Note!

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

First Edition (November 1996)

This edition applies to Version 3, Release 7, Modification Level 0, of IBM VisualAge for C++ for AS/400 (Program 5716-CX5) 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. Consult the latest edition of the applicable IBM system bibliography for current information on this 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. 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.

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 Avenue East, North York, Ontario M3C 1H7, Canada. 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 AS/400 C++ applications using the VisualAge for C++ for AS/400 product. It primarily documents General-Use Programming Interface and Associated Guidance Information provided by the VisualAge for C++ for AS/400 product.

General-Use programming interfaces allow the customer to write applications that obtain the services of the VisualAge for C++ for AS/400 compiler, debugger, browser, and class libraries.

This book documents Diagnosis, Modification, and Tuning Information, which is provided to help you debug your programs.

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, either by an introductory statement to a chapter or section.

Trademarks and Service Marks

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

AnyNet
Application System/400
AS/400
COBOL/400
C/400
IBM
IBMlink
Integrated Language Environment
Open Class
Operating System/400
OS/400
PROFS
RPG/400
SAA
VisualAge
400

Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation.

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

IBM's VisualAge products and services are not associated with or sponsored by Visual Edge Software, Ltd.

About This Guide

IBM VisualAge for C++ for AS/400 offers a productive application development environment based on the C++ programming language which supports an object-oriented design approach. Its many tools provide the means to build productive applications that exploit the strengths of both the Windows and the OS/400 platforms.

This guide describes how to use IBM VisualAge for C++ for AS/400 to compile, bind, and debug C++ applications that run in the Integrated Language Environment (ILE) on AS/400. It explains the steps required to perform these tasks, and introduces the tools that are specific to developing OS/400 applications with IBM VisualAge for C++ for AS/400.

Who Should Use This Guide

This book is written for application and systems programmers who want to create OS/400 ILE C++ applications from the Windows development environment, using IBM VisualAge for C++ for AS/400. You should have a working knowledge of the C and C++ programming languages, the Windows and OS/400 operating systems, and related products as described in the *Installation Guide and Product Overview*, as well as knowledge of ILE as explained in the *ILE Concepts*.

How to Use This Guide

Use this guide for information on how to compile, bind, debug and manage applications with IBM VisualAge for C++ for AS/400.

See the *C++ Programming Guide* for reference information on the more technical aspects of the compiler, and advanced programming techniques.

How to Read the Syntax Diagrams

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

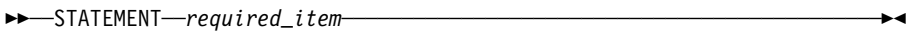
  The ►— symbol indicates that a command, directive, or statement is continued from the previous line.

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

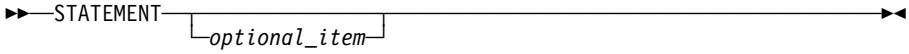
Diagrams of syntactical units other than complete commands, directives, or statements start with the ►— symbol and end with the —► symbol.

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

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

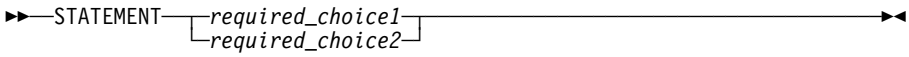


- Optional items appear below the main path.

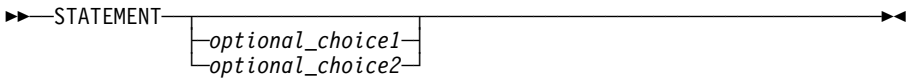


- If you can choose from two or more items, they appear vertically, in a stack.

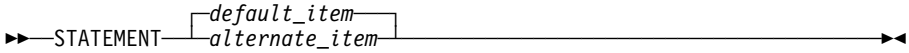
  If you must choose one of the items, one item of the stack appears on the main path.



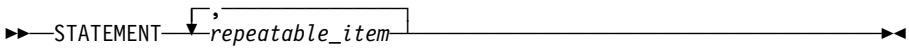
  If the items are optional, the entire stack appears below the main path.



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



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



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

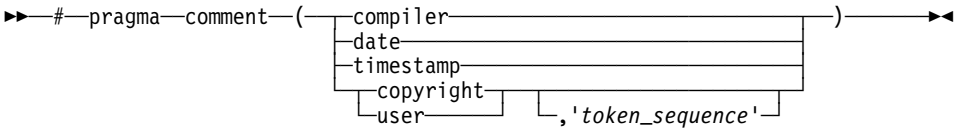
- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, **pragma**).

  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.

This syntax diagram example shows the syntax for the **#pragma comment** directive. (See *C++ Language Reference* for information on the **#pragma** directive.)



The syntax diagram is interpreted in the following manner:

- The ►►— is the start of the syntax diagram
- The symbol # must appear first
- The keyword **pragma** must appear following the # symbol
- The keyword `comment` must appear following the keyword **pragma**
- An opening parenthesis must be present
- The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`
- If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type
- A character string must follow the comma
- A closing parenthesis is required
- The —►◄ 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.
- 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.

Highlighting Conventions

The following highlighting conventions are used in this book:

Bold Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.

Italics Identify parameters whose actual names or values are to be supplied by the programmer. *Italics* are also used for the first mention of new terms.

`Example` Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

How to Get Help

You can access three kinds of online information while you are using VisualAge for C++ for AS/400:

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 for C++ for AS/400. For your convenience, the online documents are presented in a standard format. See “Getting Help Inside VisualAge for C++ for AS/400” for instructions on opening standard format documents from inside VisualAge for C++ for AS/400. See “Getting Help from the Command Line” on page xvii for instructions on opening standard format documents from the command line.

Contextual help
Contextual help is available throughout VisualAge for C++ for AS/400. 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 for C++ for AS/400 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 for C++ for AS/400, as well as individual task lists for each of its components.

Getting Help Inside VisualAge for C++ for AS/400

All three kinds of help are available directly within the VisualAge for C++ for AS/400 interface:

- To get general contextual help for the component of VisualAge for C++ for AS/400 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:
  - **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 for C++ for AS/400 being used
- To get detailed information, open the **Online Information** notebook in the VisualAge for C++ for AS/400 folder. In this notebook there are tabs for **Guides**, **References**, and **How Do I** help. Each page in the notebook lists a variety of online documents that describe, in detail, the different aspects of VisualAge for C++ for AS/400. To open a particular on online document select the radio button for the document, and click on the **View** pushbutton.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the **iview** command. The installation routine stores the online document files in the `\CTTASW\HELP` directory. To view the *VisualAge for C++ for AS/400 C++ Language Reference* make `C:\CTTASW\HELP` your current directory (substituting the drive where you installed VisualAge for C++ for AS/400 for `C:`) and enter `IVIEW CTTLNG.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. If you want to read the section on operator precedence in the *VisualAge for C++ for AS/400 C++ Language Reference*, enter `VIEW CTTLNG.INF OPERATOR PRECEDENCE`.

Getting Help for a Keyword or Construct

If you are editing a file using the VisualAge Editor, you can get help for a keyword or construct by moving the cursor to the word and pressing **Ctrl+H**. In other tools, you can set help for a keyword or construct by highlighting the word and pressing **Ctrl+H**.

Online Documents Available in VisualAge for C++ for AS/400

These documents are available in standard format:

- *C++ User's Guide*
- *C++ Programming Guide*
- *C++ Language Reference*
- *IBM Open Class Library User's Guide*
- *IBM Open Class Library Reference*

- *C Library Reference*
- *IBM Access Class Library User's Guide*
- *IBM Access Class Library for OS/400 Reference*
- *IBM Access Class Library for Windows Reference*

Part 1. Developing OS/400 Applications with WorkFrame

The sections in this part briefly introduce WorkFrame and describe how you can use the WorkFrame application development environment to manage your OS/400 projects.

Chapter 1. Introducing WorkFrame

IBM WorkFrame is an application development environment that puts your tools at your fingertips. It simplifies the process of building and organizing software projects.

Note: VisualAge for C++ for AS/400 shares WorkFrame with the VisualAge for C++ for Windows product. You must have WorkFrame installed as part of VisualAge for C++ for Windows before you can use its to develop OS/400 applications.

Whether your projects target the Windows or the OS/400 run-time environment, WorkFrame's basic functionality is the same. See the *VisualAge for C++ for Windows User's Guide* for a description of the main project-related tasks, such as managing projects, setting up project tools, and building project targets with WorkFrame.

This section provides information that is specific to developing OS/400 applications with WorkFrame.

Overview of WorkFrame

Before you start developing applications with WorkFrame, you need to be familiar with a number of WorkFrame-related concepts and tasks.

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

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 applications.

As you program, you can use tools, such as an editor, a browser, or a debugger. WorkFrame makes these tools available to you as context-sensitive actions in pop-up and pull-down menus in projects, and from the **Project** menu of many VisualAge for C++ tools.

You set options for actions quickly and easily using graphical user interfaces:

- A Build Smarts window lets you quickly modify options that affect the way your project is built. You can set debug and browse options to easily build your project for debugging and production scenarios. Build Smarts also has options to generate a new makefile with every build, and to build subprojects first.
- When you invoke a Compile action from a pop-up or pulldown menu, WorkFrame invokes the compiler with your preset options.

WorkFrame Terms and Concepts You Need to Know

WorkFrame terms and concepts are explained in detail in the *VisualAge for C++ for Windows User's Guide*. The following is a short list of the most frequently used terms:

Action An action changes a project's parts, or starts a build. Some actions, such as compile, have options that you can set through dialogs from the **Options** pull-down menu.

Build Smarts The Build Smarts notebook contains options for actions that affect the way your project target is built. Build Smarts options work in combination with the options for the individual actions. By default, Build Smarts is enabled and overrides options set in the compiler option notebook. Select **Build Smarts** from the **Options** menu, or from the toolbar.

Project A project consists of a set of *parts* (files) and *actions* to build a target.

Project Directories Project directories store project files.

Project Settings Project settings are specified in a project **Settings** notebook.

Project Smarts Project Smarts is a catalog of skeleton OS/400 applications that you can use as a starting point to write your own applications.

Project Target The project target is the result of a project build. Two WorkFrame utilities help you build a target: Build and MakeMake. The Build utility dynamically builds project targets and manages makefiles. MakeMake is WorkFrame's makefile creation utility.

Project View A project view displays the content of a project. By default, a project opens to its icon view, but you can select the tree view from the **View** menu.

Tools Options Tools options specify how a tool behaves when its action is invoked. WorkFrame tools options have default settings. You can modified these settings through the dialogs on the **Options** pull-down menu.

Choosing the Appropriate Development Tools

When you use VisualAge for C++ for AS/400 together with VisualAge for C++ for Windows, you can develop C++ applications for OS/400, or client/server applications which run on a Windows workstation and an AS/400 system. In either case, the WorkFrame environment lets you access the appropriate development tools:

- Create a new Windows client project through the Project Smarts interface that can be started from the WorkFrame IDE icon in the VisualAge for C++ for Windows product folder.

- Create a new project for an OS/400 application, or the OS/400 server portion of a client/server application, from the WorkFrame IDE icon in the VisualAge for C++ for AS/400 product folder.

Projects are stored in .IWP files. They contain source files and compiler-generated files. Table 1 shows the differences that exist between Windows and OS/400 project objects:

Table 1. Differences between Windows and OS/400 Project Objects

Windows	**OS/400**
Object module (`.OBJ`)	Module object (`*MODULE`)
Executable (`.EXE`)	Program object (`*PGM`)
Dynamic link library (`.DLL`)	Service program (`*SRVPGM`)
Library (`.LIB`)	N/A
Help file (`.HLP`)	N/A

WorkFrame Actions for OS/400 Projects

Actions are performed on project objects. Typical objects in an OS/400 project are:

- Source files
- Header files
- Module objects
- Program objects
- Service program objects
- Browser files
- Makefiles
- Listing files

When you select an object, and then press mouse button 2 to bring up a pop-up menu, you see a list of all relevant actions for this object. The following list defines the actions that can be invoked for OS/400 projects.

Build Build a project. **Build normal** builds only out-of-date files; **Rebuild all** builds all targets, even if they are not out of date with respect to their dependent files.

Debug Invoke the cooperative debugger.

Edit Invoke the VisualAge Editor.

Browse Invoke the VisualAge Browser and the VisualAge Editor.

Compile Compile project files with the cooperative compiler.

Make Invoke the Make utility.

MakeMake Create a makefile.

View	Display a project.
Bind	Bind modules into programs or service programs.

Getting Help for WorkFrame

You can access three kinds of online help while using WorkFrame:

- Contextual online help gives you help from within WorkFrame
- How Do I ? help gives you step-by-step instructions on how to perform a number of project-related tasks.
- An online version of this guide is accessible from the Help pull-down menu.

Using Contextual Help

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

- Select an item from a **Help** pull-down menu in any WorkFrame window. The Help menu in a project container gives you access to VisualAge for C++ for AS/400 manuals, How Do I ? information, and contextual 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.

Notes:

1. The help panels show information that is specific to developing Windows applications. In most cases, this information applies to OS/400 application development. Any differences are explained in this guide.
2. The notebooks for setting compiler and binder options, and the notebook for Project Smarts are specific to OS/400 application development. The help panels associated with these notebooks contain OS/400 information.

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.

Note: WorkFrame How Do I ? help is specific to Windows application development. Many WorkFrame tasks are performed in the same way, whether you are developing Windows or OS/400 applications. Information that is specific to OS/400 application development is presented in this guide.

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

- From the **Help** pull-down menu, select **How Do I ?** or **How Do I...Selections -> WorkFrame**.

- Select the **Online Information** icon located in the main VisualAge for C++ program group on your desktop. Select WorkFrame from the **How Do I ?** page of the **Online Information** notebook.
- Click on the **How Do I ?** button on the left side of the project toolbar.

Chapter 2. Creating OS/400 C++ Projects with WorkFrame

This section describes how to create and customize OS/400 WorkFrame projects.

Creating a Project

When developing an application with WorkFrame, you begin by creating a project in one of two ways:

- By copying another project's files
- From Project Smarts.

Copying Another Project

If you already have a similar project, copy and rename the project and option files. Project files have an extension of .IWP, while option files have an extension of .IWO. You can share source directories with the project being copied from, or use new source directories. If you are sharing source directories, use a different working directory so one project's build results are not overwritten by the others.

See the *VisualAge for C++ for Windows User's Guide* for information on how projects are represented in the file system.

Creating a Project from Project Smarts

Project Smarts generates a complete project with skeletal code for a particular application type. Choose one of the following application types from the OS/400 Project Smarts notebook:

- Default OS/400 project
- Project to create an OS/400 program
- Project to create an OS/400 service program
- Project to build a Collection Class Library application.

When you select one of the Project Smarts application types, a completely configured project is created on your desktop, with the appropriate environment settings and template code for the chosen type of application.

Starting Project Smarts

There are three ways to start OS/400 Project Smarts:

- Type `smart400` on the command line.
- Select **Create new project** from the pull-down menu of any OS/400 project.
- Select **Create new project** from the WorkFrame initial dialog. The initial dialog is displayed when you:
  - Double-click on the **WorkFrame IDE** icon in the **VisualAge for C++ for AS/400** product folder.

– Type `iwf400` on the command line, without specifying any arguments.

The Project Smarts notebook is opened at the **Project** page.

Using the Project Smarts Notebook

The **Project types** listbox in the notebook contains a list of VisualAge for C++ for AS/400 project types. As you select a project, the **Description** field is updated with a short description of the project.

To create one of the projects, select its name and then click on the **Next** push button. Each application is different; therefore, the notebook pages displayed next are different for each project type. Each page is labeled with a step number, and you can return to the previous page by clicking on the **Back** push button.

The **Location** page and the **AS/400** page apply to all project types. The **Source File** and the **Prolog** pages apply to some project types only. Enter the following information in the entry fields of these notebook pages:

Location page

- Name of the project file, and the directory where the project file should be created
- Source directories where the generated template files are stored
- Project name

AS/400 page

- Workstation drive to which the target AS/400 has been assigned
- AS/400 library where modules, programs and service programs for the project are created
- Unique name for the connection between your Windows workstation and the target AS/400.
- Network ID of the AS/400 to which you are connecting

Source File page

Base name for source files.

Prolog page

Prolog contents.

To create the project, click on the **Done** button when you are finished filling in project information. The new project displays on your desktop.

Specifying Project Settings

A project has a number of settings that you can modify through the Settings notebook. To open this notebook, select a Settings notebook page from the **View** menu **Settings** cascade.

All pages of the Settings notebook have four buttons:

OK Saves changes made to fields in the Settings notebook and closes the notebook.

Cancel Returns all fields on all pages to their previous values, and closes the notebook.

Undo Returns all fields on the page to their previous values.

Help Provides help for the fields on the page.

Using the Target Page

Enter information for 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 Enter the name of the project's target (`PROGRAM`).

A project can build more than one target. WorkFrame recognizes only one project part as the designated target of the project.

When you specify the target name, omit the object type (*MODULE, *PGM, or *SRVPGM). You cannot specify a fully qualified path if you want to create your target in a library other than the current library on AS/400. If you need to qualify the name of your target, you must do so in the dialogs for the VACPP400 Compile or the CRTPGM and CRTSRVPGM actions.

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. Table 2 shows the valid OS/400 project targets.

Run options This entry field is not available for OS/400 applications.

Table 2. Valid OS/400 Project Targets

OS/400 Target	Type	Name	Example
Module	*MODULE	*module-name*	MYMODULE
Program	*PGM	*program-name*	MYPGM
Service Program	*SRVPGM	*service-program-name*	MYSRVPGM

Makefile

Enter the name of the project's designated makefile that is used when a project-scoped build action is invoked. Makefiles have either the name `makefile` or the .MAK file extension.

A project can have more than one makefile. One makefile may build the target with debug information, and another may build it optimized. WorkFrame recognizes only one as the designated makefile of the project.

If you want to use a makefile to build your project target, type a file name in the **Makefile** entry field (`mymake.mak`). When you use MakeMake to generate a makefile for the project, it saves the makefile with the name that is specified in this field.

Using the Directories Page

Enter the source directories for your project in the **Directories** page. All the files in the source directories are assumed to be part of the project. Specify the following settings:

Source directories for project files

List the source directories for project files; 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. You could enter the path names:

```
D:\MYAPP\CPP
D:\MYAPP\HEADERS
G:\QSYS.LIB\MYLIB.LIB
```

The entry `G:\QSYS.LIB\MYLIB.LIB` indicates that project files are located in library `MYLIB`, on an AS/400 system that is assigned to drive `G:` on your workstation.

Working directory

Select the working directory for your project from the directories you specified in the **Source directories** field. By default, the first directory in the list of source directories is the project's working directory. This directory is used:

- To store any makefiles created by the MakeMake and Build utilities.
- As the current directory from which actions are launched. Because many tools, such as the compiler, place their output in the current directory, tool output such as listing and browser files are often stored here.

Using the Environment Page

Specify the environment variables you wish to set, one variable per line. Enter information in the format

```
name=value
```

where *name* is the name of the environment variable, and *value* is the value to which you want to set it.

The environment variables defined are active only for your project, without affecting variables in other sessions. The environment variables listed in a project's environment are set for any tool launched as an action from the project.

Using the Name Page

Specify the name of your project. The project name appears in the project window's title bar.

Setting Up Tools Options

Each WorkFrame tool has a set of options associated with it. The options represent the parameters that are passed to the tool when it is invoked. Tool options have default settings which can be modified in the option dialog notebooks available from the **Options** pull-down menu.

When you use WorkFrame projects to organize your tools and code, you need to set these options only once. When you later invoke the action, by itself or as part of a Build, the options are already set. If the tool you are using provides customized WorkFrame support, remembering command-line options is not necessary. You can set options in a graphical user interface, like a dialog or notebook, with full access to online help. The following notebooks are specific to OS/400 application development:

- Notebooks to set compile and bind options:
  - Compile Options notebook
  - Bind Options notebook CRTPGM (used when creating programs)
  - Bind Options notebook CRTSRVPGM (used when creating service programs)
- The OS/400 Project Smarts notebook

Figure 1 on page 14 shows the File page of the Compile Options notebook.

To set options for an action, select its name from the **Options** menu on the project menu bar. Only some actions are available on the **Options** menu.

Build Smarts Options

Build Smarts is a fast path for setting options for 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. By default, Build Smarts is enabled.

To display the Build Smarts window, select **Build Smarts** from the **Options** menu, or from the toolbar:

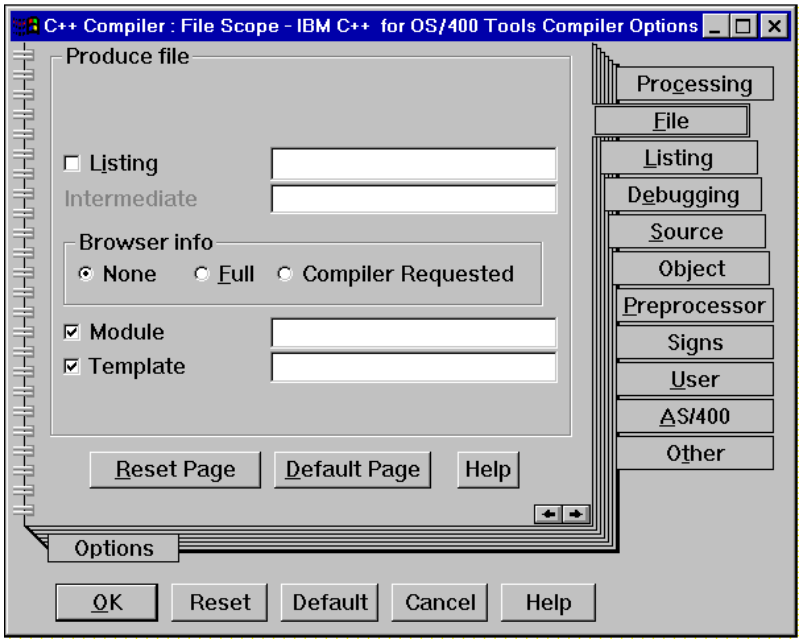


Figure 1. The VisualAge for C++ for AS/400 Compile Options Notebook

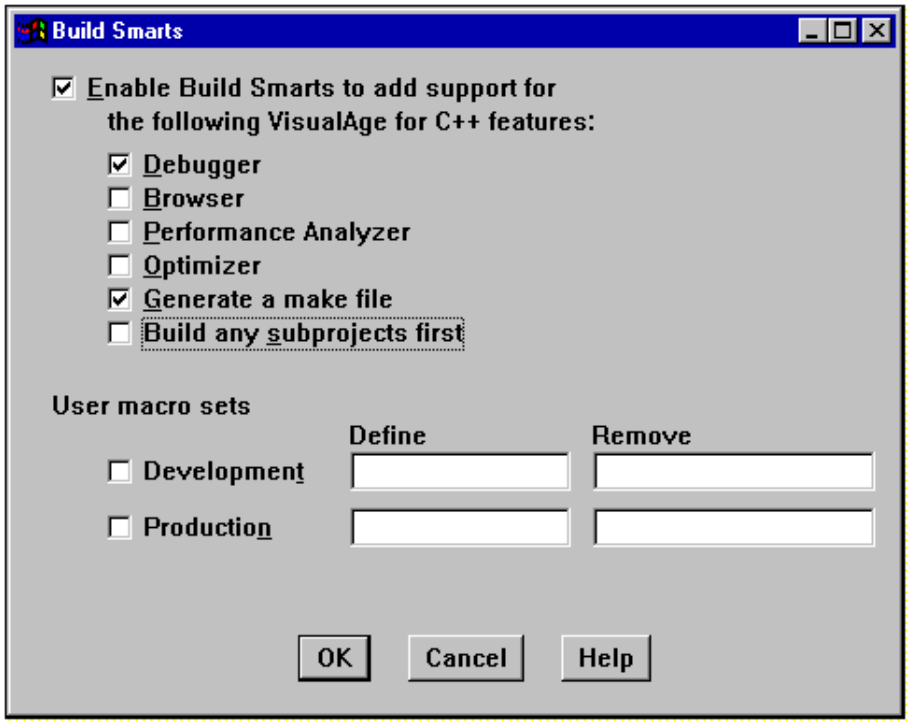


Figure 2. Build Smarts Window

Each setting in the Build Smarts window can affect one or more project actions. For instance, the **Debugger** check box affects the Compiler and Binder options. Selecting the **Debugger** check box effectively adds the `/Ti` option to the Compiler 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 build options are also included here: **Generate a makefile** and **Build any sub-projects first**. Select the first option to generate a new makefile as part of running the build. Select the second option to build any dependent 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** check box if you only want to use the options set for the individual actions.

Note: The Performance Analyzer is not available for analyzing OS/400 applications. Do not check this box in the Build Smarts window.

Building A Project Target

WorkFrame includes the Build and MakeMake utilities. The Build utility dynamically builds your project's target and manages your makefile for you. MakeMake is WorkFrame's makefile creation utility. It creates makefiles based on the actions and source files associated with your project.

Building OS/400 Targets with MakeMake

For MakeMake to work properly when building OS/400 targets, you must specify two environment variables:

ICCAS_DRIVE Specifies the mounted drive (`I`) to which the AS/400 has been assigned.

ICCAS_DIR Specifies the target AS/400 library (`MYLIB`).

Note: The drive must be mounted at the root, so that the information provided by these environment variables can be used to build a fully qualified path.

If you do not want to set these environment variables in your **autoexec.bat** file, you can set them in the **Environment** page of the Project Settings notebook.

1. Enter `ICCAS_DRIVE=N` in the **Environment Variables** entry field, where `N` is the mounted drive to which your target AS/400 has been assigned.
2. Enter `ICCAS_DIR=MYLIB` in the **Environment Variables** entry field, where `MYLIB` is the name of your target AS/400 library.

Restrictions When Using MakeMake for OS/400 Projects

MakeMake cannot generate makefiles that include the following tasks that may be required to build OS/400 projects:

- Mount network drives
- Establish a connection through the CTTCONN command
- Change the AS/400 connection environment prior to invoking the compiler and binder.

If you need to perform any of these tasks to build your OS/400 project, you should write your own makefile, or customize the makefile generated by MakeMake.

The project environment variables ICCAS_DRIVE and ICCAS_DIR are required to provide fully qualified paths to MakeMake. They are set in the **Environment** page of the Project Smarts notebook.

Creating an OS/400 Sample Project

This section shows how to create an OS/400 application using a WorkFrame project. It discusses the required development environment, and describes step by step how to set up the project and invoke project actions.

Setting up the Development Environment

Before you can create and use your first WorkFrame project, you must:

1. Have installed VisualAge for C++ for AS/400 and VisualAge for C++ for Windows.
2. Establish a connection between your workstation and the AS/400 where this sample application should be created. On a Windows command line type:

   ```
   cttconn /Hname
   ```

   where `name` is the AS/400 system, such as `APPN.SYSTEM` (in SNA), or `SYSTEM.NETWORK.COM` (in TCP/IP).

   See "Establishing an AS/400 Connection before Compiling and Binding" on page 28 for details on using the CTTCONN command to connect to an AS/400.
3. Mount your destination AS/400 at the root as drive `N:`. (See the *Client Access for Windows 95 Setup* for information on how to mount an AS/400 as a network drive).
4. Create an empty working directory `C:\EXAMPLE` on your workstation.
5. Create a library `HELLO` on the destination AS/400. This is the destination library where the compiler creates the *MODULE objects.

Creating the OS/400 Project

These steps create the sample application `HELLO`:

1. Double-click on the **WorkFrame IDE** icon in the VisualAge for C++ for AS/400 product folder.
2. Select **Create new project** from the dialog box that opens, to bring up the Project Smarts notebook.
   a. Select **Default Project** from the Project page
   b. In the **Location** page, enter `C:\EXAMPLE` as the source directory, and `HELLO` as the project name. Two files are added in the `C:\EXAMPLE` directory:
      - `hello.iwo` contains the option selections for the project
      - `hello.iwp` contains the project settings
   c. In the **AS/400** page, enter:
      - The mounted drive designation (`N:`)
      - The destination library `HELLO`
      - The connection environment variables ICCASNAME and ICCASHOST. (These variables are optional. If you leave them blank, default values are used.)
   d. Click on **Done** to create the project and have it display on your desktop.
3. To update the project settings, select **View->Settings**, which opens the Settings notebook:
   a. In the **Directories** page, ensure that `C:\EXAMPLE` is the working directory, and that `N:\QSYS.LIB\MYLIB.LIB` is added to the list of source directories.

b. In the **Environment** page, ensure that `ICCAS_DRIVE=N`, and `ICCAS_DIR=MYLIB` are specified. These environment variables construct full path names to pass dependencies and targets to the MakeMake utility.

4. The Project Smarts default project does not copy any source files to the working directory. To create your own source files, select **Project->Edit** and edit and save the following source file `hello.cpp`:

```
// hello.cpp
#include <iostream.h>
void main() {
  cout << "Hello via cout" << endl;
}
```

   The project container now shows `HELLO.CPP`.

5. Select **Options->Compiler** from the menu bar, to bring up the VACPP400 Compile option notebook:

   a. On the **Processing** page, click on the **Perform compile only** radio button.

   b. On the **AS/400** page, specify `MYLIB` in the **Destination library** entry field.

   c. Click the right mouse button on `HELLO.CPP` in the project container, and select **Compile** from the pop-up menu.

      This starts an editor session displaying the compiler command invocation, and the results of the compilation. If compiler errors are listed, you can double-click on them to bring up the corresponding line in your source code and make the necessary fixes.

      As a result of the successful compile action, the file `HELLO.MODULE` now appears in the project container.

6. To create a program object from `HELLO.MODULE`, select **Options->CRTPGM** from the menu bar, to bring up the CRTPGM option notebook.

   a. On the **Basic** page, enter `MYLIB/HI` in the **Generated program name** entry field.

   b. Specify `MYLIB/HELLO` in the **Module name(s)** entry field on the same page.

   c. Click the right mouse button on `HELLO.MODULE` in the project container, and select **CRTPGM** from the pop-up menu.

      The CRTPGM command invocation and its results are displayed in the monitor of the edit session.

      The file `HI.PGM` now appears in the project container.

7. To create a makefile for your project, select **Project->MakeMake**.

   - Select the VACPP400 and CRTPGM actions, and highlight the source file `HELLO.CPP`.
   - Generate and save the makefile. As a result of the MakeMake action, `HELLO.MAK` now appears in the project container.

   The makefile can be run from **Project->Make**.

Chapter 3. Editing, Browsing, and Debugging from WorkFrame

This section introduces tools to edit, browse, and debug OS/400 applications from the Windows workstation environment.

See the *VisualAge for C++ for Windows User's Guide* for a detailed description of the VisualAge Editor and the VisualAge Browser.

Creating and Editing Source Files

Use the VisualAge Editor to:

- Create and edit source files.
- Start other tools, such as the compiler, browser, and debugger.

Start the VisualAge Editor in one of the following ways:

- Type `iedit` on a Windows command line.
- Double-click on the **VisualAge Editor** icon in the VisualAge for C++ for AS/400 **Tools** folder.
- Start the editor from within a WorkFrame project.

Starting the VisualAge Editor from within WorkFrame

To start the editor from the project window, choose one of these methods:

- To run the editor without loading a particular file, select **Edit** from the **Project** pull-down menu on the toolbar.
- To open an existing file, highlight the file in the project container, click mouse button 2, and select **Edit** from the popup.

Browsing Source Code

The VisualAge Browser is a development tool that helps you navigate through source code:

- List program components by type, such as all classes.
- Graphically view relationships between program components, such as class-inheritance hierarchies, function calls, and included files.
- View and edit source code associated with a selected program component.
- Look up online documentation for a class or a class member.

