable (EXE) or dynamic link library (DLL) file.

/NOE Instructs the linker not to search for symbols in the extended dictionaries of the libraries being linked.

Preparing Your Program for the Performance Analyzer

The following example shows how to compile and link a program called *SAMPLE.EXE* for use with the Performance Analyzer. The required object file is highlighted in bold.

Compile:

```
icc /c /Ti /Gh sample.c
```

Link:

```
icc /b"/DE /NOE" /FeSAMPLE.EXE sample.obj cppopa3.obj
```

Tracing Dynamic Link Libraries (DLLs)

You can trace statically or dynamically linked DLLs and load-on-call DLLs using the Performance Analyzer. Compile and link any DLL to be traced. For instructions, see “Compiling and Linking Your Program” on page 479.

You can trace DLLs without tracing the main program that calls the DLLs.

The tracing of load-on-call DLLs occurs automatically; it cannot be suppressed. Also, you cannot set triggers, or enable or disable functions in dynamically linked DLLs.

Tracing System Calls

If you want to trace calls into the OS/2 Toolkit Application Programming Interface (API), specify the following Performance Analyzer libraries *before* the OS/2 libraries in your link statement. You can link one or more libraries.

If you want to trace file accesses, you must link the _DOSCALL.LIB library.

Notes:

- Output produced by *cout* stream objects is stored in the operating system’s buffer and is not shown until tracing has stopped, and DosCalls for these calls are not displayed in the trace file.
- It is not possible to trace events in the DOSCALL intercept library ***only***. The Performance Analyzer looks for at least one event from your program before logging DOS call events. If you link the _DOSCALL.LIB library, compile your program with the */Gh* option to include events from your program in the trace file.

The APIs and their corresponding Performance Analyzer libraries are as follows. Each library listed also has an associated DLL.

API	Library
DosCalls	_DOSCALL.LIB
WinCalls	_PMWIN.LIB
GpiCalls	_PMGPI.LIB

Important: The order in which these libraries are specified in the link statement is critical. If the replacement libraries do not precede the OS/2 libraries in the link statement, the Performance Analyzer will not interpret and trace the API calls.

Creating User Events in Your Program

The CPPOPA3.OBJ file contains an entry point called *PERF*, that accepts calls from the program you are tracing. Calls to the PERF entry point at run time are referred to as user events. User events cause text strings to be inserted into the trace file.

To add a call to the PERF entry point:

1. Declare a prototype for the PERF entry point.

   If you add a user event to your program source file, you must also include a prototype for the PERF entry point.

   For C and C++ programs, the prototype will be inserted for you when you compile your program if you place the following statement at the beginning of your source file:

   ```
   #include <icsperf.h>
   ```

   Note: If you want to insert the prototype yourself, the prototypes for C and C++ programs are as follows:

   C Prototype

   ```
   VOID PERF (PSZ string);
   ```

   C++ Prototype

   ```
   EXTERN "C" {VOID PERF(PSZ string);}
   ```

2. Add a call to the entry point everywhere you want a user event generated.

   The following is an example of a call to the PERF entry point:

   ```
   PERF (string);
   ```

   where:
   string is an ASCIIZ string.

When such a call is made, the string is placed in the trace file. You can see the string in the **Call Nesting**, **Statistics**, and **Time Line** diagrams.

Warning:

- The *string* must consist of unique, static, alphanumeric characters. Otherwise, you will receive unpredictable results.
- The *string* must exist in storage when your program terminates. If the *string* exists in automatic storage on the stack or storage defined in a dynamically loaded DLL, the *string* will appear in the trace file, but the text may not appear as it was defined. (The *string* can be allocated on the heap if its contents are not deleted when your program terminates.)

Starting and Stopping the Performance Analyzer from Your Program

The CPPOPA3.OBJ file contains entry points called *PerfStart* and *PerfStop* that accept calls from the program you are tracing. Calls to the PerfStart and PerfStop entry points cause the Performance Analyzer to start and stop tracing, respectively. By putting these calls into your source code, you can control precisely when the Performance Analyzer starts and stops recording events during program execution.

To turn tracing:

- On, call the PerfStart entry point.
- Off, call the PerfStop entry point.

Notes:

1. If the trace is already on, calling PerfStart has no effect. If the trace is already off, calling PerfStop has no effect.
2. You can also start and stop tracing with the **Trace on** and **Trace off** push buttons on the **Application Monitor** window.

You can place calls to the PerfStart and PerfStop entry points anywhere in your program, even in different functions, object modules, or DLLs.

To add calls to the PerfStart and PerfStop entry points:

1. Declare a prototype for the PerfStart and PerfStop entry points.

   For C and C ++ programs, the prototypes will be inserted for you when you compile your program if you place the following statement at the beginning of your source file:

   ```
   #include <icsperf.h>
   ```

   Note: If you want to insert the prototypes yourself, the prototypes for C and C++ programs are as follows:

   C Prototype

```
VOID PerfStart (VOID);
VOID PerfStop (VOID);
```

C ++ Prototype

```
EXTERN "C" {VOID PerfStart (VOID);}
EXTERN "C" {VOID PerfStop (VOID);}
```

2. Add a call to the appropriate entry point everywhere you want to start or stop tracing.

   The following example shows how calls to the PerfStart and PerfStop entry points could be placed in your program:

```
    .
    .
    .
PerfStop();  // turn off tracing here
    .
    .
    .
PerfStart();  // resume tracing here
```

Understanding Overhead Time

When you compile and link your program, the compiler generates hooks that enable the Performance Analyzer to intercept trace events. The time at which each trace event executes is recorded in the trace file.

These hooks cause a small monitoring function to be called instead of the program’s callee function. The monitoring function time stamps the event and then calls the program's callee function.

The monitoring function is run in the program’s address space, thereby avoiding the high overhead of an operating system context switch when events are recorded. As a result, it does not significantly affect the program’s runtime performance. However, the monitoring function does take some small amount of time to execute. In order to compensate for this additional time introduced by the monitoring function, the diagrams adjust the timings appropriately.

The Performance Analyzer dynamically determines how much time it takes to execute the monitoring function by internally calling it several times and computing an average prior to executing the program. As a result, it is recommended that you shut down other programs on your desktop so they will not interfere with the Performance Analyzer’s timings.

If the program is run stand-alone, events are not recorded and should run at or near the same speed as the same program compiled without the profile hooks.

Preparing Your Program for the Performance Analyzer

32 Starting the Performance Analyzer

After completing the instructions in Chapter 31, “Preparing Your Program” on page 479, you are ready to create a trace file and use the Performance Analyzer to analyze your program.

You can start the Performance Analyzer either from OS/2, or from WorkFrame.

Starting from OS/2

The command you enter to start the Performance Analyzer depends on which of the following you want to do first:

- Trace an executable
- Analyze an *existing* trace file.
- Display the Performance Analyzer’s main control window

Tracing an Executable

- If you have an executable you want to trace, you can start the analyzer from an OS/2 command prompt or a command file by entering:

  ```
  icsperf myprog parms
  ```

  Where:

  myprog Represents an executable file name. This is optional.
  parms Represents executable parameters. These are optional.

  Note: The first time you start the Performance Analyzer, a profile window appears and prompts you to specify where the Performance Analyzer’s profile file should be located. If you want the default, press the **OK** push button. See “Performance Analyzer - Specify Profile Location Window” on page 499 for more information.

- You can also include the */go* option in the command:

  ```
  icsperf /go myprog parms
  ```

  Where:

  /go Option that executes your program, creates a trace file, and then exits the Performance Analyzer. This option is useful if you have several programs (requiring no manual intervention) that you want to run in succession from an OS/2 command file. This is optional.
  myprog Represents an executable file name.
  parms Represents executable parameters. These are optional.

Starting the Performance Analyzer

Analyzing an Existing Trace File

If you want to start analyzing a trace file you have already created, you can start the analyzer from an OS/2 command prompt or a command file by entering:

```
icsperf /x myprog.trc
```

Where:

/x Represents one or more of the following analyzer options. If you have already created a trace file, these options cause the trace file to be displayed in their respective diagrams. Once you are familiar with the Performance Analyzer application, you can quickly open the diagrams by entering as many of these options as you want in your startup command. This is optional.

- **/cn** Displays the trace file in the **Call Nesting** diagram.
- **/ed** Displays the trace file in the **Execution Density** diagram.
- **/cg** Displays the trace file in the **Dynamic Call Graph**.
- **/ss** Displays the trace file in the **Statistics** diagram.
- **/tl** Displays the trace file in the **Time Line** diagram.

myprog.trc Represents a trace file name.

Displaying the Performance Analyzer's Main Control Window

If you enter the following command, the analyzer's main control window, the **Performance Analyzer - Window Manager** window appears.

```
icsperf
```

From this window, you can start either tracing an executable or analyzing an existing trace file. 📖 See Chapter 33, "Creating Trace Files" on page 489 to continue.

Analyzing WorkPlace Shell Objects

To analyze WorkPlace Shell objects:

1. Replace the *RUNWORKPLACE* line in your config.sys file with the following line:

   ```
   SET RUNWORKPLACE=C:/OS2/CMD.EXE
   ```

2. Restart your system.
3. At an OS/2 prompt, type the following:

   ```
   icsperf c:/os2/pmshell
   ```

Starting from WorkFrame

Before you start the Performance Analyzer from the WorkFrame environment, you must:

1. Create a project for the program you want to analyze.

   Note: For information on creating a project, refer to WorkFrame documentation.

2. Compile and link your program with Performance Analyzer options.

   Note: This is 📖 described in “Compiling and Linking Your Program” on page 479.

3. Open a project folder in the WorkFrame window.
4. Highlight an object that represents an executable file or a trace file.
5. Click mouse button two on the highlighted object to display a pop-up menu.
6. Select **Analyze.**

Exiting the Performance Analyzer

If you want to exit the Performance Analyzer, and are *not* in the process of creating a trace file, do the following:

1. Select the **Exit the Performance Analyzer** choice from one of the following menus:

   - **File** menu on the **Performance Analyzer - Window Manager** window
   - **Application** menu on the **Trace Generation** window
   - **Trace file** menu on any of the diagrams.

2. Select **Yes** when prompted.

If you want to exit the Performance Analyzer while a trace file is being created, do the following:

1. Click on the **Stop** push button on the **Application Monitor** window.
2. Click on the **Cancel** push button on the **Analyze Trace** window.
3. Select the **Exit the Performance Analyzer** choice from the **File** menu on the **Performance Analyzer - Window Manager** window.
4. Select **Yes** when prompted.

Exiting the Performance Analyzer

Creating Trace Files

After compiling and linking your program, you can start the Performance Analyzer and create a *trace file.* A *trace file* contains a chronological sequence of events that occur during the execution of your program.

By analyzing the trace file, you can learn about your program's structure, locate and diagnose problems, and pinpoint ways to improve performance. The Performance Analyzer provides five diagrams in which you can analyze the trace file. Each diagram presents a different view of the trace file to give you an overall idea of how your program performs. The diagrams are as follows:

- Call Nesting
- Dynamic Call Graph
- Execution Density
- Statistics
- Time Line

Note: Before creating a trace file, you must prepare your program for use by the Performance Analyzer. For more information, see Chapter 31, “Preparing Your Program” on page 479.

To create a trace file:

1. Click on the **Create Trace** push button in the **Performance Analyzer - Window Manager** window.
2. Type the full path name and the file name of the program you want to trace in the **Program Name:** entry field. If the program is in your current directory, you do not have to type the path name.

   Note: If you are not sure where the file is located, select the **Find** push button.
3. Type any parameters that you want to pass to your program in the **Program Parameters:** entry field.

   Note: This entry field is optional.
4. If you want the trace file to have a different path and file name than the defaults, type a path and file name in the **Trace File Name:** entry field.

   The default path name is the directory where your program resides. The default trace file name is *myprog.trc*, where *myprog* is the name of the program you are tracing.

   Note: This entry field is optional.

5. Type any comments that you want to make about your trace in the **Trace File Description:** entry field.

   Note: This entry field is optional.

6. Select the **OK** push button. The **Trace Generation** window appears.

7. Select the **Trace** push button in the **Trace Generation** window.

   Your program begins executing. When your program ends, the **Analyze Trace** window is displayed.

8. Click on the check box next to each diagram in which you want to view the trace file.

Creating a Customized Trace File

By default, the Performance Analyzer generates event information for every function possible. However, this sometimes causes the trace file to become large and difficult to manage. You can limit the size of your trace file by changing the parameters that control its size prior to running your program. The following parameters affect the size of the trace file.

- Enabled or disabled state of components
- Call depth setting for each thread
- Time stamp setting
- File access setting
- Trigger settings.

Using these parameters to control the size of your trace file is explained on the following pages.

You can also customize the trace file by:

- Disabling buffer flushing or changing the buffer size setting to specify how often the buffer flushes to the trace file. For more information, 📖 see “Changing the Buffer Size” on page 494. or the online Help topic “Buffer Control Window”.
- Giving the trace file a file name other than the default. The default file name is *myprog.trc*, where *myprog* is the name of the program you are tracing. For more information on specifying a different trace file name, see the online Help topics “Name Trace File Choice” and “Unique Trace File Name Choice”.
- Attaching a description to the trace file. A description can make a trace file easier to identify, especially when you create more than one trace file from the same program and use different options for each trace. The description is displayed in the **Status Area** of any open diagram. For more information on attaching a description to a trace file, see the online Help topics “Name Trace File Choice” and “Unique Trace File Name Choice”.

Enabling and Disabling Components

Your program's components are listed on the **Trace Generation** window.

Note: A component can be an executable, a dynamic load library file, an object (OBJ) file, or a function. EXEs and DLLs contain object files, and object files contain functions. To view or hide components in the window, click on the plus/minus icon to expand and contract EXEs, DLLs, and OBJ files.

The Performance Analyzer's default is to enable all components that have been compiled and linked with the proper options. When a component is enabled, data for that component will be included in the trace file when the component is executed. When the component is disabled, no data is recorded in the trace file when the component is executed.

Trace files containing data for all enabled components can sometimes become large and difficult to manage. You can limit the amount of data collected in your trace file by selecting specific components to enable and disable from the **Edit** menu on the **Trace Generation** window.

When enabling and disabling components, remember the following:

- When you disable:
  - An EXE, DLL, or OBJ file, the Performance Analyzer disables all functions within the selected file and removes any triggers set on functions within the file.
  - A function, the Performance Analyzer removes any trigger set on it.
- When you enable:
  - An EXE, DLL, or OBJ file, the Performance Analyzer enables all functions within the selected file.
- When you set a trigger on a disabled function, the Performance Analyzer enables the function.

You can enable or disable a component in one of the following ways:

- Click on the file name or icon of the component you want to enable or disable. Then select the appropriate enable or disable choice from the **Edit** menu on the **Trace Generation** window.
- Double-click on the file name or icon of the component you want to enable or disable.
- Click mouse button two on the file name or icon of the component you want to enable or disable. Then select the appropriate enable or disable choice from the pop-up menu.

Creating a Customized Trace File

Note: If the icon next to a component is:

- Green (it also has no slash mark), the component is *enabled*.
- Red (it also has a slash mark), the component is *disabled*.
- White, the component cannot be traced.

For a description of the menu choices you can use to enable and disable components, see “Trace Generation Menu Bar Summary” on page 506 or the Performance Analyzer online Help facility.

Selecting the Call Depth for Each Thread

You may want to limit the call depth to isolate an area of interest and reduce the amount of trace data.

Select **Call depth** from the **Options** menu on the **Trace Generation** window to select the number of calls you want to trace or to specify threads you want to include or exclude from the trace file. When the **Call Depth** window is displayed, you can select as many as 64 threads with a maximum nesting depth of 128 for each thread. The default is to have all threads selected with the maximum depth of 128.

Using Time Stamps

Select **Time stamp events** from the **Options** menu on the **Trace Generation** window to choose whether to time stamp events during the trace analysis. Disabling **Time stamp events** causes your trace file to be smaller because time stamps are not stored. It does not limit the number of events collected in the trace file.

The Performance Analyzer uses an internal timer to get high resolution time stamps. While logging events, the Performance Analyzer adds a small amount of overhead time to the normal runtime speed of the program. The time added is negligible, so you may not be able to tell the difference between code that has been traced and code that has not, even for highly interactive programs.

The overhead time added by the Performance Analyzer is not shown in the times reported for user code in all of the Performance Analyzer diagrams. the Performance Analyzer time stamps the buffer flush so that the buffer-flushing overhead can be removed from the diagrams.

If you choose to create a trace file without time stamps, you can only view it in the **Dynamic Call Graph**, **Call Nesting**, and **Statistics** diagrams.

Tracing File Accesses

When a DOS call references a file, the Performance Analyzer keeps track of the file name associated with the traced DOS call. These calls are shown in the diagrams under the functions that made the file accesses. They appear as function names with their corresponding files in parentheses.

The Performance Analyzer can trace file access calls that use the following DOS calls:

- DosOpen()
- DosRead()
- DosWrite()
- DosClose()
- DosDupHandle()
- DosResetBuffer()
- DosSetFilePtr()
- DosSetFileLocks()
- DosSetFileSize()
- DosQueryFileInfo()
- DosQueryHType()

To set file access:

> Select the **File access** choice from the **Options** menu on the **Trace Generation** window. A check mark appears next to the choice to indicate that the choice is enabled.

To reset file access:

> Select the **File access** choice from the **Options** menu on the **Trace Generation** window. The check mark is removed to indicate that the choice is disabled.

Notes:

- The settings you enter are saved for the current session. If you want to save the settings for subsequent sessions, select the **Options** menu and then select the **Save** choice from the **Settings** cascaded menu.
- You can only select the **File access** choice when the _DOSCALL.LIB file is linked with your program.

Creating a Customized Trace File

Setting and Removing Triggers

A trigger turns tracing on when it is called and then turns tracing off when it returns. By setting triggers to start or stop tracing at selected points in your program, you can control the size of your trace file. You can set and remove triggers on functions. The Performance Analyzer allows you to set multiple triggers.

You can set and remove triggers from the **Edit** and **Function Pop-up** menus on the **Trace Generation** window. Remember the following when setting triggers:

- If triggers are set, the Performance Analyzer traces all enabled components and only those functions on which triggers are set. If no triggers are set, the Performance Analyzer traces all enabled components.
- If a trigger function is nested within another trigger function, tracing is turned off only after the outer function returns.
- A function that has a trigger set on it has the letter *T* in the icon next to its function name in the **Trace Generation** window.
- If you disable an EXE, DLL, or object file, the Performance Analyzer disables all functions within the selected file and removes any triggers set on functions within the file.
- If you set a trigger on a disabled function, the Performance Analyzer enables the function.
- If you disable a function, the Performance Analyzer removes any trigger set on it.

Changing the Buffer Size

During a trace analysis, the Performance Analyzer and your program share memory with the trace buffer. The trace buffer allows the Performance Analyzer to log events that are running in the address space of the program.

When the trace buffer is full, the Performance Analyzer:

1. Stops the program
2. Time stamps the start of the buffer flush
3. Writes the events in the buffer to the trace file
4. Time stamps the end of the buffer flush
5. Restarts the program.

   Note: Time stamping the buffer flush allows the diagrams to remove the buffer-flushing overhead time.

You can select the size of the trace buffer, and thereby change the time spent flushing the buffer to the trace file. When you increase the size of the buffer, more events are recorded before the Performance Analyzer flushes the buffer. Likewise, when you decrease the size of the buffer, fewer events are recorded before the Performance Analyzer flushes the buffer.

You can also enable buffer wrapping, which causes the Performance Analyzer to overwrite older events in the buffer with newer ones. Since the buffer is flushed only when the program ends, less disk space is needed for the trace file, but some trace data is lost. The default for buffer wrapping is disable.

Select **Buffer Control** from the **Options** menu on the **Trace Generation** window to change the buffer size or to enable buffer wrapping. For more information on disabling buffer wrapping, see the online Help topic “Buffer Control Window”.

Naming the Trace File

The default file name for a trace file is *myprog.trc*, where *myprog* is the file name of the program you are tracing. When you run several traces of the same program, the **Name trace file** choice lets you name each trace file and describe what you did differently for each trace.

For example, if you disable an object file for the first trace and disable time stamps for the second trace, you could name the first trace file *TRACE1* and enter *Disabled object file* for its description. Likewise, you could name the second trace file *TRACE2* and enter *Disabled time stamps* for its description.

A trace file's description is displayed in the **Status Area** of any open diagram.

To name a trace file, select **Name trace file** from the **Options** menu on the **Trace Generation** window. In the **Name Trace File** window, you can enter your own trace file name and a short description.

Saving Trace File Settings

Trace settings (settings that determine how a program is traced) that you enter are saved for the current session. You can save some trace settings for subsequent traces. If you want to save settings for subsequent sessions, select the **Options** menu and then select the **Save** choice from the **Settings** cascaded menu.

For information about specific settings saved, see “Save Choice” in the Performance Analyzer Help facility.

Saving Trace File Settings

34 Using the Performance Analyzer Diagrams

After you have created your trace file, the Performance Analyzer provides five diagrams in which you can view and analyze the data. Each diagram presents a different view of the trace file to give you an overall idea of how your program performs.

The following list contains a description of each diagram and shows the icon that represents it in the **Performance Analyzer - Window Manager** window when the diagram is open.

Icon/Diagram	**Description**
Call Nesting	Shows the flow of control and interactions among the various threads. Use this diagram to diagnose problems with critical sections, sequencing protocols, thread delays, and program deadlocks and crashes.
Dynamic Call Graph	Shows an overall view of the program and the flow of the program. You can easily see where the most time was spent.
Execution Density	Shows trends of program execution by displaying the trace data chronologically from top to bottom as thin horizontal lines of various colors in different columns.
Statistics	Provides a textual report of execution time by function or executable. You can use this diagram to find hot spots in the overall execution. You can also use this diagram to determine which user functions to inline.

Time Line Places the function calls and returns in sequence along a time line.

Opening a Trace File in a Diagram

To open a trace file in any diagram, use any of the following methods:

- From the **Trace File** menu of an open diagram, select **Open as** and then select a diagram from the cascaded menu.
- Click mouse button two on the file name or icon of a trace file in the **Performance Analyzer - Window Manager** window, then select a diagram from the **Trace File** pop-up menu.
- Double-click on the file name or icon of a trace file in the **Performance Analyzer - Window Manager** window, then select one of the diagram check boxes in the **Analyze Trace** window.
- Click on the **Analyze Trace** push button in the **Performance Analyzer - Window Manager** window, and then, in **Analyze Trace** window, enter a trace file name and select one of the diagram check boxes.

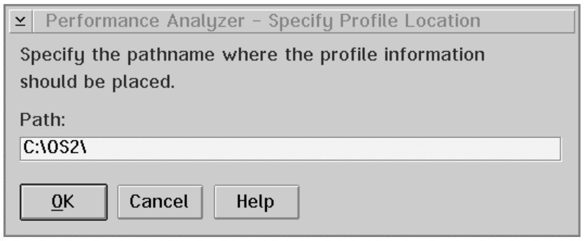
35 Introducing the Performance Analyzer Windows

The following pages briefly describe the Performance Analyzer windows. For more complete information, see the Performance Analyzer online Help facility. The primary Performance Analyzer windows are:

- Performance Analyzer - Specify Profile Location Window
- Performance Analyzer - Window Manager Window
- Create Trace Window
- Trace Generation Window
- Application Monitor Window
- Analyze Trace Window.

Performance Analyzer - Specify Profile Location Window

When you start the Performance Analyzer for the first time, the **Performance Analyzer - Specify Profile Location** window appears.


Performance Analyzer – Specify Profile Location

Specify the pathname where the profile information should be placed.

Path: C:\OS2\

Figure 119. Performance Analyzer - Specify Profile Location Window

This window prompts you to type the path name where you want to store the *ICSPERF.INI* file. The ICSPERF.INI file stores your session settings. The default path name is the drive and directory where your OS/2 operating system is installed. If you want to store the ICSPERF.INI file in a drive and directory other than the default, type the full path name in the **Path** entry field, and then select **OK.** The ICSPERF.INI file is created in the directory you specified.

Performance Analyzer - Window Manager Window

The **Performance Analyzer - Window Manager** window is the Performance Analyzer's main control window and is always displayed while the Performance Analyzer is running. Once you have properly compiled and linked your program and started the Performance Analyzer, you can start most functions from this window, including creating and analyzing trace files.

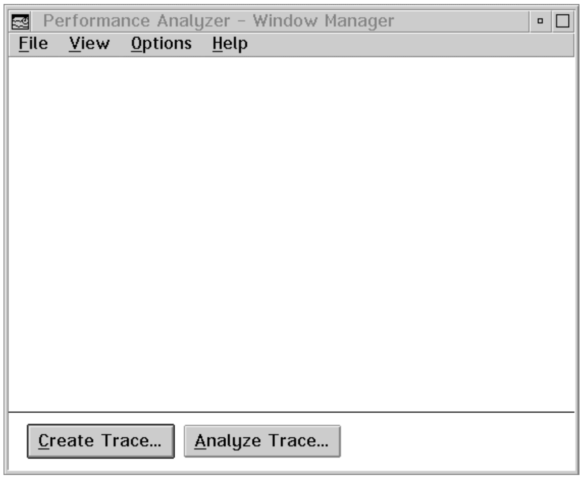


Figure 120. Performance Analyzer - Window Manager Window

When you view a trace file, this window lists the file names of your executable, your trace file, and each open diagram.

Areas of the Performance Analyzer - Window Manager Window

The following topics describe the areas of the **Window Manager** window.

Window Manager Menu Bar Summary

The menu choices in the **Window Manager** window are as follows:

File
From this menu, you can select:

Create Trace
Displays the **Create Trace** window, which lets you start creating a trace file for your program.

Analyze Trace
Displays the **Analyze Trace** window, which lets you open a diagram and start analyzing your program.

Exit Performance Analyzer
Lets you end the Performance Analyzer application.

View From this menu, you can select:

Tree lines
Displays tree lines on the **Performance Analyzer - Window Manager** window.

Show icons
Displays icons that identify file types on the **Performance Analyzer - Window Manager** window.

Remove all windows
Removes and closes all open diagrams from your screen.

Options From this menu, you can select:

Font	Displays the **Font** window, which lets you change the font, font style, and font size for the windows.
Quick exit	Provides a fast way to exit the Performance Analyzer.
Search paths	When you are working in the WorkFrame environment, this choice displays a window in which you can specify where the Performance Analyzer can locate source files for editing.
Unique trace file name	This choice gives each trace file a different name, which allows you to save several trace files created from the same program.
Settings	Displays a cascaded menu that lets you save settings or restore initial default settings.

Project (only available when working in the WorkFrame environment)
This menu appears on the **Performance Analyzer - Window Manager** window when you start the Performance Analyzer within the WorkFrame environment.

WorkFrame actions that can be launched from the **Performance Analyzer - Window Manager** window will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of EXE in the **WorkFrame Tool Setup** window.

Help Select choices from the **Help** menu to display the various types of Help information. From this menu, you can select:

Help index
Displays an index of Help topics.

General help
Displays Help for the active window.

Using help
Describes how to use Help.

How do I?
Displays task Help.

Product information
Displays information about the Performance Analyzer.

Window Manager Pop-up Menus

The pop-up menus allow you to quickly access features that are frequently used. The **Window Manager** window has the following pop-up menus:

- **Window Manager Executable** pop-up menu
- **Window Manager Trace File** pop-up menu
- **Window Manager Diagram** pop-up menu.

Window Manager Executable Pop-up Menu: To access this pop-up menu, click mouse button two on an the file name of an executable or the icon next to it. The menu is displayed with the following choices:

Create trace
Displays the window from which you can create a trace file. The file name on which you clicked appears in the window's **Program Name** entry field.

Close
Closes the selected executable, its associated trace file, and all diagrams in which the trace data is displayed.

Window Manager Trace File Pop-up Menu: To access this pop-up menu, click mouse button two on the file name of a trace file or the icon next to it. The menu is displayed with the following choices:

Analyze trace
Displays the window from which you can select Performance Analyzer diagrams to analyze the trace file. The file name on which you clicked appears in the window's **Trace File Name** entry field.

Open As Call Nesting
Opens the **Call Nesting** diagram and displays the trace file in it.

Open As Dynamic Call Graph
Opens the **Dynamic Call Graph** and displays the trace file in it.

Open As Execution Density
Opens the **Execution Density** diagram and displays the trace file in it.

Open As Statistics
Opens the **Statistics** diagram and displays the trace file in it.

Open As Time Line
Opens the **Time Line** diagram and displays the trace file in it.

Close
Closes the selected trace file and all diagrams in which the trace file is displayed.

Delete File
Deletes the selected trace file from your hard disk if you select **Yes** in the **Trace File - Delete** window (displayed when you select this choice).

Window Manager Diagram Pop-up Menu: To access this pop-up menu, click mouse button two on the file name of a diagram or the icon next to it. The menu is displayed with the following choices:

Display
Makes the selected diagram active and brings it to the foreground of your desktop.

Close
Closes the selected diagram.

Push Buttons

The **Window Manager** window has the following push buttons:

Create Trace
Displays the **Create Trace** window from which you can start creating a trace file.

Analyze Trace
Displays the **Analyze Trace** window from which you can select Performance Analyzer diagrams to analyze a trace file.

Create Trace Window

The **Create Trace** window lets you specify the name of the program that you want to trace and any parameters that you want to include in the trace.

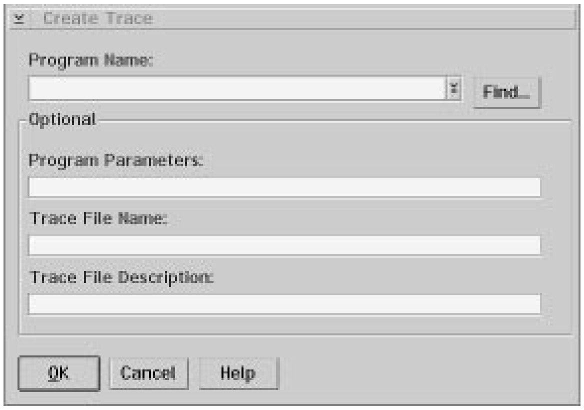


Figure 121. Create Trace Window

You can display the **Create Trace** window from the **Performance Analyzer - Window Manager** window by:

- Clicking on the **Create Trace** push button
- Selecting the **Create trace** choice from the **File** menu.
- Clicking mouse button two on an executable file name or icon (if displayed in the window), and selecting the **Create trace** choice from the pop-up menu.

Areas of the Create Trace Window

The following topics describe the areas of the **Create Trace** window.

Program Name: Entry Field

Type the full path name and program you want to trace in the **Program Name:** entry field. If the program is in your current directory, you do not have to type the path.

Note: If you are not sure where the file is located, select the **Find** push button.

Optional Entry Fields

The **Create Trace** window has the following optional entry fields:

Program Parameters:
Type any parameters that you want to pass to your program in the **Program Parameters:** entry field. This field is optional.

Trace File Name:
Type the name of the path and trace file in the **Trace File Name:** entry field.

The Performance Analyzer places the trace file in the current directory unless you specify a path. The default name for the trace file is *myprog.trc*, where *myprog* is the name of the program you are tracing. This field is optional.

Trace File Description:
Type any comments that you want to make about your trace in the **Trace File Description:** entry field. This field is optional.

Push Buttons The **Create Trace** window has the following push buttons:

Find Displays the **Find File** window, which helps you locate a file that you want to trace.

OK Saves the changes and closes the window.

Cancel Exits the current window without saving any changes.

Help Displays information about the current window.

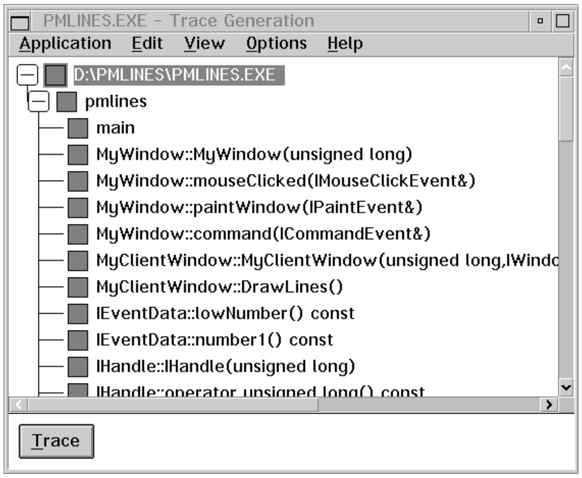
Note: Before creating a trace file, you must compile and link your program with the proper options as 📖 described in “Compiling and Linking Your Program” on page 479.

Trace Generation Window

The **Trace Generation** window lists the file names of the preloaded components in your program and lets you control which parts of your program are traced.

A component can be an executable file, a dynamic link library file, an object file, or a function. EXE and DLL files contain object files, and object files contain functions.

Introducing the Performance Analyzer Windows



D:\PMLINES\PMLINES.EXE
pmlines
main
MyWindow::MyWindow(unsigned long)
MyWindow::mouseClicked(IMouseClickEvent&)
MyWindow::paintWindow(IPaintEvent&)
MyWindow::command(ICommandEvent&)
MyClientWindow::MyClientWindow(unsigned long,IWindo
MyClientWindow::DrawLines()
IEventData::lowNumber() const
IEventData::number1() const
IHandle::IHandle(unsigned long)
IHandle::operator unsigned long() const

Figure 122. Trace Generation Window

To view or hide components, click on the plus/minus icons to expand and contract EXE, DLL, and object files.

Note: Before creating a trace file, you must compile and link your program with the proper options as described in “Compiling and Linking Your Program” on page 479.

Areas of the Trace Generation Window

The following topics describe the areas of the **Trace Generation** window.

Trace Generation Menu Bar Summary

The menu choices in the **Trace Generation** window are as follows:

Application From this menu, you can select:

Window Manager
Displays the **Performance Analyzer - Window Manager** window.

Exit the Performance Analyzer
Exits the Performance Analyzer application.

Edit The **Edit** menu is a dynamic menu, which displays choices based on the type of component selected. From this menu, you can select:

Enable all executables
Enables all functions in all executable files.

Disable all executables
Disables all functions in all executable files.

Enable executable
Enables all functions in a selected executable file. This choice is available when you select a disabled executable.

Disable executable
Disables all functions in a selected executable file. This choice is available when you select an enabled executable.

Enable object file
Enables all functions in a selected object file. This choice is available when you select a disabled object file.

Disable object file
Disables all functions in a selected object file. This choice is available when you select an enabled object file.

Enable function
Enables a function. This choice is available when you select a disabled function.

Disable function
Disables a function. This choice is available when you select an enabled function.

Set trigger
Sets a trigger on a function so that the Performance Analyzer traces the function and its associated calls. This choice is available when you select a function that ***does not*** have a trigger set on it.

Note: If triggers are set, the Performance Analyzer traces all enabled components and only those functions on which triggers are set. If no triggers are set, the Performance Analyzer traces all enabled components.

Remove trigger
Removes a trigger on a function so that the Performance Analyzer does not trace the function and its associated calls. This choice is available when you select a function that ***has*** a trigger set on it.

Note: If triggers are set, the Performance Analyzer traces all enabled components and only those functions on which triggers are set. If no triggers are set, the Performance Analyzer traces all enabled components.

View From this menu, you can select:

Traceable filter
Displays only the components that are traceable.

Tree lines
Displays lines between file names to show relationships.

Options The choices on this menu let you set options to customize your trace sessions. From this menu, you can select:

Buffer control
Select the size of the event log buffer, plus enable and disable buffer wrapping.

Call depth
Select the call depth limit for each thread.

File access
Select whether to trace file accesses.

Font
Displays the **Font** window, which lets you change the font, font style, and font size for the **Trace Generation** window.

Name trace file
Specify a new path or file name for the trace file. Also allows you to add or change the trace file description.

Timeout control
Select the maximum number of seconds your program may run without logging events. This choice is useful when your program is in a continuous loop or deadlock.

Time stamp events
Select to log or not log time stamps.

Settings
Displays a cascaded menu that lets you save settings or restore initial default settings.

Project This menu appears on the **Trace Generation** window when you start the Performance Analyzer within the WorkFrame environment. From this menu, you can select:

Edit source
Displays the source file for a selected object file or function in WorkFrame's default editor.

WorkFrame actions
WorkFrame actions that can be launched from the **Trace Generation** window will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of EXE in the **WorkFrame Tool Setup** window.

Help This menu provides choices that display various types of Help information. From this menu, you can select:

Help index choice
Displays an index of Help topics.

General help choice
Displays Help for the active window. The online Help panel displayed is the same panel that is displayed when you place your cursor inside the window and press F1.

Using help choice
Describes how to use Help.

How do I? choice
Displays task help.

Product information choice
Displays information about the Performance Analyzer.

Trace Generation Pop-up Menus

Pop-up menus allow you to quickly access features that are frequently used. The **Trace Generation** window has a pop-up menu for each type of program component displayed on the window. The pop-up menu displayed depends on the type of component you click on.

The **Trace Generation** window pop-up menus are as follows:

- **Trace Generation Executable** pop-up menu
- **Trace Generation Object File** pop-up menu
- **Trace Generation Function** pop-up menu.

Trace Generation Executable Pop-up Menu: To access this pop-up menu, click mouse button two on the file name of an executable or dynamic link library file (or the plus/minus icon next to it). The pop-up menu and the choices listed in the menu are different if you click on another file type. The menu is displayed with the following choices:

Disable executable
Disables the selected executable file so that the Performance Analyzer does not record trace analysis data for it in the trace file. This choice is available when you select an enabled executable.

Enable executable
Enables the selected executable file so that the Performance Analyzer records trace analysis data for it in the trace file. This choice is available when you select a disabled executable.

Trace Generation Object File Pop-up Menu: To access this pop-up menu, click mouse button two on the file name of an object file (or the plus/minus icon next to it). The pop-up menu and the choices listed in the menu are different if you click on another file type. The menu is displayed with the following choices:

Disable object file
Disables the selected object file so that the Performance Analyzer *does not* record trace analysis data for it in the trace file. This choice is available when you select an *enabled* object file.

Enable object file
Enables the selected object file so that the Performance Analyzer records trace analysis data for it in the trace file. This choice is available when you select a *disabled* object file.

Edit source
Displays the source file for a selected object file in WorkFrame's default editor. The Performance Analyzer finds the source file and opens it to the first line of the file.

Trace Generation Function Pop-up Menu: To access this pop-up menu, click mouse button two on the name of a function (or the plus/minus icon next to it). The pop-up menu and the choices listed in the menu are different if you click on another file type. The menu is displayed with the following choices:

Disable function
Disables the selected function so that the Performance Analyzer *does not* record trace analysis data for it in the trace file. This choice is available when you select an *enabled* function.

Enable function
Enables the selected function so that the Performance Analyzer records trace analysis data for it in the trace file. This choice is available when you select a *disabled* function.

Set trigger
Sets a trigger on a function so that the Performance Analyzer traces the function and its associated calls. This choice is available when you select a function that *does not* have a trigger set on it.

Note: If triggers *have not* been set, the Performance Analyzer traces enabled executables, DLLs, object files, and functions (and their associated calls). If triggers ***have*** been set, the Performance Analyzer traces enabled executables, DLLs, and object files, but it only traces functions (and their associated calls) on which triggers have been set.

Remove trigger
Removes a trigger on a function so that the Performance Analyzer *does not* trace the function and its associated calls. This choice is available when you select a function that *has* a trigger set on it.

Note: If triggers ***have not*** been set, the Performance Analyzer traces enabled executables, DLLs, object files, and functions (and their associated calls). If triggers ***have*** been set, the Performance Analyzer traces enabled executables, DLLs, and object files, but it only traces functions (and their associated calls) on which triggers have been set.

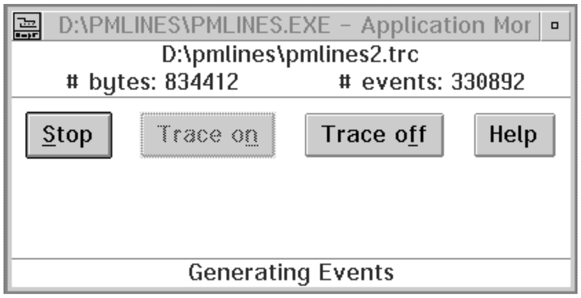
Edit source
Displays the source file for a selected function in WorkFrame's default editor. The Performance Analyzer finds the source file and opens it to the line where the function begins.

Push Button The **Trace Generation** window has the following push button:

Trace Use the **Trace** push button to close the **Trace Generation** window and begin tracing your program.

Application Monitor Window

After you select the **Trace** push button on the **Trace Generation** window to start tracing your program, the Performance Analyzer displays the **Application Monitor** window.



D:\PMLINES\PMLINES.EXE – Application Mor

D:\pmlines\pmlines2.trc

bytes: 834412 # events: 330892

Generating Events

Figure 123. Application Monitor Window

This window is displayed until your entire program has executed or you select the **Stop** push button, which causes your program to stop executing. When you stop the execution of your program, you also stop the collection of trace data.

Areas of the Application Monitor Window

The following topics describe the areas of the **Application Monitor** window.

Application Monitor Status Area The following information is displayed in the **Application Monitor** window **Status Area**:

- Name of the program being traced
- Name of the trace file
- Number of bytes written to the trace file
- Number of events written to the trace file

Push Buttons

The **Application Monitor** window has the following push buttons:

Stop Stops the execution of your program and the collection of trace data.

The **Analyze Trace** window is displayed after the trace stops so that you can select one or more diagrams in which to view the trace data.

Trace on Starts tracing. **Trace on** does not cause your program to start running; it causes trace events to start being recorded to the trace file. When the **Application Monitor** window is displayed, the Performance Analyzer has already started tracing events, so you cannot select **Trace on** until you have first selected **Trace off**.

You can also turn tracing on from your program by calling the Performance Analyzer function PerfStart. For instructions on calling PerfStart, see the Performance Analyzer online Help facility.

Trace off Stops tracing. **Trace off** does not cause your program to stop running; it stops trace events from being recorded to the trace file. When the **Application Monitor** window is displayed, the Performance Analyzer has already started tracing events, so you can select **Trace off** anytime after the window is displayed.

Note: Return events are still logged for functions that were called before you selected Trace off, and execution time will still be logged against functions that were called but have not returned to the caller.

You can also turn tracing off from your program by calling the Performance Analyzer function PerfStop. For instructions on calling PerfStop, see the Performance Analyzer online Help facility.

Analyze Trace Window

The **Analyze Trace** window lets you specify the name of the trace file that you want to analyze and the diagrams in which you want to display it.

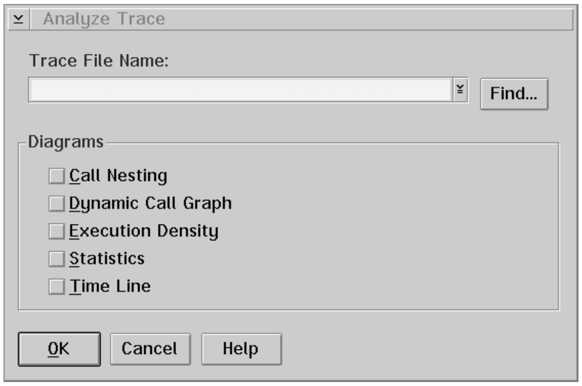


Figure 124. Analyze Trace Window

Note: You must have created a trace file before you can open a diagram. See Chapter 33, "Creating Trace Files" on page 489 for instructions.

You can display the **Analyze Trace** window from the **Performance Analyzer - Window Manager** window by:

- Clicking on the **Analyze Trace** push button
- Selecting the **Analyze trace** choice from the **File** menu.
- Clicking mouse button two on a trace file name or icon (if displayed in the window), and selecting the **Analyze trace** choice from the pop-up menu.

Areas of the Analyze Trace Window

The following topics describe the areas of the **Analyze Trace** window.

Trace File Name: Entry Field

Type the full path and file name of the trace file that you want to analyze in the **Trace File Name:** entry field. If the file is in your current directory, you do not have to type the path.

Note: If you are not sure where the file is located, select the **Find** push button.

Display Diagrams Check Boxes

Select the check boxes in the **Analyze Trace** window to open a trace file in one or more diagrams.

Push Buttons

The **Analyze Trace** window has the following push buttons:

Find Displays the **Find File** window, which helps you locate a file that you want to analyze.

OK Saves the changes and closes the window.

Cancel Exits the current window without saving any changes.

Help Displays Help information about the current window.

36 Managing Trace Files

The Performance Analyzer diagrams provide methods of making trace data easier to work with and view. The following topics describe some of these methods.

Using Filtering

Filters allow you to temporarily reduce the amount of trace data displayed in a diagram. There are several techniques for filtering the trace file and isolating interesting or problematic areas.

The following list contains filtering techniques that you can use in each of the diagrams:

Call Nesting diagram

In this diagram, you can filter data by:

- Selecting a function and viewing its call stack.
- Selecting specific functions and threads to view. The Performance Analyzer displays only those selected.
- Selecting the thread layout to arrange the starting placement of multiple threads within the diagram.
- Selecting a region of time to view.

Dynamic Call Graph

In this diagram, you can filter data by:

- Selecting specific threads to view. The Performance Analyzer displays trace information for only those threads selected.
- Zooming in on a region of the graph that is of most interest.
- Scaling node sizes to change the size of the graphical representation of functions.
- Using the Overview function to display a miniature view of the graph, which allows you to quickly navigate in the diagram.

Execution Density diagram

In this diagram, you can filter data by:

- Selecting specific functions and threads to view. The Performance Analyzer displays only those selected.
- Zooming in on a region of the diagram that is of most interest.
- Scaling the pages of the diagram to reveal more information.
- Selecting a region of time to view.

Statistics diagram

In this diagram, you can filter data by:

- Selecting specific threads to view. The Performance Analyzer displays trace information for only those threads selected.
- Sorting functions by columns that are of most interest and using the split bars to change the size of the window panes.

Time Line diagram

In this diagram, you can filter data by:

- Selecting a function and viewing its call stack.
- Zooming in on a region of the diagram that is of most interest.
- Scaling the pages of the diagram to reveal more information.
- Selecting a region of time to view.

Using Scaling

Scaling is a way to change how much detail is displayed. You can view the diagram as a whole or magnify areas to see the finer detail.

When details are hidden, you cannot spot patterns, anomalies, or features of the execution because there is too much information on the screen. The **Execution Density** and **Time Line** diagrams can be scaled along the time dimension.

Using Scrolling

You can scroll the window in all diagrams to focus on areas of interest. You can also use correlation to scroll to areas of interest in each of the chronologically-scaled diagrams (**Call Nesting**, **Execution Density**, and **Time Line**). Correlation is described in "Understanding Correlation" on page 517.

Using Multiple Views

The Performance Analyzer allows you to open a trace file in several diagrams or multiple views of the same diagram simultaneously. Sometimes opening two or more diagrams can help you better understand a program.

For instance, if you have a new program to learn, and you don't want to wade through code listings to determine how the code works, you can display and use the **Dynamic Call Graph**, **Call Nesting**, and **Time Line** diagrams to get a good understanding of the program's flow.

After you have opened a diagram, one way you can open another diagram is by selecting the **Open as** choice from a **Trace file** menu, and then selecting another diagram from the cascaded menu.

Recognizing Patterns

Program loops cause the same sequence of calls and returns to be repeated in the trace. The Performance Analyzer lets you combine like sequences in the **Call Nesting** diagram.

By using **Pattern Recognition** you can reduce the amount of screen space the diagram uses. Pattern recognition looks at a single thread and finds patterns of calls and returns. When this choice is enabled, the **Call Nesting** diagram displays these patterns as a curved arc and the number of repetitions are shown to the right.

This technique shortens the number of pages which you must scroll through to look at your trace file.

If you see a large repetition of patterns, you could group the functions together with *pragma alloc_text* statements to improve performance by limiting the number of page swaps between calls in the patterns.

Note: Pattern Recognition is only available when filtering by threads.

Understanding Correlation

Correlation is helpful because one diagram cannot show everything of interest within a trace file. Additionally, some events are easier to find in one diagram, but the information in another is more meaningful; therefore, you can locate the event in one diagram and correlate to another.

The Performance Analyzer provides three time-scaled diagrams that can be correlated: **Call Nesting**, **Execution Density**, and **Time Line**. You can correlate these diagrams based on a specific time or event, or on a range of time or events.

For example, use the **Call Nesting** diagram to identify the order and names of functions called, and then use the **Time Line** diagram to find out how long a function took to execute.

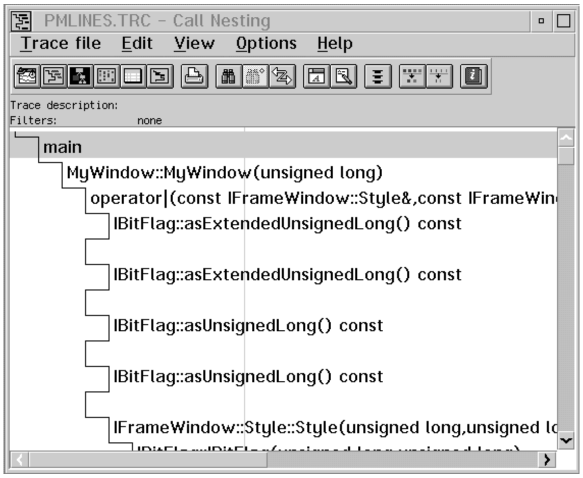
Or you can use the **Execution Density** diagram to see general patterns that lead up to a certain point, and then correlate that point to the **Call Nesting** diagram to see the exact order of the function calls.

For instructions on correlating diagrams, see the Performance Analyzer online Help facility.

Managing Trace Files

37 Call Nesting Diagram

The **Call Nesting** diagram shows the trace file as a vertical series of function calls and returns. Use this diagram to diagnose problems with critical sections, sequencing protocols, program deadlocks and crashes, and thread delays.



```
Trace description:
Filters:          none
main
  MyWindow::MyWindow(unsigned long)
    operator|(const IFrameWindow::Style&,const IFrameWin
      IBitFlag::asExtendedUnsignedLong() const
      IBitFlag::asExtendedUnsignedLong() const
      IBitFlag::asUnsignedLong() const
      IBitFlag::asUnsignedLong() const
      IFrameWindow::Style::Style(unsigned long,unsigned lo
```

Figure 125. Call Nesting Diagram

Each thread in the **Call Nesting** diagram has its own starting column of functions. A call is shown as a step to the right and a return as a line back to the left. The calls are labeled with the name of the function being called.

Use the mouse to select a call, a return, or a user event. When the call is selected, it is highlighted.

Context switches between threads are shown by dashed horizontal lines. While the vertical lines do not show elapsed scaled times in this diagram, you can clearly see the flow of control and the interactions among the various threads.

Areas of the Call Nesting Diagram

The following topics describe the areas of the **Call Nesting** diagram.

Call Nesting Menu Bar Summary

The menu choices in the **Call Nesting** diagram are as follows:

Trace file Allows you to perform functions with your existing trace file. From this menu, you can select:

Open as

Displays a cascaded menu with the following choices. Select the name of the diagram you want to view.

- Call Nesting
- Dynamic Call Graph
- Execution Density
- Statistics
- Time Line

Printer settings

Allows you to select a printer and various options for your printed output.

Print selected region

Prints a selected area of the diagram.

Window manager

Displays the **Performance Analyzer - Window Manager** window.

Exit the Performance Analyzer

Exits the Performance Analyzer application.

Edit Allows you to locate and change text in the **Call Nesting** diagram. From this menu, you can select:

Find

Displays a cascaded menu from which you can select:

Function	Locates the text of function calls.
User event	Locates user events.
Annotation	Locates annotated text.

Find next

Locates the next occurrence of the last item you searched for.

Annotate

Allows you to insert comments into your diagram.

Select time
Allows you to go to a specific time in the diagram.

Select time range
Allows you to select all events in a specified time range.

Select all
Select the entire **Call Nesting** diagram.

View Allows you to change displayed information. From this menu, you can select:

Include functions
Controls the functions to include or exclude in the diagram.

Include threads
Controls the threads to include or exclude in the diagram.

Options Allows you to customize the **Call Nesting** diagram and display additional information. From this menu, you can select:

Call stack
Displays all the functions currently on the call stack from a selected point.

Correlation
Synchronizes other diagrams to display the same highlighted region.

Font
Selects the font, font style, and font size for the function names.

Thread layout
Selects the amount of indentation for each thread column, and draws separator bars between threads.

Status area
Area at the top of the window that describes the diagram. You can select which items appear in this area.

Tool bar
Displays a cascaded menu that lets you select:

Show Choose to either show or hide the Tool bar.
Hover Choose to either enable or disable displaying the help text when the mouse pointer hovers over the Tool bar buttons.

Settings
Displays a cascaded menu that lets you either save the current settings or restore the initial default settings.

Save
Save the current session settings.

Restore initial defaults
Restore the original settings.

Project This menu appears when you start the Performance Analyzer from within the WorkFrame environment. From this menu, you can select:

Edit function
Displays the source code for the selected function in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Call Nesting** diagram will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Help This menu provides choices that display various types of Help information. From this menu, you can select:

Help index
Displays an index of Help topics.

General help
Displays Help for the active window. The online Help panel displayed is the same panel that is displayed when you place your cursor inside the window and press F1.

Using help
Describes how to use Help.

How do I?
Displays task help.

Product information
Displays information about the Performance Analyzer.

Call Nesting Status Area

The **Status Area**, located at the top of the window, describes the settings of the diagram. You can select the **Status area** choice from the **Options** menu to change the appearance of the **Status Area**. When you select **Status area**, you can select the following in the **Status Area** window:

Trace description
Select for a brief description of the trace file.

Filters
Select to display selected filters.

Call Nesting Pop-up Menus

The pop-up menus allow you to quickly access features that are frequently used. The **Call Nesting** diagram has two pop-up menus: the **Call Nesting Diagram** pop-up menu and the **Call Nesting Selected Item** pop-up menu.

The **Call Nesting Diagram** pop-up menu contains actions that can be applied to the entire diagram, and the **Call Nesting Selected Item** pop-up menu contains actions that can be applied to a highlighted area.

Call Nesting Diagram Pop-Up Menu: To access this pop-up menu, click mouse button two on the background area of the diagram. The menu is displayed with the following choices:

Find
Displays a cascaded menu from which you can select:

Function	Locates the text of function calls.
User event	Locates user events.
Annotation	Locates annotated text.

Find next
Locates the next occurrence of the last item you searched for.

Include functions
Controls the functions to include or exclude in the diagram.

Include threads
Controls the threads to include or exclude in the diagram.

Font
Selects the font, font style, and font size for the function names.

Thread layout
Selects the amount of indentation for each thread column, and draws separator bars between threads.

Call Nesting Diagram

Call Nesting Selected Item Pop-Up Menu: To access this pop-up menu, click mouse button two on the highlighted area of the diagram. The menu is displayed with the following choices:

Annotate
Allows you to insert comments into your diagram.

Call stack
Displays all the functions currently on the call stack from a selected point.

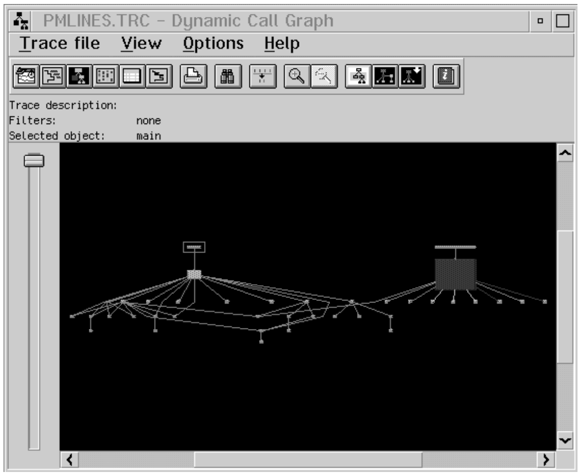
Correlation
Synchronizes other diagrams to display the same highlighted region.

Edit function
Displays the source code for the selected function in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Call Nesting** diagram will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Dynamic Call Graph

The **Dynamic Call Graph** is a graphical view of the execution of the target program. This diagram uses nodes and arcs to represent functions and calls.



```
Trace description:
Filters:            none
Selected object:    main
```

Figure 126. Dynamic Call Graph

Colors and sizes of nodes and arcs depict the time spent in the node and the number of calls between nodes.

A node represent a function and appears as a rectangle on the diagram. An arc, which is displayed as a line between a pair of nodes, represents a call from one function to another.

In this diagram, you can double-click on any:

- Node for the **Function Information** window to display additional information.
- Arc for the **Who Calls Whom** window to display additional information. Only calls made during the given execution of the program are displayed.

Dynamic Call Graph Arcs and Nodes

Colors and sizes of nodes and arcs depict the time spent in the node and the number of calls between nodes.

The time spent in a particular node and the number of calls in an arc are shown in different colors. The following table shows what each color means to nodes and arcs. Node colors are based on the maximum executable time spent in a function. Arc colors are based on the maximum number of calls between pairs of functions.

Color and Size Representation of Nodes and Arcs

Color	Nodes	Arc
Gray	0 - 1/8	0 - 1/8
Blue	1/8 - 1/4	1/8 - 1/4
Yellow	1/4 - 1/2	1/4 - 1/2
Red	1/2 - maximum	1/2 - maximum

Note: The currently selected nodes are surrounded by a green box and the currently selected arcs are shown in green.

Functions

Double clicking on a function displays a function information dialog which shows you the function name, object name and executable name. Select one of the following buttons:

- Select the **Who calls me** button to display the fully qualified function, listed in the **Function Information** window, and functions that called it.
- Select the **Whom do I call** button to display the fully qualified function, listed in the **Function Information** window and functions that it called.

Arcs

Double clicking on an arc displays an arc information dialog window which shows fully qualified function names of the nodes involved with this call. Select one of the following buttons:

- Select the **Find caller** button to display the function that originated the call. The originating caller is displayed in the center of the Dynamic Call Graph.
- Select the **Find callee** button to display the function called by the originating caller. The called function is displayed in the center of the Dynamic Call Graph.

Areas of the Dynamic Call Graph

The following topics describe the areas of the **Dynamic Call Graph**.

Dynamic Call Graph Menu Bar Summary

The menu choices in the **Dynamic Call Graph** are as follows:

Trace file Allows you to perform functions with your existing trace file. From this menu, you can select:

Open as
Displays a cascaded menu with the following choices. Select the name of the diagram you want to view.

- Call Nesting
- Dynamic Call Graph
- Execution Density
- Statistics
- Time Line

Printer settings
Allows you to select a printer and various options for your printed output.

Print selected region
Prints a selected area of the diagram.

Window manager
Displays the **Performance Analyzer - Window Manager** window.

Exit the Performance Analyzer
Exits the Performance Analyzer application.

View Allows you to change displayed information. From this menu, you can select:

Include threads
Controls the threads to include or exclude in the diagram.

Overview
Allows you to navigate through large diagrams quickly.

Zoom in
Enlarges the size of the diagram without changing the size of the window.

Zoom out
Decreases the size of the diagram without changing the size of the window.

Dynamic Call Graph

Re-lay graph
Sizes to fit all of the diagram in the window.

Restore nodes
Restores nodes to the diagram to show the default view.

Options Allows you to customize the **Dynamic Call Graph** and display additional information. From this menu, you can select:

Scale node sizes
Scales node sizes.

Find function
Searches for functions in the diagram.

Status area
Area at the top of the window describes the diagram.

Tool bar
Displays a cascaded menu that lets you select:

Show Choose to either show or hide the Tool bar.

Hover Choose to either enable or disable displaying the help text when the mouse pointer hovers over the Tool bar buttons.

Settings
Displays a cascaded menu that lets you either save the current settings or restore the initial default settings.

Save
Save the current session settings.

Restore initial defaults
Restore the original settings.

Project This menu appears when you start the Performance Analyzer from within the WorkFrame environment. From this menu, you can select:

Edit function
Displays the source code for the selected function in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Dynamic Call Graph** will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Help This menu provides choices that display various types of Help information. From this menu, you can select:

Help index
Displays an index of Help topics.

General help
Displays Help for the active window. The online Help panel displayed is the same panel that is displayed when you place your cursor inside the window and press F1.

Using help
Describes how to use Help.

How do I?
Displays task Help.

Product information
Displays information about the Performance Analyzer.

Dynamic Call Graph Status Area

The **Status Area**, located at the top of the window, describes the settings of the diagram. You can select the **Status area** choice from the **Options** menu to change the appearance of the **Status Area**. When you select **Status area**, you can select the following in the **Status Area** window:

Trace description
Select for a brief description of the trace file.

Filters
Select to display selected filters.

Selected object
Select to display selected object.

Note: If you highlight a node, a function name is displayed. If you highlight an arc, a function call is displayed.

Dynamic Call Graph Zoom Bar

When zooming in and out, the **Dynamic Call Graph** takes the selected node or arc that is highlighted, and uses it as the focal point. There are several ways to **zoom** in the **Dynamic Call Graph** window.

You can use the:

- **Zoom in** choice from the **View** menu to change the diagram view.

  When zooming in, your view is a step closer to the selected object and the object in the window appears larger.

- **Zoom out** choice from the **View** menu to change the diagram view.

  When zooming out, your view is a step further away from the selected object and the object in the window appears smaller.

- **Zoom bar** in the **Dynamic Call Graph** window by sliding the arm up or down, to change the diagram view.
- **Re-lay** choice from the **View** menu to return to the original diagram view.
- **Overview** choice from the **View** menu to quickly move around in the viewing area of the diagram.

  The **Overview** choice provides you with another option of viewing. Use it to move quickly to other areas of the diagram when zooming in and out.

  When using the **Overview** choice, you have a miniature version of the **Dynamic Call Graph** window. The gray box highlights the area currently in view in the window.

  Moving the small gray box around in the **Overview** window allows you to change the view in the **Dynamic Call Graph** quickly. You can also grab the sides of the gray box to resize the area being shown in the diagram window.

Dynamic Call Graph Function Information Window

Use the **Function Information** window to display information about:

- A selected function
- The total execution time in seconds and percentages
- Active time in seconds and percentages
- Which functions called the selected function
- The functions that the selected function called.

To display the **Function Information** window double-click on a node. The name of the function appears on the left side of the split bar area. The object file and executable name appear on the right side of the split bar area.

The following information is provided in the **Function Information Window:**

- Execution time (in time and percentage)
- Time on stack (in time and percentage)
- Number of calls

The following lists the push buttons and gives an explanation of each one:

Who calls me Select this push button to display only the selected node and the nodes that called the selected node. To return to the diagram, select **Restore graph** from the **Options** menu.

Whom do I call Select this push button to display only the selected node and the nodes that the selected node calls.

Cancel Closes the window.

Help Displays Help.

Note: If the trace file does not have time stamps, the:

- *Execution Time* and *Time on stack* will not be displayed.
- Nodes will all be displayed in the same color and size.

  and
- **Scale node sizes** choice will be disabled.

Dynamic Call Graph Who Calls Whom Window

Use the **Who Calls Whom** window to display information about a selected arc.

To display the **Who Calls Whom** window, double-click on an arc. The **Who Calls Whom** window shows the name of the calling function, the called function, and the number of times the call is made.

The calling function field and the called function field each have a **split bar** function between the function name and the object file and executable name. Use the **split bar** to change the size of the left or right field in either the calling function or the called function fields.

The **Who Calls Whom** window has the following push buttons:

Find caller
Selects and centers the node that makes the call represented by the selected arc.

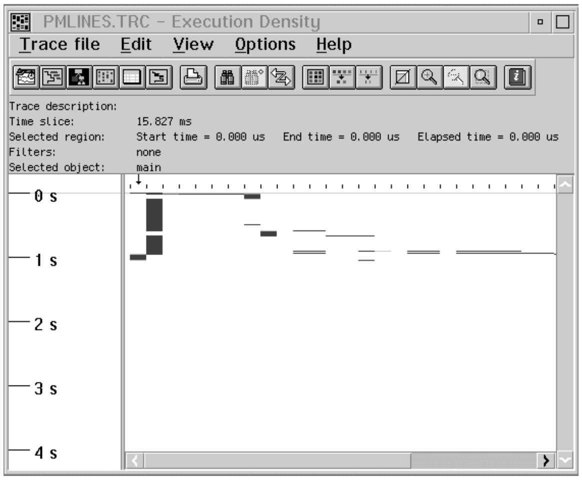
Find callee
Selects and centers the node that is called by the call represented in the selected arc.

Cancel
Closes the window.

Dynamic Call Graph

39 Execution Density Diagram

The **Execution Density** diagram shows trends of program execution by displaying the trace data chronologically from top to bottom as thin horizontal lines of various colors in different columns.



```
Trace description:
Time slice:          15.827 ms
Selected region:     Start time = 0.000 us   End time = 0.000 us   Elapsed time = 0.000 us
Filters:             none
Selected object:     main
```

Figure 127. Execution Density Diagram

This diagram consists of columns which contain thin lines of various colors. The following list describes diagram components:

- Each vertical column represents a function.
- The thickness of each line represents a unit of time called a time slice.
- The color of each line represents the percentage of program execution time spent in the given function for that time slice. Only selected threads are used in calculating this percentage.

For instance, in the default setting, functions executing more than 50 percent of a given time slice have a red line drawn in the appropriate column at the vertical location corresponding to the time slice.

Magnification is initially turned off, meaning the entire trace file is visible in the window. You can magnify or filter the diagram to vary the amount of detail displayed.

Notes:

1. When a thread switch occurs in the **Execution Density** diagram, the time between the last recorded event in the previous thread and the first event in the new thread will be allotted to the previous thread.
2. Events that take small amounts of time might be hard to distinguish until the magnification has been increased.

Areas of the Execution Density Diagram

The following topics describe the areas of the **Execution Density** diagram.

Execution Density Menu Bar Summary

The menu choices in the **Execution Density** diagram are as follows:

Trace file Allows you to perform functions with your existing trace file. From this menu, you can select:

Open as

Displays a cascaded menu with the following choices. Select the name of the diagram you want to view.

- Call Nesting
- Dynamic Call Graph
- Execution Density
- Statistics
- Time Line

Printer settings

Allows you to select a printer and various options for your printed output.

Print selected region

Prints a selected area of the diagram.

Window manager

Displays the **Performance Analyzer - Window Manager** window.

Exit the Performance Analyzer

Exits the Performance Analyzer application.

Edit Allows you to locate and change text in the **Execution Density** diagram. From this menu, you can select:

Find function
Search for a function.

Find Next
Find the next occurrence of the last item you searched for.

Select time
Go to a specific time in the diagram.

Select time range
Select a specified time in the diagram.

Select all
Select the entire **Execution Density** diagram.

View Allows you to change displayed information. From this menu, you can select:

Zoom in
Magnifies the **Execution Density** diagram to view a region of interest.

Zoom out
Reduces the **Execution Density** diagram.

Zoom to selected range
Magnifies an area of interest.

Scale pages
Controls the number of pages the diagram uses.

Include functions
Isolates specific functions to view in your program.

Include threads
Selects which threads are displayed in the diagram.

Options Allows you to customize the **Execution Density** diagram and display additional information. From this menu, you can select:

Color
Selects the colors used to display the percentage of time used by each function and the percentage of time each color represents.

Column width
Controls the width (in pixels) for each function column in the diagram.

Correlation
Synchronizes other diagrams to display the same highlighted region.

Execution Density Diagram

Status area
Control the gray area at the top of the window.

Tool bar
Displays a cascaded menu that lets you select:

Show Choose to either show or hide the Tool bar.

Hover Choose to either enable or disable displaying the help text when the mouse pointer hovers over the Tool bar buttons.

Settings
Displays a cascaded menu that lets you either save the current settings or restore the initial default settings.

Save
Save the current session settings.

Restore initial defaults
Restore the original settings.

Project This menu appears when you start the Performance Analyzer from within the WorkFrame environment. From this menu, you can select:

Edit function Displays the source code for the selected function or object file in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Execution Density** diagram will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Help This menu provides choices that display various types of Help information. From this menu, you can select:

Help index
Displays an index of Help topics.

General help
Displays Help for the active window.

Using help
Describes how to use Help.

How do I?
Displays task Help.

Product information
Displays information about the Performance Analyzer.

Execution Density Status Area

The **Status Area**, located at the top of the window, describes the settings of the diagram. You can select the **Status area** choice from the **Options** menu to change the appearance of the **Status Area**. When you select **Status area**, you can select the following in the **Status Area** window:

Trace description
Shows the description given to the trace file.

Time slice
Displays the value of the time slice.

Selected region
Displays the start, end, and total time of a selected region.

Filters
Shows when selected filters, such as functions and threads, are active.

Selected object
Displays the name of the selected object.

Execution Density Pop-up Menus

The pop-up menus allow you to quickly access features that are frequently used. The **Execution Density** diagram has two pop-up menus: the **Execution Density Diagram** pop-up menu and the **Execution Density Selected Item** pop-up menu.

Execution Density Diagram Pop-up Menu

This pop-up menu contains most of the choices from the **Edit** and **Options** menus. To access this pop-up menu, click mouse button two on the background area of the diagram. The menu is displayed with the following choices:

Find Function
Search for a function.

Find next
Find the next occurrence of the last item you searched for.

Zoom in
Magnifies the **Execution Density** diagram to view a region of interest.

Zoom out
Reduces the **Execution Density** diagram.

Scale pages
Controls the number of pages the diagram uses.

Include functions
Isolates specific functions to view in your program.

Include threads
Selects which threads are displayed in the diagram.

Color
Selects the colors used to display the percentage of time used by each function and the percentage of time each color represents.

Column width
Controls the width (in pixels) for each function column in the diagram.

Execution Density Selected Item Pop-up Menu

To access this pop-up menu, highlight a region of interest, move the mouse pointer into that area, and click on mouse button two. This menu is displayed with the following choices:

Zoom to selected range
Magnifies an area of interest.

Correlation
Synchronizes other diagrams to display the same highlighted region.

Edit function
Displays the source code for the selected function or object file in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Execution Density** diagram will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Execution Density Current Column Indicator

An arrow called the current column indicator is displayed at the top of the columns. The current column indicator lets you move to each column. Use the mouse to move to each column.

To move the current column indicator, click on the column of interest and the arrow moves. You can drag the current column indicator arrow with the mouse and the **Selected object** information will change.

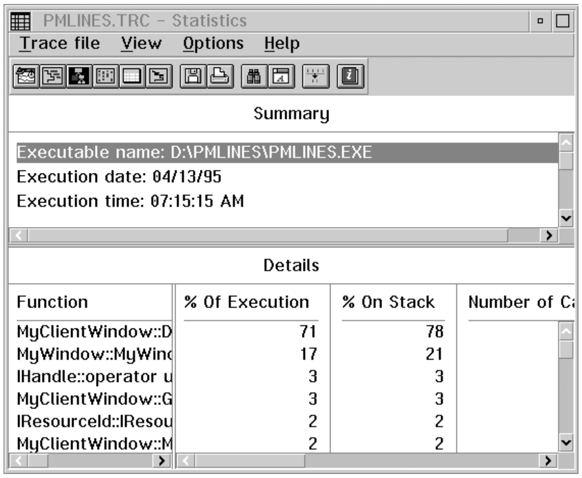
Execution Density Vertical Ruler

The **Vertical Ruler**, located to the left of the diagram, shows scale. To change the scale, select the **Scale pages** choice from the **View** menu, and the **Scale Pages** window appears.

40

Statistics Diagram

The **Statistics** diagram gives you a textual report of execution time by function or executable. Use this information to find hot spots in the overall program execution.