Start the VisualAge Browser in one of the following ways:

- Type `ibrs` on a Windows command line.
- Double-click on the **VisualAge Browser** icon in the **Tools** folder.
- Start the browser from within the VisualAge Editor.

- Start the browser from within the cooperative debugger.
- Start the browser from within a WorkFrame project.

Starting the Browser from within WorkFrame

From inside a WorkFrame project, you launch the Browse action in one of the following ways:

- Pull down the **Project** menu and select **Browse**. Specify the name of the file you want to browse in the Load Database window of the browser.
- Highlight the file(s) you want to browse, then pull down the **Selected** menu from the menu bar, and select **Browse**.
- Highlight the file(s) you want to browse, then press mouse-button 2 to display the pop-up menu of possible actions for one of the files, and select **Browse**.

Restrictions When Browsing OS/400 Applications

The VisualAge Browser accepts only the following input file types when you browse OS/400 applications:

- Compiler-generated browser files (.PDB)
- C++ source files (.CPP).

Notes:

1. Compile source files with the `/Fb` compiler option to create browser files with the extension `.PDB`.
2. You cannot browse the following OS/400 file types:
   - *MODULE
   - *PGM
   - *SRVPGM

Debugging OS/400 Applications

A debugger helps you detect and diagnose errors in your code. The cooperative debugger supplied by VisualAge for C++ for AS/400 debugs an application from the graphical user interface of your Windows workstation, while the application is running on AS/400.

You can perform the following tasks:

- Set breakpoints in your program
- Step through the statements of your program
- Display and change the values of variables
- Examine the call stack.

There are several ways to invoke the cooperative debugger:

- Type the `idebugas` command on a Windows command line.
- Double-click on the **Cooperative Debugger** icon in the **Tools** folder.
- Invoke the **Debug** action from within a WorkFrame project.

You can also debug your OS/400 applications with the ILE Source Debugger. You start the ILE source debugger from the AS/400 system on which your application is running. See Part 4, “Debugging Your Program” on page 173 for a full description of the capabilities and functionality of both debuggers.

Starting the Cooperative Debugger from within WorkFrame

From within a WorkFrame project you launch the **Debug** action in one of the following ways:

- Pull down the **Project** menu and select **Debug**.
- Highlight the file(s) you want to debug, then pull down the **Selected** menu from the menu bar, and select **Debug**.
- Highlight the file(s) you want to debug, then press mouse-button 2 to display the pop-up menu of possible actions for one of the files, and select **Debug**.

If you have not set the environment variable ICCASDEBUGHOST in your **autoexec.bat** file, you must supply information about the target AS/400 through the /e*hostname* debugger option in one of two ways :

1. In the **Parameters** entry field of the Debug notebook type /e*hostname* where *hostname* represents the TCP/IP name of the target AS/400 system.
2. On the Windows command line type `idebugas /e`*hostname* where *hostname* represents the TCP/IP name of the target AS/400 system.

When you start the debugger, the AS/400 Logon window opens, prompting you for information. When you have successfully signed on to AS/400, a Startup Information window opens. Proceed as follows:

1. In the Startup Information window enter the name of the program you want to debug, and the name of your interactive AS/400 job.
2. Start the application you want to debug on AS/400 by typing `CALL` *program-name* on an AS/400 command line, where *program-name* represents the name of the program you want to debug.
3. Click on **OK** in the Debugger Message window.

Part 2. Compiling Your Program

The sections in this part provide information on how to compile an OS/400 application from a Windows workstation with the VisualAge for C++ for AS/400 cooperative compiler.

Chapter 4. Starting the Compiler

The cooperative compiler compiles C++ applications from your Windows workstation, targeting the Integrated Language Environment on AS/400.

This section briefly introduces the concept of cooperative compiling. It explains how to establish a connection between a Windows workstation and AS/400, and how to invoke the compiler to create OS/400 module objects.

If you are new to the AS/400 system, see *ILE Concepts* for information about modules and module creation in the Integrated Language Environment on AS/400.

Compiling across Operating Systems

The compiler compiles C++ source code across two different operating systems:

- The compiler front end runs on your Windows workstation and generates an intermediate code file from each C++ source file. This intermediate code file is transparently transferred to an AS/400 system of your choice.
- The compiler back end finishes compilation of the intermediate code file on the AS/400 system. There, it produces an OS/400 module object (*MODULE) for each intermediate file. Module objects can be bound into program objects (*PGM) that run in the Integrated Language Environment on AS/400. Module objects can also be bound into service program objects (*SRVPGM), to be called by ILE programs.

This approach lets you develop OS/400 C++ applications directly from a Windows workstation, where you can use graphical user interface (GUI) tools. At the same time, you take advantage of AS/400's storage capacity, security, and data integrity.

Figure 3 on page 26 shows how the compiler compiles and binds source files across the two different platforms.

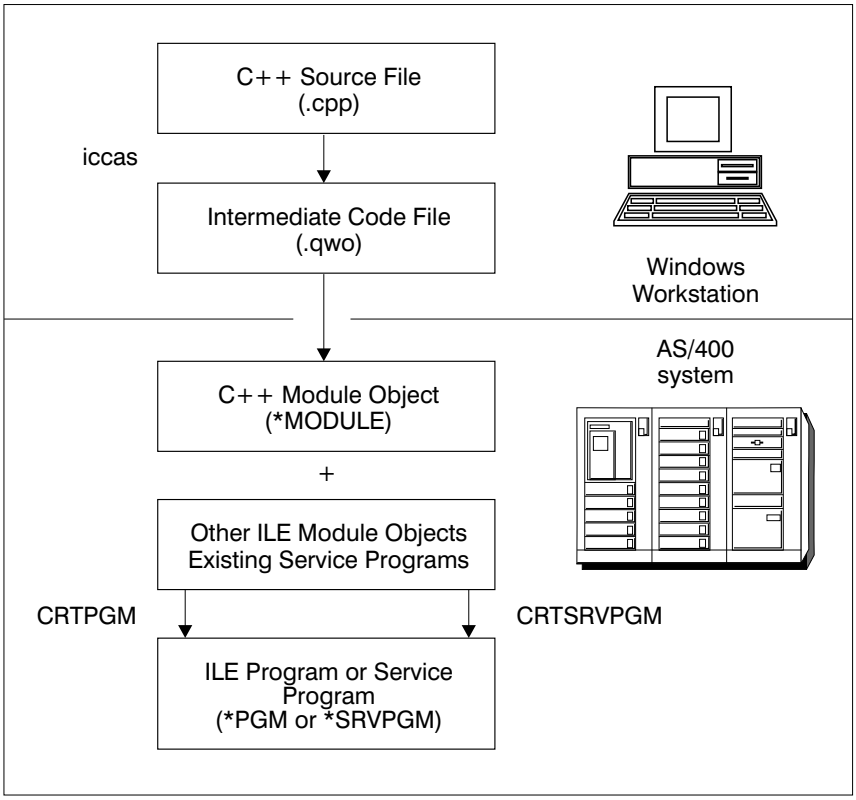


Figure 3. Compiling and Binding across Operating Systems

Communication between Windows Workstation and AS/400

To allow the entire compilation process to take place, you must start a communication session between the Windows workstation and the AS/400 system of your choice prior to invoking the compiler. Compiling after having established a connection between a Windows workstation and AS/400 is called *compiling in connected mode*.

You may also compile your code without being connected to AS/400. This is called *compiling in disconnected mode*. In disconnected mode, compilation ends with the creation of intermediate code files, which remain located on the Windows workstation. This mode is useful when you cannot log on to an AS/400 system, or when you want the compiler to perform a syntax check only. See “Compiling in Disconnected Mode” on page 39 for details on compiling without an AS/400 connection.

Creating and Transferring Intermediate Code Files

On the Windows workstation, each source file `filename.cpp` is compiled into an intermediate code file `filename.qwo`. This intermediate file is then transferred to AS/400, where the compiler back end processes it into a module object `FILENAME`. This module

object can be bound either into a program or into a service program object. By default, modules are bound into program objects.

The transfer of files to AS/400 and the program or service program creation (depending on the compiler option chosen) happen automatically, provided you establish a connection between the Windows workstation and AS/400 prior to invoking the compiler. If such a connection is not active, compilation ends after the creation of the intermediate file `filename.qwo` on your Windows workstation.

If you want to compile source code into a module object only, and not bind it into a program or a service program, specify the `/C` compiler option.

Processing Multiple Source Files

If you invoke the compiler with multiple source files, processing takes place as follows:

- If an AS/400 connection is established, the compilation of the second source file does not start until a module object is created for the first source file.
- If you specify the compiler option `/Fw`, the compilation of the second source file does not start until the creation of the first intermediate code file is complete.
- In disconnected mode, the second compilation starts after the successful creation of the first intermediate code file.
- If the compilation of a source file does not complete successfully, the compiler returns an error message. Subsequent files are not compiled.

See “Determining Compiler Output” on page 38 for more information on compiling in connected or disconnected mode, and on creating only intermediate code files.

Compiling Source Code into Module Objects

The compiler generates module objects from source files. An object of type *MODULE is the basic building block of an OS/400 ILE program or service program.

Contents of Module Objects

A module may consist of one or more procedures (functions). The module object includes a list of the imports and exports referenced within the module and may contain data that can be exported to other modules. It may include debug data, if this was requested at compile time as a compile option.

Binding Module Objects

A module cannot be run by itself. Instead, one or more modules bound together create a program object (type *PGM) that can be run (if at least one of the modules contains a program entry procedure). By default, invoking the compiler results in the creation of a program object. Alternatively, you can bind modules to create service programs (type *SRVPGM).

See Chapter 6, “Controlling Compiler Output” on page 57, Chapter 8, “Creating OS/400 Programs” on page 109, and Chapter 9, “Creating a Service Program” on

page 121 for details on how to compile source code into modules, and create programs and service programs.

Advantages of Combining Module Objects

The ability to combine modules allows you to:

- Reuse pieces of code across applications.
- Maintain shared code with little chance of introducing errors to other parts of the overall program.
- Manage large programs more effectively by dividing them into parts that you write, test, and debug separately. If the program needs to be enhanced, you only recompile those modules that have changed.
- Create mixed-language programs, binding together modules written in the ILE language that is most appropriate for the task.

Establishing an AS/400 Connection before Compiling and Binding

Before you can compile and bind an OS/400 C++ application from your Windows workstation, you must first establish a connection with AS/400, using the CTTCONN command.

Note: The CTTCONN command requires the services provided by Client Access/400 Optimized for Windows Client (hereafter called Client Access for Windows). Make sure you have configured your target AS/400 in Client Access for Windows, and have started a Client Access for Windows connection.

Setting Up Host Server Sessions on AS/400

The CTTCONN command connects your Windows workstation to AS/400 and starts a host server job on AS/400.

By default, host server jobs on AS/400 are reused. To ensure that you have a consistent start-up state, ask your system administrator to set the maximum use of prestart jobs (*MAXUSE*) that can be active at the same time for this prestart job entry to 1. The corresponding AS/400 command is:

```
CHGPJE SBSD(QSYS/QSYSWORK) PGM(QIWS/QZRCSRVS) MAXUSE(1)
```

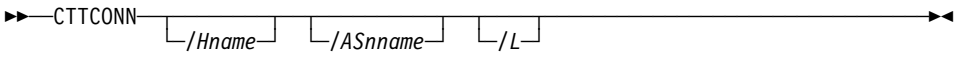
To change the prestart job to have batch type characteristics, issue the AS/400 command

```
CHGCLS CLS(QGPL/QCASERVR) RUNPTY(50) TIMESLICE(5000) DFTWAIT(120)
```

Using the CTTCONN Command to Connect to AS/400

You establish a connection between your Windows workstation and AS/400 by issuing the CTTCONN command from the command line in any window that has the environment settings for the compiler. You can invoke this command more than once, to establish more than one connection at a time between your Windows workstation and one or several AS/400 systems.

The syntax for the CTTCONN command is:



You can enter /? or ? as the first parameter to the CTTCONN command to display the command syntax.

Parameters for the CTTCONN Command

The CTTCONN command takes three parameters to uniquely identify a connection between a Windows workstation and AS/400, and to open a Job window for the connection

/H*name* This parameter identifies the AS/400 system to which you are connecting. The name can be a fully qualified network name, such as:

- APPN.SYSTEM (in SNA)
- SYSTEM.NETWORK.COM (in TCP/IP)

Instead of typing /H*name* each time you invoke the CTTCONN command, you can omit this parameter and specify its value in the ICCASHOST environment variable.

In the Windows 95 operating system you can set the ICCASHOST environment variable permanently in your **autoexec.bat** file, or at the beginning of each session through the command:

```
SET ICCASHOST=<system name>
```

In the Windows NT operating system, you can set the ICCASHOST variable in the System window.

If you later want to override the setting in the ICCASHOST environment variable, specify /H*name* for your connection when you invoke the CTTCONN command.

/ASn*name* This parameter is a user-defined unique name for the connection. The name you choose must be a string that does not exceed 10 alphanumeric characters.

Instead of typing /ASn*name* each time you invoke the CTTCONN command, you can omit this parameter and specify the connection name on the ICCASNAME environment variable.

In the Windows 95 operating system, you can set the ICCASNAME environment variable permanently, in your **autoexec.bat** file, or at the beginning of each session through the command:

```
SET ICCASNAME=<user-defined name>
```

In the Windows NT operating system, you can set the ICCASNAME environment variable permanently in the System window.

If you later want to override the setting in the ICCASNAME environment variable, specify /ASn*name* for your connection when you invoke the CTTCONN command.

The following rules apply:

1. If ICCASNAME is set, the /ASn*name* parameter on the CTTCONN command is not necessary.
2. If /ASn*name* is used, it overrides the name specified by ICCASNAME.
3. If /ASn*name* is not specified and ICCASNAME is not set, an error results.

/L This parameter opens a Job window when a connection is made. From the Job window you can:

- Modify the properties of the connection batch job associated with the Job window.
- View the job log of the connected job.

Working with Multiple AS/400 Connections

You can start connections with several different AS/400 systems, or you can start several connections with the same AS/400. Establish each connection separately through the CTTCONN command.

The command `CTTCONN /Hhost1 /ASnserver1` establishes an AS/400 connection called `server1` with AS/400 `host1`.

The command `CTTCONN /Hhost1 /ASnserver2` establishes a second connection called `server2` with the same AS/400.

The command `CTTCONN /Hhost2 /ASnserver3` establishes a third connection called `server3` with a second AS/400, `host2`.

Special Considerations When Working with Multiple Connections

When working with multiple connections between your Windows workstation and one or more AS/400 systems, consider:

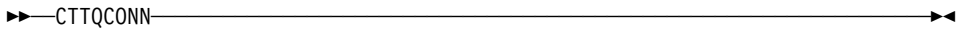
- Connection names must be unique. You cannot use the same connection name for two connections, even if you are connecting to two different AS/400 systems.
- If you run multiple connections in the same Windows session, you can use the environment variable ICCASNAME only once. For all other connections, you must override ICCASNAME with /ASn*name*.

  If you are connecting to several AS/400 systems, you must either set the environment variable ICCASHOST for each connection, or specify the /H*name* parameter each time you invoke CTTCONN.
- To avoid conflicts in case of multiple connections, you can run each connection in its own Windows session. You can set the environment variables ICCASHOST and ICCASNAME for each session, independently.

Note: Environment variables set in **autoexec.bat** or in the System window affect all Windows sessions, unless you override them by using the SET command, or by using the parameters `/H`*`name`* and `/AS`*`nname`*.

Querying Existing Connections through CTTQCONN

To find out which connections are currently active, use the CTTQCONN command. The syntax for this command is:



The CTTQCONN command does not take any parameters.

When you type `CTTQCONN` in a Windows session that has the environment settings for the compiler, a list of all established connections is displayed. At a minimum, the PATH environment variable must be set to `c:\CTTASW\BIN`, where `c:` is the drive on which VisualAge for C++ for AS/400 is installed.

Use the CTTQCONN command to display connection names already in use, or to check on the status of a connection.

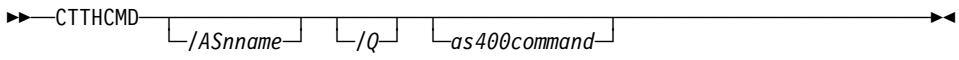
Note: A connection either exists or does not exist. If there is an error in the communication between the Windows workstation and AS/400, the connection goes down. It does not remain in a state of error.

Issuing AS/400 CL Commands from Windows through CTTHCMD

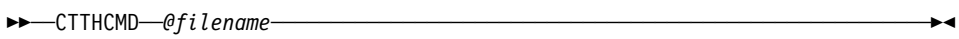
When you have established an AS/400 connection through CTTCONN, you can use the CTTHCMD command to issue AS/400 Command Language (CL) commands from the command line of your Windows workstation. Any commands that can be run in batch mode on AS/400 can be run using the CTTHCMD command from the Windows workstation. You can change the current library list on AS/400, add library list entries, or create programs and service programs.

Do not use CTTHCMD to issue AS/400 CL commands that would require an AS/400 display, such as DSPLIBL, or STRSEU, because these AS/400 displays are not visible from the Windows workstation.

You can enter `/?` or `?` as the first parameter to the CTTHCMD command to display the command syntax. The syntax for the CTTHCMD command is:



or



Parameters for the CTTHCMD Command

The CTTHCMD command takes the following parameters:

/ASnname The /ASn*name* parameter is a user-defined name for the connection that you can use to issue an AS/400 CL command.

See “Using the CTTCONN Command to Connect to AS/400” on page 28 for a description of /ASn*name*.

/Q The /Q parameter suppresses informational messages. It is an optional parameter.

as400command The as400command parameter specifies the AS/400 CL command you want to issue from your Windows workstation.

The position of /ASn*name* and /Q can be interchanged, but both must appear before the AS/400 command. The AS/400 command is parsed as is, without escape sequence processing. The command:

```
CTTHCMD /ASnmyconnection CHGCURLIB MYLIB
```

changes the current library on AS/400 implied by the connection name myconnection to MYLIB.

If the AS/400 command is too long to be placed conveniently on the Windows command line, the entire argument list to CTTHCMD may be put into a response file. In this case, the @filename argument can be used. The syntax is:

```
CTTHCMD @filename
```

The at sign (@) must precede the filename. An AS/400 command that is split into lines in the response file is separated by at least one blank:

```
crtpgm abc actgrp(my
actgrp)
```

is processed as

```
crtpgm abc actgrp(my actgrp)
```

instead of

```
crtpgm abc actgrp(myactgrp)
```

Example of Issuing an AS/400 Command with CTTHCMD

The following command file is intended to create a program so that it can run on the AS/400 system MYSYS001.

```
set iccasname=os400a
set iccashost=mysys001
cttconn
ctthcmd chgcurlib testlib
ctthcmd crtpgm *curlib/gf module(gf)
ctthcmd call testlib/gf
ctthcmd dltpgm testlib/gf
cttdis
```

If the CTTHCMD command fails on AS/400, an AS/400 error code and message are displayed on the Windows workstation, unless you have used the /Q option.

Note: You cannot access the interactive message prompts or help provided on AS/400.

Renaming the CTTHCMD Command

You can rename or copy the **ctthcmd.exe** file to a different name. In this case, the command behaves differently when it is invoked. The /AS*nname* and /Q parameters, or the @filename argument are no longer recognized. The AS/400 command issued is the new name of the .EXE file appended with the argument. For example, if the file **ctthcmd.exe** is copied to crtpgm.exe

```
CRTPGM A MODULE(A B)
```

issued from the Windows workstation sends the following command to AS/400:

```
CRTPGM A MODULE(A B)
```

Note: You must update copies if a new version of CTTHCMD is released. Failing to do so may result in an executable not working properly with the new version.

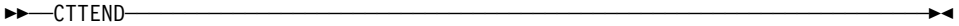
Ending an AS/400 Connection

When you no longer require the connection(s) that you have established through the CTTCONN command, you must either disconnect from AS/400 or end all connections.

The commands you can use are CTTEND and CTTDIS. You can enter /? or ? as the first parameter to both commands to display the command syntax.

Using the CTTEND Command

CTTEND affects all connections. The syntax for the command is:

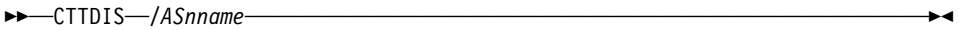


This command does not take any parameters. Type CTTEND on a Windows command line to end all connections between your Windows workstation and any AS/400 system that you have established through the CTTCONN command.

CTTEND not only disconnects your Windows workstation from all AS/400 systems, it also closes the connection window associated with the connections.

Using the CTTDIS Command

CTTDIS affects a single connection, and does not close the connection window associated with the connection. The syntax for the command is:



The parameter AS*nname* is required, unless you have set the ICCASNAME environment variable. /AS*nname* specifies the name of the connection you wish to disconnect. Type: CTTDIS /ASnname on the Windows command line to disconnect only the AS/400 connection identified by the/AS*nname* parameter. The command does not disconnect any other connections that may be active.

Use CTTDIS if you want to selectively disconnect one of a number of active connections. If you only have a single connection, use CTTDIS followed by CTTEND.

Recovering from Restarting a Windows Workstation

If you need to restart your Windows workstation while connected to AS/400, without going through the normal shutdown procedure, the AS/400 job associated with your connection may remain active. If this is the case, you must terminate the AS/400 connection explicitly through the Work with Active Jobs (WRKACTJOB) command from an AS/400 session.

Invoking the Compiler

The **iccas** command invokes the compiler, which takes C++ source code as input and produces an intermediate code file, a preprocessed file, or a module object, according to the compiler options specified. The command also invokes the AS/400 binder to bind the module object into a program or a service program object.

You can issue the **iccas** command in the following ways:

- Invoke the compiler from WorkFrame, if you prefer working with interactive compile and bind option dialogs.
- Invoke the compiler from a makefile, if you need to compile multiple files that depend on each other.
- Invoke **iccas** from a Windows command line, like any other Windows program.
- Invoke **iccas** with a response file.
- Use the `system` function, if you want to invoke the compiler from within another program. Insert the statement `system("iccas myfile.cpp");` into any other C or C++ program.

  Note: Under the Windows 95 operating system, *system()* always returns 0 and cannot be used to check a compiler return code.

Whichever approach you use, do not forget to establish communication between your Windows workstation and AS/400 prior to invoking the compiler, if you want the compilation to result in the creation of a *MODULE object on AS/400. See “Establishing an AS/400 Connection before Compiling and Binding” on page 28 for details on establishing a connection between your Windows workstation and AS/400.

Compiling from within WorkFrame

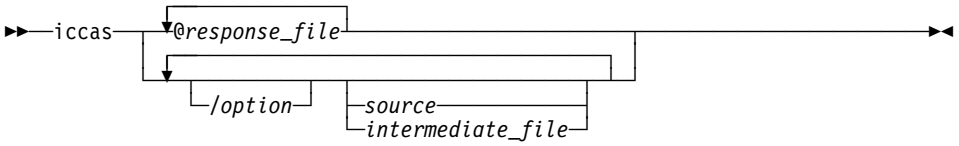
To use the compiler through WorkFrame, do the following:

1. Double click on the WorkFrame icon to open WorkFrame
2. On the initial WorkFrame dialog, either open an existing project, or create a new project. These actions are also choices on the WorkFrame **Project** pull-down menu. Once a project has been opened or created, its files are listed in the WorkFrame window.

3. Customize the compile and bind options from the **Options** pull-down menu, if you do not want the defaults. The **Options** menu contains choices that allow you to specify options for bind and other actions. Use Build Smarts to easily set build options, such as debug, that affect more than one action. If you have not already done so, you may also want to customize the project settings by selecting **Settings** on the **View** pull-down menu.
4. Select **Build** from either the **Project** pull-down menu or the project toolbar. Your project is built using the compiler as required.

Compiling from the Command Line

The syntax for the **iccas** compiler invocation command is:



Depending on how you want to compile your files and how you have set up your environment variables many of the parameters used with the **iccas** command are optional when you issue the command from a Windows command line. See "Windows Environment Variables for Compiling" on page 47 for information on ICCAS and other environment variables.

To compile the source file `myprog.cpp`, enter the following command on a Windows command line:

```
iccas myprog.cpp
```

An intermediate code file, `myprog.qwo`, is created on the Windows workstation, and, if an AS/400 connection has been established, a module object `MYPROG` and a program object `MYPROG` are created on AS/400.

For a list of all the compiler options see "Compiler Options Summary" on page 77. This summary is also available in the online version of this guide. You can jump directly to the summary (or any other topic) from the command line with the **iview** command, followed by the filename `cttug.inf` and the topic:

```
iview cttug.inf compiler options
```

To access an options summary from the command line with the `/?` option, type:

```
iccas /?
```

A list of compiler options is printed to your Windows workstation display, but you can use the Windows redirection symbols to redirect it to a file:

- In the Windows 95 operating system, use the command `iccas /? > mylist` to redirect output from `stderr` to a file `mylist`.

- In the Windows NT operating system, use the command `iccas /? 2> mylist` to redirect the file handle 2 (`stderr`) to a file `mylist`.

Note: The listing generated by these commands is not intended to be used as a programming interface.

Compiling Files Without a File Extension

When you invoke the compiler without specifying a file extension, for example by typing `iccas abc` on a Windows command line, the compiler assumes that you do not want to compile the source file `abc.cpp`, but want to call the binding step alone to create a *PGM or a *SRVPGM object on your AS/400 from module `ABC`.

The compiler adds the name of the module `ABC` to the module list for the CRTPGM or the CRTSRVPGM command. The binder issues an error message if this module does not exist. Make sure you specify the appropriate file extension when you want to compile a source code file.

Consider the following invocation cases:

- The command `iccas abc` results in the binder invocation `CRTPGM PGM(ABC) MODULE(ABC)`.
- The command `iccas /b"crtsrvpgm abc" abc` results in the binder invocation `CRTSRVPGM SRVPGM(ABC) MODULE(ABC)`.
- The command `iccas /b"crtpgm abc actgrp(xyz)" abc def` results in the binder invocation `CRTPGM PGM(ABC) ACTGRP(XYZ) MODULE(ABC DEF)`.

Note: In the examples, above, the compiler queries AS/400 for the name of the current library. If there is none, or if the query is not successful, the library defaults to `QGPL`. You can also specify a library name on the `/ASl` compiler option.

Compiling Multiple Source Files into Modules

To compile programs that use more than one source file, specify all the filenames on the Windows command line. For example, to compile a program with three source files, `mainprog.cpp`, `subs1.cpp`, and `subs2.cpp`, type:

```
iccas mainprog.cpp subs1.cpp subs2.cpp
```

The compiler creates three module objects, `MAINPROG`, `SUBS1`, and `SUBS2`, on the AS/400 system you are connected to, as well as one program object `MAINPROG`.

In case the compilation of one source file into a module object *MODULE fails, the compiler will **not** attempt to compile the remaining source files, and a program object is not created.

Note: If you want to create a separate program object for each source file, you must compile each source file separately.

For example, to compile the three source files, `file1.cpp`, `file2.cpp`, and `file3.cpp`, into three separate program objects, invoke the compiler three times, once for each source file:

```
iccas file1.cpp
iccas file2.cpp
iccas file3.cpp
```

Each successful compilation creates one module object, and one program object.

Compiling C Source Files With the C++ Compiler

C source files are compiled as C++ files. They may not compile correctly without source code modifications to follow C++ syntax and semantics. For example, the command `iccas cprog.c` is interpreted by the compiler as `iccas cprog.cpp`.

See the *C++ Programming Guide* for information on how to prepare C source code for compilation with the cooperative compiler, and for a discussion of portability considerations.

Compiling Using Response Files

Instead of specifying source files and compiler options on the Windows command line, you can use a response file. A response file is a text file that contains a string of options and filenames to be passed to the compiler.

A response file can have any valid Windows filename and extension. To use the response file, specify it with the **iccas** compiler invocation on a Windows command line preceded by the atsign (@). For example:

```
iccas @d:\response.fil
```

No space can appear between the atsign (@) and the filename. You can use multiple response files, and even nest response files within a response file.

The response file does not contain the compiler invocation **iccas**. If the response file contains the string: `/Sa /Fl inventory.cpp`, the command `iccas @d:\response.fil` would give the following Windows command line:

```
iccas /Sa /Fl inventory.cpp
```

You can specify response files in addition to other input on the Windows command line. In this case, the compiler evaluates all options and filenames specified in the response file, on the Windows command line, and in the ICCAS environment variable.

The command string in a response file can span multiple lines. No continuation character is required. The string can also be longer than the limit imposed by the Windows command line. In some situations you may need to use a response file to accommodate a long Windows command, such as when you compile C++ code containing templates.

Note: 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 filename, the compiler does not interpret the file as you intended. For example, given the following response file:

```
/Sa /F
l inventory.cpp
```

the compiler constructs the command `iccas /Sa /F l inventory.cpp` The compiler generates an error message that the `/F` option is not valid, and then tries to create a program `L *PGM` from the modules `L *MODULE` and `INVENTORY *MODULE`. Program creation fails because `L *MODULE` cannot be found.

Compiling from a Makefile

When your application comprises a number of source files, use a makefile to organize the sequence of actions (such as compiling and binding) required to build your project. You can then invoke all the actions in one step. This approach 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.

You can write the makefile yourself, or you can use the WorkFrame utility MakeMake to write the makefile for you. For information on MakeMake:

1. Open the WorkFrame folder
2. Open the Tools folder, in the WorkFrame folder
3. Double click on the MakeMake icon to open the MakeMake window
4. Select **Help->General help**

See Chapter 11, “Creating a Makefile” on page 149 for details on writing your own makefile.

Determining Compiler Output

The compiler can produce several types of output. Depending on the nature of your project, and your stage in the development process, you may want to:

- Create a program using compiler option defaults
- Create a program using specific compiler options
- Compile in connected mode
- Compile in disconnected mode
- Create an intermediate code file.

Creating a Program Using Compiler Option Defaults

To compile your program with compiler option default settings, simply invoke the compiler without specifying any compiler options. (This assumes that you have also not specified any options in the ICCAS environment variable.)

In the example invocation `iccas test.cpp`, the source file `test.cpp` located in your current directory on your Windows workstation is compiled into a module object `TEST` and bound into a program object `TEST`, both located in your current library on AS/400.

Creating a Program Using Specific Compiler Options

To override default compiler options, simply specify the options you wish to use when you invoke the **iccas** command. For example, the command

```
iccas /ASlmylib /Fomyserv /B"crtsrvpgm srvpgm(mylib/myserv)" myfile.cpp
```

results in source file `myfile.cpp` being compiled into a module object `MYSERV` and bound into a service program `MYSERV` in library `MYLIB`, on AS/400.

See Chapter 7, “Setting Compiler Options” on page 71 for a complete list of compiler options and their default settings.

Compiling in Connected Mode

Compiling in connected mode means that a connection has been established between your Windows workstation and AS/400, before you invoke the compiler.

This connection needs to be established only once, at the beginning of your compile session. See “Establishing an AS/400 Connection before Compiling and Binding” on page 28 for the steps required to connect your Windows workstation to AS/400.

When you invoke the compiler in connected mode, the invocation command **iccas** creates one intermediate code file for each C++ source file, and transfers it for you to AS/400. There, for each intermediate code file, a module object is created, which may be bound into either a program or a service program, depending on the compiler options you have specified.

In the following example, it is assumed that you have established a connection between your Windows workstation and AS/400, through the CTTCONN command.