Executable name: D:\PMLINES\PMLINES.EXE
Execution date: 04/13/95
Execution time: 07:15:15 AM

Function	% Of Execution	% On Stack	Number of Ca
MyClientWindow::D	71	78	
MyWindow::MyWinc	17	21	
IHandle::operator u	3	3	
MyClientWindow::G	3	3	
IResourceId::IResou	2	2	
MyClientWindow::M	2	2	

Figure 128. Statistics Diagram

This diagram has two panes: Statistics Summary Pane and Statistics Details Pane.

You can resize the window horizontally and the panes vertically. To resize a window or window pane, press and drag the mouse pointer on the split bar until the windows or window panes are the size you want.

Note: Times are shown in milliseconds.

Areas of the Statistics Diagram

The the following topics describe the areas of the **Statistics** diagram.

Statistics Menu Bar Summary

The menu choices in the **Statistics** diagram are as follows:

Trace file Allows you to perform functions with your existing trace file. From this menu, you can select:

Statistics Diagram

Open as
Displays a cascaded menu with the following choices. Select the name of the diagram you want to view.

- Call Nesting
- Dynamic Call Graph
- Execution Density
- Statistics
- Time Line

Save as text
Creates a file containing the summary information.

Printer settings
Allows you to select a printer and various options for your printed output.

Print
Prints the textual report of the **Statistics** diagram.

Window manager
Displays the **Performance Analyzer - Window Manager** window.

Exit the Performance Analyzer
Exits the Performance Analyzer application.

View Allows you to change displayed information. From this menu, you can select:

Details on
Select to view the **Details** window by function or executable.

Include threads
Controls the threads to include or exclude in the diagram.

Sort
Select how you want to sort the **Details** window.

Options Allows you to customize the **Statistics** diagram and display additional information. From this menu, you can select:

Find
Searches for a function or an executable.

Font
Selects the font and font size for the **Summary** and **Details** panes.

Tool bar

Displays a cascaded menu that lets you select:

Show Choose to either show or hide the Tool bar.

Hover Choose to either enable or disable displaying the help text when the mouse pointer hovers over the Tool bar buttons.

Settings

Displays a cascaded menu that lets you either save the current settings or restore the initial default settings.

Save

Save the current session settings.

Restore initial defaults

Restore the original settings.

Project This menu appears when you start the Performance Analyzer from within the WorkFrame environment. From this menu, you can select:

Edit function

Displays the source code for the selected function in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Statistics** diagram will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Help This menu provides choices that display various types of Help information. From this menu, you can select:

Help index

Displays an index of Help topics.

General help

Displays Help for the active window.

Using help

Describes how to use Help.

How do I?

Displays task Help.

Product information

Displays information about the Performance Analyzer.

Statistics Diagram

Statistics Summary Pane

The **Summary** pane is in the top area of the window. You must scroll the window panes to view all of the information. Information provided in the **Summary** pane is as follows:

- Executable name
- Trace file description

  Note: This will only appear if you entered a description on the **Create Trace** window in the **Trace File Description** field when you created the trace file.
- Execution date
- Execution time
- Number of executables generating events
- Number of functions generating events
- Number of threads generating events
- Total number of events
- Total number of annotations
- Number of user events
- Maximum call nest depth
- Number of trace buffer flushes
- Total trace time excluding overhead
- Trace overhead

Statistics Details Pane

The **Details** pane is in the bottom area of the window. You must scroll the window panes to view all of the information. The **Details** pane has a left and right pane. The left pane displays the fully qualified name of the component that the statistics have been gathered on.

Information provided on the left side of the Details pane is as follows:

When you have selected **Functions** from the **View** and **Details on** menus, you see:

- Function
- Object file
- Executable

When you have selected **Executables** from the **View** and **Details on** menus, you see:

- Executable

With function names, if user events have been included in the trace, they will appear as separate entries in the list of function names.

The user events will be the function name that made the call to the user event, followed by the user event in parentheses.

For executables, only the executable name is displayed.

Information provided on the right side of the Details pane is as follows:

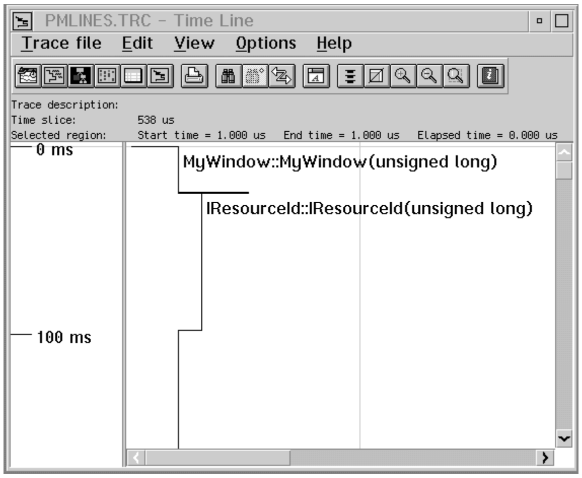
- Percent Of Execution
- Percent On Stack
- Number of Calls
- Execution Time
- Time on Stack
- Minimum Call
- Maximum Call
- Average Call

Note: If you disabled the **Time stamp events** choice before you created your trace file, only the *Number of Calls* column will be displayed on the right side of the **Details** pane.

Statistics Diagram

41. Time Line Diagram

The **Time Line** diagram displays the sequence of nested function calls and returns. Time stamps determine the exact placement of an event along the time dimension on the vertical axis. This provides a direct and natural presentation of the chronological relationships of events.



```
Trace description:
Time slice:          538 us
Selected region:     Start time = 1.000 us   End time = 1.000 us   Elapsed time = 0.000 us
```

```
MyWindow::MyWindow(unsigned long)
IResourceId::IResourceId(unsigned long)
```

Figure 129. Time Line Diagram

The **Time Line** diagram is similar to the **Call Nesting** diagram, but the distance between successive events in the diagram are drawn in proportion to the actual time between the events as recorded in the trace file.

The names of functions are only drawn when the time spent in that function is large enough to allow the name to be drawn. This value is dependent upon the size of the font being used. For example, with the default font, the distance between the call to the function and the next event must be at least twenty scan lines. This is done to ensure that the function name will not be overwritten by another function name or overdrawn by a line representing a function call or thread switch.

You can use this diagram to find where a deadlock occurred. Access violations, system exceptions and other such program errors are recorded in the trace file as *user events*, as are any messages generated in the code by calls to the Performance Analyzer. These events are indicated by a black diamond in the diagram, and if there is sufficient space, the text associated will be drawn to the right of the events.

Time Line Diagram

Areas of the Time Line Diagram

The following topics describe the areas of the **Time Line** diagram.

Time Line Menu Bar Summary

The menu choices in the **Time Line** diagram are as follows:

Trace File Allows you to perform functions with your existing trace file. From this menu, you can select:

Open as
Displays a cascaded menu with the following choices. Select the name of the diagram you want to view.

- Call Nesting
- Dynamic Call Graph
- Execution Density
- Statistics
- Time Line

Printer settings
Allows you to select a printer and various options for your printed output.

Print selected region
Prints a selected area of the diagram.

Window manager
Displays the **Performance Analyzer - Window Manager** window.

Exit the Performance Analyzer
Exits the Performance Analyzer application.

Edit Allows you to locate and change text in the **Time Line** diagram. From this menu, you can select:

Find function
Search for a function call or return.

Find Next
Find the next occurrence of the last item you searched for.

Select time
Go to a specific time in the diagram.

Select time range
Select a specified time in the diagram.

Select all
Select the entire **Time Line** diagram.

View
Allows you to change displayed information. From this menu, you can select:

Zoom in
Magnifies a region of interest in the diagram.

Zoom out
Reduces the diagram to starting size.

Zoom to selected range
Magnifies the diagram to focus on the highlighted area.

Scale pages
Selects the size of the diagram.

Options
Allows you to customize the **Time Line** diagram and display additional information. From this menu, you can select:

Call stack
Shows all functions on the call stack at a selected point.

Correlation
Synchronizes other diagrams to display the same highlighted region.

Font
Selects the font, font style, and font size for the diagram.

Thread layout
Selects the indentation amount for each thread column and whether to draw separator bars between threads.

Status area
Controls the gray area at the top of the window.

Tool bar
Displays a cascaded menu that lets you select:

Show Choose to either show or hide the Tool bar.

Hover Choose to either enable or disable displaying the help text when the mouse pointer hovers over the Tool bar buttons.

Settings
Displays a cascaded menu that lets you either save the current settings or restore the initial default settings.

Save
Save the current session settings.

Restore initial defaults
Restore the original settings.

Time Line Diagram

Project This menu appears when you start the Performance Analyzer from within the WorkFrame environment. From this menu, you can select:

Edit function
Displays the source code for the selected function in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Time Line** diagram will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Help This menu provides choices that display various types of Help information. From this menu, you can select:

Help index
Displays an index of Help topics.

General help
Displays Help for the active window.

Using help
Describes how to use Help.

How do I?
Displays task Help.

Product information
Displays information about the Performance Analyzer.

Time Line Status Area

The **Status Area**, located at the top of the window, describes the settings of the diagram. You can select the **Status area** choice from the **Options** menu to change the appearance of the **Status Area**. When you select **Status area**, you can select the following in the **Status Area** window:

Trace description
Shows the description given to the trace file.

Time slice
Displays the value of the time slice.

Selected region
Displays the start, end, and total time of a selected region.

Time Line Pop-up Menus

The pop-up menus allow you to quickly access features that are frequently used. The **Time Line** diagram has two pop-up menus: the **Time Line Diagram** pop-up menu and the **Time Line Selected Item** pop-up menu.

Time Line Diagram Pop-up Menu

This pop-up menu contains almost all the choices from the **Edit** and **Options** menus. To access this pop-up menu, click mouse button two on the background area of the diagram. The menu is displayed with the following choices:

Find function
Search for a function call or return.

Find next
Find the next occurrence of the last item you searched for.

Zoom in
Magnifies a region of interest in the diagram.

Zoom out
Reduces the diagram to starting size.

Scale pages
Selects the size of the diagram.

Font
Selects the font, font style, and font size for the diagram.

Thread layout
Selects the indentation amount for each thread column and whether to draw separator bars between threads.

Time Line Selected Item Pop-up Menu

To access this pop-up menu, highlight a region of interest, move the mouse pointer into that area, and click on mouse button two. The menu is displayed with the following choices:

Zoom to selected range
Magnifies the diagram to focus on the highlighted area.

Call Stack
Displays all the functions currently on the call stack from a selected point.

Correlation
Synchronizes other diagrams to display the same highlighted region.

Edit function
Displays the source code for the selected function in the default editor for WorkFrame's edit action.

WorkFrame actions that can be launched from the **Time Line** diagram will appear in this menu. To have your program appear in this list, you must first associate your program with a **Type Name** of **EXE** in the **WorkFrame Tool Setup** window.

Time Line Vertical Ruler

The **Vertical Ruler**, located to the left of the diagram, shows scale. To change the scale, select the **Scale pages** choice from the **View** menu, and the **Scale Pages** window appears.

Part 8. Browsing Programs and Libraries

As a C++ programmer using the object-oriented paradigm, you must deal with large and sometimes complex collections of interrelated classes. A class' behavior is not only defined by its own data and function members, but also by all the behavior of its base classes. The VisualAge C++ Browser is a tool to help you understand and use these classes and their relationships.

42 Overview

The VisualAge C++ Browser is a Presentation Manager (PM) tool for examining programs developed in VisualAge C++.

This chapter briefly describes the Browser, its major features and concepts, and explains how you access the online contextual help and **How Do I?** information while you are using the Browser.

Understanding the Browser

The graphical user interface consists of two types of Browser windows:

- A *List window* displays a list of program elements, such as source files, functions, and classes.
- A *Graph window* displays program relationships in a graphical format, for example, inheritance in a class hierarchy. You can specify the level of detail you want the graph to show, scroll over the graph, zoom in and out, and select program elements directly from the graph.

📖 For more information on List and Graph windows or other elements of the Browser user interface, see Chapter 44, “Understanding and Using the Browser User Interface” on page 563.

You can view your program files (.DLL, .EXE, .LIB) and compiler generated Browser (.PDB) files. In addition, if you have a IBM WorkFrame project, you can view your programs, without recompiling, by using the Browser QuickBrowse facility. 📖 See “Using QuickBrowse” on page 615 for more information on QuickBrowsing your programs.

With the Browser, you can look at your source code in many different ways:

- List program objects by type (for example, all classes), by content (members of a class), or by components (all files). 📖 See “Ordering the Contents of a Container View” on page 566 for more details.
- View relationships between program components graphically, such as class-inheritance hierarchies, function calls, and included files. 📖 See “Using the Browser to Aid Program Understanding” on page 604 for more information.
- View and edit the actual source code associated with a selected program element using an editor. By default, the Browser uses the VisualAge Editor, but you can select to use a different editor by changing your IBM WorkFrame project

settings. For more information on viewing and editing with the Browser, see “Editing and Viewing Source Files” on page 601.

- View online documentation for a class or class member. For more information on viewing online documentation through the Browser, see “Showing VisualAge C++ Open Class Library Documentation” on page 603.

Concepts Used by the Browser

Browsing Browsing is a navigational technique for understanding the complexities inherent in a set of classes. The VisualAge C++ Browser helps you navigate through an inheritance hierarchy, to understand the full interface of a class available to you as a programmer, to locate the body of a function or a class definition amongst dozens (possibly hundreds) of files across multiple directories, to understand calling relationships between functions, and much more.

Browser Database The Browser does not use an external database, but has an internal representation of your program. This representation reflects all the needed information for the actions that the Browser performs. This database can be populated from `.PDB` files (the richest source of information) or directly via the QuickBrowse facility in the Browser. `.PDB` files are generated when you run the compiler with the `/Fb` option. In addition, you can populate the Browser database from .DLL, .EXE, or .LIB files when you have also run the linker with the `/BROWSE` option. The Browser will save the contents of these files upon exit to a .PDD, .PDE, or .PDL Browser database file.

When you next load one of the program files, the Browser will use the stored Browser database file, updating it if you have modified your program since the last load. The Browser database files make loading your programs quicker. For more information on Browser database files, see “Creating Files to Use with the Browser” on page 559. For more information on QuickBrowse, see “Using QuickBrowse” on page 615.

Container View A container view is a list which can be further expanded using the + and - icons to expand and collapse the entries. For more information on this type of view, see “Types of List Windows” on page 565.

Objects The Browser presents program elements to you in the form of objects on the screen. Objects can be classes, types, functions, variables, and source files. Use Mouse Button 2 on any of these objects to invoke a PopUp menu of actions available for that object. For detailed descriptions of the various objects, see “Browsing List Objects” on page 567.

Object-Action Pairs You can perform actions on any Browser object from the object's PopUp menu. Use the Mouse Button 2 over the object to invoke the PopUp menus. The Browser remembers the last 40 object-action pairs and lists them in the **History** window for quick access. 🕮 "The History Window" on page 599 gives you more information on using the **History** window.

The List and Graph window each have an **Action Status Bar** which is located directly below the menubar. It indicates what object and action were used to create the current contents of the window.

Using the Mouse

The Browser makes use of both the Mouse Button 1 (usually the left mouse button) and Mouse Button 2 (usually the right mouse button).

Use **Mouse Button 1** for selecting objects in the windows, for scanning the Browser PullDown menus, and for sizing windows and selection areas.

Use **Mouse Button 2** for performing actions on the various objects displayed in the windows. Each type of object has several associated actions in the form of a PopUp menu.

🕮 See "Browsing List Objects" on page 567 for more information on the types of objects available through the Browser.

Getting Help While You Are Using the Browser

In addition to this guide, there is:

- Contextual online help, which gives you help from within the Browser.
- **How Do I?** help, which gives you step-by-step instructions on how to perform several Browser related tasks.

Using Contextual Help

You can ask for help on any menu choice, window, or entry field on how to use a particular item by accessing the online context-sensitive help. You can access it in one of the following ways:

- Select a choice from the **Help** PullDown menu.
- Press **F1** from any Browser window.
- Press **F1** while highlighting any menubar item.
- Press **F1** while highlighting any PullDown menu item.
- Press **F1** or select the **Help** PushButton on any dialog or NoteBook.

Browser: Getting Help

Using the How Do I... Information

The **How Do I?** information can help you quickly accomplish tasks when you are unclear on what to do next.

You can access the **How Do I?** information in a number of ways:

- Select the **Help** PullDown menu and select the **How Do I...** menu item from the Browser.
- Select the **Help** PullDown menu from any component of VisualAge C++, select the **How Do I... Selections** ▸ Cascade menu, and select the **Browser** menu item.
- Open the **Information** folder located in the main VisualAge C++ folder on your desktop. Open the **How Do I?** folder and select the **Browser How Do I?** information.

43

Getting Started

This chapter tells you how to start and close the Browser, and how to generate Browser database files.

Starting the Browser

You can start the Browser from four different places:

- The OS/2 command prompt.
- The OS/2 Workplace Shell.
- The IBM WorkFrame environment.
- The VisualAge C++ Debugger and Editor.

If the program you want to browse is large, the Browser may take several seconds to load. A **Progress** dialog will appear to inform you of the progress. You can see this dialog in Figure 155 on page 594.

From the OS/2 command prompt

To start a Browser session from the OS/2 command prompt, use the **icsbrs** command as follows:

```
►►──icsbrs──┬─────────────────────┬──►◄
            ├─file_name───────────┤
            │ ┌─────────────────┐ │
            └─▼─pdb_file_name───┴─┘
```

Where:

file_name Can be a Browser database file (with extension `.PDB`, `.PDE`, `.PDD`, `.PDL`) or a program file (with extension `.EXE`, `.DLL`, `.LIB`).

pdb_file_name Can be multiple .PDB file names. You can load multiple .PDB files into the Browser. You cannot load any other combination of files into the Browser, however, you can merge multiple file types. See "Merging Files" on page 595.

If you type **icsbrs** without any options, the Browser starts without any loaded files. From the **File** PullDown menu:

- Select the **Load...** menu item in order to launch the **Load Database** dialog. From this dialog, you can enter a program file name with extension .DLL, .EXE or .LIB, or a Browser database file with extension .PDB, .PDD, .PDE, or .PDL, or

- Select one of the libraries that make up the VisualAge C++ Open Class Library directly from the **Load** ► Cascade menu found on the **File** PullDown menu.

📖 For more information on loading, see “Loading Files into the Browser” on page 593.

From the OS/2 Workplace Shell

You can start the Browser from the OS/2 Workplace Shell in three ways:

- Double-click on the Browser icon. You can load a database file name or a program file name from the **Load Database** dialog which you can invoke from the **File** PullDown menu. 📖 For more information on loading, see “Loading Files into the Browser” on page 593.
- Double-click on a Browser database file (`.PDB`, `.PDD`, `.PDE`, or `.PDL`) from a Workplace Shell or WorkFrame folder.
- Drag a Browser database file (`.PDB`, `.PDD`, `.PDE`, or `.PDL`) or program file (`.DLL`, `.EXE`, or `.LIB`) or IBM WorkFrame project icon onto the Browser icon.

📖 For a description of a Browser database, see “Concepts Used by the Browser” on page 554. For information on creating database files, see “Creating Files to Use with the Browser” on page 559.

From the IBM WorkFrame environment

To start the Browser from the IBM WorkFrame environment, you can double-click on a `.PDB`, `.PDE`, `.PDD`, or `.PDL` Browser database object in any WorkFrame project folder, or you can select **Browse** from any **WorkFrame Project** PopUp menu to load a Browser session. You can also select the **Browse** action on any `.PDB`, `.PDD`, `.PDE`, `.PDL`, `.EXE`, `.DLL`, or `.LIB` file.

📖 For information on creating database files, see “Creating Files to Use with the Browser” on page 559. For more information on creating a project, setting options, and starting tools from the WorkFrame environment, see the Part 1, “Developing with WorkFrame” on page 1.

From the VisualAge C++ Debugger or Editor

To start the Browser from the Debugger:

1. Select the **Project** PullDown menu from the Debugger user interface.
2. Select the **Browse** Cascade menu.
3. Select a Browser action to perform on the program currently loaded into the Debugger or on a selected program element.

To start the Browser from the VisualAge Editor:

1. Select the **Browser** PullDown menu from the Editor user interface.
2. Select an action to perform on the file currently loaded into the VisualAge Editor or on a selected program element.

Creating Files to Use with the Browser

The Browser can view either Browser database files (.PDB, .PDD, .PDE, .PDL) or programs files (.DLL, .EXE, .DLL). The following describe how to compile and link your source files, and how the Browser generates Browser database files from your loaded source. For a description of what a Browser database file is, see "Concepts Used by the Browser" on page 554.

Created by Compiling

The compiler will generate .PDB Browser database files (one per compilation unit), if you compile your source with the /Fb option. This provides the Browser with the richest source of information. However, there are two options that you can use to compile your programs for use with the Browser: **/Fb** or **/Fb***.

The difference between the two options relates to how much Browser information is generated from system include files. That is, those included via:

```
#include <system_file.h> as opposed to:
#include "local_file.h".
```

The Generate Browser information option (**/Fb**) discards much of the non-type information from inside of system include files. For instance:

- Non-member function declarations will not be included in the .PDB file, including those C and OS/2 header files. Any friendship granted to these types of omitted functions will not be recorded for a class. For instance:

  ```
  // in <foo.h>
    int foobar(void);
  // in "bar.h"
    class bar {
    friend int foobar(void);
    }
  ```

  This friendship will not be included in the list of friends of class bar.

- No global variable declared, or defined, in the system header file will be included in the .PDB file. This includes variables of an instantiated template type.
- Class member function declarations will be added to the .PDB file, but their inline definitions, if any, will not.
- Non-inline function definitions will not be added to the .PDB file.

These restrictions are lifted when compiling with the Generate All Browser Information (/Fb*) option. It is recommended that you use the /Fb option, unless the compiler issues a message indicating that the use of the /Fb* option is appropriate.

Created by Linking

In addition, you can populate the Browser database from .DLL, .EXE, or .LIB files when you have run the linker with the `/BROWSE` option. Note that the linker will create additional .PDB files for template instantiations from `TEMPINC`.

Created by the Browser

Upon exit, the Browser will save the contents of the .DLL, .EXE, and .LIB files to Browser database files. For example, you can load an .EXE file:

```
icsbrs wombat.exe
```

or enter the program file name into the **Load Database** dialog. For more information on loading, see “Loading Files into the Browser” on page 593. When you view a program file for the first time, the Browser generates a database file with an extension corresponding to the extension of the file you are viewing.

File Extension	Database File Extension
.EXE	.PDE
.DLL	.PDD
.LIB	.PDL

In the example given above, the Browser would generate the file `wombat.pde`.

Note that this single file contains all the information linked from every compilation unit's .PDB files, in the same way that the .EXE is the linked composite of all its compilation unit's .OBJ files.

If you invoke the Browser with a program file for which the Browser has already generated a Browser database file, the Browser uses that database file instead of generating a new file. If the file has been updated, the Browser updates the corresponding database file. This is called an “incremental smart build”. For example, the next time you view `wombat.exe`, the Browser uses the `wombat.pde` file that it created, thus reducing the time it takes to load the Browser with the `wombat.exe` information. If you have made changes to `wombat.exe`, these changes are loaded from the new .PDB files. The incremental smart build reduces the loading time of the new information.

Closing the Browser

To end the Browser session, select **Exit Browser** from the **File** PullDown menu in any List or Graph window, or press **F3**. A dialog appears asking if you want to exit your current Browser session. Select **Yes** to exit or **No** to return to your Browser session. You can prevent being prompted by this dialog by selecting the **Browser...** menu item from the **Options** PullDown menu. Deselect the **Confirm on exit** CheckBox. For more information on the **Browser Settings** NoteBook, see "Changing Browser Settings" on page 587.

When you exit the Browser, the currently loaded Browser database is saved to a Browser database file. By default, the Browser names these files based on the file name and adding either the .PDD, .PDE, or .PDL extension, depending on whether the file was a .DLL, an .EXE, or a .LIB, respectively. If the Browser cannot determine what the database file should be named, a message may prompt you to name your Browser database file. Also, if you performed a merge or loaded one or more .PDB file, the Browser will ask you to give the Browser database file a name.

Browser: Closing the Browser

44 Understanding and Using the Browser User Interface

This chapter describes the Browser windows and dialogs. The descriptions cover what each window or dialog is for, and how you setup and use them. For more details on using the Browser to understand and develop your applications, see Chapter 45, "Using the Browser" on page 601. For a quick tour of the Browser features, try Chapter 46, "A Tour of the Browser" on page 621.

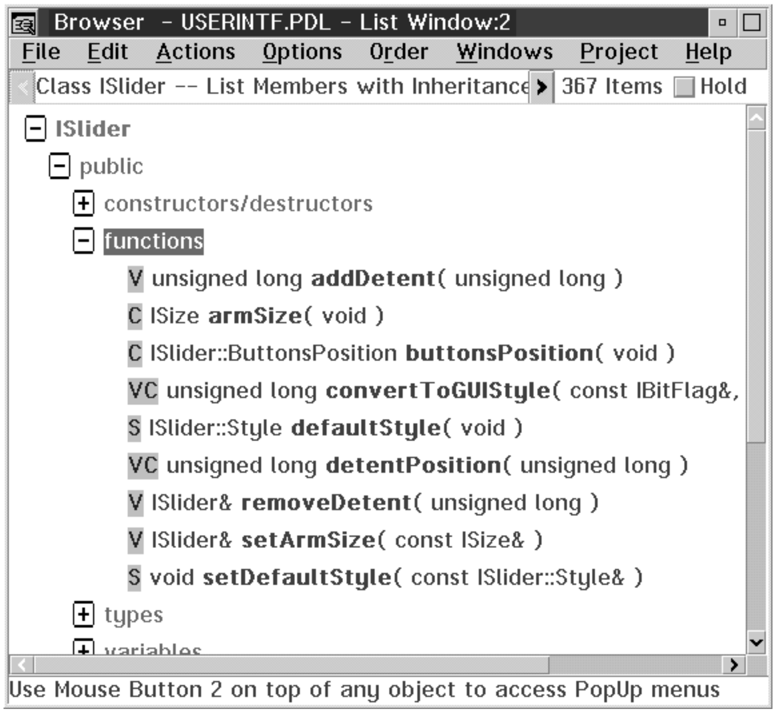
The following are described in this chapter:

- "The List Window" on page 564.
- "The Graph Window" on page 576.
- "Changing Browser Settings" on page 587.
- "Changing Fonts" on page 592.
- "Loading Files into the Browser" on page 593.
- "Merging Files" on page 595.
- "Finding Objects in the Current Window" on page 597.
- "Searching for Objects in the Entire Browser Database" on page 598.
- "The History Window" on page 599.

The List Window

The List window displays a list of program elements such as classes, functions, and source files that were used to generate your program. Many aspects of a program can be described as a list. Other aspects can be described as relationships in a Graph window described on page 576.

To view your programs with the Browser, load your program files (.DLL, .EXE, or .LIB) into the Browser using the **Load Database** dialog described on page 593.



Class ISlider -- List Members with Inheritance

```
[-] ISlider
  [-] public
    [+] constructors/destructors
    [-] functions
        V unsigned long addDetent( unsigned long )
        C ISize armSize( void )
        C ISlider::ButtonsPosition buttonsPosition( void )
        VC unsigned long convertToGUIStyle( const IBitFlag&,
        S ISlider::Style defaultStyle( void )
        VC unsigned long detentPosition( unsigned long )
        V ISlider& removeDetent( unsigned long )
        V ISlider& setArmSize( const ISize& )
        S void setDefaultStyle( const ISlider::Style& )
    [+] types
    [+] variables
```

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 130. List window showing a container view

The List window consists of:

- A main menu bar whose PullDown menu actions affect the current window. See “PullDown Menus” on page 640 for detailed descriptions of the PullDown menus.
- An **Action Status Bar** which describes what object and action produced the currently viewed list. A definition of *object-action pair* is described on page 555. In the above figure, the **Action Status Bar** is “Class ISlider - List Members with Inheritance”.

- A count that indicates how many program elements are in the current list, excluding label objects. In the above figure, there are 367 program elements listed.
- A **Hold** CheckBox that, when checked, keeps the current List window contents from being replaced when an object-action pair results in a list. This can be useful, for instance, if you want to keep a list of all classes available at the same time that you want to focus on one given class. If the **Hold** CheckBox is not checked (the default behavior), the results of the next action overwrites the current window contents. There is a limit of four List windows.
- A **List Area** where your program objects are listed. You can perform actions on the contents of the **List Area** using Mouse Button 2 on any of the listed items or on the background of the List window. You can also print, save to a file, or copy to the clipboard the contents of the List window. In the above figure, the **List Area** is displaying a container view. For a description of what a container view is, see "Types of List Windows."
- An **Information Bar** at the bottom of the window quickly defines the currently selected menu item or explains how to invoke the Object PopUp menus. You can hide the **Information Bar** by selecting the **Expert** help level in the **Browser Settings** dialog. See "Changing Browser Settings" on page 587 for more information.

The maximum number of items you can list in a List window is 65533.

The List window has a **Background** PopUp menu you use to perform actions on the window. You can access it using Mouse Button 2 on the background of the List window. See "PopUp Menu Items for List and Graph Windows" on page 648 for detailed descriptions of the window PopUps.

Types of List Windows

The first List window that you see after loading a program into the Browser displays all the classes defined in the loaded Browser database. This is referred to as a *straight view*. The Browser has one other type of List view, a *container view* to list:

- All members of a class and its base classes (**List Members with Inheritance** on a class or **List Class Members with Inheritance** on a function),
- All defined objects in a file (**List Defined Objects** on a file),
- All friends of a class (**List Friends** on a class), or
- All immediate callers and callees of a function (**List Immediate Callers & Callees** on a function).

All container views are expandable. (See Figure 130 on page 564 for an example of a class and its base classes container view). You can use the + and - icons to expand and collapse the current selection one level, or use the **F7** and **F8** keys to expand and collapse the entire contents of the window.

Browser: List Window

Ordering the Contents of a Container View

When you perform a **List Members with Inheritance** action on a class or a **List Class Members with Inheritance** on a function, the resultant list is a container view. You can rearrange the contents of this particular kind of container view. The container views that you get when you perform the **List Defined Objects**, **List Immediate Callers & Callees**, or **List Friends** actions only have one level of expansion, so it does not make sense to order them based on class, access or type.

You can order the contents of the container view resulting from a **List Members with Inheritance** or **List Class Members with Inheritance** action by:

Class Ordered by class, access, then type:

```
- Class 1
  - Public
    + Constructors/Destructors
    + Functions
    + Types
    + Variables
  + Protected
  + Private
+ Class 2
```

Access Ordered by access, type, then class:

```
- Public
  + Constructors/Destructors
  - Functions
    + Class 1
    + Class 2
  + Types
  + Variables
+ Protected
+ Private
```

Type Ordered by type, access, then class:

```
+ Constructors/Destructors
- Functions
  + Public
  - Protected
    + Class 1
    + Class 2
  + Private
+ Types
+ Variables
```

By default, the List window displays a container view with the classes at the highest level (the **Class** order). If you are interested in seeing particular program elements (like all functions used by your program) from various classes or particular access methods (like what is defined as public), it can become quite annoying to have to scroll back and forth in the list between the classes. The Browser eliminates this annoyance by giving you the ability to reorder the list contents. You can view all the program elements together by type or together by access method, eliminating the need for constant scrolling of the list contents.

Note: The resultant list when you perform the **List Members with Inheritance** or **List Class Members with Inheritance** action is not sorted alphabetically as items in a List window generally are. The classes are arranged in a depth first tree traversal of the classes inheritance hierarchy.

Browsing List Objects

The Browser presents program elements to you in the form of objects on the screen. Objects can be classes, types, functions, variables, and source files. In addition, the Browser uses Label objects to organize the program elements on the screen. You can click Mouse Button 2 on any of these objects to invoke a PopUp menu of actions available for that object.

Class Objects

C++ classes, structs, unions, class templates, class template instantiations, classes defined at file (or global) scope, and those nested inside of other classes. Classes defined inside of function bodies are not included.

Class objects can be specialized as SOM classes. A SOM class is any class that derives from SOMObject, and a SOM metaclass is any class derived from SOMClass. C++ Direct-To-SOM classes are supported by the Browser. Bindings from the C++ SOM emitter may also be browsed, but the symbols produced by the emitter may not be easy to read or understand.

Function Objects

Functions, member functions, function templates, function template instantiations, and SOM methods. Class member functions defined inside of function bodies are not included.

Variable Object

Variables, class member variables, class template member variables, member variables of class template instantiation, and SOM data members. Not included are variables defined inside of function bodies.

Type Objects

Typedefs and enums. Not included are enumeration values, nor those enums and typedefs declared inside of function bodies, or built-in C types, such as int or char, or any pointer or reference combination of these types. Classes, structs, and unions are referred to as class objects.

File Objects

Source files used to create your loaded program.

Label Objects

When you perform either the **List Members with Inheritance**, **List Friends**, **List Class Members with Inheritance**, **List Defined Objects**, or **List Immediate Callers & Callees** action, the results are placed in a List window container view. In this type of view, Label objects are used to organize the results in the List window by category. Categories can be nested. The categories are Public, Protected, Private, Constructors/Destructors, Classes, Functions, Types, Variables, Callers, and Callees. Also, the results displayed when you expand a typedef are label objects.

To browse the program elements listed in the List window, use the Object PopUp menus. You can invoke them by selecting an object and clicking on Mouse Button 2. See "Object PopUp Menu Items" on page 649 for detailed descriptions of the Object PopUp menus.

Understanding Browser Generated Flags

Sometimes you will see one of the following three flags when viewing items in the List window:

[anonymous]

The C language has the concept of anonymous structs, unions and enums which C++ has inherited. For instance, each of the types below has no name (or is anonymous):

```
struct {
  int number;
  int code;
  char *name;
} record;
union {
  int a;
  char b;
};
enum { red, yellow, green } color_1, color_2;
```

[compiler generated]

When you define a class, but do not define your own default constructor, copy constructor, or destructor, the compiler will quietly generate one for you. We thought it important to let you know in the Browser when this occurs.

('n' instances)

When you use multiple inheritance and two (or more) of your base classes inherit from a common class such as the following:

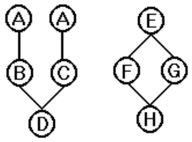


Figure 131. ('n' instances) example

Non-virtual inheritance was used by B and C when they inherited from A, but virtual inheritance was used by F and G when they inherited from E. An object of type D will contain two copies of the data contained in an object of type A. An object of type H will contain only one copy of the data contained in an object of type E.

To highlight this non-diamond shape inheritance structure to you, the Browser indicates when a derived class contains more than one copy of the data from a base class.

Changing the Default List Window Settings

The Browser provides you with a **List Windows Settings** NoteBook from which you can change:

Settings Used to change the **Action Status Bar** font, the initial action performed when you load a file into the Browser, and the double-click actions of the objects displayed in the List window. 📖 See “Changing List Settings” on page 570.

Colors Used to change the colors of objects in the List window. 📖 See “Changing Colors Used by the List Window” on page 571.

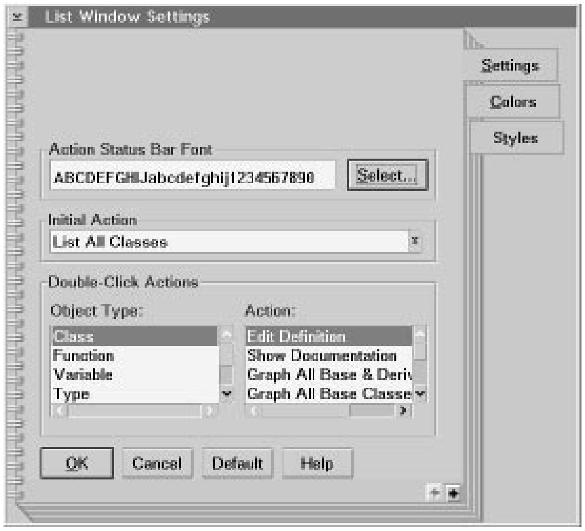
Styles Used to change the amount of text displayed by the List window. 📖 See “Changing the List Style” on page 574.

Browser: List Window Settings

Changing List Settings

You can change the **Action Status Bar** font, the initial action performed when you load a file into the Browser, and the double-click actions of the objects displayed in the List window.

To load the **List Window Settings** NoteBook, select **List Window...** from the **Options** PullDown menu.



List Window Settings

Action Status Bar Font: ABCDEFGHIJabcdefghij1234567890

Initial Action: List All Classes

Double-Click Actions

Object Type:
- Class
- Function
- Variable
- Type

Action:
- Edit Definition
- Show Documentation
- Graph All Base & Deriv
- Graph All Base Classe

Figure 132. List Window Settings NoteBook Settings Page

You can change the font used by the **Action Status Bar** independently of the font used in the **List Area** of the List window. To change the font of the **Action Status Bar** the List window, choose the **Select...** PushButton to load the **Action Status Bar Font** dialog. 🕮 For more information on the **Fonts** dialog, see “Changing Fonts” on page 592.

You can change the initial action performed when you load a program into the Browser. Use the **Initial Action** DropDown to select the action to perform when loading a program into the Browser from the List window. By default, the List window displays all the classes defined in your loaded program. You can choose from **List All Classes**, **List All Files**, or **None**.

Use the **Double-Click Actions** section to change the default action performed when you double-click on an object in the List window. By default, the double-click actions for all objects, except Labels, is to edit the object's definition. Select the object from the **Object Type:** ListBox, and select a corresponding action from the **Actions:** ListBox.

Note: These settings are independent of the selections made for the Graph window settings. See “Changing Graph Settings” on page 581.

Changing Colors Used by the List Window

You can identify objects in the List window by their color. By default, the following colors are used:

Object	Color
Class	Cyan
SOM Class	Dark Green
SOM Metaclass	Light Green
Function	Blue
Type	Blue
Variable	Blue
File	Red
Label	Brown

In addition, the following letters, displayed in light blue by default, are used to further indicate the attribute of a function, type or variable:

C	Constant (functions)
V	Virtual (functions)
E	Enumerator (types)
S	Static (functions and variables)
PV	Pure Virtual (functions)

Note: You can expand the attributes into their complete name using the **List Window Settings** NoteBook **Styles** page which is described in “Changing the List Style” on page 574.

Browser: List Window Settings

You can customize the colors used in the List window because:

- You may prefer different color schemes.
- Different monitor resolutions, or color palettes may make the default Browser color choices indistinguishable.
- Changing the colors can make the types of objects that you are interested in stand out more and tone down those that you are not interested in.
- When printed, the results may look better when different colors or fonts are used. The results may be different for the List window and Graph window views, so you have the ability to change them for each type of window.

Note that making a color change will affect all open List windows and all subsequently opened List windows. The new defaults are saved to the Browser profile (`icsbrs.ini`) when you exit from the Browser.

To change the colors used by the List window, select **List Window...** from the **Options** PullDown on a List window to start the **List Windows Settings** NoteBook. Select the **Colors** tab.

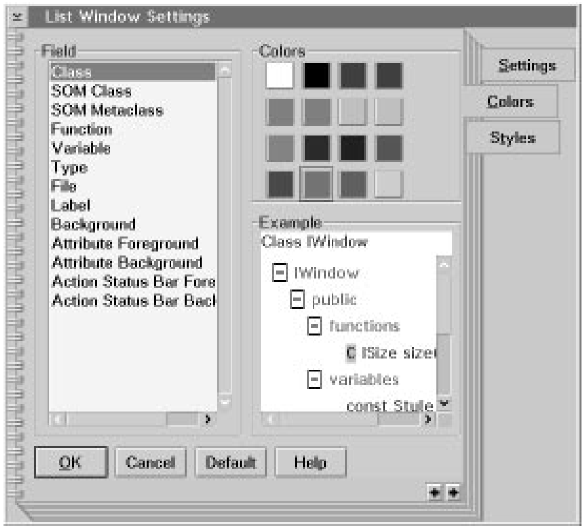


Figure 133. List Window Settings NoteBook Colors Page

On the left side of the page is a scrollable list of items for which you can set colors. On the upper right side is the color palette containing 16 colors to choose from. On the bottom right side is an **Example Area** that will give you a preview of the color selection that you are currently editing.

To change an object's color:

1. Select the object from the scrollable list.
2. Click on the color from the 16 available colors. Your change appears in the **Example Area**.
3. Select the **OK** PushButton if you want to apply your color changes to the List Window, or select the **Cancel** PushButton if you want to exit without making any changes.

The **Default** PushButton restores the default colors used by the Browser List windows. In addition to the default colors used by the objects in the List window (mentioned above), you can change:

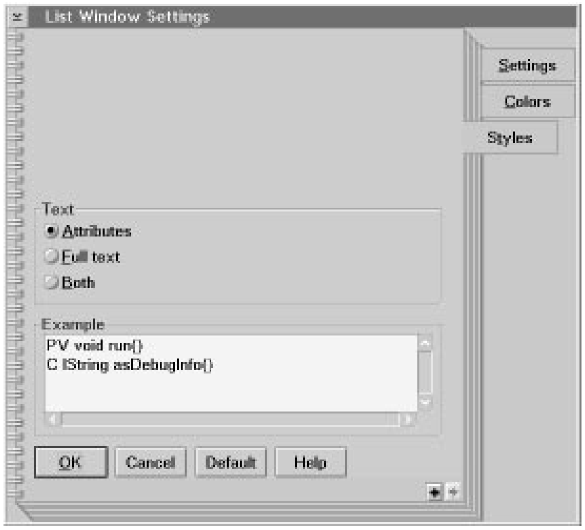
Window Attribute	**Color**
Background	White
Attribute Foreground	Dark Red
Attribute Background	Light Blue
Action Status Bar Foreground	Blue
Action Status Bar Background	White

Note: These settings are independent of the selections made for the Graph window color settings. 📖 See “Changing Colors Used by the Graph Window” on page 582.

Browser: List Window Settings

Changing the List Style

You can change the amount of text that is displayed in the List window. To change the List window text style, select **List Window...** from the **Options** PullDown menu to start the **List Window Settings** NoteBook. Select the **Styles** tab.



● Attributes
○ Full text
○ Both

```
PV void run()
C IString asDebugInfo()
```

Figure 134. List Window Settings NoteBook Style Page

There are three text style settings:

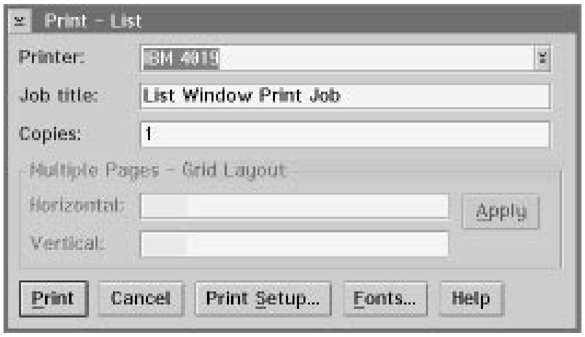
Attributes Summarizes the program element attributes (C-constant, V-virtual, E-enumerator, S-static, and PV-pure virtual).
Full Text Lists the full text of the program element attributes.
Both Lists both the summary and full text of the program element attributes.

The **Example Area** shows what the List window text will look like if you choose the different options. Select **OK** to accept the changes and **Cancel** to exit without making changes.

Note that these settings are independent of the selections made for the Graph window style settings. See “Changing Graph Styles” on page 584.

Printing and Saving your Lists

You can print the currently displayed list using **Print...** from the **File** PullDown menu. The print job will use as many pages as required to print the entire list, therefore the **Multiple Pages - Grid Layout** section of this dialog is disabled. Select the **Print** PushButton to print the current list. Select the **Print Setup...** PushButton to change the printer properties and page setup. Select the **Fonts...** PushButton to change the fonts used when printing. Note that the name of this dialog includes the type of printing you are requesting. In this case, you are requesting to print a list.



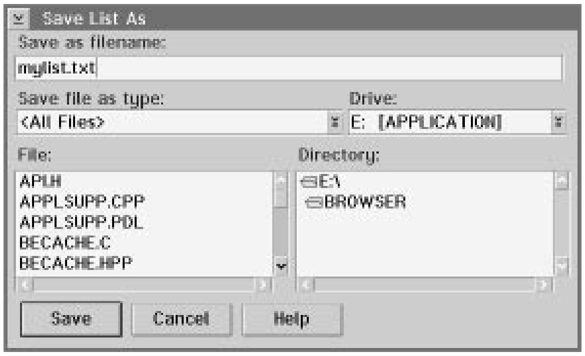
Printer: IBM 4019

Job title: List Window Print Job

Copies: 1

Figure 135. List Print Dialog

You can also save the list as an ASCII file. Note that the colors are not saved to the ASCII file. Select the drive and directory that you want to save the list to, and enter a filename into the **Save as filename:** TextEntry field, or enter the drive, path name, and file name into the TextEntry field.



Save as filename: mylist.txt

Save file as type: <All Files>

Drive: E: [APPLICATION]

File:
APLH
APPLSUPP.CPP
APPLSUPP.PDL
BECACHE.C
BECACHE.HPP

Directory:
E:\
BROWSER

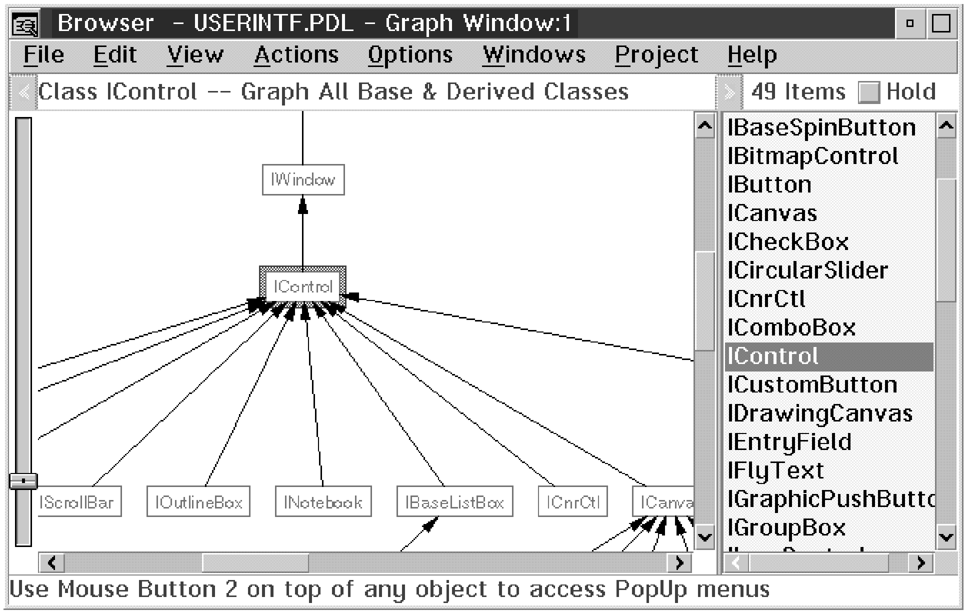
Figure 136. Save As... Dialog

The Graph Window

A Graph window displays program relationships in a graphical format. You can specify the level of detail you want the graph to show, scroll over the graph, zoom in and out, and select program elements.

Many aspects of a program can be described as relationships between program elements. Other aspects can be described by listing some group of elements in a List window described on page 564.

To use the Graph window to view your program relationships, load your source files into the Browser using the **Load Database** dialog described on page 593.



Class IControl -- Graph All Base & Derived Classes

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 137. Graph Window

The Graph window consists of:

- A main menu bar whose PullDown menu actions affect the current window. See “PullDown Menus” on page 640 for detailed descriptions of the PullDown menus.
- An **Action Status Bar** which describes what object and action produced the current graph, A definition of *object-action pair* is described on page 555. In the above figure, the **Action Status Bar** is “Class IControl - Graph All Base & Derived Classes”.

- A count that indicates how many program elements (nodes) are in the current graph. In the above figure, there are 49 class nodes.
- A **Hold** CheckBox that, when checked, keeps the current Graph window contents from being replaced when an object-action pair results in a graph. This can be useful, for instance, if you want to keep an inheritance graph available at the same time that you want to focus on another relationship. If the **Hold** CheckBox is not checked (the default behavior), the results of the next action overwrites the current window contents. There is a limit of four Graph windows.
- A **Graph Area** that contains the graphical results of your object-action pair. You can perform actions on the contents of the **Graph Area** using Mouse Button 2 on any of the nodes or on the background of the Graph window. You can also print, save to a file, or copy to the clipboard the contents of the **Graph Area**. When you select a node on the graph, the selected item is highlighted in the **List Area** of the Graph window.
- A **List Area** that alphabetically lists all the nodes on the graph. You can click on the items in this list to see where in the graph the node appears. This node is highlighted in red, by default. You can perform actions on the contents of the **List Area** using Mouse Button 2 on any of the listed items or on the background of the **List Area**. You can also save the contents of the **List Area** to an ASCII file.
- A **Slider** on the left side to quickly zoom in and out on the graph. Move the **Slider** up to reduce the size of the graph and down to increase the size.
- Scroll bars located on the right side and bottom of the **Graph Area**. Use these to scroll the graph horizontally and vertically.
- A divider located between the **Graph Area** and **List Area**. Use it to change the proportion of the screen allocated to each area.
- An **Information Bar** located at the bottom of the window that briefly describes the currently selected menu item or explains how to invoke the Object PopUp menus. You can hide the **Information Bar** by selecting the **Expert Help** level in the **Browser Settings** dialog. See “Changing Browser Settings” on page 587 for more information.

A very large number of nodes may exhaust the system resources. Note that any graph approaching this limit would not be clearly understandable in the Graph window.

The **Graph Area** has a **Background** PopUp menu you can use to perform actions on the window, such as, specifying the level of detail you want the graph to show, scrolling over the graph, zooming in and out, and changing the layout parameters. You can access this PopUp using Mouse Button 2 on the background of the **Graph Area**. See “PopUp Menu Items for List and Graph Windows” on page 648 for detailed descriptions of the window PopUps.

You can select a portion of the **Graph Area** by clicking and dragging the mouse over the area that you want to select. You can then get a PopUp menu specific for the selected region that allows you to zoom in on, copy, save, or print the selected area. See "Selecting a Graph Zone" on page 580.

Getting a Graph Overview

You can view an overview of the graph using the **Overview...** item on either the **View** PullDown or the Graph Window **Background** PopUp menu.

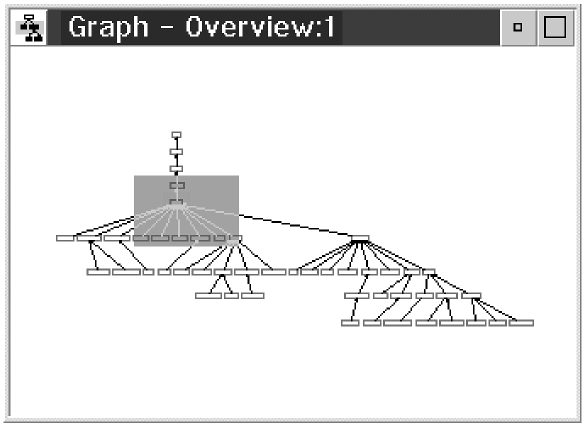


Figure 138. Overview Window

The grey shaded area indicates the current view of the **Graph Area** in the Graph window. You can move this area around or resize it. Any changes you make to the size or position of this grey area is automatically reflected in the Graph window.

Use ↔ and ↕ to size the view of the graph. The result is the same as if you had used the **Slider** on the left side of the Graph window. Use the four-way cross-arrow to move the grey area around. The result is the same as if you had use the scroll bars around the **Graph Area** of the Graph window.

Note: If you change the view of the **Graph Area** in the Graph window, this change is automatically reflected in the **Overview** window.

Organizing the Graph

You can organize the graph using two kinds of layouts that work in conjunction with one another. By default, the Graph window displays graphs in a vertical layout; that is, the nodes are drawn from top to bottom. You can change this to a horizontal layout which redraws the nodes from left to right. For instance, some wide graphs with wide nodes look best when shown horizontally, while narrower graphs look best when shown vertically. To choose a vertical or horizontal layout:

1. Select either the **View** PullDown menu from the main menu bar, or select the Graph window **Background** PopUp menu by clicking Mouse Button 2 on the background of the **Graph Area**.
2. Select the **Horizontal** or **Vertical** menu item.

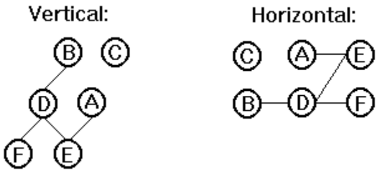


Figure 139. Horizontal versus Vertical Graph Layout

In addition to the horizontal or vertical layout, you can also weight the nodes of your graph. Some graphs are easier to understand when all root nodes or leaf nodes are at an equal level, or something somewhere in between. To select a different weighting:

1. Select either the **View** PullDown menu from the main menu bar, or select the Graph window **Background** PopUp menu by clicking Mouse Button 2 on the background of the **Graph Area**.
2. Select the **Weighting** ▸ Cascade menu.
3. Select the **Top**, **Center**, or **Bottom** menu item.

Note: If you have chosen a horizontal layout, then the root nodes will be grouped to the left and the leaf nodes will be grouped to the right.

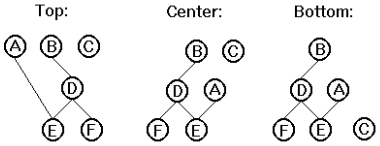


Figure 140. Top, Center and Bottom Weighting of a Graph

Selecting a Graph Zone

You can select a particular region of the graph by clicking Mouse Button 1 and dragging it across the graph. This creates a rectangular dotted box around the selected region. You can get a **Graph Zone** PopUp menu using Mouse Button 2 on this region. This PopUp has the following actions:

Zoom in Zooms in on the selected region.
Save Graph As... Saves the selected region to an OS/2 bitmap file.
Print... Prints the selected region.
Copy All Copies the selected region to the clipboard.

Browsing Graph Objects

In the Graph window, you can display class, function and file objects. See "Browsing List Objects" on page 567 for descriptions of how the Browser defines these objects.

You can launch actions from either the nodes on the graph, or their corresponding list item in the **List Area** of the Graph window, by selecting the object and clicking on Mouse Button 2 to invoke the Object PopUp menu. For more information on Object PopUp menus, see "Object PopUp Menu Items" on page 649.

Changing the Default Graph Window Settings

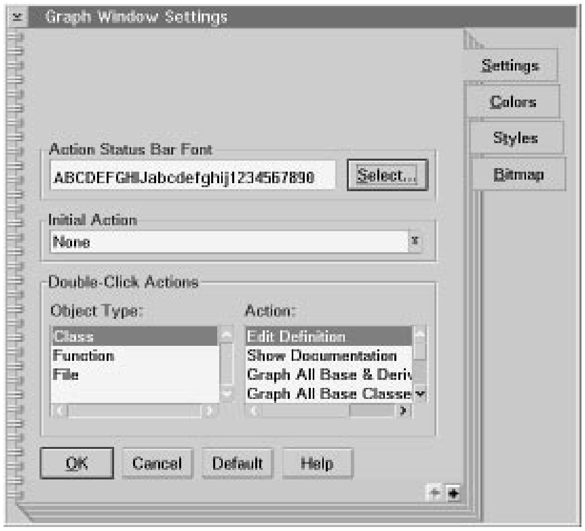
The Browser provides you with a **Graph Windows Settings** NoteBook from which you can change:

Settings Used to change the **Action Status Bar** font, the initial action performed when you load a file into the Browser, and the double-click actions of the objects displayed in the Graph window. See "Changing Graph Settings" on page 581.
Colors Used to change the colors of objects in the Graph window. See "Changing Colors Used by the Graph Window" on page 582.
Styles Used to change the shape of the nodes and arcs in the Graph window. See "Changing Graph Styles" on page 584.
Bitmap Used to change the dimensions of a saved bitmap. See "Changing Bitmap Dimensions" on page 585.

Changing Graph Settings

You can change the **Action Status Bar** font, the initial action performed when you load a file into the Browser, and the double-click actions of the objects displayed in the Graph window.

To load the **Graph Window Settings** NoteBook, select **Graph Window...** from the **Options** PullDown menu.



Action Status Bar Font: ABCDEFGHIJabcdefghij1234567890

Initial Action: None

Double-Click Actions

Figure 141. Graph Window Settings NoteBook Settings Page

You can change the font used by the **Action Status Bar** independently of the font used in the **Graph Area** and **List Area** of the Graph window. To change the font of the **Action Status Bar** on the Graph window, choose the **Select...** PushButton to load the **Action Status Bar Font** dialog. 📖 For more information on the **Fonts** dialog, see "Changing Fonts" on page 592.

You can change the initial action performed when you load a program into the Browser. Note that this action is only performed if you load a program while having a Graph window open. Use the **Initial Action** DropDown to select the action to perform when loading a program into the Browser. The Graph window does not have a default load action. You can choose from **Show Inheritance Graph**, **Show Include File Graph**, or **None**.

Use the **Double-Click Actions** section to change the double-click action of the objects in the Graph window. By default, the double-click actions for all objects is to edit the object's definition. Select the object from the **Object Types:** ListBox, and select a corresponding action from the **Actions:** ListBox.

Browser: Graph Window Settings

Changing Colors Used by the Graph Window

When you invoke an action to create a graph, the object you used to launch the action is highlighted in the Graph window in red, by default.

The Graph window displays each type of program element as a different color/shape and each relationship as a different colored/shaped arc.

Making a color change affects all open Graph windows and all subsequently opened Graph windows. The new defaults are saved to the Browser profile (`icsbrs.ini`) when you exit from the Browser.

To change the colors used by the Graph window, select **Graph Window...** from the **Options** PullDown on any Graph window to get the **Graph Windows Settings** NoteBook. Select the **Colors** tab.

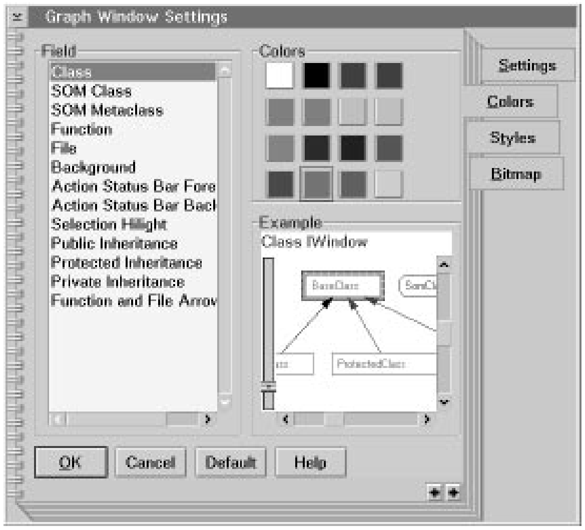


Figure 142. Graph Window Settings NoteBook Colors Page

On the left is a scrollable list of items for which you can set default colors. On the upper right is the color palette containing 16 colors to choose from. On the bottom right is an **Example Area** that previews the color selection you are currently editing.

To change an object's color:

1. Select the object from the scrollable list.
2. Click on a color from the 16 available colors. Your change appears in the **Example Area**.
3. Select the **OK** PushButton if you want to apply your color changes to the Graph Window, or select the **Cancel** PushButton if you want to exit without making any changes.

The **Default** PushButton restores the default colors used by the Browser Graph windows. By default, the color settings are:

Object	Color
Class	Cyan
SOM Class	Dark Green
SOM Metaclass	Light Green
Function	Blue
File	Red
Background	White
Action Status Bar Foreground	Blue
Action Status Bar Background	White
Selection Hilight	Red
Public Inheritance	Black
Protected Inheritance	Red
Private Inheritance	Blue
Function and File Arrows	Black

Because of the variety of colors being used in one window, the colors dropped from the OS/2 color palette will not work correctly. Any changes made this way will not be stored in the Browser profile (`icsbrs.ini`).

Note: These settings are independent of the selections made for the List window style settings. 📖 See “Changing Colors Used by the List Window” on page 571.

Browser: Graph Window Settings

Changing Graph Styles

You can change the shape of the nodes and the line style of the arcs using the **Styles** page of the **Graph Window Settings** NoteBook.

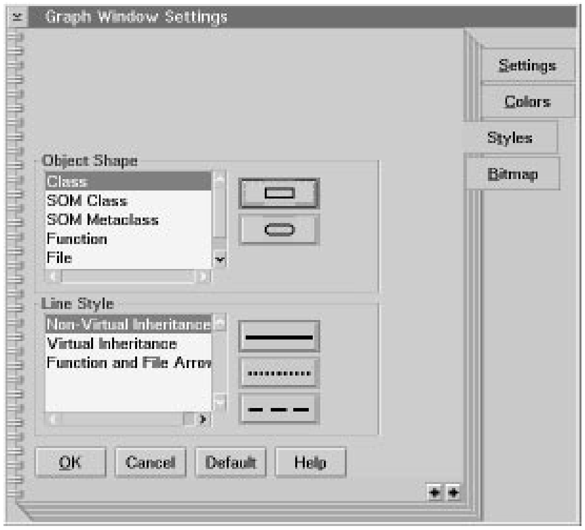


Figure 143. Graph Window Settings NoteBook Styles Page

To change the shape of the nodes and arcs:

1. Select **Graph Window...** from the **Options** PullDown on the Graph window.
2. Select the **Styles** tab on the **Graph Window Settings** Notebook.
3. Select the object/relationship from the **Object Shape**/**Line Style** ListBox.
4. Select a shape/line PushButton.
5. Select **OK** to apply the changes to the Graph window, or select **Cancel** to exit without making changes.

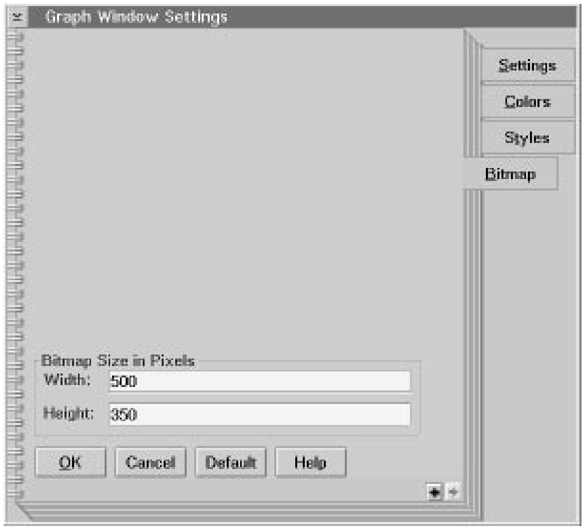
These changes will be saved to the `icsbrs.ini` profile when you exit the Browser.

The **Default** PushButton restores the Browser defaults for node shape and line style.

Changing Bitmap Dimensions

You can specify the dimensions of the bitmap to be saved to a file or copied to the clipboard. Select **Graph Window...** from the **Options** PullDown menu on the Graph window. Choose the **Bitmap** tab on the **Graph Window Settings** NoteBook. Enter the width and height of the bitmap that you want to save.

If you specify values for the width and height of the bitmap that do not correspond with the dimensions of the current graph or selection area, the output may not be as expected. For example, if the area to be saved or copied is a square and the values set specify a rectangular shape, then the image saved or copied will be stretched to fit the rectangle.



Bitmap Size in Pixels

Width: 500

Height: 350

Figure 144. Graph Window Settings NoteBook Bitmap Page

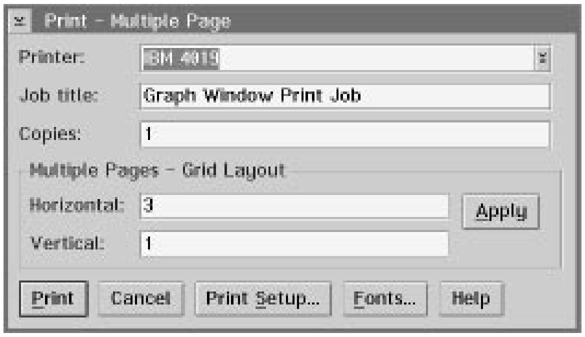
Printing and Saving your Graphs

You can print the whole graph on one or several pages, print marked sections of the graph on one page, or print the currently viewed section of the graph on one page.

Use the **Print** ▸ Cascade on the **File** PullDown menu to print:

One Page... Print the entire graph on one page.
Multiple Pages... Print the entire graph across multiple pages.
Client... Print the currently displayed portion of the graph to one page.
Zone... Print the currently selected region to one page. See “Selecting a Graph Zone” on page 580 for information on selecting areas of the graph.

By selecting any of the above print options, a **Browser Print** dialog appears. Note that the name of this dialog includes the type of printing you are requesting. In the example diagram below, you are requesting to print multiple pages.



Print - Multiple Page

Printer: IBM 4019

Job title: Graph Window Print Job

Copies: 1

Multiple Pages - Grid Layout

Horizontal: 3

Vertical: 1

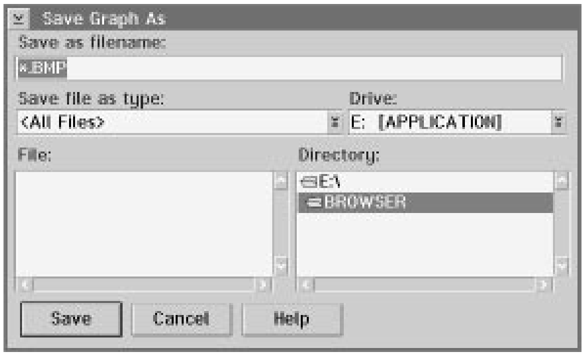
Figure 145. Graph Print Dialog

The **Multiple Pages - Grid Layout** section of the dialog is disabled when you choose to print **One Page...**, **Client...**, or **Zone...** from the **Print** ► Cascade menu on the Graph window. If you choose to print **Multiple Pages...** from the **Print** ► Cascade menu on the Graph window, you can select the layout of the pages to be printed by entering the number of horizontal and vertical pages to print. The Graph window zooms out to its maximum size, and the page layout is indicated by rectangular boxes on the graph. Each rectangular area is a page to be printed. Change the horizontal and vertical page numbers, and select the **Apply** PushButton to accept the new page layout. Selecting the **Apply** PushButton will redraw the grid layout on the graph.

Select the **Print** PushButton to print the current graph. If you chose to print multiple pages, then when the graph is finished spooling to the printer, the grid is cleared and the graph is restored to its previous zoom setting. Select the **Print Setup...** PushButton to change the printer properties and page setup. Select the **Fonts...** PushButton to change the font used when printing.

You can save either the **Graph Area** or **List Area** contents of the Graph window.

- Select **Save Graph As...** from the **File** PullDown menu on the Graph window to save the graph to an OS/2 bitmap file. See “Changing Bitmap Dimensions” on page 585 for more information on setting the size of the OS/2 bitmap saved. Note that colors, shapes and layout of the graph are saved to the OS/2 bitmap file.
- Select **Save List As...** from the **File** PullDown menu on the Graph window to save the list to an ASCII file.



Save as filename: *.BMP

Save file as type: <All Files>

Drive: E: [APPLICATION]

Directory: E:\ BROWSER

Figure 146. Save Graph As Dialog

Changing Browser Settings

The Browser provides you with a **Browser Settings** NoteBook from which you can set:

Paths Used to change the file search path, the library files to be ignored by the Browser, and the directory in which you want to save the Browser profile (`icsbrs.ini`). See “Changing Paths Used by the Browser” on page 588 for more information.

Help Level Used to set the help level provided by the Browser, as well as allow you to disable the **Exit Browser** dialog. See “Changing Help Levels” on page 590 for more information.

Browser: Path and Help Settings

Changing Paths Used by the Browser

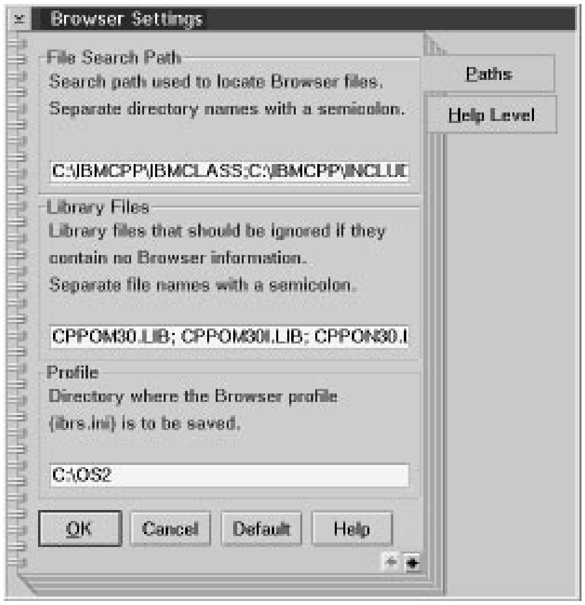
The Browser searches for files using the following sequence:

1. The path that was used to create the program.
2. The current directory.
3. The directories listed in the **Browser Settings** NoteBook **Paths** page.
4. The `INCLUDE` environment variable.
5. The `DPATH` environment variable.

Use the **Browser Settings Paths** page to specify the path names that the Browser should use in searching for Browser files and library files. You can also specify where to store the Browser profile (`icsbrs.ini`).

Note: When you list files in the Browser, the paths where these files were when the program was created will be listed. These paths may not necessarily be correct, as may be the case with the Browser shipped .PDL files for the classes that make up the IBM VisualAge C++ Open Class Library that you can access from the **Load** ► or **Merge** ► Cascade menus.

To change the paths used by the Browser, select **Browser...** from the **Options** PullDown menu to start the **Browser Settings** NoteBook. The first page in this NoteBook is the **Paths** page.



File Search Path

Search path used to locate Browser files. Separate directory names with a semicolon.

C:\IBMCPP\IBMCLASS;C:\IBMCPP\INCLUD

Library Files

Library files that should be ignored if they contain no Browser information. Separate file names with a semicolon.

CPPOM30.LIB; CPPOM30I.LIB; CPPON30.L

Profile

Directory where the Browser profile (ibrs.ini) is to be saved.

C:\OS2

Figure 147. Browser Settings NoteBook Paths Page

Browser: Path and Help Settings

Use the **File Search Path** TextEntry field to enter directory names where you want the Browser to search for files. By default, the path name contains the `INCLUDE` environment variable of the IBM VisualAge C++ Open Class Library. Add your own directories to this list in the order in which the search should be performed. Place a semicolon (;) after each directory. If two directory names in the search path contain a file with the same file name, the file in the first directory will be used.

Use the **Library Files** TextEntry field to enter the names of the library files (.LIB) that your program uses, but where you do not want to see the Browser information for those files. Such files are from another vendor or files whose internals are not important to you. The files you specify here will most likely be those that you know do not contain any relevant Browser information. By default, the IBM VisualAge C++ Open Class Library files are added to this list because they do not contain any Browser information. These library files are not used when the Browser loads all relevant information pertaining to your program or library. Place a semicolon (;) after each file name.

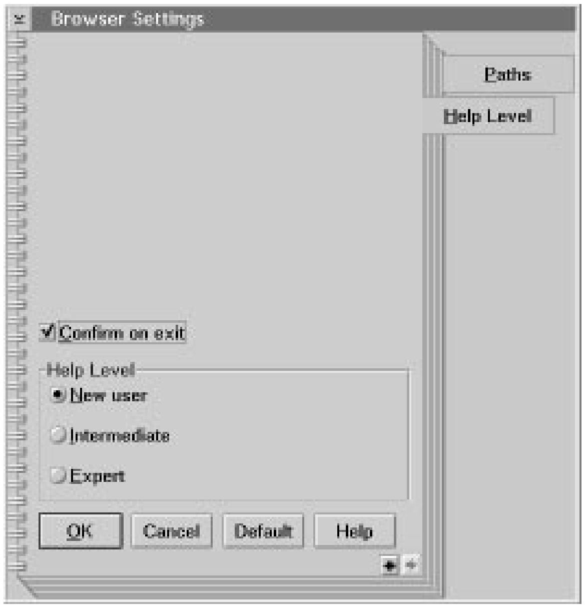
You may prefer, or need to, keep the Browser profile (`icsbrs.ini`) in a location different from the one you specified when you first used it. You can use the **Profile** TextEntry field to change its location. When you exit the Browser, it will create the profile in this new location.