The compiler invocation

```
iccas /ASlmylib /Fomyprog /B"crtpgm pgm(mylib/myprog)" myfile.cpp
```

results in source file `myfile.cpp` being compiled on the Windows workstation into an intermediate code file `myfile.qwo` which is transferred to AS/400, where a module object `MYPROG` is created. This module object is bound into a program object `MYPROG` in library `MYLIB`.

Compiling in Disconnected Mode

Compiling in disconnected mode allows you to compile and syntax check your code without being connected to an AS/400 system. In this case, the compilation process takes place on the Windows workstation only, and ends with the creation of an intermediate code file for each C++ source file that is compiled. The intermediate file is not transferred to an AS/400 system, because an AS/400 connection has not been established through CTTCONN.

Before you can further process the resulting intermediate code files, you must connect your Windows workstation to AS/400 with the CTTCONN command. When you invoke the compiler, specify the names of the intermediate code files instead of the source files to create modules, and programs or service programs on AS/400, depending on the compiler options you choose.

Accessing Header Files in Disconnected Mode

When you compile in disconnected mode, the compiler normally does not have access to the header files located on AS/400, unless you have mounted the AS/400 integrated file system as a network drive with the command:

```
NET USE I: \\hostname\QIBM\INCLUDE
```

where `I:` is a drive of your choice.

To make all header files located on AS/400 available to the compiler do the following, before compiling in disconnected mode for the first time:

1. Copy all header files from the following AS/400 libraries onto your Windows workstation
   - QCLE/H
   - QSYSINC/H
   - QSYSINC/MIH
2. Alter your include paths for these header files. You can use the compiler options `/Xi+` and `/I<path>` to specify the include path for each compilation.

Header files are maintained and updated on AS/400. Make sure you refresh any header files that you download onto your workstation as needed.

Accessing Externally Described Files in Disconnected Mode

In disconnected mode, externally described files located on AS/400 are normally not accessible to the compiler, with the exception of header files that you have previously created on your Windows workstation through the **#pragma mapinc** directive.

See the *C++ Programming Guide* for details on using **#pragma mapinc**.

Examples of Compiling in Disconnected Mode

In the following examples, it is assumed that you have not established a connection between your Windows workstation and AS/400, through the CTTCONN command.

- The compiler invocation `iccas myfile.cpp` simply results in the creation of an intermediate code file `myfile.qwo` on your Windows workstation.

  This intermediate code file is not transferred to AS/400, because a connection has not been established prior to invoking the compiler.

- The command `iccas /b"crtpgm pgm(mylib/myprog)" myfile.cpp` also results in source file `myfile.cpp` being compiled on the Windows workstation into an intermediate code file `myfile.qwo`. However, compilation halts here, because the interme-

diate code file cannot be transferred to AS/400. The corresponding module and program objects cannot be created.

Creating an Intermediate Code File Only

You may not always want to fully compile source files into module objects and bind them into programs or service programs. If this is the case, you have the possibility to compile C++ source files into intermediate code files only, which remain on your Windows workstation until you process them further.

When you later want to create a module object from an intermediate code file, you specify the name of the intermediate code file on the compiler invocation command.

For example, if you have previously compiled a file `hello.cpp` into an intermediate code file `hello.qwo`, the compiler invocation

```
iccas hello.qwo
```

results, in connected mode, in the transfer of this intermediate code file to AS/400. There, a module object `HELLO` and, by default, a program object `HELLO` are created.

There are several ways to perform a compile such that only an intermediate code file is created.

- Specify `/Fw+` or `/Fw`[*name*], which results in the creation of an intermediate file that is saved for subsequent processing. The compiler does not attempt to create a module object from the intermediate code file.

  The compiler invocation `iccas /FwC:\mydir\hello myfile.cpp` results in the creation of the intermediate code file `hello.qwo` from the C++ source file `myfile.cpp`. `hello.qwo` is located in directory `mydir` on drive `C:`

  The `/Fw` option in conjunction with the `/Asd` option may be useful when you want to be able to work in disconnected mode on a file that needs to retrieve large amounts of information from AS/400

  In this case, create a file that contains all the **#pragma mapinc** directives you need to use, as well as their corresponding `#include` statements, and compile it in connected mode with the `/Fw+ /Asd` options. This ensures that all temporary files are located on Windows , and are available when you later compile in disconnected mode.

  See the *C++ Programming Guide* for details on using **#pragma mapinc**.

- Do not establish a connection to AS/400. Your source file will be compiled into an intermediate file, but the transfer of this intermediate file to AS/400 will fail.

  The compiler invocation

  ```
  iccas myfile.cpp
  ```

  results in the creation of the intermediate code file `myfile.qwo` from the source file `myfile.cpp`.

  The compiler invocation

```
iccas file1.cpp file2.cpp file3.cpp
```

results in the creation of the intermediate code file `file1.qwo` from the source file `file1.cpp`.

Note: In the example above, the compiler does not attempt to compile the subsequent source files `file2.cpp` and `file3.cpp`, because the creation of a module object for `file1.cpp` failed.

- If you do not need an intermediate code file for further processing, you can gain time by specifying the compiler option `/Fc` which results in syntax check only. In this case, the compiler does not attempt to create an intermediate file, nor a module and a program object.

Targeting Multiple AS/400 Run-Time Environments

The compiler can target the following AS/400 run-time environments:

- OS/400 V3R7
- OS/400 V3R6
- OS/400 V3R2

The header files used for each target environment are located in different directories on your Windows workstation. To ensure that the compiler uses the appropriate set of header files in each case, three environment variables are set in your **autoexec.bat** file (for the Windows 95 operating system) or in the System window (for the Windows NT operating system) when you install VisualAge for C++ for AS/400:

- INCLUDE_ASV3R7
- INCLUDE_ASV3R6
- INCLUDE_ASV3R2

Using the INCLUDE_ASV3Rn Environment Variables

When you install VisualAge for C++ for AS/400, the following values are added to your **autoexec.bat** file (for the Windows 95 operating system) or in the System window (for the Windows NT operating system):

- INCLUDE=D:\CTTASW\INCLUDE\ACL
- INCLUDE_ASV3R7=D:\CTTASW\INCLUDE;D:\CTTASW\INCLUDE\ACL; \\hostname\QIBM\INCLUDE
- INCLUDE_ASV3R6=D:\CTTASW\INCLUDE\V3R6M0;D:\CTTASW \INCLUDE\V3R6M0\ACL;D:\CTTASW\INCLUDE;\\hostname\QIBM\INCLUDE
- INCLUDE_ASV3R2=D:\CTTASW\INCLUDE\V3R6M0;D:\CTTASW \INCLUDE\V3R6M0\ACL; D:\CTTASW\INCLUDE;\\hostname\QIBM\INCLUDE

The following list shows when to use each of the INCLUDE_ASV3Rn environment variables:

INCLUDE_ASV3R7

Use the INCLUDE_ASV3R7 environment variable when you compile source code for the AS/400 V3R7 RISC environment. The compiler uses the path specified in the INCLUDE_ASV3R7 environment variable when you invoke the **iccas** command with the default options `/ASv3r6-` and `/ASv3r2-` .

INCLUDE_ASV3R6

Use the INCLUDE_ASV3R6 environment variable when you compile source code for the AS/400 V3R6 RISC environment. The compiler uses the path specified in the INCLUDE_ASV3R6 environment variable when you specify the option `/ASv3r6+` on the **iccas** compiler invocation command.

INCLUDE_ASV3R2

Use the INCLUDE_ASV3R2 environment variable when you compile source code for the AS/400 V3R2 IMPI environment. The compiler uses the path specified in the INCLUDE_ASV3R2 environment variable when you specify the option `/ASv3r2+` on the **iccas** compiler invocation command.

Chapter 5. Controlling Compiler Input

Depending on the size and nature of your project, you may want to vary, or customize, the way you invoke the compiler and select compiler options. This section describes the methods you can use to control input to the compiler.

Naming Input Files

When working with VisualAge for C++ for AS/400, you must follow the file naming conventions for OS/400.

OS/400 filenames must be entered in unquoted form. Every file name can be 10 characters long and must begin with the characters A-Z, $, #, or @. The remaining characters can also include numbers 0-9, underscores (_), and periods (.).

Lower case letters (a-z) are changed to uppercase letters (A-Z) by the AS/400 system. File names used in IBM-supplied AS/400 commands can be no longer than 10 characters. Examples of file names are shown below:

```
A987@.442#  ONE_NAME  LIBRARY_06  $LIBX
```

Using Wild Cards in File Names

You can use wild cards (* or ?) in the name for any input file.

For example, the command `iccas module?.cpp` compiles all source code files that have the extension `.cpp` and begin with `module` plus one additional character, such as `module1.cpp` or `module2.cpp`. The resulting intermediate code files, modules and program objects are given names derived from the source file, such as `module1.qwo` and `MODULE1`.

The command `iccas my*.cpp` compiles all source files that have the `.cpp` file extension and begin with `my`, plus none, one, or several additional characters, such as `myfile` and `myprog`.

You cannot use wild cards in the name for an output file. Either accept the default output filenames, or specify an output filename in full.

Using Reserved Names

Do not give your user-defined functions names that are identical to the name of any library function or external variable defined in the library. Otherwise binding errors between user-defined identifiers (variable names or functions) and C library functions may result.

Do not use an underscore (_) at the start of any of your external names. These identifiers are reserved for use by the compiler and libraries. All internal C++ identifier names begin with an underscore (_), unless they are listed in this or in one of the following manuals:

- *C++ Language Reference*
- *IBM Open Class Library Reference*
- *IBM Open Class Library User's Guide*
- *IBM Access Class Library for OS/400 Reference*
- *IBM Access Class Library for Windows Reference*
- *IBM Access Class Library User's Guide*

If you have an application that uses a restricted name as an identifier, change your code or use a macro to globally redefine the name and avoid conflicts. You can also use the **#pragma map** directive to convert the name, but this directive may not be portable.

Distinguishing between Input File Types

The compiler uses file extensions to distinguish between the different types of files it uses for input. The input file types are:

.CPP	C++ source file
.I	preprocessed file
.HPP	C++ header file
.QWO	intermediate code file
.U	dependency file

The compiler compiles **all** input files as C++ files (.H .C .ABC), except for those with an extension of .QWO and .U. The .QWO files are intermediate code files resulting from an earlier compiler invocation during which you could not, or chose not to, process them further into module and program or service program objects. The .U files are dependency files created by the compiler option `/qmakedep`.

Note: Do not compile two source files with the same filename and a different file extension, such as `myfile.c` and `myfile.cpp`. In both cases, the compiler creates a module `MYFILE`, and the second module overwrites the first.

When you specify intermediate code files on the compiler invocation command, they are not compiled again, but are sent directly to AS/400 to create module and program or service program objects.

Note: When intermediate code files are specified on the compiler invocation command, they are not deleted from the Windows workstation after module and program or service program objects are created on AS/400.

When you specify C++ source files and intermediate code files on the same compiler invocation, no additional processing is performed on the intermediate code files which are sent directly to AS/400 and which cannot incorporate any additional compiler options specified with the current `iccas` invocation.

For example, to create a program from two source files (mainprog.cpp and file2.cpp) and one intermediate code file (file3.qwo), type:

```
iccas /Fb mainprog.cpp  file2.cpp  file3.qwo
```

The compiler creates two intermediate code files mainprog.qwo and file2.qwo. These files, as well as the intermediate code file file3.qwo are transferred to AS/400, where three module objects MAINPROG, FILE2, and FILE3 are created and are bound into a program object MAINPROG.

After the successful creation of program MAINPROG on AS/400, the intermediate code files mainprog.qwo and file2.qwo are deleted from the Windows workstation, while the intermediate code file file3.qwo is not deleted, because it resulted from an earlier compiler invocation.

The /Fb option results in the creation of two browser listing files for mainprog.cpp and file2.cpp. For file3.qwo a browser listing file is not created,

Windows Environment Variables for Compiling

The compiler makes use of the Windows environment variables to provide path information and default values for compiler options.

Environment variables can be set from the command line. In the Windows 95 operating system, they can also be set in the**autoexec.bat** file. In the Windows NT operating system, they can also be set in the System window. To get to the System window, double-click on **Main** and then on **Control Panel**. Click on **Set** to add a new environment variable to the list of User Environment Variables.

Depending on your selections, the VisualAge for C++ for AS/400 installation program may have set many of the environment variables for the compiler to default values. Optional environment variables such as ICCAS are not set by the installation program.

The following environment variables affect the operation of the compiler:

HELP Lists the directories that the compiler searches for help (.hlp) files.

ICCAS Sets compiler options and filenames. See “Specifying Compiler Options” on page 71 for a more detailed description of the ICCAS environment variable.

ICCASCP Specifies the coded character set identifier (CCSID) of your target AS/400. The value that you specify is the CCSID that is normally used for jobs running in your AS/400 environment.

ICCASHOST Specifies the target AS/400 system. The compiler enables you to extract field descriptions from files located on AS/400, or create *MODULE and *PGM objects on AS/400. Each compile session can only access one particular AS/400. You can use the ICCASHOST environment variable to specify the name of the AS/400 system you want to connect to.

You can override the ICCASHOST environment variable by specifying the /H*name* parameter on the CTTCONN command. See "Establishing an AS/400 Connection before Compiling and Binding" on page 28 for a description of using /H*name* and ICCASHOST with the CTTCONN command.

ICCASNAME Specifies a user-defined alphanumeric string of a maximum of 10 characters for the name of a connection to the AS/400 system you want to connect to. The compiler uses this information as input to the CTTCONN command when you connect to an AS/400 system.

If you want to identify your connection from the Windows workstation to AS/400 by the name CONNABC, you can set the environment variable ICCASNAME to CONNABC before starting the compile session. The default value for ICCASNAME is CON1.

You can override the ICCASNAME environment variable by specifying the /ASn*name* parameter on the CTTCONN command. See "Establishing an AS/400 Connection before Compiling and Binding" on page 28 for a description of using /ASn*name* and ICCASNAME with the CTTCONN command.

INCLUDE_ASV3R2 The compiler searches for the system or user header files in the directories listed by this variable. This environment variable is used when you compile source files with the compiler option /ASv3r2 to target an AS/400 V3R2 run-time environment.

INCLUDE_ASV3R6 The compiler searches for the system or user header files in the directories listed by this variable. This environment variable is used when you compile source files with the compiler option /ASv3r6 to target an AS/400 V3R6 run-time environment.

INCLUDE_ASV3R7 The compiler searches for the system or user header files in the directories listed by this variable. This environment variable is used when you compile source files with the default compiler options /ASv3r6- and /ASv3r2- to target an AS/400 V3R7 run-time environment.

PATH Lists the directory (or directories, separated by semicolons) to be searched for executable files (and DLLs) when the compiler is invoked. This variable should include the directories containing VisualAge for C++ for AS/400 executables, for example the compiler (iccas.exe).

TMP Sets the directory where the 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 com-

piling, 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.

Specifying Source File Names in the ICCAS Environment Variable

In addition to compiler options, you can also put file names into the ICCAS environment variable. If you specify:

```
SET ICCAS=test.cpp check.cpp
```

the command `iccas main.cpp` causes `test.cpp`, `check.cpp`, and `main.cpp` to be compiled in that order. This is the same as the command-line compiler invocation:

```
iccas test.cpp  check.cpp  main.cpp
```

You can also list existing intermediate files (`.qwo`, created with the `/Fw` option) in the ICCAS environment variable. The compiler processes them according to the options specified on the **iccas** compiler invocation command.

Controlling #include Search Paths

The #include preprocessor directive lets you retrieve source statements from secondary input files and incorporate them into your program.

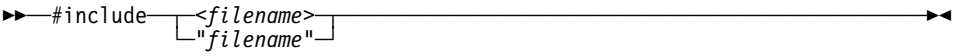
You can nest #include directives in an included file. The compiler allows up to 128 levels of nesting.

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.

Syntax for #include Directives

The following diagram shows the syntax of the #include directive:



In the above figure, angle brackets indicate a **system** `#include` file, and quotation marks indicate a **user** `#include` file.

Specifying #include Filenames

You can specify any valid Windows file name in an #include directive. The file name must be sufficiently qualified for the compiler to be able to locate the file.

If a path name is too long to fit on one line, you can place a continuation character, or backslash (\), 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:\cttas\include\mystuff\subdir1\subdir2\subdir3\myfile.h
```

you could insert one of the following #include directives in your program:

```
#include "c:\cttas\include\mystuff\subdir1\sub\
       dir2\subdir3\myfile.h"
```

or

```
#include "c:\cttas\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.

Controlling the #include Search Paths

You can control the #include search paths in three ways:

- Use the `/I`, `/Xc`, and `/Xi` compiler options on the Windows command line when invoking the compiler
- Use the `/I`, `/Xc`, and `/Xi` compiler options in the ICCAS environment variable
- Specify the search paths in the INCLUDE_ASV3R7, INCLUDE_ASV3R6, and INCLUDE_ASV3R2 environment variables

Searching for #include Files

When the compiler encounters either a user or system #include file statement with a fully-qualified file name, it looks only in the directory specified by that name.

When the compiler encounters a user #include file specification that is not fully qualified, it searches for the file in the following places, in the order given:

1. The directory where the original top-level file was found.
2. Any directories specified using `/I` that have not been removed through the use of `/Xc`. Directories specified in the ICCAS environment variable are searched before those specified on the Windows command line.
3. Any directories specified using the INCLUDE_ASV3R7, INCLUDE_ASV3R6 and INCLUDE_ASV3R2 environment variables, provided that the `/Xi` option is not currently in effect.

When the compiler encounters a system #include file specification that is **not** fully qualified, it searches for the file in the following places, in the order given:

1. Any directories specified using `/I` that have not been removed through the use of `/Xc`. Directories specified in the ICCAS environment variable are searched before those specified on the Windows command line.

2. Any directories specified using the INCLUDE_ASV3R7, INCLUDE_ASV3R6 and INCLUDE_ASV3R2 environment variables, provided that the `/Xi` option is not currently in effect.

Searching for Externally Described Files

Header files created by the **#pragma mapinc** directive follow the same convention as other user #include files. If header files are to be stored in a directory other than the current directory, this directory must be in the search path for #include files, as described for user #include files.

By default, the compiler saves header files generated by the **#pragma mapinc** directive relative to the current directory, in the filename given in the directive. When searching for user include files, as described in “Searching for #include Files” on page 50, the compiler does not search the current directory for the header file specified in the **#include** directive.

When the current directory is specified neither in any of the environment variables nor on the command line, this behavior may cause compilation to fail. For example, source file `source.cpp`, located in directory `mydir1`, specifies the following directives:

```
#pragma mapinc("incfile", "MYLIB/MYFILE(*ALL)", )
Other Code
#include "incfile"
```

If you compile this file from directory `mydir2` with the command:

```
iccas \mydir1\source.cpp
```

compilation will fail, because the header file `incfile` generated by the **#pragma mapinc** directive is stored in the current directory `mydir2`, which is not searched for user include files.

To avoid this situation, you may want to qualify the filename on the **#include** directive as shown below, to make sure the current directory will be searched:

```
#pragma mapinc("incfile", "MYLIB/MYFILE(*ALL)", )
   // other code
#include ".\incfile"
```

See the *C++ Programming Guide* for details on working with externally described files.

Accumulating Search Options

The #include search options are cumulative between the ICCAS and INCLUDE environment variables and the Windows command line. For example, given the following ICCAS and INCLUDE environment variables:

```
ICCAS=/I\payroll
INCLUDE_ASV3R7=c:\employee;\hourly
```

and the following Windows command line:

```
iccas /Xi+ /Ic:\purch test.cpp /Xi- /Xc /Id:\prod f:\moe\min\jay.cpp
```

any system #include files referenced in the file `test.cpp` will be searched for first in the directory `\payroll` and then in the directory `c:\purch`. Because the `/Xi+` option was specified, none of the directories set in the INCLUDE_ASV3R7 environment variable will be searched.

Using the same example, any user #include files referenced in `test.cpp` would be searched for first in the current directory, then in the directory `\payroll`, and then in `c:\purch`.

Any system #include files referenced in the file `f:\moe\min\jay.cpp` will be searched for first in the `d:\prod` directory, then in the `c:\employee` directory, and finally the `\hourly` directory. The directories specified in the INCLUDE_ASV3R7 variable are searched because the `/Xi-` option overrides the `/Xi+` option specified previously. The `/Xc` option removes the directories `\payroll` and `c:\purch` from the current search path.

Any user #include files referenced in `jay.cpp` will be searched for in the following directories, in the given order: `f:\moe\min`, `d:\prod`, `c:\employee`, and `\hourly`.

Setting the Source Code Language Level

The compiler supports three language levels, ANSI, Extended, and Compatible, which are described in "Choosing a Language Level."

To set a language level, do one of the following:

- Use one of the compiler options `/Sa`, `/Se`, or `/Sc`, either on the Windows command line or in the ICCAS environment variable.
- Use a **#pragma langlvl** directive.

Note that a **#pragma langlvl** directive set in your source code overrides any language-level compiler options specified on the Windows command line or in the ICCAS environment variable.

When you set the language level, you also define the macro associated with that level.

Choosing a Language Level

The choice of a language level depends on your application requirements:

- If you want your code to be portable across all ANSI-conforming systems, use the ANSI language level and compile according to a strict interpretation of the ANSI standard.
- If you want to allow constructs compatible with older levels of the C++ language, use the compatible language level to compile older code.
- If you want to allow all VisualAge for C++ for AS/400 language constructs, use the extended language level to compile code that uses extended language features or 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.

You can set the level using compiler options either on the command line or in the ICCAS environment variable, or by using the **#pragma langlvl** directive. Note that a **#pragma langlvl** directive overrides any conflicting compiler options. When you set the language level, you also define the macro associated with that level.

The levels are described in detail below:

ANSI Allow only language constructs 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.

To set this language level, use either the `/Sa` option or **#pragma langlvl(ansi)**, both of which define the macro `__ANSI__`.

Extended Allow all VisualAge for C++ for AS/400 language constructs. These include all constructs that fall under the ANSI Level and the VisualAge for C++ for AS/400 extensions to this standard. See the *C++ 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 for C++ for AS/400 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 ICCAS), use the `/Se` option or **#pragma langlvl(extended)**, both of which define the macro `__EXTENDED__`.

Compatible Allow constructs and expressions that were allowed by earlier levels of the C++ language.

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.
- `typedef`s 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, in the expression `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 operators ++ and --.
- You can use the dollar ($) 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,};`
- 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 payroll :: printCheque(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.

Compile-Time Limits

The following limits apply when you compile C++ code with the cooperative compiler:

- The maximum total size of variables in static storage is 16 773 104 bytes
- The maximum total size of variables in automatic storage for a single procedure is 16 773 104 bytes
- The compiled object cannot contain more than 16 megabytes of encapsulated code.
- The maximum number of arguments for ILE bound calls is 400

- The maximum number of arguments on a dynamic (OS-linkage) program call is 255
- The maximum length of identifiers is 4 kilobytes
- The maximum number of modules that can be specified for CRTPGM or CRTSRVPGM is 300. To bind more than 300 modules, use bind directories which allow you to specify all the modules you want to bind.

Chapter 6. Controlling Compiler Output

The compiler can produce several types of output. This section describes the various output files and the level of information that can be generated.

Distinguishing between Output File Types

Depending on the compiler options you choose, the compiler produces a variety of output files:

- A module object *MODULE for each C++ source file. The module is located on AS/400. It can be created only if you compile in connected mode.
- A program object *PGM, or a service program object *SRVPGM. These objects are created on the AS/400 when you invoke the binder.
- An intermediate code file .QWO for each source file. This file is located on the Windows workstation.
- A listing file .LST for each C++ source file. This file is located on the Windows workstation and contains information about the compilation..
- Precompiled header files, located on the Windows workstation.
- Template-include files, located on the Windows workstation.

  See the *C++ Programming Guide* for more information about these files.
- A preprocessor output file for each C++ source file. This file is located on the Windows workstation. You can use it to find possible programming errors.

  Note: This information is not intended to be used as a programming interface.
- A browser listing file for use by the VisualAge Browser.
- Dependency files that can be included in a makefile.
- Temporary files.
- Diagnostic information about possible programming errors.
- Messages (for example the IBM logo and help messages).
- A return code (0) for a compile without errors.

Creating Intermediate Code Files and Module Objects

On the Windows workstation, the compiler creates an intermediate code file with the extension .QWO for each source code file. In connected mode, this intermediate code file is transferred to the AS/400, where it is processed into a module object (*MODULE).

One or more module objects can be bound to create either program (*PGM) or service program (*SRVPGM) objects. See Code Generation Options for more information on using compiler options to specify the creation of intermediate code files, module objects, and program or service program objects.

Intermediate code files (.QWO) generated by the compiler are deleted when the corresponding module objects have been created on the AS/400. In the following cases intermediate code files remain on the Windows workstation to be used to create module objects at a later time:

- Module creation is not successful
- Module creation is not attempted because of your choice of compiler options
- You compile in disconnected mode

Optimizing Object Code

The compiler can perform many optimizations on object code, such as local optimizations, common subexpression elimination, and loop optimizations.

Generally, the higher the optimization request, the longer it takes to create the object. At run time, highly optimized programs should run faster than corresponding programs created with a lower optimization level. However, I/O intensive programs will show little or no difference.

Optimization techniques apply to individual modules. The AS/400 levels are no optimization (**NONE*), some optimization (**BASIC*), maximum optimization (**FULL*), and level 40 optimization.

These levels have the following meaning:

***NONE** Generated code is not optimized. This level has the shortest compile time. It allows variables to be displayed and modified while debugging.

***BASIC** Some optimization is performed on the code. This level allows user variables to be displayed but not modified while debugging.

FULL Full optimization is performed on the generated code. During a debug session, user variables may not be modified but may be displayed. The presented values may not be the current value of the variable.

Level 40 All optimizations done at level 30 (*FULL) are performed on the generated code. In addition, code is eliminated from procedure prologue and epilogue routines that enable instruction trace and call trace system functions.

You determine the level of optimization for a module through the `/O` compiler option. By default, optimization is turned off (`/O-`).

The following list shows the relationship between AS/400 optimization levels and iccas compiler options:

***NONE or 10** `/O-` or `/O10`

***BASIC or 20** `/O20`

***FULL or 30** `/O30` or `O+`

level 40 `/O40`

Because optimization at level **FULL*, or higher, can significantly affect the order of your program instructions, you may need to be aware of certain debugging limitations. See Part 4, “Debugging Your Program” on page 173 for debug considerations. See “Code Generation Options” on page 98 for more information on using compiler options to control optimization. For more information on how you can optimize your code, see the *C++ Programming Guide*.

Generating Debugger Information

You can specify the debug compiler option `/Ti` to instruct the compiler to include the information necessary for debugging a program in the module object.

When you specify the `/Ti` option, do not turn on optimization (`/O+`, `/Oi+`), because accurate symbol and type information may not be available, due to the optimization process. Debugging information generated with optimization is limited to setting breakpoints at function entry and function exit.

See Debugging and Diagnostic Options for more information on using compiler options to control the generation of debugging information.

See Part 4, “Debugging Your Program” on page 173 for information on debugging AS/400 applications with the cooperative debugger, or with the ILE source-level debugger.

Generating Browser Information.

To create browser information, specify the `/Fb` option which produces .PDB files that the browser can use to display information about your program. See “Output File Management Options” on page 80 for more information on generating browser files.

Creating Program and Service Program Objects

By default, in connected mode, the compiler creates one program object (*PGM) for each compiler invocation. If you specify the `/C` option, the compiler only generates module objects (*MODULE), which you can then bind separately to create a program or a service program.

Use the compiler default settings to create a program object from your source files, or use the `/B"crtsrvpgm srvpgm(xxx/yyy)"` option to create a service program. See Chapter 8, “Creating OS/400 Programs” on page 109 and Chapter 9, “Creating a Service Program” on page 121 for more information on the use of programs and service programs.

See “Other Options” on page 100 for more information on the compiler options you can use to create program and service program objects.

Generating 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.

The listing shows options, any error messages, and a standard header that includes:

- 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 85.

Generating Temporary Files

The compiler creates and uses temporary files during compilation. These files are usually erased at the end of a successful compilation; however, if the compilation is interrupted, they may be left on the disk. They are located in the path specified by the TMP environment variable. If this variable is undefined, the compiler uses the current directory.

For more information on the TMP variable, see “Windows Environment Variables for Compiling” on page 47.

Interpreting Messages and Return Codes

Messages and return codes provide information about the compilation of your files and help you analyze the potential cause for compilation errors.

Error Messages

You can use compiler options to control:

1. The level of error message that the compiler outputs and that increments the error count maintained by the compiler (with the `/Wn` option).
2. How many errors are allowed before the compiler stops compiling (with the `/Nn` option).
3. The diagnostics run against the code (with the `/Wgrp` option).

See “Debugging and Diagnostic Information Options” on page 90 for more information on using the compiler options to control messages.

Redirecting Messages to a File

Messages, warnings, and errors, are written to `stdout`. You may sometimes want to redirect them to a file instead. Use the redirection symbol (>), and a file name; for example:

```
iccas myfile.cpp > myfile.err
```

Interpreting Return Codes

The compiler indicates the highest return code it receives while performing 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`.
12	Error detected; compilation may have been completed, but no module object was created on AS/400
16	Severe error detected; compilation terminated abnormally; no module object was created on AS/400 **Note:** Problems with the transfer of intermediate code files to AS/400 or with the connection itself may result in either return code 12, or 16.
20	Unrecoverable error detected; compilation terminated abnormally and abruptly; no module object was created on AS/400. 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.

Note: If you compile multiple files with one invocation, for example:

```
iccas p1.cpp  p2.cpp  p3.cpp
```

you get the last return code before the command ended. This means:

- If all files compile successfully, the return code indicates the successful completion.
- If not all files compile successfully, the return code indicates the degree of the first failure that occurred and compilation terminates. For example, if compilation of `p2.cpp` returns a code other than 0, the compiler does not attempt to compile `p3.cpp` and returns the code for `p2.cpp`.

Viewing AS/400 Job Logs

Specify the `/L` parameter on the CTTCONN command when you establish a connection between your Windows workstation and AS/400. This parameter opens the Jobs window associated with the connection that is made. From the Jobs window, you can choose to view the job log of the connected job.

Precompiling Header Files

You can improve your compile time by using precompiled header files. Use the options `/Fi+` and `/Si+` together to automatically create and maintain precompiled header files for your application.

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.

The compiler generated 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 header file is only used in cases where the context is the same, it does not have to be reinterpreted every time it is included.

To create a precompiled header file specify the `/Fi` compiler option. This option lets you create or recreate precompiled versions of every source header file used during that compilation.

To use precompiled header files, specify the `/Si` option. You can specify a name for the precompiled header file 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 `csetcpp.pch`

Restrictions When Using Precompiled Header Files

When you use precompiled header files, keep the following restrictions in mind:

- 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.

Inlining User Code

By default, the compiler inlines intrinsic and built-in functions. Inlining means that the compiler replaces the function call with the actual code for the function at the point where the call was made. You can also request that the compiler inline the code for your own, user-defined, functions.

There are three ways to inline user code:

1. Use the function specifier `inline` for functions you want the compiler to inline.
2. Use the `/Oi` option with a *value* parameter to inline functions smaller than the value specified.
3. Define C++ member functions in a class declaration.

Note: Even when you use these methods, you do not have absolute control over whether functions are inlined or not. This decision is up to the compiler. Requesting that a function be inlined makes it a candidate for inlining. This does not necessarily mean that the function will, in fact, be inlined.