Select the **OK** PushButton to accept the changed path names. The new settings are saved to the Browser profile (`icsbrs.ini`) when you exit from the Browser. Select the **Cancel** PushButton to exit without changing the path names. Use the **Default** PushButton to reset the path and file names to the Browser defaults.

Browser: Path and Help Settings

Changing Help Levels

Use the **Browser Settings** NoteBook **Help Level** page to specify different levels of help. Select **Browser...** from the **Options** PullDown menu to start the **Browser Settings** NoteBook., and select the **Help Level** tab.



☑ Confirm on exit

Help Level

● New user

○ Intermediate

○ Expert

Figure 148. Browser Settings NoteBook Help Level Page

You can set the level of help that the Browser provides:

New user	The **New User Help** dialog and an **Information Bar**.
Intermediate	No **New User Help** dialog, but an **Information Bar**.
Expert	No **New User Help** dialog and no **Information Bar**.

The **Information Bar** is located at the bottom of each Browser window and is used to describes the various menu items as you highlight them.

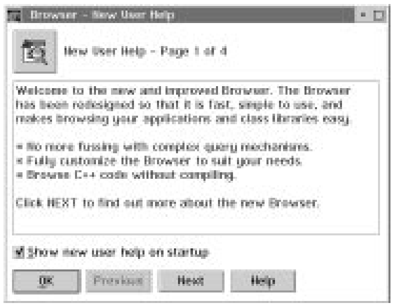
You can turn off the **Exit Browser** dialog that appears each time you close the Browser by deselecting the **Confirm on exit** CheckBox.

New User Help Dialog

When you first start the Browser, you get both a List window and the **New User Help** dialog. The information in the dialog outlines the key features of the Browser.

You can disable this dialog by unchecking the **Show new user help on startup** CheckBox or by selecting the **Intermediate** or **Expert** help levels. You can see this dialog again after disabling it by selecting the **New User** help level from the **Help Level** page in the **Browser Settings** NoteBook.

The following four figures show the contents of the **New User Help** dialog:



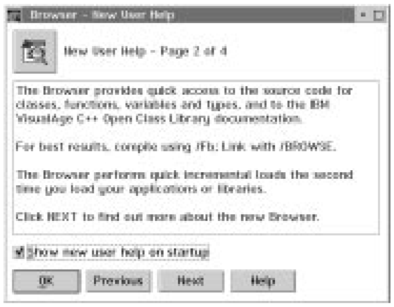
Welcome to the new and improved Browser. The Browser has been redesigned so that it is fast, simple to use, and makes browsing your applications and class libraries easy.

- No more fussing with complex query mechanisms.
- Fully customize the Browser to suit your needs.
- Browse C++ code without compiling.

Click NEXT to find out more about the new Browser.

☑ Show new user help on startup

Figure 149. New User Help Dialog Page 1



The Browser provides quick access to the source code for classes, functions, variables and types, and to the IBM VisualAge C++ Open Class Library documentation.

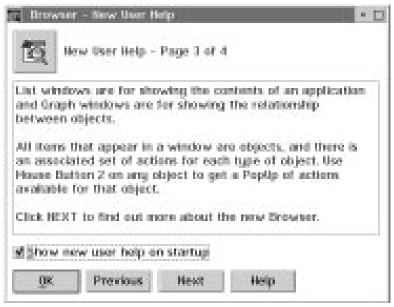
For best results, compile using /Fb; Link with /BROWSE.

The Browser performs quick incremental loads the second time you load your applications or libraries.

Click NEXT to find out more about the new Browser.

☑ Show new user help on startup

Figure 150. New User Help Dialog Page 2



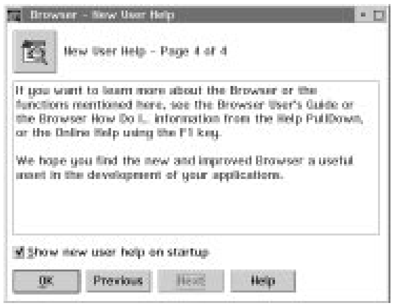
List windows are for showing the contents of an application and Graph windows are for showing the relationship between objects.

All items that appear in a window are objects, and there is an associated set of actions for each type of object. Use Mouse Button 2 on any object to get a PopUp of actions available for that object.

Click NEXT to find out more about the new Browser.

☑ Show new user help on startup

Figure 151. New User Help Dialog Page 3



If you want to learn more about the Browser or the functions mentioned here, see the Browser User's Guide or the Browser How Do I... information from the Help PullDown, or the Online Help using the F1 key.

We hope you find the new and improved Browser a useful asset in the development of your applications.

☑ Show new user help on startup

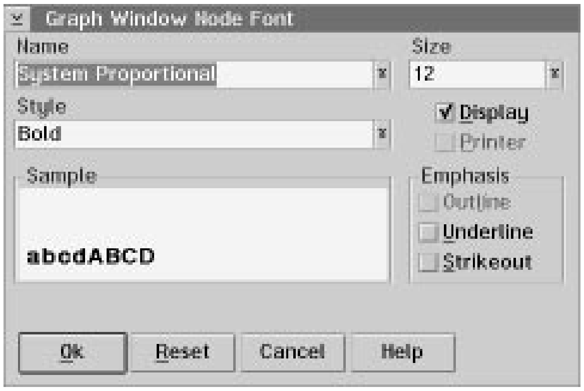
Figure 152. New User Help Dialog Page 4

Changing Fonts

You can change fonts for the List window text, the List window **Action Status Bar**, the Graph window **Graph Area** text and **List Area** text, and the Graph window **Action Status Bar**.

Change Font of:	How to:
List window text	Select **Fonts...** from the List window **Options** PullDown menu.
Action Status Bar in the List window	1. Select **List Window...** from the **Options** PullDown menu to get the **List Window Settings** NoteBook. 2. Select the **Settings** tab. 3. Choose the **Select...** PushButton.
Graph Area text in the Graph window	Select **Node Fonts...** from the Graph window **Options** PullDown menu.
List Area text in the Graph window	Select **List Fonts...** from the Graph window **Options** PullDown menu.
Action Status Bar in the Graph window	1. Select **Graph Window...** from the **Options** PullDown menu to get the **Graph Window Settings** NoteBook. 2. Select the **Settings** tab. 3. Choose the **Select...** PushButton.

All of the above actions result in a **Font** dialog. You can select the font type, size, style and emphasis.



Graph Window Node Font

Name: System Proportional

Size: 12

Style: Bold

☑ Display
□ Printer

Sample: abcdABCD

Emphasis
□ Outline
□ Underline
□ Strikeout

Figure 153. Graph Node Font Dialog

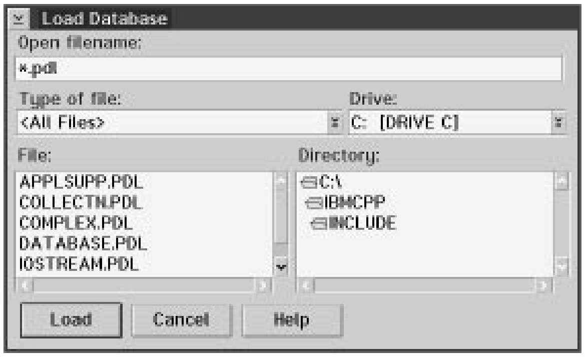
As you make selections, the **Sample Area** changes to preview the font definition you have chosen. Select the **OK** PushButton to accept the font changes. The new font settings will be saved to the Browser profile (`icsbrs.ini`) when you close the Browser. Select the **Reset** PushButton to reset the dialog selections to the last used font definition.

Loading Files into the Browser

The **Load Database** dialog is used to load your program's information into the Browser. You can load the following types of files into a Browser session:

`.DLL` Dynamic link library created using the linker option `/BROWSE`.
`.EXE` Executable created using the linker option `/BROWSE`.
`.LIB` Library file created using the linker option `/BROWSE`.
`.PDB` Browser database file created using the compiler option `/Fb`.
`.PDL` Browser database file created from a loaded .LIB file.
`.PDE` Browser database file created from a loaded .EXE file
`.PDD` Browser database file created from a loaded .DLL file.

Note: The format of the new .PDB files are incompatible with the .BRS files generated by the previous release of the Browser, and the AIX format of the .PDB files. You can probably erase these old files, unless you want to use them with the old Browsers. The good news is that the new .PDB files are between 60-95% smaller than the .BRS files for the same input files. On a large application, you will save many Megabytes of hard disk space by recompiling and generating new .PDB files.



Open filename: *.pdl
Type of file: <All Files>
Drive: C: [DRIVE C]
File:
APPLSUPP.PDL
COLLECTN.PDL
COMPLEX.PDL
DATABASE.PDL
IOSTREAM.PDL
Directory:
C:\
IBMCPP
INCLUDE

Figure 154. Load Database Dialog

Browser: Load Database Facility

For information on creating files to load into the Browser, see “Creating Files to Use with the Browser” on page 559.

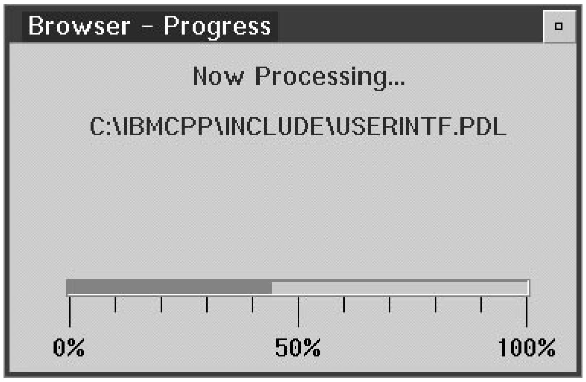
To load a file:

1. Select **Load...** from the **File** PullDown menu to start the **Load Database** dialog.
2. Change the file name extension, if appropriate, in the **Open Filename:** TextEntry field.
3. Select the drive you want to load from using the **Drive:** DropDown list.
4. Select a directory on that drive from the **Directory:** ListBox.
5. Select the file name from the **File:** ListBox. Note that you can load more than one .PDB file at a time by making multiple .PDB selections in this ListBox. You cannot do a multiple load of any other file type.
6. Select the **Load** PushButton to load the information into the Browser.

Note: You can bypass these steps if you know the name and location of the file you wish to load. Enter the path name and file name into the **Open Filename:** TextEntry field.

You can also quickly load the classes that make up the IBM VisualAge C++ Open Class Library by using the **Load** ► Cascade menu from the **File** PullDown menu. In addition, you can add your own files to the **Load** ► Cascade menu. 📖 See “Adding Menu Items to the **Load** ► and **Merge** ► Cascade menus” on page 619.

When you select the **Load** PushButton, a **Progress** dialog is displayed to show you the amount of information that has been loaded by the Browser.



Now Processing...

C:\IBMCPP\INCLUDE\USERINTF.PDL

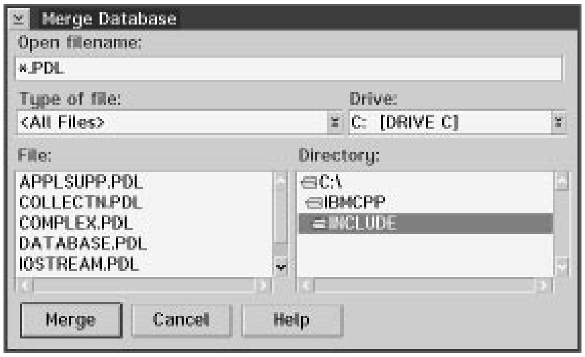
Figure 155. Progress Dialog

Merging Files

The **Merge Database** dialog is used to load more than one program into a Browser session at a time. This is useful if you are thinking of adding classes from another program. You can check out the structure while still being able to look at your own program's structure.

When you browse a target program (an .EXE, .DLL, or .LIB), you will only see those classes, functions, and files that were actually used in the program. You will not see related objects. For example, assume that you have written a small program using the IBM User Interface class library, and it contains an **IFrameWindow**, a Menu bar, and some static text. The small program will only reference the classes, functions, and files of the **IFrameWindow**, **IMenuBar**, and **IStaticText** with their parent classes. If you want to add some PushButtons and a bitmap onto your window, you can see these classes by merging the User Interface Class Library data with your small program.

Also, many programs are written as an .EXE and one or more .DLLs. If you browse the .EXE, then you only see the data from that .EXE. You can merge in the data from the .DLL(s) and see the whole program's information.



Merge Database

Open filename: *.PDL

Type of file: <All Files>

Drive: C: [DRIVE C]

File:
APPLSUPP.PDL
COLLECTN.PDL
COMPLEX.PDL
DATABASE.PDL
IOSTREAM.PDL

Directory:
C:\
IBMCPP
INCLUDE

Figure 156. Merge Database Dialog

Browser: Merge Database Facility

You can merge the following file types: .DLL, .EXE, .LIB, .PDB, .PDD, .PDE, and .PDL. If you merge more than one .PDB file, this is analogous to grouping a set of .OBJ files together into a single .LIB file, so the saved file version in this case is a .PDL file. For information on creating files to load into the Browser, see “Creating Files to Use with the Browser” on page 559.

To merge files:

1. Select **Merge...** from the **File** PullDown menu to start the **Merge Database** dialog.
2. Change the file name extension, if appropriate, in the **Open Filename:** TextEntry field.
3. Select the drive you want to load from using the **Drive:** DropDown list.
4. Select a directory on that drive from the **Directory:** ListBox.
5. Select the file name from the **File:** ListBox.
6. Select the **Merge** PushButton to merge the information into the current Browser session.

Note: You can bypass these steps if you know the name and location of the file you wish to merge. Enter the path name and file name into the **Open Filename:** TextEntry field.

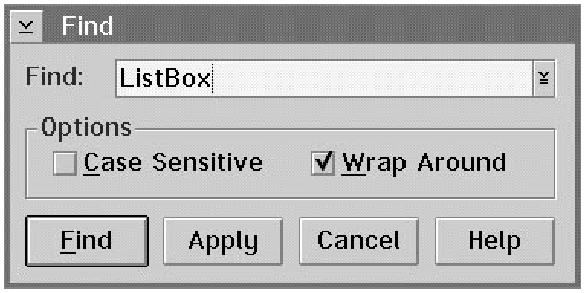
You can also quickly load the classes that make up the IBM VisualAge C++ Open Class Library by using the **Merge** ► Cascade menu from the **File** PullDown menu. In addition, you can add your own files to the **Merge** ► Cascade menu. 📖 See “Adding Menu Items to the **Load** ► and **Merge** ► Cascade menus” on page 619.

When you select the **Merge** PushButton, a **Progress** dialog is displayed to show you the amount of information that has been merged into the current Browser database.

Note: If you try to merge a file into the Browser database that duplicates some of the information that is already loaded into the Browser, a message will appear to inform you. This file will not be loaded into the Browser database.

Finding Objects in the Current Window

Use the **Find** dialog to locate text in the current window starting from the currently selected object. To launch the **Find** dialog, select **Find...** from the **Edit** PullDown menu.



Find

Find: ListBox

Options

☐ Case Sensitive ☑ Wrap Around

Figure 157. Find Dialog

Enter a text string into the **Find:** TextField or use the DownArrow icon to select from the last 10 text entries made.

Use the **Find** PushButton to initiate the search and close the **Find** dialog. Use the **Apply** PushButton to initiate the search and keep the **Find** dialog open. Use the **Cancel** PushButton to quit the dialog without performing the search.

If a match is found, the located text is brought into the view of the window and the text is highlighted. If no matches are found, a Message Box appears to let you know.

You can perform wildcard finds using:

- An asterisk (*) to match any number of characters, and
- A question mark (?) to match one character.

You can use the **Case Sensitive** CheckBox to perform case dependent searches, and use the **Wrap Around** CheckBox to search the entire contents of the current window. The message “Wrapped” appears in the **Information Bar** when the find is starting to search from the top of the list.

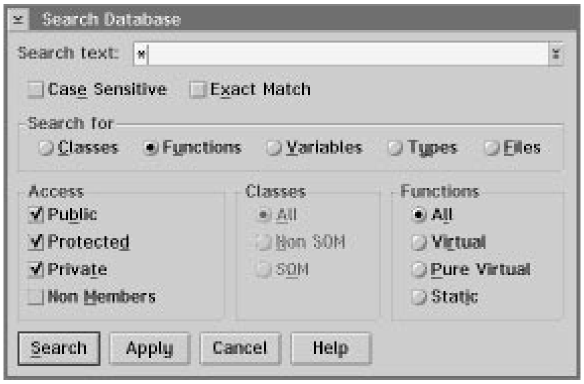
You can find the next instance of the text by either selecting **Find Next** from the **Edit** PullDown or use the **Ctrl-N** keys. Note that if you used the **Apply** PushButton, you can use it to find the next instance.

Searching for Objects in the Entire Browser Database

The **Search Database** dialog is a simple string-matching facility to help you find objects in your programs or libraries. It does not have a complex query structure. You can search the entire loaded Browser database with the following specifications:

- Object (classes, functions, variables, types, or files). For more information on the types of objects, see "Browsing List Objects" on page 567.
- Access qualifier (public, protected, private, or non-members).
- Type (all, SOM, or non-SOM).
- Function (all, virtual, pure virtual, or static).

To launch the **Search Database** dialog, select **Search...** from the **Actions** PullDown menu.



Search Database

Search text: *

☐ Case Sensitive ☐ Exact Match

Search for: ○ Classes ● Functions ○ Variables ○ Types ○ Files

Access: ☑ Public ☑ Protected ☑ Private ☐ Non Members

Classes: ● All ○ Non SOM ○ SOM

Functions: ● All ○ Virtual ○ Pure Virtual ○ Static

Figure 158. Search Database dialog

Enter the text string that you want to search for into the **Search** TextEntry field. You can also use the DownArrow icon to access the last 10 searches performed.

Select the **Search** PushButton to perform the search query and close the **Search Database** dialog. Select the **Apply** PushButton to perform the search query and keep the **Search Database** dialog available. Select **Cancel** to end the **Search Database** dialog without performing a query.

All program object names that match the search string will be listed in the List window. Note that the return types and arguments are not searched. If no matches are found, a Message Box appears to let you know.

You can perform wildcard searches using the following wildcards:

- A question mark (?) to signify specific character locations, and
- An asterisk (*) to signify any number of character locations.

You can use the **Case Sensitive** CheckBox to perform case dependent searches. If you know the exact name of the object you want to locate, then use the **Exact Match** CheckBox. Note that you cannot use wildcards in conjunction with the **Exact Match** facility. The wildcard is treated as part of the actual search string.

An example using non-exact match:

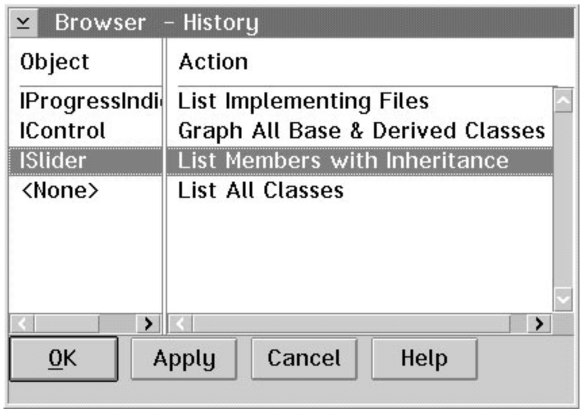
- Enter `foobar` and deselect the **Exact Match** CheckBox.
- Results: `foobar` and `realfoobar`.

An example using exact match:

- Enter `foobar` and select the **Exact Match** CheckBox.
- Results: `foobar`
  But not: `realfoobar`.

The History Window

You can use the **History** window to redo previously invoked object-action pairs. This is useful if you replaced a particular List or Graph window contents and no longer have direct access to the object. To launch the **History** window, select **History...** from the **Windows** PullDown menu on any Browser window. For a definition of object-action pair, see page 555.



Browser - History

Object	Action
IProgressIndi	List Implementing Files
IControl	Graph All Base & Derived Classes
ISlider	List Members with Inheritance
<None>	List All Classes

Figure 159. History dialog

The **History** window displays the last 40 unique object-action pairs that you have performed during your current session. The object is listed on the left hand side of the window and the action is displayed on the right side. Double-click on an object-action to invoke it, or select the **OK** PushButton to invoke the action and hide the window, or select **Apply** to invoke the action and keep the **History** window visible.

Invoking the object-action pair in the **History** window makes the Browser recalculate all the information. If the object-action pair that you want to invoke is listed at the top of the **History** window, use the **Previous** menu item from the **Actions** PullDown menu, or the **F6** key to initiate this command. The Browser does not have to recalculate the last object-action pair performed because the results are stored in a buffer. This method will be much faster.

When you perform a load, merge, or refresh, the contents of the **History** window are checked to see that the object in each object-action pair is still valid. If it is not valid, it is removed from the **History** window list.

45

Using the Browser

This chapter describes how to use the Browser to help you during your software development cycle and to help you understand your programs.

The following are covered in this chapter:

- "Using the Browser to Assist in Development."
- "Using the Browser to Aid Program Understanding" on page 604.
- "Using QuickBrowse" on page 615.

Using the Browser to Assist in Development

The Browser can be a very useful tool during program development. It provides quick access to source files for:

- "Editing and Viewing Source Files."

It gives you the ability to quickly view your uncompiled source files and the classes that make up the IBM VisualAge C++ Open Class Library:

- "Browsing without Recompiling" on page 602.
- "Browsing the IBM VisualAge C++ Open Class Library" on page 602.

It helps you in the design process by providing quick access to the VisualAge C++ Open Class Library documentation and allows you to view more than one program at the same time.

- "Showing VisualAge C++ Open Class Library Documentation" on page 603.
- "Browsing More Than One Program or Library at a Time" on page 603.

Editing and Viewing Source Files

If you have ever worked on a large project, you know how difficult it is to keep track of where program elements are defined. Sometimes large programs can be split across several files and several directories. Even using traditional search methods, such as **grep**, it can take a long time to locate where particular program elements are defined.

The Browser can help you solve this problem in one simple step:

- Select the program element that you want to edit, and use the Object PopUp menu to select the **Edit Definition** action. This is the default setting, so you can also just double-click on the object to launch the edit session.

Browser: Assisting in Development

By default, the Browser will load the VisualAge Editor and take you into the program file at the exact location where the program element is defined.

You can also quickly view your source files without searching for a specific object definition in two ways:

- Use the **List All Files** or **Show Include File Graph** menu item from the **Actions** PullDown menu, or
- Select a class object and from its PopUp menu, select the **List Implementing Files** option to get a view of file objects.

Once you have a file object, use the file's PopUp menu to select the **Edit File** menu item in order to, by default, launch the VisualAge Editor with this file loaded.

✍ The various PullDown menus are described in “PullDown Menus” on page 640.

Browsing without Recompiling

If your job is maintaining code, but it does not compile, you can use the Browser to see the program elements and their relations without having a compiled executable file. In addition, if you make modifications to your program files, you can see these changes reflected in the Browser database without having to recompile. A utility called QuickBrowse quickly examines the makefile for your loaded project and derives the compile options used to compile the required source files. The source files are then quickly parsed as if they were compiled with such options. This method is much quicker than waiting for a lengthy compile, as well as useful when you inherit code from someone else, and it does not compile.

Note: The QuickBrowse feature is only available when the Browser is started from an IBM WorkFrame project.

You will not get as much information as if you had compiled and linked your source files first: no call, exception, or template instantiation information will be available. However, you can see the class structure of your program.

✍ For more information on QuickBrowse, see “Using QuickBrowse” on page 615.

Browsing the IBM VisualAge C++ Open Class Library

The Browser provides quick access to the classes that make up the IBM VisualAge C++ Open Class Library. You can select them to view independently of any other files by selecting the library from the **Load** ▸ Cascade menu on the **File** PullDown menu.

The Browser also makes it easy to merge the classes that make up the IBM VisualAge C++ Open Class Library when you are viewing your own programs. You can select the classes from the **Merge** ▸ Cascade menu on the **File** PullDown menu.

📖 The **Load Database** dialog is described in “Loading Files into the Browser” on page 593. The **Merge Database** dialog is described in “Merging Files” on page 595. The **File** PullDown menu items are defined in “**File** PullDown Menu” on page 641.

Browsing More Than One Program or Library at a Time

When you browse a target program (an .EXE, .DLL, or .LIB), you will only see those classes, functions, and files that were actually used in the program. You will not see related objects. For example, assume that you have written a small program using the user interface classes defined in the IBM VisualAge C++ Open Class Library, and it contains an **IFrameWindow**, a menu bar, and some static text. Next, you want to add a couple of PushButtons and a Bitmap onto your window. To see these classes, you can merge the **User Interface Classes** data (all of it) with your program's data, and see all the interface facts about these particular classes that make up part of the IBM VisualAge C++ Open Class Library.

Also, many programs are written as an .EXE and one or more .DLLs. If you browse the .EXE, then you only see the data from that .EXE. You can merge in the data from the .DLL(s) and see the whole program's information.

Note that all IBM VisualAge C++ Open Class Library classes start with the letter “I” with the exception of the I/O Stream and Complex Mathematics classes. 📖 The **File** PullDown menu is described in “**File** PullDown Menu” on page 641. The **Merge Database** dialog is described in “Merging Files” on page 595.

Showing VisualAge C++ Open Class Library Documentation

The Browser gives you quick access to the class and function references provided for the IBM VisualAge C++ Open Class Library. If you want to find out more information on one of these classes or functions, from the **Class** or **Function** PopUp menu, select the **Show Documentation** option.

📖 The PopUp menus are defined in “PopUp Menus” on page 648.

Using the Browser to Aid Program Understanding

The Browser provides two kinds of windows for displaying your program elements and their association with one another: List and Graph windows.

Many aspects of your program can be described by listing some group of elements in a List window:

📖 “List All Classes Defined in the Currently Loaded Program.”
📖 “List All Files Used to Create the Currently Loaded Program.”
📖 “Listing All Objects Defined in a File” on page 605.
📖 “Listing All Friends of a Class” on page 606.
📖 “Listing All Friendships of a Class or Function” on page 607.
📖 “Listing Immediate Callers and Callees for a Function” on page 608.
📖 “Listing Instantiations of Classes or Functions” on page 611.
📖 “Listing Implementing Files” on page 606.
📖 “Listing All Class Members” on page 609.
📖 “Listing Overriding Derived Classes” on page 610.
📖 “Listing All the Exceptions That A Function May Encounter” on page 611.

Many aspects of your program can be described as relationships in a Graph window:

📖 “Viewing Class Relationships” on page 612.
📖 “Viewing Call Chains” on page 613.
📖 “Viewing Include File Relationships” on page 614.

List All Classes Defined in the Currently Loaded Program

By default, when you first load a program into the Browser, a list of all the classes defined for that program is displayed in a List window. If you no longer have this list visible, you can list all the classes by either:

- Selecting the **List All Classes** from the **Actions** PullDown menu, or
- Selecting the **List All Classes** from the List or Graph window **Background** PopUp menu.

List All Files Used to Create the Currently Loaded Program

You can list all the files that were used to create the currently loaded program:

- Select the **List All Files** from the **Actions** PullDown menu
- Select the **List All Files** from the List or Graph window **Background** PopUp menu.

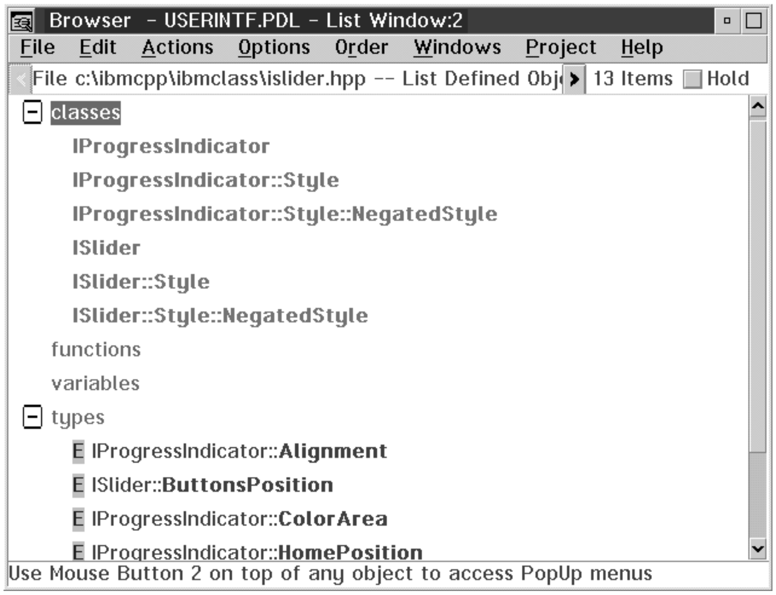
Note that the directories displayed in the list may not be accurate. These are the directories that were used when the program was created. 📖 For more information on where the Browser searches for files, see “Changing Paths Used by the Browser” on page 588.

Listing All Objects Defined in a File

If you have a list of files or an include file graph, then you can use the file object to determine all the program elements that are defined in that file.

1. Select the file object with Mouse Button 2 to get the **File** PopUp menu.
2. Select the **List Defined Objects** menu item.

A list of all the program elements defined in the file will be displayed in a List window.



File c:\ibmcpp\ibmclass\islider.hpp -- List Defined Obj

```
⊟ classes
    IProgressIndicator
    IProgressIndicator::Style
    IProgressIndicator::Style::NegatedStyle
    ISlider
    ISlider::Style
    ISlider::Style::NegatedStyle
  functions
  variables
⊟ types
    E IProgressIndicator::Alignment
    E ISlider::ButtonsPosition
    E IProgressIndicator::ColorArea
    E IProgressIndicator::HomePosition
```

Use Mouse Button 2 on top of any object to access PopUp menus

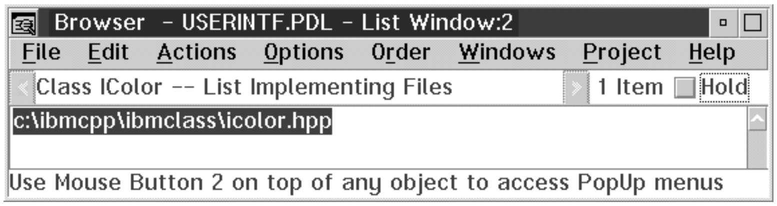
Figure 160. An Example List: File islider.hpp - List Defined Objects

Listing Implementing Files

If you want to know where class definitions are defined:

1. Select the class object with Mouse Button 2 to get the **Class** PopUp menu.
2. Select the **List Implementing Files** menu item.

A list of all files that contain definitions for the currently selected class and all of its members will be displayed in a List window.


Browser - USERINTF.PDL - List Window:2
File Edit Actions Options Order Windows Project Help
1 Item
□ Hold

Class IColor -- List Implementing Files

c:\ibmcpp\ibmclass\icolor.hpp

Use Mouse Button 2 on top of any object to access PopUp menus

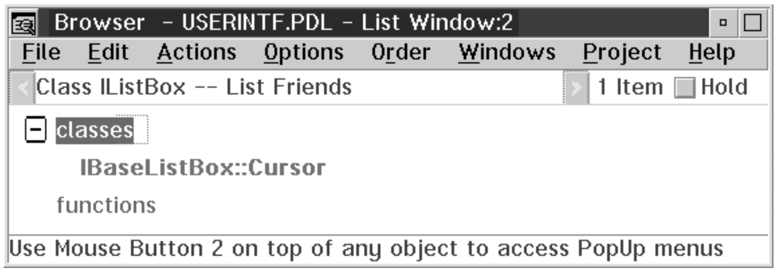
Figure 161. An Example List: Class IColor - List Implementing Files

Listing All Friends of a Class

If you have a list of classes or an inheritance graph, you can use the class object to display all the friends of the currently selected class.

1. Select the class object with Mouse Button 2 to get the **Class** PopUp menu.
2. Select the **List Friends** menu item.

A list of all the friends defined for the currently selected class are displayed in a List window. If there are no friends defined for the currently selected class, then this menu item will be disabled.


Browser - USERINTF.PDL - List Window:2
File Edit Actions Options Order Windows Project Help
1 Item
□ Hold

Class IListBox -- List Friends

classes
IBaseListBox::Cursor
functions

Use Mouse Button 2 on top of any object to access PopUp menus

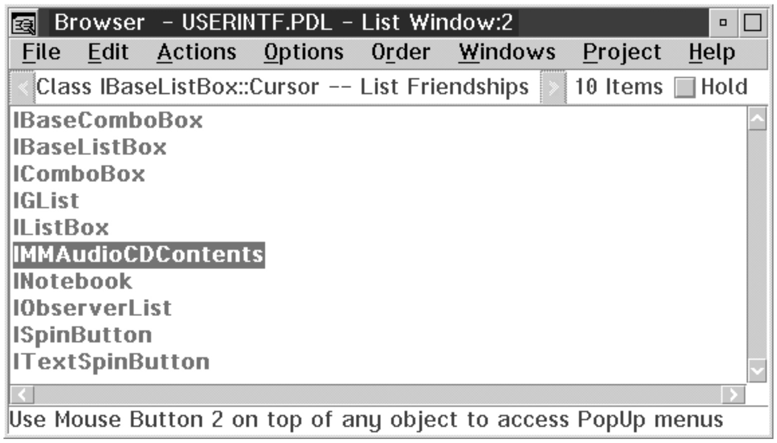
Figure 162. An Example List: Class IBaseListBox - List Friends

Listing All Friendships of a Class or Function

Friendships can be granted to either a single function or a member function, or to all the member functions of a class at once. You can list all the friendships defined for a class or function:

1. Select the class or function object with Mouse Button 2 to get either the **Class** or **Function** PopUp menu.
2. Select the **List Friendships** menu item.

A list of all the friendships that are defined for the selected object are displayed in a List window. If there are no friendships defined for the currently selected function, then this menu item will be disabled.



Class IBaseListBox::Cursor -- List Friendships

IBaseComboBox
IBaseListBox
IComboBox
IGList
IListBox
IMMAudioCDContents
INotebook
IObserverList
ISpinButton
ITextSpinButton

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 163. An Example List: Class IListBox::Cursor - List Friendships

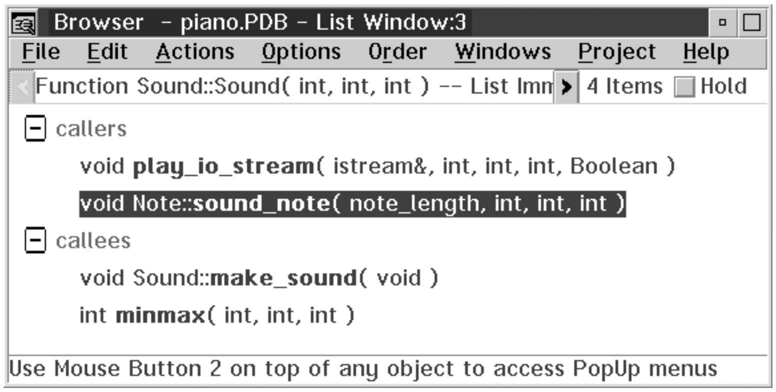
Listing Immediate Callers and Callees for a Function

When debugging your programs, it is often important to know what impact changing a particular function may have on other functions that it calls or call it. You can list all the functions that call a particular function and that a particular function calls:

1. Select the function object with Mouse Button 2 to get the **Function** PopUp menu.
2. Select the **List Immediate Callers & Callees** menu item.

A list of all the callers and callees for the selected function will be displayed in a List window. If there are no callers or callees are defined for the currently selected function, then this menu item will be disabled.

Note: This information is not available if you have used QuickBrowse.



Function Sound::Sound(int, int, int) -- List Imm

⊟ callers

void **play_io_stream**(istream&, int, int, int, Boolean)

void Note::**sound_note**(note_length, int, int, int)

⊟ callees

void Sound::**make_sound**(void)

int **minmax**(int, int, int)

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 164. An Example List: Function Sound::Sound - List Immediate Callers & Callees

Listing All Class Members

It is often necessary to know what members are defined for a given class or what class a given member function is a member of. You can list all members of a class and its base classes:

1. Select the class or function object with Mouse Button 2 to get either the **Class** or **Function** PopUp menu.
2. Select the **List Members with Inheritance** or **List Class Members with Inheritance** menu item.

A List window container view of all the classes, base classes, and members is displayed. A container view is a list which can be further expanded using the + and - icons to expand and collapse the entries. There are three ways to order this window: by classes, by access, or by type. By default, the items are ordered by class. Note that this list is not ordered alphabetically, but are arranged in a depth first tree traversal of the classes' inheritance hierarchy.

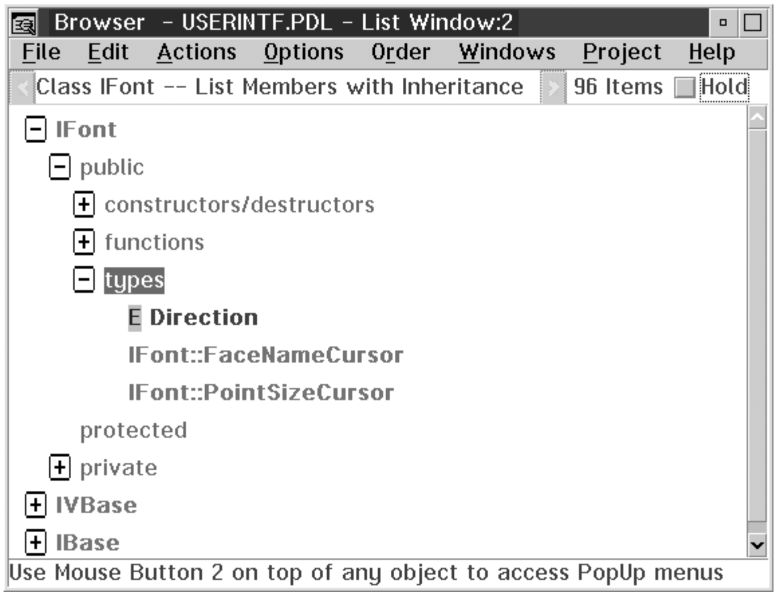

Browser - USERINTF.PDL - List Window:2
File Edit Actions Options Order Windows Project Help
Class IFont -- List Members with Inheritance
96 Items
Hold
IFont
public
constructors/destructors
functions
types
E Direction
IFont::FaceNameCursor
IFont::PointSizeCursor
protected
private
IVBase
IBase
Use Mouse Button 2 on top of any object to access PopUp menus

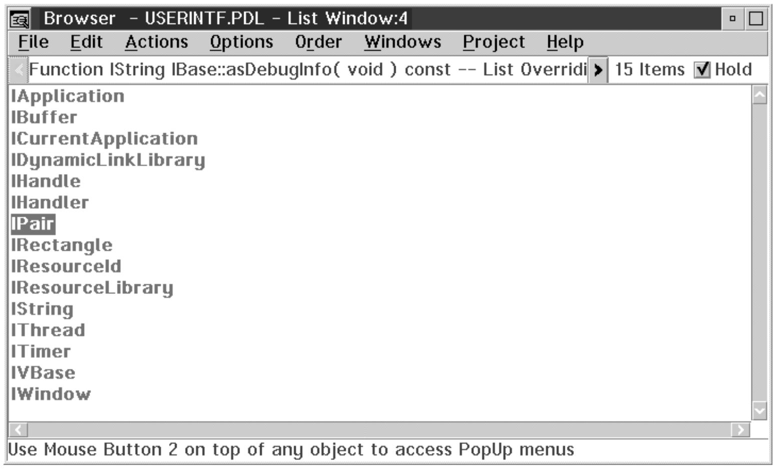
Figure 165. An Example List: Class IFont - List Members with Inheritance

Listing Overriding Derived Classes

It is often important to know when, where, and if a function is overrided. You can list all the overriding derived classes for a function:

1. Select the function object with Mouse Button 2 to get the **Function** PopUp menu.
2. Select the **List Overriding Derived Classes** menu item.

A list of all the overriding derived classes will be displayed in a List window. This menu item is disabled if the function is not the member of a class, or there are no derived classes for the class this function is a member of.



Function IString IBase::asDebugInfo(void) const -- List Overridi

IApplication
IBuffer
ICurrentApplication
IDynamicLinkLibrary
IHandle
IHandler
IPair
IRectangle
IResourceId
IResourceLibrary
IString
IThread
ITimer
IVBase
IWindow

Use Mouse Button 2 on top of any object to access PopUp menus

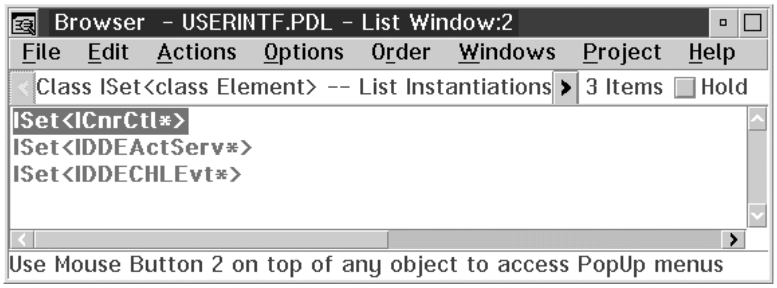
Figure 166. An Example List: Function IBase::asDebugInfo - List Overriding Derived Classes

Listing Instantiations of Classes or Functions

You can list the template instantiations for classes and functions:

1. Select the class or function object with Mouse Button 2 to get either the **Class** or **Function** PopUp menu.
2. Select the **List Instantiations** menu item.

A list of all the template instantiations for the currently selected object will be displayed in a List window. This menu item is disabled if the currently selected class or function is not a class or function template.


Browser - USERINTF.PDL - List Window:2
File Edit Actions Options Order Windows Project Help
Class ISet<class Element> -- List Instantiations
3 Items
□ Hold

ISet<ICnrCtl*>
ISet<IDDEActServ*>
ISet<IDDECHLEvt*>

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 167. An Example List: Class ISet<class Element> - List Instantiations

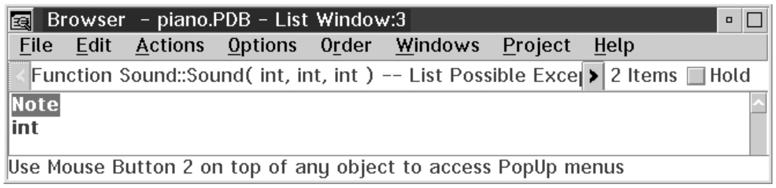
Listing All the Exceptions That A Function May Encounter

C++ uses exception handling to support error handling because throwing or catching an exception can affect the way a function relates to other functions. You need to know what these exceptions are.

You can quickly list all the exceptions for a given function using the **List Possible Exceptions Thrown** item on the **Function** PopUp menu.

Note: This information will not be available for any data loaded using QuickBrowse.

The **Function** PopUp menu is described in “Object PopUp Menu Items” on page 649.


Browser - piano.PDB - List Window:3
File Edit Actions Options Order Windows Project Help
Function Sound::Sound(int, int, int) -- List Possible Exce
2 Items
□ Hold

Note
int

Use Mouse Button 2 on top of any object to access PopUp menus

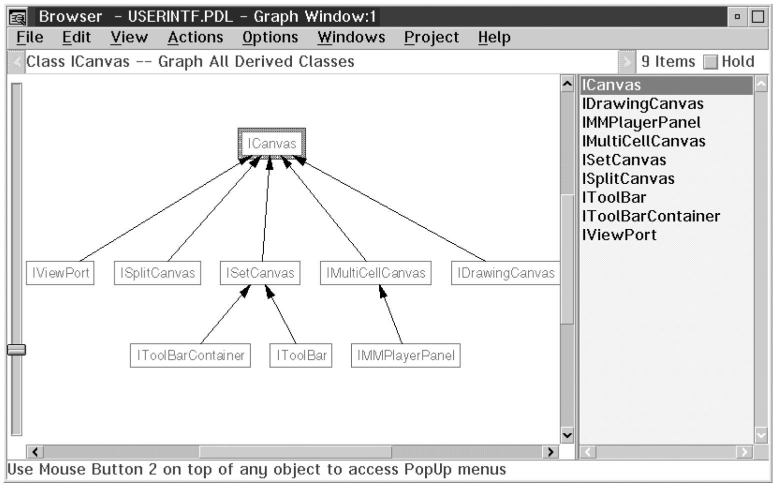
Figure 168. An Example List: Function Sound::Sound - List Possible Exceptions Thrown

Viewing Class Relationships

You can quickly graph the class inheritance relationships using one of the following **Class** PopUp menu items:

- **Graph All Base & Derived Classes**
- **Graph All Base Classes**
- **Graph All Derived Classes**
- **Graph Immediate Derived Classes**

The **Class** PopUp menu is described in “Object PopUp Menu Items” on page 649.



Browser - USERINTF.PDL - Graph Window:1

Class ICanvas -- Graph All Derived Classes

9 Items □ Hold

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 169. An Example Graph: Class ICanvas - Graph All Derived Classes

Viewing Call Chains

When debugging your programs, you often have to follow the call chain. Sometimes you may have a function that is acting unexpectedly. You need to know all the functions that may be calling it or that it calls, so that you can determine what effect the function may have on other program components.

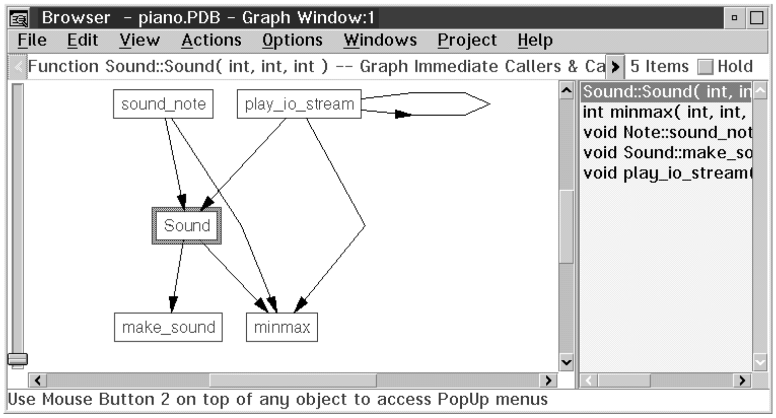
You can quickly graph the call chain of a function using one of the following **Function** PopUp menu items:

- **Graph All Callers**
- **Graph All Callees**
- **Graph All Callers & Callees**
- **Graph Immediate Callers & Callees**

depending on how much of the call chain you want to see. You will not see calls to C functions defined in the system header files. See "/Fb" on page 269 for a description of how to compile for use with the Browser.

Note: This information will not be available for any data loaded using QuickBrowse.

The **Function** PopUp menu is described in "Object PopUp Menu Items" on page 649.



Function Sound::Sound(int, int, int) -- Graph Immediate Callers & Ca

Sound::Sound(int, in
int minmax(int, int,
void Note::sound_not
void Sound::make_so
void play_io_stream(

Use Mouse Button 2 on top of any object to access PopUp menus

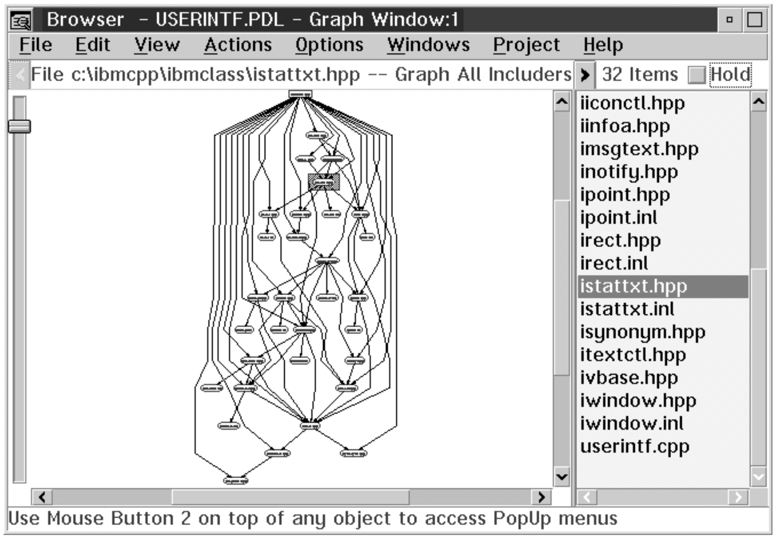
Figure 170. An Example Graph: Function Sound::Sound - Graph Immediate Callers & Callees

Viewing Include File Relationships

You can quickly graph the file structure of your programs showing you where header files are included. You can show this structure using the following PopUp menu items:

- **Graph All Includees**
- **Graph All Includers**
- **Graph All Includers & Includees**

The **File** PopUp menu is described in “Object PopUp Menu Items” on page 649.



File c:\ibmcpp\ibmclass\istattxt.hpp -- Graph All Includers

iiconctl.hpp
iinfoa.hpp
imsgtext.hpp
inotify.hpp
ipoint.hpp
ipoint.inl
irect.hpp
irect.inl
istattxt.hpp
istattxt.inl
isynonym.hpp
itextctl.hpp
ivbase.hpp
iwindow.hpp
iwindow.inl
userintf.cpp

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 171. An Example Graph: File istattxt.hpp - Graph All Includers & Includees

When the Browser lists files, it displays the path name of the files when the program was compiled. However, this path name may not be correct, as is the case with the Browser shipped .PDL files for the IBM VisualAge C++ Open Class Library classes that you can load or merge from the **Load** ▸ and **Merge** ▸ Cascade menus. For information on where the Browser searches for files, See “Changing Paths Used by the Browser” on page 588.

Using QuickBrowse

QuickBrowse is a code analysis technology that allows the VisualAge C++ Browser to extract type information from C++ source code without the associated overhead of compilation. Declarations local to functions and function call information are not provided by QuickBrowse.

The QuickBrowse feature allows you to quickly obtain and browse type information for code for which there is no compiler generated (**/Fb**) Browser information. Use QuickBrowse for the following reasons:

- It is faster than compiling the code
- You may be able to browse files that do not compile

Note: The QuickBrowse feature is only available when the Browser is started from an IBM WorkFrame project.

QuickBrowse parses the top level declarations which must be valid C++ statements, and ignores the bodies of function definitions.

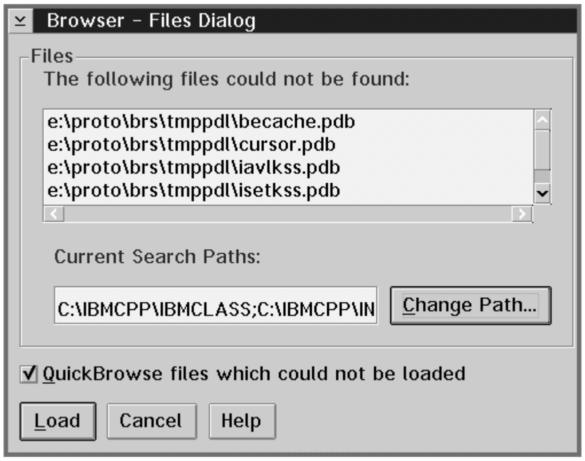
You may want to use QuickBrowse if you are not interested in function call information. Also, if you have code where the type information is well defined, but function bodies will not compile, you can browse the type information with QuickBrowse.

If you are browsing in a project, and the Browser detects that some, or all, information is missing, a dialog will appear telling you that this information is missing, and will give you the option of QuickBrowsing the files for which data is missing. Messages will appear in the Project's monitor, just as if you were doing a build.

Note that the QuickBrowse feature is not a complete replacement to the Generate Browser information (**/Fb**) compiler option. The speed of QuickBrowse does come at a cost in the richness of information provided. Since the QuickBrowse feature does not look inside of function bodies, function call, exception, and template instantiation information are not available. If you need to know this kind of information, then you will need to compile the file and use the Generate Browser information (**/Fb**) option.

What Do You See When QuickBrowse Starts

When QuickBrowse is started, you will be presented with the **Browser Files** dialog. It lists all the files that will require QuickBrowsing. In addition, you can select the **Change Path** PushButton to change the paths used by the Browser. 📖 For more information on changing paths, see “Changing Paths Used by the Browser” on page 588. Make sure the **QuickBrowse files which could not be loaded** CheckBox is selected in order to QuickBrowse the files.



The following files could not be found:

e:\proto\brs\tmppdl\becache.pdb
e:\proto\brs\tmppdl\cursor.pdb
e:\proto\brs\tmppdl\iavlkss.pdb
e:\proto\brs\tmppdl\isetkss.pdb

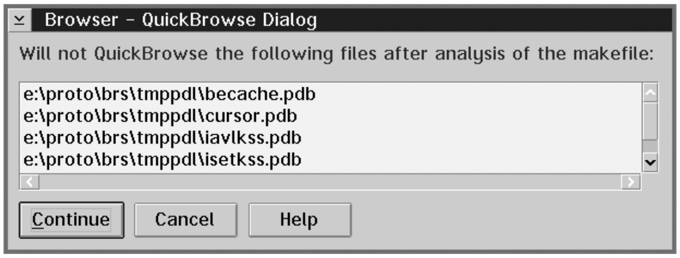
Current Search Paths:

C:\IBMCPP\IBMCLASS;C:\IBMCPP\IN

☑ QuickBrowse files which could not be loaded

Figure 172. Browser Files Dialog

Select the **Load** PushButton on the **Browser Files** dialog to QuickBrowse the listed files. If the Browser cannot find the compiler option information for some of the files listed in the **Browser Files** dialog, then the **Browser QuickBrowse** dialog will appear. 📖 For more information on where the Browser searches for files, see “Changing Paths Used by the Browser” on page 588.



Will not QuickBrowse the following files after analysis of the makefile:

e:\proto\brs\tmppdl\becache.pdb
e:\proto\brs\tmppdl\cursor.pdb
e:\proto\brs\tmppdl\iavlkss.pdb
e:\proto\brs\tmppdl\isetkss.pdb

Figure 173. QuickBrowse Dialog

The **QuickBrowse** dialog lists the files for which compiler information could not be found, but which require QuickBrowsing or recompiling. Select **Continue** to load the already QuickBrowsed information of the other files listed in the **Browser Files** without the information of these listed files.

Scenarios for Using QuickBrowse

The following are some sample QuickBrowse scenarios:

"Porting Code Scenario."
"Design Scenario."
"Browsing Libraries Scenario."
"Code Understanding Scenario."

Porting Code Scenario

You may be porting from one operating system to OS/2. Your types are correct, but many of your functions cannot compile, since they use system functions which do not exist on OS/2. You can use the QuickBrowse feature on such code to understand the type structure.

Design Scenario

The QuickBrowse feature can help you with your designing, especially if you are doing design in a group, or you wish to communicate your design with others. To do this, you need to get your types straight - or at least to the point where they are "correct" C++ code. Then, you can use QuickBrowse.

You could compile at this point too, but QuickBrowse will generally be quicker, and the function bodies can be in whatever state you like, as long as the { } match up.

Browsing Libraries Scenario

Coming to terms with class libraries and frameworks means understanding the types they provide. Typically, intra-library function calls are not exposed to the user, except for trivial inline functions. You can create a single source file which includes all the interfaces to a third party library, and either use the compiler to generate the Browser information, or use the QuickBrowse feature.

Code Understanding Scenario

When you get new code, use QuickBrowse to understand the type structure of the code while ignoring the functional details. Then, when you want to look at the function call relationships, compile the code. In well-designed and well-built code, the key abstractions will be found in the type structure. Dealing with just the types also reduces the amount of information that you have to comprehend at the beginning.

Updating the Browser Database

While you are browsing your program, you may also be modifying the source files. By selecting the **Refresh** action from the **File** PullDown menu, the Browser will check to see if the loaded program is out of date by checking the dates of the source files against the program files. If you have modified your source code, but have not regenerated the Browser information, the **Browser Files** dialog appears.

- If you are browsing an IBM WorkFrame project:

  You can choose to QuickBrowse your source by selecting the **Load** PushButton. The QuickBrowse facility is fast, but there may be a loss of information provided for your program. The benefit is that you do not have to wait for a lengthly recompile of your source to browse the declarations in your programs. For more information on QuickBrowse, see “Browsing without Recompiling” on page 602 and “Using QuickBrowse” on page 615.

- If you are not browsing an IBM WorkFrame project:

  The **Browser Files** dialog will list all files that are out of date and cannot be found. You cannot use the QuickBrowse facility to load these files, since QuickBrowse is only available for IBM WorkFrame projects. If you select the **Load** PushButton, the Browser will update the Browser database with the information that is available and will delete all the information associated with the listed files. For example, if you are browsing an .EXE file that has five .PDB files (call them A, B, C, D and E), and you have updated A and B, but not C and D, and E did not change, then the Browser will delete the old information (A, B, C and D), and load in the new (A and B). The result is that after the refresh, the Browser has information for A, B and E, but not for C and D. To regain the information for C and D, you will have to rebuild the .EXE file and load it into the Browser. If you do not want to loose the information contained in the files listed in the **Browser Files** dialog, then select the **Cancel** PushButton, and rebuild these listed files. Now you can select the **Refresh** action again to update all the information in the Browser database.

Note: The **Refresh** action will only load or modify the current Browser database as required by performing an incremental smart load. This means that the current Browser database will be refreshed much quicker than if it had to regenerate the entire Browser database.

Adding Menu Items to the Load ► and Merge ► Cascade menus

If you perform repeated loads or merges of a particular program, you can add it to the **Load** ► and **Merge** ► Cascade menus for quick access. To do this:

1. Create an ASCII file called `brsmenu.txt` and place it in a directory in your `DPATH`.
2. Use the following format:

   ```
   Menu Item Name"path_name\file_name
   ```

   Where `Menu Item Name` is the name you want to have appear on the **Load** ► and **Merge** ► Cascade menus, and `path_name\file_name` is the path name and file name of the file to be loaded or merged. You can have spaces in the `Menu Item Name`. Be sure to separate the `Menu Item Name` from the `path_name\file_name` with a double quote (").

   Note: You must have a blank line at the end of the `brsmenu.txt` file.
3. You can add upto a maximum of six files.

The new menu items will be added to the **Load** ► and **Merge** ► Cascade menus the next time you start the Browser.

🕮 For more information on loading, see “Loading Files into the Browser” on page 593. For more information on merging, see “Merging Files” on page 595.

Browser: Adding Menu Items

46

A Tour of the Browser

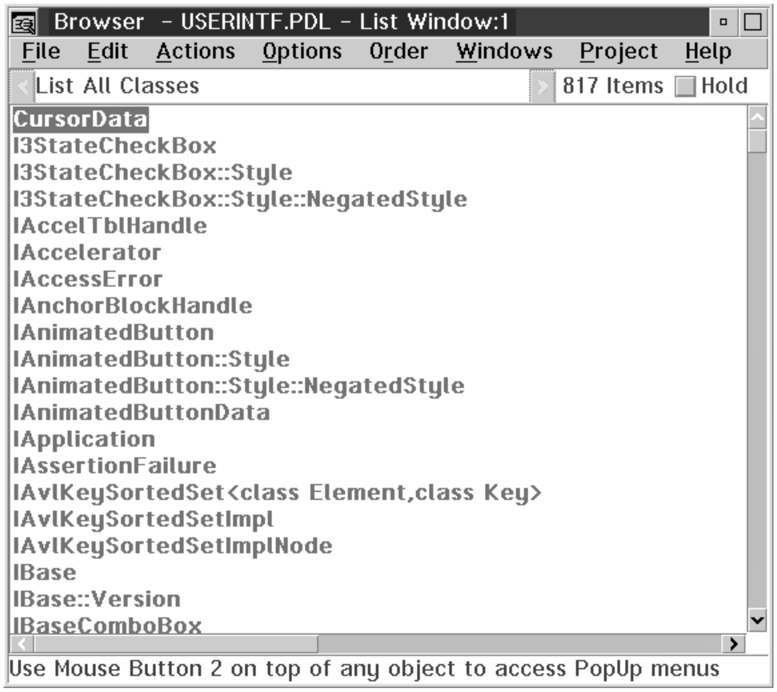
This tour takes you through some of the features of the Browser using the User Interface classes that make up part of the IBM VisualAge C++ Open Class Library. The following is a list of tasks performed during this tour:

Starting the Browser and Loading User Interface Classes

You will be browsing the User Interface classes of the IBM VisualAge C++ Open Class Library.

1. On the OS/2 command line, enter: `icsbrs`.
2. Select the **File** PullDown menu.
3. Select the **Load** ► Cascade menu.
4. Select the **User Interface Classes** menu item.

By default, the List Window will populate with a list of all the classes defined in the User Interface classes of the IBM VisualAge C++ Open Class Library.



```
CursorData
I3StateCheckBox
I3StateCheckBox::Style
I3StateCheckBox::Style::NegatedStyle
IAccelTblHandle
IAccelerator
IAccessError
IAnchorBlockHandle
IAnimatedButton
IAnimatedButton::Style
IAnimatedButton::Style::NegatedStyle
IAnimatedButtonData
IApplication
IAssertionFailure
IAvlKeySortedSet<class Element,class Key>
IAvlKeySortedSetImpl
IAvlKeySortedSetImplNode
IBase
IBase::Version
IBaseComboBox
```

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 174. List window showing all classes in the User Interface classes

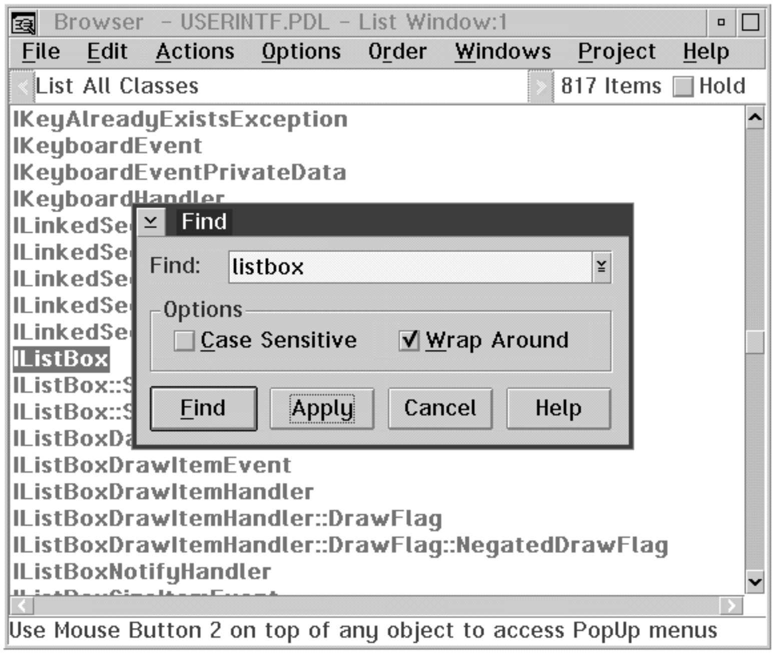
For more information on loading, see “Loading Files into the Browser” on page 593.

Finding A Class

Suppose that in your application, you want to use a Listbox. You will need to find more about Listboxes. From the List window:

1. Select the **Edit** PullDown menu.
2. Select the **Find...** menu item. The **Find** dialog appears.
3. Enter `listbox` as the search text.
4. Check that case sensitive is off.
5. Select the **Apply** PushButton until `IListbox` is highlighted.
6. Cancel the **Find** dialog.

The list scrolls to the `IListBox` class.



List All Classes
817 Items □ Hold

IKeyAlreadyExistsException
IKeyboardEvent
IKeyboardEventPrivateData
IKeyboardHandler
ILinkedSe
ILinkedSe
ILinkedSe
ILinkedSe
ILinkedSe
IListBox
IListBox::S
IListBox::S
IListBoxDa
IListBoxDrawItemEvent
IListBoxDrawItemHandler
IListBoxDrawItemHandler::DrawFlag
IListBoxDrawItemHandler::DrawFlag::NegatedDrawFlag
IListBoxNotifyHandler

Find: listbox

Options
□ Case Sensitive ☑ Wrap Around

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 175. Finding a class name

For more information on the **Find** facility, see "Finding Objects in the Current Window" on page 597.

Showing the Inheritance Relationship of a Class

Now you will need to show the relationship of IListBox in the class hierarchy. Note that there are other classes that start with IListBox which you could look at later.

1. Click Mouse Button 2 on top of `IListBox` to get the **Class** PopUp menu.
2. Select **Graph All Base Classes** from the **Class** PopUp menu.

The Graph window shows the inheritance hierarchy for `IListBox`, and lists an alphabetical list of the classes that appear in the graph.

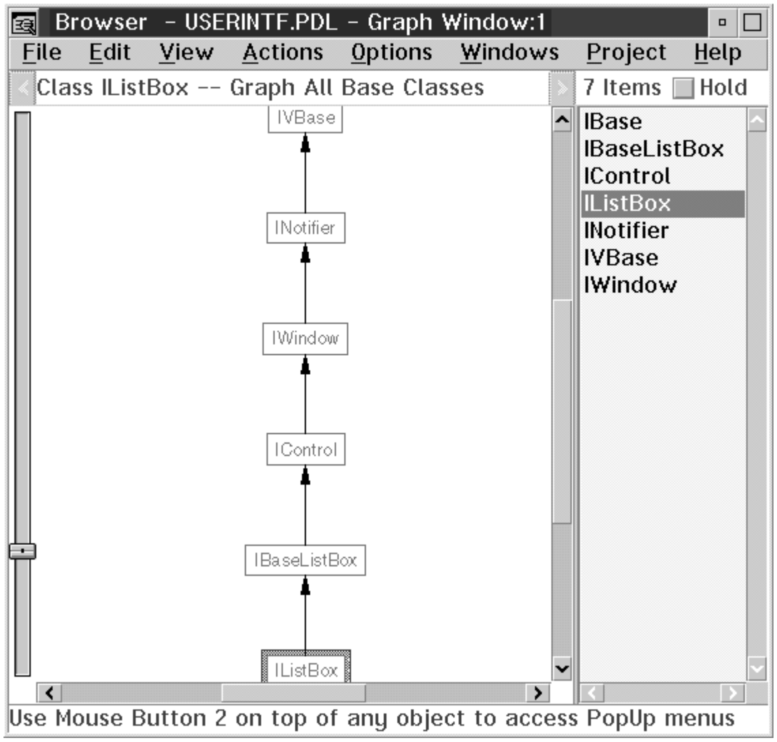

Browser - USERINTF.PDL - Graph Window:1
File Edit View Actions Options Windows Project Help
Class IListBox -- Graph All Base Classes
7 Items
Hold
IVBase
INotifier
IWindow
IControl
IBaseListBox
IListBox
IBase
IBaseListBox
IControl
IListBox
INotifier
IVBase
IWindow
Use Mouse Button 2 on top of any object to access PopUp menus

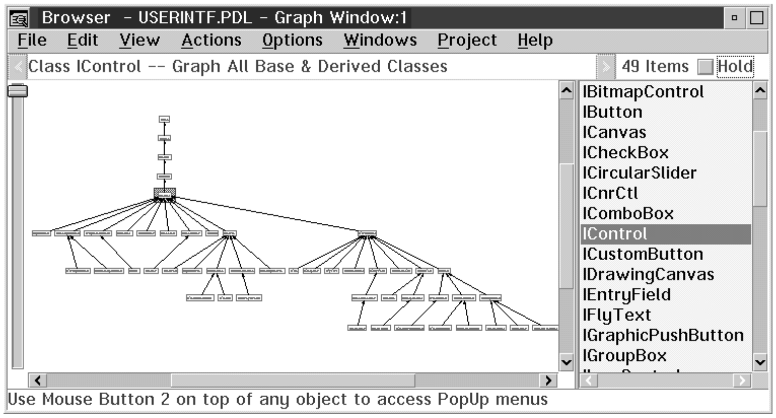
Figure 176. Graph window showing all base classes of the IListBox class

Finding Another Class

Now you may want to investigate what other controls are in the User Interface classes of the IBM VisualAge C++ Open Class Library. Maybe there is a special color Listbox.

1. Select the `IControl` class object (either using the node on the graph or the alphabetical graph-list item) using Mouse Button 2. The **Class** PopUp menu appears.
2. Select the **Graph <u>A</u>ll Base & Derived Classes** menu item.

The new graph shows all the classes that inherit from `IControl`. However, the graph is too large to view all at once.


Browser - USERINTF.PDL - Graph Window:1
File
Edit
View
Actions
Options
Windows
Project
Help
Class IControl -- Graph All Base & Derived Classes
49 Items
□ Hold

IBitmapControl
IButton
ICanvas
ICheckBox
ICircularSlider
ICnrCtl
IComboBox
IControl
ICustomButton
IDrawingCanvas
IEntryField
IFlyText
IGraphicPushButton
IGroupBox

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 177. Graph window showing all base and derived classes of IControl class

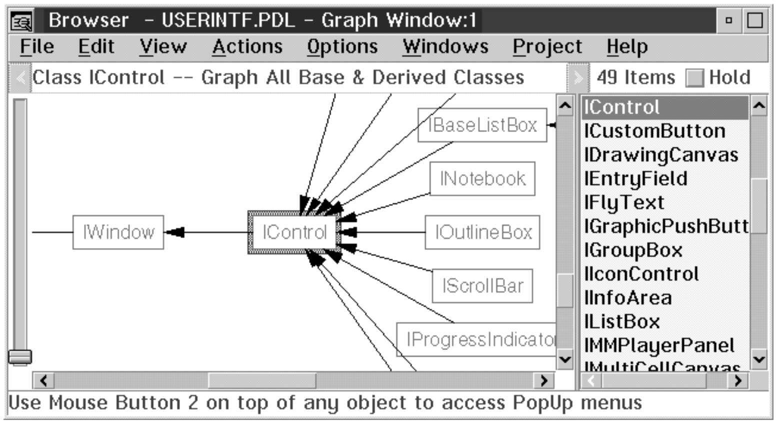
Changing the View of a Graph

Some graphs are better viewed using a horizontal organization rather than vertical (the default). For example, wide trees with long names are better shown horizontal, while tall trees with short names are better shown vertical. The inheritance graph of `IControl` falls into this category. To change the organization of the nodes from vertical to horizontal:

1. Click Mouse Button 2 on the background of the Graph window to display the **Background** PopUp menu. (Note that the List window also has a **Background** PopUp menu).
2. Select the **Horizontal** menu item.

Now adjust the zoom factor of the graph:

1. Display the Graph window **Background** PopUp menu again.
2. Select the **Max Zoom in** menu item.
3. Display the PopUp menu again
4. Select the **Center** menu item in order to center the selected node (Class `IControl`) in the **Graph Area** of the Graph window.



Use Mouse Button 2 on top of any object to access PopUp menus

Figure 178. Graph window showing all base and derived classes of IContrl class

Note the difference between the vertical organization of the nodes in the previous graph and the horizontal organization in this graph.

Investigating the Members of a Class

Now you need to find out more details about `IListBox` and its members.

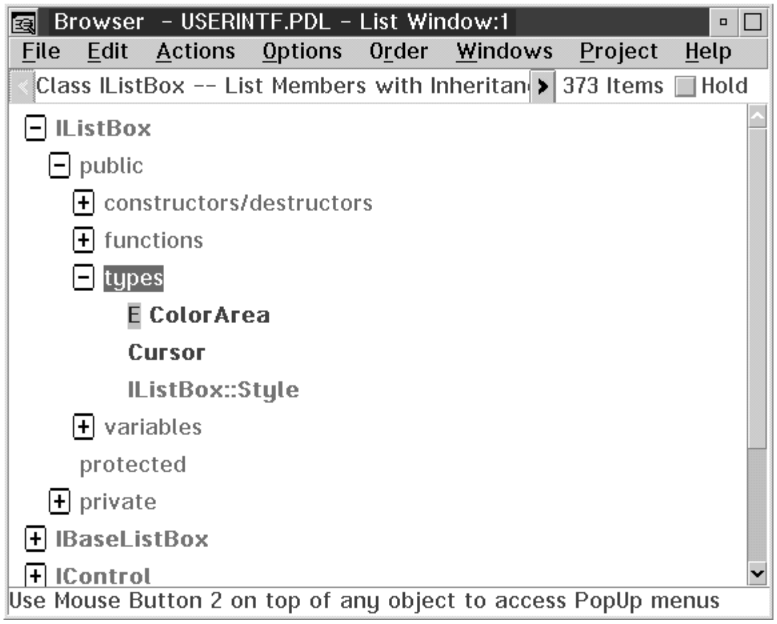
1. Select either the graph node or the graph-list item for `IListBox`.
2. Select **List Members with Inheritance** on the **Class** PopUp menu.