Using the inline Keyword

Use the `inline` function specifier to qualify either the prototype or definition of the functions you want to have inlined. For example, `inline int silly(char c);` declares that `silly` is to be inlined.

The `inline` keyword has the same meaning and syntax as the storage class `static`. When you turn inlining on, the keyword also causes the function it qualifies to be inlined. In addition, C++ member functions that are defined in a class declaration are considered candidates for inlining by the compiler.

Using the /Oi Option to Inline User Code

The `/Oi` option controls whether user functions are inlined or invoked through a function call:

`/Oi-` No user code is inlined. This is the default.

`/Oi+` Functions qualified with the `inline` keyword are inlined.

`/Oi`*value* Functions qualified with the `inline` keyword are inlined, as are all other functions that are less than or equal to *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 does not affect the inlining of intrinsic VisualAge for C++ for AS/400 library functions. Instead, you must parenthesize the function call to disable the inlining of intrinsic library functions, for example:

```
(strcpy)(str1, str2);
```

While parenthesizing works with intrinsic functions, there is no way to disable inlining for your user-defined functions, meaning you cannot request that specific functions **not** be inlined.

Parenthesizing does also not work with some library functions that are are implemented as built-in functions. In their case, there is no backing code in the library. Therefore, the compiler must inline them in every case. See the *C Library Reference* for a list of all the intrinsic and built-in library functions.

If you use auto-inlining, the parameter *value* on the `/Oi` command has a range between 0 and 65535 abstract code units (ACUs). The number of ACUs that comprise 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;
   }
```

Messages are generated to tell you which functions are inlined based on the *value* you specified. No messages are generated for functions qualified with the `inline` keyword, 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.

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 nontrivial, 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. 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** 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 declare a function as `inline` after it has been called.
- To use `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 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.
- 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 does not work.
- 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.

Compiling Applications to Run in an AS/400 V3R2 CISC Environment

VisualAge for C++ for AS/400 has been developed primarily to provide the capability of building C++ applications for the new AS/400 Reduced Instruction Set Computer (RISC), beginning with Version 3, Release 6 of the Operating System/400.

Under certain circumstances you may want to compile your applications instead to target Version 3, Release 2 of the Operating System/400 that is available on Complex Instruction Set Computers (CISC). VisualAge for C++ for AS/400 supports both run-time environments, but you should be aware of certain limitations discussed below.

V3R7 Development Environment

The development environment for VisualAge for C++ for AS/400 is a workstation connected to an AS/400 RISC machine running OS/400, Version 3, Release 7. It is **not** possible to connect your Windows workstation to an AS/400 CISC system, and successfully compile C++ source code with VisualAge for C++ for AS/400

Instead, you must compile source files with the /ASv3r2 compiler option, create program (*PGM) and service program (*SRVPGM) objects on the V3R7 RISC system, and then save/restore these objects on a V3R2 CISC system.

V3R2 Run-Time Environment

On the **iccas** compiler invocation command you specify the environment in which you plan to later run your C++ application through the /ASv3r2 compiler option.

/ASv3r2- is the default for the **iccas** command and results in the creation of program and service program objects that run on an AS/400 V3R7 RISC machine. If you specify /ASv3r2+ instead, the resulting program or service program objects can be run on an AS/400 V3R2 CISC machine only.

Note: Applications specifically compiled to run on AS/400 V3R2 CISC machines are **not** upward compatible with V3R7 RISC machines. If you are planning to develop applications for V3R2 that you may later want to run on a V3R7 RISC machine, you must compile such applications twice, using once the /ASv3r2 option, and once the /ASv3r2- option. This results in the creation of two separate applications, each targeting a different run-time environment.

Characteristics of the V3R2 Run-Time Environment

The V3R2 run-time environment has the following characteristics:

- The C++ run-time environment provides support for standard, collection, application support, binary coded decimal classes, and access classes. This run-time environment is not the same as the V3R7 run-time environment, but both are supported by VisualAge for C++ for AS/400.
- The default target file system is the integrated file system.
- Applications can be debugged with the ILE source debugger.
- The language environment supports static initialization.

Limitations of the V3R2 Run-Time Environment

The V3R2 run-time environment has the following limitations:

- There is no support for multiple entry points.
- There is no system support for long procedure names.
- Optimization level 40 is not available; you cannot use compiler option `/O40`.
- Performance explorer tools are not available; you cannot use compiler option `/ASp`.
- You cannot generate information needed for the Run Time Type Information (RTTI) typeid operator and the dynamic_cast operator; you cannot use compiler option `/qrtti`.

Limitations when Debugging Applications Targeting V3R2

When debugging applications that have been compiled to run in a V3R2 CISC environment you encounter the following limitations:

- You must use the ILE system debugger to debug your application.
- You cannot display reference variables.
- You must use hashed names; a listing for mangled names and their corresponding hashed names is provided.
- You cannot look at bit fields in anonymous unions.
- You cannot debug within class scope.
- You may encounter problems with casting.

Compiling Applications to Run in an AS/400 V3R6 RISC Environment

In some cases, you may want to compile your applications to target Version 3, Release 6 of OS/400. The V3R6 run-time environment is not the same as the V3R7 run-time environment, but both are supported by VisualAge for C++ for AS/400. You should be aware of certain limitations discussed below.

V3R7 Development Environment

The development environment for VisualAge for C++ for AS/400 is a workstation connected to an AS/400 RISC machine running OS/400, Version 3, Release 7. It is not possible to establish a connection for the compiler between your Windows workstation and an AS/400 V3R6 RISC system.

Instead, you must compile source files with the `/ASv3r6` compiler option, create program (*PGM) and service program (*SRVPGM) objects on the V3R7 RISC system, and then save/restore these objects on a V3R6 RISC system.

V3R6 Run-Time Environment

On the **iccas** compiler invocation command you specify the environment in which you plan to later run your C++application through the `/ASv3r6` compiler option.

`/ASv3r6-` is the default for the **iccas** command and results in the creation of program and service program objects that run on an AS/400 V3R7 RISC system. If you specify `/ASv3r6+` instead, the resulting program or service program objects can be run on an AS/400 V3R6 RISC system. Applications compiled to run on AS/400 V3R6 RISC systems are upward compatible with V3R7 RISC systems, but are limited to V3R6 functionality.

Note: You should not bind modules compiled for V3R6 to modules compiled for V3R7, because they are bound to different service programs (as a result of the differences between the V3R6 and V3R7 runtimes), which may result in conflicts.

Characteristics of the V3R6 Run-Time Environment

The V3R6 run-time environment has the following characteristics:

- The V3R6 C++ run-time environment provides specific support for standard, collection, application support, access, and binary coded decimal classes. This support is somewhat different from the support offered by the V3R7 C++ run-time environment.
- The default target file system is the Integrated File System.
- Applications can be debugged with the cooperative debugger, or with the ILE source debugger.
- The language environment supports static initialization.

Limitations of the V3R6 Run-Time Environment

The V3R6 run-time environment has the following limitations:

- You cannot generate information needed for the RTTI typeid operator and the dynamic_cast operator; therefore do not use compiler option `/qrtti`.
- You cannot bind modules compiled for V3R7 to modules compiled for V3R6.

Using Pre-Defined Macros to Specify Target Environment

If you may want to compile a C++ application for more than one OS/400 run-time environment, you can imbed the following pre-defined macros in your source code:

- __ASV3R7__ to target V3R7
- __ASV3R6__ to target V3R6
- __ASV3R2__ to target V3R2.

These macros are set to 1, depending on the target release specified through the /ASv3rn compiler options. The following code fragment illustrates how you use these macros in your code.

```
// sample.cpp

#include <iostream.h>

main() {
   #if __ASV3R7__

        cout <<"aaaaa" << endl;

   #elif __ASV3R6__

        cout << "bbbbb" << endl;
   #else

        cout << "ccccc" << endl;

   #endif

}
```

- If you compile the source file `sample.cpp` with the default options /ASv3r6- and /ASv3r2-, the pre-defined macro __ASV3R7__ is set to 1, and the resulting program `SAMPLE` can be run in the V3R7 runtime environment, producing the output `aaaaa`.
- If you compile the source file `sample.cpp` with the options /ASv3r6+ and /ASv3r2-, the pre-defined macro __ASV3R6__ is set to 1, and the resulting program `SAMPLE` can be run in the V3R6 runtime environment, producing the output `bbbbb`.
- If you compile the source file `sample.cpp` with the options /ASv3r6- and /ASv3r2+, the pre-defined macro __ASV3R2__ is set to 1, and the resulting program `SAMPLE` can be run in the V3R2 runtime environment, producing the output `ccccc`.

Chapter 7. Setting Compiler Options

Use compiler options to specify different aspects of the compilation and binding of your program. This section describes the options and 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 hyphen (-) for the slash (/) preceding the option. For example, -Rn is equivalent to /Rn. Lower-and uppercase, hyphens, and slashes can all be used on one command line, as in:

```
iccas  /ls  -SI  -oI  /Fl  prog.cpp
```

You do not need the plus symbol (+) when specifying an option: the forms /Fb+ and /Fb are equivalent.

Many options have parameters. See “Using Parameters with Compiler Options” on page 73 for more information on these parameters.

You can specify compiler options in several ways:

On the Windows command line

Compiler options specified on the Windows command line override any options previously specified in the ICCAS environment variable (as described below and in “Windows Environment Variables for Compiling” on page 47). To compile a source file from an Windows command line, enter the following command at the Windows prompt:

```
iccas /option  filename
```

To compile the source file myprog.cpp with the browser option /Fb enter:

```
iccas /Fb myprog.cpp
```

In the ICCAS environment variable

Frequently used Windows command-line options can be stored in the ICCAS environment variable. This method is useful if you find yourself repeating Windows command line options every time you compile. You can also specify source filenames in ICCAS.

The ICCAS environment variable can be set:

- From the Windows command line
- In a batch (.BAT) file
- In the **autoexec.bat** file (in the Windows 95 operating system)
- In the System window (in the Windows NT operating system).

The ICCAS environment variable has the following characteristics:

- If it is set on the command line or by running a batch file, the options will only be in effect for the current session.

- If it is set in the **autoexec.bat** file or in the System window, the options will be in effect every time you use the **iccas** command, unless you override them by:
  - Using a `.BAT` file
  - Specifying options on the command line.

To specify that a source listing be generated for all compilations and that the macro `DEBUG` be defined as 1, use the following command at the Windows prompt (or in your **autoexec.bat** file or the System window, if you want these options every time you use the compiler):

```
SET ICCAS=/Ls+  /DDEBUG::1
```

(The double colon must be used because the "=" sign is not allowed in Windows environment variables.)

Now, when you type `iccas prog1.cpp` to compile `prog1.cpp`, the macro `DEBUG` will be defined as 1, and a source listing will be produced.

Options specified on the Windows command line override the options in the ICCAS environment variable. For example, the following compiler invocation takes precedence over the ICCAS setting in the last example:

```
iccas /Ls- /UDEBUG myprog.cpp
```

See “Windows Environment Variables for Compiling” on page 47 for more information about using ICCAS and other environment variables.

Combining ICCAS and Windows Command Line Options

When you specify compiler options both in the ICCAS environment variable and on the Windows command line, the compiler evaluates both sets of options. When the compiler is invoked:

1. The string associated with ICCAS is retrieved.
2. The Windows command line is retrieved.
3. The Windows command line is appended to the ICCAS string, combining the two into a single Windows command line.
4. This combined Windows command line is read from left to right, and the compiler-option-precedence rules are applied.
5. The files are compiled using the options as interpreted in the previous step.

In the WorkFrame Environment

If you have installed WorkFrame, you can set compiler options through options dialogs. You can use these dialogs when you create or modify a project.

Options you select while creating or changing a project are saved with that project. See Part 1, “ Developing OS/400 Applications with WorkFrame” on page 1 for more information on using WorkFrame.

Using Parameters with Compiler Options

All compiler options that take parameters obey the following rules:

- If a parameter is required, zero or more spaces may appear between the option and the parameter. For example, both `/DDEBUG` and `/D DEBUG` are valid.
- If a parameter is optional, no space is allowed 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 five types of parameters:

- Strings
- Switches
- Numbers
- Filenames
- File extensions.

Strings

If the option has a `string` parameter, the string must be enclosed by a pair of quotation marks if there are spaces in the string. 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 contains double quotation marks, precede them with the backslash (\) character. For example, if the string is `abc"def`, specify it on the Windows command line as `"abc\"def"`. This combination is the only valid escape sequence within string options. Do not end a string with a backslash, as in `"abc\"`.

Do not put a space between the option and the string.

Switches

Some options are used with plus (+) or minus (-) signs. If you do not use either sign, the compiler processes the option as if you had used the + sign. When you use an option that uses switches, you can combine them. For example, the following compiler invocations have the same result:

```
iccas  /Fb+ /Fi+ /Fl+ /Fo-  file.cpp
iccas  /Fbilo-  file.cpp
```

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. For example:

```
/AScp37
```

Filenames, Extensions, and Paths

When compiler options take filenames as parameters, the following rules apply:

- A filename that contains spaces must be enclosed in quotation marks. In such a case, do not put a space between the option and a filename or directory.
- The compiler uses defaults unless you specify output-file extensions. For example, if you specify `/Flcome`, the listing file will be called `come.lst`.

  Note: If you really want to call the listing file `come`, specify `/Flcome.`. The period at the end of the filename specifies a null extension.
- If you use an option without using an optional `name` parameter, the name of the following source file and the default extension are used.
- If you want to use a file that is in the current directory, specify only the filename. For example, if the current directory is `E:\`, and the source file is `E:\myprog.cpp`, you invoke the compiler with `iccas myprog.cpp`, using default values.
- If the file is not in the current directory, specify the path and filename; for example:

  ```
  iccas /Fb F:\mydir\myprog.cpp
  ```

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.cpp` is compiled with the option `/Fb-` because this option follows `/Fb+`:

```
iccas /Fb+ /Fb- module1.cpp
```

In the following invocation, the file `module1.cpp` is compiled with the `/Fb+` option, while `module2.cpp` is compiled with `/Fb-`:

```
iccas /Fb+ module1.cpp /Fb- module2.cpp
```

When you are compiling programs with multiple source files, an option is in effect for all the source files that follow it. For example, if you enter the following Windows command:

```
iccas /Oi+ prog1.cpp /Fb sub1.cpp /Xc /Oi- sub2.cpp
```

- The file `prog1.cpp` is compiled with the option `/Oi+`.
- The file `sub1.cpp` is compiled with the options `/Oi+` and `/Fb+`.
- The file `sub2.cpp` is compiled with the options `/Oi-`, `/Fb+` and `/Xc`.

The name of the program object is the same as the name of the first source file without its extension `.cpp`. In this example the program object will be called `PROG1`.

Options Having a Special Scope

Most options only apply to source files that follow the option, and can be overridden with a conflicting option that appears later in the Windows command line.The following options do not behave as described above:

`/D` Defines a preprocessor macro. `/D` is different from other options in that the **first** definition of a macro 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:

```
iccas /Ia: /Ib:\cde /Ic:\fgh prog.cpp
```

causes the search path `a:;b:\cde;c:\fgh` to be built.

`/B` Passes options to the binder. 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 Windows command line, and in environment variables, are accumulated **before** the modules are bound. The options affect all modules to be bound.

For example, the Windows command:

```
iccas /B"crtpgm pgm(mylib/cost)" /Ti+ /B"bndsrvpgm(mylib/serv1)"
    cost.cpp
```

is equivalent to specifying:

```
iccas /B"crtpgm pgm(mylib/cost) bndsrvpgm(mylib/serv1)"
   /Ti+ cost.cpp
```

`/Q` Controls the display of the compiler logo. This option is global and applies to all source files on the Windows command line. It can follow the last file on the Windows command line. If it is specified more than once, the last occurrence of the option is the one in effect.

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.

Conflicting Options

Some options are incompatible with other options. If options specified on the Windows command line are in conflict, the following rules apply:

- The option `/Fb` to produce a browser file takes precedence over the precompiled header file option (`/Si`).
- The syntax check option `/Fc` takes precedence over the output file generation `/Fb`, `/Fo`, and `/Ft`, intermediate code generation `/Fw`, and preprocessor `/P`, `/Pc`, `/Pd`, and `/Pe` options.

- The option /Fo- to not create a module object (*MODULE) takes precedence over the option (/Ti to include debug information in the object.
- The options /Li and /Lj to expand #include files in the listing take precedence over the precompiled header file options /Fi and /Si.
- The option /Lj+ to expand user and system #include files takes precedence over the option /Li to expand user #include files only.
- The preprocessor options /P, /Pc, /Pd, and /Pe take precedence over the output file generation /Fb, /Fl, /Fo, and /Ft, intermediate code generation /Fw, precompiled header file /Fi and /Si, and all listing file /L options.
- If both the option /ASv3r6+ to compile for a V3R6 RISC runtime environment and the option /ASv3r2+ to compile for a V3R2 CISC runtime environment are specified, the second option is ignored.

Compiler Option Classification

The compiler options are divided into groups by function:

- "Output File Management Options" on page 80

  /ASd /ASl /ASr /F
- "#include File Search Options" on page 84

  /I /X
- "Listing File Options" on page 85

  /L
- "Debugging and Diagnostic Information Options" on page 90

  /N /Ti /W
- "Source Code Options" on page 93

  /qbitfields /S /Si /Sn
- "Preprocessor Options" on page 95

  /D /P /U
- "Code Generation Options" on page 98

  /ASi /ASp /O
- "Other Options" on page 100

  /ASa /AScp /ASn /ASt /ASv3r2 /ASv3r6 /B /C /J
  /Q /qmakedep /qrtti /qnortti /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

Table 3 (Page 1 of 4). Compiler Options Summary

Option	Description	Default	Page
/?	Display list of compiler options with descriptions	None	"/?" on page 100
/Asa[l|c|u|a|e] /Asa"list-name"	Specify module authority granted to users who do not have specific authority	/ASal	"/ASa" on page 100
/AScp[n]	Specify the code page number for the target AS/400 system	Host CCSID	"/AScp" on page 101
/ASd[+|-] /ASddir	Save header files generated from the **#pragma mapinc** directive	/ASd+	"/ASd" on page 80
/ASi[+|-]	Use integrated file system APIs instead of standard, non-integrated file system function	/ASi+	"/ASi" on page 98
/ASlname	Specify destination library name	none	"/ASl" on page 81
/ASnname	Specify the name of the AS/400 system connection to use	None	"/ASn" on page 102
/ASp[-|2a|2n|3a|3n]	Generate performance-data-collection code	/ASp-	"/ASp" on page 98
/ASr[+|-]	Replace a module object	/ASr+	"/ASr" on page 81
/ASt"text"	Specify text description for module	/Blanks	"/ASt" on page 102
/ASv3r2[+|-]	Compile source files to create program or service program objects that run in a V3R2 IMPI environment	/ASv3r2-	"/ASv3r2" on page 102
/ASv3r6[+|-]	Compile source files to create program or service program objects that run in a V3R6 RISC environment	/ASv3r6-	"/ASv3r6" on page 102
/B"options"	Pass options to the binder, in addition to default options	/B"crtpgm pgm(xxx/yyy)"	"/B" on page 103
/C[+|-]	Perform compile without binding, instead of compiling and binding	/C-	"/C" on page 103
/Dname[=n] /Dname[::n]	Define preprocessor macros	None	"/D" on page 96
/Fb[+|-]	Produce a browser file	/Fb-	"/Fb" on page 81
/Fc[+|-]	Perform syntax check only, instead of a full compile	/Fc-	"/Fc" on page 81
/Fi[+|-]	Create precompiled header file	/Fi-	"/Fi" on page 82

Table 3 (Page 2 of 4). Compiler Options Summary			
Option	**Description**	**Default**	**Page**
/Fl[+|-][dir][name]	Produce listing file	/Fl-	"/Fl" on page 82
/Fo[+|-][name]	Control and name module object	/Fo+	"/Fo" on page 83
/Ft[+|-]/Ftdir	Control and direct files for template resolution	/Ft+	"/Ft" on page 83
/Fw[+|-][dir][name]	Create intermediate code files only, instead of a full compilation	/Fw-	"/Fw" on page 84
/Ipath[;path]	Specify #include search paths, in addition to directory of source file and paths in the INCLUDE_ASV3R7, INCLUDE_ASV3R6 or INCLUDE_ASV3R2 environment variables	No additional paths	"/I" on page 85
/J[+|-]	Set unspecified `char` variables to `unsigned char`	/J+	"/J" on page 103
/L[+|-]	Produce a minimal listing file	/L-	"/L" on page 86
/La[+|-]	Include a minimal layout in the listing file	/La-	"/La" on page 86
/Lb[+|-]	Include a layout in the listing file	/Lb-	"/Lb" on page 87
/Le[+|-]	Expand macros in the listing file	/Le-	"/Le" on page 87
/Lf[+|-]	Set all listing options on	/Lf-	"/Lf" on page 87
/Li[+|-]	Expand user #include files in the listing file	/Li-	"/Li" on page 88
/Lj[+|-]	Expand user and system #include files in the listing file	/Lj-	"/Lj" on page 88
/Lpnum	Set page length of the listing file	/Lp66	"/Lp" on page 88
/Ls[+|-]	Include the source code in the listing file	/Ls-	"/Ls" on page 88
/Lt"string."	Set title string for the listing file	Name of first source file	"/Lt" on page 89
/Lu"string"	Set subtitle string in the listing file	/Lu""	"/Lu" on page 89
/Lx[+|-]	Generate a minimal cross-reference table in the listing file	/Lx-	"/Lx" on page 89
/Ly[+|-]	Generate a cross-reference table in the listing file	/Ly-	"/Ly" on page 89
/Nn	End compilation when error count reaches *n*	No limit.	"/N" on page 90

Table 3 (Page 3 of 4). Compiler Options Summary

Option	Description	Default	Page
/O[+|-|10|20|30|40]	Optimize code	/O-	"/O" on page 99
/Oi[+|-] /Oivalue	Inline specified user functions	/Oi-	"/Oi" on page 99
/P[+|-]	Run the preprocessor only, instead of a full compile	/P-	"/P" on page 96
/Pc[+|-]	Preserve source code comments in preprocessor output	/P-	"/Pc" on page 96
/Pd[+|-]	Redirect preprocessor output	/P-	"/Pd" on page 97
/Pe[+|-]	Suppress `#line` directives in preprocessor output	/P-	"/Pe" on page 97
/Q[+|-]	Display compiler logo when invoking the compiler	/Q-	"/Q" on page 104
/qbitfields=<signed |unsigned>	Specify default sign of bitfields	/qbitfields=unsigned	"/qbitfields" on page 93
/qmakedep	Create dependency files to include in makefile	None	"/qmakedep" on page 104
/qro/qnoro	Use read-only, or not read-only (read/write) storage for string literals	/qro	"/qro" on page 104
/qrtti=rtoption /qnortti	Generate information for the typeid operator and the dynamic cast operator	/qnortti	"/qrtti" on page 104
/S[a|c|e]	Set the language level	/Se	"/S" on page 93
/Si[+|-][dir][name]	Use precompiled header files, if they exist and are current	/Si-	"/Si" on page 94
/Sn[+|-]	Parse double-byte character set C++ source files	/Sn-	"/Sn" on page 94
/Sp[1|2|4|8|16]	Specify alignment or packing of data items within structures and unions	/Sp16	"/Sp" on page 94
/Su[+|-|1|2|4]	Control size of `enum` variables, instead of using the SAA rules	/Su-	"/Su" on page 95
/Ti[+|-|s|l|n]	Generate debugger information	/Ti-	"/Ti" on page 90
/U<name|*>	Undefine macros	Retain macros	"/U" on page 97
/V"string"	Include a version string in the module and program or service program objects	/V""	"/V" on page 105
/W[0|1|2|3]	Set severity level of messages the compiler produces and counts	/W3	"/W" on page 91

Table 3 (Page 4 of 4). Compiler Options Summary			
Option	**Description**	**Default**	**Page**
/Wgrp[+|-][grp]	Generate or suppress messages in the grp group	/Wall-	"/Wgrp" on page 91
/Xc[+|-]	Do not search paths specified using /I	/Xc-	"/Xc" on page 85
/Xi[+|-]	Do not search paths specified in the INCLUDE_ASV3R7, INCLUDE_ASV3R6 and INCLUDE_ASV3R2 environment variables	/Xi-	"/Xi" on page 85

Output File Management Options

Use the Output File Management options to control the files that the compiler produces.

Examples of Output File Management Options

- Perform syntax check only:

  ```
  iccas /Fc+ myprog.cpp
  ```

- Name the module object:

  ```
  iccas /FoSTOCK myprog.cpp
  ```

  This names the module object `STOCK` instead of the default, `MYPROG`.

- Name the listing file:

  ```
  iccas /Floutput.my /L fred.cpp
  ```

  This creates a listing output called `output.my` instead of `fred.lst`.

/ASd

Syntax:

`/ASd[+|-]`

`/ASd[dir]`

Default:

`/ASd+`

Use `/ASd` to save header files generated from the **#pragma mapinc** directive.

You can specify a directory of your choice through the `/ASd`*dir* compiler option. The header files will be saved relative to that directory.

Note: If the **#pragma mapinc** directive specifies a fully qualified name, the file will be stored there, and **not** relative to the `/ASd` directory.

By default, the compiler saves header files relative to the current directory, in the filename given on the **#pragma mapinc** directive.

See the *C++ Programming Guide* for a more detailed discussion of **#pragma mapinc**.

/ASl

Syntax:	Default:
/ASl*name*	none

Use /ASl*name* to designate a target library for the module and program or service program objects to be created. Specify this library as *name*.

Note: The target library name must be a valid AS/400 library name and cannot be longer than 10 characters. If the library name contains invalid characters or is longer than 10 characters, compilation of the source file is terminated with an error message indicating that the library name is invalid. See “Naming Input Files” on page 45 for a description of valid AS/400 names.

There is no default for ASl. If /ASlname is specified, this library is used. Otherwise, the compiler probes the host for the current library. If no current library has been specified, the compiler uses the library QGPL.

/ASr

Syntax	Default
/ASr[+|-]	/ASr+

Use /ASr to replace a module object.

By default, the compiler replaces the existing object.

/Fb

Syntax:	Default:
/Fb[+|-|*]	/Fb-

Use /Fb to produce a browser file. The file has the same name as the next source file with the extension .PDB. The browser information generated excludes information about system include files.

Use /Fb* to generate browser information that includes information about system include files.

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. This option is useful when you compile in disconnected mode.

By default, the compiler performs a full compilation.

/Fi

Syntax: /Fi[+|-]

Default: /Fi-

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 not current.

If you specify a filename or directory with the option, the precompiled headers are placed in a file with the name and in the directory you specify.

If you do not specify a name or directory, the file is named csetcpp.pch, and is 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 filenames or directories with the two options, the name or directory specified last is used with both options.

By default, the compiler does not create a precompiled header file.

/Fl

Syntax: /Fl[+|-]
/Fl[dir][name]

Default: /Fl-

Use /Fl to produce, name, and direct a listing file. The listing file will be *name*.lst, and will be placed in directory *dir*, for example:

```
iccas /Fl mydir/myname
```

The compiler produces a separate listing file for each source file that follows the option on the Windows command line. The name you provide applies only to the first listing file.

If you do not specify a name or directory, the listing takes the same filename 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.

/Fo

Syntax:	Default:
/Fo[+|-]	/Fo+
/Fo[name]	

The object will be called *name* and will be placed in the current library on the AS/400. If you want to specify a different library, use the /ASl*name* option. If the specified library is invalid, the compiler will not create a module object.

The compiler produces a separate module object for each source file that follows the options on the Windows command line. The name you provide applies only to the first module object.

By default, the compiler produces a module object with the same name as the source file. This module object is created in library QGPL, if there is no current library.

Specify /Fo- if you do not want the compiler to create a module object.

Note: The module name must be a valid AS/400 object name. See “Naming Input Files” on page 45 for a description of valid AS/400 names. If the module name is not specified, the source filename of the first file to be compiled is used, which must be a valid AS/400 name. An invalid module name results in an error message and the iccas invocation is terminated.

/Ft

Syntax:	Default:
/Ft[+|-]	/Ft+
/Ft*dir*	

Use /Ft to control generation of files for template resolution.

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:

`/Fw[+|-]`

`/Fw[dir][name]`

Default:

`/Fw-`

Use `/Fw` to produce, name, and direct intermediate code files, without completing compilation. The intermediate code file will be `name.qwo`, and will be placed in directory `dir`.

The compiler produces a separate intermediate code file for each source file that follows the option on the Windows command line. The name you provide applies only to the first intermediate code file.

If you do not specify a name, the intermediate code file takes the same filename as the source file, with the extension .QWO.

If the directory you specify is not valid, the compiler does not save the intermediate code file; it generates a warning message and the option does not take effect.

By default, the compiler performs regular compilation, without saving the intermediate code file.

#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_ASV3R7, the INCLUDE_ASV3R6, the INCLUDE_ASV3R2, and the ICCAS environment variables, combined with the compiler options.

Specifying Search Paths

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 several directories, 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 ICCAS environment variable and on the Windows command line, all the paths are searched. The paths specified in ICCAS are searched before those specified on the Windows 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. See “Controlling #include Search Paths” on page 49 for further information on #include files and search paths.

/I

Syntax:	Default:
/Ipath[;path]	Directory of source file paths in INCLUDE_ASV3Rn

Use /I to specify #include search path(s). The compiler will search path[;path]. Note that the directory of the source file is always searched first for user include files.

By default, the compiler searches the directory of the source file (for user files only), and then the search paths given in the INCLUDE_ASV3R7, INCLUDE_ASV3R6, or the INCLUDE_ASV3R2 environment variables.

/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_ASV3R7, INCLUDE_ASV3R6 or the INCLUDE_ASV3R2 environment variables.

By default, the compiler searches paths specified in the INCLUDE_ASV3R7, INCLUDE_ASV3R6 or the INCLUDE_ASV3R2 environment variables.

Listing File Options

The options listed below 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 produce a listing. Use them with one of the other listing file options, or the /Fl option, to produce a listing:

```
La- /Lb- /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 the 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.

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 on which they appear.
/Ly+	Includes a cross-reference table that contains a list of all identifiers, together with the numbers of the lines on which they appear.

/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.

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 all 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 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 does not include a layout.

/Le

Syntax:	**Default:**
/Le[+|-]	/Le-

Use /Le to expand all macros in the listing file.

By default, the listing does not show macros expanded.

/Lf

Syntax:	**Default:**
/Lf[+|-]	/Lf-

Use /Lf to set all listing options on.

By default, all listing options are off.

/Li

Syntax:	Default:
/Li[+|-]	/Li-

Use /Li to expand user #include files in the listing file.

By default, the listing 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 other file systems and have very long filenames, there may not be enough room for the filenames on the lines showing the included code. Counters are used in the INCLUDE column of the listing output, and the filename that corresponds to each number is given at the bottom of the source listing.

By default, the listing does not show user and system #include files expanded.

/Lp

Syntax:	Default:
/Lp*num*	/Lp66

Use /Lp to set the page length of the listing. Each page will be *num* lines long. *num* must be between 15 and 65535 inclusive.

By default, the listing 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 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.

/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).

/Lx

Syntax:	Default:
/Lx[+|-]	/Lx-

Use /Lx to generate a cross-reference table of referenced variable, structure, and function names. The table shows line numbers where names are declared.

By default, the listing does not include the cross-reference table.

/Ly

Syntax:	Default:
/Ly[+|-]	/Ly-

Use /Ly to generate a cross-reference table of all variable, structure, and function names, plus all local variables specified by the user. The table shows line numbers where names are declared.