A List window appears showing the contents view of `IListBox`. This is called a container view. It can be expanded to show all the members of `IListBox` or any of the members of the base classes of `IListBox`.

1. Select the + icon on IListBox to expand and show the classes public, protected, and private sections.
2. Expand the + icon on public to show the constructors, destructors, functions, variable, and types.
3. Expand the + icon on types.

Note how different colors and highlighting techniques are used for the various kinds of program elements.

Note that the classes are arranged in a depth first tree traversal of the classes inheritance hierarchy.



Use Mouse Button 2 on top of any object to access PopUp menus

Figure 179. List window showing all members with inheritance of IListBox class

Customizing Program Elements

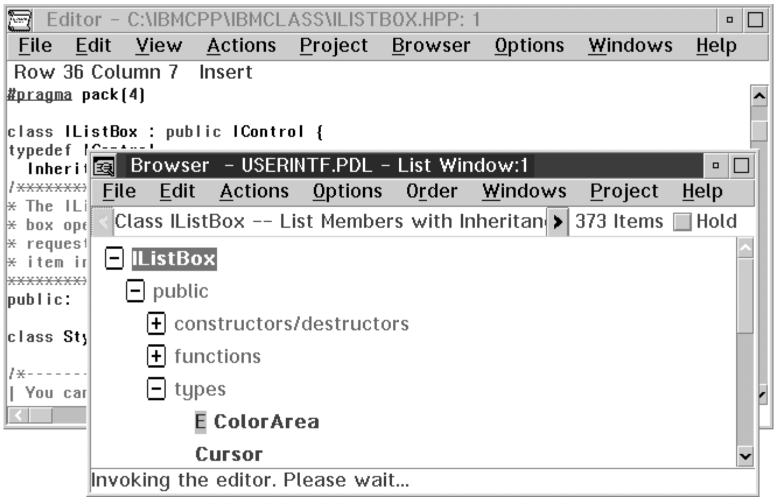
Each program element has its own unique customization for either the Graph or List windows. You can associate different colors and shapes to different program element types. Also, each object has a double-click action associated with it, as indicated by a check mark on the PopUp menu. For information on changing the Graph or List window settings, see “Changing the Default Graph Window Settings” on page 580 or “Changing the Default List Window Settings” on page 569.

In the List window, some listed items have a one-letter attribute, and bold is used to highlight function names from the rest of the string. The one letter attributes are: V-Virtual (on functions), PV-Pure Virtual (on functions), C-Const (on functions), S-Static (on functions and variables), and E-Enum (on types).

Editing Files from the Browser

The default double-click action for class and function objects is to edit the file containing the object's definition. You can make changes to the object's definition or just read the code/comments in the source code.

1. Double-click on `IListBox` to load `ilistbox.hpp` into the VisualAge Editor. Note how the VisualAge Editor places you at the start of the class definition.
2. Close the VisualAge Editor


Editor – C:\IBMCPP\IBMCLASS\ILISTBOX.HPP: 1
File Edit View Actions Project Browser Options Windows Help
Row 36 Column 7 Insert
Browser – USERINTF.PDL – List Window:1
File Edit Actions Options Order Windows Project Help
Class IListBox -- List Members with Inheritan
373 Items
Hold
IListBox
public
constructors/destructors
functions
types
E ColorArea
Cursor

```
#pragma pack(4)

class IListBox : public IControl {
typedef I
  Inheri
/*******
* The ILi
* box ope
* reques
* item ir
********
public:

class Sty

/*------
| You car
```

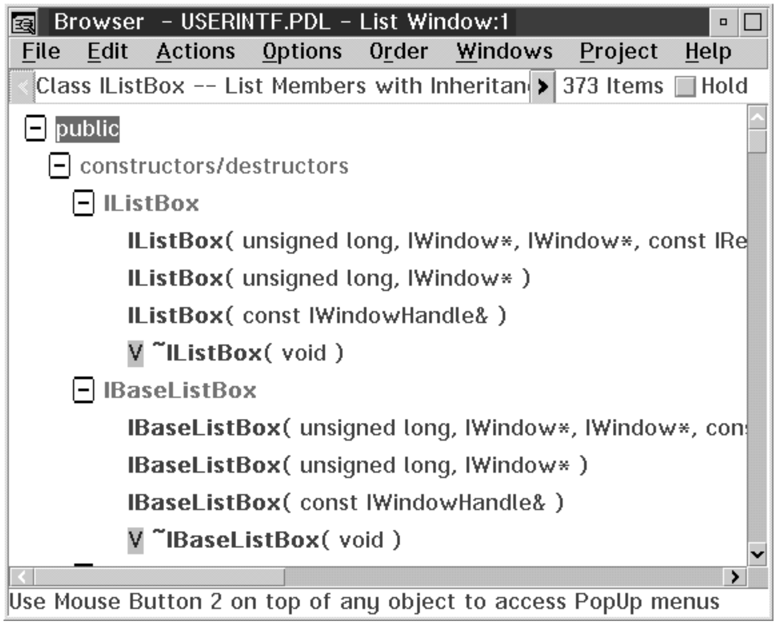
Invoking the editor. Please wait...

Figure 180. List window and VisualAge Editor

Organizing the Information in a List Window

You can organize the information about `IListBox` in different ways. For example, suppose that you are only interested in the public members:

1. Select the **Order** PullDown menu.
2. Select the **Access** menu item to change the order of the list items. Previously, the order was by **Class**; the class was at the highest level, followed by the access type (public, protected, private), followed by the program element type. Now, the access methods are placed at the highest level, followed by program element type, followed by class.
3. Double click on public to expand the whole tree under that label.
4. Display the List window PopUp menu by clicking Mouse Button 2 on the List window background.
5. Select the **Expand All** menu item.


Browser - USERINTF.PDL - List Window:1
File
Edit
Actions
Options
Order
Windows
Project
Help
Class IListBox -- List Members with Inheritan
373 Items
Hold

```
⊟ public
   ⊟ constructors/destructors
      ⊟ IListBox
            IListBox( unsigned long, IWindow*, IWindow*, const IRe
            IListBox( unsigned long, IWindow* )
            IListBox( const IWindowHandle& )
            V ~IListBox( void )
      ⊟ IBaseListBox
            IBaseListBox( unsigned long, IWindow*, IWindow*, con
            IBaseListBox( unsigned long, IWindow* )
            IBaseListBox( const IWindowHandle& )
            V ~IBaseListBox( void )
```

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 181. List window showing organization by access

Compare the order of the contents in this List window with the previous List window contents.

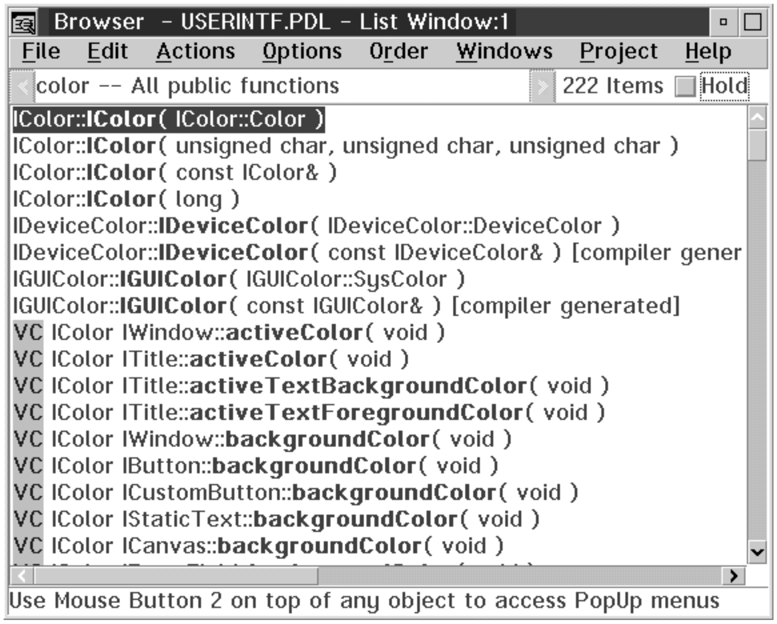
For more information on ordering a container view, see "Ordering the Contents of a Container View" on page 566.

Finding A Function

Now search for all the functions that contain the word `color`.

1. Select the **Actions** PullDown menu.
2. Select the **Search...** menu item. The **Search Database** dialog appears.
3. Type `color` as the search text.
4. Select **Functions** as the objects to search for.
5. Select **Public** as the access type.
6. Select **All** as the type of function.
7. Select the **Search** PushButton.

A list of all the function names that have the text string `color` are displayed. Scroll the list of functions until you find a function that allows you to set the color (`setColor`).


Browser - USERINTF.PDL - List Window:1
File Edit Actions Options Order Windows Project Help
color -- All public functions
222 Items
Hold
Use Mouse Button 2 on top of any object to access PopUp menus

IColor::**IColor**(IColor::Color)
IColor::**IColor**(unsigned char, unsigned char, unsigned char)
IColor::**IColor**(const IColor&)
IColor::**IColor**(long)
IDeviceColor::**IDeviceColor**(IDeviceColor::DeviceColor)
IDeviceColor::**IDeviceColor**(const IDeviceColor&) [compiler gener
IGUIColor::**IGUIColor**(IGUIColor::SysColor)
IGUIColor::**IGUIColor**(const IGUIColor&) [compiler generated]
VC IColor IWindow::**activeColor**(void)
VC IColor ITitle::**activeColor**(void)
VC IColor ITitle::**activeTextBackgroundColor**(void)
VC IColor ITitle::**activeTextForegroundColor**(void)
VC IColor IWindow::**backgroundColor**(void)
VC IColor IButton::**backgroundColor**(void)
VC IColor ICustomButton::**backgroundColor**(void)
VC IColor IStaticText::**backgroundColor**(void)
VC IColor ICanvas::**backgroundColor**(void)

Figure 182. List window showing all public functions that contain the "color" string

For more information on the **Search Database** facility, see "Searching for Objects in the Entire Browser Database" on page 598.

Showing the VisualAge C++ Documentation for a Particular Function

To find out more information about the `setColor` function with regards to Listboxes:

1. Click Mouse Button 2 on the `IListBox::setColor` function.
2. Select the **Show Documentation** menu item from the **Function** PopUp menu. The VisualAge C++ online reference manual opens to this function.

More About the PopUp Menu Actions

You can also perform other actions from the various PopUp menus.

On the **Function** PopUp menu, select any of:

- **Graph All Callers & Callees** to display the call graph for that function. For an example, see “Viewing Call Chains” on page 613.
- **List Possible Exceptions Thrown** to see the exceptions that could be thrown by this function. For an example, see “Listing All the Exceptions That A Function May Encounter” on page 611.
- **List Overriding Derived Classes** to see the derived classes that override this function. For an example, see “Listing Overriding Derived Classes” on page 610.

From the **Function** or **Class** PopUp menus, select:

- **List Friends**, **List Friendships**, or **List Instantiations** of a template. Note you can only list friends from a class, not a function. For examples, see “Listing All Friends of a Class” on page 606, “Listing All Friendships of a Class or Function” on page 607, and “Listing Instantiations of Classes or Functions” on page 611.

The above actions help you to understand a class library, someone else's code, or your own programs. 📖 For more information, see “PopUp Menus” on page 648.

Invoking Actions Again

The last action done in any window is always saved. For example:

1. Select the **Actions** PullDown menu.
2. Select the **List All Files** menu item. A list of all the files used to create the User Interface classes of the IBM VisualAge C++ Open Class Library are listed in a List window.
3. Select the **Actions** PullDown menu again.
4. Select the **Previous** menu item. The results of your last action are displayed.
5. Select the **F6** key to go back to the list of files.

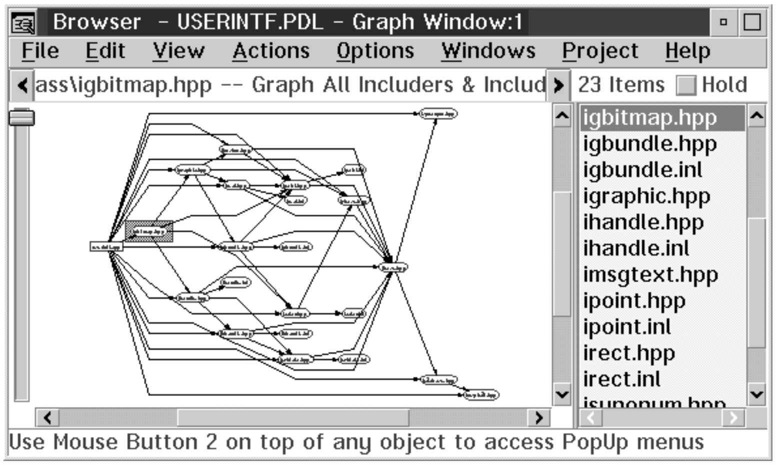
By selecting the **Previous** or the **F6** key, you can toggle back and forth between displays.

Graphing Include File Relationships

To find out where classes, types, variables, and code included in your program come from, you will need to look at an include file relationship.

1. Click Mouse Button 2 on the `igbitmap.hpp` file to invoke the **File** PopUp menu.
2. Select the **Graph All Includers & Includees** menu item.

A graph of the include file structure is displayed. This graph indicates which files include other files.



Browser - USERINTF.PDL - Graph Window:1
File
Edit
View
Actions
Options
Windows
Project
Help
ass\igbitmap.hpp -- Graph All Includers & Includ
23 Items
Hold

igbitmap.hpp
igbundle.hpp
igbundle.inl
igraphic.hpp
ihandle.hpp
ihandle.inl
imsgtext.hpp
ipoint.hpp
ipoint.inl
irect.hpp
irect.inl
isynonym.hpp

Use Mouse Button 2 on top of any object to access PopUp menus

Figure 183. Graph window showing all includers and includees of igbitmap.hpp

Returning to Previous Queries/Displays

If you performed a query or had a display that you want to return to, use the **History** window.

1. Select the **Windows** PullDown menu.
2. Select the **History...** menu item.

The last 40 object-actions that were performed are listed, and you can perform any of those actions again by double-clicking on the object-action pair.

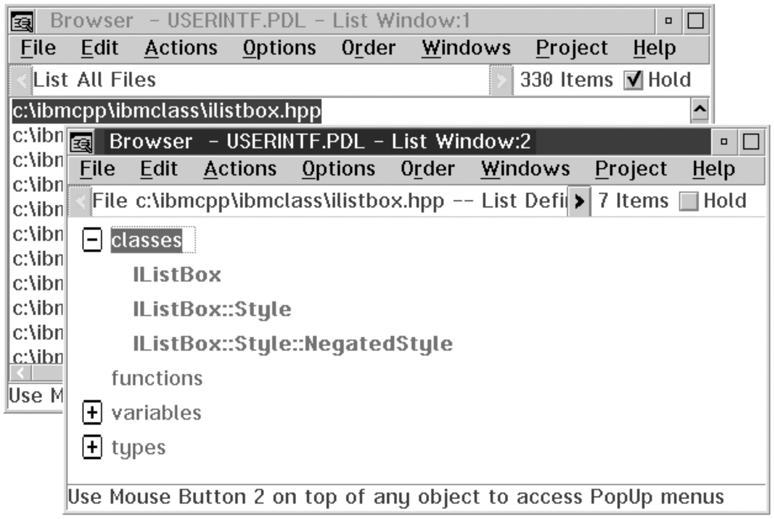
For more information on the **History** facility, see “The History Window” on page 599.

Keeping Your Windows From Being Replaced

By default, the Browser replaces the contents of the current window. You can keep the current window and do the next action in a new window by using the **Hold** CheckBox. From the List window:

1. Select the **Actions** PullDown menu.
2. Choose the **List All Files** menu item.
3. Select the **Hold** CheckBox on this window.
4. Click Mouse Button 2 on `ilistbox.hpp` from the list of files to invoke the **File** PopUp menu.
5. Choose the **List Defined Objects** menu item. A new List window will open with the results. You can have a maximum of four List windows and four Graph windows open at any time.
6. Double-click on the classes label to see all the classes defined in `ilistbox.hpp`.

Use the **Windows** PullDown menu to get back to another window.



Browser - USERINTF.PDL - List Window:1
File Edit Actions Options Order Windows Project Help
List All Files
330 Items ☑ Hold
c:\ibmcpp\ibmclass\ilistbox.hpp
c:\ibn
c:\ibn
c:\ibn
c:\ibn
c:\ibn
c:\ibn
c:\ibn
c:\ibn
c:\ibn
c:\ibn
Use M
Browser - USERINTF.PDL - List Window:2
File Edit Actions Options Order Windows Project Help
File c:\ibmcpp\ibmclass\ilistbox.hpp -- List Defi
7 Items □ Hold
⊟ classes
IListBox
IListBox::Style
IListBox::Style::NegatedStyle
functions
⊞ variables
⊞ types
Use Mouse Button 2 on top of any object to access PopUp menus

Figure 184. Two List windows open at same time

Changing the Default Settings for List and Graph Windows

You can alter the default settings to the List and Graph windows by selecting:

- **Fonts...** to change the font of the List window. 📖 See "Changing Fonts" on page 592.
- **List Fonts...** and **Node Fonts...** to change the fonts in the Graph window. 📖 See "Changing Fonts" on page 592.
- **List Window...** to change the double-click actions for the objects in the List window, the colors used by the List window, and the text style used by the List window. 📖 See "Changing the Default List Window Settings" on page 569.
- **Graph Window...** to change the double-click actions for the objects in the Graph window, the colors used by the Graph window, the node and line shape used by the graph, and the size of the bitmap to save. 📖 See "Changing the Default Graph Window Settings" on page 580.

All options are saved between uses of the Browser to the Browser profile (`icsbrs.ini`).

Manipulating Graphs

You can perform many actions on a graph. You can:

- Select **Show Inheritance Graph** from the **Actions** PullDown PullDown menu to get a large graph.
- Select **Overview...** from the **View** PullDown menu to get the overview window for navigating around large graphs.
- Drag the Zoom slider on the left side of the window up and down to zoom the graph in and out.
- Press Mouse Button 1 and drag it. This creates a dotted selection box. There is a PopUp for the selected area to Zoom, Print, etc. the selected area.
- Select **Center** from the **Graph** PopUp menu to center the currently selected node on the graph area display.

You can print the graph to a single page or across multiple pages. You can also print a selcted zone in the graph. 📖 For more information on printing graphs, see "Printing and Saving your Graphs" on page 585.

You can save graphs to an OS/2 bitmap file, or copy them to the clipboard for including graphs in your own documentation.

Of course, there is Print, Save to a file, and Copy to the clipboard in the List Window too.

The Browser and WorkFrame

The Browser is fully integrated with WorkFrame.

- You can browse a WorkFrame project.
- You can invoke Browser actions from the VisualAge Editor and the Debugger, and perform browse actions on words highlighted in those tools.
- Other project actions appear in the **Project** PullDown.

Browser: A Tour

47

Trouble Shooting

This chapter helps you solve problems you may encounter while using the Browser.

The Browser Won't Start

If you cannot start the Browser either by using the `icsbrs` on the command line or by launching the **Browser** action from any VisualAge C++ tools menubar, then delete the `icsbrs.ini` file and start the Browser.

This file contains your user defined settings, such as color, size, placement, paths, etc. You can find this file in the `\OS2` directory on your boot partition, unless you specified a different location to store this file through the **Browser Settings Paths** NoteBook page. 📖 For more information on this dialog, see "Changing Paths Used by the Browser" on page 588.

Error Loading a .EXE, .DLL, or .LIB file

If you are having problems loading an .EXE, a .DLL, or a .LIB file, the file may have been created using a back-level version of the compiler. Recompile the source with the VisualAge C++ compiler and linker version 3.0.

Error Loading a .BRS File

The .BRS files generated with the C Set ++ V2.1 compiler and linker are not compatible with the VisualAge C++ Version 3.0 compiler and linker. You will need to recompile and link your program using the VisualAge C++ compiler and linker version 3.0 for OS/2.

Error Loading a .PDB File

If your .PDB file was created using the VisualAge C++ compiler for AIX version 3.1, you will need to recompile and link your program using the VisualAge C++ compiler and linker version 3.0 for OS/2. The AIX and OS/2 .PDB files are not compatible.

If the above is not the problem, please contact your IBM service representative.

Adding Files to the Load ▸ and Merge ▸ Menus Doesn't Work

You need to create an ASCII file called `brsmenu.txt`. 📖 See “Loading Files into the Browser” on page 593 or “Merging Files” on page 595 for instructions on how to create this file.

If you have already created this file, make sure that it is located in a directory in your `DPATH`. You may need to reboot if you just added the directory to the `DPATH` in your `config.sys` file.

If you have the `brsmenu.txt` file in a directory in the `DPATH`, but the files are still not displayed in the **Load** ▸ and **Merge** ▸ Cascade menus, make sure that there is a blank line at the end of the `brsmenu.txt` file.

The Graph Zone Will Not Maximum Zoom

If you select a region of a graph and perform a **Zoom in**, but the select region does not fill the entire client area of the Graph window, this is not an error. The graph can only be zoomed to its maximum size allowed, it cannot zoom in beyond that. In other words, you have selected a region of the graph that is smaller than the maximum size of the Graph window client area.

48 Browser Fast-Path Keys and Menu Descriptions

This chapter lists the combinations of keys that you can use to perform Browser functions, the List and Graph window PullDown and PopUp menus, and the Object PopUp menus.

- “Fast-Path Keys”
- “PullDown Menus” on page 640
- “PopUp Menus” on page 648

Fast-Path Keys

The following two lists outline the fast-path keys for using the Browser. The format is as follows: `<key>` or `<key> – <key>`.

Note: The letter keys are shown in uppercase for clarity. You do NOT have to press the **Shift** key unless this key is specifically mentioned.

Application-Provided Keys:

F3 Close the Browser session.
F6 Perform the previous object-action pair.
F7 Expand all the items in the List window.
F8 Collapse all the items in the List window.
Ctrl-C Centers the currently selected node on a graph
Ctrl-E Edit the currently selected definition or file.
Ctrl-F Initiate **Find** dialog to find text in the current window.
Ctrl-G Graph all the base and derived classes for the currently selected class.
Ctrl-H Show the documentation for the currently selected class or function.
Ctrl-L List all members with inheritance for the currently selected class.
Ctrl-N Find next instance of text in the current window.
Ctrl-S Initiate **Search** dialog to search for text in the loaded database.
Ctrl-Insert Copies the current window contents to the clipboard.
Ctrl-+ Zoom in on a graph.
Ctrl-– Zoom out on a graph.
Alt-+ Maximum zoom in on a graph
Alt-– Maximum zoom out on a graph

System-Provided Keys:

Alt-F4	Close window.
Alt-F7	Move window.
Alt-F8	Size window.
Alt-F9	Minimize window.
Alt-F10	Maximize window.

PullDown Menus

The PullDown menus can be found on both the List and Graph window, although some PullDowns are specific to each window.

- "**File** PullDown Menu" on page 641.
- "**Edit** PullDown Menu" on page 642.
- "**View** PullDown Menu" on page 643.
- "**Actions** PullDown Menu" on page 644.
- "**Options** PullDown Menu" on page 645.
- "**Order** PullDown Menu" on page 646.
- "**Windows** PullDown Menu" on page 646.
- "**Project** PullDown Menu" on page 647.
- "**Help** PullDown Menu" on page 647.

File PullDown Menu

Menu Item	Description	Window
Load ►	Loads your program or Browser database files, or you can quickly load the classes that make up the VisualAge C++ Open Class Library (**User Interface Classes**, **Collection Classes**, **I/O Stream Classes**, **Complex Math Classes**, **Database Access Classes**, and **Application Support Classes**). 📖 See "Loading Files into the Browser" on page 593 for more information on loading files into the Browser. See "Adding Menu Items to the **Load** ► and **Merge** ► Cascade menus" on page 619 for information on adding your own menu items.	List, Graph
Merge ►	Extends programs with additional controls or features from another program, or merge the classes that make up the VisualAge C++ Open Class Library (**User Interface Classes**, **Collection Classes**, **I/O Stream Classes**, **Complex Math Classes**, **Database Access Classes**, and **Application Support Classes**). 📖 See "Merging Files" on page 595 for more information on merging files into the Browser. See "Adding Menu Items to the **Load** ► and **Merge** ► Cascade menus" on page 619 for information on adding your own menu items.	List, Graph
Refresh	Updates your current Browser database with the best source of data possible. 📖 See "Updating the Browser Database" on page 618.	List, Graph
Save Graph As...	To save the Graph Area to an OS/2 bitmap. 📖 See "Printing and Saving your Graphs" on page 585.	Graph
Save List As...	To save the contents of the List window or **List Area** of a Graph window to an ASCII file. 📖 See "Printing and Saving your Lists" on page 575 or "Printing and Saving your Graphs" on page 585.	List, Graph
Print...	Prints the List window contents to the printer. 📖 See "Printing and Saving your Lists" on page 575.	List
Print ►	Prints the Graph window contents to the printer (**One Page...**, **Multiple Pages...**, **Client...**, and **Zone...**). 📖 See "Printing and Saving your Graphs" on page 585.	Graph

Browser: Edit Menu

Menu Item	Description	Window
New Window	Creates another List window if launched from a List window, or another Graph window if launched from a Graph window. The new window will be blank.	List, Graph
Copy Window	Copies the current List or Graph window contents and puts them into a new List or Graph window.	List, Graph
Exit Browser (F3)	Ends your current Browser session. All List and Graph windows are closed.	List, Graph

Edit PullDown Menu

Menu Item	Description	Window
Find... (Ctrl-F)	Launches the **Find** dialog. You can search the objects in the current window for the first instance matching the text string that you entered into the **Find** dialog. 📖 See “Finding Objects in the Current Window” on page 597.	List, Graph
Find Next (Ctrl-N)	Searches the objects of the current window for the next instance of the text string that was last entered into the Find dialog. It does not search the entire Browser database. Note that the search begins from the current selection position in the window.	List, Graph
Copy	Copies the currently selected line to the clipboard.	List
Copy All (Ctrl-Insert)	Copies the entire contents of the List or Graph window to the clipboard.	List, Graph

View PullDown Menu

Menu Item	Description	Window
Overview...	Launches a window which displays a miniature version of the current graph. 📖 See “Getting a Graph Overview” on page 578.	Graph
Zoom in (Ctrl-+)	Increases the magnification of the current graph by approximately 10%.	Graph
Zoom out (Ctrl-–)	Decreases the magnification of the current graph by approximately 10%.	Graph
Max Zoom in (Alt-+)	Increases the current graph to the maximum magnification.	Graph
Max Zoom out (Alt-–)	Decreases the current graph to the minimum magnification.	Graph
Center (Ctrl-C)	Moves the currently selected node to the center of the graph area.	Graph
Vertical	Draws the graph with a vertical orientation. That is, the nodes are read from top to bottom.	Graph
Horizontal	Draws the graph with a horizontal orientation. That is, the nodes are read from left to right.	Graph
Weighting ►	Adjusts where the nodes on the graph are displayed depending on the **Vertical**/**Horizontal** setting. **Top** - Aligns all the root nodes at the top/right of the graph. **Center** - Aligns the nodes around the center of the graph. **Bottom** - Aligns all the leaf nodes to the bottom/left of the graph.	Graph

Browser: Actions Menu

Actions PullDown Menu

Menu Item	Description	Window
Previous (F6)	Returns to the last object-action pair performed in that window. 📖 See “The History Window” on page 599.	List, Graph
Search... (Ctrl-S)	Launches the Search dialog in order to search the current Browser database for classes, functions, types, variables, and files. 📖 See “Searching for Objects in the Entire Browser Database” on page 598.	List, Graph
Show Inheritance Graph	Displays the tree structure that represents how all the defined classes in the entire Browser database are related by inheritance. 📖 See “Viewing Class Relationships” on page 612.	List, Graph
Show Include File Graph	Displays a tree structure that represents how all the source files in the current Browser database are related by the C and C++ include mechanism. 📖 See “Viewing Call Chains” on page 613.	List, Graph
List All Classes	Creates an alphabetical list of all the classes defined or declared in the Browser database.	List, Graph
List All Files	Creates an alphabetical list of all the source files used to create the currently loaded Browser database.	List, Graph

Options PullDown Menu

Menu Item	Description	Window
Fonts...	Launches the **List Window Fonts** dialog to change the fonts used by the Browser.	List
Node Fonts...	Launches the **Graph Node Fonts** dialog in order to change the default text font used in the nodes.	Graph
List Fonts...	Launches the **Graph List Fonts** dialog in order to change the default text font used in the **Graph List Fonts** of the Graph window.	Graph
List Window...	Launches the **List window Settings** NoteBook. It has three pages: **Settings**, **Colors**, and **Styles**. 📖 See “Changing the Default List Window Settings” on page 569.	List, Graph
Graph Window...	Launches the **Graph window Settings** NoteBook. It has four pages: **Settings**, **Colors**, **Styles**, and **Bitmap**. 📖 See “Changing the Default Graph Window Settings” on page 580.	List, Graph
Browser...	Launches the **Browser Settings** NoteBook. It has two pages: **Paths** and **Help Level**. 📖 See “Changing Browser Settings” on page 587.	List, Graph

Order PullDown Menu

Menu Item	Description	Window
Class	Reorders the current objects in the list. The base classes are at the highest level, and class members are listed by type below. 📖 See "Ordering the Contents of a Container View" on page 566.	List
Access	Reorders the current objects in the list with the access method at the highest level. 📖 See "Ordering the Contents of a Container View" on page 566.	List
Type	Reorders the current objects in the list with the type of object at the highest level. 📖 See "Ordering the Contents of a Container View" on page 566.	List

Windows PullDown Menu

Menu Item	Description	Window
History...	Launches the History window. It lists the last 40 object-action pairs performed. 📖 See "The History Window" on page 599.	List, Graph
List 1 **List 2** **List 3** **List 4**	Gives you quick access to the List windows that you opened. If you do not have any List windows open, these menu items do not appear.	List, Graph
Graph 1 **Graph 2** **Graph 3** **Graph 4**	Gives you quick access to the Graph windows that you opened. If you do not have any Graph windows open, these menu items do not appear.	List, Graph

Project PullDown Menu

Menu Item	Description	Window
IBM WorkFrame actions	Items in this menu are dependent on what you have installed and how you launched the Browser.	List, Graph

Help PullDown Menu

Menu Item	Description	Window
Help Index	Launches the index for the Browser online help.	List, Graph
General Help	Launches the online help panel for either the List window or Graph window depending on which window you chose this item from.	List, Graph
Using Help	Launches help information for using the Information Presentation Facility (IPF).	List, Graph
How Do I...	Launches the **How Do I...** information which provides step-by-step instructions on how to perform tasks using the Browser.	List, Graph
VisualAge C++ Documentation Cascades	Launches the various online documentation for the VisualAge C++. These cascades are populated with the documents depending on what you have installed.	List, Graph
Product Information	Provides information about this release of the Browser.	List, Graph

PopUp Menus

The Browser has two types of PopUp menus:

- 🕮 “PopUp Menu Items for List and Graph Windows.”
- 🕮 “Object PopUp Menu Items” on page 649.

PopUp Menu Items for List and Graph Windows

You can access the List and Graph window PopUps by clicking Mouse Button 2 on the background area of the window. The following items will appear, depending on the window you accessed the PopUp from:

Menu Item	Description	Window
Show Inheritance Graph **Show Include File Graph** **List All Classes** **List All Files**	🕮 See “**Actions** PullDown Menu” on page 644.	List, Graph
Expand All	Expand all items in the List window container view.	List (container view)
Collapse All	Collapse all items in the List window container view.	List (container view)
Overview... **Zoom in** **Zoom out** **Max Zoom in** **Max Zoom out** **Center** **Vertical** **Horizontal** **Weighting ▸**	🕮 See “**View** PullDown Menu” on page 643.	Graph

Object PopUp Menu Items

You can access the Object PopUps can be accessed by clicking Mouse Button 2 on any program element displayed in the List or Graph windows. The following items will appear, depending on which window you accessed the PopUp from:

Menu Item	Description	Object
Collapse	Collapse the currently selected object entirely if it has a minus icon (-) next to it.	class, label
Edit Definition (Ctrl-E)	Launches either the editor as defined for your project, if you have started the Browser from a WorkFrame project, or the VisualAge Editor. The file containing the definition of the currently selected object is loaded into the editor. The editor is positioned to the first line of the definition of this object.	class, function, type, variable
Edit File (Ctrl-E)	Launches either the editor as defined for your project, if you have started the Browser from a WorkFrame project, or the VisualAge Editor. The file is loaded into the editor.	file
Expand	Expands the currently selected object entirely if it has a plus icon (+) next to it.	class, label
Expand Typedef	Creates a list which contains successive expansions of a typedef, until it contains only fundamental types.	type
Graph All Base & Derived Classes (Ctrl-G)	Creates an inheritance graph of all the base classes of the selected class, and all the classes that derive from the selected class, in one relationship graph. These include both direct and indirect base and derived classes.	class
Graph All Base Classes	Creates an inheritance graph of all the base classes of the selected class. These include both direct and indirect base classes.	class
Graph All Derived Classes	Creates an inheritance graph of all the classes that are derived from the selected class. These include both direct and indirect derived classes.	class
Graph Immediate Derived Classes	Create a Graph window which displays one level of derived classes for this class.	class
Graph All Callers & Callees	Creates a graph of all the functions that call the currently selected function and the functions that are called by the currently selected function. These include the functions that the selected function calls and is called from, both directly or indirectly.	function

Browser: Object PopUp Menus

Menu Item	Description	Object
Graph All Callers	Creates a graph of all the functions that call the currently selected function. These include the functions which call the selected function directly, and those functions that call it indirectly through other functions.	function
Graph All Callees	Creates a graph of all the functions that the currently selected funtion calls. These include the functions that the selected function calls directly or indirectly.	function
Graph Immediate Callers & Callees	Creates a graph of all the functions that either call the selected function directly or is called from directly.	function
Graph All Includers & Includees	Creates a graph of all the files included by the currently selected file and all the files that include the currently selected file. These include both direct and indirect inclusion.	file
Graph All Includers	Creates a graph of all the files that include the currently selected file. These include both direct and indirect inclusion.	file
Graph All Includees	Creates a graph of all the files that are included by the currently selected file. These include both direct and indirect inclusion.	file
List Class Members with Inheritance	Creates a List window container view of all the members of the classes and base classes for the class that the selected function is a member of.	function
List Defined Objects	Creates a list of all the Browser objects which are defined in the selected file.	file
List Friends	Lists all the friends of the currently selected class. A friend of a class is a function that has been granted access to the private members of the class. A friend class obtains access to private members for all its member functions. The result of this action is a List window container view with functions and classes as the two labels in the container.	class
List Friendships	Lists all the friendships that are defined for the currently selected class or function. You can grant friendships to either a single function or member function at a time, or to all the member functions of a class at once. This action results in a list of all those classes that have granted the selected class or function friendship.	class, function

Menu Item	Description	Object
List Immediate Callers & Callees	Creates a list of all the functions that directly call or are directly called by the selected function.	function
List Instantiations	Creates a list of the instantiations of the selected class or function template.	class, function
List Implementing Files	Create a list of files that contain definitions for the class, and for any of the members of the class.	class
List Members with Inheritance (Ctrl-L)	Creates a List window container view of all the members for the selected class and its base classes.	class
List Overriding Derived Classes	Creates a list of all the derived classes of the class of which this function is a member which have a member function that overrides this function.	function
List Possible Exceptions Thrown	Creates a list of all the possible types of exceptions that could be thrown by the selected function, or by any function that this function calls.	function
Show Documentation (Ctrl-H)	Launches the help panel in the VisualAge C++ class library documentation for the currently selected object.	class, function

Browser: Object PopUp Menus

Part 9. Managing Libraries

This part of the *User's Guide* describes tools that can help you manage libraries.

To create and maintain .LIB files, use the ILIB utility.

If you are using DLL files, you can use the DLLRNAME utility to globally rename a DLL, and the FWDSTAMP utility to maintain compatibility between old executables and new versions of a DLL.

49 Using ILIB

Use the IBM Library Manager (also referred to as ILIB in this reference) to create and maintain libraries of object code.

Library files are given the extension of .LIB (as in MYLIB.LIB). High Performance File System (HPFS) files with names that end with ".LIB" (as in MYLIBRARYFILE.NEW.LIB) are also supported.

ILIB works with standard libraries and OS/2 import libraries. It does not work with dynamic link libraries (DLLs).

Use the ILIB utility to:

- Create a new library (standard only)
- Add, delete, or replace modules in a library (import or standard)
- Copy object modules in a library to object files (from import or standard)
- List the contents of a library (import or standard)

Note: The Library Manager in VisualAge C++ is different from the one included in previous releases of C Set ++. The new Library Manager (ILIB) uses a new format for libraries that improves link time. It also features several options that the old Library Manager (referred to as LIB in previous releases) did not have:

```
/NOBACKUP
/NOBROWSE
/CONVFORMAT
```

ILIB does not support the `/PAGESIZE` option.

For compatibility with previous releases, the LIB utility is included in VisualAge C++ with the name LIBV2R1.EXE. You can also convert existing libraries created with LIB to the new library format using ILIB with the `/CONVFORMAT` option (📖 described in “/CONVFORMAT (Convert to New Format)” on page 669).

Running ILIB

Run ILIB by typing `ILIB` at the operating system prompt.

You can specify parameters in one of three ways:

1. Enter them directly on the command line.
2. Respond to prompts.
3. Put them in a text file called a response file and specify the file name after the ILIB command.

To enter more commands than can be conveniently entered on one line, type an ampersand (&) at the end of the line and press Enter to extend the command field to a new line. You can use the ampersand with all three input methods.

You can press Ctrl+C or Ctrl+Break at any time while running ILIB to return to the operating system. Interrupting ILIB before completion restores the library from a backup.

Notes:

1. When started, ILIB makes a backup copy of the original library in case it is interrupted or a mistake is made. Make sure you have enough disk space for both your original library and the modified copy.
2. The library must end with the extension .LIB. If an extension is not specified, the default extension, .LIB, will be appended. HPFS file names are supported. Hence, MYLIBRARYNAME.NEW.LIB is still a valid library. Note that this implies that MYLIBRARYNAME.NEW refers to MYLIBRARYNAME.NEW.LIB.
3. If you enter an input library name and follow it immediately with a semicolon (;), ILIB performs a consistency check on the library and takes no other action.

Using the Command Line

You can specify all the input ILIB needs on the command line. The syntax of the command line is:

```
ILIB [options] inlibrary [commands] [[,listfile] [, outlibrary]] [;]
```

options Options that affect the behavior of ILIB.

inlibrary The input library to be created or modified.

commands Commands used to add, delete, replace, copy, and move modules within the library.

listfile
The name for a listing file. If you don't specify a name, no file is created.

outlibrary
The output library created from the input library. If you don't specify an output library, your input library is replaced with the modified version (see below).

Commas are used to separate commands and options. The semicolon (;) is used to mark the end of the command line.

Using ILIB Prompts

If you don't provide input to ILIB on the command line, ILIB prompts you for the information it needs by displaying the following messages, one at a time:

PROMPT	ENTER
Library name	Name of the input library to be modified. If the library you specify does not exist, the following prompt appears: `Library does not exist.  Create library? (y or n)`
Operation(s)	Commands to modify the library. If no operations are specified, the input library is unchanged.
List file	Name for a listing file. If no listing file is specified, no listing file is created.
New Library Name	Name of the output library to be created from the input library. If no output library is specified, ILIB modifies the input library.

Enter the same information that you would enter when using the ILIB command line. You can enter ILIB options at any prompt.

Notes:

- ILIB waits for you to respond to each prompt before displaying the next prompt. If you notice that you have entered an incorrect response to a previous prompt, press Ctrl+C or Ctrl+Break to exit ILIB and begin again.
- A file name must be entered at the **Library name:** prompt. To choose a default response for any of the other prompts, press Enter. To choose default responses for all remaining prompts, type a semicolon (;) and press Enter.

Using an ILIB Response File

To provide input to ILIB with a response file, type:

```
LIB @responsefile;
```

where *responsefile* is the name of a file containing the same information that can be specified on the command line.

In a sense, a response file extends the command line to include everything in the response file. To split input to ILIB between the command line and a response file, put part of your input on the command line and specify a response file (preceding the response file name with the at sign (@)). The response file name can be any valid OS/2 file. To use special characters such as a space or the @ symbol, the filename must be enclosed in quotes.

ILIB responds to input you place in a response file just as it does to input you enter on a command line or after a prompt. Using a newline character in the response file is the equivalent of pressing the Enter key after an ILIB prompt.

A response file uses one text line for each prompt. To extend an ILIB command to multiple lines, end each line except the last with an ampersand (&). Responses must appear in the same order as the prompts. If a response for one of the prompts does not appear, the default is used.

Use a response file for:

- Complex and long commands you type frequently.
- Strings of commands that exceed the limit for command line length.

Specifying ILIB Parameters - Examples

The following examples show different methods for specifying parameters to ILIB.

The operations shown in each example create a new library, NEWLIB.LIB, and its listing file, NEWLIB.LST, from the existing MYLIB.LIB library. MYLIB.LIB is unchanged, but NEWLIB.LIB has these changes:

- The contents are case-insensitive.
- The module TIM is deleted.
- The object file SIMON.OBJ is appended as an object module with the name SIMON.
- The module KEHM is deleted and is replaced by a new KEHM which is appended after SIMON.
- The module LAM is copied into an object file named LAM.OBJ.

Command Line Method

At the operating system prompt, enter the following two lines.

```
LIB /I MYLIB, SIMON-TIM-+KEHM &
*LAM, NEWLIB.LST, NEWLIB;
```

ILIB Prompts Method

To have ILIB prompt you for input, enter ILIB with no parameters.

```
Library name: /I MYLIB
 Library does not exist.  Create library? (y or n)  y
Operations: +SIMON-TIM-+KEHM &
Operations: *LAM
List file: NEWLIB.LST
New Library Name: NEWLIB
```

Response File Method

First, create a response file with the following contents.

```
/I MYLIB
+SIMON-TIM-+KEHM &
*LAM
NEWLIB.LST
NEWLIB
```

Then, assuming the name of the response file is `response.fil`, invoke ILIB with:

```
   ILIB @response.fil;
```

Note that the lines in the response file match the entries you would have made with the prompting method. Even the ampersand character (&), the continuation character, is used in the same way.

Creating a New Library

To create a new library file, specify the name of the library file you want to create on the command line (or at the **Library name:** prompt when using ILIB prompts).

Note: A library file is automatically created if the library file name you specify is immediately followed by a command, comma, or semicolon. In this case, the prompt does not appear.

If the name you specify for the new library file already exists, ILIB assumes that you want to modify the existing file.

When you give the name of a file that does not currently exist without specifying any operations, ILIB displays the following prompt:

```
Library does not exist.  Create library? (y or n)
```

Type `y` to create the file; type `n` to terminate the ILIB run. If you specified an extension other than .LIB, the ILIB utility will try to append the .LIB extension to the entire file name. If a library name is not entered, ILIB will prompt you for a library name.

Modifying a Library

You can use ILIB to alter the contents of any object code library. For example, if you work with high level language libraries, you may want to replace a standard routine with your own version of the routine. You may also want to add a new routine to the standard library so that your routine is available along with the standard routines.

To modify an existing library file, specify the name of the library file you want to modify on the ILIB command line (or at the **Library name:** prompt when using ILIB prompts).

In the *commands* field, enter one or more commands to add, delete, or replace modules in the input library. Each command consists of a command character immediately followed by the name of the module or object file. Note that the Add command can be used to combine libraries as well as to add object files to a library.

ILIB creates a backup file of the library being modified if it already exists. This backup file has the same name as the original library with a .BAK filename extension.

Copying Object Modules to Object Files

To copy a module from a library file to an object file. specify the name of the library file on the ILIB command line (or at the *Library name:* prompt when using ILIB prompts).

To move or copy object modules, use the *commands* field on the ILIB command line:

Command	Action
Copy (*)	Copy the module to an object file and retain the module in the library.
Move (-*)	Copy the module to an object file and delete the module from the library.

Listing the Contents of a Library

Listings give you the exact names of modules and public symbols, allowing you to inspect the contents within a library.

To generate a listing file, enter the following on the command line (or at the appropriate ILIB prompt):

- The name of the library file in the *inlibrary* field.
- The name of the listing file in the *listfile* field.

When generating a listing file, the amount of detail can be varied. The level of detail is specified with the

```
    /Listlevel:n
```

option, with three different levels available.

Level **1** is the default. It is the fastest to generate and contains the least amount of information. All modules are listed in order of occurrence. For each module, the level 1 option:

- Shows the size of each module, and each module's file offset within the library.
- Lists all the public symbols defined in the module.
- Lists all external symbols referenced by the module.

Level **2** contains all the information of level 1. In addition, for each external symbol, level 2 shows which module in the library (if any) contains the required public symbols for resolving at link time. This can be overridden if a module is linked to another module that already contains the symbol.

Level **3** contains all the information of level 2. In addition, Level 3 displays:

- The technical characteristics of the library.
- A dump of the extended dictionary. This is useful to determine which modules will be implicitly linked in whenever a particular module is linked in.
- A dump of all browse information for each module in the library.

Note: If you are using the VisualAge C++ product, definitions with mangled names will be listed with the demangled form in brackets.

Sample Cross Reference Listing

```
LIB /LISTLEVEL:2 NEWLIB, NEWLIB.LST;
```

The command above directs ILIB to place a listing of the contents of NEWLIB.LIB into the file NEWLIB.LST. No path specification is given for NEWLIB.LST. By default, the file created is put in the current directory.

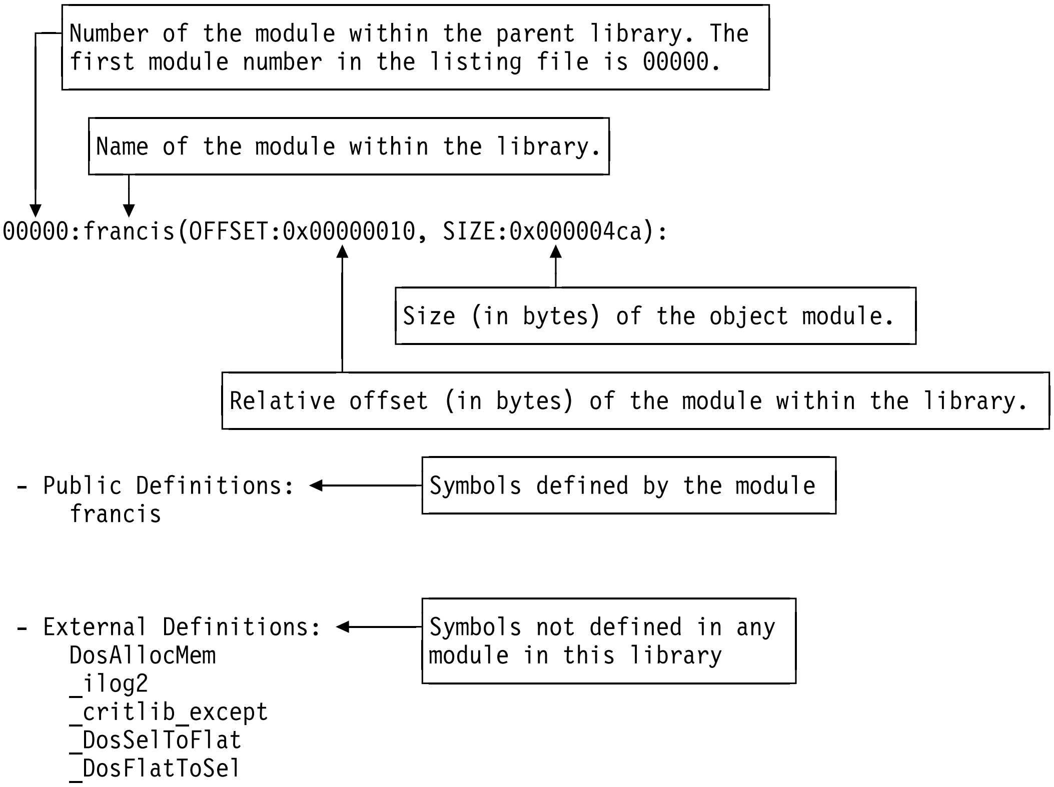
Listing Example

The syntax for generating a level 3 listing file is:

```
LIB /L:3 NEWLIB, NEWLIB.LST;
```

This command generates a listing file called NEWLIB.LST containing the following text:

```
IBM (R) Library Manager Version 3.00
Copyright (C) IBM Corporation 1991, 1995.  All rights reserved.

Library name : D:\TEMP\NEWLIB.LIB

Listing detail level : 3
```



```
00000:francis(OFFSET:0x00000010, SIZE:0x000004ca):

 - Public Definitions:
    francis


 - External Definitions:
    DosAllocMem
    _ilog2
    _critlib_except
    _DosSelToFlat
    _DosFlatToSel
```

Number of the module within the parent library. The first module number in the listing file is 00000.

Name of the module within the library.

Size (in bytes) of the object module.

Relative offset (in bytes) of the module within the library.

Symbols defined by the module

Symbols not defined in any module in this library

```
00001:lam (OFFSET:0x000004e0, SIZE:0x000001d1):
 - Public Definitions:
     lam

 - External Definitions:
     francis                            <- 00000:francis
     _critlib_except
     _DosSelToFlat
     _DosFlatToSel     Number and name of the module within the library
                       that defines the corresponding public symbol.


00002:hazlett (OFFSET:0x000006c0, SIZE:0x0000021a):
 - Public Definitions:
     hazlett

 - External Definitions:
     DosFreeMem
     _critlib_except
     _DosSelToFlat
     _DosFlatToSel
     _pBucketArr


00003:simon (OFFSET:0x000008e0, SIZE:0x00000428):
 - Public Definitions:
     simon

 - External Definitions:
     _ilog2
     hazlett                            <- 00002:hazlett
     francis                            <- 00000:francis
     _critlib_except
     _DosSelToFlat
     _DosFlatToSel
     _pBucketArr


00004:kehm (OFFSET:0x00000d10, SIZE:0x00000342):
 - Public Definitions:
     _kehm
 - External Definitions:
     DosFreeMem
     _critlib_except
     _DosSelToFlat
     _DosFlatToSel
     _pBucketArr
```

The following information describes the characteristics of the library. The **Flags** field determines case sensitivity. **0x1** indicates case sensitivity. **0x0** indicates no case sensitivity.

```
Flags = 0x0

Contains extended dictionary

Total number of modules = 5

Total bytes for modules = 4592

Total number of symbols in dictionary = 10

Maximum number of symbols in dictionary = 74

Total number of pages for the dictionary = 2
```

The following is the extended dictionary information. For each module, the listing provides the following information:

- The total number of modules in the library that contain definitions for external references in the current module. This number is listed in parenthese ( ).
- A list of the identifying module numbers for each of these modules.

In this case, modules 0, 2, and 4 have no dependencies, module 1 is dependent on module 0, and module 3 is dependent on modules 0 and 2.

```
======== Dependencies by Module ========
Module 00000 : (00000)
Module 00001 : (00001) 00000
Module 00002 : (00000)
Module 00003 : (00002) 00000 00002
Module 00004 : (00000)
```

ILIB Commands

ILIB commands are used to manipulate modules in a library. When you run ILIB, you can specify multiple commands in any order.

Each command consists of a one- or two-character command symbol immediately followed by the name of the module or file that is the subject of the command. For example,

```
+LEMKE.OBJ
```

adds the LEMKE.OBJ object file to a library as LEMKE.

Command	**Action**
[+]	Adds an object file or library to a library
-	Deletes a module from a library
-+	Replaces a module in a library
*****	Copies a module from a library to an object file
-*	Moves a module (copies the module and then deletes it)

Notes:

- If you want to enter more commands than can be conveniently entered on one line, type an ampersand (&) and press Enter at the end of the line. This extends the command field to the next line.
- When processing commands, ILIB processes all copy commands first. ILIB processes the deletions next, and the additions last.
- ILIB never makes changes to your input library while it runs; it copies the library and makes changes to the copy. However, if you do not specify an output library, ILIB overwrites the input library with the modified copy at the end of normal processing. See "Using the Command Line" on page 656 for more information.

Add Command (+)

Syntax:	**Default:**
`[+]`*`filename`*	*`+filename`*

Use the add command to add an object module or library to a library. The add command is issued by using the plus (+) sign or by leaving a blank space.

Adding an Object Module to a Library

Type the name of the object file to be added immediately after the plus sign. The .OBJ extension may be omitted.

ILIB uses the base name of the object file as the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is CURSOR.

Object modules are always added to the end of a library file.

Combining Two Libraries

Specify the name of the library file to be added, including the .LIB extension, immediately after the plus sign (+). A copy of the contents of that library is added to the library file being modified. If both libraries contain a module with the same name, ILIB generates a warning message (LIB0003), and uses only the first module with that name.

ILIB adds the modules of the library to the end of the library being changed. Note that the added library still exists as an independent library because ILIB copies the modules without deleting them.

Examples

```
ILIB MYLIB +EFREM;
```

The command above adds the file EFREM.OBJ to the library MYLIB.LIB.

```
ILIB NEWLIB +KAREN.LIB;
```

The command above adds the contents of the library KAREN.LIB to the library NEWLIB.LIB. The library KAREN.LIB is unchanged after this command is executed.

Delete Command (–)

Syntax:	**Default:**
[–]*filename*	When no command, assumes +

Use the delete command (–) to delete an object module from a library. After the minus sign, specify the name of the module to be deleted. Module names do not have path names or extensions.

Example

```
ILIB MYLIB -EFREM;
```

The command above deletes the module EFREM from the library MYLIB.LIB.

Replace Command (–+)

Syntax:	**Default:**
[–+]*filename*	When no command, assumes +

Use the replace command (–+) to replace a module in a library. Following the symbol, specify the name of the module to be replaced.

To replace a module, ILIB performs the following steps:

1. Deletes the existing module
2. Searches the current directory for the .OBJ file with the same file name as the deleted module
3. Appends to the library a copy of the object file with the original module name

Example

```
LIB MYLIB -+EFREM;
```

The command above replaces the module EFREM in the MYLIB.LIB library with the contents of EFREM.OBJ from the current directory. The file EFREM.OBJ in the current directory is not altered.

Copy Command (*)

Syntax:	**Default:**
[*]*filename*	When no command, assumes +

Use the copy command (*) to copy a module from the library into an object file of the same name. The module remains in the library.

When ILIB copies the module to an object file, it adds the .OBJ extension to the module name and places the file in the current directory. If a file with this name already exists, ILIB overwrites the existing .OBJ file.

Example

```
LIB MYLIB *EFREM;
```

The command above copies the module EFREM from the MYLIB.LIB library to a file called EFREM.OBJ in the current directory. The module EFREM in MYLIB.LIB is not altered.

Move Command (–*)

Syntax:	**Default:**
`[–*]`*filename*	When no command, assumes +

Use the move command (–*) to copy an object module from the library file to an object file. The object module is then deleted from the library file. This operation is equivalent to copying the module to an object file, then deleting the module from the library.

Example

```
LIB MYLIB -*KEELING;
```

The command above moves the module KEELING from the MYLIB.LIB library to a file called KEELING.OBJ in the current directory. Upon completion of this process, MYLIB.LIB no longer contains the module KEELING.

ILIB Options

Usage Notes:

- Option characters are not case sensitive; /H and /h are equivalent.
- The characters in brackets can be omitted; /H and /HELP are equivalent.
- Unless otherwise specified, most options and commands need only the first letter of their names to be used.

The following is a summary of ILIB options:

Option	**Action**
/C[ONVFORMAT]	Convert input library to the new ILIB format
/H[ELP] or /? or ?	Display Help
/I[GNORECASE]	Turn case sensitivity off
/NOBA[CKUP]	Do not create backup copy of library
/NOBR[OWSE]	Do not create browse information, remove any existing browse information
/NOE[XTDICTIONARY]	Do not generate extended dictionary
/NOI[GNORECASE]	Turn case sensitivity on
/NOL[OGO]	Suppress ILIB banner
/Q[UIET]	Suppress ILIB banner
/L[ISTLEVEL]	List current contents of library

/CONVFORMAT (Convert to New Format)

Syntax:	**Default:**
`/C[ONVFORMAT]`	Do not convert input libraries

Use `/CONVFORMAT` to convert an existing library to the new ILIB format used by the VisualAge C++ linker.

ILIB only produces libraries in the new format; the VisualAge C++ linker accepts libraries in both the new format and in older formats, but will link faster with libraries in the newer ILIB format.

To take advantage of the faster linking, you can convert a library to the new format using the `/CONVFORMAT` option.

Note: When you use `/CONVFORMAT`, you should also specify `/NOBROWSE` to exclude browse information because libraries in the old format do not include browse information.

Example

The following example converts the library `OLDLIB.LIB` from the previous LIB format to the new ILIB format:

```
ILIB OLDLIB.LIB /C
```

/HELP (Display Help)

Syntax:	**Default:**

`/H[ELP]`
`/?`
`?`
	None

Use `/HELP` to display a brief summary of ILIB syntax.

/IGNORECASE (Turn Case Sensitivity Off)

Syntax:	**Default:**
`/I[GNORECASE]`	`/I`

Use `/IGNORECASE` to turn off case sensitivity for symbols.

By default, case sensitivity is off. Use this option when you are combining a library that was created with case sensitivity on (using the /NOI option) with others that are not case sensitive. The resulting library is not case sensitive.

/LISTLEVEL (Set Detail Level of Listing)

Syntax:	**Default:**
`/L[ISTLEVEL][:`*`level`*`]`	`/L:1`

Use `/LISTLEVEL` to set the detail level of an ILIB listing. You can set the detail *`level`* as follows:

Level 1 Is the default. It is the fastest to generate and contains the least amount of information. All modules are listed in order of occurrence. For each module, the level 1 option:

1. Shows the relative position and size of each module.

2. Lists all the public symbols defined in the module, and their attributes.
3. Lists all external symbols which must be resolved at link time.

Level 2 Contains all the information of level 1. In addition, for each external symbol, level 2 shows which module in the library contains the required public symbols for resolving at link time. This can be overridden if a module is linked to another module that already contains the symbol.

Level 3 Contains all the information of level 2. In addition, Level 3 displays the technical characteristics of the library, and all browse information. This option also contains a dump of the extended dictionary. This is useful to determine which modules will be implicitly linked in whenever a particular module is linked in.

Note: If you are using the VisualAge C++ product, definitions with mangled names will be listed with the demangled form in brackets.

Sample Cross Reference Listing

```
LIB /LISTLEVEL:2 NEWLIB, NEWLIB.LST;
```

The command above directs ILIB to place a listing of the contents of NEWLIB.LIB into the file NEWLIB.LST. No path specification is given for NEWLIB.LST. By default, the file created is put in the current directory.

/NOBACKUP (Do Not Create Backup)

Syntax:	**Default:**
/NOBA[CKUP]	Create backup of library

Use /NOBACKUP to prevent ILIB from creating a backup of the library.

By default, ILIB creates a backup of the library before it is modified.

/NOBROWSE (Do Not Include Browse Information)

Syntax:	**Default:**
/NOBR[OWSE]	Include browse information in output library

Use /NOBROWSE to exclude browse information from the output library. The VisualAge C++ Browser can browse libraries and executable files that contain browse information. If you exclude browse information from the library, it cannot be browsed.

By default, ILIB adds or updates browse information in the output library, as follows:

- For all object files you use as input
- For library files you use as input, if they contain browse information

/NOEXTDICTIONARY (Do Not Generate Extended Dictionary)

Syntax:	**Default:**
`/NOE[XTDICTIONARY]`	Generate extended dictionary

Use `/NOEXTDICTIONARY` to disable generation of the extended dictionary.

The extended dictionary is an optional part of the library that increases linking speed. However, using an extended dictionary requires more memory. The space reserved for the extended dictionary is limited to 64K, no more can be allocated. If ILIB reports an *out-of-memory* error, you may want to use this option. As an alternative, you can split large libraries into smaller libraries to use in linking.

/NOIGNORECASE (Turn Case Sensitivity On)

Syntax:	**Default:**
`/NOI[GNORECASE]`	Ignore case of symbol names

Use `/NOIGNORECASE` to turn on case sensitivity.

By default, case sensitivity is off (/I option). Using this option allows symbols that differ only in case, such as Sine and SINE, to be included as separate symbols in the same library.

Note that when you create a library with the /NOI option, ILIB *marks* the library internally to indicate that /NOI is in effect. If you combine multiple libraries, and any one of them is marked /NOI, then the output library is marked /NOI.

/NOLOGO|/QUIET (Supress Banner)

Syntax:	**Default:**

`/NOL[OGO]`
`/QUIET`
	Suppress ILIB banner

Use `/NOLOGO` to suppress the ILIB copyright notice.

This option suppresses the banner message when ILIB is started. It can be used in batch files.

50 Packaging the VisualAge C++ Runtime DLLs

If your application uses functions from the VisualAge C++ libraries, you need to ensure the code for those libraries is always available to your application. You cannot ship the VisualAge C++ DLLs themselves with your application because of the product licensing agreement and because if more than one application included the VisualAge C++ DLLs, but at different levels, at least one application would be using the wrong level.

If you are shipping your application to other users who do not have access to the library DLLs, you can use one of three methods to include the VisualAge C++ library code:

1. Statically bind every module to the library (.LIB) files. Compile with `/Gd-`, which is the default.

   This method increases the size of your modules and slows the performance because the library environment has to be initialized for each module. Having multiple library environments also makes signal handling, file I/O, and other operations more complicated.

2. Create your own runtime DLLs.

   This method provides one common runtime environment for your entire application. It also lets you apply changes to the runtime library without relinking your application, meaning that if the VisualAge C++ DLLs change, you need to rebuild only your DLL.

   For a description of how to build your own runtime DLL, or subsystem runtime DLL, see the *Programming Guide*.

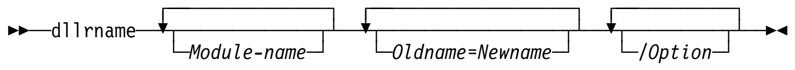
3. Use the DLL rename utility, DLLRNAME, to rename the VisualAge C++ library DLLs. This utility also changes the names in your executable files that call the DLLs. DLLRNAME is part of the VisualAge C++ product, and is described in “Using the DLLRNAME Utility” on page 676.

Using the DLLRNAME Utility

To use the DLL rename utility, build your application using the import libraries provided with VisualAge C++ (compiling with the `/Gd+` option). Then, before you ship your application:

1. Copy the VisualAge C++ DLLs that your application uses into a working directory.
2. Run the DLL rename utility, DLLRNAME, against your executable files and your working copies of the VisualAge C++ DLLs. The utility will rename the DLLs as well as all internal names that need to be changed as a result.

The syntax for the `dllrname` command is:



Module Names

The list of module names includes the VisualAge C++ library DLLs your application uses, along with the EXEs and DLLs that reference them. They must be present in the current directory, unless their paths are specified.

Note: It is important that you include all the modules in your application in this list, since the names by which the modules reference the VisualAge C++ library DLLs must also be changed in the modules themselves.

Name Changes

You specify the list of the name changes to be made by indicating *Oldname=Newname* on the DLLRNAME command line.

Oldname is the name of the VisualAge C++ library DLL as it was shipped with VisualAge C++.

Newname is the name under which you will be shipping the VisualAge C++ library DLL with your application.

Note: The DLLRNAME utility requires that *Oldname* and *Newname* have the same number of characters.

For example, to rename the VisualAge C++ library DLL CPPOS30.DLL to MYLIBRY.DLL in the modules `myprog.exe` and `mydll.dll`, specify the following on the DLLRNAME command line:

```
DLLRNAME myprog.exe mydll.dll CPPOS30=MYLIBRY
```

How DLLRNAME Works

All modules (.EXE and .DLL files) that use other DLLs contain records specifying a set of external file names that are needed to run the module. The DLLRNAME utility manipulates only these records; it does not modify your executable code.

One of the external names specified in a module is the name of the module itself. The name of module, as it appears in its own internal records, is called its *internal name*. You specify this name with the `NAME` or `LIBRARY` record in a DLL module definition (.DEF) file when you link the module. In the case of an .EXE file, the loader essentially ignores the internal name. For a .DLL, its internal name must match its filename otherwise the loader will refuse to load the DLL. By default, DLLRNAME will also change the filename of a DLL if it changed its internal name.

The rest of the external names specified in a module are the names of the DLLs to be loaded when the OS/2 loader loads the module. All of these DLLs must be loaded for the OS/2 loader to successfully load the module. If you specify any of these required DLLs for rename on the DLLRNAME command line, DLLRNAME also changes the names of the required DLLs in the module itself.

The DLLRNAME utility accepts 16- or 32-bit OS/2 protected mode executables. It will not change DOS or Windows executables, or any other file with a different format.

What DLLRNAME Will Not Do

The DLLRNAME utility will not:

- Modify DLLs named explicitly in your executable code. This is important to remember if you use the OS/2 `DosLoadModule()` API or the VisualAge C++ `_loadmod` function to load a required DLL. You must modify your code to load the DLL using its modified name.
- Modify DOS or Windows executables.
- Change the name of a DLL to a name of different length.

Other Uses for DLLRNAME

The DLLRNAME utility can also be used to:

- Rename your own DLLs so that you can have multiple versions of your application resident on the same machine for testing purposes.
- Obtain a report that lists all the DLLs used by a module. Simply invoke the DLLRNAME utility without specifying any DLL name changes.

DLLRNAME Options

The following options control the operation of the DLLRNAME utility:

/H or /?	Display help
/N	Do not rename DLL
/Q	Suppress display of logo
/R	Do not generate report

Note: You can specify options using the slash form (/R) or the dash form (-R).

/H (Help)

Syntax:	**Default:**
`/H` `/?`	None

Specify the `/H` or `/?` option on the DLLRNAME command line to see a short online help on the `dllrname` command syntax and options.

If you do not specify this option, the default is not to display any online help.

/N (Do Not Rename DLL)

Syntax:	**Default:**
`/N`	Rename all DLLs

Specify the `/N` option on the DLLRNAME command line to instruct the DLLRNAME utility not to rename any DLLs that appear in both the modules list and the list of name changes.

If you do not specify this option, the default is to rename any DLLs that appear in both the modules list and the list of name changes.

/Q (Do Not Display Logo)

Syntax:	**Default:**
`/Q`	Display logo and copyright notice

Specify the `/Q` option on the DLLRNAME command line to suppress the display of the logo and copyright notice for the DLLRNAME utility.

If you do not specify this option, the default is to display the logo and copyright notice.

/R (Do Not Generate Report)

Syntax:	**Default:**
`/R`	Generate report

Specify the `/R` option on the DLLRNAME command line to suppress the generation of a report detailing the name changes.

If you do not specify this option, the default is to generate a report detailing the name changes.

An Example

If you compiled your application using the following command lines:

```
ICC /Gd+ /Ge- /FeA.DLL A.C B.C C.C D.C A.DEF

IMPLIB A.LIB A.DLL

ILIB /CONV /NOE /NOBR A.LIB

ICC /Gd+ /FeE.EXE E.C F.C G.C H.C A.LIB
```

your application would be made up of the files A.DLL and E.EXE. Since you specified the /Gd+ compile option, your application also requires the file CPPOS30.DLL from VisualAge C++.

To obtain a renamed copy of CPPOS30.DLL which you may ship with your application, use the following set of commands:

```
REM Get a working copy of the VisualAge C++ library DLL
COPY D:\IBMCPP\DLL\CPPOS30.DLL

REM Change all the names
DLLRNAME A.DLL E.EXE CPPOS30=MYS30LB

DLLRNAME CPPOS30.DLL CPPOS30=MYS30LB
```

These commands will change A.DLL and E.EXE so that they will now require MYS30LB.DLL instead of CPPOS30.DLL. DLLRNAME will also rename CPPOS30.DLL to MYS30LB.DLL.

The following is the text of the report generated by the DLLRNAME utility:

```
> DLLRNAME A.DLL E.EXE CPPOS30.DLL CPPOS30=MYS30LB

Licensed Materials - Property of IBM
IBM VisualAge C++ Version 2.01 - DLL Rename Utility
(C) Copyright IBM Corp., 1993, All Rights Reserved
US Government Users Restricted Rights - Use, duplication or disclosure
restricted by GSA ADP Schedule Contract with IBM Corp.



Processing file A.DLL.
1 external names in file A.DLL have been left unchanged.
2 names found in file A.DLL.
Executable name A has been left unchanged.
Imported DLL name CPPOS30 has been changed to MYS30LB.

Processing file E.EXE.
2 external names in file E.EXE have been left unchanged.
3 names found in file E.EXE.
Executable name e has been left unchanged.
Imported DLL name A has been left unchanged.
Imported DLL name CPPOS30 has been changed to MYS30LB.

Processing file cppos30.dll.
5 external names in file CPPOS30.DLL have been left unchanged.
6 names found in file CPPOS30.DLL.
Executable name CPPOS30 has been changed to MYS30LB.
Imported DLL name DOSCALLS has been left unchanged.
Imported DLL name KBDCALLS has been left unchanged.
Imported DLL name VIOCALLS has been left unchanged.
Imported DLL name NLS has been left unchanged.
Imported DLL name MSG has been left unchanged.
File CPPOS30.DLL has been renamed to MYS30LB.DLL to match internal DLL name.


Complete.  0 error(s) detected.
```

51 Forwarded Entry Point (FWDSTAMP)

FWDSTAMP adds entry points, called *forwarders*, to a dynamic link library file (.DLL). Forwarders point to API functions or other exported code or data. They contain an import reference so that the final target address of the forwarded entry is contained in a different module. A forwarder might be called an *imported export*.

When a file has a fix-up to a forwarded entry point, the loader resolves that fix-up to the address of the entry point that the forwarder imports, by traversing the chain of forwarders until the end of the chain (a nonforwarded export) is reached. All forwarders are implicitly exported.

The imported entry point that a forwarder refers to may itself be another forwarder. The loader will process a chain of forwarders until a nonforwarder entry point is encountered.

There is no *run-time* cost to forwarders; however, there is a slight *load-time* cost as the loader resolves forwarder chains with their final addresses.

Using Forwarders

You use forwarders to combine several DLLs into one without having to relink old applications. For example, if MOUCALLS and VIOCALLS were combined into a single DLL called NEWLIB.DLL, then MOUCALLS and VIOCALLS could be replaced with special DLLs containing forwarders to NEWLIB.DLL.

Important Notes

- FWDSTAMP parses only the IMPORTS and EXPORTS section of the module definition file. FWDSTAMP does not verify the syntax of the other sections.
- When exported names already exist in the input file, their attributes are kept, such as resident or nonresident names table, and ordinal numbers. Any new conflicting attributes are ignored.
- If there is no exported name, FWDSTAMP adds the one defined by the EXPORTS statement in the module definition file.