By default, the listing does not include the cross-reference table.

Debugging and Diagnostic Information Options

The options listed here are useful for debugging your programs.

Use /*Wgrp* to control what types of diagnostic messages are produced.

Note: The information generated by the debugger and /*Wgrp* options is provided to 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 error messages before compilation is terminated. Compilation ends when the error count reaches *n*.

By default, the compiler does not set a limit on the number of errors.

/Ti

Syntax:	**Default:**
/Ti[+|-|l|n|s]	/Ti-

Use /Ti to enable all options to debug the program and produce a listing view. This option is the equivalent of the AS/400 *DBGVIEW(*ALL)* parameter.

Use /Til to generate a listing view to debug the program. This option is the equivalent of the AS/400 *DBGVIEW(*LIST)* parameter.

Use /Tin to generate debug data without creating a listing or a source view. This option allows the program to be debugged using symbolic identifiers. It is the equivalent of the AS/400 *DBGVIEW(*STMT)* parameter.

Use /Tis to generate a source view for debugging the program. This option is the equivalent of the AS/400 *DBGVIEW(*SOURCE)* parameter.

By default, the compiler does not generate debug information.

Recommendations:

If you use the /Ti option to generate debugger information, it is recommended that you turn optimization off (/O-).

Because of the effects of optimization, debugging information generated with optimization is limited to setting breakpoints at function entry and function exit. Accurate symbol and type information is not always available.

/W

Syntax:	Default:
`/W[0|1|2|3]`	`/W3`

Use `/W` to set the severity level of message the compiler produces and that causes the error count to increment.

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:
`/Wgrp[+|-][grp]`	`/Wall-`

Use */Wgrp* to generate messages in the *grp* group. You can specify more than one group.

By default, the compiler suppresses all informational messages.

The /Wgrp 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.

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 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 /Wgrp options.

The following table lists the message groups and the message numbers that each controls.

Table 4. /Wgrp Options

grp	Controls Messages About	Messages
`all`	All diagnostics	All message numbers listed in this table
`cls`	Use of classes	CTT3110, CTT3253, CTT3266
`cmp`	Possible redundancies in unsigned comparisons	CTT3138, CTT3139, CTT3140
`cnd`	Possible redundancies or problems in conditional expressions	CTT3107, CTT3130
`cns`	Operations involving constants	CTT3131, CTT3219
`cnv`	Conversions	CTT3313
`cpy`	Problems generating copy constructors	CTT3199, CTT3200
`eff`	Statements without effect	CTT3165, CTT3215
`enu`	Consistency of `enum` variables	CTT3137
`ext`	Unused external definitions	CTT3127
`gen`	General diagnostics	CTT3101
`gnr`	Generation of temporary variables	CTT3151
`lan`	Effects of the language level	CTT3116
`par`	Unused parameters	CTT3126
`por`	Nonportable language constructs	CTT3132, CTT3133, CTT3135, CTT3136
`ppc`	Possible problems with using the preprocessor	CTT3645
`rea`	Code that cannot be reached	CTT3119
`trd`	Possible truncation or loss of data or precision	CTT3108, CTT3135, CTT3136
`und`	Casting of pointers to or from an undefined class	CTT3098
`use`	Unused `auto` and `static` variables	CTT3002, CTT3099, CTT3100
`vft`	Generation of virtual function tables	CTT3280, CTT3281, CTT3282

Examples of /Wgrp Options

- Produce all diagnostic messages:

  ```
  iccas /Wall blue.cpp
  iccas /Wall+ blue.cpp
  ```

- Produce diagnostic messages about:
  - Unreferenced parameters
  - Missing function prototypes
  - Uninitialized variables

  by turning on the appropriate suboptions:

```
iccas /Wpar+por+uni blue.cpp
iccas /Wparporuni blue.cpp
```

- Produce all diagnostic messages except:
  - Warnings about assignments that can cause a loss of precision
  - External variable warnings

  by turning on all options, and then turning off the ones you do not want:

  ```
  iccas /Wall+trd-ext- blue.cpp
  ```
- Produce only basic diagnostics, with all other suboptions turned off:

  ```
  iccas /Wgen+ blue.cpp
  ```
- Produce only basic diagnostics and messages about severe errors, errors, or warnings (/W2):

  ```
  iccas /Wgen2 blue.cpp
  ```

Source Code Options

The options described in this section control how the compiler interprets your source files. This control is especially useful if you want to migrate code or ensure consistency with a particular language standard.

/qbitfields

Syntax:	Default:
/qbitfields=<signed|unsigned>	/qbitfields=<unsigned>i

Use /qbitfields to specify the sign of bitfields.

By default, bitfields are unsigned.

/S

Syntax:	Default:
/S[a|c|e]	/Se

Use /S to set the language level.

You can set the following language levels:

/Sa	Conform to ANSI standards.
/Sc	Allow constructs compatible with older levels of the C++ language.
/Se	Allow all VisualAge for C++ for AS/400 language constructs

By default, the compiler allows all VisualAge for C++ for AS/400 language extensions.

See “Setting the Source Code Language Level” on page 52 for details.

/Si

Syntax:	Default:
/Si[+|-]	/Si-
/Si[dir][name]	

Use /Si to use precompiled header files, if they exist and are current.

If you specify a *name* or *directory* with the option, the compiler looks for a precompiled header file with the name and in the directory you specify.

If you do not specify a name or a directory, the compiler looks for a file named csetcpp.pch, in the current working directory.

Use the /Fi option to create or update precompiled header files. Use /Si and /Fi in combination to ensure that 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 filenames or directories with the same options, the name or directory specified last is used with both options.

By default, the compiler does not use precompiled header files.

/Sn

Syntax:	Default:
/Sn[+|-]	/Sn-

Use /Sn to parse double-byte character set C++ source files.

By default, the compiler parses C++ source files as single-byte character set files.

/Sp

Syntax:	Default:
/Sp[1|2|4|8|16]	/Sp16

Use /Sp to specify alignment or packing of data items within structures and unions.

By default, structures and unions are aligned along 16-byte boundaries (normal alignment).

You can align structures and unions along 1-byte, 2-byte, 4-byte, 8-byte, or 16-byte boundaries. /Sp is equivalent to /Sp1.

Note: Pointer members of a structure always align at the 16-byte boundary.

/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 compiler makes 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.

Preprocessor Options

The options listed control the use of the preprocessor:

/D /P /Pc /Pd /Pe /U

Note that the /Pc, /Pd, and /Pe options are actually suboptions of /P. Specifying /Pc- is the same as specifying /P+c-, and only the preprocessor is run.

Using Preprocessor Directives

Preprocessor directives, such as #include, allow you to include C++ code from another source file into yours, to define macros, and to expand macros.

See the *C++ 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 to debug a program. All the preprocessor directives have been executed, but no code has been compiled. For example, all macros are expanded, and the code of 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.

The /P, /Pc, /Pd, and /Pe options can be used in combination with each other. For example, to preserve comments, suppress #line directives, and redirect the pre-processor output to stdout, specify /Pcde.

/D

Syntax:	Default:
/Dname[::n]	Do not define macros on the Windows command line.
/Dname[=n]	

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 Windows command line override macros defined in the source code.

By default, no macros are defined on the Windows 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 preprocessor output is not generated.

/Pc

Syntax:	Default:
/Pc[+|-]	/P-

Use /Pc to run the preprocessor only, and control the inclusion of comments in the preprocessor output.

Specify /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.

By default, both the compiler and preprocessor run, and preprocessor output is not generated.

/Pd

Syntax:	Default:
/Pd[+|-]	/P-

Use /Pd to run the preprocessor only, and control redirection of the preprocessor output.

Specify /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

By default, both the compiler and preprocessor run, and preprocessor output is not generated.

/Pe

Syntax:	Default:
/Pe[+|-]	/P-

Use /Pe to run the preprocessor only, and control the creation of #line directives in the preprocessor output.

Specify /Pe+ to run the preprocessor only, and suppress creation 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.

By default, both the compiler and preprocessor run, and preprocessor output is not 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 the macro __FUNCTION__, nor does it undefine macros defined in source code.

By default, the preprocessor retains all macros.

Code Generation Options

The following options specify how the code produced by the compiler is optimized and if functions are inlined:

/ASi /ASp /O /Oi

Notes:

1. The /Oi+ option is more effective when /O+ is also specified.
2. Using optimization (/O+) limits the use of the debugger. The /Ti option is not recommended for use with optimization.

/ASi

Syntax:	**Default:**
/ASi[+|-]	/ASi+

Use /ASi to generate code that uses integrated file system APIs.

You can alternatively use /D__IFS_IO__.

By default, the compiler uses integrated file system functions defined in stdio.h. (This option has no effect on iostream.h.)

Note: When you compile with /ASi, the hexadecimal value of \n is 25. When you compile with /ASi-, the hexadecimal value of \n is 15.

See the *C++ Programming Guide* for details on the Integrated File System.

/ASp

Syntax:	**Default:**
/ASp[-|2a|2n|3a|3n]	/ASp-

Use /ASp to control the generation of performance-data collection code

Use /ASp- to generate performance-data-collection code at entry to and exit from the program entry procedure only.

Use /ASp2a to generate performance-data-collection code at entry to and exit from all the procedures in the program.

Use /ASp2n to generate performance-data-collection code at entry to and exit from non-leaf procedures only.

Use /ASp3a to generate performance-data-collection code at procedure entry/exit and before and after calls to external procedures, for all procedures in the program.

Use /ASp3n to generate performance-data-collection code at procedure entry/exit and before and after calls to external procedures, for non-leaf procedures only.

By default, the compiler generates performance-data-collection code at entry to and exit from the program entry procedure only.

/O

Syntax:	Default:
/O[+|-|10|20|30|40]	/O-

Use /O to control the optimization of your code.

Note: Do not optimize code if you want to use debugging or diagnostic options.

Use /O20 to set basic optimization on.

Use O+ or /O30 to set full optimization on.

Use /O40 to set super optimization (level 40) on.

By default, the code is not optimized. /O- and /O10 are equivalent.

/Oi

Syntax:	Default:
/Oi[+|-]	/Oi-
/Oi*value*	

Use /Oi to control inlining of user code.

By default, no user code is inlined. When /O[+|1|2] is specified, /Oi+ becomes the default.

You can specify the following types of inlining:

/Oi+	Inline all user functions qualified with the `inline` keyword.
/Oi-	Do not inline any user code.
/Oi*value*	Inline all user functions qualified with the `inline` keyword or that are smaller than *value* in abstract code units.

See “Inlining User Code” on page 62 for more information.

Other Options

Use the following options to control binder parameters, logo display, default `char` type, and other VisualAge for C++ for AS/400 options:

`/? /ASa /AScp /ASn /ASt /ASv3r2 /ASv3r6 /B /C /J /Q /qmakedep /qrtti /qnortti /V`

Examples of Other Options

- Passing a parameter to the binder:

  ```
  iccas /B"crtsrvpgm a" file1.cpp file2.cpp
  ```

  The `/B"crtsrvpgm a"` option tells the compiler to create a service program A on the host system, using *MODULE objects FILE1 and FILE2 which are created from `file1.cpp` and `file2.cpp`.

- Imbedding a version string or copyright:

  ```
  iccas /V"Version 1.0" myfile.cpp
  ```

  This imbeds the version notice in the module and program objects `MYFILE`.

/?

Syntax:	**Default:**
`/?`	None

Use `/?` to display a list of compiler options with descriptions.

/ASa

Syntax:	**Default:**
`/ASa[l|c|u|a|e|"list-name"]`	`/ASal`

Use `/ASa` to specify the module and program authority granted to users who do not have specific authority.

Use `/ASal` to take the authority for the module object to be created from the *CRTAUT* keyword of the target library (the library that contains the created module object, on AS/400). This option is the equivalent of the **LIBCRTAUT* parameter on the AS/400 create module commands

Use `/ASac` to provide all data authority and the authority to perform all operations on the module object, except those limited to the owner or controlled by object authority and object management authority. The module object can be changed and basic functions

can be performed on it. This option is the equivalent of the **CHANGE* parameter on the AS/400 create module commands.

Use /ASau to provide object operational authority; read authority; and authority to perform basic operations on the module object such as binding the module object. Users without specific authority are prevented from changing the module object. This option is the equivalent of the **USE* parameter on the AS/400 create module commands.

Use /ASaa to provide authority to perform all operations on the module object, except those limited to the owner or controlled by authorization list management authority. Any user can control the module object's existence, specify the security for it, change it, and perform basic functions on it, but cannot transfer ownership of the object. This option is the equivalent of the **ALL* parameter on the AS/400 create module commands.

Use /ASae to ensure that users without special authority cannot access the module object. This option is the equivalent of the **EXCLUDE* parameter on the AS/400 create module commands.

Use /ASa"*list-name*" to specify the name of an authorization list of users and authorities to which the module object is added. The module object is secured by this authorization list, and the public authority for the module object is set to **AUTL*. The authorization list must exist on the system when this option is specified.

By default, the compiler takes the authority for the module object to be created from the *CRTAUT* keyword of the target library.

/AScp

Syntax:	**Default:**
/AScp[n]	/AScp37

Use /AScp[n] to specify the coded character set ID (CCSID) for the destination AS/400 system. n represents the code page number.

By default, the compiler uses the CCSID of the job running on the AS/400 identified by the ICCASNAME environment variable, when running in connected mode. When running in disconnected mode, the compiler uses CCSID 37 (US English).

When a CCSID is specified in several places, the precedence rules are as follows:

1. CCSID specified on the /AScp compiler option
2. CCSID specified on the ICCASCP environment variable
3. AS/400 CCSID (none if you compile in disconnected mode)

/ASn

Syntax:	Default:
/ASn*name*	Value of ICCASNAME

Use /ASn*name* to specify the name of the connection to the host system of your choice.

If you have specified the name of the connection in the ICCASNAME environment variable, the ASn*name* option is not necessary. Otherwise it will override the name specified in ICCASNAME.

By default, the compiler uses the name specified in the ICCASNAME variable. If ICCASNAME is not set, the compiler issues an error message.

/ASt

Syntax:	Default:
/ASt*text*	Blanks

Use /AS*ttext* to specify a text description for the module object. If necessary, the text is truncated to 50 characters. The compiler issues a warning if truncation occurs.

By default, the compiler inserts blank spaces in the text string.

/ASv3r2

Syntax:	Default:
/ASv3r2[+|-]	/ASv3r2-

Use /ASv3r2 to compile source files into program or service program objects that will run in a V3R2 CISC environment.

By default, the compiler targets the V3R7 RISC run-time environment.

/ASv3r6

Syntax:	Default:
/ASv3r6[+|-]	/ASv3r6-

Use /ASv3r6 to compile source files into program or service program objects that will run in a V3R6 RISC environment.

By default, the compiler targets the V3R7 RISC run-time environment.

/B

Syntax:	Default:
/B"*binding command*"	/B"crtpgm pgm(xxx/yyy) "

Note: Make sure you use a forward slash (/) to separate the library and module name in the module parameter of the /B option. If you use a backslash instead, binding fails.

Use /B to pass the *binding command* string to the binder.

If you specify the /B option, you must specify either

"crtpgm pgm (library-name/program-name)"

or

"crtsrvpgm srvpgm (library-name/srvprogram-name)"

The string may also contain additional binding options, such as "actgrp(abc)"

By default, /B"crtpgm PGM (xxx/xxx)" is passed to the binder with the default options of the CRTPGM host command. xxx and yyy are filled in with the appropriate library and program name by the **iccas** compiler invocation command.

See Chapter 8, “Creating OS/400 Programs” on page 109 and Chapter 9, “Creating a Service Program” on page 121 for information on binding module objects into programs and service programs from the workstation.

/C

Syntax:	Default:
/C[+|-]	/C-

Use /C to perform a compile only, without binding. This option results in the creation of a module object (*MODULE).

By default, code is both compiled and bound into a program object (*PGM).

If you want to create a service program (*SRVPGM), you must also specify the /B"crtsrvpgm srvpgm(library-name/service-program-name)" option.

/J

Syntax:	Default:
/J[+|-]	/J+

Use /J- to set unspecified char variables to signed char.

By default unspecified `char` variables are set to `unsigned char`.

/Q

Syntax:	Default:
`/Q[+|-]`	`/Q-`

Use `/Q+` to suppress the compiler logo that appears when you invoke the compiler.

By default, the compiler logo appears on `stderr`.

/qmakedep

Syntax:	Default:
`/qmakedep`	`None`

Use `/qmakedep` to generate make file information and output it to a dependencies (.u) file. This information can then be used by the NMAKE utility.

Note: Only workstation file dependencies are handled by this option.

An example of the output format is as follows:

```
myprog.module: myprog.cpp
myprog.module: C:\cttasw\include\iostream.h
myprog.module: C:\cttasw\include\stdlib.h
```

By default, the information is not generated.

/qro

Syntax:	Default:
`/qro`	`/qro`
`/qnoro`	

Use `/qro` to specify that string literals be placed in read-only storage. If `/qnoro` is specified, string literals are placed in read/write storage.

By default, string literals are placed in read-only storage.

/qrtti

Syntax:	Default:
`/qrtti=rtoption /qnortti`	`/qnortti`

Use /qrtti to generate information for the typeid operator and the dynamic_cast operator.

The suboptions (*rtoption*) are:

ALL The compiler generates the information needed for the RTTI typeid and dynamic_cast operators.

TYPEinfo The compiler generates the information needed for the RTTI typeid operator, but the information for the dynamic_cast operator is not generated.

DYNAmiccast The compiler generates the information needed for the RTTI dynamic_cast operator, but the information for the typeid operator is not generated.

By default, the compiler does not generate information needed for the RTTI typeid and dynamic_cast operators.

Note: The /qrtti option is only available for V3R7. Do not use this option when you compile your application to run in the V3R6 RISC or the V3R2 CISC runtime environment.

/V

Syntax:	**Default:**
/V"*string*"	/V""

Use /V to include a version string in a module, program, or service program. The version string is set to *string*. The length of the string can be up to 256 characters.

By default, no version string is set.

Part 3. Binding and Running Your Program

The sections in this part explain how to bind modules into programs or service programs, and how to run ILE C++ applications.

Chapter 8. Creating OS/400 Programs

This section describes how to use the binder to bind one or more module objects to the VisualAge for C++ for AS/400 run-time libraries to create a program object (*PGM).

See Chapter 9, “Creating a Service Program” on page 121 for details on binding module objects into service programs (*SRVPGM).

Binding Modules into Programs

In the AS/400 Integrated Language Environment, creating a program consists of compiling each source-code file into a module object (*MODULE), and then binding either one or multiple module objects into a program object (*PGM).

The creation of permanent modules allows you to modularize an application, and make changes to individual modules without having to recompile the whole application. The same module can be reused in different applications.

Binding is the process of combining one or multiple modules and optional service programs, and resolving external symbols between them. The system code that combines modules and resolves symbols is called the *binder*.

While the compilation of source code into module objects takes place on both the Windows workstation and AS/400, binding takes place on AS/400 only, although you may invoke the binder from the workstation.

Identifying Program and User Entry Procedures

As part of the binding process, a procedure must be identified as the startup procedure, or *program entry procedure* (PEP). When a program is called, the PEP receives the command line parameters and is given initial control for the program. The procedures that get control from the PEP are called *user entry procedures* (UEP).

An ILE C++ module contains a program entry procedure only if it contains a main() function. Therefore, one of the modules being bound into the program must contain a main() function.

Understanding the Internal Structure of a Program Object

Figure 4 on page 110 shows the internal structure of a typical program object, MYPROG, created by binding two modules, TRNSRPT and INCALC. In this example, TRNSRPT is the entry module containing the PEP, in addition to a UEP. Module INCALC contains a UEP only.

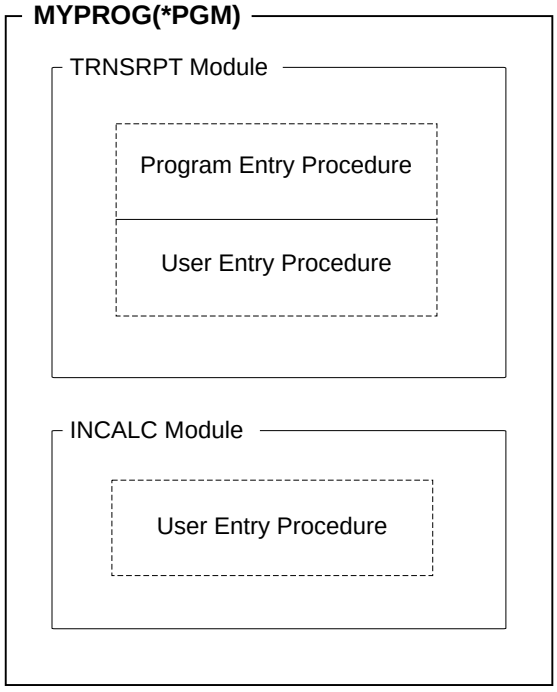


Figure 4. Structure of Program MYPROG

Using Static Procedure Calls

Within a bound object, procedures can be called using *static procedure calls*. These bound calls are faster than external calls. Therefore, an application consisting of a single bound program with many bound calls should perform faster than a similar application consisting of separate programs with many external inter-program calls.

Binding Modules into Service Programs

In addition to binding several modules together, you can also bind modules to *service programs*.

Common routines may be created as service programs, and if the routine changes, the change can be incorporated by binding the service program again.

The programs that use these common routines do not have to be recreated. See Chapter 9, “Creating a Service Program” on page 121 for information on creating service programs.

Working with Binding Directories

A *binding directory* contains the names of the modules and service programs that you may need when creating an ILE program or service program.

Binding directories are optional. They are objects identified to the system by the **BNDDIR* parameter on the CRTPGM command.

Modules or service programs listed in a binding directory are used only if they provide an export that can satisfy any currently unresolved import requests. Entries in the binding directory may refer to objects that do not yet exist at the time the binding directory is created, but will exist later.

See *ILE Concepts* for more information on the binding process, the binder, and binding directories, as well as for a list of related CL commands.

Invoking CL Commands through the CTTHCMD Command

You can issue CL commands directly from the workstation with the CTTHCMD command, provided you have established a host connection. Use CTTHCMD to issue the following CL commands for program objects :

- Create Program (CRTPGM)
- Change Program (CHGPGM)
- Delete Program (DLTPGM)
- Update Program (UPDPGM)

Make sure you are familiar with the syntax and the parameters of these commands when you issue them with CTTHCMD, because AS/400 help and prompts are not accessible from the Windows workstation.

Other CL commands such as Display Program (DSPPGM), Display Program References (DSPPGMREF), or Work with Program (WRKPGM) should not be issued with the CTTHCMD, because they produce spool files. The interactive AS/400 displays associated with these commands are not visible from the Windows workstation. See “Issuing AS/400 CL Commands from Windows through CTTHCMD” on page 31 for details on the syntax and parameters of the CTTHCMD.

The *CL Reference* contains further information on all CL commands and their parameters.

Invoking the Binder to Create a Program

The binder is invoked through the Create Program (CRTPGM) command. This command creates a program object from one or more module objects and, if required, one or more service programs.

You can bind modules created by the C++ compiler with modules created by any of the other ILE Create Module commands, including CRTRPGMOD, CRTCMOD, CRTCBLMOD, or CRTCLMOD.

Note: The modules or service programs to be bound must already have been created.

Preparing for Creating a Program

Before you create a program object using the CRTPGM command, you should:

1. Establish a program name.
2. Identify the module(s) and, if required, the service programs you want to bind into a program object.
3. Make sure that the program has a program entry procedure that gets control when a dynamic program call is made. (That is, one module must contain the `main()` function of the program.)

   You indicate which module contains the program entry procedure through the *ENTMOD* parameter. The default is *ENTMOD(*FIRST)*, which means that the module containing the first program entry procedure found in the list for the *MODULE* parameter is the entry module.

   If you are binding more than one ILE C++ module together, you should specify *ENTMOD(*FIRST)* or else specify the module name with the program entry procedure. You can use *ENTMOD(*ONLY)* when you are binding only one module into a program object, or if you are binding several modules but only one contains a program entry procedure. For example, if you bind a C++ module with a `main()` function to a C module without a `main()` function, you can specify *ENTMOD(*ONLY)*.
4. Identify the activation group that the program is to use.

   Specify *ACTGRP(*NEW)* if your program has no special requirements or if you are not sure which group to use.

   Note that *ACTGRP(*NEW)* is the default activation group for CRTPGM. This means that your program will run in its own activation group, and the activation group will terminate once the program terminates. This default ensures that your program has a refresh of the resources necessary to run, every time you call it.

   See “Specifying an Activation Group” on page 163 for more information on unnamed and named activation groups.

Specifying Parameters for the CRTPGM Command

Table 5 on page 113 lists CRTPGM command parameters and their default values. See the *CL Reference* , for a full description of the CRTPGM command and its parameters.

Table 5. Parameters for CRTPGM Command and their Default Values

Parameter Group	Parameter(Default Value)
Identification	PGM(*library name/program name*) MODULE(*PGM)
Program access	ENTMOD(*FIRST)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Run time	ACTGRP(*NEW)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER)

Invoking the CRTPGM Command

Although the binding process always takes place on AS/400, you may invoke the binder either from your workstation or from the AS/400 host system. You may choose one of the following four invocation methods:

1. From the WorkFrame environment
2. Through the **iccas** command from a Windows command line
3. Through the CTTHCMD command from a Windows command line
4. From an AS/400 command line

Invoking CRTPGM from the WorkFrame Environment

If you have already created a project that contains all the files for the application you are working on, you can invoke CRTPGM through the CRTPGM action.

If you have not yet created a project for your application and need help, see Part 1, " Developing OS/400 Applications with WorkFrame" on page 1 first.

Invoking CRTPGM through the /B Compiler Option

When you invoke the compiler from the Windows command line through the **iccas** command (using default compiler options), the binder attempts to create a program object on AS/400 (if you had established an AS/400 connection before invoking the compiler). `iccas myprog` is equivalent to specifying:

```
CRTPGM PGM(MYLIB/MYPROG) MODULE(MYPROG)
```

on an AS/400 command line, where `MYLIB` is the current library. (If there is no current library, `QGPL` is used.)

This invocation results in the creation of a program `MYPROG` on AS/400, using default parameters on the CRTPGM command.

Specifying Command Parameters: Instead of using default parameters, you may prefer to specify parameters of your choice. You can do so through the /B compiler option, which passes binding commands and binding options to the host. For example:

```
iccas /B"crtpgm pgm(mylib/abc) actgrp(xyz)" abc.cpp  def.cpp
```

results first in the creation of the module objects ABC and DEF, and then the command

```
CRTPGM PGM(MYLIB/ABC) ACTGRP(XYZ) MODULE(MYLIB/ABC MYLIB/DEF)
```

is passed to AS/400, where the program object ABC is created in library MYLIB, from the modules ABC and DEF located in the current library. The program will run in the activation group XYZ, as specified.

If you want to bind a module to be created to a module object that already exists, you do not need to recompile the source code for the existing module. For example, if module object DEF already exists in MYLIB, the command:

```
iccas /B"crtpgm mylib/abc" abc.cpp  def
```

results first in the creation of the module object ABC. Next, the command

```
CRTPGM PGM(MYLIB/ABC) MODULE(MYLIB/ABC MYLIB/DEF)
```

is passed to AS/400, where the program object ABC is created in library MYLIB, from the modules ABC and DEF.

Binding Several Module Objects: To bind two module objects ABC and DEF located in library MYLIB, invoke the compiler as follows:

```
iccas /B"crtpgm mylib/abc" abc def
```

which passes the command:

```
CRTPGM PGM(MYLIB/ABC) MODULE(MYLIB/ABC MYLIB/DEF)
```

to AS/400, where the program object ABC is created in library MYLIB, from the modules ABC and DEF.

Binding Modules Located in Different Libraries: You can also bind modules located in different libraries. For example

```
iccas /B"crtpgm mylib/mainpgm" /ASlmyl mainpgm.cpp /ASlmyrpg
   rpgmod1 /ASlmycobol cblmod2
```

which results first in the creation of the module object MAINPGM in library MYL. Next, the command

```
CRTPGM MYLIB/MAINPGM MODULE(MYL/MAINPG MYRPG/RPGMOD1 MYCOBOL/CBLMOD2)
```

is passed to AS/400, where the program object MAINPGM is created in library MYLIB, from the modules RPGMOD1 and CBLMOD2.

Invoking CRTPGM through the CTTHCMD Command

The CTTHCMD allows you to send a CL command to AS/400 directly from your workstation. If you want to use this interface to send the binding command, follow the syntax:

```
CTTHCMD /ASnname  hostcommand
or
@filename
```

as explained in “Invoking CL Commands through the CTTHCMD Command” on page 111.

The command invocation:

```
CTTHCMD /ASnconn1 CRTPGM PGM(MYLIB/ABC) ACTGRP(XYZ) MODULE(ABC DEF)
```

results in a program `ABC` being created in library `MYLIB`, from modules `ABC` and `DEF` located in the current library. The program will run in a named activation group `XYZ`.

Note: If you also specify the `/Q` parameter, you do not see the generated informational messages.

Invoking CRTPGM From an AS/400 Command Line

To create a program object using the CRTPGM command from AS/400:

1. Sign on to your AS/400
2. Type CRTPGM followed by the parameters you wish to specify.

If you are not sure of the exact parameters to choose from, type CRTPGM, then press the F4 Prompt key and enter the appropriate values for the command parameters as prompted. Pressing the F1 Help key may provide further assistance.

Resolving Import Requests

Once you have entered the CRTPGM command, the system performs the following actions:

1. Copies listed modules into what will become the program object and links any service programs to the program object.
2. Identifies the module containing the program entry procedure and locates the first import in this module.
3. Checks the modules in the order in which they are listed and matches the first import with a module export.
4. Returns to the first module and locates the next import.
5. Resolves all imports in the first module.
6. Continues to the next module and resolves all imports.
7. Resolves all imports in each subsequent module until all of the imports have been resolved.
8. If any imports cannot be resolved with an export, terminates the binding process without creating a program object.
9. Once all the imports have been resolved, completes the binding process and creates the program object.

Handling Duplicate Variable Names

If you have specified in the binder language that a variable is to be exported (using the **EXPORT** keyword), it is possible that the variable name will be identical to a variable in another procedure within the bound program object.

Use the **DUPPROC* option on the CRTPGM *OPTION* parameter to allow duplicate procedure names. See *ILE Concepts* for further information on how to handle this situation.

Using a Binder Listing

The binding process can optionally produce a *binder listing* that describes the resources used, symbols and objects encountered, and problems that were resolved, or not resolved, in the binding process.

The listing is produced as a spooled file for the job you use to enter the CRTPGM command. You can choose a *DETAIL* parameter value to generate the listing at three levels of detail:

- *BASIC
- *EXTENDED
- *FULL

The default is not to generate a listing. If it is generated, the binder listing includes the sections described in Table 6, depending on the value specified for *DETAIL*.

Table 6. Sections of the Binder Listing based on the DETAIL Parameter

Section Name	***BASIC**	***EXTENDED**	***FULL**
Command Option Summary	X	X	X
Brief Summary Table	X	X	X
Extended Summary Table		X	X
Binder Information Listing		X	X
Cross-Reference Listing			X
Binding Statistics			X

The information in this listing can help you diagnose problems if the binding was not successful, or give feedback about what the binder encountered during the binding process.

Figure 5 on page 117 shows the basic binder listing for a program `CVTHEXPGM`. Note that this listing is taken out of context. It only serves to illustrate the type of information you may find in a binder listing.