FWDSTAMP

Starting FWDSTAMP

You can start FWDSTAMP and specify all input from the command line. The syntax is:

```
FWDSTAMP [options] infile deffile outfile
```

[*options*] Specifies one of the following:

- **/?** Displays FWDSTAMP help panel.
- **/V** Increases the level of information FWDSTAMP should output.

infile Specifies the name of the dynamic link library file that the linker created. Use the file-name extension .DLL.

deffile Specifies the name of the module definition file (.DEF) that contains the forwarders. (See “Example”).

outfile Specifies the name of the .DLL file that will contain the added forwarders.

Example

Forwarders are specified in the module definition file so that an exported name, which is also imported, is a forwarder. For example:

```
IMPORTS
    VIOMODEWAIT=NEWLIB.123
EXPORTS
    VIOMODEWAIT @ 25
```

In the example, a forwarder entry point for VIOMODEWAIT is created and contains an import reference to NEWLIB.123.

Part 10. Defining National Characteristics

When you create applications for international markets, you can define nation- or locale-specific characteristics of the application separately, and then bind different sets of characteristics to your application for the different locales it will be used in.

This part of the *User's Guide* describes three utilities you can use in this process:

ICONV, GENXLT	Convert a file from one code set to another
LOCALDEF	Define a locale

For more information on binding resources to your application, see Part 11, “Adding Application Resources” on page 693.

52 Code Set Conversion Utilities

This chapter describes the code set conversion utilities that help you convert a file from one code set to another.

ICONV Converts a file from one code set encoding to another. It can be used to convert C source code before compilation or to convert input files.

GENXLT Generates a translate table for use by the ICONV utility and iconv functions to perform code set conversion.

The ICONV utility calls the `iconv_open`, `iconv`, and `iconv_close` functions to perform the code set translation. set translation. These functions can be called from any program requiring code set translation. For more information on these functions, see the *C Library Reference*.

ICONV Utility

The ICONV utility converts the characters from the input file from one coded character set (code set) definition to another code set definition, and writes the characters to the output file (or **stdout** if no file is specified).

The ICONV utility uses the `iconv_open`, `iconv`, and `iconv_close` functions to convert the input file records from the coded character set definition for the input code page to the coded character set definition for the output code page. There is one record in the output file for each record in the input file. No padding or truncation of records is performed.

When conversions are performed between single-byte code pages, the output records are the same length as the input records. When conversions are performed between double-byte code pages, the output records may be longer or shorter than the input records because the shift-out and shift-in characters may be added or removed.

Code Set Conversion Utilities

The syntax of the `iconv` command is as follows:

```
►►──iconv──/F──fromcode──/T──tocode──┬──────────┬──►◄
                                     └─Filename─┘
```

where the options are:

/F Specify the input code set as *fromcode*

/T Specify the output code set as *tocode*

and you are converting code sets in *Filename*. If you do not specify *Filename*, ICONV reads from standard in (**stdin**). The converted file is sent to standard output (**stdout**).

If *Filename* contains codes that are not included in the *fromcode* code set, ICONV stops translating.

If *fromcode* specifies characters that do not exist in *tocode*, the characters are translated to a default character defined by the particular conversion taking place.

The values for *fromcode* and *tocode* identify either a converter within the converter DLL, or a conversion table created with GENXLT.

The function `iconv_open` is used to find a converter that matches the *fromcode* and *tocode* names.

ICONV opens the input and output files as binary. All characters from the input file (including any trailing cntrl-Z character) are converted from the input codepage to the output codepage and written to the output file.

Return Codes

The ICONV utility has the following return codes.

0 No errors were detected and the file was converted successfully.
1 An error occurred and the file was not converted.
2 An encoding error was encountered in the input file.

GENXLT Utility

Use the GENXLT utility to create a conversion table that you can use with ICONV to convert a file from one code set to another.

The syntax of the `genxlt` command is as follows:

```
►►──genxlt──┬───────────────────┬──┬─────────────┬──►◄
            └─/f──outputfile────┘  └─inputfile───┘
```

GENXLT reads a source translation file from *inputfile* and writes the compiled version to *outputfile*. If you do not specify an input file, GENXLT uses standard input (**stdin**). If you do not specify an output file, GENXLT uses standard output (**stdout**).

Format of the Translation Source File

The translation source file can contain comment lines and directives. Start comment lines with the number sign (#). Directive lines have the following format:

source target comment

where:

source Specify the source bytes to be translated to *target*. Use hexadecimal values.

target Specify the value want *source* translated to. Use either a hexadecimal value, or the keyword `invalid` to indicate that there is no valid target for the specified source.

comment Any additional text on the line is considered a comment, and ignored.

If you provide multiple assignments for the same *source*, only the last assignment is used.

Separate the *source*, *target*, and *comment* parameters with space or tab characters.

GENXLT Return Codes

The GENXLT utility has the following return codes:

0 No errors were detected and the file was converted successfully.

1 An error occurred and the translation table was not successfully built.

Code Set Conversion Utilities

53

LOCALDEF Utility

A *locale* is the definition of the subset of the user's environment that depends on language and cultural conventions. The locale object contains the rules and pointers to methods to implement the language and cultural conventions. A locale object is made up of a number of categories, identified by name, that control specific aspects of the behavior of components of the system.

The locale object is generated by the LOCALDEF utility according to the rules defined in the locale definition file. The locale object is implemented as a dynamic link library (DLL), but with the extension .LCL instead of .DLL. The locale can be loaded using the `setlocale()` function. Each .LCL file contains only one locale.

The LOCALDEF utility reads the locale source and produces a locale object that can be used by the locale-specific library functions.

Using LOCALDEF

Run the LOCALDEF utility with `localdef` command. The syntax of the `localdef` command is as follows:

```
             ┌────────────────────────────────────┐
►►──localdef──▼─┬────────────────────────────────────┬┴───────────────────►
              ├─/c─────────────────────────────────┤
              ├─/f──charmap filename───────────────┤
              ├─/i──locale definition filename─────┤
              └─/W──┬─1─┬──────────────────────────┘
                    └─2─┘

►──localename──►◄
```

where *localename* specifies the name of the output file for the locale generated. If you do not specify an extension for *localename*, LOCALDEF assumes the extension .LCL. If you do not qualify *localename* with a path, the locale is written to the current directory.

You can use the following options:

`/C` Generate locale even if there are errors
`/F` Specify file that maps symbols to character encodings
`/I` Specify locale source file
`/W` Control messages produced

LOCALDEF Options

/C (Continue If Errors)

Syntax:	**Default:**
`/C`	Stop process when error encountered.

Use `/C` to generate a locale object even if LOCALDEF encounters an error during the locale definition process.

By default, LOCALDEF does not generate a locale object if there are errors.

/F (Character Map)

Syntax:	**Default:**
`/F` *filename*	`/F IBM-850.CM`

Use `/F` to specify the name of the file that maps character symbols and collating element symbols to actual character encodings. If you do not specify an extension for the file name, LOCALDEF assumes `.CM`.

By default, LOCALDEF uses `IBM-850.CM`.

If you specify the file without including its path, LOCALDEF searches for it in the following places:

1. The current directory
2. The directories listed in the DPATH environment variable

/I (Locale Source File)

Syntax:	**Default:**
`/I`	Standard input (**`stdin`**)

Use `/I` to specify the source file for the locale. If you do not specify an extension, LOCALDEF assumes the extension `.LOC`.

If you specify the file without including its path, LOCALDEF searches for it in the following places:

1. The current directory

2. The directories listed in the DPATH environment variable

/W (Control Warnings)

Syntax:	**Default:**
`/W[1|2]`	`/W2`

Use `/W` to control the type of message LOCALDEF produces.

`/W1` Produce severe errors and errors
`/W2` Produce severe errors, errors, and warnings

By default, LOCALDEF produces all three kinds of messages.

LOCALDEF Return Codes

The LOCALDEF utility has the following return codes.

0 No errors were detected and the locale was generated successfully.
1 Warning messages were issued and the locale was generated successfully.
2 The locale specification exceeded implementation limits or the coded character set or sets used were not supported by the implementation. The locale was **not** generated.
>3 Warnings or errors were detected and the locale was **not** generated.

Locale Build Process

To build the locale object, LOCALDEF performs the following steps:

1. Reads the locale source (from **`stdin`**, or from the file specified with the `/I` option).

2. Writes a temporary file that contains C source code.

   Note: Temporary files are written to the directory specified by the TMP environment variable. If the TMP variable does not exist, the files are written to the current directory.

3. Invokes the VisualAge C++ compiler to compile the C source code, with the following compiler options:

`/C+`	Compile without linking.
`/Gd-`	Statically link the run-time library.
`/Ge-`	Compile a dynamic link library (DLL).
`/Rn`	Generate subsystem code.
`/Q+`	Do not display logo.
`/NdLOCALE`	Rename the data and const segments.
`/Fo`*`tmpobjname`*	Specify the temporary object file name.

4. Builds a module definition file for ILINK to use, as follows:

```
LIBRARY INITINSTANCE TERMINSTANCE
EXPORTS
   instantiate
SEGMENTS
   LOCALEDATA32   CLASS 'DATA'  READONLY SHARED
   LOCALECONST32  CLASS 'CONST' READONLY SHARED
```

5. Invokes the VisualAge C++ linker linker with the following parameters:

`/NOE` Specify the `/NOEXTDICTIONARY` option, because LOCALDEF uses the `_DLL_InitTerm()` function built into the locale source and not listed in the extended dictionary.

`/NOFR` Specify the `/NOFREE` option, to use LINK386-compatible command-line syntax.

`/NOI` Specify the `/NOIGNORECASE` option, to respect capitalization in identifiers.

`/NOL` Specify the `/NOLOGO` option, to suppress display of logo.

object Specify the name of the temporary object file created by the compiler.

target Specify the name of the output DLL (as defined when you invoked LOCALDEF)

library Specify `CPPOMTHI.LIB` as an additional library to be searched

def_file Specify the name of the temporary module definition file created by LOCALDEF in the previous step.

6. Deletes the temporary files.

Part 11. Adding Application Resources

To make your application customisable, you can maintain some resources (such as dialogs and message strings) in separate files, that you then bind to your application with the Resource Compiler.

This part of the *User's Guide* describes the use of the following utilities:

Resource Compiler	Allows you to define or modify application resources without recompiling the application.
Dialog Editor	Allows you create and modify dialog boxes.
Font Editor	Allows you to create and modify fonts.
Icon Editor	Allows you to create and modify icons.

Resource Compiler

The OS/2 Resource Compiler (RC) is an application-development tool that lets you add application resources, such as message strings, pointers, menus, and dialog boxes, to the executable file of your application. The Resource Compiler is primarily intended to prepare data for OS/2 applications that use functions such as WinLoadString, WinLoadPointer, WinLoadMenu, and WinLoadDlg. These functions load resources from the executable file of your application or another specified executable file. The application can then use the loaded resources as needed.

The Resource Compiler and the resource functions let you quickly define and/or modify application resources without recompiling the application itself. That is, RC can modify the resources in an executable file at any time without affecting the rest of the file. This means that you can create custom applications from a single executable file — you just use RC to add the custom resources you need to each application.

The Resource Compiler is especially important for international applications because it lets you define all language-dependent data, such as message strings, as resources. Preparing the application for a new language is simply a matter of adding new resources to the existing executable file.

Note: Make sure the file RCPP.EXE (the Resource Compiler preprocessor) is available for the use of the Resource Compiler. It can be in the current directory, or in a directory to which there is a path.

Command-Line Options

The following options can be specified on the Resource Compiler command line:

-d<defname>[=<value>]	Define macro to preprocessor
-i <pathspec>	Include file path
-r	Create .res file
-p	Pack - 386 resources will not cross 64K boundaries.
-cp (or -k) {<codepage>|<lbs,lbe>...}	-DBCS code page or lead byte information
-x[{1|2}]	Exepack - compress resources, using method 1 or 2.
-cc <countrycode>	Country code
-h	Access Help

Leave a blank after the letter when using option -cc, -D, -I , -cp or -k. Upper or lowercase can be used.

Explanation of Command-Line Options

The -D option is useful for passing conditional-compilation flags to the preprocessor. The <defname> is a sequence of letters, underscore symbols, and digits which does not begin with a digit. The <value> is a sequence of symbols which you want to substitute for the <defname> wherever it appears in the input script file. If you omit the =<value>, the <defname> will be set to the default value 1. For example, the option -D_3d is equivalent to including at the beginning of the input file this line:

```
#define    _3d        1
```

You can use the -D option up to eight times to define different macros from the command line.

The -I option defines paths for files to be included with the source file. The <pathspec> is any path where you want RC to search for files included by the preprocessor #include directive. The <pathspec> must not contain embedded blanks. To include more than one path, code the -I option once for each path. The preprocessor reads paths from the INCLUDE environment after reading the paths you provide with -I options.

The -R switch will create in your current directory a binary resource file containing the resources you compile. The -R switch takes no argument. The name given to this binary resource file will be the same as the name of the input resource script file except that the extension will be .RES instead of .RC. When you use -R, you do not bind resources to an executable file.

The -P switch is used only when binding resources to an executable. It positions resources so that they do not cross 64K boundaries.

The -CP or -K option is used to specify code page information for the resource script file to be compiled. The <codepage> is a numeric code page value from the "Code Page Table" on page 701. Instead of specifying a code page, you may provide a sequence of pairs of DBCS lead byte code points. Each pair of numbers gives the lower and upper limit of a range of code points which are to be interpreted as DBCS lead bytes. The code page must be valid for the country code in effect: either the default country code or the country specified using the -CC option.

The -CC option allows you to specify a country code for the resource script file to be compiled. The <countrycode> is a number from the table. For more information see "Code Page Table" on page 701.

The -X option is used only when binding resources to an executable. It causes resources to be compressed. These resources will be decompressed automatically when the resource is accessed.

The -X1 option causes Resource Compiler to use the compression algorithm that is compatible with OS/2 2.0, 2.1, and 2.11.

The -X2 option causes Resource Compiler to use a compression algorithm that is incompatible with OS/2 2.0, 2.1 and 2.11. The -X2 option will produce smaller executable files that can access resources faster.

-X with no number defaults to -X1.

When you use the -H switch, Resource Compiler displays on your screen a summary of the available options and environment variables that it uses. When you specify -H, the resource compiler does not read any input files. This is equivalent to entering on the command line "RC" with no operands.

Help

To display Resource Compiler help, type `RC` at the prompt, with no parameters. The appropriate copyright statement will be displayed, along with a list of Resource Compiler options. You can also display this list by using the command-line option -h.

```
Usage:  rc [<options>] <.RC input file>[<.EXE output file>]

  -d defname        - Preprocessor define
  -Ddefname         - Preprocessor define
  -i                - Include file path
  -r                - Create .res file
  -x[1|2]           - Exepack-compress resources using method 1or 2
  -cc cc            - country code
  -p                - Pack - 386 resources will not cross 64K boundaries
  -cp cp | lb,tb,...- DBCS codepage or lead/trail byte information
  -h                - Access Help

Environment variables:
  DBCS=cp | lb,tb,...
  TMP=temporary file path
  TEMP=temporary file path
  INCLUDE=include file path
```

Note: Option -X2 will produce executable files that are incompatible with OS/2 2.0, 2.1, and 2.11.

Resource Script Files

This topic describes the resource script file used to define your application resources and explains how to compile the file and add the resources to your executable file.

Use the Resource Compiler to perform the following actions:

- Create a resource script file.
- Compile the file.
- Add the file to the executable file of your application (optional).

The following sections describe the resource script file and the RC program.

Resource Script Files

A resource script file consists of one or more resource statements that define the type, identifier, and data for each resource. For example, the following multiple-line resource statement defines a menu to be used with an application:

```
MENU 1
BEGIN
    MENUITEM "Alpha", 101
    MENUITEM "Beta", 102
END
```

A resource script file is a text file you can create by using an ordinary text editor. Since some resources may contain binary data that cannot be created using a text editor, many resource statements let you specify additional files to include when compiling the resource script file. For example, the following statement defines an icon and specifies the file MYICON.ICO as containing the icon data:

```
ICON 1 myicon.ico
```

Directives

A resource script file can also contain directives. For example, the following directive includes the header file OS2.H when RC processes the resource script file:

```
#include <os2.h>
```

Resource script files typically have the .RC filename extension. .RC is the default extension; use it for all your resource script files.

Note: Although the Resource Compiler is C-like in syntax, it is not a C compiler. Use only the Resource Compiler statements.

Directives

A directive is a Resource Compiler statement that carries out a task such as including a header file, defining constants, or conditionally compiling portions of the resource script file.

Directives

elif Directive
else Directive
endif Directive
if Directive
ifdef Directive
ifndef Directive

Using the Resource Compiler

The Resource Compiler (RC) compiles a resource script file to create a new file called a binary resource file.

The binary resource file can be added to the executable file of the application, replacing any existing resources in that file.

You can start RC in any of three ways.

- Compile and add a resource definition file to an executable file
- Compile a resource script file
- Add a binary resource file to an executable file

The RC command line has the following three basic forms:

```
rc resource-script-file [executable-file]

rc resource-file [executable-file]

rc -r  resource-script-file [resource-file]
```

Note: The third option does not add to the executable file.

The **resource-script-file** field must be the filename of the resource script file to be compiled. If the file is not in the current directory, you must provide a full path. If you provide a filename without specifying a filename extension, RC automatically appends the .RC extension to the name.

The **executable-file** field must be the name of the executable file to receive the compiled resources. This is a file having a filename extension of either .EXE or .DLL. If the file is not in the current directory, you must provide a full path. If you omit the executable-file field, RC adds the compiled resources to the executable file that has the same name as the resource script file but which has the .EXE filename extension. If you specify the executable-file field but omit the filename extension, RC will append the .EXE extension. If this executable file does not exist, RC displays an error message.

The **-r** option directs RC to compile the resource script file without adding it to an executable file. You can use this option to prepare a binary resource file that you can add to an executable file at a later time. If you do not explicitly name a binary resource file along with the -r option, RC uses the same name as the resource script file but with the .RES filename extension.

The **resource-file** field must be the name of the binary resource file to be added to the executable file. If the binary resource file does not already exist, RC creates it; otherwise, RC replaces the existing file. If the file is not in the current directory, you must provide a full path. The binary resource file must have the .RES filename extension.

For example, to compile the resource script file EXAMPLE.RC, and add the result to the executable file EXAMPLE.EXE, use the following command:

```
rc example
```

You do not need to specify the .RC extension. RC creates the binary resource file EXAMPLE.RES and adds the compiled resource to the executable file EXAMPLE.EXE.

To compile the resource script file EXAMPLE.RC into a binary resource file without adding the resources to an executable file, use the following command:

```
rc -r example
```

The compiler creates the binary resource file EXAMPLE.RES. To create a binary resource file that has a name different from the resource script file, use the following command:

```
rc -r example newfile.res
```

To add the compiled resources in the binary resource file EXAMPLE.RES to an executable file, use the following command:

```
rc example.res
```

To specify the name of the executable file, if the name is different from the resource file, use the following command:

```
rc example.res newfile.exe
```

To add the compiled resources to a dynamic-link-library (.DLL) file, use the following command:

```
rc example.res dynalink.dll
```

In addition to -r, RC offers two other command-line options: **-cp** and **-cc**. The -cp option lets you specify a code-page identifier or DBCS lead byte information. The -cc option lets you specify a country code. The syntax is as follows:

```
-cp {codepage-id | lead-byte-start, lead-byte-end,...}
-cc country-code
```

The lead-byte-start and lead-byte-end fields give the upper and lower limits of each interval of DBCS lead bytes which you are defining for the code page. You may specify these values instead of a codepage-id.

The codepage-id or country-code field contains one of the valid code page or country codes, for example:

Code Page Table

CODE PAGE	COUNTRY CODE	MEANING
932	81	Japan
934	82	Korea
936	86	China
938	88	Taiwan

Defining Constants

The –d option lets you define up to eight symbolic constants on the command line. The syntax is as follows:

```
-d defname[=value]
```

In the previous example, defname is a name, and value is an integer constant, or an expression. The -d option is useful for passing conditional-compilation flags to the RC preprocessor.

The following example specifies a Japanese code-page identifier and also defines two symbolic constants to be passed to the preprocessor as conditional-compilation flags.

```
rc -cp 932 -d DEBUG -d VERSION=2 example
```

Note: To process directives in the resource script file, RC uses the files RCPP.EXE and RCPP.ERR. Be sure that these files are in the current directory or in a directory specified by your PATH environment variable. RC creates many temporary files and writes them to the directory indicated by the TMP or TEMP environment variable. If RC cannot write these temporary files to this directory, it writes them to the current directory.

About Resource Statements

Each resource statement consists of one or more keywords, numbers, character strings, constants, or filenames. You combine these to define the resource type, identifier, and data.

Keywords are words that have a special meaning to the Resource Compiler. In a statement, keywords specify the resource type, the load and memory options, and the beginning and end of nested statements. You can use the RC keywords only as specified in the statement syntax.

Keywords, except for those specifying directives, can be any combination of uppercase and lowercase letters. Note that the curly braces, { and }, are reserved characters. You can use them in place of the BEGIN and END keywords.

Numbers are integers that represent coordinates, dimensions, styles, and other numeric data. You can specify numbers in decimal, octal, or hexadecimal notation:

Decimal numbers must contain decimal digits but can start with a minus sign (-) when they represent a negative number.
Hexadecimal numbers must contain hexadecimal digits (uppercase or lowercase) and must start with the characters `0x`.
Octal numbers are similar to hexadecimal numbers, except that a lowercase letter `o` replaces the `x`.

The following example shows several numbers represented in decimal, octal, and hexadecimal notation:

DECIMAL	OCTAL	HEXADECIMAL
1	0o1	0x1
10	0o12	0xA
255	0o377	0xFF
-1	0o177777	0xFFFF
65535	0o177777	0xFFFF

Statements that create controls in dialog windows and menu items in menus require that you specify an identifier for each control or menu item. Statements that create controls also require you to specify coordinates and dimensions. You specify identifiers, coordinates, and dimensions using integers in the range -32768 through 32767. Optionally, you can use simple expressions that evaluate to integers from 0 through 65535; this lets you specify identifiers, dimensions, and coordinates that are relative to those of the corresponding dialog window or menu.

Character strings represent names, labels, titles, and messages. A character string consists of one or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark (") is required in a string, you must include the double quotation mark twice. The meaning of each character value (that is, the character each value represents) depends on the code page (character set) defined for the resource script file.

The Resource Compiler interprets the backslash (\) as an escape character in character strings. You can include any ASCII character in a character string by specifying either \xdd, where dd is the hexadecimal representation of an ASCII character, or \nnn, where nnn is the octal representation of an ASCII character. If a backslash is required in a string, you must include the backslash twice.

Constants are names that have been assigned values by using the define directive. A constant can represent a number, a character string, or other data. Most resource statements in a resource script file use constants, and many use the constants defined in the OS/2 header files (for example, os2.h). For this reason, you should always use the include directive to include OS2.H in your resource script file.

Filenames are OS/2 filenames. If the specified file is not in the current directory, you must specify the drive, directory, and filename.

Resource statements have three basic forms:

Single-line statements
Multiple-line statements
Directives

Single-line statements consist of a keyword identifying the resource type, a constant or number specifying the resource identifier, and a filename specifying the file containing the resource data. For example, this ICON statement defines an icon resource:

```
ICON 1 myicon.ico
```

The icon resource has the icon identifier 1. The file MYICON.ICO contains the icon data.

Multiple-line statements consist of a keyword identifying the resource type, a constant or number specifying the resource identifier, and, between the BEGIN and END keywords, additional resource statements that define the resource data. For example, this MENU statement defines a menu resource:

```
MENU 1
BEGIN
    MENUITEM "Alpha", 101
    MENUITEM "Beta",  102
END
```

The menu identifier is 1. The menu contains two MENUITEM statements that define the contents of the menu.

In multiple-line statements such as DLGTEMPLATE and WINDOWTEMPLATE, RC allows any level of nested statements. For example, the DLGTEMPLATE and WINDOWTEMPLATE statements typically contain a single DIALOG or FRAME statement. These statements can contain any number of WINDOW and CONTROL statements; the WINDOW and CONTROL statements can contain additional WINDOW and CONTROL statements; and so on. The nested statements let you define controls and other child windows for the dialog boxes and windows. If a nested statement creates a child window or control, the parent and owner of the new window is the window created by the containing statement. (FRAME statements occasionally create frame controls whose parent and owner windows are not the same.)

Directives consist of the reserved character # in the first column of a line, followed by the directive keyword and any additional numbers, character strings, or filenames.

Binary Resource Files

The binary resource file created by the Resource Compiler consists of one or more resource entries, each in the following form:

```
struct {
    UCHAR  fResType;
    USHORT usResType;
    UCHAR  fResID;
    USHORT resid;
    USHORT fsOptions;
    ULONG  cb;
    BYTE   bytes[1];
};
```

The fields in each entry have the following meanings:

fRestype Specifies whether the resource-type identifier is a string or an integer. For OS/2, the resource type is always an integer and this field is set to 0xFF.

usResType Specifies the resource-type identifier. This value is an integer in the range -32768 through 32767. The following resource types are predefined:

RT_ACCELTABLE	Accelerator table
RT_BITMAP	Bitmap
RT_CHARTBL	Character table
RT_DIALOG	Dialog template
RT_DISPLAYINFO	Display information
RT_DLGINCLUDE	Dialog include-file name
RT_FKALONG	Long-form function-key area
RT_FKASHORT	Short-form function-key area
RT_FONT	Font
RT_FONTDIR	Font directory
RT_HELPSUBTABLE	Help subtable
RT_HELPTABLE	Help table
RT_KEYTBL	Key table
RT_MENU	Menu template
RT_MESSAGE	Error-message table

RT_POINTER Mouse-pointer shape

RT_RCDATA Binary data

RT_STRING String table

RT_VKEYTBL Virtual key table

fResID Specifies whether the resource identifier is a string or an integer. For OS/2, the resource identifier is always an integer and this field is set to 0xFF.

resid Specifies the resource identifier. This value is an integer in the range -32768 through 32767.

fsOptions Specifies the load and memory options. This value can be a combination of the following:

0x0010 MOVEABLE resource. If not given, the resource is FIXED.
0x0040 PRELOAD resource. If not given, the resource is LOADONCALL.
0x1000 DISCARDABLE resource.

cb Specifies the size of the resource (in bytes).

bytes Contains the resource.

Note: There is a size limitation of 65,280 bytes for a binary resource file.

Statements and Directives

The following statements and directives are used by the Resource Compiler (RC):

ACCELTABLE Statement
ASSOCTABLE Statement
AUTOCHECKBOX Statement
AUTORADIOBUTTON Statement
BITMAP Statement
CHECKBOX Statement
CODEPAGE Statement
COMBOBOX Statement
CONTAINER Statement
CONTROL Statement
CTEXT Statement
CTLDATA Statement
DEFAULTICON Statement
define Directive
DEFPUSHBUTTON Statement
DIALOG Statement
DLGINCLUDE Statement
DLGTEMPLATE Statement
EDITTEXT Statement
elif Directive
else Directive
endif Directive
ENTRYFIELD Statement
FONT Statement
FRAME Statement
GROUPBOX Statement
HELPITEM Statement
HELPSUBITEM Statement
HELPSUBTABLE Statement
HELPTABLE Statement
ICON Statement (Resource)
ICON Statement (Control)
if Directive
ifdef Directive
ifndef Directive
include Directive
LISTBOX Statement
LTEXT Statement
MENU Statement
MENUITEM Statement

ACCELTABLE Statement

MESSAGETABLE Statement
MLE Statement
NOTEBOOK Statement
POINTER Statement
PRESPARAMS Statement
PUSHBUTTON Statement
RADIOBUTTON Statement
RCDATA Statement
RCINCLUDE Statement
RESOURCE Statement
RTEXT Statement
SLIDER Statement
SPINBUTTON Statement
STRINGTABLE Statement
SUBITEMSIZE Statement
SUBMENU Statement
undef Directive
VALUESET Statement
WINDOW Statement
WINDOWTEMPLATE Statement

ACCELTABLE Statement

Syntax:

```
ACCELTABLE acceltable-id [mem-option][code-page]
BEGIN
key-value, command[, accelerator-options]...
    .
    .
    .
END
```

Description

The ACCELTABLE statement creates a table of accelerators for an application. An accelerator is a keystroke that gives the user a quick way to choose a command from a menu or carry out some other task. An accelerator table can be loaded when needed from the executable file by using the WinLoadAccelTable function.

You can provide any number of ACCELTABLE statements in a resource script file. Each statement must specify a unique table identifier. You can provide any number of accelerator definitions in an accelerator table; however, no two definitions in a table can specify the same key.

Each accelerator definition must specify a key value and command. The WinSetAccelTable function used in the application translates the accelerator keystroke into a WM_COMMAND, WM_HELP, or WM_SYSCOMMAND message that has the corresponding command value. The message type depends on the accelerator-option field.

acceltable-id Specifies the accelerator-table identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range. Each accelerator table in a resource script file must have a unique identifier.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one of the following:

OPTION	Meaning
FIXED	System keeps the resource at a fixed memory location.
MOVEABLE	System moves the resource as necessary to compact memory. This is the default option.
DISCARDABLE	System discards the resource if it is no longer needed.

code-page Specifies a code page value. For a list of valid code pages see “CODEPAGE Statement” on page 718.

key-value Specifies the character, scan, or virtual-key code of the accelerator key. The meaning depends on the accelerator-options field. The key-value field must be a single character enclosed in double-quotation marks or an integer in the range 0 through 255. If you specify an integer, you must specify the CHAR, SCANCODE, or VIRTUALKEY accelerator option; otherwise, the default option is CHAR. Integers must be in decimal or hexadecimal notation.

command Specifies the command value for the corresponding WM_COMMAND, WM_HELP, or WM_SYSCOMMAND message. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to an integer in that range.

accelerator-options Specifies the accelerator type. This value can be a combination of the following:

- **VIRTUALKEY** Specifies that the key-value field is a virtual-key code.
- **SCANCODE** Specifies that the key-value field is a keyboard scan code.
- **CHAR** Specifies that the key-value field is a character code.
- **SHIFT** Specifies that the user must press the Shift key and the key corresponding to the key-value field to generate the accelerator.
- **CONTROL** Specifies that the user must press the Ctrl key and the key corresponding to the key-value field to generate the accelerator.
- **ALT** Specifies that the user must press the Alt key and the key corresponding to the key-value field to generate the accelerator.
- **LONEKEY** Specifies that the user needs to press only the key corresponding to the key-value field to generate the accelerator.
- **SYSCOMMAND** Specifies that the accelerator translates to a WM_SYSCOMMAND message. If you do not include this option, the accelerator translates to a WM_COMMAND message.
- **HELP** Specifies that the accelerator translates to a WM_HELP message. If you do not include this option, the accelerator translates to a WM_COMMAND message.

Note: VIRTUALKEY, SCANCODE, and CHAR are mutually exclusive. SYSCOMMAND and HELP are also mutually exclusive.

Comments

If two accelerators use the same key with different Shift, Control, or ALT options, you should specify the more restrictive accelerator first in the table. For example, you should place Shift+Enter before Enter.

If you include the `<os2.h>` header file, you can use the following constants to specify the accelerator options:

AF_ALT
AF_CHAR
AF_CONTROL
AF_HELP
AF_LONEKEY
AF_SCANCODE
AF_SHIFT
AF_SYSCOMMAND
AF_VIRTUALKEY

To combine these constants, you must use the bitwise OR (|) operator.

Example

This example creates an accelerator table whose accelerator-table identifier is 1. The table contains two accelerators: Ctrl+S and Ctrl+G. These accelerators generate WM_COMMAND messages with values of 101 and 102, respectively, when the user presses the corresponding keys.

```
ACCELTABLE 1
BEGIN
    "S", 101, CONTROL
    "G", 102, CONTROL
END
```

ASSOCTABLE Statement

Syntax:

```
ASSOCTABLE assoctable-id [load-option][mem-option][code-page]
BEGIN
association-name, file-match-string[, extended-attribute-flag]
  [, icon-filename]
  .
  .
  .
END
```

Description

The ASSOCTABLE statement defines a file-association table for an application. This table associates the data files that an application creates with the executable file of the application. When the user selects one of these data files from File Manager, the associated application begins executing.

A file-association table can also associate icons with the data files that an application creates. The shell uses these icons to identify the data files graphically. Because a file-association table associates icons by file type, all data files having the same file type have the same icon.

ASSOCTABLE Statement

You can provide any number of ASSOCTABLE statements in a resource script file, but each statement must specify a unique assoctable-id value. The file-association tables are written not only to the resources within your executable file, but also to the .ASSOC extended attribute. However, only the last file-association table specified in the resource script file is actually written to the extended attribute.

assoctable-id Specifies the association-table identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the DosGetResource or DosGetResource2 function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory. This is the default option.
- **DISCARDABLE** System discards the resource if it is no longer needed.

code-page Specifies a code page value. For a list of valid code pages see “CODEPAGE Statement” on page 718.

association-name Specifies the name of the file type the application recognizes. This field must contain zero or more characters enclosed in double quotation marks.

Character values must be in the range 1 through 255. If a double quotation mark is required in the name, you must include the double quotation mark twice.

file-match-string Specifies the file-matching string of a particular type of data file that the application creates. This field must contain zero or more characters enclosed in double quotation marks. You can only use characters that are valid in OS/2 filenames and extensions and the OS/2 wildcard characters question mark (?) and asterisk (*).

extended-attribute-flag
Specifies the extended-attribute options. This value can be a combination of the following:

EAF_DEFAULTOWNER
Specifies that the application containing the file-association table starts when the user selects any file matching the file-match-string field from File Manager.

EAF_REUSEICON
Specifies that the icon defined in the previous entry of the file-association table is used as the icon for the current data-file type.

EAF_UNCHANGEABLE
Specifies that the entry should not be edited.

icon-filename
Specifies the name of the file containing an icon. File Manager uses this icon to represent all application-created data files matching the file-match-string field. The file must be in the current directory.

AUTOCHECKBOX Statement

Syntax:

```
AUTOCHECKBOX text, id, x, y, width, height[, style]
```

The AUTOCHECKBOX statement creates an automatic-check-box control. The control is a small rectangle (check box) that contains an X when the user selects it. The specified text is displayed to the right of the check box. An X appears in the square when the user first selects the control and disappears the next time the user selects it. The AUTOCHECKBOX statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify the style, the default style is BS_AUTOCHECKBOX and WS_TABSTOP.

text
Specifies text that is displayed to the right of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice. A tilde (˜) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_BUTTON. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates an automatic-check-box control that is labeled "Italic."

```
  AUTOCHECKBOX "Italic", 101, 10, 10, 100, 100
```

AUTORADIOBUTTON Statement

Syntax:

```
AUTORADIOBUTTON text, id, x, y, width, height[, style]
```

The AUTORADIOBUTTON statement creates an automatic-radio-button control. This control is a small circle with the given text displayed to its right. The control highlights the circle and sends a message to its parent window when the user selects the button. The control also removes the selection from any other automatic-radio-button controls in the same group. When the user selects the button again, the control removes the highlight before sending a message. The AUTORADIOBUTTON statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_AUTORADIOBUTTON.

text Specifies text that is displayed to the right of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice. A tilde (˜) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range 32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_BUTTON. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates an automatic-radio-button control that is labeled "Italic."

```
  AUTORADIOBUTTON "Italic", 101, 10, 10, 24, 50
```

BITMAP Statement

Syntax:

```
BITMAP bitmap-id [load-option] [mem-option] [codepage] filename
```

The BITMAP statement defines a bit map resource for an application. A bit map resource, typically created using the Icon Editor, is a custom bit map that an application uses in its display or as an item in a menu. The BITMAP statement copies the bit-map resource from the file specified in the filename field and adds it to the application's other resources. A bit-map resource can be loaded from the executable file when needed by using the GpiLoadBitmap function.

You can provide any number of BITMAP statements in a resource script file, but each statement must specify a unique bitmap-id value.

bitmap-id Specifies the bit-map-resource identifier. This value must be an integer in the range -32768 through 32767 or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the GpiLoadBitmap function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory. This is the default option.
- **DISCARDABLE** System discards the resource if it is no longer needed.

codepage Specifies a code page value. For a list of valid code pages see “CODEPAGE Statement” on page 718.

filename Specifies the name of the file containing the icon resource. If the file is not in the current directory, filename must be preceded by a full path.

Example

This example defines a bit map whose bit-map identifier is 12. The bit-map resource is copied from the file CUSTOM.BMP.

```
  BITMAP 12 custom.bmp
```

CHECKBOX Statement

Syntax:

```
CHECKBOX text, id, x, y, width, height[, style]
```

The CHECKBOX statement creates a check-box control. The control is a small rectangle (check box) that has the specified text displayed to the right. The control highlights the rectangle and sends a message to its parent window when the user selects the control. The CHECKBOX statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_CHECKBOX and WS_TABSTOP.

text Specifies text that is displayed to the right of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice. A tilde (˜) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_BUTTON. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a check-box control that is labeled "Italic."

```
  CHECKBOX "Italic", 101, 10, 10, 100, 100
```

CODEPAGE Statement

Syntax:

```
CODEPAGE codepage-id
```

The CODEPAGE statement sets the code page for all subsequent resources. The code page specifies the character set used for characters in the resource.

If the CODEPAGE statement is not given in a resource script file, RC uses the code page set up for the individual system. If more than one CODEPAGE statement is given in the file, each CODEPAGE statement applies to the resource statements between it and the next CODEPAGE statement.

codepage-id Identifies the code page to be used for subsequent resources. This value can be one of the following:

437	United States
850	Multilingual
860	Portuguese
863	Canadian French
865	Norwegian
932	Japanese
934	Korean
936	Chinese
938	Taiwan

Comments

You may also specify a code page by placing a code-page identifier in the load-options or memory-options field of any RC statement that uses those fields.

Example

In this example, the code page for the character-string resources is set to Portuguese (860).

```
CODEPAGE 860

STRINGTABLE
BEGIN
    1 "Filename not found"
    2 "Cannot open file for reading"
END
```

COMBOBOX Statement

Syntax:

```
COMBOBOX text, id, x, y, width, height[, style]
```

The COMBOBOX statement creates a combination-box control. This control combines a list-box control with an entry-field control. It allows the user to place the selected item from a list box into an entry field.

The COMBOBOX statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_COMBOBOX. If you do not specify a style, the default style is CBS_SIMPLE, WS_GROUP, WS_TABSTOP, and WS_VISIBLE.

text Specifies text that is displayed in the entry field of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_COMBOBOX. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a combination-box control.

```
  COMBOBOX "", 101, 10, 10, 24, 50
```

CONTAINER Statement

Syntax:

```
CONTAINER  id, x, y, width, height [,style]
```

The CONTAINER statement creates a container control within a dialog window. The container control is a visual component that holds objects. The CONTAINER statement defines the identifier, position, dimensions, and attributes of a container control. The predefined class for this control is WC_CONTAINER. If you do not specify a style, the default style is WS_TABSTOP, WS_VISIBLE, and CCS_SINGLESEL.

id Specifies the control identifier. This value is any integer -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window containing the container control.

y Specifies the y-coordinate of the lower-left corner of the control. This value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window containing the container control.

width Specifies the width of the control. This value is any integer 0 through 65535, or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value is any integer 0 through 65535, or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_CONTAINER. Use the bitwise OR (|) operator to combine styles.

Comments

A CONTAINER statement is only used in a DIALOG or WINDOW statement.

Example

This example creates a container control at position (30,30) within the dialog window. The container has a width of 70 character units and a height of 25 character units. Its resource ID is 301. The default style CCS_SINGLESEL has been overridden by the style specification CCS_MULTIPLESEL. The default styles WS_TABSTOP and WS_GROUP are both in effect, though only the latter is specified.

```
#define IDC_CONTAINER    301
#define IDD_CONTAINERDLG 504
DIALOG "Container", IDD_CONTAINERDLG, 23, 6, 120, 280, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
     CONTAINER   IDC_CONTAINER, 30, 30, 70, 200, CCS_MULTIPLESEL |
                       WS_GROUP
  END
```

CONTROL Statement

Syntax:

```
  CONTROL text, id, x, y, width, height, class[, style]
  [ data-definitions ]
  [ BEGIN
  control-definition
      .
      .
      .
  END ]
```

The CONTROL statement defines a control as belonging to the specified class. The statement defines the position and dimensions of the control within the parent window, as well as the control style. The CONTROL statement is most often used in a DIALOG or WINDOW statement.

Typically, several CONTROL statements are used in each DIALOG statement, and each CONTROL statement must have a unique id value. The optional BEGIN and END statements enclose any CONTROL statements that may be given with the control. CONTROL statements given in this manner represent child windows belonging to the control created by the CONTROL statement.

text Specifies text that is displayed to the right of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice. In the appropriate styles, a tilde (˜) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the parent window.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the parent window.

width Specifies the width of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in 1/8-character units.

class Specifies the control class. This value can be one of the control classes specified in the “Control Classes” table, in the Presentation Manager Programming Reference, or the name of the control class, enclosed in double quotation marks.

style Specifies the control style. This value can be a combination of control styles. You can use the bitwise OR (|) operator to combine styles.

data-definitions

Specifies a CTLDATA and/or PRESPARAMS statement. These statements define control and presentation data for the control. For more information, see “CTLDATA Statement” on page 724 and “PRESPARAMS Statement” on page 762.

control-definition

Specifies a CONTROL statement or any one of several predefined control statements. These statements define the style, position, and dimensions of controls in the control.

Comments

The CONTROL statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements. But typically, a CONTROL statement contains no such statements.

Example

This example creates a pushbutton control with the WS_TABSTOP and WS_VISIBLE styles.

```
CONTROL "OK", 101, 10, 10, 20, 50, WC_BUTTON, BS_PUSHBUTTON |
                                              WS_TABSTOP    |
                                              WS_VISIBLE
```

CTEXT Statement

Syntax:

```
CTEXT text, id, x, y, width, height[, style]
```

The CTEXT statement creates a centered-text control. The control is a simple rectangle displaying the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The CTEXT statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of the control. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_TEXT, DT_CENTER, and WS_GROUP.

text Specifies text that is centered in the rectangular area of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_STATIC. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a centered-text control that is labeled "Filename."

```
CTEXT "Filename", 101, 10, 10, 100, 100
```

CTLDATA Statement

Syntax:

```
CTLDATA word-value[, word-value][...]

CTLDATA string

CTLDATA MENU
BEGIN
menuitem-definition
   .
   .
   .
END
```

The CTLDATA statement defines control data for a custom dialog box, window, or control. The statement has three basic forms to permit specifying a menu or specifying data in words or characters. The data can be in any format, since only your window procedure will use it. The window procedure of the dialog box, window, or control receives this data when the item is created. It is up to the window procedure to process the data.

word-value Specifies a 16-bit value. This value must be an integer in the range -32768 through 32767. It must be in decimal notation.

string
Specifies a string of 8-bit characters. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the string, you must include the double quotation mark twice.

menuitem-definition
Specifies a MENUITEM or SUBMENU statement. These statements define the individual commands or submenus in the given menu. For details about these statements, see "MENUITEM Statement" on page 753 and "SUBMENU Statement" on page 774.

Comments

CTLDATA is often used to supply data that controls the subsequent operation of the custom window. For example, the CTLDATA statement may contain extended style bits — that is, style bits designed specifically for your customized window.

You should reserve the CTLDATA statement for window classes that you create yourself.

Example

This example creates a menu for the window created with the WINDOW statement.

```
WINDOWTEMPLATE 1
BEGIN
    WINDOW "Sample", 1, 0, 0, 100, 100, "MYCLASS", 0, FCF_STANDARD
    CTLDATA MENU
    BEGIN
        MENUITEM "Exit", 101
    END
END
```

DEFAULTICON Statement

This statement installs the named icon file definition under the ICON Extended Attribute of the program file. An icon with an icon-id of 1 is the default icon by default, unless you supply a different icon.

Example DEFAULTICON filename.ico

define Directive

Syntax:

```
  define name value
```

The define directive assigns the given value to the specified name. All subsequent occurrences of the name are replaced by the value.

name Specifies the name to be defined. This value is any combination of letters, digits, and punctuation.

value Specifies any integer, character string, or line of text. This value can contain another defined name, which creates a level of nested defines. You are limited to 64 levels of nested defines.

Example

This example assigns values to the names "NONZERO" and "USERCLASS".

```
  #define     NONZERO     1
  #define     USERCLASS   "MyControlClass"
```

DEFPUSHBUTTON Statement

Syntax:

```
DEFPUSHBUTTON text, id, x, y, width, height[, style]
```

The DEFPUSHBUTTON statement creates a default pushbutton control. The control is a round-cornered rectangle containing the given text. The rectangle has a bold outline to represent that it is the default response for the user. The control sends a message to its parent window when the user chooses the control. The DEFPUSHBUTTON statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of the control. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_PUSHBUTTON, BS_DEFAULT, and WS_TABSTOP.

text Specifies text that is centered in the rectangular area of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice. A tilde (˜) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_BUTTON. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a default pushbutton control that is labeled "Cancel."

```
DEFPUSHBUTTON "Cancel", 101, 10, 10, 24, 50
```

DIALOG Statement

Syntax:

```
DIALOG text, id, x, y, width, height[, style[,
framectl]][data-definitions]
BEGIN
control-definition
    .
    .
    .
END
```

The DIALOG statement defines a window that an application can use to create dialog boxes. The statement defines the position and dimensions of the dialog box on the screen, as well as the dialog-box style. The DIALOG statement is most often used in a DLGTEMPLATE statement.

Typically, you use only one DIALOG statement in each DLGTEMPLATE statement, and the DIALOG statement contains at least one control definition.

text Specifies the dialog-box title if the style specifies a title bar. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the title, you must include the double quotation mark twice.

id Specifies the dialog-box identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the dialog box. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in dialog units, but its exact meaning depends on the dialog style. See the "Comments" section for details.

y Specifies the y-coordinate of the lower-left corner of the dialog box. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in dialog units, but its exact meaning depends on the dialog style. See the "Comments" section for details.

width Specifies the width of the dialog box. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in n-character units.

height Specifies the height of the dialog box. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in 1/8-character units.

style Specifies the dialog-box style. This value can be any of the window, dialog-box, or frame styles. You can use the bitwise OR (|) operator to combine styles.

framectl Specifies the styles for frame controls belonging to the dialog box. This value can be any of the frame-control styles specified in the “Frame-Control Flags” table in the Presentation Manager Programming Reference. You can use the bitwise OR (|) operator to combine styles.

data-definitions
Specifies a CTLDATA and/or PRESPARAMS statement. These statements define control and presentation data for the dialog box. For more information, see “CTLDATA Statement” on page 724 and “PRESPARAMS Statement” on page 762.

control-definition
Specifies a CONTROL statement or any one of several predefined control statements. These statements define the style, position, and dimensions of controls in the dialog box.

Comments

The exact meaning of the coordinates depends on the style defined by the style field. For dialog boxes with FS_SCREENALIGN style, the coordinates are relative to the origin of the display screen. For dialog boxes with the style FS_MOUSEALIGN, the coordinates are relative to the position of the mouse pointer at the time the dialog box is created. For all other dialog boxes, the coordinates are relative to the origin of the parent window.

The DIALOG statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements. Typically, a DIALOG statement contains one or more CONTROL statements.

Example

This example creates a dialog box that is labeled "Disk Error."

```
DLGTEMPLATE 1
BEGIN
    DIALOG  "Disk Error", 100, 10, 10, 300, 110
    BEGIN
        CTEXT "Select One:", 1, 10, 80, 280, 12
        RADIOBUTTON "Retry", 2, 75, 50, 60, 12
        RADIOBUTTON "Abort", 3, 75, 30, 60, 12
        RADIOBUTTON "Ignore", 4, 75, 10, 60, 12
    END
END
```

DLGINCLUDE Statement

Syntax:

```
DLGINCLUDE id filename
```

The DLGINCLUDE statement adds the specified filename to the resource file. The DLGINCLUDE statement is typically used to let the application access the definitions file for the dialog box with the corresponding identifier. The file named by filename must contain the define directives used by the dialog box.

You can provide any number of DLGINCLUDE statements in a resource script file, but each must have a unique identifier.

id Specifies the dialog-box identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

filename Specifies the name of the file containing the define directives. If the file is not in the current directory, filename must be preceded by a full path.

Example

This example includes the name of the definition file dlgdef.h. The dialog-box identifier is 5.

```
DLGINCLUDE 5 \\INCLUDE\\DLGREF.H
```

DLGTEMPLATE Statement

Syntax:

```
DLGTEMPLATE dialog-id  [load-option]  [mem-option][codepage]
BEGIN
dialog-definition
    .
    .
    .
END
```

The DLGTEMPLATE statement creates a dialog-box template. A dialog-box template consists of a series of statements that define the identifier, load and memory options, dialog-box dimensions, and controls in the dialog box. The dialog-box template can be loaded from the executable file by using the WinLoadDlg function.

You can provide any number of dialog-box templates in a resource script file, but each template must have a unique dialog-id value.

dialog-id Specifies the dialog-box identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the WinLoadDlg function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory.
- **DISCARDABLE** System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

codepage Specifies a code-page value. For a list of valid code pages see “CODEPAGE Statement” on page 718.

dialog-definition Specifies a DIALOG statement. The statement defines the dimensions and style of the given dialog box. For details about the statement, see “DIALOG Statement” on page 728.

Comments

A DLGTEMPLATE statement can actually contain DIALOG, CONTROL, and WINDOW statements. Typically, you include only one DIALOG statement.

Example

This example uses a DLGTEMPLATE statement to create a dialog box.

```
DLGTEMPLATE ID_GETTIMER
BEGIN
    DIALOG "Timer", 1, 10, 10, 100, 40
    BEGIN
        LTEXT "Time (0 - 15):", 4, 8, 24, 72, 12
        ENTRYFIELD "0", ID_TIME, 80, 28, 16, 8, ES_MARGIN
        DEFPUSHBUTTON "Enter", ID_TIMEOK, 10, 6, 36, 12
        PUSHBUTTON "Cancel", ID_TIMECANCEL, 52, 6, 40, 12
    END
END
```

EDITTEXT Statement

Syntax:

```
EDITTEXT text, id, x, y, width, height [,style]
```

The EDITTEXT statement creates an entry-field control. This control is a rectangle in which the user can type and edit text. The control displays a pointer when the user selects the control. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. By using the mouse or the DIRECTION keys, the user can select the character or characters to delete or select the place to insert new characters.

The EDITTEXT statement defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_ENTRYFIELD. If you do not specify a style, the default style is ES_AUTOSCROLL and WS_TABSTOP.

text Specifies text that is displayed in the rectangular area of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value is any integer -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value is any integer -32768 through 32767, or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

y Specifies the y-coordinate of the lower-left corner of the control. This value is any integer -32768 through 32767, or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

width Specifies the width of the control. This value is any integer 0 through 65 535, or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value is any integer 0 through 65 535, or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_ENTRYFIELD. You can use the bitwise OR (|) operator to combine styles.

Comments

The EDITTEXT control statement is identical to the ENTRYFIELD control statement.

Use the EDITTEXT statement only in a DIALOG or WINDOW statement.

Example

This example creates an entry-field control that is not labeled.

```
   EDITTEXT "", 101, 10, 10, 24, 50
```

elif Directive

Syntax:

```
elif constant-expression
```

The elif directive marks an optional clause of a conditional-compilation block defined by a ifdef, ifndef, or if directive. The directive controls conditional compilation of the resource file by checking the specified constant expression. If the constant expression is nonzero, elif directs the compiler to continue processing statements up to the next endif, else, or elif directive and then skip to the statement after endif. If the constant expression is zero, elif directs the compiler to skip to the next endif, else, or elif directive. You can use any number of elif directives in a conditional block.

constant-expression Specifies the expression to be checked. This value is a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

In this example, elif directs the compiler to process the second BITMAP statement only if the value assigned to the name "Version" is less than 7. The elif directive itself is processed only if Version is greater than or equal to 3.

```
#if Version < 3
BITMAP 1 errbox.bmp
#elif Version < 7
BITMAP 1 userbox.bmp
#endif
```

else Directive

Syntax: `else`

The else directive marks an optional clause of a conditional-compilation block defined by a ifdef, ifndef, or if directive. The else directive must be the last directive before the endif directive.

This directive has no arguments.

Example

This example compiles the second BITMAP statement only if the name "DEBUG" is not defined.

```
    #ifdef DEBUG
        BITMAP 1 errbox.bmp
    #else
        BITMAP 1 userbox.bmp
    #endif
```

endif Directive

Syntax: `endif`

The endif directive marks the end of a conditional-compilation block defined by a ifdef directive. One endif is required for each if, ifdef, or ifndef directive.

This directive has no arguments.

ENTRYFIELD Statement

```
ENTRYFIELD text, id, x, y, width, height , [style]
```

The ENTRYFIELD statement creates an entry-field control. This control is a rectangle in which the user can type and edit text. The control displays a pointer when the user selects the control. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. By using the mouse or the DIRECTION keys, the user can select the character or characters to delete or select the place to insert new characters. The ENTRYFIELD statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_ENTRYFIELD. If you do not specify a style, the default style is ES_AUTOSCROLL and WS_TABSTOP.

text Specifies text that is displayed in the rectangular area of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_ENTRYFIELD. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates an entry-field control that is not labeled.

```
ENTRYFIELD "", 101, 10, 10, 24, 50
```

FONT Statement

Syntax:

```
FONT font-id  [load-option][ mem-option] [codepage]  filename
```

The FONT statement defines a font resource for an application. A font resource, typically created by using the OS/2 Font Editor, is a bit map defining the shape of the individual characters in a character set. The FONT statement copies the font resource from the file specified in the filename field and adds it to the other resources of the application. A font resource can be loaded from the executable file when needed by using the GpiLoadFonts function.

You can provide any number of FONT statements in a resource script file, but each statement must specify a unique font-id value.

font-id Specifies the font-resource identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the GpiLoadFonts function. This is the default option.

mem-option Specifies how the system manages the resource when it is i memory. This value must be one or more of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory.

DISCARDABLE System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

codepage Specifies a code page value. For a list of valid code pages see “CODEPAGE Statement” on page 718.

filename Specifies the name of the file containing the font resource. If the file is not in the current directory, filename must be preceded by a full path.

Example

This example defines a font whose font identifier is 5. The font resource is copied from the file cmroman.fon.

```
FONT 5 cmroman.fon
```

FRAME Statement

Syntax:

```
FRAME text, id, x, y, width, height[, style[, framectl]]
  data-definitions
[ BEGIN
window-definition
    .
    .
    .
END ]
```

The FRAME statement defines a frame window. The statement defines the title, identifier, position, and dimensions of the frame window, as well as the window style. The FRAME statement is most often used in a WINDOWTEMPLATE statement, and typically, only one FRAME statement is used. The FRAME statement, in turn, typically contains at least one WINDOW statement that defines the client window belonging to the frame window.

The frame window has no default style. You must use the **framectl** field to define additional frame controls, such as a title bar and system menu, to be created when the frame window is created. If the text field is not empty, the statement automatically adds a title-bar control to the frame window, whether or not you specify the FCF_TITLEBAR style. Frame controls are given default styles and control identifiers depending on their class. For example, a title-bar control receives the identifier FID_TITLEBAR.

text Specifies the title of the frame window. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the name, you must include the double quotation mark twice.

id Specifies the window identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the window. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified window.

y Specifies the y-coordinate of the lower-left corner of the window. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified window.

width Specifies the width of the window. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the window. This value must be a integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the frame and window styles. This value can be a combination of frame styles. You can use the bitwise OR (|) operator to combine styles.

framectl Specifies the styles of frame controls belonging to the frame window. This value can be a combination of the styles specified in the "Frame-Control Styles" table in the Presentation Manager Programmers Reference. You can use the bitwise OR (|) operator to combine styles.

data-definitions

Specifies a CTLDATA and/or PRESPARAMS statement. These statements define control and presentation data for the frame window. For more information, see "CTLDATA Statement" on page 724 and "PRESPARAMS Statement" on page 762.

window-definition

Specifies a WINDOW statement or any one of several predefined control statements. These statements define the style, position, and dimensions of windows or controls in the frame window.

Comments

The FRAME statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements. Typically, a FRAME statement contains one WINDOW statement.

Example

This example creates a standard frame window, with a title bar, a system menu, minimize and maximize boxes, and a vertical scroll bar. The FRAME statement contains a WINDOW statement defining the client window belonging to the frame window.

```
WINDOWTEMPLATE 1
BEGIN
    FRAME "My Window", 1, 10, 10, 320, 130, 0,
            FCF_STANDARD | FCF_VERTSCROLL
    BEGIN
        WINDOW "", FID_CLIENT, 0, 0, 0, 0, "MyClientClass"
    END
END
```

GROUPBOX Statement

Syntax:

```
GROUPBOX text, id, x, y, width, height [, style]
```

The GROUPBOX statement creates a group-box control. The control is a rectangle that groups other controls together. The controls are grouped by drawing a border around them and displaying the given text in the upper-left corner. The GROUPBOX statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_GROUPBOX and WS_TABSTOP.

text Specifies text that appears in the upper-left corner of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_STATIC. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a group-box control that is labeled "Options."

```
GROUPBOX "Options", 101, 10, 10, 100, 100
```

HELPITEM Statement

Syntax:

```
HELPITEM application-window-id, help-subtable-id, extended-helppanel-id
```

The HELPITEM statement defines the help items in a help table. The statement, permitted only in a HELPTABLE statement, specifies the resource identifier of an application window for which help is provided, and the resource identifiers of the help subtable and extended help panel associated with the application window.

You can provide any number of HELPITEM statements in a HELPTABLE statement. You should provide one HELPITEM statement for each application window for which help is provided.

application-window-id Specifies the resource identifier of an application window for which help is provided.

help-subtable-id Specifies the resource identifier of the help subtable associated with the specified application window.

extended-helppanel-id Specifies the resource identifier of the extended help panel associated with the specified application window.

Example

This example defines a help item that associates a help subtable called IDSUB_FILEMENU and an extended help panel called IDEXT_APPHLP with an application window called IDWIN_FILEMENU.

```
HELPITEM IDWIN_FILEMENU, IDSUB_FILEMENU, IDEXT_APPHLP
```

HELPSUBITEM Statement

Syntax:

```
HELPSUBITEM child-window-id, helppanel-id [ , integer...]
```

The HELPSUBITEM statement defines the help subitems in a help subtable. This statement, which is permitted only in a HELPSUBTABLE statement, specifies the identifier of a child window for which help is provided, the identifier of the help panel associated with the child window, and one or more optional, application-defined integers.

You can provide any number of HELPSUBITEM statements in a HELPSUBTABLE statement. You should provide one HELPSUBITEM statement for each child window for which help is provided.

child-window-id Specifies the resource identifier of the child window for which help is provided.

helppanel-id Specifies the resource identifier of the help panel associated with the specified child window.

integer Specifies optional, application-defined integers. If you use this field, you must include the SUBITEMSIZE statement in the help subtable to specify the size, in words, of each help subitem in the help subtable. For details about this statement, see “SUBITEMSIZE Statement” on page 773.

Example

This example defines a help subitem that associates a child window called IDCLD_FILEMENU with a help panel called IDHP_FILEMENU.

```
HELPSUBITEM IDCLD_FILEMENU, IDHP_FILEMENU
```

HELPSUBTABLE Statement

Syntax:

```
HELPSUBTABLE helpsubtable-id
 SUBITEMSIZE size
BEGIN
helpsubitem-definition
    .
    .
    .
END
```

The HELPSUBTABLE statement defines the contents of a help-subtable resource. A help-subtable resource contains a help-subitem entry for each item that can be selected in an application window. Each of these items should be a child window of the application window specified in the help-table resource. The help subtable should contain a help subitem for each control, child window, and menu item in the application window.

You can provide any number of HELPSUBTABLE statements in a resource script file, but each statement must specify a unique helpsubtable-id value. You can also provide any number of helpsubitem-definition statements in the help subtable. These specify the child window for which help is provided, the help panel containing the help text for the child window, and one or more application-defined integers.

If you include optional integers in the helpsubitem-definition statements, you must also include a SUBITEMSIZE statement to specify the size, in words, of each help subitem. All help subitems in a help subtable must be the same size. The default size is two words per help subitem.

helpsubtable-id
Specifies the resource identifier of the help subtable. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

helpsubitem-definition
Specifies a HELPSUBITEM statement. A HELPSUBITEM statement specifies a child window, the help panel associated with the child window, and one or more optional, application-defined integers. For details about this statement, see “HELPSUBITEM Statement” on page 741.

Example

This example creates a help-subtable resource whose help-subtable identifier is IDSUB_FILEMENU. Each HELPSUBITEM statement specifies a child window and a help panel.

```
HELPSUBTABLE IDSUB_FILEMENU
BEGIN
    HELPSUBITEM IDCLD_OPEN, IDPNL_OPEN
    HELPSUBITEM IDCLD_SAVE, IDPNL_SAVE
END
```

HELPTABLE Statement

Syntax:

```
HELPTABLE helptable-id
BEGIN
helpitem-definition
    .
    .
    .
END
```

The HELPTABLE statement defines the contents of a help-table resource. A help-table resource contains a help-item entry for each application window, dialog box, and message box for which help is provided.

You can provide any number of HELPTABLE statements in a resource script file, but each statement must specify a unique helptable-id value. You can also provide any number of helpitem-definition statements in the help table. These specify the application windows for which help is provided, the help subtables associated with each application window, and the extended help panels associated with each application window.

helptable-id Specifies the resource identifier of the help table. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

helpitem-definition Specifies a HELPITEM statement. A HELPITEM statement specifies an application window and the associated help subtable and extended help panel. For details about this statement, see “HELPITEM Statement” on page 740.

Example

This example creates a help-table resource whose help-table identifier is 1. Each HELPITEM statement specifies an application window, a help subtable, and an extended help panel.

```
HELPTABLE 1
BEGIN
    HELPITEM IDWIN_FILEMENU, IDSUB_FILEMENU, IDEXT_APPHLP
    HELPITEM IDWIN_EDITMENU, IDSUB_EDITMENU, IDEXT_APPHLP
END
```

ICON Statement (Resource)

Syntax:

```
ICON icon-id  [load-option]  [ mem-option] [codepage] filename
```

This form of the ICON statement defines an icon resource for an application. An icon resource, typically created by using Icon Editor, is a bit map defining the shape of the icon to be used for a given application. The ICON statement copies the icon resource from the file specified in the filename field and adds it to the application's other resources. An icon resource can be loaded when creating a window by using the WinCreateStdWindow function with the FS_ICON style.

You can provide any number of ICON statements in a resource script file, but each statement must specify a unique icon-id value.

icon-id Specifies the icon-resource identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range. A icon-id of 1 has a special meaning; for details, see the "Comment" section.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the WinCreateStdWindow function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory.

DISCARDABLE System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

codepage Specifies a code page value. For a list of valid code pages see "CODEPAGE Statement" on page 718.

filename Specifies the name of the file containing the icon resource. If the file is not in the current directory, filename must be preceded by a full path.

Comments

An icon with an icon-id of 1 is the default icon. The RC program writes the icon not only to the resources in your executable file, but also as the .ICON extended attribute. File Manager will display this icon next to the name of the executable file.

Example

This example defines an icon whose icon identifier is 11. The icon resource is copied from the file custom.ico.

```
ICON 11 custom.ico
```

ICON Statement (Control)

Syntax:

```
ICON icon-id, id, x, y, width, height , [style]
```

This form of the ICON statement creates an icon control. This control is an icon displayed in a dialog box. The ICON statement, which you can use only in a DIALOG or WINDOW statement, defines the icon-resource identifier, icon-control identifier, position, and attributes of a control window. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_ICON. For the ICON statement, the width and height fields are ignored; the icon automatically sizes itself.

icon-id Specifies the resource identifier of an icon that is defined elsewhere in the resource file.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies a reserved value. Can be set to zero.

height Specifies a reserved value. Can be set to zero.

style Specifies the control styles. This value can be a combination of the styles specified for WC_STATIC. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates an icon control whose icon identifier is 99.

```
ICON 99, 101, 10, 10, 0, 0
```

if Directive

Syntax:

```
if constant-expression
```

The if directive controls conditional compilation of the resource file by checking the specified constant expression. If the constant expression is nonzero, if directs the compiler to continue processing statements up to the next endif, else, or elif directive and then skip to the statement after the endif directive. If the constant expression is zero, if directs the compiler to skip to the next endif, else, or elif directive.

constant-expression Specifies the expression to be checked. This value is a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

This example compiles the BITMAP statement only if the value assigned to the name "Version" is less than 3.

```
#if Version < 3
BITMAP 1 errbox.bmp
#endif
```

ifdef Directive

Syntax: `ifdef name`

The ifdef directive controls conditional compilation of the resource file by checking the specified name. If the name has been defined by using a define directive or by using the -d command-line option of rc, ifdef directs the compiler to continue with the statement immediately after the ifdef directive. If the name has not been defined, ifdef directs the compiler to skip all statements up to the next endif directive.

name Specifies the name to be checked by the directive.

Example

This example compiles the BITMAP statement only if the name "Debug" is defined.

```
#ifdef Debug
BITMAP 1 errbox.bmp
#endif
```

ifndef Directive

Syntax: `ifndef name`

The ifndef directive controls conditional compilation of the resource file by checking the specified name. If the name has not been defined or if its definition has been removed by using the undef directive, ifndef directs the compiler to continue processing statements up to the next endif, else, or elif directive and then skip to the statement after the endif directive. If the name is defined, ifndef directs the compiler to skip to the next endif, else, or elif directive.

name Specifies the name to be checked by the directive.

Example

This example compiles the BITMAP statement only if the name "Optimize" is not defined.

```
#ifndef Optimize
BITMAP 1 errbox.bmp
#endif
```

include Directive

Syntax:

```
include filename
```

The include directive causes RC to process the file specified in the filename field. This file should be a header file that defines the constants used in the resource script file. Only the define directives in the specified file are processed; all other statements are ignored.

filename Specifies the OS/2 name of the file to be included. This value must be an ASCII string enclosed either in double quotation marks (if the file is in the current directory) or in less-than and greater-than characters (<>) (if the file is in the directory specified by -i command-line options or by the INCLUDE environment variable). You must give a full path enclosed in double quotation marks if the file is not in the current directory or in the directory specified by -i command-line options or by the INCLUDE environment variable.

Comments

The filename field is handled as a C string. Therefore, you must include two backslashes wherever one is required in the path. (As an alternative, you can use a single forward slash (/) instead of two backslashes.)

Example

This example processes the header files `<os2.h>` and `<HEADERS\MYDEFS.H>` while compiling the resource script file.

```
#include <os2.h>
#include "headers\\\\mydefs.h"
```

LISTBOX Statement

Syntax:

```
LISTBOX id, x, y, width, height[, style]
```

The LISTBOX statement creates commonly used controls for a dialog box or window. The control is a rectangle containing a list of user-selectable strings, such as filenames.

The LISTBOX statement, which you can use only in a DIALOG or WINDOW statement, defines the identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_LISTBOX. If you do not specify a style, the default style is WS_TABSTOP.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_LISTBOX. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a list-box control whose identifier is 101.

```
LISTBOX 101, 10, 10, 100, 100
```

LTEXT Statement

Syntax:

```
LTEXT text, id, x, y, width, height[ , style]
```

The LTEXT statement creates a left-aligned text control. The control is a simple rectangle displaying the given text left-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The LTEXT statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of the control. The predefined class for this control is WC_STATIC. If you do not specify a style, the default style is SS_TEXT, DT_LEFT, and WS_GROUP.

text Specifies text that is left-aligned in the rectangular area of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_STATIC. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a left-aligned text control that is labeled "Filename."

```
LTEXT "Filename", 101, 10, 10, 100, 100
```

MENU Statement

Syntax:

```
MENU menu-id  [load-option][ mem-option][codepage]
BEGIN
menuitem-definition
    .
    .
    .
END
```

The MENU statement defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user choose commands from a list of command names. A menu resource can be loaded from the executable file when needed by using the WinLoadMenu function.

You can provide any number of MENU statements in a resource script file, but each statement must specify a unique menu-id value. You can provide any number of menuitem-definition statements in the menu. These define the submenus and menu items (commands) in the menu. The order of the statements defines the order of the menu items.

menu-id Specifies the menu-resource identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the WinLoadMenu function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

- **FIXED** System keeps the resource at a fixed memory location.

MOVEABLE System moves the resource as necessary to compact memory.

DISCARDABLE System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

codepage Specifies a codepage value. For a list of valid code pages see “CODEPAGE Statement” on page 718.

menuitem-definition Specifies a PRESPARAMS, MENUITEM, or SUBMENU statement. You can use one or more PRESPARAMS statements to control the appearance of a menu, such as the font and the foreground and background colors. If used, PRESPARAMS statements must be the first statements following the BEGIN keyword. For details about the PRESPARAMS statement, see “PRESPARAMS Statement” on page 762.

MENUITEM and SUBMENU statements define the individual commands or submenus in the given menu. For details about these statements, see “MENUITEM Statement” on page 753 and “SUBMENU Statement” on page 774.

Example

This example creates a menu resource whose menu identifier is 1. The menu contains a menu item named Alpha and a submenu named Beta. The submenu contains two menu items, Item 1 and Item 2.

```
MENU 1
BEGIN
    MENUITEM "Alpha", 100
    SUBMENU "Beta", 101
    BEGIN
        MENUITEM "Item 1", 200
        MENUITEM "Item 2", 201, , MIA_CHECKED
    END
END
```

MENUITEM Statement

Syntax:

```
MENUITEM text, menu-id[,  menuitem-style][,menuitem-attribute]
```

MENUITEM SEPARATOR

The MENUITEM statement creates a menu item for a menu. The statement, permitted only in a MENU or SUBMENU statement, defines the text, identifier, and attributes of a menu item. The system displays the text when it displays the corresponding menu. If the user chooses the menu item, the system generates a WM_COMMAND message that includes the specified menu-item identifier and sends it to the window owning the menu.

You can provide any number of MENUITEM statements, but each must have a unique menu-id value.

The alternative form of the MENUITEM statement, MENUITEM SEPARATOR, creates a menu separator. A menu separator is a horizontal dividing bar between two menu items in a submenu. The separator is not active — that is, the user cannot choose it, it has no text associated with it, and it has no identifier.

text
Specifies the text of the menu item. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the string, you must include the double quotation mark twice. The tilde character (˜) and the \t and \a character combinations have special meanings in the string; for details, see the "Comments" section.

If the menuitem-style field is MIS_BITMAP, item-name must be a bit-map identifier instead of a name. The bit-map identifier must have been previously defined using a BITMAP statement, must be preceded by the \b character, and must be enclosed in double quotation marks.

menu-id
Specifies the menu-item identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range. Each identifier must be unique.

menuitem-style

MENUITEM Statement

Specifies the menu-item style. This value can be a combination of the following:

MIS_BITMAP Specifies that item-name is a bit map identifier.

MIS_BREAK Specifies that the menu has multiple columns of items in one pull-down menu or multiple lines of menus in the top-level menu.

MIS_BREAKSEPARATOR
Specifies that the menu has a vertical line between the columns in a pull-down menu.

MIS_BUTTONSEPARATOR
Specifies that the user can activate the menu item only by using the mouse. The text is centered in the item, rather than left justified. This option is used for the Help item on the right side of the menu bar.

MIS_HELP Specifies that the menu item generates a WM_HELP message.

MIS_OWNERDRAW Specifies that the menu item is drawn by the owner window.

MIS_SEPARATOR Specifies that the menu item is a menu separator. Although the item-name and menu-id fields are ignored, you must still give values if you specify this style.

MIS_STATIC Specifies that the user cannot choose the menu item.