See the *ILE Concepts* , for more information on binder listings.

```
                                        Create Program                                          Page     1
5716SS1 V3R70  961101                                                MYLIB /CVTHEXPGM    AS400S01  11/02/96
23:24:00
 Program  . . . . . . . . . . . . . . . . . . . . . :   CVTHEXPGM
   Library  . . . . . . . . . . . . . . . . . . . . :    MYLIB
 Program entry procedure module . . . . . . . . . . :   *FIRST
   Library  . . . . . . . . . . . . . . . . . . . . :
 Activation group . . . . . . . . . . . . . . . . . :   *NEW
 Creation options . . . . . . . . . . . . . . . . . :   *GEN        *NODUPPROC  *NODUPVAR   *WARN       *RSLVREF
 Listing detail . . . . . . . . . . . . . . . . . . :   *BASIC
 Allow Update . . . . . . . . . . . . . . . . . . . :   *YES
 User profile . . . . . . . . . . . . . . . . . . . :   *USER
 Replace existing program . . . . . . . . . . . . . :   *YES
 Authority  . . . . . . . . . . . . . . . . . . . . :   *LIBCRTAUT
 Target release . . . . . . . . . . . . . . . . . . :   *CURRENT
 Allow reinitialization . . . . . . . . . . . . . . :   *NO
 Text . . . . . . . . . . . . . . . . . . . . . . . :   *ENTMODTXT
 Module      Library             Module      Library             Module      Library             Module      Library
 CVTHEXPGM  MYLIB
 Service                         Service                         Service                         Service
 Program     Library             Program     Library             Program     Library             Program     Library
 CVTTOHEX   MYLIB
 Binding                         Binding                         Binding                         Binding
 Directory   Library             Directory   Library             Directory   Library             Directory   Library
 *NONE
                                        Create Program                                          Page     2
5716SS1 V3R7M0  961101                                                 MYLIB/CVTHEXPGM    AS400S01  11/02/96
23:24:00
                                            Brief Summary Table
 Program entry procedures . . . . . . . . . . . . :   1
   Symbol    Type      Library     Object      Identifier
             *MODULE  MYLIB      CVTHEXPGM   _QRNP_PEP_CVTHEXPGM
 Multiple strong definitions  . . . . . . . . . . :   0
 Unresolved references  . . . . . . . . . . . . . :   0
                          * * * * *  E N D  O F  B R I E F  S U M M A R Y  T A B L E  * * * * *
                                        Create Program                                          Page     3
5716SS1 V3R7M0  961101                                                 MYLIB/CVTHEXPGM    AS400S01  11/02/96
23:24:00
                                             Binding Statistics
 Symbol collection CPU time . . . . . . . . . . . . . . . . . :               .016
 Symbol resolution CPU time . . . . . . . . . . . . . . . . . :               .004
 Binding directory resolution CPU time  . . . . . . . . . . . :               .175
 Binder language compilation CPU time . . . . . . . . . . . . :               .000
 Listing creation CPU time  . . . . . . . . . . . . . . . . . :               .068
 Program/service program creation CPU time  . . . . . . . . . :               .234
 Total CPU time . . . . . . . . . . . . . . . . . . . . . . . :               .995
 Total elapsed time . . . . . . . . . . . . . . . . . . . . . :              3.531
                              * * * * *  E N D  O F  B I N D I N G  S T A T I S T I C S  * * * * *
*CPC5D07 - Program CVTHEXPGM created in library MYLIB.
                          * * * * *  E N D  O F  C R E A T E  P R O G R A M  L I S T I N G  * * * * *
```

Figure 5. Example of a Basic Binder Listing

Changing a Module or a Program Object

There are many reasons why you may want to change a module or a program object:

- An object may need to be changed to accommodate enhancements, or for maintenance reasons.

  You can isolate what needs to be changed by using debugging information or the binder listing from the CRTPGM command. From this information you can determine what modules, procedures, or fields need to change.

- You may want to change the optimization level or observability of a module or program.

This is often the case when you want to debug a program or module, or when you are ready to put a program into production. Such changes can be performed more quickly and use fewer system resources than the re-creation of the object in question.

- You may want to reduce the program size once you have completed an application.

  ILE program objects have additional data added to them, which makes them larger than similar OPM or EPM program objects.

Each of the above approaches requires different data to make the change.

Updating a Program

In general, you can update a program by replacing modules as needed. You do not have to re-create the program object. The ability to replace specific modules is helpful if, for example, you are supplying an application to other sites that are already using the program. You need only send the revised modules, and the receiving site can update the application using the UPDPGM and UPDSRVPGM commands.

The update commands work with both program and module objects. The parameters for these commands are very similar to those for the Create Program (CRTPGM) command. For example, to replace a module in a program, you would enter the module name for the *MODULE* parameter and the library name.

To use the UPDPGM command, the modules to be replaced must be located in the same libraries they were in when the program was created. You can specify that all modules, or only some subsets of modules, are to be replaced.

See *ILE Concepts* and the *CL Reference* for more information on the UPDPGM command.

Changing the Optimization Level

Optimization is the process through which the system looks for processing shortcuts that reduce the amount of system resources necessary to produce output. Processing shortcuts are translated into machine code, allowing the procedures in a module to run more efficiently. A highly optimized program or service program should run faster than it would without optimization.

You control the level of optimization through the `/O` compiler options. Changing the desired optimization level requires recompiling your source code. See “Code Generation Options” on page 98 for details on the `/O` compiler options and optimization levels.

You should be aware of the following limitations when working with optimized code:

- In general, the higher the optimizing request, the longer it takes to create an object.
- At higher levels of optimization, the values of fields may not be accurate when they are displayed in a debug session, or after the program recovers from an exception.

- Optimized code may have altered breakpoints and step locations used by the source debugger, since the optimization changes may rearrange or eliminate some statements.

  To circumvent this restriction while debugging, you can lower the optimization level of a module to display fields accurately as you debug a program, and then raise the level again afterwards, to improve the program efficiency as you get the program ready for production.

Removing Observability

A module has **observability** when it contains data that allows it to be changed without being compiled again. Two types of data can make a module observable:

Create Data This data is necessary to translate the code into machine instructions. The object must have this data before you can change the optimization level. It is represented by the **CRTDTA* value on the *RMVOBS* parameter of the Change Program (CHGPGM) command.

Debug Data This data enables an object to be debugged. It is represented by the **DBGDTA* value on the *RMVOBS* parameter of the CHGPGM command.

The addition of these types of data increases the size of the object. Consequently, you may at some point want to remove the data in order to reduce object size. However, once the data is removed, so is the object's observability. To regain it, you must recompile the source and re-create the program.

To remove either kind of data from a program or module, use the CHGMOD or the CHGPGM command. Again, once you have removed the data, it is not possible to change the object in any way unless you re-create it. Therefore, ensure that you have access to all source required to create the program, or that you have a comparable program object with create data.

Reducing an Object's Size

The Create Data (**CRTDTA*) value associated with an ILE program or module may make up more than half of the object's size. By removing or compressing this data, you reduce the secondary storage requirements for your programs significantly.

An alternative is to compress the object through using the Compress Object (CPROBJ) command. A compressed object takes up less system storage than an uncompressed one. When the compressed program is called, the part of the object containing the executable code is automatically decompressed. You can also decompress an object by using the Decompress Object (DCPOBJ) command.

See the *CL Reference* for more information on the CPROBJ and DCPOBJ commands.

Chapter 9. Creating a Service Program

Creating a service program involves compiling source code into module objects and binding one or more module objects into a service program object.

An example illustrates how to create a service program.

Overview of the Service-Program Concept

A service program is an OS/400 object of type *SRVPGM. Service programs are typically used for common functions that are frequently called by other procedures within an application and across applications. For example, the ILE compilers use service programs to provide run-time services such as math functions and input/output routines.

Service programs simplify maintenance, and reduce storage requirements, because only a single copy of a service program is maintained and stored.

Differences between Programs and Service Programs

A service program differs from a program in two ways:

- A service program is bound to existing programs or other service programs. It cannot run independently.
- A service program does not contain a program entry procedure. Therefore, you cannot call a service program using an "OS" linkage specification. However, you can call a service program with a "c" linkage specification, because it does contain at least one user entry procedure.

  See the *C++ Programming Guide* for information on calling ILE procedures, and linkage specifications.

Binding a Service Program to a Program

Service programs are bound *by reference*. This means that the content of the service program is not copied into the program to which it is bound. Instead, *linkage information* about the service program is bound into the program.

This process is different from the static binding process used to bind modules into programs. However, you can still call the service program's exported procedures as if they were statically bound. The initial activation is longer, but subsequent calls to any of the service program's exported procedures are faster than program calls.

Invoking the Binder to Create a Service Program

Once an ILE module has been created, it can be used to create a service program with the Create Service Program (CRTSRVPGM) command. You can also use modules created with other ILE language compilers, such as ILE C/400, ILE RPG/400, or ILE COBOL/400.

Specifying Parameters for the CRTSRVPGM Command

You can specify a number of parameters on the CRTSRVPGM command, as listed in Table 7. Each parameter has default values which are used whenever you don't specify your own values.

See the *CL Reference* for a full description of the CRTSRVPGM command and its parameters.

Table 7. Parameters and Default Values for CRTSRVPGM Command

Parameter Group	Parameter(Default Value)
Identification	SRVPGM(*library name*/*service program name*) MODULE(*SRVPGM)
Program access	EXPORT(*SRCFILE)
	SRCFILE(*LIBL/QSRVSRC)
	SRCMBR(*SRVPGM)
Binding	BNDSRVPGM(*NONE)
	BNDDIR(*NONE)
Run time	ACTGRP(*CALLER)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF)
	DETAIL(*NONE)
	ALWUPD(*YES)
	ALWRINZ(*NO)
	REPLACE(*YES)
	AUT(*LIBCRTAUT)
	TEXT(*ENTMODTXT)
	TGTRLS(*CURRENT)
	USRPRF(*USER)

Invoking the CRTSRVPGM Command

You issue the CRTSRVPGM command in the same way as you issue the CRTPGM command:

- From the WorkFrame environment
- From a Windows command line through the `/B` compiler option
- Through the CTTHCMD from a Windows command line
- From an AS/400 command line

Invoking CRTSRVPGM from the WorkFrame Environment

If you have already created a project that contains all the files for the application you are working on, you can invoke CRTSRVPGM through the CRTSRVPGM action.

If you have not yet created a project for your application and need help, see Part 1, “ Developing OS/400 Applications with WorkFrame” on page 1 first.

Invoking CRTSRVPGM through the /B Compiler Option

By default, when you invoke the compiler from the Windows command line through **iccas**, a program object (*PGM) is automatically created on AS/400 (if a host connection was established prior to the invocation).

To create a service program (*SRVPGM) instead, you must specify the `/B` compiler option, and pass the CRTSRVPGM command and its parameters as a character string following the option. The command:

```
iccas /ASlmylib /B"crtsrvpgm srvpgm(mylib/sp1)" myprog.cpp
```

is equivalent to specifying the command:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/MYPROG)
```

on an AS/400 command line. Both commands result in the creation of the service program `SP1` on AS/400, from the source file `myprog.cpp`, using default parameters on the CRTSRVPGM command.

Creating a Service Program from an Existing Module: If the module already exists, you specify the module name instead of the name of the source file, when you invoke the compiler by specifying:

```
iccas /ASlmylib /B"crtsrvpgm srvpgm(mylib/sp1)" myprog
```

This command is equivalent to specifying:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/MYPROG)
```

on an AS/400 command line.

Creating a Service Program from a Source File and a Module: You can create a service program from a source file and an existing module by specifying:

```
iccas /B"crtsrvpgm srvpgm(service1)" myprog1.cpp  myprog2
```

which is equivalent to specifying:

```
CRTSRVPGM SRVPGM(MYLIB/SERVICE1) MODULE(MYLIB/MYPROG1 MYLIB/MYPROG2)
```

on an AS/400 command line. This command results in the creation of the service program `SERVICE1` on AS/400, from source file `myprog1.cpp`, and module `MYPROG2`.

Invoking CRTSRVPGM through the CTTHCMD Command

The CTTHCMD lets you send a command to AS/400 directly from your workstation. If you want to use this interface to send the binding command, follow the syntax:

```
CTTHCMD /ASn name hostcommand
or
CTTHCMD @filename
```

as explained in “Invoking CL Commands through the CTTHCMD Command” on page 111.

In the following example, the command invocation

```
CTTHCMD /ASnconn1 CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(M1)
```

results in the service program `SP1` being created in the library `MYLIB` from module `M1` located in the current library.

Invoking CRTSRVPGM from the AS/400 Command Line

To create a service program using the CRTSRVPGM command from AS/400:

1. Sign on to your AS/400.
2. Type CRTSRVPGM followed by the parameters you wish to specify.

If you are not sure of the exact parameters to choose from, type CRTSRVPGM, then press the F4 Prompt key and enter the appropriate values for the command parameters as prompted. Pressing the F1 Help key may provide further assistance.

Updating or Changing a Service Program

You can update or change a service program in the same way you modify a program object. In other words, you can:

- Update the service program (using UPDSRVPGM)
- Change the optimization level (using CHGSRVPGM)
- Remove observability (using CHGSRVPGM)
- Reduce the size (using CPROBJ).

See “Changing a Module or a Program Object” on page 117 for more information on any of the above points.

If you use binder language, a service program can be updated without requiring programs calling it to be recompiled. For example, to add a new procedure to an existing service program:

1. Create a module object for the new procedure.
2. Modify the binder-language source file to handle the interface associated with the new procedure. Add any new export statements following the existing ones. See “Updating a Service Program Export List” on page 135 for details on modifying binder-language source files.
3. Recreate the original service program and include the new module.

New programs can access the new functions. Since the old exports are in the same order, they can still be used by the existing programs. Until it is necessary to also update the existing programs, they do not have to be recompiled.

See *ILE Concepts* for more information on updating service programs.

Using Related CL commands

The following CL commands can be used with service programs:

- Create Service Program (CRTSRVPGM)
- Change Service Program (CHGSRVPGM)
- Display Service Program (DSPSRVPGM)
- Delete Service Program (DLTSRVPGM)

- Update Service Program (UPDSRVPGM)
- Work with Service Program (WRKSRVPGM).

Creating a Sample Service Program

The following example shows how to create a service program SEARCH that can be called by other programs to locate a character string in any given string of characters.

Creating the Source Files

The SEARCH program is implemented as a class object Search. The class Search contains:

- Three private data members: skippat, needle_p, and needle_size
- Three constructors, each taking different arguments
- A destructor
- An overloaded function where(), which takes four different sets of arguments

The service program is composed of the following files:

- A user-defined header file search.h
- A source code file search.cpp
- A source code file where.cpp

User Header File

The class and function declarations are placed into a separate header file, search.h, shown below:

```
// header file search.h
  // contains declarations for class Search, and inlined function
  // definitions

  #include <iostream.h>

  class Search  {
    private:
      char skippat[256];
      char * needle_p;
      int  needle_size;
    public:

  // Constructors
      Search( unsigned char * needle, int size);
      Search ( unsigned char * needle);
      Search ( char * needle);

  //Destructor
     ~Search ()  { delete needle_p;}
```

```
  //Overloaded member functions
      unsigned int where ( char * haystack)  {
         return where (haystack, strlen(haystack));
      }
      unsigned int where ( unsigned char * haystack)  {
         return where (haystack, strlen((const char *)haystack));
      }
      unsigned int where ( char * haystack, int size)  {
          return where ( (unsigned char *) haystack, size);
      }
      unsigned int where ( unsigned char * haystack, int size);
  };
```

Source Code Files

The definitions for the member functions of class `Search` that are not inlined in the class declaration are contained in two separate files: the source file `search.cpp`, which contains constructor definitions for class `Search`; and `where.cpp`, which contains the member function definition. These files are shown below:

```
// source file search.cpp
  // contains the definitions for the constructors for class Search

 #include "search.h"

    Search::Search( unsigned char * needle, int size)
       : needle_size(size) , needle_p ( new char [size])
    {
       memset (skippat, needle_size, 256);
       for (unsigned int i=0; i<size; ++i) {
          skippat [needle [i]] = size -i-1;
       }
       memcpy (needle_p, needle, needle_size);
    }
    Search::Search ( unsigned char * needle) {
       needle_size = strlen( (const char *)needle) ;
       needle_p = new char [needle_size];
       memset (skippat, needle_size, 256);
       for (unsigned int i=0; i<needle_size; ++i) {
          skippat [needle [i]] = needle_size -i-1;
       }
       memcpy(needle_p, needle, needle_size);
    }
    Search::Search ( char * needle) {
       needle_size = strlen( needle) ;
       needle_p = new char ffneedle_size“;
       memset (skippat,needle_size, 256);
       for (unsigned int i=0; i<needle_size; ++i) {
          skippat [needle [i]] = needle_size -i-1;
       }
       memcpy(needle_p, needle, needle_size);
    }
```

```
// where.cpp
    // contains definition of overloaded member function for class Search

    #include "search.h"

    unsigned int Search:: where ( unsigned char * haystack, int size)
    {  unsigned int i, t;
       int j;
       for ( i= needle_size-1, j = needle_size-1; j >= 0; --i, --j ){
           while ( haystack[i] != needle_p[j]) {
               t = skippat [ haystack [i]] ;
               i += (needle_size - j > t) ? needle_size - j : t ;
               if (i >= size)
                  return size;
               j = needle_size -  1;
           }
       }
       return ++i;
    }
```

The modules that result from the compilation of these source files, `SEARCH` and `WHERE` are bound into a service program, `SERVICE1`.

Compiling and Binding the Service Program

Establish a connection between your workstation and an AS/400 system, and decide how you want to invoke the compiler and binder.

For example, on the Windows command line, enter:

```
iccas /ASlmylib /B"crtsrvpgm srvpgm(mylib/service1)
   export(*all)" search.cpp where.cpp
```

to create service program `SERVICE1` from the C++source files `search.cpp` and `where.cpp`.

By default, the binder creates the service program in your current library on AS/400. The compiler option `ASl`*`name`* changes this target library to `MYLIB`.

The parameter *EXPORT(*ALL)* specifies that all data and procedures exported from the modules are also exported from the service program.

Binding the Service Program to a Program

A very short application consisting of a program `MYPROG` is bound to the service program to complete this example. The source code for `MYPROG`, `myprog.cpp`, is shown below.

Note: This sample application has been reduced to minimum functionality, its main purpose being to demonstrate how to create a service program.

```
// myprog.cpp
// Finds a character string in another character string.

   #include <stdio.h>
   #include <iostream.h>
   #include <stdlib.h>
   #include "search.h"
   #define HS "Find the needle in this haystack"

   void main () {
      int i;
      Search token("needle");
      i = token.where (HS, sizeof(HS));
      cout << "The string was found in position " << i << endl;
   }
```

The program creates an object of class `Search`. It invokes the constructor with a value that represents the string of characters ("needle") to be searched for. It calls the member function `where()` with the string to be searched ("Find the needle in this haystack"). The string "needle" is located, and its position in the target string "Find a needle in this haystack" is returned and printed.

To create the program `MYPROG` in library `MYLIB`, and bind it to the service program `SERVICE1`, enter the following on the Windows command line:

```
iccas /B"crtpgm pgm(mylib/myprog) bndsrvpgm(mylib/service1)" myprog.cpp
```

Figure 6 shows the internal and external function calls between program `MYPROG` and service program `SERVICE1`.

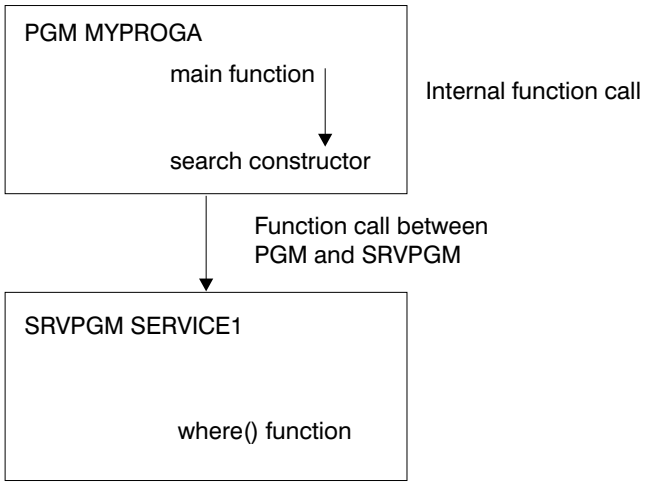


Figure 6. Calls between Program and Service Program

When `MYPROG` is created, it includes information regarding the interface it uses to interact with the service program.

To run the program from a Windows command line type:

```
CTTHCMD call mylib/myprog
```

During the process of making MYPROG ready to run, the system verifies that:

- The service program SERVICE1 in library MYLIB can be found.
- The public interface used by MYPROG when it was created is still valid at run time.

If either of the above is not true, an error message is issued.

The output of MYPROG is:

```
The string was found in position 9
```

Note: The output of this call is placed in a spool file QPRINT on AS/400. It is not visible from the Windows workstation.

Chapter 10. Working With Exports From Service Programs

This sections describes how to work with procedures and data items that can be exported from a service program.

Determining Exports from Service Programs

A service program exports procedures and data items that can be imported by other programs. These exports represent the *interface* to the service program. In the C++ programming language, procedures and data items correspond to functions and variables.

You can use information about exports that are available from the modules that form a particular service program to create a binder-language source file that defines the interface to this service program. A *binder-language source file* specifies the exports the service program makes available to all programs that call it. This file can be specified on the *EXPORT* parameter of the CRTSRVPGM command.

Binder language gives you better control over the exports of a service program. This control can be very useful if you want to:

- Determine export and import mismatches in an application.
- Add functionality to service programs.
- Reduce the impact of changes to a service program on the users of an application.
- Mask certain service-program exports from service-program users. That is, by not listing certain functions or variables in the binder-language source file, you can prevent any calling programs from having access to these exports.

Considerations when Creating a Service Program

When creating a service program, you should consider:

- Whether or not you intend to update the program at a later date.
- Whether or not any updates involve changes to its interface.

If the interface to a service program changes, you may have to rebind all programs bound to the original service program. However, depending on the changes and how you implement them, you may be able to reduce the amount of rebinding if you create the service program using binder language. In this case, after updating the binder language source to identify new exports, you need to rebind only those programs that require the new exports.

Displaying Export Symbols With the Display Module Command

To find out which exports are available from a module, type:

```
DSPMOD library-name/module-name
```

on an AS/400 command line, indicating the module name and the library where the module is stored. This command brings up the Display Module Information display. At the bottom of this display, you find information about exported defined symbols, consisting of the name and type of each symbol that can be exported from the module.

Note: When the compiler compiles a C++ source file, it encodes all C++ symbolic names to include type and scoping information. This encoding process is called *name mangling*. The symbol names in the sample display below are shown in mangled form. The source code for module `SEARCH` is shown in "Source Code Files" on page 126.

```
                          Display Module Information        Display 3 of 3
  Module . . . . . . . . . . . . . :   SEARCH
    Library  . . . . . . . . . . . :     MYLIB
  Detail . . . . . . . . . . . . . :   *EXPORT
  Module attribute . . . . . . . . :
                          Exported defined symbols:
  Symbol Name                                     Symbol Type
  __ct__6SearchFPc                                PROCEDURE
  __ct__6SearchFPUc                               PROCEDURE
  __ct__6SearchFPUci                              PROCEDURE
```

Figure 7. Display Module Information Screen for a Sample Module SEARCH

Creating a Binder-Language Source File

Binder language is based on the exports available from modules that are bound into service programs. A binder-language source file must contain the following entries:

1. The Start Program Export (STRPGMEXP) command identifies the beginning of the list of exports from the service program.
2. Export Symbol (EXPORT) commands identify each a symbol name available to be exported from the service program.
3. The End Program Export (ENDPGMEXP) command identifies the end of the list of exports from the service program.

The following example shows the structure of a binder-language source file:

```
STRPGEXP PGMLEVEL(*CURRENT)
  EXPORT SYMBOL("mangled_procedure_name_a")
  EXPORT SYMBOL("mangled_procedure_name_b")
  ...
  ...
  EXPORT SYMBOL("mangled_procedure_name_x")
ENDPGMEXP
```

Note: You must specify the mangled name of each symbol on the EXPORT command, because the binder looks for the mangled names of exports when it tries to resolve import requests from other modules.

Once all the modules to be bound into a service program have been created, you can create the binder-language source file. You can write this file yourself, using the Source Entry Utility (SEU), or you can let the AS/400 system generate it for you, through the Retrieve Binder Source (RTVBNDSRC) command.

Creating Binder Language Using SEU

You can use the Source Entry Utility (SEU) to create a binder-language source file:

1. Create a source physical file `QSRVSRC` on the AS/400 in library `MYLIB`.
2. Create a member `MEMBER1` that will contain the binder language
3. Use the DSPMOD command to display the symbols that can be exported from each module.
4. Decide which exports you want to make available to calling programs.
5. Use the Source Entry Utility (SEU) to enter the syntax of the binder language.

You need one export statement for each procedure whose exports you want to make available to the caller of the service program. Do not list symbols that you do not want to make available to calling programs.

For example, based on the information shown in Figure 7 on page 132, the binder-language source file for module `SEARCH` could list the following export symbols:

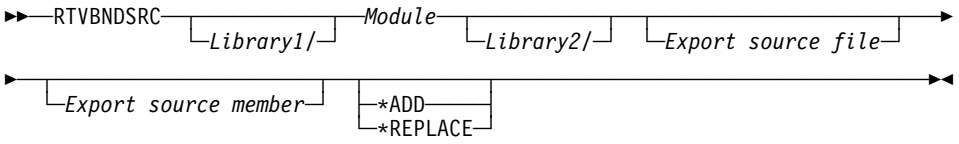
```
STRPGEXP PGMLEVEL(*CURRENT)
  EXPORT SYMBOL(" __ct__6SearchFPc")
  EXPORT SYMBOL(" __ct__6SearchFPUc")
  EXPORT SYMBOL(" __ct__6SearchFPUci")
ENDPGMEXP
```

Creating Binder Language Using the RTVBNDSRC Command

The Retrieve Binder Source (RTVBNDSRC) command can automatically create a binder-language source file. It retrieves the exports from a module, or a set of modules. It generates the binder language for these exports, and places exports and binder language in a specified file member. This file member can later be used as input to the *EXPORT* parameter of the CRTSRVPGM command.

Note: After the binder language has been retrieved into a source file member, you can edit the binder language and modify it as needed, for example, if you make changes to a module, or if you want to make certain exports unavailable to calling programs.

The syntax for the RTVBNDSRC command is:



The RTVBNDSRC command takes the following parameters:

Library1 Qualifies the name of a module.

Module Specifies the name(s) of the module(s) from which to retrieve the exported symbols.

Library2 Qualifies the name of the export source file.

Export source file
Specifies the name of the source file that is to hold the export source member. If the source file does not exist, it is created.

Export source member
Specifies the name of the export source member that is to hold the binder language for the exported symbols. If the member does not exist, it is created.

Add or replace statements
Specifies whether the generated binder language statements are added to existing statements or replace them.

For detailed information on the RTVBNDSRC command and its parameters enter RTVBNDSRC on an AS/400 command line and press F1 for Help.

Example of Creating Binder Language With RTVBNDSRC

The following example shows how to create a binder-language source file for module `SEARCH`, located in library `MYLIB`, using the RTVBNDSRC command. The source code for module `SEARCH` is shown in “Source Code Files” on page 126. On the Windows command line type:

```
CTTHCMD RTVBNDSRC MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QSRVSRC) SRCMBR(ONE)
```

This command automatically:

1. Creates a source physical file `QSRVSRC` on the AS/400 in library `MYLIB`.
2. Adds a member `ONE` to `QSRVSRC`.
3. Generates binder language from module `SEARCH` in library `MYLIB` and places it in member `ONE`.

Member `ONE` in file `MYLIB/QSRVSRC` now contains the following binder language:

```
 Columns . . . :    1  71           Browse                    MYLIB/QSRVSRC
 SEU==>                                                          ONE
 FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
        *************** Beginning of data ***********************************
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****************************************************************/
0000.03 /*   *MODULE      SEARCH       MYLIB        95/06/10  17:34:41       */
0000.04 /*****************************************************************/
0000.05   EXPORT SYMBOL("__ct__6SearchFPc")
0000.06   EXPORT SYMBOL("__ct__6SearchFPUc")
0000.07   EXPORT SYMBOL("__ct__6SearchFPUci")
0000.08 ENDPGMEXP
        ****************** End of data ***************************************
```

Figure 8. Binder-Language Source File Generated for Module SEARCH

Updating a Service Program Export List

You can use binder language to reflect changes in the list of exports a service program makes available. When you create binder language, a signature is generated from the order in which the modules that form a service program are processed, and from the order in which symbols are exported from these modules. The **EXPORT** keyword in the binder language identifies the procedure and data item names that make up the signature for the service program.

When you make changes to the exports of a service program this does not necessarily mean that all programs that call this service program must be re-created. You can implement changes in the binder language such that they are backward compatible. Backward-compatible means that programs which depend on exports that remain unchanged do not need to be re-created.

To ensure backward compatibility, add new procedure or data item names to the end of the export list, and recreate the service program with the same signature. This lets existing programs still use the service program, because the order of the unchanged exports remains the same.

Note: When changes to a service program result in a loss of exports, or in a change of existing exports, it becomes difficult to update the export list without affecting existing programs that require its services. Changes in the order, number, or name of exports result in a new signature that requires the re-creation of all programs and service programs that use the changed service program.

See *ILE Concepts* for a discussion of the signature of a service program.

Using the Demangling Functions

You can retrieve the mangled names of exported symbols with the RTVBNDSRC command. To help you find the corresponding demangled names, the runtime library contains a small class hierarchy of functions that you can use to demangle names and

examine the resulting parts of the name. The interface is documented in the *<demangle.h>* header file.

Using the demangling functions, you can write programs to convert a mangled name to a demangled name and to determine characteristics of that name, such as its type qualifiers or scope. For example, given the mangled name of a function, the program returns the demangled name of the function and the names of its qualifiers. If the mangled name refers to a class member, you can determine if it is `static`, `const`, or `volatile`. You can also get the whole text of the mangled name.

To demangle a name, which is represented as a character array, create a dynamic instance of the `Name` class and provide the character string to the class's constructor. For example, to demangle the name `f__1XFi`, create:

```
char *rest;
Name *name = Demangle("f__1XFi", rest);
```

The demangling functions classify names into five categories: function names, member function names, special names, class names, and member variable names. After you construct an instance of class `Name`, you can use the `Kind` member function of `Name` to determine what kind of `Name` the instance is. Based on the kind of name returned, you can ask for the text of the different parts of the name or of the entire name.

For the mangled name `f__1XFi`, you can determine:

```
name->Kind() == MemberFunction
((MemberFunctionName *) name)->Scope()->Text() is "X"
((MemberFunctionName *) name)->RootName() is "f"
((MemberFunctionName *) name)->Text() is "X::f(int)"
```

If the character string passed to the `Name` constructor is not a mangled name, the `Demangle` function returns `NULL`.

For further details about the demangling functions, refer to the information contained in the *<demangle.h>* header file. If you installed VisualAge for C++ for AS/400 using default settings, this header file should be located in the INCLUDE\V3R7M0 directory, the INCLUDE\V3R6M0 directory, and the INCLUDE\V3R2M0 directory under the main VisualAge for C++ for AS/400 installation directory.