MIS_SUBMENU Specifies that the MENUITEM statement is to be treated as a SUBMENU statement. When you specify this option, you must follow the MENUITEM statement with a BEGIN and END clause, as in a SUBMENU statement. You may include a PRESPARAMS statement immediately after the BEGIN keyword.

MIS_SYSCOMMAND
Specifies that the menu item generates a WM_SYSCOMMAND message.

MIS_TEXT Specifies that item-name is a character string. This is the default option.

menuitem-attribute Specifies the menu-item attributes. This value can be a combination of the following:

MIA_CHECKED Places a check mark next to the menu-item name.

MIA_DISABLED Disables the menu item, preventing the system from generating a message when the user chooses the command.

MIA_FRAMED Places a frame (heavy border) around the menu item.

MIA_HILITED Places a highlight on the menu-item name when it is displayed, by inverting the name and background.

MIA_NODISMISS Causes a submenu or menu item to remain displayed after the user chooses an item.

Comments

You can use the \t or \a character combination in any item name. The \t character inserts a tab when the name is displayed and is typically used to separate the menu-item name from the name of an accelerator key. The \a character aligns to the right all text that follows it. These characters are intended to be used for menu items in submenus only. The width of the displayed submenu is always adjusted so that there is at least one space (and usually more) between any pieces of text separated by a \t or a \a. (When compiling the menu resource, the compiler stores the \t and \a characters as control characters. For example, the \t is stored as 0x09.)

A tilde (˜) character in the item name indicates that the following character is used as a mnemonic character for the item. When the menu is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the menu item by pressing the key corresponding to the underlined mnemonic character.

Example

This example creates a menu item named Alpha. The item identifier is 101.

```
MENUITEM "Alpha", 101
```

This example creates a menu item named Beta. The item identifier is 102. The menu item has a text style and a checked attribute.

```
MENUITEM "Beta", 102, MIS_TEXT, MIA_CHECKED
```

This example creates a menu separator between menu items named Gamma and Delta.

```
MENUITEM "Gamma", 103
MENUITEM SEPARATOR
MENUITEM "Delta", 104
```

This example creates a menu item that has a bit map instead of a name. The bit-map identifier, 1, is first defined using a BITMAP statement. The identifier for the menu item is 301. Note that a sign must be placed in front of the bit map identifier in the MENUITEM statement.

```
BITMAP 1 mybitmap.bmp

MENUITEM "#1", 301, MIS_BITMAP
```

MESSAGETABLE Statement

Syntax:

```
MESSAGETABLE  [load-option]  [ mem-option][codepage]
BEGIN
string-id string-definition
    .
    .
    .
END
```

The MESSAGETABLE statement creates one or more string resources for an application. A string resource is a null-terminated character string that has a unique string identifier. A string resource can be loaded from the executable file when needed by using the DosGetResource function with the RT_MESSAGE resource type. RT_MESSAGE resources are bundled together in groups of 16, with any missing IDs replaced with zero length strings. Each group, or bundle, is assigned a unique sequential ID. The resource string ID is not necessarily the same as the ID specified when using DosGetResource. The formula for calculating the ID of the resource bundle, for use in DosGetResource, is as follows:

```
bundle ID = ( id / 16) +1,
```

where id is the string ID assigned in the RC file.

Thus, bundle 1 contains strings 0 to 15, bundle 2 contains strings 16 to 31, and so on. Once the address of the bundle has been returned by DosGetResource (using the calculated ID), the buffer can be parsed to locate the particular string within the bundle. The number of the string is calculated by the formula:

```
string = id % 16
```

(string = remainder for id/16).

The buffer returned consists of the CodePage of the strings in the first USHORT, followed by the 16 strings in the bundle. The first BYTE of each string is the length of the string (including the null terminator), followed by the string and the terminator. A zero length string is represented by two bytes: 01 (string length) followed by the null terminator.

You can provide any number of MESSAGETABLE statements in a resource script file. The compiler treats all the strings from the various MESSAGETABLE statements as if they belonged to a single statement. This means that no two strings in a resource script file can have the same string identifier.

Although the MESSAGETABLE and STRINGTABLE statements are nearly identical, most applications use the STRINGTABLE statement instead of the MESSAGETABLE statement to create string resources.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the DosGetResource or DosGetResource2 function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory.
- **DISCARDABLE** System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

codepage Specifies a code page value. See “CODEPAGE Statement” on page 718 for a list of valid code pages.

string-id Specifies the character-string identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range. The value can be specified in decimal or hexadecimal notation. Each string identifier in a resource script file must be unique.

string-definition Specifies a character string. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the string, you must provide the double quotation mark twice.

Comments

You can continue a string on multiple lines by terminating the line with a backslash (\) or by terminating the line with a double quotation mark (") and then starting the next line with a double quotation mark.

Example

This example creates two string resources whose string identifiers are 1 and 2.

```
MESSAGETABLE
BEGIN
    1 "Filename not found"
    2 "Cannot open file for reading"
END
```

MLE Statement

Syntax:

```
MLE text, id, x, y, width, height[, style]
```

The MLE statement creates a multiple-line entry-field control. The control is a rectangle in which the user can type and edit multiple lines of text. The control displays a pointer when the user selects it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. By using the mouse or the DIRECTION keys, the user can select the character or characters to delete or select the place to insert new characters. The MLE statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_MLE. If you do not specify a style, the default style is MLS_BORDER, WS_GROUP, and WS_TABSTOP.

text Specifies text that is displayed in the rectangular area of the control. If the MLS_READONLY style is not specified, the user can edit the text. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_MLE. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a multiple-line entry-field control that is not labeled.

```
MLE "", 101, 10, 10, 50, 100
```

NOTEBOOK Statement

Syntax:

```
NOTEBOOK   id, x, y, width, height[, style]
```

The NOTEBOOK statement creates a notebook control within the dialog window. This control is used to organize information on individual pages so that it can be located and displayed easily. The NOTEBOOK statement defines the identifier, position, dimensions, and attributes of a notebook control. The predefined class for this control is WC_NOTEBOOK. If you do not specify a style, the default style is WS_TABSTOP and WS_VISIBLE.

id Specifies the control identifier. The value is any integer -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

y Specifies the y-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

width Specifies the width of the control. The value is any integer 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. The value is any integer 0 through 65535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_NOTEBOOK. You can use the bitwise OR (|) operator to combine styles.

Comments

The NOTEBOOK statement is used only in a DIALOG or WINDOW statement.

Example

This example creates a notebook control at position (20, 20) within the dialog window. The notebook has a width of 200 character units and a height of 50 character units. Its resource ID is 201. The tabs style BKS_ROUNDEDTABS specification overrides the notebook default style of square tabs. The default styles WS_TABSTOP and WS_GROUP are both in effect, though only the latter is specified.

```
#define    IDC_NOTEBOOK     201
#define    IDD_NOTEBOOKDLG  503
DIALOG "Notebook", IDD_NOTEBOOKDLG, 11, 11, 420, 420, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    NOTEBOOK   IDC_NOTEBOOK, 20, 20, 200, 400, BKS_ROUNDEDTABS | WS_GROUP
  END
```

POINTER Statement

Syntax:

```
POINTER pointer-id  [load-option]  [ mem-option]
[codepage]  filename
```

The POINTER statement defines a pointer resource for an application. A pointer resource, typically created by using the OS/2 Icon Editor, is a bit map defining the shape of the mouse pointer on the screen. The POINTER statement copies the pointer resource from the file specified in the filename field and adds it to the application's other resources. A pointer resource can be loaded from the executable file when needed by using the WinLoadPointer function.

You can provide any number of POINTER statements in a resource script file, but each statement must specify a unique pointer-id value.

pointer-id Specifies the pointer-resource identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the WinLoadPointer function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory.
- **DISCARDABLE** System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

codepage Specifies a code page value. See “CODEPAGE Statement” on page 718 for a list of valid code pages.

filename Specifies the name of the file containing the pointer resource. If the file is not in the current directory, filename must be preceded by a full path.

Example

This example defines a pointer whose pointer identifier is 10. The pointer resource is copied from the file custom.cur.

```
POINTER 10 custom.cur
```

PRESPARAMS Statement

Syntax:

```
PRESPARAMS presparam, value, presparam, value, ...
```

The PRESPARAMS statement defines presentation fields that customize a ;i2 refid=control.customizing presentation fields dialog box, menu, window, or control. PRESPARAMS data is a series of types and values. The window procedure of the dialog box, menu, window or control receives and processes this data when the item is created. The data for custom controls can be in any format.

presparam Specifies a presentation-field type.
value Specifies the presentation-field value.

Comments

PRESPARAMS is often used to supply data to control the appearance of the customized window when it is first created. For example, the PRESPARAMS statement may specify the colors to be used in the window.

Example

This example creates a menu resource with a menu identifier of 1. The PRESPARAMS statement specifies that the following three menu items be displayed in the 12-point Helvetica font.

```
MENU 1
BEGIN
    PRESPARAMS PP_FONTNAMESIZE, "12.Helv"
    MENUITEM "New", 100
    MENUITEM "Open", 101
    MENUITEM "Save", 102
END
```

PUSHBUTTON Statement

Syntax:

```
PUSHBUTTON text, id, x, y, width, height[, style]
```

The PUSHBUTTON statement creates a pushbutton control. The control is a round-cornered rectangle containing the given text. The control sends a message to its parent whenever the user chooses the control. The PUSHBUTTON statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_PUSHBUTTON and WS_TABSTOP.

text Specifies text that is centered in the rectangular area of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice. A tilde (˜) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_BUTTON. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a pushbutton control that is labeled "OK."

```
PUSHBUTTON "OK", 101, 10, 10, 100, 100
```

RADIOBUTTON Statement

Syntax:

```
RADIOBUTTON text, id, x, y, width, height[, style]
```

The RADIOBUTTON statement creates a radio-button control. The control is a small circle that has the given text displayed to its right. The control highlights the circle and sends a message to its parent window when the user selects the button. The control removes the highlight and sends a message when the button is next selected. The RADIOBUTTON statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of a control window. The predefined class for this control is WC_BUTTON. If you do not specify a style, the default style is BS_RADIOBUTTON.

text Specifies text that is displayed to the right of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice. A tilde (˜) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_BUTTON. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a radio-button control that is labeled "Italic."

```
RADIOBUTTON "Italic", 101, 10, 10, 24, 50
```

RCDATA Statement

Syntax:

```
RCDATA resource-id
BEGIN
data-definition  , data-definition   ...
    .
    .
    .
END
```

The RCDATA statement defines a custom-data resource for an application. The custom data can be in whatever format the application requires. You can provide any number of RCDATA statements in a resource script file, but each statement must specify a unique resource-id value. A custom-data resource can be loaded from the executable file when needed by using the DosGetResource or DosGetResource2 functions with the RT_RCDATA resource type.

resource-id Specifies the custom-data identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

data-definition Specifies the custom data. The data may be simple expressions or strings.

Example

This example defines custom data that has a resource identifier of 5.

```
RCDATA 5
BEGIN
    "E. A. Poe", 1849, -32, 3L, 0x8000000l, 3+4+5
END
```

RCINCLUDE Statement

Syntax:

```
RCINCLUDE filename
```

The RCINCLUDE statement causes RC to process the resource script file specified in the filename field along with the current resource script file. The contents of both files are compiled by RC and the results are placed in one binary resource file and/or executable file.

filename Specifies the name of the resource script file to be included. If the file is not in the current directory, filename must be preceded by a full path.

Comments

RCINCLUDE statements are processed before any other processing is done, including preprocessing by RCPP.EXE, which removes comments, replaces values in the define directives, and so on.

When specifying a high performance file system (HPFS) file name on an RCINCLUDE statement, enclose the path and file name in double quotes; for example:

```
   RCINCLUDE "d:\project\long dialog.dlg"
```

Double quotes enables the Resource Compiler to recognize a name containing embedded blank characters.

Example

This example includes the file DIALOGS.RC as part of the current resource script file.

```
RCINCLUDE dialogs.rc
```

RESOURCE Statement

Syntax:

```
RESOURCE type-id resource-id  [load-option]
[mem-option][code page]  filename
```

The RESOURCE statement defines a custom resource for an application. A custom resource can be any data in any format. The RESOURCE statement copies the custom resource from the specified file and adds it to the application's other resources. A custom resource can be loaded from the executable file when needed by using the DosGetResource or DosGetResource2 function and specifying the resource's type and resource identifier.

You can provide any number of RESOURCE statements in a resource script file, but each statement must specify a unique combination of type-id and resource-id values. That is, RESOURCE statements having the same type-id value are permitted as long as the resource-id value for each is unique.

type-id Specifies the custom-resource type. This value must be an integer in the range 256 through 65 535, or a simple expression that evaluates to a value in that range. (Values 0 through 255 are reserved.)

resource-id Specifies the custom-resource identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the DosGetResource or DosGetResource2 function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory.
- **DISCARDABLE** System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

codepage Specifies a code page value. See “CODEPAGE Statement” on page 718. for a list of valid code pages.

filename Specifies the name of the file containing the custom resource. If the file is not in the current directory, filename must be preceded by a full path.

Example

This example defines a custom resource whose type identifier is 300 and whose resource identifier is 14. The custom resource is copied from the file CUSTOM.RES.

```
RESOURCE 300 14 custom.res
```

RTEXT Statement

Syntax:

```
RTEXT text, id, x, y, width, height[, style]
```

The RTEXT statement creates a right-aligned text control. The control is a simple rectangle displaying the given text right-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The RTEXT statement, which you can use only in a DIALOG or WINDOW statement, defines the text, identifier, dimensions, and attributes of the control. The predefined class for the control is WC_STATIC. If you do not specify a style, the default style is SS_TEXT, DT_RIGHT, and WS_GROUP.

text Specifies text that is right-aligned in the rectangular area of the control. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the text, you must include the double quotation mark twice.

id Specifies the control identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y Specifies the y-coordinate of the lower-left corner of the control. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

width Specifies the width of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. This value can be a combination of the styles specified for WC_STATIC. You can use the bitwise OR (|) operator to combine styles.

Example

This example creates a right-aligned text control that is labeled "Filename."

```
RTEXT "Filename", 101, 10, 10, 100, 100
```

SLIDER Statement

Syntax:

```
SLIDER   id, x, y, width, height[, style]
```

The SLIDER statement creates a slider control within the dialog window. This control lets the user set, display, or modify a value by moving a slider arm along a slider shaft. The SLIDER statement defines the identifier, position, dimensions, and attributes of a slider control. The predefined class for this control is WC_SLIDER. If you do not specify a style, the default style is WS_TABSTOP and WS_VISIBLE.

id Specifies the control identifier. The value is any integer -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

y Specifies the y-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

width Specifies the width of the control. The value is any integer 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. The value is any integer 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. The value can be any combination of the styles specified for WC_SLIDER. You can use the bitwise OR (|) operator to combine styles.

Comments

The SLIDER statement is only used in a DIALOG or WINDOW statement.

Example

This example creates a slider control at position (40, 30) within the dialog window. The slider has a width of 120 character units and a height of 2 character units. Its resource ID is 101. The style specification SLS_BUTTONSLEFT adds buttons to the left of the slider shaft. The default styles WS_TABSTOP and WS_VISIBLE are both in effect, though only the latter is specified.

```
#define    IDC_SLIDER       101
#define    IDD_SLIDERDLG    502
DIALOG "Slider", IDD_SLIDERDLG, 11, 11, 200, 240, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    SLIDER   IDC_SLIDER, 40, 30, 120, 16, SLS_BUTTONSLEFT | WS_VISIBLE
  END
```

SPINBUTTON Statement

Syntax:

```
SPINBUTTON   id, x, y, width, height[, style]
```

The SPINBUTTON statement creates a spinbutton control within the dialog window. This control gives the user quick access to a finite set of data. The SPINBUTTON statement defines the identifier, position, dimensions, and attributes of a spinbutton control. The predefined class for this control is WC_SPINBUTTON. If you do not specify a style, the default style is WS_TABSTOP, WS_VISIBLE, and SPBS_MASTER.

id Specifies the control identifier. The value is any integer -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

y Specifies the y-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

width Specifies the width of the control. The value is any integer 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. The value is any integer 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. The value is any combination of the styles specified for WC_SPINBUTTON. You can use the bitwise OR (|) operator to combine styles.

Comments

The SPINBUTTON statement is used only in a DIALOG or WINDOW statement.

Example

This example creates a spinbutton control at position (80, 20) within the dialog window. The spinbutton has a width of 60 character units and a height of 3 character units. Its resource ID is 302. The style specification SPBS_NUMERICONLY creates a control which accepts only the digits 0-9 and virtual keys. The default styles SPBS_MASTER, WS_TABSTOP, and WS_VISIBLE are all in effect, though only WS_TABSTOP is specified.

```
#define    IDC_SPINBUTTON   302
#define    IDD_SPINDLG    502
DIALOG "Spin button", IDD_SPINDLG, 11, 11, 200, 240, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    SPINBUTTON  IDC_SPINBUTTON, 80, 20, 60, 24, SPBS_NUMERICONLY | WS_TABSTOP
  END
```

STRINGTABLE Statement

Syntax:

```
STRINGTABLE  [load-option]  [mem-option][codepage]
BEGIN
string-id string-definition
    .
    .
    .
END
```

The STRINGTABLE statement creates one or more string resources for an application. A string resource is a null-terminated character string that has a unique string identifier. A string resource can be loaded from the executable file when needed by using the WinLoadString or with DosGetResource with the RT_STRING resource type. RT_STRING resources are bundled together in groups of 16, with any missing IDs replaced with zero length strings. Each group, or bundle, is assigned a unique sequential ID. The resource string ID is not necessarily the same as the ID specified when using DosGetResource. The formula for calculating the ID of the resource bundle, for use in DosGetResource, is as follows:

```
bundle ID = ( id / 16) +1
```

where id is the string ID assigned in the RC file.

Thus, bundle 1 contains strings 0 to 15, bundle 2 contains strings 16 to 31, and so on. Once the address of the bundle has been returned by DosGetResource (using the calculated ID), the buffer can be parsed to locate the particular string within the bundle. The number of the string is calculated by the formula:

```
string = id % 16
```

(string = remainder for id/16).

The buffer returned consists of the CodePage of the strings in the first USHORT, followed by the 16 strings in the bundle. The first BYTE of each string is the length of the string (including the null terminator), followed by the string and the terminator. A zero length string is represented by two bytes: 01 (string length) followed by the null terminator.

You can provide any number of STRINGTABLE statements in a resource script file. The compiler treats all the strings from the various STRINGTABLE statements as if they belonged to a single statement. This means that no two strings in a resource script file can have the same string identifier.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the WinLoadString function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

code-page Specifies a code page value. See “CODEPAGE Statement” on page 718 for a list of valid code page values.

- **FIXED** System keeps the resource at a fixed memory location.
- **MOVEABLE** System moves the resource as necessary to compact memory.
- **DISCARDABLE** System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

string-id Specifies the character-string identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range. The value can be specified in decimal or hexadecimal notation. Each string identifier in a resource script file must be unique.

string-definition Specifies a character string. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the string, you must include the double quotation mark twice.

Comments

You can continue a string on multiple lines by terminating the line with a backslash (\) or by terminating the line with a double quotation mark (") and then starting the next line with a double quotation mark.

Example

This example creates two string resources whose string identifiers are 1 and 2.

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
BEGIN
    IDS_HELLO   "Hello"
    IDS_GOODBYE "Goodbye"
END
```

SUBITEMSIZE Statement

Syntax:

```
SUBITEMSIZE  size
```

The SUBITEMSIZE statement specifies the size, in words, of each help subitem in a help subtable. The minimum size is two words, and each help subitem in a help subtable must be the same size. When used, the SUBITEMSIZE statement must appear after the HELPSUBTABLE statement and before the BEGIN keyword.

You do not need to use the SUBITEMSIZE statement if the help subitems are the default size (2).

size Specifies the size of each help subitem. This value must be an integer.

Example

The SUBITEMSIZE statement in this example specifies that each HELPSUBITEM statement contains three words.

```
HELPSUBTABLE 1
SUBITEMSIZE 3
BEGIN
    HELPSUBITEM IDCLD_FILEMENU, IDHP_FILEMENU, 5
    HELPSUBITEM IDCLD_HELPMENU, IDHP_HELPMENU, 6
END
```

SUBMENU Statement

Syntax:

```
SUBMENU text, submenu-id [, menuitem-style]
BEGIN
menuitem-definition
    .
    .
    .
END
```

The SUBMENU statement creates a submenu for a given menu. A submenu is a vertical list of menu items from which the user can choose a command.

You can provide any number of SUBMENU statements in a MENU statement, but each SUBMENU statement must specify a unique submenu-id value. You can provide any number of menuitem-definition statements in the SUBMENU statement. These define the menu items (commands) in the menu. The order of the statements determines the order of the menu items.

text Specifies the text of the submenu. This field must contain zero or more characters enclosed in double quotation marks. Character values must be in the range 1 through 255. If a double quotation mark is required in the string, you must include the double quotation mark twice. A tilde (˜) character in the item name indicates that the following character is used as a mnemonic character for the item. When the menu is displayed, the tilde is not shown, but the mnemonic character is underlined. The user can choose the menu item by pressing the key corresponding to the underlined mnemonic character.

submenu-id Specifies the submenu identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

menuitem-style Specifies the submenu style. This value can be a combination of MIS_ values. For details on the MIS_ values, see “MENUITEM Statement” on page 753.

menuitem-definition Specifies a PRESPARAMS or MENUITEM statement. You can use the PRESPARAMS statement to control the appearance of a submenu, such as the font and the foreground and background colors. If used, the PRESPARAMS statement must immediately follow the BEGIN keyword. For details about the PRESPARAMS statement, see “PRESPARAMS Statement” on page 762.

The MENUITEM statement defines an individual command in the given menu. For details, see “MENUITEM Statement” on page 753.

Example

This example creates a submenu named Elements. Its identifier is 2. The submenu contains three menu items, which are created by using MENUITEM statements.

```
SUBMENU "Elements", 2
BEGIN
    MENUITEM "Oxygen", 200
    MENUITEM "Carbon", 201, , MIA_CHECKED
    MENUITEM "Hydrogen", 202
END
```

undef Directive

Syntax: `undef name`

This directive removes the current definition of the specified name. All subsequent occurrences of the name are processed without replacement.

name Specifies the name to be removed. This value is any combination of letters, digits, and punctuation.

Example

This example removes the definitions for the names "nonzero" and "USERCLASS".

```
#undef     nonzero
#undef     USERCLASS
```

VALUESET Statement

Syntax:

```
VALUESET   id, x, y, width, height[, style]
```

The VALUESET statement creates a value set control within the dialog window. This control lets a user select one choice from a group of mutually exclusive choices. The VALUESET statement defines the identifier, position, dimensions, and attributes of a value set control. The predefined class for this control is WC_VALUESET. If you do not specify a style, the default style is WS_TABSTOP and WS_VISIBLE.

id Specifies the control identifier. The value is any integer -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

y Specifies the y-coordinate of the lower-left corner of the control. The value is any integer -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The coordinate is assumed to be in dialog units and is relative to the origin of the dialog window.

width Specifies the width of the control. The value is any integer 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The width is in n-character units.

height Specifies the height of the control. The value is any integer 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The height is in 1/8-character units.

style Specifies the control styles. The value is any combination of the styles specified for WC_VALUESET. You can use the bitwise OR (|) operator to combine styles.

Comments

The VALUESET statement is used only in a DIALOG or WINDOW statement.

Example

This example creates a value set control at position (40, 40) within the dialog window. The value set control has a width of 220 character and a height of 20 character units. Its resource ID is 302. The style specification VS_ICON creates a control to show items in icon form. The default styles WS_TABSTOP and WS_VISIBLE are both in effect, though only WS_TABSTOP is specified.

```
#define    IDC_VALUESET     302
#define    IDD_VALUESETDLG  501
DIALOG "Value set", IDD_VALUESETDLG, 11, 11, 260, 240, FS_NOBYTEALIGN |
        WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
  BEGIN
    VALUESET  IDC_VALUESET, 40, 40, 220, 160, VS_ICON | WS_TABSTOP
  END
```

WINDOW Statement

Syntax:

```
WINDOW text, id, x, y, width, height, class[ , style[  , framectl]]
  data-definitions
[ BEGIN
control-definition
    .
    .
    .
END ]
```

The WINDOW statement creates a window of the specified class. The statement defines the position and dimensions of the window relative to its parent window, as well as the window-box style. The WINDOW statement is typically used in a WINDOWTEMPLATE or FRAME statement.

Typically, only one WINDOW statement is used in a FRAME statement. It defines the client window belonging to the corresponding frame window. The optional BEGIN and END keywords enclose any CONTROL statements that are given with the window. CONTROL statements given in this manner represent child windows belonging to the window created by the WINDOW statement.

text Specifies the window title if the style specifies a title bar. This field must contain zero or more characters enclosed in double quotation marks. The character values must be in the range 1 through 255. If a double quotation mark is required in the title, you must include the double quotation mark twice.

id Specifies the window identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

x Specifies the x-coordinate of the lower-left corner of the window. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in dialog units. The position is relative to the origin of the parent window.

y Specifies the y-coordinate of the lower-left corner of the window. This value must be an integer in the range -32768 through 32767 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in dialog units. The position is relative to the origin of the parent window.

width Specifies the width of the window. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in n-character units.

height Specifies the height of the window. This value must be an integer in the range 0 through 65 535 or an expression consisting of integers and the addition (+) or subtraction (-) operator. The value is in 1/8-character units.

class Specifies the window class. This value can be one of the control classes specified in the "Control Classes" table in the *Presentation Manager Programmer Reference* or the name of the window class, enclosed in double quotation marks.

style Specifies the window style. This value can be any of the window, dialog box, or frame styles specified.

framectl Specifies the style of the frame controls belonging to the window. This value can be a combination of the styles specified in the table, "Frame-Control Styles." You can use the bitwise OR (|) operator to combine styles.

data-definitions Specifies a CTLDATA and/or PRESPARAMS statement. These statements define control and presentation data for the window. For more information, see "CTLDATA Statement" on page 724 and "PRESPARAMS Statement" on page 762.

control-definition Specifies a CONTROL statement or any one of several predefined control statements. These statements define the style, position, and dimensions of controls in the window.

Comments

The WINDOW statement can actually contain any combination of CONTROL, DIALOG, and WINDOW statements. Typically, a WINDOW statement contains one or no such statements.

Example

This example creates a client window belonging to the frame window. The client window belongs to the "MyClientClass" window class and has the standard window identifier FID_CLIENT.

```
WINDOWTEMPLATE 1
BEGIN
    FRAME "My Window", 1, 10, 10, 320, 130,
             0, FCF_STANDARD | FCF_VERTSCROLL
    BEGIN
        WINDOW "", FID_CLIENT, 0, 0, 0, 0, "MyClientClass"
    END
END
```

WINDOWTEMPLATE Statement

Syntax:

```
WINDOWTEMPLATE window-id  [load-option]  [ mem-option][code-page]
BEGIN
window-definition
    .
    .
    .
END
```

The WINDOWTEMPLATE statement creates a window template. A window template consists of a series of statements that define the window identifier, load and memory options, window dimensions, and controls in the window. The window template can be loaded from the executable file by using the WinLoadDlg function.

You can provide any number of window templates in a resource script file, but each template must have a unique window-id value.

window-id Specifies the window identifier. This value must be an integer in the range -32768 through 32767, or a simple expression that evaluates to a value in that range.

load-option Specifies when the system loads the resource from the executable file into memory. This value must be one of the following:

- **PRELOAD** System loads the resource when the application starts.
- **LOADONCALL** System loads the resource when the application calls the WinLoadDlg function. This is the default option.

mem-option Specifies how the system manages the resource when it is in memory. This value must be one or more of the following:

FIXED System keeps the resource at a fixed memory location.

MOVEABLE System moves the resource as necessary to compact memory.

DISCARDABLE System discards the resource if it is no longer needed.

The default setting is MOVEABLE and DISCARDABLE.

code-page Specifies a code page value. 📖 See “CODEPAGE Statement” on page 718 for a list of valid code pages.

window-definition Specifies a WINDOW statement. The statement defines the dimensions and style of the given window. For details about the statement, 📖 see “WINDOW Statement” on page 777.

Comments

A WINDOWTEMPLATE statement can contain DIALOG, CONTROL, and WINDOW statements. Typically, only one WINDOW statement is used in the WINDOWTEMPLATE statement.

Dialog Editor

You use the *Dialog Editor* to create and modify dialog boxes, and to create and modify the controls and text within dialog boxes. As you create the dialog box and its controls, you see them on the screen as the user will see them when your program is run. You can place each dialog box and its controls where you want them on the screen. In addition, you can test the dialog box before you incorporate it into your application.

Each dialog box and control can have either an integer identifier or a symbolic identifier that equates to an integer identifier. You use the identifier in your application to refer to the dialog box or control. If you intend to use symbolic identifiers in your application, you must enter the symbolic and integer identifiers in an include file. If you do not use symbolic names, the Dialog Editor supplies an integer identifier for each control and for the dialog box itself. You can use the Dialog Editor to create the include file, or you can use a text editor to create the include file before using the Dialog Editor.

It is good programming practice to plan the resources that your application will use and to choose a naming and numbering convention for the symbolic or integer identifiers *before* you create them. Keep the include file separate from other include files used by your application. Include files used by the Dialog Editor can contain only #define statements that define their symbolic identifiers and equivalent integers.

Although the Dialog Editor draws dialog boxes and controls on the screen so you can see what they look like when used by your application, it does *not* save them as graphics. Instead, the Dialog Editor saves them in an ASCII-text format file that has a .DLG extension. Refer to the dialog template section of this chapter.

The Dialog Editor also creates a compiled form of the .DLG file in a resource file with a .RES extension. The .DLG and .RES files can contain more than one dialog box. The resource file can contain other application resources, such as icons, bit maps, and string tables. It is attached to the executable (.EXE) file of the application during the compile and link process.

Designing Dialog Boxes

Dialog boxes should be designed to clearly identify the information that the user is required to complete. The following are a few Common User Access guidelines:

- Lay out the controls in columns, starting at the upper-left corner, for left-to-right or top-to-bottom scanning.
- Vertically and horizontally align selection and entry fields so that the cursor moves in a straight line.
- Arrange the controls in the sequence in which the user would complete them.
- If there are only a few entry fields, locate them at the top of the dialog box.
- Make groups of controls obvious by use of group boxes and white space.
- Align group boxes, where possible. Group boxes can be extended to the right to line up with other group boxes.
- Use field identifiers to identify the purpose of single and multiple groups of choices.

Creating a Dialog Box

To run the Dialog Editor, select **Dialog Editor** from the **Development Tools** folder. The main window appears, displaying the menu bar choices **File**, **Edit**, **Control**, **Arrange**, **Options**, and **Help**. On line help that tells you how to use the editor is available on most Dialog Editor windows.

To create a new dialog box, start with either one of the following steps:

- Select **New Dialog** from the **Edit** menu. The editor opens new files with the extensions .RES and .DLG. This also opens a new include file.
- Select **New** from the **File** menu. This opens new files with the extensions .RES and .DLG. You can open a new include file or an existing one.

The above steps have the same effect.

When you edit a dialog box, the names of the resource and include files are shown in the title bar of the Dialog Editor. If you are editing a new file that has not yet been named or saved, **(Untitled)** appears in the title bar in place of a name. If **(Untitled)** appears in the title bar in place of a name, there are unsaved changes.

The **Dialog Box ID** field appears in the status area. A default integer number is supplied in the entry field. Type a symbolic identifier for the dialog box, such as MYDIALOG. Tab to the integer field and type the integer number. Press Enter to place them both in the include file.

The new dialog box appears in the lower-left corner of the editor screen enclosed by a frame. The frame contains eight small squares called drag handles, which allow you to change the width and height of the selected item. This indicates that the dialog box is selected for editing. If you are creating a new dialog box, the dialog is automatically selected; at all other times, before you edit the dialog box or a control, you must click on it to select it.

To continue creating the new dialog box, follow these steps:

1. Make the dialog box larger by clicking on one of its drag handles with the left mouse button and dragging until the box is the size you want it to be. This can be done in one operation by clicking on the upper-right corner of the frame and dragging diagonally upwards and to the right.

   Information about the item you are editing is displayed in the **Selected Item Status** box in the left half of the status area. As you move the shadow box, the x-y-coordinates change. These are the coordinates of the origin of the dialog box relative to the origin of the window. The cx-cy-coordinates are the width and height of the dialog box. The symbolic identifier is also shown.

2. Select **Styles** from the **Edit** menu. The Dialog Box Styles pop-up window appears.

3. Click on the text entry field in the status area, and then type the dialog box title (for instance, `Sample dialog box`) into the field.

4. Press Enter and the title appears at the top of your dialog box.

You can reposition the entire dialog box by moving the pointer inside the top area enclosed by the frame, holding the left mouse button down, and dragging the shadow box across the screen. When the shadow box is in the position where you want the dialog box to appear, release the mouse button. The dialog box appears in that position. Alternatively, you can move the dialog box using the keyboard arrow keys. You can reposition the dialog box at any time during the edit.

Using a Grid

Before you start adding controls to the dialog box, you might want to first select the grid option to make laying out your dialog easier.

You can use a mouse to place controls in a dialog box and to move the controls in line with each other. However, you can more accurately position the controls by using the keyboard arrow keys or mouse after grid values have been set.

The **Settings-change** dialog lets you set the number of character spaces (in dialog units) by which you can move dialog boxes and controls when using the Dialog Editor.

To set the grid size, follow these steps:

1. Select **Settings** from the **Arrange** menu. The **Settings-change** dialog is displayed. The initial grid setting for both *x* and *y* is 1 unit.
2. Change the *x*-setting to 10 and the *y*-setting to 5. Click on **OK**.

The horizontal (x) and vertical (y) values are in dialog units. A horizontal dialog unit is 0.25 of the standard character size. A vertical dialog unit is 0.125 of the standard character size. For example, if you move a control to the left or the right (using the mouse or keyboard arrow keys) with **x** set at 20, it moves in steps of twenty dialog units.

When you subsequently position dialog boxes or controls, the objects move by the specified number of dialog units on an invisible grid. Large values make it easier to align controls, while small values allow you to position controls in the dialog box more precisely.

Now that the grid is in place, you are ready to start adding controls.

Ordering Control Groups

This option allows you to gather controls into groups and to change the order in which the tab keys and arrow keys move the selection cursor around the controls.

When you use group boxes to group controls, always create the group box before the controls that are to go inside it.

It is good practice to put group markers around all separate groups of controls, including putting a marker before the first control in the list.

The list box shows the order in which the selection cursor moves between the controls when the user presses the arrow and tab keys. (The coordinate position of a control when displayed in the dialog box does not affect the order.) Initially, the controls are listed in the order in which they were created.

There are three functions involved in grouping controls:

- Setting Group Markers
- Setting Tab Markers
- Moving Control Order

Setting Group Markers

To set up groups in a dialog that has various types of controls, follow these steps:

1. Select **Order Groups** from the **Arrange** menu. The **Groups - order** dialog is displayed.
2. Click on the first radio button in the list box.
3. Click on the **Group Marker** push button. A group marker is now displayed between the **Text** control and the first radio button in the list.
4. Scroll down the list and click on the first push button in the list. Click on the **Group Marker** push button. This has organized your controls into groups of text, radio buttons, check boxes, and push buttons.

Setting Tab Markers

After setting group markers, you will want to set tab-stops. The controls marked with an asterisk already have tab-stops.

To make the tab-stop at only the first control in each group, delete the tab-stops from the second and third radio button and check box, following these steps:

1. Click on the second radio button in the list to mark it.
2. Click on the **Delete Tab** push button.

3. Repeat the above steps for the third radio button, and then perform the same operation for the second and third check box in the list. When this is complete, press Enter.

Moving Control Order

You can move controls in the list and then see during testing how the changes affect the movement of the cursor. To change the position of a control in the list, follow these steps:

1. Click on the name of the control to select it.
2. Position the pointer in the list where you want the name to appear. The pointer changes shape to a short horizontal line when it is over a place where you can insert the name.
3. To insert the control name, click the mouse button.

After grouping controls, you might want to test or edit the dialog, or enter additional controls.

Adding Controls

The control menu lists, in alphabetic order, all the controls that you can put in a dialog box. To add controls, follow these steps:

1. Select a control from the **Control** menu or click on an icon on the Control Palette at the right side of the window.

   The pointer becomes a small plus sign (+) in a square. The center marks the position where the lower-left corner of the frame for the control will be set.
2. Click the mouse to position the control.
3. A dialog might appear (depending on the type of control) in which you must enter data or check preferences to define the control. Complete this and close the dialog.

For an example of adding controls in a typical dialog, see “Adding Controls Example” on page 787.

You might want to test the dialog.

For detailed descriptions of individual controls and how they work, see the individual controls in the on line help (while using the Dialog Editor) by following these steps:

1. Select **Help Index** from the **Help** menu (or press F1 and select **Help Index**).
2. Select **Options** or press Alt-O.

3. Select **Contents** or press Ctrl-C.
4. Select **Control Menu** for an alphabetic list of controls, or select **Control Palette** for the icons as they appear on the Control Palette.
5. Select the control you want to read about.

Adding Controls Example

The control menu lists, in alphabetic order, all the controls that you can put in a dialog box. The sample dialog is 📖 “Sample Dialog Template File” on page 795 . To add controls for a sample dialog, follow these steps:

1. Select **Text** from the **Control** menu or select a control by clicking on its icon on the Control Palette at the right side of the window.

   The pointer becomes a small plus sign (+) in a square. The center marks the position where the center of the control will be.

2. Position the pointer inside the dialog box near the upper-left corner and click the mouse.
3. Type `Student Level:` in the **Text** entry field. Observe that the next sequential integer is supplied in the **Symbol** entry field. Press Enter.
4. Replace the symbol with **ID_GRAD** and press Enter.

   The Dialog editor assigns the next integer to the symbolic identifier you entered and places it in the include file. This is another technique for entering symbolic identifiers.

5. To view or change the include file at any time, select **Symbols** from the **Edit** menu. The **Symbols** dialog appears.

   The symbolic and integer identifier for the dialog box and the text control are displayed in the list box. The dialog allows you to add, delete, and change the identifiers and to view the hexadecimal equivalents of the integers.

6. Select the **OK** push button to remove the dialog and register any changes. Select **Cancel** if you have not made any changes.
7. In your dialog box, the static control is not large enough for you to see all the text. To remedy this, click on the text, and a frame appears around it. Drag the right-hand edge of the frame to the right to enlarge the field.

   When you release the mouse button, you should be able to see all the tex. When a control has a frame around it, it is selected and you can use a shadow box to position it, as you did with the dialog box.

8. To add another control, select **Radio Button** from the **Control** menu and position the cursor just beneath the **Student Level** text. Press Enter.

9. Type Elementary in the **Button Text** entry field and press Enter. Drag the right edge of the frame that surrounds the radio button until you can see all of the text.

10. Select **Radio Button** again and type Intermediate in the **Text** entry field. Position this radio button below the first one.

11. Select **Radio Button** again and type Advanced in the **Text** entry field. Position this radio button below the other two.

12. Select **Group Box** from the **Control** menu. Position the cursor to the right of the column of radio buttons and press Enter.

13. Type Media in the **Text** entry field and press Enter to title the group box.

14. Click on the lower-right corner of the group box frame and drag it diagonally down and to the right to enlarge it. The bottom of the group box frame should be lower than the last of the radio buttons, and the right-hand side of the group box should be almost at the far right of the dialog box. This is to make room for a group of check boxes that will go inside the group box.

    When you use group boxes to group controls, you always create the group box before the controls that are to go inside it.

15. Select **Check Box** from the **Control** menu. Position the cursor inside the group box in line with the first radio button in the list, and click the mouse.

16. Type TextBooks in the **Button Text** entry field and press Enter. Enlarge the frame of the check box until all of the text is displayed.

17. Select **Check Box** again and position the cursor below the first check box. Type Video in the **Text** entry field and click Enter. Enlarge the check box frame until all of the text is displayed.

18. Select **Check Box** again and position the cursor below the previous two check boxes. Type Diskettes in the **Text** entry field.

    In the left-hand side of the dialog box, you should now have a column of radio buttons with a heading of **Student Level**, and on the right a group box with a heading of **Media** that contains three check boxes.

19. Finally, add three push buttons to the dialog box. Select **Pushbutton** from the **Control** menu. Position the cursor in the lower-left side of the dialog box and click the mouse. Type OK in the **Text** entry field and press Enter.

20. Position another push button to the right of the first one (in the lower middle of the dialog box) and type Cancel in the **Text** entry field.

21. Select a third push button and position it to the right of the second. Type Help in the **Text** entry field.

The dialog box and its controls are now complete.

Try selecting each of the controls, and observe the information in the **Selected Item Status**. It holds information about each control that you edit.

You might now want to test the dialog box.

Selecting Color and Font

The Presentation Parameters dialog allows you to select the color and font for individual controls or for an entire dialog box.

You can select all of the following:

- Foreground Color
- Background Color
- Foreground Color Highlight
- Background Color Highlight
- Disabled (greyed out) Foreground Color
- Disabled (greyed out) Background Color
- Font Size
- Font Name

To set presentation parameters, follow these steps:

1. Select a control or the dialog box.
2. Select **Presentation Parameters** from the **Edit** menu.
3. Type the number, from 1 to 255 parts of each color, in the appropriate fields.
4. Type the font size and name, if you want to change the default, in the last two fields.
5. Select **OK** or press Enter to close the dialog.

You might now want to test the dialog.

Arranging Controls

The Arrange menu allows you to arrange and align controls in a logical and easy-to-understand layout.

Align Aligns controls along an edge.

Even spacing Evenly spaces controls

Same size Sets controls to the same size.

Push buttons Arranges push buttons.

Order groups Displays the **Groups-order** dialog, so you can change the order of controls and groups.

Settings Displays the **Settings-change** dialog, so you can change the grid and spacing constants.

Changing the Dialog Box

To change the properties of a dialog box or a single control, use the following functions of the **Edit** menu:

- Select **New Dialog** to create another dialog box in the same resource file. Your existing dialog box will stay in memory.
- Select **Select Dialog** to switch to another open dialog box.
- Select **Symbols** to define symbols.

Eight of the editing functions require that you first select the control to be edited. The selected control will appear in the *drag window*, surrounded by eight dots, one in each corner and one at the midpoint of each side.

The following functions require that a control must first be selected:

- Select **Cut** to cut a control you would like to move or delete.
- Select **Copy** to copy to the clipboard a control you would like to duplicate elsewhere in the same dialog or in another dialog.
- Select **Paste** to place a control you have marked with **Cut** or **Copy**.
- Select **Clear** to erase a control.
- Select **Duplicate** to create another control in this dialog box that is identical to the selected control.
- Select **Styles** to define the style of the selected control.
- Select **Presentation parameters** to select the colors and fonts.
- Select **Size to text** to adjust the size of an entry field to the text inside.

Using the Options Menu

On the Options menu, a check mark next to each option shows whether it is selected (on) or not (off).

To toggle your selection of options on and off, use the following functions of the Options menu:

- Select **Test mode** to test the dialog.
- Select **Hex mode** to toggle between hexadecimal and decimal display of ID Values of symbols.
- Select **Translate mode** to toggle translate mode on and off.
- Select **Enable 2.x styles** to use controls and their styles which are specific to OS/2 2.x, but not prior releases.
- Select **Show status area** to toggle display of the status area on and off.

Testing the Dialog Box

To test the dialog box, select **Test Mode** from the **Options** menu. The dialog box is displayed as it will appear to the user in a program. In test mode, you can select controls, and their appearance changes in the same way as they do in an application. To return to work mode, click on **Test Mode** again to de select it.

If you want to make changes, you can edit the dialog box.

Ending an Edit Session

To end the edit session, select **Close** from the system pull-down menu. You see prompts for the file names of the files you want to save.

If you want to edit the same file the next time you use the editor, select **Open** from the **File** menu.

Dialog Templates

The Dialog Editor creates an ASCII text file that has the file-name extension .DLG. The compiled form of this file, created using the Resource Compiler, has the file-name extension .RES.

The .DLG file contains a series of statements, collectively termed a *dialog template*, that define each dialog box and each control in each dialog box. The statement for each dialog box contains the data required to create it, namely its class, size, position, window text, and any other special information required for the window.

Normally, the template consists of a dialog box window followed by the controls contained within it, which are child windows.

The first statement in the template is the DLGINCLUDE statement, which specifies the file name of the include file.

The next statement is the DLGTEMPLATE statement, which specifies the symbolic identifier of the dialog box (MYDIALOG). The DLGTEMPLATE statement also specifies any loading and memory options. The actual dialog template is contained within the first BEGIN and last END statement. There is a CONTROL statement for each of the controls in the dialog box. The CONTROL statement is a general statement that is followed by parameters that further specify the control, such as:

- Text, where appropriate. For example, the text **OK** is defined for one of the push buttons.
- Application-defined symbolic or integer identifiers for each control. Your application uses the identifier to track the responses from controls. For example, ID_NULL is the identifier of the text control.
- The types and positions of the various controls. For example, the group box control is a control window of window class WC_STATIC. The **Cancel** and **Help** push buttons are of window class WC_BUTTON.
- The appearance and operation of the dialog box and its controls, which are specified in detail by combinations of style parameters. For example, the check boxes have a class style of BS_CHECKBOX, and radio buttons have a class style of BS_RADIOBUTTON. You can also specify appropriate WS_* styles.

If necessary, you can use a text editor to edit the .DLG file, for example, to *fine-tune* the dialog template produced by the dialog box editor. You can even use a text editor to produce your own .DLG file. The Dialog Editor uses the general CONTROL statement with window classes and control styles to define controls.

You can use the CONTROL statement in the same way to define your controls, or you can use any of several predefined control statements that give you the same result. For example, the predefined control statement PUSHBUTTON gives you a WC_BUTTON class window with default styles of BS_PUSHBUTTON and WS_TABSTOP.

The predefined controls are the same ones you would use to write a resource file yourself. The controls are described in “Statements and Directives” on page 707. control window. The predefined class for

A dialog template can be in either of the following:

- A resource.res file (generated from the .DLG file by the Resource Compiler)
- A block of memory that has the DLGTEMPLATE data structure, in which case you use `WinCreateDlg` to create the dialog box from the template.

The dialog template uses device-independent *dialog units* for the coordinate system that define the layout of controls in the dialog box.

A dialog unit is expressed in terms of the *default standard character size*, which can vary from device to device. You do not need to put code in your application to reformat the dialog box when displaying it on different devices. (Dialogs might need editing if a different system font is loaded.) A horizontal dialog unit is 0.25 of the standard character size. A vertical dialog unit is 0.125 of the standard character size. Dialog units are expressed as offsets from the origin (lower-left corner) of the dialog box.

A dialog template is a general structure. It could be termed a window template, because you can use it to define any window in an application. If you prefer, use the statement WINDOWTEMPLATE instead of DLGTEMPLATE, because it is functionally identical. This could reduce the initialization phase of the application to registering the application window classes and calling WinLoadDlg to load the template.

If you use the Dialog Editor to define a standard window, you will have to edit the resulting .DLG file to ensure that you have a client window and the required parent-child relationships. You will also have to use WinLoadMenu in your application, to create a menu bar for the window, because you cannot create menus using the Dialog Editor.

The .RES file is an object-format compiled version of the .DLG file, created when the Dialog Editor compiles the dialogs. The Dialog Editor uses the .RES file as input on any subsequent edit of the same dialog. This means that, if you use a text editor to fine-tune a .DLG file, and you want subsequently to re-edit the dialog using the Dialog Editor, you must first use the Resource Compiler to generate a new .RES file from the .DLG file.

Your application can use either the .RES file output by the Dialog Editor or a .RES file created from the .DLG file and the other resources. If your application uses the .DLG file, it must be included by the resource script file of your application.

Dialog Editor

The rcinclude statement includes the .DLG file created by the Dialog Editor; for example:

```
rcinclude dbe.dlg /* Includes .DLG file */
```

The corresponding .H file created by the Dialog Editor must also be included in the .RC file.

Using OS/2-defined control windows, OS/2 draws and operates the controls specified in the resource file for your application. Controls are windows and can be used within any other window.

Sample Dialog Template File

The following dialog template is used for the dialog described in “Adding Controls Example” on page 787.

```
DLGINCLUDE 1 "DBE.H"


DLGTEMPLATE mydialog LOADONCALL MOVEABLE DISCARDABLE

BEGIN

    DIALOG "Sample Dialog Box", mydialog, 11, 8, 170, 105,
        FS_NOBYTEALIGN | FS_DLGBORDER | WS_VISIBLE |
        WS_SAVEBITS, FCF_TITLEBAR

    BEGIN

        CONTROL "Student Level:", id_null, -1, 94, 63, 9, WC_STATIC,
            SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE

        CONTROL "Elementary", 258, 7, 82, 62, 11, WC_BUTTON,
            BS_RADIOBUTTON | WS_GROUP | WS_TABSTOP | WS_VISIBLE

        CONTROL "Intermediate", 259, 7, 67, 73, 9, WC_BUTTON,
            BS_RADIOBUTTON | WS_VISIBLE

        CONTROL "Advanced", 260, 7, 51, 52, 13, WC_BUTTON,
            BS_RADIOBUTTON | WS_VISIBLE

        CONTROL "Media", 261, 87, 48, 75, 54, WC_STATIC,
            SS_GROUPBOX | WS_GROUP | WS_VISIBLE

        CONTROL "Textbooks", 262, 97, 83, 60, 10, WC_BUTTON,
            BS_CHECKBOX | WS_TABSTOP | WS_VISIBLE

        CONTROL "Video", 263, 97, 68, 46, 10, WC_BUTTON,
            BS_CHECKBOX | WS_VISIBLE

        CONTROL "CBT", 264, 97, 53, 32, 10, WC_BUTTON,
            BS_CHECKBOX | WS_VISIBLE

        CONTROL "OK", 265, 7, 20, 38, 12, WC_BUTTON,
            BS_PUSHBUTTON | WS_GROUP | WS_TABSTOP | WS_VISIBLE

        CONTROL "Cancel", 266, 61, 20, 38, 12, WC_BUTTON,
            BS_PUSHBUTTON | WS_TABSTOP | WS_VISIBLE

        CONTROL "Help", 267, 117, 20, 38, 12, WC_BUTTON,
            BS_PUSHBUTTON | WS_TABSTOP | WS_VISIBLE

    END

END
```

Dialog Editor

56 Font Editor

You can use the OS/2 Font Editor to design and save your own fonts for use in applications.

A *font* is a set of alphanumeric characters, punctuation marks, and other symbols that share a common typeface design and line weight. An application loads a font from a dynamic-link library file (.DLL file).

The Font Editor allows you to edit an enlarged version of each character in an editing window, using the mouse to switch the enlarged representation of pels to black or white.

You can change a series of pels by dragging the mouse pointer through them while holding down the mouse button. An enlarged scale version of the character is shown in a viewing window to the right of the edit window.

Using the Font Editor

To run the Font Editor, select **Font Editor** from the **PM Development Tools** folder.

Select one of the options in the **File** menu to open a new or existing font. The letter `A` appears in both the editing and viewing windows. The rest of the font appears in the character selection scroll box at the bottom of the Font Editor window.

To edit any other character in the font, select it from the character selection scroll box. The character appears in the editing and viewing windows.

Font Editing Functions

Functions for defining fonts are found on the **Header** menu.

Functions for editing character width are found on the **Width** and **Shift** menus.

Editing Fonts with Font Editor

Defining Fonts

Use the **Header** menu to define the typestyle that you want to create:

- Select **Naming** to specify the identification details such as the type-face name.
- Select **General** to specify spacing (fixed or proportional), type face style, line width, and type weight.
- Select **Sizes** to specify the font character dimensions.
- Select **Relations** to specify the position of characters.
- Select **Definition** to change character spacing in a proportional font.

Editing Character Width

The **Width** and **Shift** menus allow you to change the width of individual characters.

The **Width** Menu

Use the **Width** menu to alter the width of a single character. This menu is enabled only when you are editing a proportional space font. You can make a character wider or narrower by adding or deleting columns of pels from the right, the left, or both sides. You may also use the **Set Character Increment** option to set the width of a character. On-line help panels describe how to perform these functions.

The **Shift** Menu

Use the **Shift** menu to insert a one-pel-wide row or column into (or delete from) the character that you are editing. When you select shift, the pointer becomes a flat horizontal or vertical bar when inside the edit window. This enables you to position it exactly where you want the operation to take place.

To cancel a shift you have selected before execution, select **Cancel Choice**.

Font Resource Files

The Font Editor creates a file with a .FNT extension. The .FNT file is not referred to in the same resource file as other resources.

Instead, it has its own resource file that contains a single-line statement that has a similar format to the ICON, POINTER, and BITMAP statements, for example:

```
FONT  101      myfont.fnt     /* Font */
```

The FONT keyword identifies the resource type.

The resource type is followed by an integer identifier that is used by the application to identify the resource. The integer is used as a parameter to the WinCreateStdWindow call. You cannot use a symbolic name for a font.

The integer identifier can be followed by loading and memory options. Again, the example lets them default.

The last part of the statement is the file name of the resource created by the Font Editor. A full path name must be given if it is not in the current directory.

Producing a font file uses a process similar to binding resources to an .EXE file. You bind one or more .FNT files to a dummy .DLL, to produce a file containing the font or fonts. The final file should have the extension .FON.

The .FON file created by the process is installed on the system and becomes a *public* font, a font that can be used by any application in the system.

A font not installed on the system is called a *private* font. Before your application can use the font, your application must use GpiLoadFonts to load the .FON file.

Editing Fonts with Font Editor

57

Icon Editor

The Icon Editor lets you create your own art (icons, pointers, and bit maps) and save them for use by applications.

Icons, pointers, and bit maps produced by the Icon Editor are graphic symbols comprised of pels (also known as pixels) in any of the following display states:

- Black
- White
- Color
- Screen (background color)
- Inverse screen (inverse of background color)

An application can use an icon to represent a minimized standard window. For example, an application that lists telephone numbers could use a telephone icon when minimized. An application can also use icons as warning symbols in message boxes (for example, an exclamation mark or an upraised hand).

An application can associate a pointer with the mouse or similar pointing device, so that the user can move the pointer around the screen, to select controls or text. A pointer could also be used in an interactive graphics application to draw graphics on the screen. For example, a free-hand graphics drawing application could use a pencil shape to represent the pointer.

Using the Icon Editor

To run the Icon Editor, select the **Development Tools** folder and then select **Icon Editor**.

The Icon Editor display consists of three parts: the information panel, the palette window, and the editing window.

The information panel at the top of the Icon Editor window displays the following information:

- A picture of a two-button mouse, showing the colors currently selected for each button
- An actual-size image of the current figure that you are editing

- The status area, showing the following:
  - Size (defined as 32 x x 32 for icons and pointers; user-defined for bit maps)
  - Pen location
  - Pen size
    (from 1 x 1 to 9 x 9)
  - Hotspot (for icons and pointers, but not bit maps)
  - Figure type (icon, pointer, or bit map)
  - Form name

The palette window, in the lower right corner, displays the colors that are available for use during editing. The colors currently selected are marked with frames.

The editing window is the largest part of your working area. Use the mouse or keyboard to move the pointer, clicking or dragging the pointer to paint the enlarged representation of pels with the selected color.

Creating a Figure

The **Edit** menu includes the functions used to select an icon, pointer, or bit map for editing, and to save it after you are through.

Selecting your icon, pointer, or bit map

1. To create a new icon, pointer, or bit map, select **New** from the **File** menu. The New Figure pop-up window appears, prompting you for further information.

   Select the figure type: Icon, Pointer, or Bit map. For a bit map you must specify the width and height in pels. Select Enter.

   You can also create new art by modifying or editing an existing art of the same type.

2. To edit existing art, select **Open** from the **File** menu. You will be prompted for a name.

   Note: Unless you have turned off Safe Prompting (which is described under “Setting Preferences” on page 805) on the **Options** menu, you will be prompted to save if you select **Open** or **New** while there is unsaved art on your screen.

3. If you started Iconedit from a command prompt and specified multiple files, you can use the **Next** option on the **File** menu to select the next file.

   The Next option will be greyed out if you did not start from the command line and specify multiple files.

Saving your icon, pointer, or bit map

To save your art, select either of the following:

- Select **Save** to save it under its current file name. If this is new art, you will be prompted for a name.
- Select **Save As** to save the art under a different name. You will be prompted for a new name.

Editing Art

To edit your art, use the functions of the **Edit** menu.

Select **Undo** to restore the art to the way it was before the most recent editing operation.

Four of the editing functions require that you first mark the area to be edited, using **Select** or **Select All**.

If you choose **Select**, the cursor changes to a plus (+) inside a square. Hold the left mouse button down to anchor one corner, and then drag the mouse. Release the button to anchor the opposite corner of the rectangular area you want to edit.

If you choose **Select All**, the entire figure is selected.

Selected Edit Menu Functions

The following functions all require that an area must first be selected:

- Select **Fill** to fill the selected area with the current palette color. For additional information, 📖 see "Filling Areas With Color" on page 808.
- Select **Cut** to cut an area you would like to move or delete.
- Select **Copy** to copy an area you would like to duplicate elsewhere in the same file or in another file.
- Select **Paste** to place an area you have marked with **Cut** or **Copy**. Drag the outlined area that you have marked to the place you would like to paste it.
- Select **Clear** to erase all drawing within an area you have selected and leave transparent pels. If you have used **Select All**, this will clear your entire icon, pointer, or bit map.
- Select **Stretch Paste** to paste the clipboard contents into your art, stretching and positioning to fit.
- Select **Flip Horizontal** to flip the art on its horizontal axis, reversing bottom and top.

- Select **Flip Vertical** to flip the art on its vertical axis, reversing left and right. You can create a symmetrical drawing by copying one side of the art to the other side, and then flipping one of them.
- Select **Circle** to inscribe a circle or ellipse within the selected area.

Using Options

The choices on the **Options** menu enable you to test your art and vary your editing environment. To change an option, from the **Options** menu select:

Test
To test view the pointer or icon you are editing. The pointer or icon will be displayed, actual size, as the pointer until you toggle back by again selecting **Test** from the **Options** menu.

Grid
To superimpose a grid over the editing window. This can be useful when you want to draw a symmetrical figure. Each cell of the grid represents one pel in the figure.

X background
To make the transparent pels (where the background is visible) apparent when editing an icon or pointer. All screen or inverse screen colors will be shown with an *X*. This option does not apply to bit maps because they have no transparent pels.

Draw Straight
To temporarily restrict your drawing to drawing straight vertical and horizontal lines. Even if you deviate from the horizontal row, a horizontal line is produced when the mouse pointer is dragged across the editing window. Dragging the mouse up or down produces straight vertical lines.

Changing Pen Shape
📖 See "Changing Pen Shape and Size" on page 805

Changing Pen Size
📖 See "Changing Pen Shape and Size" on page 805

Setting Preferences
📖 See "Setting Preferences" on page 805

Defining a Hotspot
📖 See "Defining a Hotspot" on page 806

Changing Pen Shape and Size

You can change the shape and size of the pen by using choices on the **Options** menu.

Changing Pen Shape

Before you select **Pen Shape**, you must first select the shape using the **Select** function on the Edit Menu. 📖 See “Editing Art” on page 803 for information about **Select**. Then select **Set Pen Shape** on the **Options** menu.

Changing Pen Size

Select **Pen size** on the **Options** menu to specify how many pels the pointer paints at a time. You can select any of nine square pen sizes:

1 x 1
2 x 2
3 x 3
4 x 4
5 x 5
6 x 6
7 x 7
8 x 8
9 x 9

Shortcut: Select a pen size by pressing Ctrl and the size, such as Ctrl+6 for a 6 x 6 pen size.

Setting Preferences

To change your preferences, select **Preferences** from the **Options** menu. Then select any of the following:

Safe Prompting
To be warned before destructive operations such as file overwrites.

Suppress Warnings
To suppress display of informational messages.

Save State on Exit
To save settings for your next session.

Display Status Area
To toggle on and off the picture of the mouse and art from the status area.

Reset Options and Modes
To deselect the following items:

Select
Hotspot
Color Fill
Find Color

The palette will not be reset.

Defining a Hotspot

The *Hotspot* is the pel where mouse input for an icon or pointer is directed. The default hotspot location is 16 x 16, the center of the icon or pointer. Bit maps do not have hotspots.

Select **Hotspot** from the **Options** menu to designate this pel. The cursor changes shape, and the screen coordinates of the current hotspot are displayed in the information window. When you click on a new hotspot, the screen coordinates of the new hotspot are displayed.

Select **Hotspot** again to return to editing.

When an application uses WinQueryPointerPos to query the screen position of a pointer, the OS/2 operating system returns the coordinates of the pointer hot spot.

Selecting Colors

Use the **Palette** to select a new drawing color, using the left or right mouse button.

The currently selected color for the right mouse button is framed on the palette in red; the color for the left mouse button is framed in green. The currently selected colors for both mouse buttons are also displayed at the left side of the status area.

Changing Palettes or Palette Colors

To change palettes or palette colors, select the **Palette** menu. On the **Palette** menu, you can:

- Select **New** to create a new palette. The default palette will appear for you to edit.
- Select **Open** to open an existing palette.
- Select **Save** to save your current palette. If it is a new palette, you will be prompted for a name.
- Select **Save As** to save the palette under a different name. You will be prompted for a new name.
- Select **Edit Color** to edit a color in your palette.
- Select **Swap colors** to swap the colors of the left and right mouse buttons. A submenu will appear, asking whether you want to preserve these colors in your art. Unless you choose Preserve Figure, the colors in your art will be changed accordingly.
- Select **Set default palette** to save the existing palette as your default palette.

Editing Palette Colors

You can change the colors that appear on your palette. To edit palette colors, follow these steps:

1. Select the color to be edited with the mouse. A frame appears around it on the palette.
2. Select **Edit color** from the **Palette** menu.

   Shortcuts:

   - Double-click on the color to be edited.
   - To select a color that you have already used in your art, use **Find color** on the Tools menu.

   The Edit color window will appear.
3. You can change the way you define palette colors by checking **Dynamic editing** and **Important** and choosing between **RGB** and **HSV** terms.

   - **Dynamic editing**, when checked, will make your art change dynamically as you edit individual colors, so that you can see how the changes will affect your art.
   - **Important**, when checked, will require that the color be accurately rendered, without dithering (approximating the color).
   - Every color can be described numerically in either RGB or HSV terms.

     RGB As proportions of primary colors red, blue, and green

     HSV In terms of hue, saturation, and value

     To toggle between RGB and HSV, select the appropriate radio button.
4. Use the scroll bars to change RGB or HSV values, or change these numbers from the keyboard.
5. Select **OK** to save the edited color.

Filling Areas With Color

There are two different ways to fill an area with color:

- To fill an irregularly-shaped area with the current palette color, select **Color fill** from the Tools menu.

  After you click on a specific pel, all adjoining areas that are the same color as that pel will be colored with the selected color.

- To fill a previously-selected area with the current palette color, select **Fill** from the Edit menu. You must first select an area. 📖 See “Editing Art” on page 803 for information about **Select** and **Select All**.

Note: To select a color that you have already used in your art, use **Find color** on the Tools menu. A question-mark-arrow cursor will appear.

Click on a specific pel of that color, and that color is selected.

Creating Icons for Specific Displays

Although the Icon Editor edits and saves a device-independent form of the icon, the **Device** menu enables you to create versions of the icon for specific display devices. The **Device** menu displays a choice of three functions: create a new device form, select an existing form, and delete a form.

An independent form is automatically created when you create a new icon or pointer and all other forms are derived from it. If you select any of the other device forms listed in the menu, a new form is created for the specified device. The **Custom** option enables you to create an icon or pointer for any other device.

Select **List** to view a list of all existing forms, including custom and standard forms. Any item on this list can be selected and edited or deleted. However, you must have at least one device-independent form. Select **Add** in the list dialog to add a new device form.

Several icon bit maps can be saved in a single icon resource; when the icon is saved, all versions are saved with it in a format that includes a device resolution tag for each version. When the icon is loaded from a resource file, the display device resolution is matched against the device for which each device-dependent icon was intended. If a match is found, that icon is used. If no match is found, the application uses the device-independent icon, which always exists.

Figure files can contain any of the following forms to support multiple devices:

- Independent
- CGA (2 colors)
- EGA (16 colors)
- VGA (16 colors)
- XGA/8514 (256 colors)
- XGA/8514 (16 colors)
- XGA/8514 Small Color Form (16 colors)
- XGA/8514 Small BW Form
- Custom

Device-dependent icons are icons that are designed for a particular display resolution.

An application can display icons or bit maps in dialog boxes or windows.

The file name extension depends on the type of resource you are creating. The Icon Editor produces a file with any of the following extensions:

.ICO for icons
.PTR for pointers
.BMP for bit maps

The .ICO, .PTR, or .BMP files must be referred to in the resource script file for your application. The external files containing icons, pointers, and bit maps are all referenced in the resource script file by single-line statements that have a similar format. For example:

```
ICON    ID_MAINWND myprog.ico   /* Icon    */

POINTER ID_PTR     mypoint.ptr  /* Pointer */

BITMAP  ID_BMP     mybtmp.bmp   /* bit map  */
```

ICON, POINTER, and BITMAP keywords identify the resource type.

The resource type is followed by a symbolic name or integer identifier that is used by your application to identify the resource. For example, with ICON, the ID_MAINWND identifier can be used by the application in the control data parameter of the WinCreateWindow call (or as a parameter to the WinCreateStdWindow call) that creates the frame of the main window of your application. The OS/2 operating system then associates the icon with the main window.

The symbolic name or identifier can be followed by any loading and memory options. The options are not used in the example, as it lets the options default.

The last part of the statement is the file name and file type of the resource created by the Icon Editor. A fully qualified path name must be given if the file is not in the current directory. An icon that it used for a minimized application main window should have the same file name as the executable file of the application.

Using a Command Line

If you start the Icon Editor from a command line, rather than from an icon, you have an additional option available. You can load more than one file at a time by specifying the files on the command line. For instance, the following command would load the two specified icons, a bit map, and a pointer:

```
ICONEDIT Ruth.ico gurp.ico alex.bmp pamela.ptr
```

If you specify multiple files when you start the Icon Editor from the command line, you can use the **Next** option on the File Menu to select the next file. This option is available *only* if you specify multiple files from the command line.

Part 12. Additional Utilities You May Find Useful

VisualAge C++ provides a number of utilities that can help you complete your applications and make programming tasks easier:

NMAKE	Builds your application based on dependencies and rules.
MKMSGF	Creates message files.
MSGBIND	Binds messages to your application.
KwikINF	Provides quick online information.
IPFC	Creates online documentation.
PACK and PACK2	Compress files.
CPPFILT	Demangles compiled C++ names.
EXEHDR	Displays and modifies executable-file header contents.
MARKEXE	Displays and modifies program type for executable files.
MAPSYM	Creates symbolic debugging files (for kernel debugger) from map files.
Workplace Class List	Creates and modifies Workplace object classes.
Object Utility/2	Registers and instantiates Workplace object classes.
T Terminal Emulator	Provides ASCII terminal emulation.

Program Maintenance Utility (NMAKE)

The Program Maintenance Utility (NMAKE) automates the process of updating project files. NMAKE compares the modification dates for one set of files (the target files) with those of another set of files (the dependent files). If any dependent files have changed more recently than the target files, NMAKE executes a series of commands to bring the targets up-to-date.

Why Use NMAKE?

The most common use of NMAKE is to automate the process of updating a project after you make a change to a source file. Large projects tend to have many source files. Often, only a few of your source files need to be compiled when you make a change. You set up a special text file called a “description” file (or “makefile”) that tells NMAKE:

- Which files depend on others
- Which commands, such as compile and link commands, need to be carried out to bring your program up-to-date

This use of NMAKE is only one example of its power. By building suitable description files, you can use NMAKE to

- Make backups
- Configure data files
- Run programs when data files are modified

Running NMAKE

Run NMAKE by typing `NMAKE` on the operating-system command line. Supply input to NMAKE by either of two methods:

- Enter the input directly on the command line.
- Put your input into a *command file* (a text file, also called a *response file*) and enter the file name on the command line.

Press CTRL+C at any time during an NMAKE run to return to the operating system.

Note: Under the OS/2 operating system, do not use the ampersand character (&) to combine the NMAKE command with the CD, CHDIR, or SET command.

Using the Command Line

When using NMAKE at the command line, keep the following in mind:

- All fields are optional.
- NMAKE always looks first in the current directory for a description file called MAKEFILE. If MAKEFILE does not exist, NMAKE uses the <filename> given with the /F (specify description file) option.

Command-Line Syntax

```
NMAKE [options] [macrodefinitions] [targets] [/F filname]
```

<options>
Specifies options that modify NMAKE's actions.

<macrodefinitions>
Lists macro definitions for NMAKE to use. Macro definitions that contain spaces must be enclosed by double quotation marks.

<targets>
Specifies the names of one or more target files to build. If you do not list any targets, NMAKE builds the first target in the description file.

/F <filename>
Gives the name of the description file where you specify file dependencies and which commands to execute when a file is out-of-date.

The following example:

```
NMAKE /S "program = flash" SORT.EXE SEARCH.EXE
```

- Invokes NMAKE with the /S option
- Defines a macro, assigning the string "flash" to the macro "program"
- Specifies two targets: SORT.EXE and SEARCH.EXE

By default, NMAKE uses the file named MAKEFILE as the description file.

Command-Line Help

To display NMAKE help, type `NMAKE /?` at the prompt. The appropriate copyright statement appears, along with the following:

```
Usage:
        NMAKE @commandfile
        NMAKE /help

        NMAKE [/nologo] [/acdeinpqrst?] [/f makefile] [/x stderrfile]

        [macrodefs][targets]

Where the options stand for
  /a      force All targets to be built
  /c      Cryptic mode; suppress sign-on banner & warning messages
  /d      Display modification dates
  /e      Environment variables override macros in the makefile
  /i      Ignore exit codes of commands invoked
  /n      No execute mode; display commands only
  /p      Print macro definitions & target descriptions
  /q      Query if target is up to date; for use in batch files
  /r      inference Rules from 'tools.ini' to be ignored
  /s      Silent execution of commands
  /t      Touch targets with current date & time
  /?      Help message
  /help   Help message
  /nologo do not display sign-on banner
```

Using NMAKE Command Files

A command file is a *response file* used to extend command-line input to NMAKE.

You can split input to NMAKE between the command line and a command file. Use the name of a command file (preceded by @) where you normally type the input information on the command line.

Why Use a Command File?

Use a command file for

- Complex and long commands you type frequently
- Strings of command-line arguments, such as macro definitions, that exceed the limit for command-line length

Note: A command file is not the same as a description file. For information about description files, see “Description Files” on page 821

Command File Syntax

To provide input to NMAKE with a command file, type

```
NMAKE @commandfile
```

In the <commandfile> field, enter the name of a file containing the same information as is normally entered on the command line.

NMAKE treats line breaks that occur between arguments as spaces. Macro definitions can span multiple lines if you end each line except the last with a backslash (\). Macro definitions that contain spaces must be enclosed by quotation marks, just as if they were entered directly on the command line.

Example

The following is a command file called UPDATE:

```
/S "program \
= flash" SORT.EXE SEARCH.EXE
```

You can use this command file by typing the following command:

```
NMAKE @UPDATE
```

This runs NMAKE using:

- The /S option
- The macro definition "program = flash"
- The targets specified as SORT.EXE and SEARCH.EXE
- The description file MAKEFILE by default

Note that the backslash allows the macro definition to span two lines.

Options

The following describes the options you can use with NMAKE. Keep the following in mind when using options:

- Option characters are not case sensitive; /I and /i are equivalent.
- You can use either a slash or dash before the option characters; -a and /a are equivalent.

Produce Error File (/X)

Syntax: `/X stderrfile`

This option produces a standard error file.

Build All Targets (/A)

Syntax: `/A`

This option builds all specified targets even if they are not out-of-date with respect to their dependent files.

📖 See “Description Files” on page 821.

Suppress Messages (/C)

Syntax: `/C`

This option suppresses display of the NMAKE sign-on banner, nonfatal error messages, and warning messages. To suppress the sign-on banner without suppressing other messages, use the /NOLOGO option.

Display Modification Dates (/D)

Syntax: `/D`

This option displays the modification date of each file when the dates of target and dependent files are checked.

📖 See “Description Files” on page 821.

Override Environment Variables (/E)

Syntax: `/E`

This option disables inherited macro redefinition.

NMAKE *inherits* all current environment variables as macros, which can be redefined in a description file. The /E option disables any redefinition — the inherited macro always has the value of the environment variable.

Specify Description File (/F)

Syntax: `/F filename`

This option specifies <filename> as the name of the description file to use. If a dash (-) is entered instead of a file name, NMAKE reads a description file from the standard input device, typically the keyboard.

If a filename is not specified, it defaults to MAKEFILE.

Display Help (/HELP or /?)

Syntax: `/HELP` OR `/?`

This option displays a brief summary of NMAKE syntax.

Ignore Exit Codes (/I)

Syntax: `/I`

This option ignores exit codes (also called error level or return codes) returned by programs such as compilers or linkers called by NMAKE. If this option is not specified, NMAKE ends when any program returns a nonzero exit code.

Display Commands (/N)

Syntax: `/N`

This option causes NMAKE commands to be displayed but not executed. Use the /N option to:

- Check which targets are out-of-date with respect to their dependents
- Debug description files

Suppress Sign-On Banner (/NOLOGO)

Syntax: `/NOLOGO`

This option suppresses the sign-on banner display when NMAKE is started. If you want to suppress nonfatal error messages and warnings as well, use the suppress messages (/C) option.

Print Macro and Target Definitions (/P)

Syntax: `/P`

This option writes out all macro definitions and target definitions. Output is sent to the standard output device (typically the display).

Return Exit Code (/Q)

Syntax: `/Q`

This option causes NMAKE to return either of the following:

- A 0 exit code if all targets built during-an-NMAKE run are up-to-date
- A nonzero exit code if they are not up-to-date

Use this option to run NMAKE from within a batch file.

Ignore TOOLS.INI File (/R)

Syntax: `/R`

This option ignores the following:

- All inference rules and macros contained in the TOOLS.INI file
- All predefined inference rules and macros

Suppress Command Display (/S)

Syntax: `/S`

This option suppresses the display of commands as they are executed by NMAKE. It does not suppress the display of messages generated by the commands themselves.

The /N command (Display Commands) takes precedence over the /S option. If you use /N and /S together, commands are displayed but not executed.

Change Target Modification Dates (/T)

Syntax: `/T`

This option changes or "touches" the modification dates for out-of-date target files to the current date. No commands are executed, and the target file is left unchanged.

Description Files

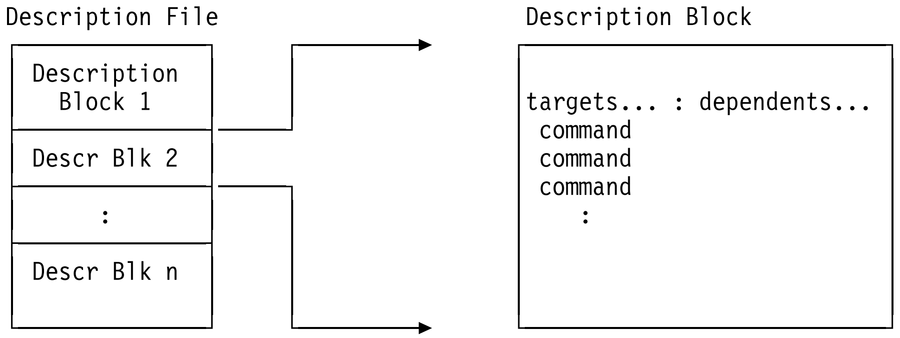
NMAKE uses a description file to determine what to do. In its simplest form, a description file tells NMAKE which files depend on others and which commands need to be executed if a file changes.

A description file looks like this:

```
targets...: dependents...
     command
        :

targets... : dependents...
     command
```

Description Blocks

A dependent relationship between files is defined in a *description block*. A description block indicates the relationship among various parts of the program. It contains commands to bring all components up to date. The description file can contain up to 1048 description blocks.



Special Features

The following are special features of description files and blocks:

- Description files can contain macro definitions and use macros in description blocks. Macros allow easy substitution of one text string for another.
- Description files can contain inference rules. Inference rules allow NMAKE to infer which commands to execute based on the file-name extensions used for targets and dependents.
- You can specify directories for NMAKE to search for dependent files by using the following syntax:

  ```
  targets : {directory1;directory2...}dependents
  ```

  NMAKE searches the current directory first, then <directory1>, <directory2>, and so on.
- A command can be placed on the same line as the target and dependent files by using a semicolon (;) as depicted below:

  ```
  targets... : dependents... ; command
  ```
- A long command can span several lines if each line ends with a backslash (\ ):

  ```
  command \
    continuation of command
  ```
- The execution of a command can be modified if you precede the command with special characters.
- If you do not specify a command in a description block, NMAKE looks for an inference rule to build the target.

- DOS and OS/2 wild card characters (* and ?) can be used in description blocks. For example, the following description block compiles all source files with the .C extension:

  ```
  ASTRO.EXE : *.C
    ICC $**
  ```

- NMAKE will expand the *.C specification into the complete list of C files in the current directory. $** is a complete list of dependents specified for the current target.
- NMAKE uses several punctuation characters in its syntax. To use one of these characters as a literal character, place an escape character (^) in front of it. For a list of punctuation characters, see "Escape Characters" on page 839.
- Normally a target file can appear in only one description block. A special syntax allows you to use a target in several description blocks.
- A special syntax allows you to determine the drive, path, base name, and extension of the first dependent file in a description block.

Targets in Several Description Blocks

Using a file as a target in more than one description block causes NMAKE to end. You can overcome this limitation by using two colons (::) as the target/dependent separator instead of one colon.

The following description block is permissible:

```
X :: A
  command
X :: B
  command
```

The following causes NMAKE to end:

```
X : A
  command
X : B
  command
```

It is permissible to use single colons if the target/dependent lines are grouped above the same commands. The following is permissible:

```
X : A
X : B
  command
```

Double Colon (::) Target/Dependent Separator Example

```
TARGET.LIB :: A.ASM B.ASM C.ASM
  ML A.ASM B.ASM C.ASM
  LIB TARGET -+A.OBJ -+B.OBJ -+C.OBJ;

TARGET.LIB :: D.C E.C
  ICC /C D.C E.C
  LIB TARGET -+D.OBJ -+E.OBJ;
```

These two description blocks both update the library named TARGET.LIB. If any of the assembly-language files have changed more recently than the library file, NMAKE executes the commands in the first block to assemble the source files and update the library. Similarly, if any of the C-language files have changed, NMAKE executes the second group of commands to compile the C files and update the library.

Macros

Macros provide a convenient way to replace one string with another in the description file. The text is automatically replaced each time NMAKE is run. This feature makes it easy to change text throughout the description file without having to edit every line that uses the text. Two common uses of macros are:

Two Common Uses of Macros

- To create a standard description file for several projects. The macro represents the file names in commands. These file names are defined when you run NMAKE. When you switch to a different project, changing the macro changes the file names NMAKE uses throughout the description file.
- To control the options that NMAKE passes to the compiler, assembler, or linker. When using a macro to specify the options, you can quickly change the options throughout the description file in one easy step.

A macro can be defined :

In a Description File
On the Command Line
In TOOLS.INI
Through inheritance from Environment Variables

Macros Example

```
program = FLASH
c = LINK
options =

$(program).EXE : $(program).OBJ
  $c $(options) $(program).OBJ;
```

The example above defines three macros. The description block executes the following commands:

```
FLASH.EXE : FLASH.OBJ
  LINK  FLASH.OBJ;
```

Special Features

Macros have the following special features:

- When using a macro, you can substitute text in the macro itself.
- Several macros have been predefined for special purposes.
- If a macro is defined more than once, precedence rules govern which definition is used.
- You can also put macros into your TOOLS.INI file.

Macros in a Description File

Before using a macro, you need to define it, either on the NMAKE command line or in your description file. Description file macro definitions look like this:

```
macroname = macrostring
```

Macro names can be any combination of alphanumeric characters and the underscore character (_), and they are case-sensitive. A macro string can be any string of characters.

The first character of the macro name must be the first character on the line. NMAKE ignores any spaces before or after the equal sign (=).

The macro string can be a null string and can contain embedded spaces. Do not enclose the macro string in quotation marks; quotation marks are used only when you define macros on the command line.

Macros on the Command Line

Before using a macro, you need to define it, either on the NMAKE command line or in your description file. Command-line macro definitions look like this:

```
macroname=macrostring
```

No spaces can surround the equal sign. If you embed spaces, NMAKE might misinterpret your macro. If your macro string contains embedded spaces, enclose it in double quotation marks (") like this:

```
macroname="macro string"
```

or simply enclose the entire macro definition in double quotation marks (") like this:

```
"macroname = macro string"
```

Macro names can be any combination of alphanumeric characters and the underscore character (_), and they are case-sensitive. A macro string can be any string of characters or a null string.

Inherited Macros

NMAKE *inherits* all current environment variables as macros. For example, if you have a PATH environment variable defined as `PATH = C:\TOOLS\BIN`, the string `C:\TOOLS\BIN` is substituted when you use PATH in the description file.

You can redefine inherited macros by including a line such as the example above in a description file. While NMAKE is executing, the macro takes on the redefined definition. When NMAKE terminates, however, the environment variable resumes its original value.

The Override Environment Variables (/E) option disables inherited macro redefinition. If you use this option, NMAKE ignores any attempt to redefine an inherited macro.

Defined Macros

After you have defined a macro, you can use it anywhere in your description file with the following syntax:

```
$(macroname)
```

The parentheses are not required if the macro name is only one character long. To use a dollar sign ($) without using a macro, enter two dollar signs ($$), or use the caret (^) before the dollar sign as an escape character.

When NMAKE runs, it replaces all occurrences of $(macroname) with the defined macro string. If the macro is undefined, nothing is substituted. After a macro is defined, you can cancel it only with the !UNDEF directive.

Macro Substitutions

Just as you use macros to substitute text within a description file, you use the following syntax to substitute text within a macro:

```
$(macroname: string1 = string2)
```

Every occurrence of <string1> is replaced by <string2> in <macroname>. Spaces between the colon and <string1> are considered part of <string1>. If <string2> is a null string, all occurrences of <string1> are deleted from the macro. The colon (:) must immediately follow <macroname>.

Note: The replacement of <string1> with <string2> in the macro is not a permanent change. If you use the macro again without a substitution, you get the original unchanged macro.

Example

```
SOURCES = ONE.C TWO.C THREE.C
PROGRAM.EXE : $(SOURCES:.C=.OBJ)
  LINK $**;
```

The example above defines a macro called SOURCES, which contains the names of three C source files. With this macro, the target/dependent line substitutes the .OBJ extension for the .C extension. Thus, NMAKE executes the following command:

```
LINK ONE.OBJ TWO.OBJ THREE.OBJ;
```

Note: $** is a special macro that translates to all dependent files for a given target.

Special Macros

NMAKE predefines several macros. The first six macros below return one or more file specifications for the files in the target/dependent line of a description block. Except where noted, the file specification includes the path of the file, the base file name, and the file-name extension.

Macro	Value
$@	The specification of the target file.
$*	The base name (without extension) of the target file. Path information is also returned if the path was specified as part of the target file name. This macro cannot be used in a dependent list.
$**	The specifications of the dependent files.
$?	The specifications for only those dependent files that are out-of-date with respect to the targets.
$<	The specification of a single dependent file that is out-of-date with respect to the targets. This macro is used only in inference rules.
$$@	The file specification of the target that NMAKE is currently evaluating. This is a dynamic dependency parameter, used only in dependent lists.

$(CC) The string ICC, which is the command to run the C Set ++ Compiler. You can redefine this macro to use a different command.

$(AS) The string MASM, which is the command to run the Macro Assembler (MASM). You can redefine this macro to use a different command.

$(MAKE) The command name used to run NMAKE. This macro is used to invoke NMAKE recursively. If you redefine this macro, NMAKE issues a warning message.

Note: NMAKE executes the command line in which $(MAKE) appears, even if the display commands (/N) option is on.

$(MAKEFLAGS)
The NMAKE options currently in effect. You cannot redefine this macro.

Note: The special macros $** and $$@ are the only exceptions to the rule that macro names longer than one character must be enclosed in parentheses.

You can append characters to any of the first six macros in this list to modify the meaning of the macro. However, you cannot use macro substitutions in these macros.

Special Macros Examples

```
TRIG.LIB : SIN.OBJ COS.OBJ ARCTAN.OBJ
  !LIB TRIG.LIB -+$?;
```

In the example above, the macro $? represents the names of all dependent files that are out-of-date with respect to the target file. The exclamation point (!) preceding the LIB command causes NMAKE to execute the LIB command once for each dependent file in the list. As a result of this description, the LIB command is executed up to three times, each time replacing a module with a newer version.

```
DIR=C:\INCLUDE
$(DIR)\GLOBALS.H : GLOBALS.H
 COPY GLOBALS.H $@
$(DIR)\TYPES.H : TYPES.H
 COPY TYPES.H $@
$(DIR)\MACROS.H : MACROS.H
 COPY MACROS.H $@
```

The example above shows how to update a group of include files. Each of the files GLOBALS.H, TYPES.H, and MACROS.H in the directory C:\INCLUDE depends on its counterpart in the current directory. If one of the include files is out-of-date, NMAKE replaces it with the file of the same name from the current directory.

The following description file, which uses the special macro $$@, is equivalent:

```
DIR=C:\INCLUDE
$(DIR)\GLOBALS.H $(DIR)\TYPES.H $(DIR)\MACROS.H : $$(@F)
!COPY $? $@
```

The special macro $$(@F) signifies the file name (without the path) of the current target.

When NMAKE evaluates the description block, it evaluates the three targets, one at a time, with respect to their dependents. Thus, NMAKE first checks whether C:\INCLUDE\GLOBALS.H is out-of-date compared with GLOBALS.H in the current directory. If so, it executes the command to copy the dependent file GLOBALS.H to the target. NMAKE repeats the procedure for the other two targets.

Note that on the command line, the macro $? refers to the dependent for this target. The macro $@ specifies the full file specification of the target file.

File-Specification Parts

A full file specification gives the base name of the file, the file-name extension, and the path. The path provides the disk-drive identifier and the sequence of directories needed to locate the file on the disk.

For example, the file specification

```
C:\SOURCE\PROG\SORT.OBJ
```

has the following parts:

Path Name	C:\SOURCE\PROG
Base File Name	SORT
File-Name Extension	.OBJ

Characters That Modify Special Macros

The following six macros all resolve to a file specification (or possibly several file specifications for $** and $?):

```
    $*    $@    $**    $<    $?    $$@
```

You can append characters to any of these macros to modify the file name returned by the macro. Depending on which character you use, parts of the full file specification are returned:

	Appended Character			
File Part Returned	D	F	B	R
File Path	Yes	No	No	Yes
Base File Name	No	Yes	Yes	Yes
File Name Extension	No	Yes	No	No

Modified Special Macros Example

If the macro $@ has the value

```
C:\SOURCE\PROG\SORT.OBJ
```

then the following values are returned for the modified macro:

Macro	Value
$(@D)	C:\SOURCE\PROG
$(@F)	SORT.OBJ
$(@B)	SORT
$(@R)	C:\SOURCE\PROG\SORT

Note: Modified macros are always longer than a single character — they must be enclosed by parentheses when used.

Macro Precedence Rules

When the same macro is defined in more than one place, the definition with the highest priority is used:

Priority	Definition
1 (Highest)	Command line
2	Description file
3	Environment variables
4	TOOLS.INI file
5 (Lowest)	Predefined macros (such as CC and AS)

If you invoke NMAKE with the Overriding Macro Definitions (/E) option, macros defined by environment variables take precedence over those defined in a description file.

Inference Rules

Inference rules are templates from which NMAKE infers what to do with a description block when no commands are given. Only those extensions defined in a .SUFFIXES list can have inference rules. The extensions .C, .OBJ, .ASM, and .EXE are automatically included in .SUFFIXES.

When NMAKE encounters a description block with no commands, it looks for an inference rule that specifies how to create the target from the dependent files, given the two file extensions. Similarly, if a dependent file does not exist, NMAKE looks for an inference rule that specifies how to create the dependent from another file with the same base name.

NMAKE applies an inference rule only if the base name of the file it is trying to create matches the base name of a file that already exists.

In effect, inference rules are useful only when there is a one-to-one correspondence between the files with the "from" extension and the files with the "to" extension. You cannot, for example, define an inference rule that inserts a number of modules into a library.

The use of inference rules eliminates the need to put the same commands in several description blocks. For example, you can use inference rules to specify a single ICC command that changes any C source file (with a .C extension) to an object file (with a .OBJ extension).

You define an inference rule by including text of the following form in your description file or in your TOOLS.INI file — see “Special Features”.

```
.fromext.toext:
commands
:
```

The elements of the inference rule are:

<fromext>
The file-name extension for dependent files to build a target

<toext>
The file-name extension for target files to be built

<commands>
The commands to build the <toext> target from the <fromext> dependent.

For example, an inference rule to convert C source files (with the .C extension) to C object files (with the .OBJ extension) is

```
.C.OBJ:
 ICC $<
```

Note: The special macro $< represents the name of a dependent out-of-date relative to the target.

Special Features

- You can specify a path where NMAKE should look for target and dependent files used in inference rules.
- Inference rules are predefined for compiling and linking C programs, and for assembling programs.
- NMAKE looks for inference rules in the TOOLS.INI file if it cannot find a rule in a description file.
- Only those extensions defined in a .SUFFIXES list can have inference rules. The extensions .C, .OBJ, .ASM, and .EXE are automatically included in .SUFFIXES.

Inference Rules Example

```
.OBJ.EXE:
  LINK $<;

EXAMPLE1.EXE: EXAMPLE1.OBJ

EXAMPLE2.EXE: EXAMPLE2.OBJ
  LINK /CO EXAMPLE2,,,LIBV3.LIB
```

The first line above defines an inference rule that causes the LINK command to create an executable file whenever a change is made in the corresponding object file. The file name in the inference rule is specified with the special macro $< so that the rule applies to any .OBJ file with an out-of-date executable file.

When NMAKE does not find any commands in the first description block, it checks for a rule that might apply and finds the rule defined on the first two lines of the description file. NMAKE applies the rule, replacing $< with EXAMPLE1.OBJ when it executes the command, so that the LINK command becomes

```
LINK EXAMPLE1.OBJ;
```

NMAKE does not search for an inference rule when examining the second description block, because a command is explicitly given.

Inference-Rule Path Specifications

When defining an inference rule, you can indicate to NMAKE where to look for target and dependent files. Use the following syntax:

```
{frompath}.fromext{topath}.toext
 commands
 :
```

NMAKE looks in the directory specified by <frompath> for files with the <fromext> extension. It executes the commands to build files with the <toext> extension in the directory specified by <topath>.

Predefined Inference Rules

NMAKE predefines three inference rules:

Inference Rule	Default	Command Action
.C.OBJ	$(CC) $(CFLAGS) /C $*.C	ICC /C $*.C
.C.EXE	$(CC) $(CFLAGS) $*.C	ICC $*.C
.ASM.OBJ	$(AS) $(AFLAGS) $*;	MASM $*;

Notes:

1. The first two rules automatically compile and link C programs.
2. The last rule automatically assembles programs.

Directives

Using directives, you can construct description files similar to batch files. NMAKE provides directives that:

- Conditionally execute commands
- Display error messages
- Include the contents of other files
- Turn some NMAKE options on or off

Each directive begins with an exclamation point (!) in the first column of the description file. Spaces can be placed between the exclamation point and the directive keyword.

The list below describes the directives:

!IF <expression>

Executes the statements between the !IF keyword and the next !ELSE or !ENDIF directive if <expression> evaluates to a nonzero value.

The <expression> used with the !IF directive can consist of integer constants, string constants, or exit codes returned by programs. Integer constants can use the C unary operators for numerical negation (-), one's complement (˜), and logical negation (!). You can also use any of the C binary operators listed below:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^^	Bitwise XOR
&&	Logical AND
||	Logical OR
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Notes:

1. You can use parentheses to group expressions.
2. Values are assumed to be decimal values unless specified with a leading 0 (octal) or leading 0x (hexadecimal).
3. Strings are enclosed by quotation marks ("). You can use the equality (==) and inequality (!=) operators to compare two strings.

4. You can invoke a program in an expression by enclosing the program name in square brackets ([ ]). The exit code returned by the program is used in the expression.

!ELSE

Executes the statements between the !ELSE and !ENDIF directives if the statements preceding the !ELSE directive were not executed.

!ENDIF

Marks the end of the !IF, !IFDEF, or !IFNDEF block of statements.

!IFDEF <macroname>

Executes the statements between the !IFDEF keyword and the next !ELSE or !ENDIF directive if <macroname> is defined in the description file. If a macro has been defined as null, it is still considered to be defined.

!IFNDEF <macroname>

Executes the statements between the !IFNDEF keyword and the next !ELSE or !ENDIF directive if <macroname> is not defined in the description file.

!UNDEF <macroname>

Undefines a previously defined macro.

!ERROR <text>

Prints text and then stops execution.

!INCLUDE <filename>

Reads and evaluates the file <filename> before continuing with the current description file. If <filename> is enclosed by angle brackets (<>), NMAKE searches for the file in the directories specified by the INCLUDE macro; otherwise, it looks only in the current directory. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.

!CMDSWITCHES {+|-}<opt>

Turns on or off one of four NMAKE options: /D, /I, /N, and /S. If no options are specified, the options are reset to the values they had when NMAKE was started. To turn an option on, precede it with a plus sign (+); to turn it off, precede it with a minus sign (-). This directive updates the MAKEFLAGS macro.

📖 See “Special Macros” on page 827.

Directives Example

```
!INCLUDE <INFRULES.TXT>
!CMDSWITCHES +D
WINNER.EXE:WINNER.OBJ
!IFDEF DEBUG
! IF "$(DEBUG)"=="y"
     LINK /CO WINNER.OBJ;
! ELSE
     LINK WINNER.OBJ;
! ENDIF
!ELSE
! ERROR Macro named DEBUG is not defined.
!ENDIF
```

The directives in this example do the following:

- The !INCLUDE directive causes the file INFRULES.TXT to be read and evaluated as if it were part of the description file.
- The !CMDSWITCHES directive turns on the /D option, which displays the dates of the files as they are checked.
- If WINNER.EXE is out-of-date with respect to WINNER.OBJ, the !IFDEF directive checks to see whether the macro DEBUG is defined. If it is defined, the !IF directive checks to see whether it is set to y. If it is, the linker is invoked with the /CO option; otherwise, it is invoked without the /CO. If the DEBUG macro is not defined, the !ERROR directive prints the message and NMAKE stops executing.

Pseudotargets

A "pseudotarget" is a target in a description block that is not a file. Instead, it is a name that serves as a "handle" for building a group of files or executing a group of commands. In the following example, UPDATE is a pseudotarget:

```
UPDATE: *.*
  !copy $** A:\PRODUCT
```

When NMAKE evaluates a pseudotarget, it always considers the dependents to be out-of-date. In the description above, NMAKE copies each of the dependent files to the specified drive and directory.

NMAKE predefines several pseudotargets for special purposes.

See “Predefined Pseudotargets” on page 837.

Predefined Pseudotargets

NMAKE predefines several pseudotargets that provide special rules within a description file:

.SILENT Pseudotarget

Syntax: .SILENT : dependents...

This pseudotarget suppresses the display of executed commands for a single description block. The /S option does the same thing for all description blocks.

📖 See “Suppress Command Display (/S)” on page 821.

.IGNORE Pseudotarget

Syntax: .IGNORE : dependents...

This pseudotarget ignores exit codes returned by programs for a single description block. The /I option does the same thing for all description blocks.

📖 See “Ignore Exit Codes (/I)” on page 820.

.SUFFIXES Pseudotarget

Syntax: .SUFFIXES : extensions...

This pseudotarget defines file extensions to try when NMAKE needs to build a target file for which no dependents are specified. NMAKE searches the current directory for a file with the same name as the target file and an extension in <extensions...>. If NMAKE finds such a file, and if an inference rule applies to the file, NMAKE treats the file as a dependent of the target.

The .SUFFIXES pseudotarget is predefined as

```
.SUFFIXES : .OBJ .EXE .C .ASM
```

To add extensions to the list, specify .SUFFIXES : followed by the new extensions. To clear the list, specify

```
.SUFFIXES:
```

Note: Only those extensions specified in .SUFFIXES can have inference rules. NMAKE ignores inference rules unless the extensions have been specified in a .SUFFIXES list.

.PRECIOUS Pseudotarget

Syntax: .PRECIOUS : targets...

This pseudotarget tells NMAKE not to delete a target even if the commands that build it are terminated or interrupted. This pseudotarget overrides the NMAKE default. By default, NMAKE deletes the target if it cannot be sure that the target was built successfully.

For example,

```
.PRECIOUS : TOOLS.LIB
TOOLS.LIB : A2Z.OBJ Z2A.OBJ
 command
   :
```

If the commands to build TOOLS.LIB are interrupted, leaving an incomplete file, NMAKE does not delete the partially built TOOLS.LIB.

Note: The pseudotarget .PRECIOUS is useful only in limited circumstances. Most professional development tools have their own interrupt handlers and "clean up" when errors occur.

Inline Files

You may need to issue a command in the description file with a list of arguments exceeding the command-line limit of the operating system. Just as NMAKE supports the use of command files, it can also generate inline files which are read as response files by other programs.

To generate an inline file, use the following syntax for your description block:

```
target : dependents
  command @<<[filename]
inline file text
<< [KEEP | NOKEEP]
```

All of the text between the two sets of double less than signs (<<) is placed into an inline file and given the name <filename>. You can refer to the inline file at a later time by using <filename>. If <filename> is not given, NMAKE gives the file a unique name in the directory specified by the TMP environment variable if it is defined. Otherwise, NMAKE creates a unique file name in the current directory.

The inline file can be temporary or permanent. If you do not specify otherwise, or if you specify the keyword NOKEEP, the inline file is temporary. Specify KEEP to retain the file.

Note: The at sign (@) is not part of the NMAKE syntax but is the typical character used by utilities (such as LINK386) to designate a file as a response file.

In-Line Files Example

```
MATH.LIB :  ADD.OBJ SUB.OBJ MUL.OBJ DIV.OBJ
  LIB @<<
MATH.LIB
-+ADD.OBJ-+SUB.OBJ-+MUL.OBJ-+DIV.OBJ
listing
<<
```

The above example creates an inline file and uses it to invoke the Library Manager (LIB). The inline file is used as a response file by (LIB). It specifies which library to use, the commands to execute, and the listing file to produce. The inline file contains the following:

```
MATH.LIB
-+ADD.OBJ-+SUB.OBJ-+MUL.OBJ-+DIV.OBJ
listing
```

Because no file name is listed after the LIB command, the inline file is given a unique name and placed into the current directory (or the directory defined by the TMP environment variable).

Escape Characters

NMAKE uses the following punctuation characters in its syntax:

(	)	#	$	^	\
{	}	!	@	-	

To use one of these characters in a command and not have it interpreted by NMAKE, use a caret (^) in front of the character.

For example,

```
BIG^#.C
```

is treated as

```
BIG#.C
```

With the caret, you can include a literal newline character in a description file. This capability is useful in macro definitions, as in the following example:

```
XYZ=abc^<ENTER>
def
```

The effect is equivalent to the effect of assigning the C-style string abc\ndef to the XYZ macro. Note that this effect differs from the effect of using the backslash (\ )

to continue a line. A newline character that follows a backslash is replaced with a space.

NMAKE ignores a caret that is not followed by any of the characters it uses in its syntax. A caret that appears within quotation marks is not treated as an escape character.

Note: The escape character cannot be used in the command portion of a dependency block.

Characters That Modify Commands

Any of three characters can be placed in front of a command to modify how the command is run:

— (dash) Turns off error checking for the command

@ (at sign) Suppresses display of the command

! (exclamation point)
Executes the command for each dependent file

NOTE

1. Spaces can separate the modifying character from the command. Any command on a separate line — whether modified or not — must be indented by one or more spaces or tabs.
2. You can use more than one character to modify a single command.

Turn Error Checking Off (-)

Syntax: -[n] command

The /I option globally turns command error-checking off. The dash (-) command modifier overrides the global setting to turn error checking off for commands individually. This modifier is used in two ways:

- A dash without a number turns off all error checking.
- A dash followed by a number causes NMAKE to abort only if the exit code returned by the command is greater than the number.

See “Ignore Exit Codes (/I)” on page 820.

Dash Command Modifier Examples

```
LIGHT.LST : LIGHT.TXT
  - FLASH LIGHT.TXT
```

In the example above, NMAKE never ends, regardless of the exit code returned by FLASH.

```
LIGHT.LST : LIGHT.TXT
  -1 FLASH LIGHT.TXT
```

In the example above, NMAKE ends if the exit code returned by FLASH is greater than 1.

Suppress Command Display (@)

Syntax: @ command

The /S option globally suppresses the display of commands while NMAKE is running. The at sign (@) modifier suppresses the display for individual commands.

Note: Regardless of the /S option or the @ modifier, output generated by the command itself always appears.

See “Suppress Command Display (/S)” on page 821.

At Sign (@) Command Modifier Example

Suppress Command Display (@)

```
SORT.EXE:SORT.OBJ
  @ ECHO sorting
```

The command line calling the ECHO command is not displayed. The output of the ECHO command, however, is displayed.

Execute Command for Dependents (!)

Syntax: ! command

The exclamation-point command modifier causes the command to be executed for each dependent file if the command uses one of the special macros $? or $**. The $? macro refers to all dependent files out-of-date with respect to the target. The $** macro refers to all dependent files in the description block.

See “Special Macros” on page 827.

Exclamation Point (!) Command Modifier Examples

```
LEAP.TXT : HOP.ASM SKIP.BAS JUMP.C
  ! print $** lpt1:
```

The example above executes the following three commands, regardless of the modification dates of the dependent file:

```
print HOP.ASM lpt1:
print SKIP.BAS lpt1:
print JUMP.C lpt1:

LEAP.TXT : HOP.ASM SKIP.BAS JUMP.C
  ! print $? lpt1:
```

The example above executes the print command only for those dependent files with modification dates later than that of the LEAP.TXT file. If HOP.ASM and JUMP.C have modification dates later than LEAP.TXT, the following two commands are executed:

```
print HOP.ASM lpt1:
print JUMP.C lpt1:
```

EXTMAKE Syntax

Description files can use a special syntax to determine the drive, path, base name, and extension of the first dependent file in a description block. This syntax is called the *extmake* syntax.

The characters `%s` represent the complete file specification of the first dependent file. Various parts of the file specification are represented using the syntax

`%|partsF`

where <parts> is a combination of the following letters:

d Drive
p Path
f Base name
e Extension

For example, to specify the drive and path name of the first dependent file in a description block, use:

%|dpF

The percent symbol (%) is a replacement in DOS and OS/2 command lines. To use the percent symbol in command-line arguments, use a double percent (%%).

Macros and Inference Rules in TOOLS.INI

You can place either macros or inference rules in your TOOLS.INI file. NMAKE looks for the TOOLS.INI file first in the current directory and then in the directory indicated by the INIT environment variable.

If NMAKE finds a TOOLS.INI file, it looks for the following tag:

```
[nmake]
```

You can place macros and inference rules below this tag in the same format you would use in a description file.

If a macro or inference rule is defined in both the TOOLS.INI file and the description file, the definition in the description file takes precedence. Also, if you use the /R option, the TOOLS.INI file is ignored.

TOOLS.INI Example

```
[nmake]
CFLAGS=/ss /ms /Gd-
.C.OBJ:
  $(CC) -c $(CFLAGS) $*.C
```

These lines in the TOOLS.INI file do the following:

- Define the CFLAGS macro as "/ss /ms /Gd-"
- Redefine the predefined inference rule to build .OBJ files from .C source files

Program Maintenance Utility (NMAKE)

59

Creating Message Files with MKMSGF

The MKMSGF program reads the input message file that you specify and creates an output message file that DosGetMessage uses to display messages.

There are two ways that the output message file can be used:

- Selected messages can be bound to the message segment of an executable file using the MSGBIND program.
- Messages can be accessed directly from the output message file.

See "How Message Retrieval Works" on page 855 for additional information.

MKMSGF Syntax

```
MKMSGF infile outfile [options]
```

OR

```
MKMSGF @controlfile
```

The infile field specifies the input file that contains message definitions. The input-file name can be any valid OS/2 file name, optionally preceded by a drive letter and a path.

The outfile field specifies the output file created by MKMSGF. The output-file name can be any valid OS/2 file name, optionally preceded by a drive letter and a path.

To differentiate between the two files, the following convention is recommended, using the same file name.

- The **infile** file should have a .TXT extension.
- The **outfile** file should have a .MSG extension.

Note: The output file ***cannot*** have the same file name and extension as the input file.

Creating Message Files with MKMSGF

Help

There are two ways to display MKMSGF help.

Short Syntax Help

To display a short version of MKMSGF syntax help, type `MKMSGF` at the prompt, with no parameters. The following will be displayed:

```
MKMSGF infile[.ext] outfile[.ext] [/V]
[/D <DBCS range or country>] [/P <code page>]
[/L <language id,sub id>]
```

Long Help

To display a longer version of MKMSGF help, including defaults, country codes, and language IDs, type `MKMSGF /?` at the prompt. The following will be displayed:

```
Use MKMSGF as follows:

MKMSGF <inputfile> <outputfile> [/V]
        [/D <DBCS range or country>]
        [/P <code page>]
        [/L <language family id,sub id>]

where the default values are:
   code page  -  none
   DBCS range -  none
A valid DBCS range is: n10,n11,n20,n21,...,nn0,nn1
A single number is taken as a DBCS country code.
```

The valid OS/2 language/sublanguage ID values are:

Figure 185. Language and Sublanguage ID Values

Code	Family	Sub	Language	Principal country
ARA	1	2	Arabic	Arab Countries
BGR	2	1	Bulgarian	Bulgaria
CAT	3	1	Catalan	Spain
CHT	4	1	Traditional Chinese	R.O.C.
CHS	4	2	Simplified Chinese	P.R.C.
CSY	5	1	Czech	Czechoslovakia
DAN	6	1	Danish	Denmark
DEU	7	1	German	Germany
DES	7	2	Swiss German	Switzerland
EEL	8	1	Greek	Greece
ENU	9	1	US English	United States
ENG	9	2	UK English	United Kingdom
ESP	10	1	Castilian Spanish	Spain
ESM	10	2	Mexican Spanish	Mexico
FIN	11	1	Finnish	Finland
FRA	12	1	French	France
FRB	12	2	Belgian French	Belgium
FRC	12	3	Canadian French	Canada
FRS	12	4	Swiss French	Switzerland
HEB	13	1	Hebrew	Israel
HUN	14	1	Hungarian	Hungary
ISL	15	1	Icelandic	Iceland
ITA	16	1	Italian	Italy
ITS	16	2	Swiss Italian	Switzerland
JPN	17	1	Japanese	Japan
KOR	18	1	Korean	Korea
NLD	19	1	Dutch	Netherlands
NLB	19	2	Belgian Dutch	Belgium
NOR	20	1	Norwegian - Bokmal	Norway
NON	20	2	Norwegian - Nynorsk	Norway
PLK	21	1	Polish	Poland
PTB	22	1	Brazilian Portuguese	Brazil
PTG	22	2	Portuguese	Portugal
RMS	23	1	Rhaeto-Romanic	Switzerland
ROM	24	1	Romanian	Romania
RUS	25	1	Russian	U.S.S.R.
SHL	26	1	Croato-Serbian (Lat	Yugoslavia
SHC	26	2	Serbo-Croatian (Cyr	Yugoslavia
SKY	27	1	Slovakian	Czechoslovakia
SQI	28	1	Albanian	Albania
SVE	29	1	Swedish	Sweden
THA	30	1	Thai	Thailand
TRK	31	1	Turkish	Turkey
URD	32	1	Urdu	Pakistan
BAH	33	1	Bahasa	Indonesia

Creating Message Files with MKMSGF

Input Message File

The input message file is a standard ASCII file that contains three types of lines:

- Comment lines
- Component identifier line
- Component message lines

Comment Lines

Comment lines are allowed anywhere in the input message file, except between the component identifier and the first message. Comment lines must begin with a semicolon (;) in the first column.

In the Input Message File Example, the comment lines are

```
; This is a sample of an input
; message file for component DOS
; starting with three comment lines.
```

Component Identifier Line

The component-identifier line contains a three-character name identifier that precedes all MKMSGF message numbers.

In the example, the component identifier is `DOS`.

Component-Message Lines

Each component-message line consists of a message header and an ASCII text message.

The message header is comprised of the following parts:

- A three-character component identifier
- A four-digit message number
- A single character specifying message type (`E, H, I, P, W, ?`)
- A colon (:)
- Followed by a blank space.

The following message types are used:

Type	**Meaning**
E	Error
H	Help
I	Information
P	Prompt
W	Warning

? no message assigned to this number

The message header must begin in the first column of the line. Only one header can precede the text of a message, although a message can span multiple lines.

Message numbers can start at any number, but messages must be numbered sequentially. If you do not use a message number, you must insert an empty entry in its place in the text file. An empty entry consists of the message number, with `?` as the message type, and no text.

The character % has a special meaning when used within the text of a message:

%0 is placed at the end of a prompt (type P) to prevent DosGetMessage from executing a carriage return and line feed. This allows the user to be prompted for input on the same line as the message text.

%1 – %9 are used to identify variable string insertion within the text of a message. These variables correspond to the Itable and IvCount parameters in the DosGetMessage call.

Component-Message Example

For example, `DOS0100E:` is DOS error message 100. For additional examples, see the "Input Message File Example" on page 852.

Output File

The output file contains the indexed message file that DosGetMessage will use. The output-file name can be any valid OS/2 file name, optionally preceded by a drive letter and a path. The output file ***cannot*** have the same name as the input file.

To differentiate between the two files, the following convention is recommended, using the same file name.

- The **infile** file should have a .TXT extension.
- The **outfile** file should have a .MSG extension.

Help-message file names begin with the component identifier, followed by H.MSG. For example, the help file associated with the component identifier DOS would be DOSH.MSG.

Options

Text-based messages in different code pages can be created using MKMSGF to display errors, help information, prompt, or provide general information to the application user.

MKMSGF uses the following parameter formats to build message files:

```
MKMSGF infile outfile /Pcodepage

MKMSGF infile outfile /Ddbcsrange or country id

MKMSGF infile outfile /LlangID,VerId

MKMSGF infile outfile /V

MKMSGF infile outfile /?

MKMSGF @controlfile
```

- Infile is the ASCII-text source file.

  Example:

  ```
  MSG
  MSG0001I: (mm%4dd%4yy) %2%4%1%4%3
  MSG0002I: (dd%4mm%4yy) %1%4%2%4%3
  MSG0003I: Current date is: %0
  ```

  %0 is a special argument that displays a prompt for user input.

  %1 – %9 are the arguments the user can use to insert text in a message.

- Outfile is the binary output message file.
- @controlfile is the message definition file.

Options

/P Code-page ID for the input message file. See “/P Option” on page 851

/D DbcsRange or country ID for the input message file. See “/D Option” on page 851

/L Language family ID (one word) and language version ID (one word). See “/L Option” on page 851

/V Verbose display of message file control variables as the message file is being created. See “/Verbose Option Output Example” on page 851

/? Help display of command syntax for MKMSGF.

Note: Any combination of /P, /D, /L, and /V switches can be used for either the command line or @controlfile execution method.

The / switch prefix and the - prefix can be used interchangeably when defining switches to MKMSGF.

/D Option

The DBCS option (/D) specifies the DBCS Range or country ID for that input message file.

Valid DBCS country ID will cause the initialization of valid DBCS ranges to be set up for this file.

/L Option

The Language option (/L) specifies the language family ID (one word) and language version ID (one word).

Valid combination of language family and language version will be set for this file.

A valid language family with invalid or undefined language version id will cause a default value of 1 to be set for language version.

/P Option

The Code-page option (/P) specifies the code-page ID for that input message file.

Up to 16 /P combinations can be saved with the message file.

/P cannot be used to identify DBCS data.

/Verbose Option Output Example

Following is a sample of MKMSGF output, using the Verbose option (/V). This output was produced using the following command:

```
mkmsgf myapp.txt myapp.msg /v
```

```
  strIn     = myapp.txt
  strOut    = myapp.msg
  StrIncDir = (null)
  CodePages = 437
  Language family id = 0 and sub id = 0
  Language family id and sub id = unspecified
  flags     = none
  CP_type   = SBCS
"myapp.txt": length = 382 bytes.
29 messages scanned.  Writing output file...
Size of table entry: word
```

Control Files

The control file (specified as @*controlfile*) is used to create multiple-code-page message files. The at sign (@) is not part of the file name, but rather, a delimiter required before a control-file name.

The control file has the following format:

Example:

```
root.in root.out /Pcodepage /Ddbcsrang/ctryid /LlangID,VerId
sub.001 sub1.out /Pcodepage /Ddbcsrang/ctryid /LlangID,VerId
                 .
                 .
sub.00n subn.out /Pcodepage /Ddbcsrang/ctryid /LlangID,VerId
```

The help option (/?) is invalid due to the purpose of the definition file.

Note: Any combination of /P /D /L and /V switches can be used for either the command line or msg_definition_file execution method.

Input Message File Example

Following is an example of an input message file:

```
; This is a sample of an input
; message file for component MAB
; starting with three comment lines.
MAB
MAB0100E: File not found
MAB0101?:
MAB0102H: Usage: del [drive:][path] filename
MAB0103?:
MAB0104I: %1 files copied
MAB0105W: Warning! All data will be destroyed!
MAB0106?:
MAB0107?:
MAB0108P: Do you wish to apply these patches (Y or N)? %0
MAB0109E: Divide overflow
```

Binding Messages with MSGBIND

The MSGBIND program binds a message segment to an executable program. It does this by reading an input file that specifies the executable files to modify. For each executable file, MSGBIND specifies which message files to scan, and for each message file, it specifies which messages to include in the executable file. Although the resulting executable file will be larger, access to messages will be faster.

In the OS/2 operating system, message segment/objects are packed with other application code. If the size of the code segment/object and the bound messages exceeds 64KB, the following statement in the program definition file (.DEF) isolates the application code from the message statement/object:

16 Bit Applications
SEGMENT

32 Bit Applications
SEGMENT '_MSGSEG32' CLASS 'CODE'

MSGBIND Syntax

The MSGBIND command line has the following form:

```
MSGBIND infile
```

The **infile** field specifies the input file that identifies the executable files, output message files, and message numbers that will be bound. The input-file name can be any valid OS/2 file name and can include an optional file name extension.

Input File

The input file contains the following three types of lines:

- > Executable file
- < Message file
- Message numbers.

Executable file

The File in which messages are to be bound is preceded by a greater-than symbol (>). The name of the file can be any valid OS/2 file name.

Binding Messages with MSGBIND

Two or more different executable files can be modified by the specifications found in one input file. MSGBIND continues to use this file until it encounters another greater-than symbol.

Message file

The message file to be read from is preceded by a less-than sign (<). You create this file by using the MKMSGF program. The name can be any valid OS/2 file name. All message numbers that follow it are located in the specified message file and are copied to the current output executable file. MSGBIND reads the message-number list until it encounters one of the following: the end of the input file, a new output specification, or a new input message file.

Message Numbers

The messages in the message file are listed below the message-file name. Only those message numbers that you specify will be added. You can also specify an asterisk (*) to indicate that all messages within the message file will be added. Message numbers must consist of a three-letter component identifier followed by a four-digit message number.

See "Input Message File" on page 848 for a more detailed description of message numbers.

VisualAge C++ Message Files

If you plan to run your application on workstations where VisualAge C++ is not installed, and your application generates runtime messages from any of the VisualAge C++ message files, you must bind the appropriate VisualAge C++ messages to your application as well as any of your own.

The VisualAge C++ message files you may need are:

DDE4.MSG	C runtime library and I/O Stream and Complex class libraries.
IBMCRERR.MSG	Regular expressions.
DDE4C01E.MSG	Collection class libraries.
CPPOOC3U.MSG	User Interface class libraries.
DAXCLS.MSG	Database Access class libraries.

You can choose to bind only the messages you expect your application to generate, or all messages.

Multiple Code-Page Message Files

Multiple code-page message files can also be bound to an executable file, which enables a user to bind to an application messages for different countries.

The following example shows how three messages in two different languages can be bound to an executable file:

```
MSGBIND infile
```

where *infile* consists of the following:

```
>PROG1.EXE
<TEXTUS.MSG
MSG0001
MSG0002
MSG0003
<TEXTIT.MSG
MSG0001
MSG0002
MSG0003
```

where:

- PROG1.EXE is the executable file to be modified.
- TEXTUS.MSG is the file, created using MKMSGF, which contains messages in US English.
- TEXTIT.MSG is the file, created using MKMSGF, which contains the same messages translated into Italian.
- MSGnnnn defines the messages to be bound to the application:

MSG	Message component ID
0001	Message number

Displaying Help

To display MSGBIND help, type `MSGBIND` at the prompt, with no parameters. The following will be displayed:

```
usage: MSGBIND scriptfile
```

How Message Retrieval Works

When an application requests the message retriever for text associated with a message number, a test is made to determine if there is a bound message segment with this executable file. If true, each bound message segment is searched for a match with the current session's code-page number.

If a match is made, then the message number is searched for. If it is found, the message will be returned to the caller. Otherwise, the search of remaining bound message segments will continue.

If no match results from a search of all message segments, the message file on the disk is searched. DosGetMessage will access the message file under any of the following conditions:

- The message file is in the current directory.
- The message file is in the path specified in the DPATH environment variable (protect mode).
- The message file is in the path specified in the APPEND environment variable (real mode).
- The fully-qualified file name is specified in DosGetMessage.

Sample Input File

```
>c:\cmd.exe
<c:\os20\dosutil.msg
DOS0100
DOS0123
DOS0245
>c:\format.exe
<c:\os20\dosutil.msg
DOS0001
DOS0006
<c:\format.msg
FMT0001
FMT0002
<c:\myown.msg
*
```

The first line of a MSGBIND input file specifies that the executable file to modify is CMD.EXE. The messages DOS0100, DOS0123, and DOS0245 are read from the file DOSUTIL.MSG and added to the CMD.EXE file. The MSGBIND program then encounters an executable-file option for the FORMAT.EXE file. The messages DOS0001 and DOS0006 are read from DOSUTIL.MSG and added to FORMAT.EXE. Next, the messages FMT0001 and FMT0002 are read from the file FORMAT.MSG and added to FORMAT.EXE. Finally, because an asterisk is specified, all the messages are read from the file MYOWN.MSG and added to FORMAT.EXE.

The files DOSUTIL.MSG and FORMAT.MSG in this example are two output-message-file names from the MKMSGF program.

Getting Quick Information with KwikINF

KwikINF provides you with a quick and convenient method of accessing information in online documents stored in the OS/2 BOOKSHELF from anywhere on the desktop, with the exception of DOS or WIN-OS/2 sessions.

When KwikINF has been started, you can open a dialog with KwikINF by pressing a user-selectable hot key. Until you configure KwikINF, your KwikINF hot key is ALT+Q. The KwikINF window includes a **Configure** push button. This opens another dialog with KwikINF: the Configure KwikINF window. The KwikINF window also allows you to initiate searches for text strings in online documents of choice.

Automatic Text Retrieval

The KwikINF window includes a **Search String** entry field. You can specify the text string you want KwikINF to search for. Or, under certain conditions, this entry field automatically contains the word located under the cursor when you press your KwikINF hot key.

This text retrieval feature is available from OS/2 full-screen and Presentation Manager (PM) multi-line entry (MLE) fields. This feature is also available from PM AVIO and VIO windows. Communication Manager's 3270 emulator is a common example of a PM AVIO window.

An OS/2 Window is a VIO window. This means that if, for example, you open an OS/2 Window and start an OS/2 text-based application, KwikINF will automatically retrieve the word under the cursor when you press your KwikINF hot key. This automatic text-retrieval feature is not available from graphic-text PM windows.

BOOKSHELF Online Documents

The KwikINF window includes a **Volume to Search** list box of all online documents stored in the OS/2 BOOKSHELF subdirectories. KwikINF initiates searches for information in any online document in this list.

Quick Information with KwikINF

The BOOKSHELF is an environment variable, set in CONFIG.SYS, that contains a list of subdirectories containing online documents created as viewable .INF files with the Information Presentation Facility (IPF). The BOOKSHELF environment variable is set as follows:

```
SET BOOKSHELF=<subdirectory>;...;<subdirectory>;
```

Online documents for OS/2 (for example, the *Command Reference*) are stored in the \OS2\BOOK subdirectory of the drive on which OS/2 is installed. Online documents for VisualAge C++ (for example, the *Programming Guide*) are stored in the \HELP subdirectory under the drive and directory you specified when you installed the VisualAge C++ information.

If you had installed OS/2 on drive C:, VisualAge C++ on drive D: in the IBMCPP directory, and no other products, your BOOKSHELF environment variable would look like:

```
SET BOOKSHELF=C:\OS2\BOOK;D:\IBMCPP\HELP;
```

The online document where KwikINF looks for the **Search String** is selected from the **Volume to Search** list box by KwikINF or by you. KwikINF selects the **Volume to Search** by looking for the text string that has a matching entry in the KwikINF index file or, if there is no matching entry in the index file, in the **Default Volume** you have selected in the Configure KwikINF window.

Index Files for Rapid Search

The KwikINF index files provide a rapid-search mechanism for locating specific kinds of information in online documents in the BOOKSHELF. The KwikINF index file consists of one or more concatenated files stored in the BOOKSHELF and defined by the HELPNDX variable in CONFIG.SYS as shown in the following example:

```
SET HELPNDX=EPMKWHLP.NDX+CPP.NDX+CPPBRS.NDX
```

where EPMKWHLP.NDX is the KwikINF index file for the OS/2 Toolkit information, and CPP.NDX and CPPBRS.NDX are the index files for the VisualAge C++ C and C++ information.

Index File Format

An index file provides a mapping for keywords to the online information for that keyword. Index files are provided with VisualAge C++, but you can also create your own index files for your own or other online information.

The first line of an index file contains the keyword `EXTENSIONS:`, followed by a list of file extensions that identify the types of files for which the keywords in the file apply. Each extension is 1 to 3 case-insensitive characters, separated by spaces. You can use an asterisk (*) to indicate that all file extensions are valid. For example, for an index file of C keywords, you might put:

```
EXTENSIONS:  C H I SQC
```

The extension is used to resolve ambiguities when you request contextual help from a tool such as the VisualAge C++ editor; if you are editing a REXX file, the editor would not use any keywords from an index file marked with the above extensions.

The second line of the index file contains the keyword `DESCRIPTION:` followed by text that describes what the index file is for. For example, the second line of the VisualAge C++ index file is:

```
DESCRIPTION: IBM VisualAge C++ language, library, and class libraries
```

The remaining lines in the index file contain one or more entries of the form:

(*keyword*, *command*)

where

`(` Indicates the beginning of an entry.

`)` Indicates the end of an entry.

keyword Is a string on which a case-sensitive match should be made. The keyword should not contain the character (or). It can end with an asterisk (*) to match any strings that begin with the same characters. For example, the entries for the OS/2 APIs use Dos* as the search keyword. If * is one of the characters in the keyword, use a second asterisk as the escape character (for example, **).

If more than one instance of the same keyword is found, which is used depends on the implementation of the tool you are using.

command Is any valid command. Most often, the command invokes the `view` utility for the online document that contains information for the keyword. You can use a tilde (˜) in the command to represent the exact *keyword* matched. For example, given the entry `(API*, view mydoc.inf ˜)`, if you were looking for `APIsamp`, the command invoked would be `view mydoc.inf APIsamp`.

Any characters not contained within parentheses are considered comments.

Quick Information with KwikINF

The following shows a sample index file:

```
EXTENSIONS: *
DESCRIPTION: Sample Index File
(Dos*, VIEW CPREF ~)
(_Optlink, VIEW CPPPROG.INF Linkage Keywords)
(WinCreateWindow, VIEW PMWIN.INF ~)
```

You can also look at the VisualAge C++ index files for real-life examples.

Enabling Online Documents

You can enable any online document for KwikINF by:

1. Creating the online document as a viewable .INF file using the Information Presentation Facility (IPF).
2. Appending the name of the subdirectory where it is stored to the BOOKSHELF in CONFIG.SYS.
3. Creating an index file to support the KwikINF rapid-search mechanism, storing it in the BOOKSHELF, and adding it to the HELPNDX variable in CONFIG.SYS.

For example, you can enable your online document MYDOC stored in MYSUBDIR subdirectory for KwikINF by:

1. Compiling the tagged source for MYDOC with the IPF compiler by entering:

   ```
   IPFC MYDOC /INF
   ```

2. Modifying the BOOKSHELF statement in CONFIG.SYS as follows:

   ```
   SET BOOKSHELF=...;C:\MYSUBDIR;
   ```

3. Creating MYINDEX file in MYSUBDIR as described in “Index File Format” on page 858.
4. Modifying the HELPNDX variable in CONFIG.SYS as follows:

   ```
   SET HELPNDX=EPMKWHLP.NDX+CPP.NDX+CPPBRS.NDX+MYINDEX.NDX
   ```

For more information on creating an IPF-viewable online document, see Chapter 62, “Creating Online Documentation” on page 867 and the online *IPF Guide and Reference* in the VisualAge C++ Information folder.

Using KwikINF

KwikINF is installed as a program object in the OS/2 Toolkit Information folder. You start KwikINF by double-clicking on the KwikINF object or by entering KwikINF from the command line of an OS/2 Window. You can start KwikINF automatically when you start OS/2 by placing a shadow of the KwikINF object in the Startup folder in the OS/2 System folder on the desktop. You shadow an object by pressing CTRL + SHIFT while dragging the object.

KwikINF installs a PM system hook to monitor keystrokes in PM sessions and OS/2 character device monitors to monitor keystrokes in OS/2 full-screen sessions. KwikINF will install only one copy of the hook and monitors, even if you attempt to re-start KwikINF.

When KwikINF has been started, you can initiate searches for text strings in online documents of choice by pressing a user-selectable hot key.

Note: You cannot initiate searches for text strings in online documents from DOS or WIN-OS/2 sessions.

Until you configure KwikINF, your KwikINF hot key is ALT+Q. You configure KwikINF by pressing your KwikINF hot key and then pressing the **Configure** push button to open the Configure KwikINF window.

Note: When you start KwikINF by double-clicking on the KwikINF object in the Toolkit Information folder, a message box tells you what hot key opens the KwikINF window. This technique can also be used to determine what your current KwikINF hot key is, after KwikINF has been started.

How you initiate a search for information in online documents is dependent on where you are on the desktop when you press the KwikINF hot key:

- From an OS/2 full-screen session, a PM VIO or AVIO window, or PM MLE: position the cursor on the string you want to search for and press the KwikINF hot key. KwikINF retrieves the word at the cursor. If you have configured KwikINF to display the KwikINF window when the KwikINF hot key is pressed, KwikINF automatically places the retrieved word in the **Search String** entry field of the KwikINF window. When you press the **Search** push button or Enter, KwikINF displays the information. If you have configured KwikINF to bypass the KwikINF window when the KwikINF hot key is pressed, KwikINF automatically displays the information.

  If no word is under the cursor, the previous **Search String** is used. If no previous **Search String** exists, KwikINF displays the Contents of the **Default volume to search**.

- From a graphic-text PM window: press the KwikINF hot key, then type the string you want to search for in the **Search String** entry field of the KwikINF window. The KwikINF text-retrieval feature is not available from graphic-text PM windows.

The online document where KwikINF looks for the text string is selected from the **Volume to Search** list box by KwikINF or by you. To open the online document to the panel that contains the information, press the **Search** push button or press Enter.

KwikINF From the Command Line

You can start, terminate, and configure KwikINF from the command line in an OS/2 Window by entering:

```
KwikINF [no options] [/C] [/T] [/?]
```

where:

no options starts KwikINF. After entering this command, the default KwikINF hot key (ALT + Q) is enabled.

/T terminates KwikINF and disables the KwikINF hot key.

/C opens the Configure KwikINF window. Use this window to select another KwikINF hot key, to select a default online document from the BOOKSHELF to search, and to select the activation behavior of the KwikINF window.

/? displays the following information.

```
Usage: KwikINF [Option]
 Option    Description
   /C      Configure KwikINF
   /T      Terminate KwikINF
   /?      This short help list
```

Configuring KwikINF

You configure KwikINF through the Configure KwikINF window. KwikINF displays this window when you press the **Configure** push button on the KwikINF window or when you enter the following from the command line of an OS/2 Window:

```
KwikINF /C
```

The Configure KwikINF window allows you to:

- Select another KwikINF hot key.
- Specify the number of OS/2 full-screen sessions enabled for KwikINF.
- Specify the name of the default online document KwikINF searches.

- Select the activation behavior of the KwikINF window.

Use the push buttons on the Configure KwikINF window as follows:

- Press **OK** to enable your configuration choices.
- Press **Cancel** to cancel your configuration choices. This closes the Configure KwikINF window.
- Press **Help** to get general help for the current window.

Activation Key Sequence

The **Activation Key Sequence** provides a selectable list of KwikINF hot keys. To access the list, single-click on the down arrow. Select the KwikINF hot key of your choice from the following list:

CTRL	+ A
CTRL	+ H
CTRL	+ Q
ALT	+ A
ALT	+ Q (this is the default hot key)

The KwikINF hot key initiates searches for information in online documents from anywhere on the desktop, with the exception of DOS or WIN-OS/2 sessions.

Full Screen Sessions

Use the **Number of Fullscreen Sessions to Monitor** spin button to specify the number of OS/2 full-screen sessions enabled for KwikINF. KwikINF is implemented as a PM system hook to monitor keystrokes in PM sessions and as OS/2 character device monitors to monitor keystrokes in OS/2 full-screen sessions. For OS/2 full-screen sessions, KwikINF will monitor only the number of sessions specified here.

Default Volume to Search

Use the **Default Volume to Search** single selection list box to specify which online document in the BOOKSHELF you want KwikINF to search by default. KwikINF looks for the **Search String** in this online document, when there is no matching entry in the KwikINF index file. Select the online document, then select the **OK** push button to activate the selection.

Activation Behavior

Use the **Activation Behavior** radio buttons to select the behavior of the KwikINF window. The KwikINF window can be displayed or bypassed when the user presses the KwikINF hot key after KwikINF has been installed.

- Select the **Display KwikINF Window** radio button to tell KwikINF that you always want the KwikINF window to be displayed when you press the KwikINF hot key to initiate searches for information. This is the default behavior of the KwikINF window.

  When the you press the KwikINF hot key, you can initiate a search for the text string that may be automatically displayed in the **Search String** entry field or for the text string that you enter into this field. You can also specify which online document in the BOOKSHELF KwikINF searches for the text string.

- Select the **Bypass KwikINF Window** radio button to tell KwikINF that you do not want the KwikINF window to be displayed when you press the KwikINF hot key to initiate searches for information. This is typically used when working under conditions where the KwikINF automatic text-retrieval feature is available.

  When you press the KwikINF hot key, KwikINF automatically looks for the text string under the cursor in the online document that has a matching entry in the KwikINF index file or, if there is no matching entry in the index file, in the **Default volume** selected from the Configure KwikINF window.

  You configure KwikINF by pressing the **Configure** push button in the KwikINF window. To configure KwikINF when this window is bypassed, press SHIFT + your KwikINF hotkey to display the Configure KwikINF window.

Searching Using the KwikINF Window

If you have configured KwikINF to display the KwikINF window (this is the default condition), the KwikINF window is displayed when you press your KwikINF hot key.

The KwikINF window allows you to search for a text string in an online document in the OS/2 BOOKSHELF. The text string is typed by you in the **Search String** entry field or is automatically retrieved by KwikINF, under certain conditions, from under the cursor when you press your KwikINF hot key.

The online document that KwikINF searches for the text string is selected from the **Volume to Search** list box by KwikINF or by you. KwikINF selects the **Volume to Search** by looking for the text string that has a matching entry in the KwikINF index file or, if there is no matching entry in the index file, in the **Default Volume** you have selected in the Configure KwikINF window. Or you can override KwikINF's selection of the **Volume to Search** by making your own selection from the list box.

To initiate the search for the text string in the online document, press the **Search** push button or press Enter. If the search is successful, KwikINF opens the online document to the online panel that contains the information and displays a window with a title bar that matches the search string.

If the search is not successful, you can search any online document for the information by following this procedure:

- Clear the **Search String** entry field.
- Select an online document from the **Volume to Search** list box. The Contents window of the online document appears.
- Select **Services** from the menu bar.
- Select **Search** from the **Services** pull down menu. The **Search** help window appears.
- Type the text string, then select the **All libraries** radio button.
- Select the **Search** push button or press Enter.

Use the push buttons on the KwikINF window as follows:

- Press **Search** to initiate the search for the text string in the **Search String** entry field in the selected online document.
- Press **Cancel** to cancel the request to search for the text string and to close the KwikINF window.
- Press **Configure** to display the Configure KwikINF window.
- Press **Help** to get general help for the current window.

Search String Entry Field

KwikINF searches for the text string in this entry field in the selected online document in the OS/2 BOOKSHELF.

Under certain conditions, KwikINF automatically retrieves the word under the cursor when you press your KwikINF hot key. Letters, numbers, underscores, and the pound sign are retrievable by KwikINF. Blank spaces and other special characters are used as delimiters and are not retrievable by KwikINF.

You may also type any text string you want into this field. All characters are valid in the entry field to allow for special search criteria.

VOLUME TO SEARCH List Box

The KwikINF window includes a list box of all online documents stored in the OS/2 BOOKSHELF subdirectories. KwikINF initiates searches for information in any online document in this list. The online document where KwikINF looks for the **Search String** is selected from the **Volume to Search** list box by KwikINF or by you. KwikINF selects the **Volume to Search** by looking for the text string that has a matching entry in the KwikINF index file or, if there is no matching entry in the index file, in the **Default Volume** you have selected in the Configure KwikINF window. Or you can override KwikINF's selection of the **Volume to Search** by making your own selection from the list box.

You can also open and display the Contents of an online document by double-clicking on an entry in this list box.

KwikINF Keys Help

Use your KwikINF hot key (ALT+Q or the hot key you select from the Configure KwikINF window) to display the KwikINF window. You can also use your KwikINF hot key to initiate a search for a text string automatically, when you configure KwikINF to bypass the KwikINF window.

To re-configure KwikINF, when you have configured KwikINF to bypass the KwikINF window, press SHIFT + your KwikINF hot key.

To determine what your KwikINF hot key is, double-click on the KwikINF program object in the OS/2 Toolkit Information folder.

62

Creating Online Documentation

The Information Presentation Facility (IPF) is a Toolkit tool that you can use to create online information, to specify how it will appear on the screen, to connect various parts of the information, and to provide help information that can be requested by the user. IPF features include:

- A tagging language that formats text and provides ways to connect information and customize the information display.
- A compiler that creates online documents and help windows.
- A viewing program that displays formatted online documents (`view`). This program is part of the OS/2 operating system.

The syntax for the IPF compiler command is:

```
                       ┌───────────────┐
►►──IPFC──source_file──▼───────────────┴──┬──────────────────────────┬──►◄
                       └─/──options─┘     └─>──output_message_file─┘
```

Enabling help for applications requires programming code that communicates with IPF and with the PM APIs to display help windows.

For more information on creating help for applications, see the online *IPF Guide and Reference* in the VisualAge C++ Information folder.

Creating Online Documentation

Compressing Files with PACK and PACK2

PACK and PACK2 reduce the size of a file by compressing its data. You can use these packing programs for a single file or for a group of files, thereby reducing the disk space required for your OS/2 application.

The following sections refer only to PACK, but the information applies to both PACK and PACK2. The options, parameters, and function for PACK2 are identical to PACK. The only difference between PACK and PACK2 is that PACK2 has a better compression algorithm.

Starting PACK

You start PACK with a single command from the command line. You can specify the files to be compressed in one of two ways:

- Specify the names of all the files you want to compress on the command line. This compresses each file individually.
- Specify a single file on the command line that contains a list of all the files you want to compress. This compresses all files listed into a single file. 📖 See “Creating a List File” on page 871 for details about the list file format.

Include the drive and path if the files are not in the working directory. You can specify file names with any combination of uppercase and lowercase letters. File-name extensions are not required; however, if you specify a file name that has an extension, also type the extension. Global file-name characters are permitted on the command line, but not in a file containing a file list.

The command-syntax is as follows:

```
PACK sourcefile [packedfile] [options]
```

where:

sourcefile Specifies the names of the files to be compressed, or a single file that lists the files to be compressed. This parameter is required.

When the data is compressed, the name of the source file is placed in the header of the compressed file and is used as the destination file name during unpacking.

packedfile Specifies the name of the compressed file to create. Files that contain compressed data can be recognized by the @ symbol as the last character in the file name. If you do not specify this parameter, PACK gives the compressed file the same name as *sourcefile* and changes the last character of the file extension to @.

options Specifies one or more options described under "PACK Options."

PACK Options

You can specify the following options for PACK:

/A Adds data from *sourcefile* to the data in *packedfile*.

The source file can be either in a compressed or uncompressed state. If the source file is in an uncompressed state, the data is compressed before being added to the file containing the compressed data.

To use /A, *sourcefile* must specify a single source file; it cannot be a list of files.

/C Specifies that the current path be placed in the header of the file that contains the compressed data. When the UNPACK command is used, this path will be the destination path for the file that contains the uncompressed data.

You cannot use /C when you specify /H:*hdrpath*\.

/D:*hdrdate* Records the date in the header of the file that has the compressed data, and also in the destination file when it is uncompressed.

The date must follow the format /D:MM-DD-YYYY (for example: /D:08-20-1991 and /D:12-30-2010)

/H:[*hdrpath***\][***hdrfile***]**

Specifies the path and/or file name to use when restoring the compressed file. This information is stored in the header of the compressed file to be used by UNPACK. (Note that you can override this option when you use UNPACK by specifying the target path and file name.)

hdrpath must end with a backslash (\), regardless of whether *hdrfile* is specified. You cannot specify a drive.

If you do not specify *hdrpath*, no directory information is included in the header. If you do not specify *hdrfile*, PACK uses *sourcefile*.

You cannot use both /H:*hdrpath*\ and /C.

/L Indicates that *sourcefile* is a list of files to be compressed. The list file itself is not compressed; all files listed within it are compressed into a single file.

/R Removes the file specified by *sourcefile* from the file that contains only compressed data. The *sourcefile* parameter must specify the path and file name exactly as they appear in the header of the file with the compressed data; otherwise, the following error message appears on the screen.

```
The specified file to remove was not found.
```

To use /R, *sourcefile* must specify a single source file; it cannot be a list of files.

/T:*hdrtime* Records the time in the header of the file that has the compressed data, and also in the destination file when it is uncompressed.

The time must follow the format /T:HH.MM (for example /T:02.06 and /T:14.54). Hour 00 represents 12 a.m. and hour 12 represents 12 p.m.

To display the path and file-name information stored in the header of the compressed file, use UNPACK with the /SHOW option. UNPACK is described in "Restoring Compressed Files with UNPACK" on page 872 and in the online *OS/2 Command Reference*.

Creating a List File

To use a list file with PACK, you must first create a file that contains the names of the files you want to compress. You can give the list file any name. Following is an example of specifying a list file at the command line.