Handling Unresolved Import Requests During Program Creation

An *unresolved import* is an import whose type and name do not yet match the type and name of an export. Unresolved import requests do not necessarily prevent you from creating a program or a service program. You can proceed in two ways:

- Specify the **UNRSLVREF* option on the CRTPGM or CRTSRVPGM commands to tell the binder to go ahead and create a program or service program, even if there are imports in the modules, and no matching exports can be found.
- Change the order of program creation to avoid unresolved references.

Both approaches are demonstrated in “Creating a Program with Circular References” on page 138.

Use the **UNRSLVREF* option to convert, create, or build pieces of code when all the pieces of code are not yet available. After the development or conversion phase has finished and all import requests can be resolved, make sure you re-create the program or service program that has the unresolved imports.

If you use the **UNRSLVREF* option, specify *DETAIL(*EXTENDED)* or *DETAIL(*FULL)*, or keep the job log when the object is created, to identify the procedure or data item names that are not found.

Note: If you have specified **UNRSLVREF* and a program is created with unresolved import requests, you receive an error message (MCH3203) when you try to run the program.

Creating a Service Program Using Binder Language

To create the service program described in “Creating a Sample Service Program” on page 125 using binder language, follow these steps:

1. To create modules from all source files enter the following compiler invocation on a Windows command line:

   ```
   iccas /C search.cpp  where.cpp
   ```

   The `/C` compiler option allows you to stop the compilation process after the creation of the modules `SEARCH` and `WHERE`. The binder will not be invoked.

2. To create the corresponding binder-language source file, enter the following command on the Windows command line:

   ```
   CTTHCMD rtvbndsrc module(mylib/search mylib/where)
      srcfile(mylib/qsrvsrc) srcmbr(two)
   ```

   This command creates the binder-language source file shown in Figure 9 on page 138.

3. To create service program `SERVICE2` invoke the binder as follows:

   ```
   iccas /B"crtsrvpgm srvpgm(mylib/service2) srcfile(mylib/qsrvsrc)
      srcmbr(two)" search where
   ```

```
 Columns . . . :    1  71          Browse                       MYLIB/QSRVSRC
 SEU==>   TWO
 FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
       ********** Beginning of data *************************************
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*********************************************************************/
0000.03 /*   *MODULE      SEARCH       MYLIB        95/06/11  15:30:51*/
0000.04 /*********************************************************************/
0000.05   EXPORT SYMBOL("__ct__6SearchFPc")
0000.06   EXPORT SYMBOL("__ct__6SearchFPUc")
0000.07   EXPORT SYMBOL("__ct__6SearchFPUci")
0000.08 /*********************************************************************/
0000.09 /*   *MODULE      WHERE        MYLIB        95/06/11  15:30:51*/
0000.10 /*********************************************************************/
0000.11   EXPORT SYMBOL("where__6SearchFPUci")
0000.12 ENDPGMEXP
       ****************** End of data ***************************************
```

Figure 9. Binder-Language Source File Generated by the RTVBNDSRC Command

Creating a Program with Circular References

A **circular reference** is a special case of unresolved import requests. It occurs, for example, when a service program `SP1` depends on imports from a service program `SP2`, which in turn depends on an import from service program `SP1`. Figure 10 illustrates the unresolved import requests between program `A` and two service programs, `SP1` and `SP2`.

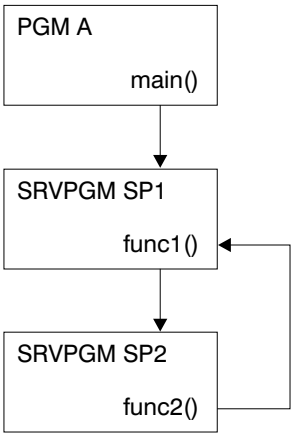


Figure 10. Unresolved Import Requests in a Program With Circular References

The following import requests occur between program `A` and the two service programs, `SP1` and `SP2`, that are called by A:

1. Program `A` uses function `func1()`, which it imports from service program `SP1`.

2. Service program SP1 needs to import function func2() provided by service program SP2, in order to provide func1() to program A.
3. Service program SP2, in turn, first needs to import func1 from service program SP1 before being able to provide func2.

Creating the Source Files

The application consists of three source files, m1.cpp, m2.cpp, and m3.cpp, shown below:

```
// m1.cpp
     #include <iostream.h>
     int main(void)
     {
       void func1(int);
       int n = 0;
       func1(n);             // Function func1() is called.
     }

// m2.cpp
     #include <iostream.h>
     void func2 (int);
     void func1(int x)
     {
       if (x<5)
       {
          x += 1;
          cout << "This is from func1(), n=" << x << endl;
          func2(x);          // Function func2() is called.
        }
     }

// m3.cpp
     #include <iostream.h>
     void func1(int);
     void func2(int y)
     {
       if (y<5)
       {
          y += 1;
          cout << "This is from func2(), n=" << y << endl;
          func1(y);          // Function func1() is called.
       }
     }
```

Creating Modules

Compile the source files m2.cpp and m3.cpp into module objects from which you later create the service programs SP1 and SP2. This allows you to display their exports with the DSPMOD command, or to generate binder-language source with the RTVBNDSRC command. To create module objects from the source files described above, invoke the compiler from the Windows command line with the command:

```
iccas /C /ASlmylib  m2.cpp m3.cpp
```

The /C compiler option indicates to the compiler that you do not want to create a program object from the source files. The /ASlmylib option instructs the compiler to create the module objects M2, and M3 in library MYLIB on the AS/400.

Creating Binder-Language

To generate binder language for module M2, from which you want to create service program SP1, issue the following command on a Windows command line:

```
CTTHMD rtvbndsrc module(mylib/m2) srcfile(mylib/qsrvsrc) srcmbr(bndlang1)
```

This command results in the following binder language being created for module M2, in library MYLIB, source file QSRVSRC, file member BNDLANG1:

```
 Columns . . . :    1  71          Browse                       MYLIB/QSRVSRC
 SEU==>                                                             BNDLANG1
 FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
        *************** Beginning of data ************************************
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*********************************************************************/
0000.03 /*   *MODULE      M2           MYLIB        95/06/11  18:07:04*/
0000.04 /*********************************************************************/
0000.05   EXPORT SYMBOL("func1__Fi")
0000.06 ENDPGMEXP
        ****************** End of data ***************************************
```

Figure 11. Binder Language for Service Program SP1

To generate binder language for module M3, from which you want to create service program SP2, issue the following command on a Windows command line:

```
CTTHMD rtvbndsrc module(mylib/m3) srcfile(mylib/qsrvsrc) srcmbr(bndlang2)
```

This command results in the following binder language being created for module M3, in library MYLIB, source file QSRVSRC, file member BNDLANG2:

```
 Columns . . . :    1  71          Browse                       MYLIB/QSRVSRC
 SEU==>                                                             BNDLANG2
 FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
        *************** Beginning of data ************************************
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*********************************************************************/
0000.03 /*   *MODULE      M3           MYLIB        95/06/11  18:08:14       */
0000.04 /*********************************************************************/
0000.05   EXPORT SYMBOL("func2__Fi")
0000.06 ENDPGMEXP
        ****************** End of data ***************************************
```

Figure 12. Binder Language for Service Program SP2

Creating the Program

Program A will be created from m1.cpp. Service program SP1 will be created from M2. Service program SP2 will be created from M3.

If you try and create service program SP1 from module M2, using the binder language shown in Figure 11 on page 140 and the compiler invocation:

```
iccas /B"CRTSRVPGM SRVPGM(MYLIB/SP1)
    SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)" m2
```

you find that the binder tries to resolve the import for function func2(), but fails, because it is not able to find a matching export. Therefore, service program SP1 is not created.

If SP1 is not created, this leads to problems if you try and create service program SP2 from module M3 using the binder language shown in Figure 12 on page 140 and the compiler invocation:

```
iccas /B"CRTSRVPGM SRVPGM(MYLIB/SP2)
      SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)" m3
```

Service program SP2 is not created, because the binder fails in searching for the import for func1() in service program S1P, which has not been created in the previous step.

If you try and create program A with the compiler invocation:

```
iccas/B"crtpgm pgm (A) bndsrvpgm(MYLIB/SP1 MYLIB/SP2) m1.cpp
```

the binder fails, since service programs SP1 and SP2 do not exist.

Handling Unresolved Import Requests with *UNRSLVREF

The following scenario shows how to use the parameter **UNRSLVREF* to handle the unresolved import requests which would otherwise prevent you from creating program A.

1. To create service program SP1 from m2.cpp, type:

   ```
   iccas /B"CRTSRVPGM SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRC)
         SRCMBR(BNDLANG1) OPTION(*UNRSLVREF)" m2.cpp
   ```

   Since the **UNRSLVREF* option is specified, service program SP1 is created even though the import request for func2() is not resolved.

2. To create service program SP2 from module M3, type:

   ```
   iccas /B"CRTSRVPGM SRVPGM(MYLIB/SP2) SRCFILE(MYLIB/QSRVSRC)
         SRCMBR(BNDLANG2) OPTION(*UNRSLVREF)" m3.cpp
   ```

   Since service program SP1 now exists, the binder resolves all the import requests required, and service program SP2 is created successfully.

3. To re-create the service program SP1, type:

```
iccas /B"CRTSRVPGM SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRC)
      SRCMBR(BNDLANG1) BNDSRVPGM(MYLIB/SP2)" m2
```

Although service program SP1 does exist, the import request for func2() is not resolved. Therefore, the re-creation of service program SP1 is required. Since service program SP2 now exists, the binder resolves all import requests required and, service program SP1 is created successfully.

4. To create program A, type:

   ```
   iccas/B"CRTPGM PGM(MYLIB/A) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)" m1.cpp
   ```

   Since service programs SP1 and SP2 do exist, the binder creates the program A.

Handling Unresolved Import Requests by Changing Program-Creation Order

You can also change the order of program creation to avoid unresolved references, by first creating a service program with all modules, and then re-creating this same service program later.

1. To generate binder language for modules M2 and M3, from which you want to create service program SP1, issue the following command on a Windows command line:

   ```
   CTTHMD RTVBNDSRC MODULE(MYLIB/M2 MYLIB/M3) SRCFILE(MYLIB/QSRVSRC)
      SRCMBR(BNDLANG3)
   ```

   This command results in the binder language shown in *Figure 13 on page 143* being created in library MYLIB, source file QSRVSRC, file member BNDLANG3.

2. To create service program SP1 from module M2 and module M3 type:

   ```
   iccas /B"CRTSRVPGM SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRC)
      SRCMBR(BNDLANG3)" m2.cpp m3.cpp
   ```

   Since modules M2 and M3 are specified, all import requests are resolved, and service program SP1 is created successfully.

3. To create service program SP2, type:

   ```
   iccas /B"CRTSRVPGM SRVPGM(MYLIB/SP2) SRCFILE(MYLIB/QSRVSRC)
       SRCMBR(BNDLANG2) BNDSRVPGM(MYLIB/SP1)" m3.cpp
   ```

   Since service program SP1 exists, the binder resolves all the import requests required and service program SP2 is created successfully.

4. To re-create service program SP1, type:

   ```
   iccas /B"CRTSRVPGM  SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRC)
       SRCMBR(BNDLANG1) BNDSRVPGM(MYLIB/SP2)" m2.cpp
   ```

   Although service program SP1 does exist, the import request for func2() is not resolved to the one in service program SP2. Therefore, a re-creation of service program SP1 is necessary to make the circular reference work.

   Since service program SP2 now exists, the binder can resolve the import request for func2() from service program SP2, and service program SP1 is successfully created.

5. To create program A, type:

```
iccas /B"CRTPGM PGM(MYLIB/A) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)" m1.cpp
```

Since service programs SP1 and SP2 do exist, the binder creates program A.

```
 Columns . . . :    1  71          Browse                      MYLIB/QSRVSRC
SEU==>                                                            BNDLANG3
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
       ************** Beginning of data ***********************************
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*********************************************************************/
0000.03 /*   *MODULE      M2           MYLIB        95/06/11  18:50:23      */
0000.04 /*********************************************************************/
0000.05   EXPORT SYMBOL("func1__Fi")
0000.06 /*********************************************************************/
0000.07 /*   *MODULE      M3           MYLIB        95/06/11  18:50:23      */
0000.08 /*********************************************************************/
0000.09   EXPORT SYMBOL("func2__Fi")
0000.10 ENDPGMEXP
       ***************** End of data ************************************
```

Figure 13. Binder Language for Service Program SP1

Binding a Program to a Non-existing Service Program

To successfully create a program or a service program, all required modules must exist prior to invoking the binder.

However, if you want to bind a program to a non-existing service program, you can create a "place-holder" service program first. Consider the following example:

A program MYPROG requires a function myprint() to be exported by a service program PRINT. The code for the program is available in myprog.cpp. However, the source for the service program does not yet exist. To work around this problem:

1. Create a source file dummy.cpp, such as:

   ```
   //dummy.cpp
      #include <iostream.h>
      void function(void) {
          cout << "I am a placeholder only" << endl;
          return;
      }
   ```

2. Compile and bind dummy.cpp into a service program PRINT:

   ```
   iccas /B"crtsrvpgm srvpgm(mylib/print)" dummy.cpp
   ```

3. Create the source file for program MYPROG:

   ```
   // myprog.cpp
   #include <iostream.h>
   #define size  80
   void print(char *);
   main() {
   ```

```
  char text[size];
  cout << "Enter text" << endl;
  cin >> text;
  print(text);
  return;
{
```

4. Create the program MYPROG from myprog.cpp and bind it to the service program PRINT:

   ```
   iccas /B"crtpgm pgm(mylib/myprog) bndsrvpgm(mylib/print)
         option(*unrslvref)" myprog.cpp
   ```

   The option **UNRSLVREF* ensures that the program will bind to the service program, although there is no matching export for MYPROG's import `void print(char *)`.

Before you can run program MYPROG successfully, you must re-create service program PRINT from the real source code, instead of from the place-holder code in dummy.cpp.

Note: MYPROG will only run successfully if PRINT actually exports a function that matches MYPROG's import request.

Updating a Service-Program Export List

The following example shows how to add a new procedure called cost2() to service program COST without having to re-create the existing program COSTDPT1 that requires an export from COST.

Program Description

The figure below shows the exports in the existing version of service program COST, and in the updated version.

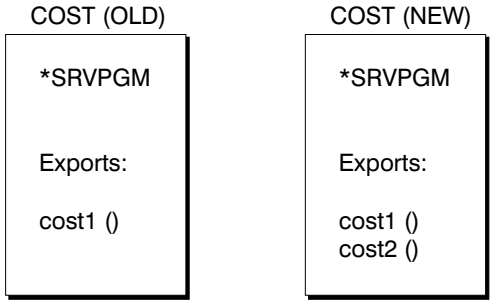


Figure 14. Exports from Service Program COST

The figure below shows the import requests in the existing program COSTDPT1, and in the new program COSTDPT2.

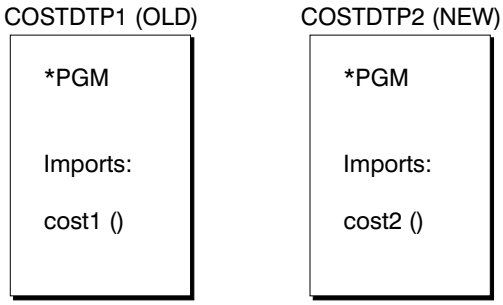


Figure 15. Import requests in Programs COSTDPT1 and COSTDPT2

The binder language for the old version of service program `COST` is located in member `BND` of source file `QSRVSRC`, in library `MYLIB`:

```
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

The export signature is 94898385315FD06BB65E44D38A852904.

The updated binder language includes the new export procedure `cost2()`. It is located in member `BNDUPD` of source file `QSRVSRC`, in library `MYLIB`:

```
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
  EXPORT SYMBOL("cost2__Fi9_DecimalTXSP10SP2_9_DecimalTXSP3SP1_")
ENDPGMEXP
```

The new export signature is 61E595C21D3EC9FDFD29749FB36B42D0.

In the binder language source that defines the old service program, the *PGMLVL* value is changed from **CURRENT* to **PRV*:

```
STRPGMEXP PGMLVL(*PRV)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

Its export signature is unchanged.

Note: If you want to ensure that existing programs can call the new version of the service program without being re-created, make sure to:

1. Add the new exports to the end of the symbol list in the binder language
2. Explicitly specify a signature for the new version of the service program that is identical to the signature of the old version.

   See *ILE Concepts* for detailed information on how to specify an explicit signature for a service program.

Creating the Source Files

The source code for service program COST, module COST2, and programs COSTDPT1 and COSTDPT2 is shown below:

```
// cost1.cpp
// contains the export function cost1() for the old service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1 (
  int q,                   // The quantity.
  _DecimalT<10,2> p )      // The price.
{
 _DecimalT<10,2> c;        // The cost.
 c = q*p;
 return c;
}


// cost2.cpp
// contains the export function cost2() for the new service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost2 (int quantity, _DecimalT<10,2> price,
                       _DecimalT<3,1>  discount )
{
  _DecimalT<10,2>  c = __D(quantity*price*discount/100);
  return c;
}

// costdpt1.cpp
// This program prompts users (from dept1) to enter the
// quantity, and price for a product. It uses function
// cost1() to calculate the cost, and prints the result out.
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1(int, _DecimalT<10,2>);
int main(void)
{
 int                quantity;
 _DecimalT<10,2>  cost;
 _DecimalT<10,2>  price;
 cout << "Enter the quantity, please." << endl;
 cin  >>  quantity;
 cout << "Enter the price, please." << endl;
 cin  >>  price;
 cost = cost1(quantity, price);
 cout << "The cost is $" << cost << endl;
}

// costdpt2.cpp
// This program prompts users (from dept2) to enter the
// quantity, price, and discount rate for a product.
// It uses function cost2() to calculate the cost, and prints
// the result out.
#include <iostream.h>
#include <decimal.h>
_DecimalT<10,2> cost2(int, _DecimalT<10,2>, _DecimalT<3,1>);
int main(void)
}
 int            quantity;
```

```
 _DecimalT<10,2>   price;
 _DecimalT<10,2>   cost;
 _DecimalT<3,1>    discount;
 cout << "Enter the quantity, please." << endl;
 cin  >>  quantity;
 cout << "Enter the price, please." << endl;
 cin  >> price;
 cout << "Enter the discount, please.( %)" << endl;
 cin  >> discount;
 cost = cost2(quantity, price, discount);
 cout << "The cost is be $" << cost << endl;
}
```

Compiling and Binding Programs and Service Programs

1. Create service program COST from source file cost1.cpp, using the binder source member BND, located in source file QSRVSRC, in library MYLIB:

   ```
   iccas /B"CRTSRVPGM SRVPGM(MYLIB/COST) SRCFILE(MYLIB/QSRVSRC)
       SRCMBR(BND) DETAIL(*EXTENDED)" cost1.cpp
   ```

2. Create program COSTDPT1 from source file costdpt1.cpp and service program COST, located in library MYLIB:

   ```
   iccas /B"CRTPGM PGM(MYLIB/COSTDPT1) BNDSRVPGM(MYLIB/COST)" costdpt1.cpp
   ```

3. Update service program COST to include module COST2, using the updated binder language source BNDUPD, located in source file QSRVSRC in library MYLIB:

   ```
   iccas /B"CRTSRVPGM SRVPGM(MYLIB/COST) SRCFILE(MYLIB/QSRVSRC)
       SRCMBR(BNDUPD) DETAIL(*EXTENDED)" cost1 cost2.cpp
   ```

   It is necessary to re-create the service program COST, using the two modules COST1 and COST2 and the updated version of the binder language BNDUPD, so that it supports the new cost2() function. Program COSTDPT1, which used COST before it was re-created, remains unchanged.

   In order to update service program COST, it is necessary to re-create it from the two modules COST1 and COST2, using the updated version of the binder language BNDUPD. The **EXTENDED* option in the *DETAIL* parameter creates an extended output listing, so that you can look at the current and previous signature of COST.

4. Create program COSTDPT2 from source file costdpt2:

   ```
   iccas /B"CRTPGM PGM(MYLIB/COSTDPT2) BNDSRVPGM(MYLIB/COST)" costdpt2.cpp
   ```

Running the Program

Run program COSTDPT1 from an OS/400 command line using the CL command CALL COSTDPT1.

Run program COSTDPT2 from an OS/400 command line using the CL command CALL COSTDPT2.

Chapter 11. Creating a Makefile

Use makefiles to simplify the process of compiling programs that are composed of several source files. This section describes how to create your own makefile for AS/400 applications compiled from a Windows workstation.

If you have created a WorkFrame project for your application, you can use WorkFrame's makefile-creation utility MakeMake to create your makefiles. See the *VisualAge for C++ for Windows User's Guide* for information on creating makefiles using the MakeMake utility.

Introducing Makefiles

Using a makefile simplifies the task of compiling programs that have more than one source file, especially when there have been changes to only some of the files. The makefile saves time by performing actions only on the files that have changed, and on the files that incorporate or depend on these changed files.

A *makefile* is a special text file that contains information about the files in your application.

- It describes:
  - Which files depend on others.
  - Which commands such as compile, create program, create service program, need to be carried out on each file to bring the application up-to-date.

  Based on this information, the makefile determines what actions are to be performed when you need to recompile after having made changes to one or more files.
- It compares the modification dates for your target files with the time stamps of all dependent files.

  Note: Makefiles compare object time stamps. They do not make adjustments for system-time differences. See “Restrictions When Comparing Time Stamps” on page 152 for the impact this behavior has in a distributed computing environment.
- If any dependent files have changed more recently than the target files, a series of commands specified in the makefile are executed.
- The use of a makefile is not limited to compiling and binding. You can also:
  - Make backups
  - Configure data files
  - Run programs when data files are modified.

  Note: The make process is not tied to the C++ compiler. You can also use it, for example, to build COBOL or RPG programs.

See “Writing a Makefile” on page 150 for additional information.

Writing a Makefile

The minimum information to include in a makefile is the name of the target file for compilation, the modules this file depends on, and the command with which you want to invoke the compiler, as shown in Figure 16:

```
//Description block
target: dependencies
    command

Example:
$(drive)a.pgm: $(drive)a.module $(drive)b.module
    iccas /b"crtpgm *curlib/a" a b
```

Figure 16. Format of a Makefile

Note: `$(drive)` must be replaced with the mounted drive that shadows your AS/400 libraries and files on the Windows workstation.

A dependent relationship between files is defined in a *description block*. The description block contains commands to bring all components up-to-date. The makefile can contain any number of description blocks.

If a dependency list is very long, you may have to split lines using a backslash (\), as shown in the following example:

```
$(drive)a.pgm: $(drive)a.module \
     $(drive)b.module \
     $(drive)c.module \
     $(drive)d.module \
     $(drive)e.module \
     $(drive)f.module
     iccas /b"crtpgm *curlib/a" a b c d e f
```

Note: The backslash must be immediately followed by a carriage return. Any other white space after this character invalidates the continuation.

You can include additional information in your makefile, such as commands to:

- Establish a connection between AS/400 and the Windows workstation prior to compilation
- Set up your library list on AS/400
- Mount a network drive.

Example of a Makefile

The makefile `aa.mak` contains information that is necessary to compile (or recompile) two source files, `aa.cpp` and `bb.cpp`. The objective is to build program AA on an AS/400 system called SYSAS1.

```
ALL: STARTUP Z:AA.PGM END
STARTUP:
  cttconn /hsysas1       // establish a host connection
  ctthcmd call mysetup/b // call to a program that performs specific setup task
  ctthcmd chgcurlib mylib // change the current library on the host
  ctttime sync            // optional synchronization of workstation and host time
  z:                      // change to the appropriate directory of the mounted drive
  cd \qsys.lib\mylib.lib
  e:                      // change to the appropriate directory of the workstation drive
  cd \sample

Z:AA.PGM: Z:BB.MODULE Z:AA.MODULE // Program AA is to be created from
  iccas -q aa bb                  // modules AA and BB, located on the
                                  // mounted drive Z:

Z:AA.MODULE: aa.cpp
  iccas -q -c aa.cpp

Z:BB.MODULE: bb.cpp
  iccas -q -c bb.cpp

END:
 cttdis                           // ends the connection established earlier
```

The source files for this example are shown below:

```
// aa.cpp
#include <iostream.h>
extern void bb(void);
void main()
{
  cout << "Here is program AA" << endl
       << "Calling program BB" << endl;
  bb();
  cout << "back to program AA" << endl;
}
```

```
// bb.cpp
#include <iostream.h>
extern void bb(void);
void bb()
{
  cout << "Here is program BB" << endl;
}
```

To create program `AA`, invoke the makefile by entering `nmake aa.mak` on a Windows command line.

Conditions and Restrictions of Makefiles

When working with VisualAge for C++ for AS/400, your files are usually scattered across a number of machines. Source files and intermediate code files may reside on your Windows workstation, while module and program objects are located on AS/400. This requires special consideration when you use a makefile to selectively recompile project files. The following conditions and restrictions apply:

- All AS/400 objects must be visible as files on the Windows workstation.

- Time stamps can only be compared reliably when all files involved are created and located on the same system, or in the same LAN domain.

Mounting a Host Path as a Network Drive on Windows

To mount a host path as a network drive on Windows type `net use d: \\machineid\netname` where:

- `d:` is any free drive letter on your system
- *machineid* is the network name of the server
- *netname* is the directory resource on the server.

To ensure automatic EBCDIC to ASCII conversion when you access files located on AS/400 from a Windows workstation you must update Client Access for Windows to add .H and .MBR to the list of file extensions to be translated:

1. Open the Client Access for Windows product folder on your Windows workstation
2. Open the Client Access Properties notebook
3. Select the Network Drives page
4. Enter `.H` in the **File extension** entry filed and click on **Add**
5. Enter `.MBR` in the **File extension** entry filed and click on **Add**
6. Click on **OK** and close the Client Access for Windows product folder.

You must restart your Windows workstation for these changes to take effect.

Restrictions When Comparing Time Stamps

Using makefiles works well only when the source and the target both reside on the same system, or in the same LAN domain. Otherwise, time-stamp mismatches are always a possibility.

Time-Stamp Mismatches

The time stamps of objects on a mounted drive reflect the moment when the corresponding object was created in AS/400 time, not in Windows workstation time. The time stamp of objects created on the Windows workstation reflects when the object was created in workstation time.

If the AS/400 time is earlier than the Windows workstation time, there is a chance that module objects will appear newer to a makefile than they actually are, and consequently, some necessary recompilations may be missed.

If the Windows workstation time is earlier than the AS/400 time, there is a chance that module objects will appear older than they are, and some unnecessary compilations may take place.

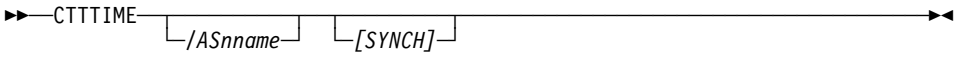
In some cases, it may be possible to synchronize the Windows workstation time and the AS/400 time, although, in most cases, this may lead to problems in other areas. For example, if two systems are in different time zones, time stamps of file objects may be

misleading, because they reflect local object-creation times. In such cases, synchronization of the two systems would not be useful.

If you are in doubt about whether object time stamps accurately reflect the order in which objects were created, you should recompile the entire application rather than rely on a makefile.

Synchronizing System Times with CTTTIME

In case you want to synchronize your Windows workstation time with the time of the AS/400 system you are connected to, you can use the Create Time (CTTTIME) command. The syntax for this command is:



The CTTTIME command takes the following parameters

/ASn*name* Identifies the connection you want to use between your Windows workstation and AS/400.

Specifying /ASn*name* is optional if you have already set a value for the environment variable ICCASNAME.

SYNC The keyword **SYNC** requests that Windows workstation and AS/400 time be synchronized. This keyword is optional.

Results of Specifying the Keyword SYNC

If you specify the keyword **SYNC**, CTTTIME changes the Windows workstation time to the time of the AS/400 system identified by /ASn*name*. If the synchronization is successful, the command returns a return code of zero, which means that the workstation local time is set to the local time of the host.

If you do not specify the keyword **SYNC**, CTTTIME returns the difference between workstation and host time, in seconds ranging from -32766 to +32767. If the command encountered an error, such as a communication problem between workstation and host, the return code is -32767.

The exact value of the return code has no significant meaning. Of more importance is to know that:

- A positive return code means that the AS/400 time is ahead of the Windows workstation time.
- A negative return code means that the Windows workstation time is ahead of the AS/400 time.
- A return code Zero indicates that both system times are identical.

Note: Calculated time differences and synchronized times are based on the local time of both systems, including daylight-savings adjustments. The local time will not be normalized into GMT.

Scope of Synchronizing Windows Workstation and AS/400

Synchronizing Windows workstation time and AS/400 system time will only affect objects created in the future. It will not affect objects created in the past, because they retain their former time stamp. To solve this problem, you could change the time stamp of the source file, but this may be difficult, if the source is not located on a local drive.

Changing the workstation time may create new problems, such as being out of synchronization with the LAN domain, or causing other existing applications to fail. The system time might also be changed again by other applications during the course of the makefile processing, which may cause unexpected results.

Chapter 12. Running a Program

This section describes how to run an AS/400 application.

Calling a Program

Once you have created an OS/400 program, the next step is to run it on an AS/400 system. So far, most of the development work has been done on your Windows workstation. Now, you must switch to AS/400 to run your application. Although you can launch an OS/400 application from a Windows session, any output is sent to a spool file QPRINT instead of displaying on the terminal, and is not visible from the Windows workstation.

Note: Only programs can run independently. Service programs or other bound procedures must be called from a program that requires their services.

When you call a program, the OS/400 system locates the corresponding executable code and performs the instructions found in the program.

There are several ways to call a program:

- Issue the CL CALL command.
- Issue the CL Transfer Control (TFRCTL) command.
- Issue a user-created CL command.

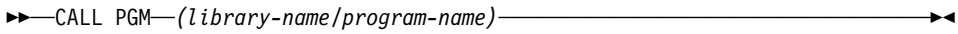
In addition, you can run a program using:

- The Programmer Menu (see *CL Programming* for information on this menu.)
- The Start Programming Development Manager (STRPDM) command (see *ADTS/400: Programming Development Manager* for information on this command).
- The EVOKE statement from an ICF file (see *ICF Programming* for information on the EVOKE statement).
- The QCAPEXC program (see *CL Programming* for information on this program)
- A high-level language (see the *C++ Programming Guide* for information on inter-language calls in the Integrated Language Environment. See Chapter 9, “Creating a Service Program” on page 121 for information on calling service programs and procedures).
- The REXX interpreter (See the *REXX/400 Programming Guide* for information on using REXX/400).

Using the CL CALL Command

From the AS/400 command line, you can use the CALL command to run a program interactively, or as part of a batch job.

The syntax for this CL command is:

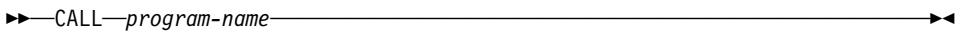


For example, the command

```
CALL PGM(MYLIB/MYPROG)
```

invokes the program `MYPROG` located in the library `MYLIB`.

If the program object specified by `program-name` exists in a library that is contained in your library list, you can omit the library name in the command, and the syntax is:



For example, if `MYLIB` appears in your library list, you can simply type:

```
CALL MYPROG
```

If you need prompting for the command parameters, type CALL and press F4 (Prompt). If you need help, type CALL and press F1 (Help).

Calling a Program Using the TFRCTL Command

You can run an application from within a CL program that transfers control to your program using the Transfer Control (TFRCTL) command. This command:

1. Calls the program specified on the command.
2. Transfers control to the called program.
3. Removes the transferring CL program from the call stack.