```
PACK DEVICE.LST DEVICE.DRV /L
```

The /L indicates that DEVICE.LST is a list file. If the list file is not in the working directory, you must specify the drive and path. Global file-name characters are not permitted in the list-file name. DEVICE.DRV is the destination file for the end-to-end-compressed data. (End-to-end compressed data is the data from each of the files contained in the list file. This data is stored in a contiguous format in the destination file.)

The syntax used in the list file is similar to that used in the command line. The syntax for a single line in the list file is:

```
sourcefile [options]
```

where *options* are any of /C, /D, /H, or /T. These options work as if you specified them for *sourcefile* on the PACK command line. For a description of these options, see "PACK Options" on page 870.

When using a list file, you cannot specify global file-name characters in the source-file name. You can include comments or blank lines by entering a semicolon as the first character of the line. An example of a list file follows:

```
;This is a comment
C:\OS2\COMMAND.COM
CONFIG.SYS /H:CONFIG.BAK /C
\OS2\INSTALL\DDINSTAL.EXE
 /H:\OS2\DDINSTAL.TMP
 /D:10-15-91 /T:11.45
```

Restoring Compressed Files with UNPACK

UNPACK restores a file of compressed data to its original size and copies it to a specified drive and path.

To start the UNPACK command, type:

```
UNPACK sourcefile [options]
```

where *sourcefile* specifies an existing compressed file, and *options* specifies one or more options, described in "UNPACK Options." If *sourcefile* contains more than one compressed file, UNPACK restores each file it contains.

UNPACK Options

You can specify one or more of the following options:

targetdrv: Specifies the drive where you want UNPACK to copy one or more restored files.

When you specify a target drive but not a path, UNPACK uses the path information stored in the header of the compressed file.

targetpath Specifies the name of the directory (and its subdirectories) where you want UNPACK to copy one or more restored files.

If you do not specify *targetpath*, UNPACK uses the path information stored in the header of the compressed file. Specifying *targetpath* overrides the path information in the header.

/SHOW Displays the destination path and file-name information placed by PACK in the header of the compressed file.

/N:singlefile Extracts and uncompresses one file from a file that contains multiple files of compressed data.

/V Verifies that sectors written to the target disk are recorded properly. This parameter lets you know that critical data has been correctly recorded.

This parameter causes UNPACK to run more slowly because a check is made for each entry recorded on the disk.

/F Specifies that files with extended attributes should not be unpacked or copied if the destination file system does not support extended attributes.

Restoring Compressed Files with UNPACK

64

Demangling Compiled C++ Names with CPPFILT

When the VisualAge C++ compiler compiles a C++ program, it encodes all C++ symbolic names to include type and scoping information. This encoding process is called *mangling*. The linker uses the mangled names in the object files to resolve external references using the exported names.

Tools that use the files with mangled names do not have access to the original source code names, and therefore present the mangled names in their output. Using a process called *demangling*, the CPPFILT utility converts the mangled names to their original source code names so that they can be easily identified.

Note: The demangling routines in the runtime library provided with VisualAge C++ offer another method for converting mangled names to their original source code names. You can use these routines to develop tools that manipulate mangled names. For more information on the demangling routines, refer to the appendix on *Mapping* in the *IBM VisualAge C++: Programming Guide*, and the information in the **<demangle.h>** header file.

Using the CPPFILT Utility

The CPPFILT utility converts mangled names to demangled names in two separate modes:

Text Specify the names of one or more ASCII text files to have the CPPFILT substitute demangled names for mangled names wherever it finds them in the text. Input can also be read from **`stdin`**.

Binary Specify the names of one or more object (.OBJ) and library (.LIB) files to produce demangled output in a format that is suitable for inclusion in a DLL module definition (.DEF) file.

All output is sent to **`stdout`**.

Text Mode

Use the CPPFILT utility in Text mode when you want to simply substitute demangled names for mangled names wherever they are found in the text.

To use the CPPFILT utility in text mode, type `cppfilt` followed by any valid text mode options on the command line.

The syntax for the `cppfilt` command in Text mode is:

```
          ┌───────────┐ ┌───────────────────┐
►►──cppfilt──▼───────────┴─▼───────────────────┴──────────────────────────►◄
          └─/option─┘   └─text-filename─┘
```

Text Files

The file you specify for the `cppfilt` command in text mode should be an ASCII text file that is present in the current directory, unless you specify its path.

If you do not specify a text file, the input is read from **`stdin`**.

Output in Text Mode

Output in text mode would be the input text with the mangled names replaced by their demangled names wherever they are found in the text.

All output is sent to **`stdout`**.

Text Mode Options

The CPPFILT utility has the following options in Text mode:

/C	Demangle stand-alone class names
/H or /?	Display help for CPPFILT
/M	Produce symbol map
/Q	Suppress display of logo
/S	Demanlge compiler-generated symbol names
/T	Produce side-by-side demangling
/W*nnn*	Specify width of field for demangled names

Note: You can specify options using the slash form (/S) or the dash form (-S).

/C (Class Names)

Syntax:	Default:
`/C`	Do not demangle stand-alone class names

Use `/C` to instruct the CPPFILT utility to demangle stand-alone class names. Stand-alone class names are names that do not appear within the context of a function name or member variable. These names are not normally produced by the compiler.

For example, the stand-alone class name `Q2_1X1Y` would be demangled as `X::Y` if you specify the `/C` option.

If you do not specify the `/C` option, the default is not to demangle stand-alone class names.

/H (Help)

Syntax:	Default:
`/H` `/?`	None

Specify the `/H` or `/?` option on the CPPFILT command line to see a short online help on the `cppfilt` command syntax and options.

If you do not specify this option, the default is not to display any online help.

/M (Symbol Map)

Syntax:	Default:
`/M`	Do not produce symbol map

Specify the `/M` option on the CPPFILT command line to produce a symbol map on standard output. The map contains a list of the mangled names and their corresponding demangled names. This output follows the normal filtered output.

If you do not specify the `/M` option, the default is not to produce a symbol map.

/Q (Do Not Display Logo)

Syntax:	**Default:**
`/Q`	Display logo and copyright notice

Specify the `/Q` option on the CPPFILT command line to suppress the display of the logo and copyright notice for the CPPFILT utility.

If you do not specify this option, the default is to display the logo and copyright notice.

/S (Special Symbol Names)

Syntax:	**Default:**
`/S`	Do not demangle special compiler-generated symbol names

Specify the `/S` option on the CPPFILT command line to instruct the CPPFILT utility to demangle special compiler-generated symbol names.

For example, the compiler-generated symbol name, `__ct__3FooFUi`, that represents a constructor for the class `Foo` would be demangled as `Foo::Foo(unsigned int)`.

If you do not specify the `/S` option, the default is not to demangle special compiler-generated symbol names.

/T (Mangled and Demangled Names Together)

Syntax:	**Default:**
`/T`	Replace mangled name with demangled name only

Specify the `/T` option on the CPPFILT command line to produce side-by-side demangling. That is, each mangled name is replaced with the demangled name followed by the original mangled name in the text.

If you do not specify the `/T` option, the default is to replace the mangled name by the demangled name only.

/Wnnn (Width)

Syntax:	**Default:**
/W*width*	No fixed field width

Specify the /W*width* option on the CPPFILT command line to have the CPPFILT utility print the demangled names in fields *width* characters wide. If the name is shorter than *width*, it is padded to the right with blanks; if longer, it is truncated to *width* characters.

If you do not specify the /W *width* option, the default is not to have a fixed field width when printing demangled names in the text.

Binary Mode

Use the CPPFILT utility in Binary mode when you want to demangle names in object (.OBJ) and library (.LIB) files to produce output that is suitable for inclusion in a DLL module definition (.DEF) file.

To use the CPPFILT utility in Binary mode, type cppfilt /B followed by any valid Binary mode options on the command line. The CPPFILT utility switches to the Binary mode of operation when it encounters the /B option.

The syntax for the cppfilt command in Binary mode is:

```
                  ┌─────────────┐  ┌─────────────────┐
►►──cppfilt──/B──▼│             │─▼│                 │─────────────────────►◄
                  └─/option─────┘  ├─.OBJ-filename───┤
                                   └─.LIB-filename───┘
```

.OBJ and .LIB Files

The file names that you specify after the cppfilt /B binary-mode command must be object (.OBJ) or library (.LIB) files. They must be present in the current directory, unless their paths are specified. CPPFILT will also search for library files along the paths specified in the LIB environment variable if the files are not found in the current directory.

Input cannot be read from **stdin** in binary mode. You must specify object or library filenames for input.

All output is sent to **stdout**.

Output in Binary Mode

CPPFILT output in Binary mode lists any libraries and any object files within each library. If you specify the /X, /R, and /P options, the exported, referenced, and public symbols are also listed for each object file.

For example, the command

```
cppfilt /B /P /O 1000 /N c:\ibmcpp\lib\dde4cc.lib
```

would produce the following output for a library file called DDE4CC.LIB:

```
;From library:  c:\ibmcpp\lib\dde4cc.lib
  ;From object file:  C:\ibmcpp\src\IILNSEQ.C
    ;PUBDEFs (Symbols available from object file):
      ;ILinkedSequenceImpl::setToPrevious(ILinkedSequenceImpl::Node*&) const
      setToPrevious__19ILinkedSequenceImplCFRPQ2_19ILinkedSequenceImpl4Node    @1000 NONAME
      ;ILinkedSequenceImpl::allElementsDo(void*,void*) const
      allElementsDo__19ILinkedSequenceImplCFPvT1    @1001 NONAME
      ;ILinkedSequenceImpl::isConsistent() const
      isConsistent__19ILinkedSequenceImplCFv    @1002 NONAME
      ;ILinkedSequenceImpl::setToNext(ILinkedSequenceImpl::Node*&) const
      setToNext__19ILinkedSequenceImplCFRPQ2_19ILinkedSequenceImpl4Node    @1003 NONAME
      ;ILinkedSequenceImpl::addAsNext(ILinkedSequenceImpl::Node*,ILinkedSequenceImpl::Node*)
      addAsNext__19ILinkedSequenceImplFPQ2_19ILinkedSequenceImpl4NodeT1    @1004 NONAME
  ;From object file:  C:\ibmcpp\src\IITBSEQ.C
    ;PUBDEFs (Symbols available from object file):
      ;ITabularSequenceImpl::setToPrevious(ITabularSequenceImpl::Cursor&) const
      setToPrevious__20ITabularSequenceImplCFRQ2_20ITabularSequenceImpl6Cursor    @1034 NONAME
      ;ITabularSequenceImpl::allElementsDo(void*)
      allElementsDo__20ITabularSequenceImplFPv    @1035 NONAME
      ;ITabularSequenceImpl::removeAll(void*,void*)
      removeAll__20ITabularSequenceImplFPvT1    @1036 NONAME
      ;ITabularSequenceImpl::addAllFrom(const ITabularSequenceImpl&)
      addAllFrom__20ITabularSequenceImplFRC20ITabularSequenceImpl    @1037 NONAME
  ;From object file:  IIAVLKSS.C
    ;PUBDEFs (Symbols available from object file):
      ;IAvlKeySortedSetImpl::allElementsDo(void*,void*) const
      allElementsDo__20IAvlKeySortedSetImplCFPvT1    @1080 NONAME
      ;IAvlKeySortedSetImpl::isFirst(const IAvlKeySortedSetImpl::Node*) const
      isFirst__20IAvlKeySortedSetImplCFPCQ2_20IAvlKeySortedSetImpl4Node    @1081 NONAME
      ;IAvlKeySortedSetImpl::setToPosition(unsigned long,IAvlKeySortedSetImpl::Node*&) const
      setToPosition__20IAvlKeySortedSetImplCFUlRPQ2_20IAvlKeySortedSetImpl4Node    @1082 NONAME
      ;IAvlKeySortedSetImpl::locateOrAddElementWithKey(const void*)
      locateOrAddElementWithKey__20IAvlKeySortedSetImplFPCv    @1083 NONAME
```

Note: This is only a partial listing of the actual output.

Binary Mode Options

In binary mode, the CPPFILT utility has the following options:

/B	Operate in Binary mode
/H or /?	Display help
/N	Reference exported names by ordinal number
/O[*ord*]	Generate ordinals after each demangled name
/P	Include public symbols in output
/Q	Suppress display of logo
/R	Include referenced symbols in output
/S	Demangle special compiler-generated symbol names
/X	Include exported symbols in output

Note: You can specify options using the slash form (/R) or the dash form (-R).

/B (Operate in Binary Mode)

Syntax:	**Default:**
`/B`	Operate in Text mode

Specify `/B` on the CPPFILT command line to instruct CPPFILT to operate in Binary mode.

If you do not specify the `/B` option, CPPFILT will operate in the default Text mode.

/H (Help)

Syntax:	**Default:**
`/H` `/?`	None

Specify the `/H` or `/?` option on the CPPFILT command line to see a short online help on the `cppfilt` command syntax and options.

If you do not specify this option, the default is not to display any online help.

/N (NONAME Keyword)

Syntax:	**Default:**
`/N`	Reference exported names by name

Specify the `/N` option on the CPPFILT command line to instruct CPPFILT to generate the `NONAME` keyword as consistent with the module definition (.DEF) file EXPORTS statement syntax. The `NONAME` keyword indicates that the exported names should be referenced by their ordinals, and not by their names.

Note: You should also specify `/N` together with the `/O` option to generate ordinals.

For example, if you specify `/N /O 1000`, the output for a single name would look something like this:

```
;ILinkedSequenceImpl::isConsistent() const
isConsistent__19ILinkedSequenceImplCFv    @1000 NONAME
```

For more information on using module definition files, see Chapter 21, “Creating Module Definition Files” on page 369.

/O (Ordinals)

Syntax:	**Default:**
`/O`	Do not generate ordinals

Specify the `/O[`*`ord`*`]:` option on the CPPFILT command line to generate ordinals after each demangled name. If the optional numeric parameter *`ord`* is specified, CPPFILT will generate the ordinals starting from *`ord`*.

The ordinals are generated along with the @ character as consistent with the module definition (.DEF) file EXPORTS statement syntax.

For example, if you specify `/O 1000`, the output for a single name would look something like this:

```
;ILinkedSequenceImpl::isConsistent() const
isConsistent__19ILinkedSequenceImplCFv    @1000
```

If you do not specify the `/O` option, the default is not to generate ordinals after each demangled name.

For more information on using module definition files, see Chapter 21, “Creating Module Definition Files” on page 369.

/P (Public Symbols)

Syntax:	**Default:**
`/P`	Do not include public symbols in output

Specify the `/P` option on the CPPFILT command line to include all public (PUBDEF, COMDAT, COMDEF) symbols in the output.

Note: Only the first occurrence of a symbol found within the COMDAT sections will be included in the output. Subsequent occurrences of the same COMDAT symbol will appear as comments in the output.

If you do not specify this option, the default is not to include public symbols in the output.

Note: If you do not specify any of the /X, /R, or /P options, the Binary mode output will include only the library and object names, without any symbol names.

/Q (Do Not Display Logo)

Syntax:	**Default:**
`/Q`	Display logo and copyright notice

Specify the `/Q` option on the CPPFILT command line to suppress the display of the logo and copyright notice for the CPPFILT utility.

If you do not specify this option, the default is to display the logo and copyright notice.

/R (Referenced Symbols)

Syntax:	**Default:**
`/R`	Do not include referenced symbols in output

Specify the `/R` option on the CPPFILT command line to include all referenced (EXTDEF) symbols in the output.

If you do not specify this option, the default is not to include referenced symbols in the output.

Note: If you do not specify any of the /X, /R, or /P options, the binary-mode output will include only the library and object names, without any symbol names.

/S (Special Symbol Names)

Syntax:	**Default:**
`/S`	Do not demangle special compiler-generated symbol names

Specify the `/S` option on the CPPFILT command line to instruct the CPPFILT utility to demangle special compiler-generated symbol names.

For example, the compiler-generated symbol name, `__ct__3FooFUi`, that represents a constructor for the class `Foo` would be demangled as `Foo::Foo(unsigned int)`.

If you do not specify the `/S` option, the default is not to demangle special compiler-generated symbol names.

/X (Exported Symbols)

Syntax:	**Default:**
`/X`	Do not include exported symbols in output

Specify the `/X` option on the CPPFILT command line to include all exported (EXPDEF) symbols in the output.

If you do not specify `/X`, the default is not to include exported symbols in the output.

Note: If you do not specify any of the /X, /R, or /P options, the Binary mode output will include only the library and object names, without any symbol names.

65

Using EXEHDR

The Executable File Header Utility (EXEHDR) displays and modifies the contents of an executable-file header. EXEHDR generates an Output listing (📖 see page 890) showing the contents of the file header and information about each object or segment in the file. EXEHDR Options are provided (📖 see page 886) that let you change values in the file header.

Uses of EXEHDR include:

- Determining whether a file is an application or a dynamic link library
- Viewing and changing the attributes set by the module definition file
- Viewing the number and size of code and data segments.

You can use EXEHDR with DOS or OS/2 applications and dynamic-link libraries.

EXEHDR Syntax

```
EXEHDR [options] filename
```

<options>
Options used to modify Output or change the file header.

<filename>
One or more names of applications or dynamic-link library files.

Regardless of options, EXEHDR always creates an Output listing of the file header.

Displaying Help

To display EXEHDR help, type `EXEHDR /?` at the command prompt. The appropriate copyright statement appears along with a brief list of EXEHDR options.

Changing Executable Headers with EXEHDR

```
Usage: EXEHDR [options] filename...
Valid options are:
  /?
  /HEAP:(0H - ffffH)
  /HELP
  /MAX:(0H - ffffH)
  /MIN:(0H - ffffH)
  /NEWFILES
  /NOLOGO
  /PMTYPE:(PM | VIO | NOVIO |
               WINDOWAPI | WINDOWCOMPAT |
               NOTWINDOWCOMPAT)
  /RESETERROR
  /STACK:(0H - ffffH)
  /VERBOSE
```

EXEHDR Options

Usage Notes:

- Option characters are not case sensitive: /R and /r are equivalent.
- Options can be shortened to the fewest characters that uniquely identify them. The characters in brackets can be omitted: /N and /NOLOGO are equivalent.
- Although use of the minimum one-letter abbreviations is allowed, if a future release has an additional option starting with the same letter, the one-letter option will no longer be usable.

Formats Affected by Options

The EXEHDR options that can change executable files are MIN, MAX, STACK, PMTYPE, HEAP, RESETERROR, and NEWFILES.

Executable headers are used by the operating system to determine characteristics of the executable file, such as stack size, entry point, number of objects (or segments), and so on. EXEHDR recognizes three different executable file formats: DOS, OS/2 16-bit, and OS/2 32-bit.

Note: VisualAge C++ generates only OS/2 32-bit executable files.

An X in the following table indicates which options change each type of executable:

Option	DOS	OS/2 16-bit	OS/2 32-bit
HEAP		X	X
MAX	X		
MIN	X		
NEWFILES		X	
PMTYPE		X	X
RESETERROR		X	X
STACK	X	X	X

For compatibility purposes, executable files generated by the VisualAge C++ linker include both a DOS header and an OS/2 header.

/HEA[P]

Set Heap Allocation (/HEAP)

Syntax: `/HEA[P]:nnnn`

This option sets the size of the local heap and is applicable to OS/2 applications only. The field <nnnn> contains the local heap size in bytes.

You can specify <nnnn> in decimal, octal, or hexadecimal radix using standard C language notation.

Note: For 32-bit OS/2 applications, the loader ignores this option.

/HEL[P] and /?

Display Help (/HELP or /?)

Syntax: `/HEL[P]` OR `/?`

This option displays a brief summary of EXEHDR syntax.

/MA[X]

Set Maximum Allocation (/MAX)

Syntax: `/MA[X]:nnnn`

This option sets the maximum allocation of memory for the program. The field <nnnn> contains the maximum number of 16-byte paragraphs required to load and run the program. This value must be equal to or greater than the minimum allocation.

Compare to “/MI[N]” on page 888 .

You can specify <nnnn> in decimal, octal, or hexadecimal radix using standard C language notation.

/MI[N]

Set Minimum Allocation (/MIN)

Syntax: `/MI[N]:nnnn`

This option sets the minimum allocation of memory for the program. The field <nnnn> contains the minimum number of 16-byte paragraphs required to load and run the program. This value must be equal to or less than the maximum allocation.

Compare to “/MA[X]” on page 887 .

You can specify <nnnn> in decimal, octal, or hexadecimal radix using standard C language notation.

/NE[WFILES]

New Files (/NEWFILES)

Syntax: `/NE[WFILES]`

This option enables long file name support for OS/2 16-bit executable files. (OS/2 32-bit executable files generated by the VisualAge C++ linker always support long file names.)

/NO[LOGO]

Suppress Sign-On Banner (/NOLOGO)

Syntax: `/NO[LOGO]`

This option suppresses the sign-on banner displayed by EXEHDR when it starts.

/P[MTYPE]

Set Application Type (/PMTYPE)

Syntax: `/P[MTYPE]:type`

This option specifies the type of application. It pertains only to OS/2 applications. The /PMTYPE option in EXEHDR is equivalent to either the NAME statement in the module-definition file (see “NAME” on page 384) or the /PMTYPE option in the VisualAge C++ linker (see “/PMTYPE” on page 363).

A keyword in <type> is equivalent to a keyword in a NAME statement, as shown in the following list:

Field Keyword	Equiv. Keyword
PM	WINDOWAPI
VIO	WINDOWCOMPAT
NOVIO	NOTWINDOWCOMPAT

The NAME statement keyword is also accepted.

/R[ESETERROR]

Reset Linker Error (/RESETERROR)

Syntax: `/R[ESETERROR]`

This option clears an error flag stored in OS/2 applications. The error flag is set by the linker when the link has unresolved external references or duplicate symbol definitions (any linker error messages starting with L2xxx).

OS/2 does not load the application if the error flag is set. This option allows you to attempt to run a program with VisualAge C++ linker errors and is useful during application development.

/S[TACK]

Set Stack Allocation (/STACK)

Syntax: `/S[TACK]:nnnn`

This option sets the size of the stack. The field <nnnn> contains the stack size in bytes.

You can specify <nnnn> in decimal, octal, or hexadecimal radix using standard C language notation.

/V[ERBOSE]

Display in Verbose Mode (/VERBOSE)

Syntax: `/V[ERBOSE]`

This option displays the executable-file header in verbose mode.

Output

EXEHDR lists the current contents of the file header and information about each object (or segment) in the file. To redirect this output to a printer or disk file, use the operating system redirection operator.

The output is in two parts: a “Header Listing” giving the contents of the file header; and an “Object or Segment Listing” on page 891 giving attributes of all objects (or segments) in the file. If the /VERBOSE option is specified, additional output is generated.

Header Listing

The header listing is comprised of the following fields:

<Module> Name of Application

This field lists the name of the application as specified in the NAME statement of the module-definition file.

If no module definition was used to create the executable file, this field displays the name assumed by default.

If a module definition was used to create the file, but the LIBRARY statement appeared instead of the NAME statement (thus specifying a dynamic-link library), the name of the library is given and EXEHDR uses the word "Library" instead of "Module" to identify the field.

<Description> Description of Application

This field gives the contents, if any, of the DESCRIPTION statement of the module-definition file used to create the file being examined.

<Data> Type of Automatic Data Object

This field indicates the type of automatic data segment in a program: SHARED, NONSHARED, or NONE. This type can be specified in a module-definition file. The defaults are NONSHARED for applications and SHARED for dynamic-link libraries.

<Initial CS:IP> Program Starting Address

This field gives the program starting address (if an application is being examined) or address of the initialization routine (if a dynamic-link library is being examined).

<Initial SS:SP> Initial Stack Pointer

This field gives the value of the initial stack pointer.

<Extra Stack Allocation> Additional stack allocation

This field gives the value of the extra stack location.

<DGROUP> Automatic-Data-Object Number

Object or Segment Listing

The object listing is comprised of the following fields:

no. Object index number, starting with 1, in decimal

type Identification of the object as a code or data object

A code object is comprised of segments with class name ending in `CODE`. All other objects are data objects.

address Location, within the file, of the contents of the object (in hexadecimal)

file Size of the object (in bytes), as contained in the file (in hexadecimal)

mem Size of the object (in bytes), as it is stored in memory (in hexadecimal)

If the value of this field is greater than the value of <file>, the operating system pads the additional space with zero values at load time.

flags Object attributes

If the /VERBOSE option is not used, only nondefault attributes are listed. Attributes are given in the form specified in the module-definition file.

Output Example

The following output is generated by EXEHDR for the executable file IMPLIB.EXE:

```
Module:                  IMPLIB
Description:             Operating System/2 Import Library Manager
Data:                    NONSHARED
Initial CS:IP:           seg   1 offset 234c
Initial SS:SP:           seg   4 offset 0000
Extra stack allocation:  0a00 bytes
DGROUP:                  seg   4

no. type address  file  mem   flags
  1 CODE 00003400 045f2 045f2
  2 DATA 00007c00 00374 00374
  3 DATA 00000000 00000 02d3a
  4 DATA 00008000 01635 01680
```

Verbose Output

When you specify the /VERBOSE option, EXEHDR generates additional output:

- DOS-specific header information. All OS/2 executable files have a DOS header, whether bound or not. If the program is not bound, the DOS portion typically consists of a stub that simply terminates the program.
- OS/2-specific header information. The object-table display in verbose mode is described below.
- File addresses and lengths of the various tables in the executable file. For each table, the following is generated:
  - Name of the table
  - Address of the table within the file
  - Length of the table in hexadecimal radix
  - Length of the table in decimal radix
- Object table with complete attributes, not just the nondefault attributes. The /VERBOSE option displays two additional attributes:
  - The RELOCS attribute is displayed for each object that has address relocations. Relocations occur in each object that references objects in other objects or makes dynamic-link references.
  - The ITERATED attribute is displayed for each object that has iterated data. Iterated data consist of a special code that packs repeated bytes.
- Run-time relocations and fixups.
- All exported entry points.

Setting Program Type with MARKEXE

The MARKEXE program enables you to view and set the program type for an executable file. The program type identifies the OS/2 sessions in which a program can run.

Use MARKEXE with the VisualAge C++ linker to change or set the program type of programs you have created.

You can set DLL initialization and termination. If you are using a 16-bit linker, MARKEXE can also enable long file name support for 16-bit executables.

Note: VisualAge C++ generates only 32-bit executables, which include long file name support.

Command-Line Syntax

MARKEXE uses the following syntax:

```
MARKEXE [/?] [FORCE] [NO] [options] filename...
```

Filename is a file name or a list of file names. Global file-name characters (*.EXE) also can be used. For descriptions of the above terms, 🕮 see “Syntax Definitions” on page 894. If no option is given, DISPLAY is assumed.

Typing MARKEXE /? at the command line displays the appropriate copyright statement along with a list of options.

```
  DISPLAY         - display status of flags
  DLLINIT         - per-process initialization
  DLLTERM         - per-process termination
  WINDOWAPI       - window api (PM application)
  WINDOWCOMPAT    - window compatible application
  NOTWINDOWCOMPAT - not window compatible application
  UNSPECIFIED     - unspecified application type
  LFNS            - long file name support
```

Setting Executable Type with MARKEXE

Syntax Definitions

MARKEXE has the following keywords, options, and program types. You can also specify any number of files to be viewed or marked.

KEYWORDS

FORCE Marks the executable file with OS/2 as the target operating system even though the file was marked for another operating system. Using FORCE might produce internally inconsistent executable files.

NO Sets the command to the opposite condition. This keyword does not apply to the DISPLAY, UNSPECIFIED, or WINDOWAPI options.

OPTIONS

DISPLAY Displays the application type in a message; does not make any changes.

DLLINIT Sets per process initialization for the dynamic link library. (Use with VisualAge C++ linker and other 32-bit linkers.)

DLLTERM Sets per process termination for the dynamic link library. (Use with 16-bit linkers only.)

LFNS Enables support of long file names. (Use with 16-bit linkers only.)

PROGRAM TYPES

MARKEXE does not modify the file if the executable file's program type is the same as the requested type. It displays a message instead.

WINDOWAPI The application is a Presentation Manager application and can run in the Presentation Manager session only.

WINDOWCOMPAT The application can run in a Presentation Manager window or in an full-screen session.

NOTWINDOWCOMPAT
The application must run in an OS/2 full-screen session.

UNSPECIFIED The application type is not known. By default, the OS/2 operating system will force the program to run in a full-screen session.

Note: Specifying an incorrect program type might cause undesirable results when you try to run that program. For example, do not change a WINDOWCOMPAT program to WINDOWAPI.

Viewing Program Type

To display the program type of an executable file without changing the file, specify only a file name, omitting an option.

```
MARKEXE filename.exe
```

Example

To view the program type of MYPROG.EXE, type the following:

```
MARKEXE myprog.exe
```

MARKEXE displays the type in a message that looks like this:

```
myprog.exe: OS/2 1.x, WINDOWCOMPAT, LFNS
```

Setting Program Type

To set the program type of an executable file, specify one of the program types. More than one executable file can be set to the same program type on a single command line.

```
MARKEXE type filename.exe another.exe
```

Examples

To set WINDOWCOMPAT as the program type of MYPROG.EXE, type:

```
MARKEXE WINDOWCOMPAT myprog.exe
```

To set WINDOWAPI as the program type of several executable files, type:

```
MARKEXE WINDOWAPI marion.exe alex.exe
```

Setting Executable Type with MARKEXE

Creating Symbolic Debugger Files with MAPSYM

The MAPSYM program creates .SYM files from .MAP files. .SYM files are used by the kernel debugger for symbolic debugging.

Note: You must run MAPSYM from the directory in which the file to be converted is located.

To create a .SYM file, follow these steps:

1. Make sure you are in the correct directory.
2. At the prompt type the following:

   ```
   mapsym filename
   ```

   Note that the .MAP extension is not required.

Displaying Help

To display MAPSYM help, type *MAPSYM* at the prompt, with no arguments. The appropriate copyright statement appears, along with the following:

usage: mapsym [-aln] mapfile

MAPSYM Options

You can use the following options with MAPSYM:

/A Omits Alphabetical sorting of symbols.

/N Includes source code line Numbers in *.SYM file.

/L Produces verbose Listing.

Creating Symbolic Debugger Files with MAPSYM

68

Creating Workplace Object Classes

The Workplace Class List is a tool that creates a workplace object class and an instance of a workplace object class. Workplace objects are constructed using the SOM protocol and are instances of one of the following workplace object classes:

Predefined These classes are defined by the system. Examples of predefined workplace object classes are WPObject, WPFileSys, and WPAbstract.

Subclass These classes are derived from existing predefined workplace object classes. They add or remove function; however, they retain the basic behavior of that class.

Replaced These classes replace the class being *subclassed*. They modify the behavior of an instance of a predefined workplace object class without the instance being aware of the new class.

Starting Workplace Class List

To start Workplace Class List, select the **PM Development Tools** folder, and then select **Workplace Class List**. A window appears. The window contains a list of the workplace object classes currently registered in the OS/2 Workplace Shell. Using the window, you can:

- Create an instance of a workplace object class
- Replace a workplace object class
- Unreplace a workplace object class
- Add a workplace object class
- Delete a workplace object class

Creating an Object Class Instance

To create an instance of a workplace object class:

1. Select the class from the list in the Workplace Object Class window.
2. Display the pop-up menu by clicking mouse button 2.
3. Select the **Create Instance** choice.

Note: Only an instance of a physical workplace object class can be created. In other words, you cannot create instances of WPObject or WPClass because they are not physical classes.

4. Fill in the following input fields:

Object Title The text string you assign to the object. The text string becomes the object title and appears under the object when the object is displayed on the Workplace Shell. When the object is in an opened state, the text string appears in the title bar of the window.

Class of new object The name of the class of which the object you selected is a member.

Parameters A series of **keyname=value** pairs (separated by semicolons) that change the behavior of the object. Each object class defines the keynames and parameters it expects to see. All parameters have safe defaults, so it is never required to pass parameters to an object.

Location A real name specified by a fully qualified file specification, such as C:\OS2\DLL\MINXOBJ.DLL, or a logical name specified by a predefined symbol.

Examples of logical names include the following:

LOCATION_NOWHERE	Hidden folder
LOCATION_DESKTOP	OS/2 Desktop (Workplace)
LOCATION_SYSTEM	OS/2 System folder
LOCATION_TEMPLATES	Template folder
LOCATION_SYSTEMSETUP	System setup folder
LOCATION_STARTUP	Startup folder
LOCATION_INFORMATION	Information folder
LOCATION_INFORMATION	Information folder
LOCATION_DRIVES	Drives folder

Replacing a Workplace Object Class

To replace an existing registered class:

1. Select the class from the list in the Workplace Object Class window.
2. Display the pop-up menu by single-clicking mouse button 2.
3. Select the **Replace** choice. Note that only classes that have already been registered are valid.
4. Fill-in the following input fields: Original class and Replacement class.

Note: The replacement class must be a descendant of the original class. Replacing an object class is useful for modifying the behavior of objects which are instances of the original class but are not aware of the replacement class.

Original class The name of the object class being replaced in the Workplace.

Replacement class The name of the object class replacing the original class.

Unreplaceing a Workplace Object Class

To return a replaced class to its original class:

1. Select the replaced class from the list on the workplace object class window.
2. Display the pop-up menu by clicking mouse button 2.
3. Select the **Unreplace** choice. Note that only classes that have already been replaced are valid.
4. Fill-in the following input fields:

Original class The name of the replaced object class being returned to its original object class in the Workplace.

Replacement class The name of the replaced object class being returned to its original object class.

Adding a Workplace Object Class

To add a class to the Workplace Shell:

1. Display the pop-up menu by clicking mouse button 2.
2. Select the **Add Class** choice.
3. Fill-in the following input fields:

New class name The name of object class you want to add to the Workplace. Type the class name exactly as it is built, case-sensitive.

Library module The name of the dynamic link library (DLL) that holds the object definition. Type the library name with complete file specification information.

Note: The DLL must be created by the IBM System Object Model.

Deleting an Object Class

To delete a class from the Workplace Shell:

1. Select the class you want to delete from the list in the Workplace Object Class window.
2. Display the pop-up menu by clicking mouse button 2.
3. Select the **Delete a Class** choice.
4. Fill-in the name of the class you want to delete from the Workplace.

Note: You cannot delete system predefined classes, such as WPObject or WPClass.

69 Registering Workplace Objects with Object Utility/2

Object Utility/2 provides a facility for registering Workplace Shell classes, creating instances of Workplace Shell classes, and modifying instances of Workplace Shell classes.

The following attributes can be set or modified for instances of Workplace Shell objects:

- Template
- Copy
- Delete
- Rename
- Print
- Link
- Move
- Drag

The attributes modify the behavior of the objects to allow or not allow the above actions. For example, the Template attribute allows you to create a template. Some objects do not allow specific behaviors even if the attribute is selected.

A Workplace Shell Class must be registered with the Workplace Shell before it will be recognized by Object Utility/2. After the object class is registered, an instance of that class can be created. The Object Utility/2 automates these procedures of object class registration and instantiation. This tool can create an instance of an object from a class that has already been instantiated or can modify an existing instance.

Registration of a class is performed by opening the main view of Object Utility/2. The class name and DLL name must be provided. The class is not registered if it has been registered previously.

To modify an existing instance, the icon representing the class is dragged to and dropped on top of Object Utility/2. You can enter the object ID and class name after opening the main view.

After the item to be installed is dropped, a dialog box is displayed to obtain registration and instantiation information.

To destroy an object created by this tool, the object can be dragged and dropped onto the shredder object on the Workplace Shell desktop (if the **no drag** and **no delete** options are not selected and the object allows deletion). A mechanism to deregister an object class is not provided with this tool.

Class Name

Class name is a list of all the registered classes that have DLLs available on your system. OS/2 allows classes to be registered without the DLLs available, but Object Utility/2 does not. You can select a class from the list or enter one manually. This field is required when registering a new class, modifying an existing instance that was not dropped on Object Utility/2, or creating a new instance.

DLL Name

The DLL name must be a fully qualified path and file name if the DLL is not located in a DLL search path. This field is required if you are registering a class.

Object ID

The Object ID must be enclosed in angle brackets(<>). This field is required when you modify an existing object that was not dropped on Object Utility/2. You are warned if you try to create an instance that is not a template, without an Object ID. You may create the new instance without an object ID. The Object ID must be unique, if specified, when creating an instance. Templates cannot have an object ID. Instances with an object ID cannot be made into a template.

Title Field

The Title field is required when creating a new object. You can alter the title of an existing object by providing a different title in this field.

Location Field

You can select an existing location from the location list or enter a location manually. The location must be an object ID that represents a folder (enclosed in angle brackets) or a fully qualified path name.

Options

Create Instance Creates an instance of the class.

Template Creates a template of the class in the Templates folder.

No Copy Removes **Copy** from the pop-up menu.

No delete Removes **Delete** from the pop-up menu.

No Rename Removes **Rename** from the pop-up menu.

No Print Removes **Print** from the pop-up menu.

No Link Removes **Link** from the pop-up menu.

No Move Removes **Move** from the pop-up menu.

No drag Prevents dragging of the object.

Registering Workplace Objects with Object Utility/2

Using the T Terminal Emulator

The Kernel Debugger uses the T Terminal Emulator to communicate with the machine to be debugged, also known as the MUT (Machine Under Test).

You can also use T to send and receive ASCII files.

All functions of T are listed on the Help menu. Press F1 to view the Help menu, which is shown below.

```
  TERMINAL - OS/2 ASCII Terminal Program

  Version 2.00.00

 F1 or ALT-H   Help
 F2            Terminal Setup
 F3            Sending ASCII Files
 F4            Pausing and Scrolling
 F6            Receiving ASCII Files
 F8  or ALT-X  Exit terminal program
```

Command-Line Syntax

To display help for command-line syntax, type `T -?` at the prompt.

```
OS/2 Terminal program
Version 2.00.00
November 1, 1991

Valid command line switches:
  -L[ines]X      X={lines}
  -C[om]N        N={1..8}
  -Q[uiet]       enter quiet mode
  -V[tp]:name    name=vtp server
  -S[end]:name   name=auto-send file name
  -R[emark]:text text=status line remark (20 chars max)
```

Command-Line Options

Use command line options when invoking T to specify the screen size and the COM port to be used:

`-L[ines]X`	Where *x* = {25, 43, 50}
`-C[om]N`	Where *n* = {1,2,3}
`-?`	To display these options

Terminal Setup

All terminal setup functions are listed here. Press F2 from the program's main screen to view the Terminal Setup menu.

The **Terminal Setup Options** are as follows:

- Help (press F1)
- Port setup (Press F2)
- Terminal emulation (Press F3)
- Keyboard macros (Press F4)
- Bells & Whistles (Press F5)
- Exit setup mode (Press Esc or F8)

You must press Esc to return to the main screen before continuing with any functions other than the above setup functions.

Setup Terminal Emulation

To set up terminal emulation, follow these steps:

1. Press F2 at the main screen. The Terminal Setup dialog will be displayed.
2. Press F3. The Terminal Emulation Setup dialog will be displayed.

```
Terminal Emulation Setup
Emulator status: None loaded.

Help:                      F1
Z19 emulator:              F2

Exit emulator setup mode:  Esc, F8
```

Setup Bells & Whistles

To change bells and whistles, follow these steps:

1. Press F2 at the main screen. The Terminal Setup dialog will be displayed.
2. Press F5. The Bells & Whistles Setup dialog will be displayed.
3. Make your selections.

```
Bells and Whistles Setup

Filter NULL characters: Yes
Disable beeps:          No

                      Background:   Foreground:
Normal screen:        Black         White
Status line:          Blue          Bright White
Scroll screen:        Blue          Red
Scroll status line:   White         Blue
Help screen:          Blue          Bright White
Menu:                 Blue          Bright White
Menu highlight:       Cyan          Black

Next Value:
```

Setting Communications Parameters

To change communications parameters, follow these steps:

1. Press F2 at the main screen. The Terminal Setup dialog will be displayed.
2. Press F2. The Current COM2 Port Parameters dialog will be displayed.

   ```
   Current COM2 Port parameters:

   Baud Rate:                  9600
   Parity:                     NONE
   Data Bits:                     8
   Stop Bits:                     1
   Write Timeout (sec.):       1.00
   Read Timeout (sec.):        0.10
   Handshaking:            XON/XOFF

   Next Value: ->   Previous Value: <-
   Next Field: Dn   Previous Field: Up
   Don't Change:   Esc
   Accept Changes: Enter
   ```

3. Use the Up Arrow and Down Arrow cursor keys to scroll backward and forward through the parameter list, and the -> and <- keys to scroll through allowable values for each parameter.

4. Press Escape to exit this dialog without changing values, or press Enter to save these values and exit the dialog.

Note: The communications port may be changed from the command line by using the `-c` option. 📖 See “Command-Line Options” on page 908

Sending ASCII Files

To send an ASCII file, follow these steps:

1. Press F3. The Send File Control dialog will be displayed.

```
 Send File Control

 Send file name:
 SEND.TXT

 Don't send file:               Esc
 Accept changes and send file:  Enter
```

2. Enter the filename in the Send File name field.

3. Press Enter to send the file and exit the dialog, or press Escape to exit this dialog without sending a file.

Pausing and Scrolling

To enter Scroll Mode and pause display of communications, follow these steps:

1. Press F4. A status line will appear at the bottom of the screen.

```
F1=Help  ESC=Active mode  Screen Top is 100%
through the buffer.  Scroll Mode
```

2. Press F1 to display the Scroll Mode commands.

```
 Screen Scroll Mode

 F1 or ALT-H       Help
 Dn                Down a line
 Up                Up a line
 PgUp              Up a page
 PgDn              Down a page
 ESC or Enter      Return to active mode
```

3. Select a scroll mode command.

Receiving ASCII Files

To receive an ASCII file, follow these steps:

1. Press F6. The Capture File Control dialog will be displayed.

   ```
   Capture File Control

   Capture file name:
   Capture.Txt

   Capture entire buffer: F3
   START Capture:         F5
   Delete file:           F9
   Don't Change:          Esc
   Accept Changes:        Enter
   ```

2. Select an item from the **Capture File Control** menu.

Using the T Terminal Emulator

Glossary

This glossary defines terms and abbreviations that are used in this book. Included are terms and definitions from the following sources:

- *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Such definitions are indicated by the symbol *ANSI* after the definition.
- *IBM Dictionary of Computing*, SC20-1699. These definitions are indicated by the registered trademark *IBM* after the definition.
- *X/Open CAE Specification. Commands and Utilities, Issue 4. July, 1992*. These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990*. These definitions are indicated by the symbol *ISO.1* after the definition.
- *The Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

abstract class. (1) A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot have a direct object of an abstract class. See also *base class*. (2) A class that allows polymorphism. There can be no objects of an abstract class; they are only used to derive new classes.

abstraction (data). A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration.

access specifier. One of the C++ keywords: public, private, and protected, used to define the access to a member.

action. A description of tool or function that can be used to manipulate a project's parts, or build a project's target. Examples are Compile, Link, and Edit.

action class. A grouping of action that perform a similar function

actions support DLL. A dynamic link library that provides such support for an action as determining dependencies and targets if the action is to participate in a build, providing a user interface for setting options, and integration with the monitor and editor.

additional heap. (1) A *Language Environment* heap created and controlled by a call to CEECRHP. See also *below heap, anywhere heap*, and *initial heap*.

address space. (1) The range of addresses available to a computer program. *ANSI*. (2) The complete range of addresses that are available to a programmer. See also *virtual address space*.
(3) In the AIX operating system, the code, stack, and data that are accessible by a process.
(4) The area of virtual storage available for a

particular job. (5) The memory locations that can be referenced by a process. *X/Open*. *ISO.1*.

aggregate. (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) An array or a class object with no private or protected members, no constructors, no base classes, and no virtual functions. (4) In programming languages, a structured collection of data items that form a data type. *ISO-JTC1*.

alert. (1) A message sent to a management services focal point in a network to identify a problem or an impending problem. *IBM*. (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). *X/Open*.

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM*.

American National Standards Institute. See *ANSI*.

angle brackets. The characters < (left angle bracket) and > (right angle bracket). When used in the phrase “enclosed in angle brackets,” the symbol < immediately precedes the object to be enclosed, and > immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used. *X/Open*.

ANSI (American National Standards Institute). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *ANSI*.

anywhere heap. The VisualAge C++ heap controlled by the ANYHEAP run-time option. It contains library data, such as VisualAge C++ control blocks and data structures not normally accessible from user code. The anywhere heap may reside above 16M. See also *below heap, additional heap, initial heap*.

application. (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM*. (2) A collection of software components used to perform specific types of user-oriented work on a computer. *IBM*.

application program. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM*.

argument. (1) A parameter passed between a calling program and a called program. *IBM*. (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter*. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open*.

array. In programming languages, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. *IBM*.

array element. A data item in an array. *IBM*.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM*.

Note: IBM has defined an extension to ASCII code (characters 128-255).

assembler user exit. In the *Language Environment* a routine to tailor the characteristics of an enclave prior to its establishment.

automatic data. Data that does not persist after a routine has finished executing. Automatic data may be automatically initialized to a certain value upon entry and reentry to a routine.

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

B

backslash. The character \. This character is named <backslash> in the portable character set.

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

base project. A project from which another project inherits its **Tools setup**. Distinguished from *parent project*. from.

based on. The use of existing classes for implementing new classes.

below heap. The VisualAge C++ heap controlled by the BELOWHEAP runtime option, which contains library data, such as VisualAge C++ control block and data structures not normally accessible from user code. Below heap always resides below 16M. See also *anywhere heap, initial heap, additional heap*.

binary stream. (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM*.

bit field. A member of a structure or union that contains a specified number of bits. *IBM*.

block. (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

brackets. The characters [(left bracket) and] (right bracket), also known as *square brackets*. When used in the phrase “enclosed in (square) brackets” the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

breakpoint. A point in a computer program where execution may be halted. A breakpoint is usually at the beginning of an instruction where halts, caused by external intervention, are convenient for resuming execution. *ISO Draft*.

build. An action that invokes the WorkFrame Build tool. The Build tool manages the project's make file, as well as build dependencies between projects in a project hierarchy.

build actions. A series of actions that are invoked to build a project's target. These actions are set in the Build options window, or in MakeMake, WorkFrame's make file creation utility.

built-in. (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with predefined. *IBM*.

C

C++ class library. See *class library*.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

call. To transfer control to a procedure, program, routine, or subroutine. *IBM*.

caller. A routine that calls another routine.

carriage-return character. A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by '\r' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line. *X/Open*.

CASE (Computer-Aided Software Engineering). A set of tools or programs to help develop complex applications. *IBM*.

cast. In the C and C++ languages, an expression that converts the type of the operand to a specified data type (the operator). *IBM*.

character. (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI*. (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of the multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open*. *ISO.1*.

character array. An array of type char. *X/Open*.

character class. A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. *X/Open*.

character constant. (1) A constant with a character value. *IBM*. (2) A string of any of the characters that can be represented, usually enclosed in apostrophes. *IBM*. (3) In some languages, a character enclosed in apostrophes. *IBM*.

character set. (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft*. (2) All the valid characters for a programming language or for a computer system. *IBM*. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM*.
(4) See also *portable character set*.

character string. A contiguous sequence of characters terminated by and including the first null byte. *X/Open*.

child. A node that is subordinate to another node in a tree structure. Only the root node is not a child.

class. (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be derived from another, and may restrict access to its members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

class library. A collection of C++ classes.

class name. A unique identifier of a class type that becomes a reserved word within its scope.

class template. A blueprint describing how a set of related classes can be constructed.

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

client program. A program that uses a class. The program is said to be a *client* of the class.

COBOL (Common Business-Oriented Language). A high-level language, based on English, that is primarily used for business applications.

coded character set. (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM*. (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft*.
(3) Loosely, a code. *ANSI*.

code page. (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code.

(2) A particular assignment of hexadecimal identifiers to graphic characters.

code point. (1) A 1-byte code representing one of 256 potential characters. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

codeset. Synonym for code element set. *IBM*.

collating element. The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. *X/Open*.

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI*. (3) The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

collation. The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting or multiple collating elements. *X/Open*.

collection. (1) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection. (2) In a general sense, an implementation of an abstract data type for storing elements.

Collection Class Library. A set of classes that provide basic functions for collections, and can be used as base classes.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

compilation unit. (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM*. (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

condition. (1) A relational expression that can be evaluated to a value of either true or false. *IBM*. (2) An exception that has been enabled, or recognized, by the *Language Environment* and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

constant. (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1*. (2) A data item with a value that does not change. *IBM*.

constant expression. An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM*.

constructor. A special C++ class member function that has the same name as the class and is used to create an object of that class.

control character. (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft*. (2) Synonymous with nonprinting character. *IBM*. (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open*.

conversion. (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1*. (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM*. (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM*. (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

coordinated universal time (UTC). Equivalent to Greenwich Mean Time (GMT)

copy constructor. A constructor that copies a class object of the same class type.

current working directory. (1) A directory, associated with a process, that is used in path-name resolution for path names that do not begin with a slash. *X/Open*. *ISO.1*. (2) In DOS, the directory that is searched when a file name is entered with no indication of the directory that lists the file name. DOS assumes that the current directory is the root directory unless a path to another directory is specified. *IBM*. (3) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*. (4) In the AIX operating system, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM*.

cursor. A reference to an element at a specific position in a data structure.

D

data definition (DD). (1) In the C and C++ languages, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM*. (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI*. (3) A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

data definition name. See *ddname*.

data member. The smallest possible piece of complete data. Elements are composed of data members.

data set. Under MVS, a named collection of related data records that is stored and retrieved by an assigned name. Equivalent to a CMS *file*.

data structure. The internal data representation of an implementation.

data type. The properties and internal representation that characterize data.

DBCS (double-byte character set). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM*.

ddname (data definition name). (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to `fopen` or `freopen` to refer to the data definition stored in the environment.

DD statement (data definition statement). (1) In MVS, serves as the connection between the logical name of a file and the physical name of the file. (2) A job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

decimal constant. (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *IBM*.

declaration. (1) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) Establishes the names and characteristics of data objects and functions used in a program.

default action. Each action class has a default action. It is defined as the first action listed for the class.

default constructor. A constructor that takes no arguments, or, if it takes arguments, all its arguments have default values.

default editor. The editor that is first in the list of editors in the **Tools setup** window. This editor is invoked when you double-click on an error message in the monitor, or when another tool requests an Edit action to be invoked.

default locale. (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

delete. (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by `new`.

demangling. The conversion of mangled names back to their original source code names. During C++ compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

denormal. Pertaining to a number with a value so close to 0 that its exponent cannot be represented normally. The exponent can be represented in a special way at the possible cost of a loss of significance.

derived class. A class that inherits from a base class. All members of the base class become members of the derived class. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

descriptor. *PL/I* control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one *PL/I* routine to another during run time.

destructor. A special member function that has the same name as its class, preceded by a tilde (˜), and that "cleans up" after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

device. A computer peripheral or an object that appears to the application as such. *X/Open*. *ISO.1*.

difference. Given two sets A and B, the difference (A-B) is the set of all elements contained in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if *m*>*n*, the difference contains that element *m*-*n* times. If *m*≤*n*, the difference contains that element zero times.

directory. A type of file containing the names and controlling information for other files or other directories. *IBM*.

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open*.

dot. The file name consisting of a single dot character (.). *X/Open*. *ISO.1*.

double-byte character set. See *DBCS*.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. *ISO-JTC1*. *ANSI*.

doubleword. A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. *IBM*.

dump. To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM*.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM*.

dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time. The code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic storage. Synonym for *automatic storage*.

E

EBCDIC (extended binary-coded decimal interchange code). A coded character set of 256 8-bit characters. *IBM*.

element. The component of an array, subrange, enumeration, or set.

empty string. (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

encapsulation. Hiding the internal representation of data objects and implementation details of functions from the client program. This enables the end user to focus on the use of data objects and functions without having to know about their representation or implementation.

enclave. In the Language Environment for MVS and VM, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

entry point. In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

enumeration constant. In the C or C++ language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed. *IBM*.

enumerator. In the C and C++ language, an enumeration constant and its associated value. *IBM*.

environment variable. In a WorkFrame project, an environment variable is an operating system variable, like PATH and DPATH, and any other environment variables that are defined using the OS/2 SET command, such as TMP.

equivalence class. (1) A grouping of characters that are considered equal for the purpose of collation; for example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. *IBM*. (2) A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight. *X/Open*.

escape sequence. (1) A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `x`, `\`, or followed by one or more octal or hexadecimal digits. (2) A sequence of characters that represent, for example, nonprinting characters, or the exact code point value to be used to represent variant and nonvariant characters regardless of code page. (3) In the C and C++ language, an escape character followed by one or more characters. The escape character indicates that a different code, or a different coded character set, is used to interpret the characters that follow. Any member of the character set used at runtime can be represented using an escape sequence. (4) A character that is preceded by a backslash character and is interpreted to have a special meaning to the operating system. (5) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen. Synonymous with multibyte control. *IBM*.

exception. (1) Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1*.

exception handler. (1) Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications. (2) A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM*.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open*.

extension. (1) An element or function not included in the standard language. (2) File name extension.

F

file scope. A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

file-scoped action. Distinguished from a project-scoped action in that it is invoked on files. Only file-scoped actions can participate in a project build.

filter. In WorkFrame, the value of a type. The filter of a type can be expressed as a file mask, regular expression, a logical-OR, a logical-AND, or logical-NOT of a list of types, or a filter determined by a PAM.

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

for statement. A looping statement that contains the word *for* followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM*.

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM*.

function definition. The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a **;** (semicolon). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

function template. Provides a blueprint describing how a set of related individual functions can be constructed.

G

global. Pertaining to information available to more than one program or subroutine. *IBM*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

GMT (Greenwich Mean Time). The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by *coordinate universal time (UTC).*

Greenwich Mean Time. See GMT.

H

header file. A text file that contains declarations used by a group of functions, programs, or users.

heap. An unordered flat collection that allows duplicate elements.

heap storage. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits. Three examples for the hexadecimal constant with value 0 would be '\x00', '0x0', or '0X00'.

I

I18N. Abbreviation for *internationalization*.

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI*. (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.

if statement. A conditional statement that contains the keyword if, followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action). *IBM*.

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. See *header file*.

include statement. In the C and C++ languages, a preprocessor statement that causes the preprocessor to replace the statement with the contents of a specified file. *IBM*.

incomplete type. A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures and unions of unspecified content. A void type can never be completed. Arrays of unknown size and structures or unions of unspecified content can be completed in further declarations.

indirection. (1) A mechanism for connecting objects by storing, in one object, a reference to another object. (2) In the C and C++ languages, the application of the unary operator * to a pointer to access the object the pointer points to.

inheritance. (1) A technique that allows the use of an existing class as the base for creating other classes. (2) In WorkFrame, refers to the mechanism in which the **Tools setup** of a project is shared by another project.

initial heap. The VisualAge C++ heap controlled by the HEAP runtime option and designated by a `heap_id` of 0. The initial heap contains dynamically allocated user data.

initializer. An expression used to initialize data objects. In the C++ language, there are three types of initializers:

1. An expression followed by an assignment operator is used to initialize fundamental data

type objects or class objects that have copy constructors.

2. An expression enclosed in braces ({ }) is used to initialize aggregates.
3. A parenthesized expression list is used to initialize base classes and members using constructors.

input stream. A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM*.

instance. An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration:

```
box box1, box2;
```

instantiate. To create or generate a particular instance or object of a data type. For example, an instance `box1` of class `box` could be instantiated with the declaration:

```
box box1;
```

instruction. A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

instruction scheduling. An optimization technique that reorders instructions in code to minimize execution time.

integer constant. A decimal, octal, or hexadecimal constant.

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open*.

Synonymous with *I18N*.

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

K

keyword. (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

L

label. An identifier within or attached to a set of data elements. *ISO Draft*.

Language Environment. Abbreviated form of IBM Language Environment for MVS and VM. Pertaining to an IBM software product that provides a common runtime environment and runtime services to applications compiled by Language Environment-conforming compilers.

last element. The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, calls, subroutines, or other data. *IBM*. (2) A set of object modules that can be specified in a link command.

line. A sequence of zero or more non-new-line characters plus a terminating new-line character. *X/Open*.

link. To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

linker. A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM*.

literal. (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, “APRIL” represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1*. (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM*. (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM*.

loader. A routine, commonly a computer program, that reads data into main storage. *ANSI*.

load module. All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft*.

local. (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1*.
(2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI*.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open*.

localization. The process of establishing information within a computer system specific to the operation of particular native languages, local customs, and coded character sets. *X/Open*.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

main function. An external function with the identifier `main` that is the first user function—aside from exit routines and C++ static object constructors—to get control when program execution begins. Each C and C++ program must have exactly one function named `main`.

make. An action in which a project's target is built from a make file by a make utility.

makefile. A text file containing a list of your application's parts. The make utility uses makefiles to maintain application parts and dependencies.

MakeMake. WorkFrame's make file generation utility.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. The prelinker uses these mangled names to ensure type-safe linkage. See also *demangling*.

map file. A listing file that can be created during the prelink or link step and that contains information on the size and mapping of segments and symbols.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters. *ISO-JTC1*. *ANSI*.

member. A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

member function. (1) An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class. Member functions are also called methods. (2) A function that performs operations on a class.

method. In the C++ language, a synonym for member function.

migrate. To move to a changed operating environment, usually to a new release or version of a system. *IBM*.

mode. A collection of attributes that specifies a file's type and its access permissions. *X/Open*. *ISO.1*.

module. A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

module definition file. a file used by the linker that contains module statements that define general attributes of the executable being linked, segment attributes, and imported or exported functions and data.

Monitor. A window that displays output from monitored actions. The Monitor window is attached to the project container.

monitored action. An action that has been set to run in the **Monitor** window, er actions may run in a full-screen and outputs to standard out. Actions may also run in full-screen and windowed sessions.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multicharacter collating element. A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements *ch* and *ll*. *X/Open*.

multiple inheritance. An object-oriented programming technique implemented in the C++ language through derivation, in which the derived class inherits members from more than one base class.

mutex. A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

N

name. In the C++ language, a name is commonly referred to as an identifier. However, syntactically, a name can be an identifier, operator function name, conversion function name, destructor name or qualified name.

nested class. A class defined within the scope of another class.

nested project. A project that appears inside another project. Nesting expresses a dependency of the parent project on the child project's target. This dependency is managed by WorkFrame's Build utility.

newline character. A character that in the output stream indicates that printing should start at the beginning of the next line. The newline character is designated by '\n' in the C and C++ language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. *X/Open*.

node. In a tree structure, a point at which subordinate items of data originate. *ANSI*.

NULL. In the C and C++ languages, a pointer that does not point to a data object. *IBM*.

null character (NUL). The ASCII or EBCDIC character '\0' with the hex value `00`, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

null pointer. The value that is obtained by converting the number 0 into a pointer; for example, `(void *) 0`. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open*.

null string. (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

null value. A parameter position for which no value is specified. *IBM*.

number sign. The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

O

object. (1) A region of storage. An object is created when a variable is defined or new is invoked. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM*. (3) An instance of a class.

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as the C++ language). For programs that must be linked, object code consists of relocatable machine code.

object module. (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft*. (2) A set of instructions in machine language produced by a compiler from a source program. *IBM*.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

open file. A file that is currently associated with a file descriptor. *X/Open*. *ISO.1*.

operand. An entity on which an operation is performed. *ISO-JTC1*. *ANSI*.

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1*.

overflow. (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM*.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

pack. To store data in a compact form in such a way that the original form can be recovered.

parameter. (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

parent process. (1) The program that originates the creation of other processes by means of `spawn` or `exec` function calls. See also *child process*. (2) A process that creates other processes.

parent project. A project that contains other projects. Distinguished from *child project*.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM*.

path name. (1) A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes will be treated as a single slash. The interpretation of the path name is described in *pathname resolution*. *ISO.1*. (2) A file name specifying all directories leading to the file.

pattern. A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively.

The syntaxes of the two patterns are similar, but not identical. *X/Open*.

period. The character (**.**). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named <period> in the portable character set.

pipe. To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (|). Two commands connected in this way constitute a pipeline. *IBM*.

pointer. In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

pointer to member. An operator used to access the address of non-static members of a class.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4:

`<NUL>`	
`<alert>`	
`<backspace>`	
`<tab>`	
`<newline>`	
`<vertical-tab>`	
`<form-feed>`	
`<carriage-return>`	
`<space>`	
`<exclamation-mark>`	`!`
`<quotation-mark>`	`"`
`<number-sign>`	`#`
`<dollar-sign>`	`$`
`<percent-sign>`	`%`
`<ampersand>`	`&`
`<apostrophe>`	`'`
`<left-parenthesis>`	`(`
`<right-parenthesis>`	`)`
`<asterisk>`	`*`
`<plus-sign>`	`+`
`<comma>`	`,`
`<hyphen>`	`-`
`<hyphen-minus>`	`-`
`<period>`	`.`
`<slash>`	`/`
`<zero>`	`0`
`<one>`	`1`
`<two>`	`2`
`<three>`	`3`
`<four>`	`4`
`<five>`	`5`
`<six>`	`6`
`<seven>`	`7`
`<eight>`	`8`
`<nine>`	`9`
`<colon>`	`:`
`<semicolon>`	`;`
`<less-than-sign>`	`<`
`<equals-sign>`	`=`
`<greater-than-sign>`	`>`
`<question-mark>`	`?`
`<commercial-at>`	`@`

<A>	A
<B>	B
<C>	C
<D>	D
<E>	E
<F>	F
<G>	G
<H>	H
<I>	I
<J>	J
<K>	K
<L>	L
<M>	M
<N>	N
<O>	O
<P>	P
<Q>	Q
<R>	R
<S>	S
<T>	T
<U>	U
<V>	V
<W>	W
<X>	X
<Y>	Y
<Z>	Z
<left-square-bracket>	[
<backslash>	\
<reverse-solidus>	\
<right-square-bracket>	]
<circumflex>	^
<circumflex-accent>	^
<underscore>	_
<low-line>	_
<grave-accent>	`
<a>	a
<b>	b
<c>	c
<d>	d
<e>	e
<f>	f
<g>	g
<h>	h
<i>	i
<j>	j
<k>	k
<l>	l
<m>	m
<n>	n
<o>	o
<p>	p
<q>	q
<r>	r
<s>	s
<t>	t
<u>	u
<v>	v
<w>	w
<x>	x
<y>	y
<z>	z
<left-brace>	{
<left-curly-bracket>	{
<vertical-line>	|
<right-brace>	}
<right-curly-bracket>	}
<tilde>	~

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

precedence. The priority system for grouping different types of operators with their operands.

predefined macros. Frequently used routines provided by an application or language for the programmer.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

private. Pertaining to a class member that is only accessible to member functions and friends of that class.

process. (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the `fork`[] function. The process that issues the `fork`[] function is known as the parent process, and the new process created by the `fork`[] function is known as the child process. *X/Open. ISO.1.*

project. The central WorkFrame model of the complete set of data and actions required to build a single target, such as a dynamic link library (DLL) or other executable. A project consists of a set of *project parts* and a **Tools setup.**.

Project Access Method (PAM). A dynamic link library that contains a set of methods through which a simple abstraction of a file system or repository is provided to WorkFrame. PAMs enable a WorkFrame project to contain any kind of object that a PAM can support, for example a version of a file in a source control library, or another file system like MVS or AIX.

project hierarchy. A project tree that represents dependencies between projects. The WorkFrame project paradigm requires that one project should be created for every target. Dependencies between projects and their targets should be expressed in a project hierarchy. That is, if a project's build depends on the target of another project, the dependent project should contain the project it depends on. The dependent project is then said to *nest* the other project. This enables the Build tool to perform Builds in a depth-first search manner from anywhere in the project hierarchy.

project-scoped action. An action that applies to a project as a whole, or to a project's specially designated parts. Specially designated project parts are the project's make file and target. An example of a project-scoped action is Debug, which is invoked on the project's target.

Project Smarts. A project catalog that contains templates for common types of applications.

Project Smarts application. A skeletal application that consists of template source code and a configured project revolving around an application theme. It serves as a starting point for similar applications.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

public. Pertaining to a class member that is accessible to all functions.

Q

qualified name. Used to qualify a nonclass type name such as a member by its class name.

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

R

register storage class specifier. A specifier that indicates to the compiler within a block scope data definition, or a parameter declaration, that the object being described will be heavily used.

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open*.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

regular expression. (1) A mechanism to select specific strings from a set of character strings. (2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open*. *ISO.1*.

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

runtime library. A compiled collection of functions whose members can be referred to by an application program during runtime execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The runtime library itself is not statically bound into the application modules.

S

scalar. An arithmetic object, or a pointer to an object of any type.

scope. (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

semaphore. An object used by multithread applications for signalling purposes and for controlling access to serially reusable resources.

Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence. A sequentially ordered flat collection.

session. A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership. There can be multiple process groups in the same session. *X/Open*. *ISO.1*.

shell. A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open*.

This feature is provided as part of OpenEdition MVS Shell and Utilities feature licensed program.

signal. (1) A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open*. *ISO.1*. (3) In AIX operating system operations, a method of interprocess communication that simulates software interrupts. *IBM*.

signal handler. A function to be called when the signal is reported.

slash. The character /, also known as *solidus*. This character is named <slash> in the portable character set.

S-name. An external non-C++ name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

source directory. A directory where a project's parts are physically stored. A project may have many source directories.

source file. A file that contains source statements for such items as high-level language programs and data description specifications. *IBM*.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM*.

source type. A source type appears in an action's list of source types. An action's list of source types specifies the kind of parts or files to which the action applies.

space character. The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open*.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack storage. Synonym for *automatic storage*.

standard error. An output stream usually intended to be used for diagnostic messages. *X/Open*.

standard input. (1) An input stream usually intended to be used for primary data input. *X/Open*. (2) The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM*.

standard output. (1) An output stream usually intended to be used for primary data output. *X/Open*. (2) In the AIX operating system, the primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM*.

statement. An instruction that ends with the character **;** (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

storage class specifier. One of: auto, register, static, or extern.

stream. (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the `fdopen` or `fopen` functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open*.

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open*.

string literal. Zero or more characters enclosed in double quotation marks.

struct. An aggregate of elements, having arbitrary types.

structure. A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subsystem. A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft*.

superset. Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

support. In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM*.

switch statement. A C or C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile. *IBM*.

T

tab character. A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*.

This character is named <tab> in the portable character set.

target. A project's target is the file that is produced as a result of a project build.

target type. A target type appears in an action's list of target types. Target types only apply to actions that participate in a project build, such as Compile and Link. The Build tool and MakeMake utility use the source and target types of build actions to determine the order in which the actions should be run to produce the project's target.

task. (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. *ISO-JTC1*. *ANSI*. (2) A routine that is used to simulate the operation of programs. Tasks are said to be *nonpreemptive* because only a single task is executing at any one time. Tasks are said to be *lightweight* because less time and space are required to create a task than a true operating system process.

task library. A class library that provides the facilities to write programs that are made up of tasks.

template. (1) A family of classes or functions with variable types. (2) An object that you can use as a model to create other objects. When you drag a *template*, you create a copy of the original object. The new object has the same settings and contents as the original *template* object.

template class. A class instance generated by a class template.

template function. A function generated by a function template.

text file. A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX}—which is defined in limits.h—bytes in length, including the new-line character. The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). *X/Open*.

this. A C++ keyword that identifies a special type of pointer in a member function, that references the class object with which the member function was invoked.

thread. The smallest unit of operation to be performed within a process. *IBM*.

tilde. The character ˜. This character is named <tilde> in the portable character set.

Tools setup. A view of a project where you can see and manipulate the actions, types, and environment variables available to the project. From this view, you can add, delete, and change actions, types, and variables. You can also set the options for any action in this view.

trap. An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred. *ISO-JTC1*.

type. (1) The description of the data and the operations that can be performed on or by the data. See also *data type*. (2) In WorkFrame, describes a group of project files of parts in terms of an expression, such as file masks, regular expressions, or a list of other types, logical-OR'd.

type class. In WorkFrame, represents the method by which an object is determined to be a member of a type. “File mask” is an example of a type class. Membership to a “File mask” type is determined by matching the file mask filter to the object's name. Other examples of type classes are “Regular expression,” and “PAM Name,” where the named Project Access Method determines membership to a type.

type definition. A definition of a name for a data type. *IBM*.

type specifier. Used to indicate the data type of an object or function being declared.

U

undefined behavior. Referring to a program or function that may produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data.

underflow. (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation. *IBM*.

union. (1) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m*+*n* times.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

V

variable. In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

variant character. A character whose hexadecimal value differs between different character sets. On EBCDIC systems, such as S/390, these 13 characters are an exception to the portability of the portable character set.

```
<left-square-bracket>  [
<right-square-bracket> ]
<left-brace>           {
<right-brace>          }
<backslash>            \
<circumflex>           ^
<tilde>                ~
<exclamation-mark>     !
<number-sign>          #
<vertical-line>        |
<grave-accent>         `
<dollar-sign>          $
<commercial-at>        @
```

virtual function. A function of a class that is declared with the keyword `virtual`. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called, which is determined at run time.

visible. Visibility of identifiers is based on scoping rules and is independent of *access.*

W

white space. (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

wide character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

wide-character string. A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

word boundary. Any storage position at which data must be aligned for certain processing operations. The halfword boundary must be divisible by 2; the fullword boundary by 4; and the doubleword boundary by 8. *IBM*.

working directory. (1) Synonym for *current working directory*. (2) The directory where files that are copied or dragged into the project are stored. Actions are also executed in this directory, so this directory is where many output files are placed.

write. (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1*. *ANSI*.

Bibliography

This bibliography lists the publications that make up the IBM VisualAge C++ library and publications of related IBM products referenced in this book. The list of related publications is not exhaustive but should be adequate for most VisualAge C++ users.

The IBM VisualAge C++ Library

The following books are part of the IBM VisualAge C++ library.

- *Read Me First!*, S25H-6956
- *Welcome to VisualAge C++*, S25H-6957
- *User's Guide*, S25H-6961
- *Programming Guide*, S25H-6958
- *Visual Builder User's Guide*, S25H-6960
- *Visual Builder Parts Reference*, S25H-6967
- *Building VisualAge C++ Parts for Fun and Profit*, S25H-6968
- *Open Class Library User's Guide*, S25H-6962
- *Open Class Library Reference*, S25H-6965
- *Language Reference*, S25H-6963-00
- *C Library Reference*, S25H-6964

The IBM VisualAge C++ BookManager Library

The following documents are available in VisualAge C++ in BookManager format.

- *Read Me First!*, S25H-6956
- *Welcome to VisualAge C++*, S25H-6957
- *User's Guide*, S25H-6961
- *Programming Guide*, S25H-6958
- *Visual Builder User's Guide*, S25H-6960
- *Visual Builder Parts Reference*, S25H-6967
- *Building VisualAge C++ Parts for Fun and Profit*, S25H-6968
- *Open Class Library User's Guide*, S25H-6962
- *Open Class Library Reference*, S25H-6965
- *Language Reference*, S25H-6963-00
- *C Library Reference*, S25H-6964

C and C++ Related Publications

- *Portability Guide for IBM C*, SC09-1405
- *American National Standard for Information Systems / International Standards Organization — Programming Language C (ANSI/ISO 9899-1990[1992])*
- *Draft Proposed American National Standard for Information Systems — Programming Language C++ (X3J16/92-0060)*

IBM OS/2 2.1 Publications

The following books describe the OS/2 2.1 operating system and the Developer's Toolkit 2.1.

- *OS/2 2.1 Using the Operating System*, S61G-0703
- *OS/2 2.1 Installation Guide*, S61G-0704
- *OS/2 2.1 Quick Reference*, S61G-0713
- *OS/2 2.1 Command Reference*, S71G-4112
- *OS/2 2.1 Information and Planning Guide*, S61G-0913
- *OS/2 2.1 Keyboard and Codepages*, S71G-4113
- *OS/2 2.1 Bidirectional Support*, S71G-4114
- *OS/2 2.1 Book Catalog*, S61G-0706
- *Developer's Toolkit for OS/2 2.1: Getting Started*, S61G-1634

IBM OS/2 3.0 Publications

- *User's Guide to OS/2 Warp*, G25H-7196-01

The following books make up the OS/2 3.0 Technical Library (G25H-7116).

- *Control Program Programming Guide*, G25H-7101
- *Control Program Programming Reference*, G25H-7102

- *Presentation Manager Programming Guide - The Basics*, G25H-7103
- *Presentation Manager Programming Guide - Advanced Topics*, G25H-7104
- *Presentation Manager Programming Reference*, G25H-7105
- *Graphics Programming Interface Programming Guide*, G25H-7106
- *Graphics Programming Interface Programming Reference*, G25H-7107
- *Workplace Shell Programming Guide*, G25H-7108
- *Workplace Shell Programming Reference*, G25H-7109
- *Information Presentation Facility Programming Guide*, G25H-7110
- *OS/2 Tools Reference*, G25H-7111
- *Multimedia Application Programming Guide*, G25H-7112
- *Multimedia Subsystem Programming Guide*, G25H-7113
- *Multimedia Programming Reference*, G25H-7114
- *REXX User's Guide*, S10G-6269
- *REXX Reference*, S10G-6268

Other Books You Might Need

The following list contains the titles of IBM books that you might find helpful. These books are not part of the VisualAge C++ or OS/2 libraries.

BookManager READ/2 Publications

- *IBM BookManager READ/2: General Information*, GB35-0800
- *IBM BookManager READ/2: Getting Started and Quick Reference*, SX76-0146
- *IBM BookManager READ/2: Displaying Online Books*, SB35-0801
- *IBM BookManager READ/2: Installation*, GX76-0147

Non-IBM Publications

Many books have been written about the C++ language and related programming topics. The authors use varying approaches and emphasis. The following is a sample of some non-IBM C++ publications that are generally available. This sample is not an exhaustive list. IBM does not specifically recommend any of these books, and other C++ books may be available in your locality.

- *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley Publishing Company.
- *C++ Primer* by Stanley B. Lippman, Addison-Wesley Publishing Company.
- *Object-Oriented Design with Applications* by Grady Booch, Benjamin/Cummings.
- *Object-Oriented Programming Using SOM and DSOM* by Christina Lau, Van Nostrand Reinhold.
- *OS/2 C++ Class Library: Power GUI Programming with C Set* ++ by Kevin Leong, William Law, Robert Love, Hiroshi Tsuji, and Bruce Olson, Van Nostrand Reinhold.

Index

Communicating Your Comments to IBM

IBM VisualAge C++ for OS/2
IBM VisualAge C++ for OS/2 User's Guide

Version 3.0

Publication No. S25H-6961-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@vnet.ibm.com
  - IBMLink: toribm(torrcf)
  - IBM/PROFS: torolab4(torrcf)
  - IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

IBM VisualAge C++ for OS/2
IBM VisualAge C++ for OS/2 User's Guide

Version 3.0

Publication No. S25H-6961-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name ______________________ Address ______________________

Company or Organization ______________________

Phone No. ______________________

Readers' Comments — We'd Like to Hear from You
S25H-6961-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Part Number: 25H6961
Program Number: 5622-679
30H1664
30H1665
30H1666
30H1679

Printed in U.S.A.

25H6961

S25H-6961-00