In the following example, the TFRCTL command in a CL program `RUNCP` calls a C++ program `XRUN2`, which is specified on the TFRCTL command. `RUNCP` transfers control to `XRUN2`. The transferring program `RUNCP` is removed from the call stack.

Figure 17 on page 157 illustrates the call to the CL program `RUNCP`, and the transfer of control to the C++ program `XRUN2`.

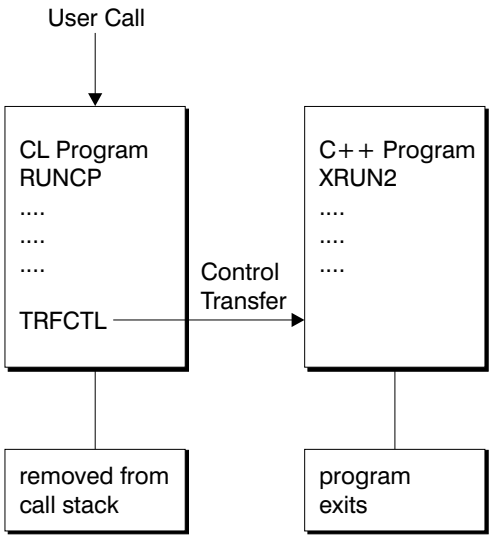


Figure 17. Calling Program XRUN2 Using the TFRCTL Command

To create and run programs RUNCP and XRUN2, follow the steps below:

1. Enter the source code shown below into a source physical file QCLSRC, in library MYLIB.

```
/* Source for CL Program RUNCP                                */
PGM        PARM(&STRING)
DCL        VAR(&STRING)   TYPE(*CHAR) LEN(20)
DCL        VAR(&NULL)     TYPE(*CHAR) LEN(1) VALUE(X'00')

/* ADD NULL TERMINATOR FOR THE ILE C++ PROGRAM                */
CHGVAR     VAR(&STRING) VALUE(&STRING *TCAT &NULL)
TFRCTL     PGM(MYLIB/XRUN2) PARM(&STRING)

/* THE DSPJOBLOG COMMAND IS NOT CARRIED OUT SINCE             */
/* WHEN PROGRAM XRRUN2 RETURNS, IT DOES NOT RETURN TO THIS    */
/* CL PROGRAM.                                                */
DSPJOBLOG
ENDPGM
```

2. Create the CL program RUNCP. From a Windows command line, specify:

   ```
   CTTHCMD CRTCLPGM PGM(MYLIB/RUNCP) SRCFILE(MYLIB/QCLSRC)
   ```

   If you prefer to create the program from the AS/400 command line, type:

   ```
   CRTCLPGM PGM(MYLIB/RUNCP) SRCFILE(MYLIB/QCLSRC)
   ```

   Program RUNCP uses the TFRCTL command to pass control to the ILE C++ program XRUN2, which does not return to RUNCP.

3. Create program XRUN2 in library MYLIB from the source file xrun2.cppshown below:

```
// xrun2.cpp
// Source for Program XRUN2
// Receives and prints a null-terminated character string

#include <iostream.h>

int main(int argc, char *argv[])
{
   int    i;
   char * string;
   string = argv[1];
   cout << "string = " <<  string << endl;
}
```

On a Windows command line type:

```
iccas /ASlmylib xrun2.cpp
```

Program XRUN2 receives a null terminated character string from the CL program and prints the string.

4. Run program RUNCP from an AS/400 command line, passing it the string "nails", with the command:

   ```
   CALL PGM(MYLIB/RUNCP) PARM('nails')
   ```

   The output from program XRUN2 is:

   ```
   string = nails
   Press ENTER to end terminal session.
   ```

Running a Program from a User-Created CL Command

You can also run a program from your own CL command. To create a command:

1. Enter a set of command statements into a source file.
2. Process the source file and create a command object (type *CMD) using the Create Command (CRTCMD) command.

   The CRTCMD command definition includes the command name, parameter descriptions, and validity-checking information, and identifies the program that performs the function requested by the command.
3. Enter the command interactively, or in a batch job.

   The program called by your CL command is run.

See *CL Programming* for further information about using commands.

The following example illustrates how to run a program from a user-created CL command:

Program Description

A newly created command COST prompts for and accepts user input values. It then calls a C++ program CALCOST and passes it the input values. CALCOST accepts the input values from the command COST, performs calculations on these values, and prints results. Figure 18 on page 159 illustrates this example.

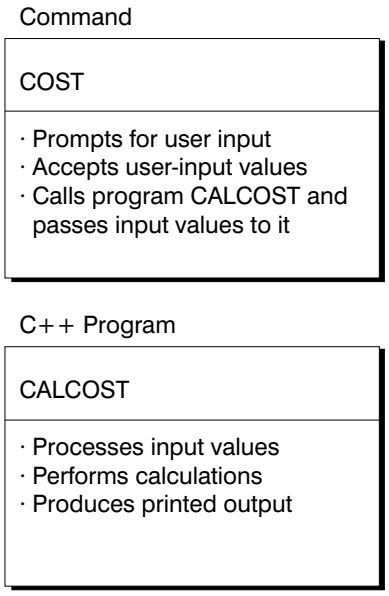


Figure 18. Calling Program CALCOST from a User-Defined Command COST

To create and run the example, follow the steps below:

1. Enter the source code for the command prompt COST shown below into a source file QCMDSRC in library MYLIB:

```
// Source for Command Prompt COST
     CMD        PROMPT('CALCULATE TOTAL COST')
     PARM       KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
                  MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
     PARM       KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
                  RANGE(0.01 99999999.99) MIN(1) +
                  ALWUNPRT(*YES) PROMPT('Unit price' 2)
     PARM       KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
                  9999) MIN(1) ALWUNPRT(*YES) +
                  PROMPT('Number of items' 3)
```

2. Create the CL command prompt COST. On an AS/400 command line, enter:

   ```
   CRTCMD CMD(MYLIB/COST) PGM(MYLIB/CALCOST) SRCFILE(MYLIB/QCMDSRC)
   ```

   or on a Windows command line enter:

   ```
   CTTHCMD CRTCMD CMD(MYLIB/COST) PGM(MYLIB/CALCOST) SRCFILE(MYLIB/QCMDSRC)
   ```

   Use the CL command prompt COST to enter item name, price, and quantity for the ILE C++ program CALCOST.

3. Create program CALCOST from the source file calcost.cpp shown below:

```
// calcost.cpp
// Source for Program CALCOST

#include <iostream.h>
#include <string.h>
#include <bcd.h>

int main(int argc, char *argv[])
{
  char                 *item_name;
  _DecimalT<10,2>      *price;
  short int            *quantity;
  const _DecimalT<2,2> taxrate=__D("0.15");
  _DecimalT<17,2>      cost;
  item_name = argv[1];
  price     = (_DecimalT<10,2> *) argv[2];
  quantity  = (short *) argv[3];
  cost = (*quantity)*(*price)*(__.D"1.00"+taxrate);
  cout << "\nIt costs $" << cost << " to buy "
       << *quantity << " " << item_name << endl;
}
```

This program receives the incoming arguments from the CL command `COST`, calculates a cost, and prints values. All incoming arguments are pointers.

To compile and bind source file`xrun2.cpp`, invoke the compiler from a Windows command line using the command:

```
iccas /B"crtpgm(mylib/calcost) output(*print)" calcost.cpp
```

4. Enter data for the program `CALCOST`. From the AS/400 command line, type `COST` and press F4 (Prompt).

   Type the data shown below into `COST`:

   ```
   Hammers
   1.98
   5000
   Nails
   0.25
   2000
   ```

   The output of program `CALCOST` is:

   ```
        It costs $11385.00 to buy 5000 HAMMERS
        Press ENTER to end terminal session.
     >
        It costs $575.00 to buy 2000 NAILS
        Press ENTER to end terminal session.
   ```

Passing Parameters to a Program

To pass parameters to an ILE program when you run it, use the *PARM* option of the CL CALL command. The syntax for this command is:

```
CALL PGM(program-name) PARM(param-1 param-2 ... param-n)
```

You can also type the parameters without specifying any keywords:

```
CALL library/program-name (param-1 param-2 ... param-n)
```

The following example demonstrates how to pass the value `'Hello, World'` to program `XRUN1` which expects parameters at run time. The source file `xrun1.cpp` for program `XRUN1` is shown below:

```
// xrun1.cpp
// Prints out command line arguments.

#include <iostream.h>
int main ( int argc, char *argv[])
{
   int i;
   for ( i = 1; i < argc; ++i )
      cout << argv[i] << endl;
}
```

Follow the steps below, to create and run program `XRUN1`:

1. Ensure that you have established a connection between your workstation and an AS/400 system.
2. Compile the source shown above with default compiler options. On the Windows command line type:

   ```
   iccas xrun1.cpp
   ```

   The resulting module and program objects are created into the default library on AS/400, in this example, `MYLIB`.
3. Run the program from an AS/400 command line using the command:

   ```
   CALL PGM(MYLIB/XRUN1) PARM('Hello, World')
   ```

   The output of program `XRUN1` is:

   ```
   Hello, World
   Press ENTER to end terminal session.
   ```

   Note: If you run the same program from a Windows command line, using the command:

   ```
   CTTHCMD /ASnmycon CALL PGM(MYLIB/XRUN1) PARM('Hello World')
   ```

   the output is sent to a spool file `QPRINT` on AS/400, and is not visible on the Windows workstation.

Processing Parameters

When you call a program from a CL command line, the parameters you pass on the CALL command are changed as follows:

- String literals passed are terminated by a null character.
- Numeric constants are passed as binary coded decimal digits.
- Characters that are not enclosed in single quotation marks are folded to uppercase and are passed with a null character.
- Characters that are enclosed in single quotation marks are not changed, and mixed-case strings are supported and are passed with a null character.

Table 8. Parameter Conversion from a CL Command to a Program

CL Call Command	Parameter Received	Conversion
CALL PGM(XRUN1) PARM(abc)	ABC\0	(converted to uppercase; passed as a string)
CALL PGM(XRUN1) PARM('123.4')	123.4\0	(passed as a string)
CALL PGM(XRUN1) PARM(123.4)	123.4	(passed as binary coded decimal digits (15,5))
CALL PGM(XRUN1) PARM('abc')	abc\0	(passed as a string)
CALL PGM(XRUN1) PARM('abC')	abC\0	(passed as a string)

Note: These changes only apply when you call a program from a command line, not to interlanguage calls.

See the *C++ Programming Guide* for information on interlanguage calls.

Ending a Program

When a program ends normally, the system returns control to the caller. The caller could be a workstation user or another program.

If a program ends abnormally during run time, and the program had been running in a different activation group from its caller, the escape message CEE9901 is issued and control is returned to the caller:

```
Application error <msgid> unmonitored by <pgm> at
     statement <stmtid>, instruction <instruction>
```

A CL program can monitor for this exception by using the Monitor Message (MONMSG) command.

See the *CL Reference* for more information about such CL commands.

If the program is running in the same activation group as is its caller and the program ends abnormally, what message is issued depends on how the program ended. If it ended with a function check, CPF9999 is issued. If the exception is issued by a C++ procedure, it has a message prefix of CTT.

See the section on handling exceptions in the *C++ Programming Guide* for more information on exception messages.

Managing Activation Groups

Activation groups make it possible for multiple ILE programs to run in the same job independently, without intruding on each other.

An activation group is a substructure of a job. It consists of system resources such as storage, commitment definitions, and open files. These resources are allocated to run one or more ILE or OPM programs. For example, the storage space for the static variables of a program is allocated from an activation group.

Once a program (type `*PGM`) is called, it remains activated until the activation group it runs in is deleted. Because service programs are not called directly, they are activated during the call to the program that requires their services.

See *ILE Concepts* for additional information on program activation.

Specifying an Activation Group

When an OS/400 job is started, the system automatically creates two activation groups to be used by OPM programs. One activation group is reserved for OS/400 system code. The other activation group is used for all other OPM programs. The symbol used to represent this activation group is *DFTACTGRP. You cannot delete the OPM default activation groups. The system deletes them when your job ends.

Note: OPM programs always run in the default activation group; you cannot change their activation group specification.

For ILE programs you specify the activation group that should be used at run time through the ACTGRP parameter of the Create Program or Create Service Program commands. You can choose between:

- Running your program in a named activation group.
- Accepting the default activation group:
  - **NEW* for programs
  - **CALLER* for service programs.
- Activating a program into the activation group of a calling program.

Running a Program in a Named Activation Group

To manage a collection of ILE programs and service programs as one application, you create a named activation group for them by specifying a user-defined name on the *ACTGRP* parameter.

The system creates the named activation group as soon as the first program that has specified this activation group is called. This group is then used by all programs and service programs that have specified its name.

A named activation group ends when it is deleted through the Reclaim Activation Group (RCLACTGRP) command. This command can only be used when the activation group is no longer in use. It also ends when you call the `exit()` function in your code.

When a named activation group ends, all resources associated with the programs and service programs of the group are returned to the system.

Note: Using named activation groups may result in non-ANSI compliant run-time behavior. If a C++ program created using named activation groups remains activated by a return statement, you encounter the following problems:

- Static variables will not be reinitialized.
- Static constructors will not be called again.
- Static destructors will not be called on return.
- Other programs activated in the same activation group may terminate your program, although they seem to be independent.
- Your program is not portable, if you count on the behavior of the named activation group.

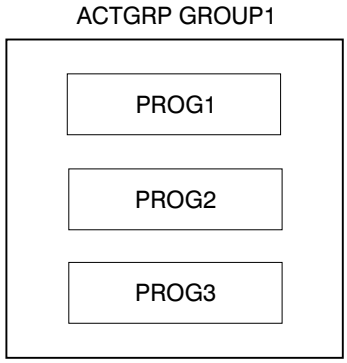


Figure 19. Running Programs in a Named Activation Group

In the following example, programs `PROG1`, `PROG2`, and `PROG3` are part of the same application and run in the same activation group, `GROUP1`. Figure 19 illustrates this scenario:

To create these programs in the same activation group, you specify `GROUP1` on the *ACTGRP* parameter when you create each program:

```
iccas /B"crtpgm pgm(prog1) actgrp(group1)" prog1.cpp
iccas /B"crtpgm pgm(prog2) actgrp(group1)" prog2.cpp
iccas /B"crtpgm pgm(prog3) actgrp(group1)" prog3.cpp
```

Running a Program in Activation Group *NEW

To create a new activation group whenever your program is called, specify **NEW* on the *ACTGRP* parameter. In this case, the system creates a name for the activation group that is unique within your job.

**NEW* is the default value of the ACTGRP parameter on the CRTPGM command. An activation group created with **NEW* always ends when the last program associated with it ends.

Note: **NEW* is not valid for a service program, which can only run in the activation group of its caller, or in a named activation group.

If you create an ILE C++ program with *ACTGRP(*NEW)*, more than one user can call the program at the same time without using the same activation group. Each call uses a new copy of the program. Each new copy has its own data and opens its files.

In the following example, programs `PROG4`, `PROG5`, and `PROG6` run in separate unnamed activation groups. Figure 20 illustrates this scenario:

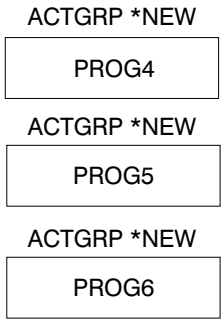


Figure 20. Running Programs in Unnamed Activation Groups

By default, each program is created into a different activation group, identified by the *ACTGRP* parameter *(*NEW)*.

```
iccas /B"crtpgm pgm(prog4) actgrp(*NEW)" prog4.cpp
iccas /B"crtpgm pgm(prog5) actgrp(*NEW)" prog5.cpp
iccas /B"crtpgm pgm(prog6) actgrp(*NEW)" prog6.cpp
```

Because **NEW* is the default, you obtain the same result with the following invocations:

```
iccas /B"crtpgm pgm(prog4)" prog4.cpp
iccas /B"crtpgm pgm(prog5)" prog5.cpp
iccas /B"crtpgm pgm(prog6)" prog6.cpp
```

Note: If you invoke three source files in one command, a single program object `PROG` is created in activation group **NEW*:

```
iccas /B"crtpgm pgm(prog)" prog7.cpp prog8.cpp prog9.cpp
```

Running a Program in Activation Group (*CALLER)

To run a program or service program in the activation group of a calling program, specify **CALLER* on the ACTGRP parameter.

If an ILE program created with *ACTGRP(*CALLER)* is called by an OPM program, it is activated into the OPM default activation group (*DFTACTGRP).

In the following example, a service program `SRV1` is activated into the respective activation groups of programs `PROG7` and `PROG8`. `PROG7` runs in a named activation group `GROUP2`, while `PROG8` runs in an unnamed activation group **NEW*. Figure 21 illustrates this scenario:

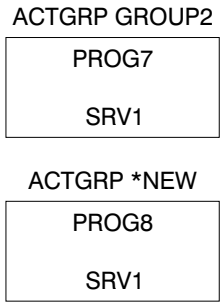


Figure 21. Running a Service Program in the Activation Groups of Calling Programs

By default,the service program `SRV1` is created into the activation group of each calling program.

```
iccas /B"crtsrvpgm srvpgm(srv1)" srv1.cpp
```

Presence of a Program on the Call Stack

Even though it is activated, a program does not appear on the call stack unless it is running. But an activation group can continue to exist even when the `main()` function of the program is not on the call stack.

This occurs when the program was created with a named activation group, and the `main()` function issues a return. It can also occur when the program performs a `longjmp()` across a control boundary by using a jump buffer that is set in an ILE C++ procedure. (This procedure is higher in the call stack and before the nearest control boundary.)

See *ILE Concepts* for information on control boundaries.

Deleting an Activation Group

When an activation group is deleted, its resources are reclaimed. The resources include static storage and open files. A **NEW* activation group is deleted when the program it is associated with returns to its caller.

Named activation groups are persistent. You must delete them explicitly. Otherwise they will end only when the job ends. The storage associated with programs running in named activation groups is not released until these activation groups are deleted.

The OPM default activation group is also a persistent activation group. The storage associated with ILE programs running in the default activation group is released either when you sign off (for an interactive job) or when the job ends (for a batch job).

Reclaiming System Resources

You may encounter situations where system storage is exhausted, for example:

- If many ILE programs are activated (that is, called at least once).
- If ILE programs that use large amounts of static storage run in the OPM default activation group (storage will not be reclaimed until the job ends).
- If many service programs are called into named activation groups (resources are only reclaimed when the job ends).

In such situations, you may want to reclaim system resources that are no longer needed for a program, but are still tied up because an activation group has not been deleted. You have the following options:

- Delete a named activation group that is not in use through the Reclaim Activation Group (RCLACTGRP) command .

  The command provides options to either delete all eligible activation groups or to delete an activation group by name. See the *CL Reference* for more information on RCLACTGRP.

- Free resources for programs that are no longer active through the Reclaim Resources (RCLRSC) command.

Using the Reclaim Resources Command (RCLRSC)

The RCLRSC command works differently depending on how the program was created:

- For OPM programs, the RCLRSC command closes open files and frees static storage.
- For ILE programs that were activated into the OPM default activation group (because they were created with **CALLER*), The RCLRSC command closes files and reinitializes storage. However, the storage is not released.
- For ILE programs associated with a named activation group, the RCLRSC command has no effect. You must use the RCLACTGRP command to free resources in a named activation group.

See the *CL Reference* for more information on the RCLRSC command. See *ILE Concepts* for more information on the RCLRSC and activation groups.

Run-Time Compliance with ANSI C++ Draft Semantics

The ILE C++ run-time model complies with ANSI C++Draft semantics when all programs in an application are created with the following options on the CRTPGM command:

ACTGRP(*NEW)
A new activation group is created on every call of the program, and the activation group is destroyed when the program is terminated.

OPTION(*NODUPPROC)
No duplicate procedure definitions in the same bound program are allowed.

OPTION(*NODUPVAR)
No duplicate variable definitions in the same bound program are allowed.

Service programs should be created with ACTGRP(*CALLER) when they are called by ANSI-compliant programs.

ILE C++ Run-Time Library Functions

All programs activated in the same activation group share one instance of the run-time library, because the ILE C++run-time library functions are bound to an application in the same activation group in which it is called.

The state of the run time propagates across program call boundaries. That is, if one program in an activation group changes the state of the run time (for example, the locale setting of an application), other programs in the same activation group are affected.

Non-ANSI Behavior with Named Activation Groups

If the *ACTGRP* parameter of the CRTPGM command is specified as a value other than **NEW*, the application's run-time behavior may not follow ANSI semantics. C++ run-time and class libraries assume that programs are built with ACTGRP(*NEW)

Non-ANSI behavior may occur during:

- Program termination - exit(), abort(), atexit()
- Signal handling - signal(), raise()
- Multibyte string handling - mblen()
- Any locale-dependent library functions - isalpha(), qsort().

In the default activation groups, I/O files are not automatically closed. The I/O buffers are not flushed unless explicitly requested.

Activation Within the Activation Group of a Calling Program

You can specify that an ILE program or an ILE service program be activated within the activation group of a calling program, by setting *ACTGRP* to **CALLER*. With this attribute, a new activation group is never created when the program or service program is activated. Through this option, ILE C++ programs can run within the OPM default activation groups when the caller is an OPM program.

Certain restrictions exist for ILE C++ programs running in the OPM default activation groups. For example, you are not allowed to register `atexit()` functions within the OPM default activation groups.

If the activation group is named, all calls to programs in this activation group within the same job share the same instance of the ILE C++ run-time library state.

It is possible to create an ANSI-compliant application whose programs are created with options other than *ACTGRP(*NEW)*. While non-ANSI behavior may be desirable in certain cases, it is the responsibility of the application designer to ensure that the sharing of resources and run-time states across all programs in the activation group does not result in incorrect behavior.

Managing Run-Time Storage

ILE allows you to directly manage run-time storage from your program, by managing heaps. A heap is an area of storage used for allocations of dynamic storage. The amount of dynamic storage required by an application depends on the data being processed by the programs and procedures that use the heap. You manage heaps by using ILE bindable APIs.

You are not required to explicitly manage run-time storage. However, you may wish to do so if you want to make use of dynamically allocated storage, for example, if you do not know exactly how big an array should be. In this case you could acquire the actual storage for the array at runtime, once your program determines how big the array should be.

There are two types of heaps available on the system:

- The default heap
- A user-created heap.

You can use one or more user-created heaps to isolate the dynamic storage required by some programs and procedures within an activation group.

The rest of this section explains how to use a default heap to manage run-time storage in a C++ program.

See *ILE Concepts* for information on creating a user-created heap and other ILE storage-management concepts.

Managing the Default Heap

The first request for dynamic storage within an activation group results in the creation of a **default heap** from which the storage allocation takes place. Additional requests for dynamic storage are met by further allocations from the default heap. If there is insufficient storage in the heap to satisfy the current request for dynamic storage, the heap is extended, and the additional storage is allocated.

Allocated dynamic storage remains allocated until it is explicitly freed, or until the heap is discarded. The default heap is discarded only when the owning activation group ends.

Programs in the same activation group all use the same default heap. If one program accesses storage beyond what has been allocated, it can cause problems for another program.

For example, assume that two programs, `PGM1` and `PGM2` are running in the same activation group. 10 bytes are allocated for `PGM1`, but 11 bytes are changed by it. If the extra byte was in fact allocated for `PGM2` problems may arise for `PGM2`.

Using Bindable APIs to Manage the Default Heap

You can use the following ILE bindable APIs on the default heap:

Free Storage (CEEFRST)
: Frees one previous allocation of heap storage

Get Heap Storage (CEEGTST)
: Allocates storage within a heap

Reallocate Storage (CEECZST)
: Changes the size of previously allocated storage.

See the *System API Reference* for specific information about the storage-management bindable APIs.

Dynamically Allocating Storage at Run Time

In an ILE C++ program, you manage dynamic storage belonging to the default heap using the operators `new` and `delete` to create and delete dynamic objects. Dynamic objects are never created and deleted automatically. Their creation can fail if there is not enough free heap space available, and your programs must provide for this possibility.

The following examples illustrate dynamic storage allocation with `new` and `delete`:

1. Dynamic allocation and de-allocation of storage for a class object:

```
TClass *p;                        // Define pointer
     p= new TClass;               // Construct object
     if (!p) {
        Error("Unable to construct object");
        exit(1);
     }
     ...
     delete p;                    // Delete object
```

2. Dynamic allocation and de-allocation of storage for an array of objects:

```
TClass *array                           // Define pointer
     array = new TClass[100];// Construct array of 100 objects
     ...
     delete[] array;         // Delete array
```

Note: In this example, you use `delete[]` to delete the array. Without the brackets, `delete` deletes the entire array, but calls the destructor only for the first element in the array. If you have an array of values that do not have destructors, you can use `delete` or `delete[]`.

Part 4. Debugging Your Program

The following sections describe how to debug AS/400 applications from your Windows workstation using the cooperative debugger, or from AS/400 using the ILE source debugger.

Chapter 13. Using the VisualAge for C++ for AS/400 Cooperative Debugger

The IBM VisualAge for C++ for AS/400 cooperative debugger (just called the debugger for the rest of this chapter) is a client/server program that helps you detect and diagnose errors in code developed with AS/400 ILE languages or the OPM languages COBOL, RPG, and CL. You can use the debugger to run your program, set breakpoints and watches, step through program instructions, examine variables, and examine the call stack.

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.

Preparing for Debugging

Before you can run the debugger on an OS/400 application, you must:

- Compile a program with debug data
- Set environment variables for the debugger
- Set up a debugger port
- Start the debug server if it is not already started
- Start the debugger.

To perform most of these tasks, your user profile must have the appropriate authorities.

Authorities Required for Using the Debugger

The user profile that you use to sign on to an AS/400 system from the debugger must have the following authorities:

- *USE authority to the Start Debug (STRDBG) command
- *USE authority to the End Debug (ENDDBG) command
- *USE authority to the Start Service Job (STRSRVJOB) command
- *USE authority to the End Service Job (ENDSRVJOB) command
- Either *CHANGE authority to the program being debugged, or *USE authority to the program being debugged and *SERVICE special authority.

If the job that you are debugging is running under a different user profile from the user profile you use to sign on to AS/400 from the debugger, the user profile that you use must have the following authorities:

- *USE authority to the user profile that the job you are debugging is running under.
- *JOBCTL special authority if you do not explicitly use fully qualified program names (library/program). In other words, if you use *CURLIB or *LIBL or you do not specify a library name, you need to have *JOBCTL special authority.

The group profile QPGMR will give a user the correct authority to the STRDBG, ENDDBG, STRSRVJOB, and ENDSRVJOB commands and *JOBCTL special authority.

Configuring the Debugger

Because the debugger is a client/server program, a method of communication is required to allow the client and server to communicate. The debugger uses the sockets programming services for communication. A Windows debugger client must know the name of the host where the debugger server and your application will run.

You can specify the name of the debug host through the ICCASDEBUGHOST environment variable, or through the */e**hostname*** command-line parameter. See "Setting Environment Variables for Debugging" for information on the ICCASDEBUGHOST environment variable. See "Starting a Debugging Session" on page 179 for information on the */e**hostname*** command-line parameter.

Setting Debugger Ports

The debugger client and server communicate with each other using TCP sockets. TCP sockets communicate through ports. When shipped, the debug server is set up to listen for connection requests on TCP port 3001. If port 3001 is being used by another application, you can change it by using the Work Server Table (WRKSRVTBLE) command on the AS/400.

To change the port used by the server, modify the port called QDBGSVR. Then set the ICCASDBGPORT environment variable on all debugger clients to match the port specified on the WRKSRVTBLE command. For information on environment variables, see "Setting Environment Variables for Debugging."

Note: Before changing the port, end the debug server by using the End Debug Server (ENDDBGSVR) command. Then change the port and start the debug server again.

Starting the Debug Server

Before you can use the debugger, the debug server must be started on the AS/400 with the Start Debug Server (STRDBGSVR) command.

Ending the Debug Server

Normally, you do not need to end the debug server at the end of your debug session, because the debug server is active for the entire AS/400 system. If necessary, the debug server can be ended using the End Debug Server (ENDDBGSVR) command. This prevents any new debug sessions from starting, but active sessions continue until completion or until they are cancelled.

Setting Environment Variables for Debugging

You can use several environment variables with the debugger. The following environment variables may be set for use with the debugger:

ICCASDEBUGHOST

Specifies the name of the host. ICCASDEBUGHOST is an optional environment variable.

To set the AS/400 environment for the debugger, type the following at the Windows command prompt:

```
SET ICCASDEBUGHOST=debughostname
```

where `debughostame` is one of the following:

- An IP address, such as `9.21.30.29`
- A hostname which is defined in your local host table or on a name server to which you link, such as `name.toronto.ibm.com`.

To determine the IP address of your AS/400, type GO CFGTCP on an AS/400 command line, and choose menu option 12. (Be careful not to change the local domain name and hostname when you use option 12.)

ICCASTAB

Specifies the number of spaces per tab. ICCASTAB is an optional environment variable.

To set tab stops at particular intervals, type the following at the Windows command prompt:

```
SET ICCASTAB=number
```

where `number` is between 1 and 64 inclusive. If ICCASTAB is not set, the default is 8. For example, SET ICCASTAB=5 will cause text after a tab to begin in columns 6, 11, 16, 21, 26, and so on.

ICCASUPRD

Specifies whether to allow the update of production files. ICCASUPRD is an optional environment variable.

To allow the update of production files, type the following at the Windows command prompt:

```
SET ICCASUPRD=Y
```

where `Y` allows production files to be updated, and `N` does not allow production files to be updated. By default, ICCASUPRD is set to `N`.

ICCASDEBUGPATH

Specifies the search paths for the source code files. ICCASDEBUGPATH is an optional environment variable.

To set the search path for the source code file used by the debugger, type the following at the Windows command prompt:

```
SET ICCASDEBUGPATH=path1;path2;
```

where `path*` is the path, including the drive and library names, to the location where the source code files are stored. Multiple paths must be separated with semicolons.

ICCASDBGPORT

Specifies the port used to connect to the AS/400. ICCASDBGPORT is an optional environment variable.

To connect to the AS/400 using a port other than the default port, type the following at the Windows command prompt:

```
SET ICCASDBGPORT=port number
```

where `port number` is a value between 1 and 64,767 that matches the port number specified for the QDBGSVR entry on the AS/400.

Compiling a Program with Debug Data

If you want to debug your application, you must first compile your modules or programs to include debug data. The debugger lets you debug applications that are written in one or more of the following AS/400 languages: ILE C++, ILE C, ILE COBOL, ILE RPG, ILE CL, as well as in OPM COBOL, OPM RPG, and OPM CL.

Compiling ILE C++ Programs

To create debug data in your program do the following:

1. Compile your code with one of the following options to request one or more debug views:

   /Ti+ Compiles your program to produce a module that includes a source view, a listing view, and a statement view for debugging purposes.

   /Til Compiles your program to produce a module that includes a listing view and a statement view for debugging purposes.

   /Tis: Compiles your program to produce a module that includes a source view and a statement view for debugging purposes.

   /Tin: Compiles your program to produce a module that includes only a statement view for debugging purposes.

2. Bind the program. (This step is omitted for OPM programs.)

Compiling ILE Languages Other than C++

If you want to debug programs written in languages other than C++, see the appropriate ILE language manual for compiling with debug data.

Compiling OPM Languages with Debug Data

Programs written in OPM COBOL or OPM RPG need to be compiled with either the *SRCDBG or the *LSTDBG option to be debuggable in the ILE environment. OPM CL programs need to be compiled with the *SRCDBG option.

Debugging Optimized Code

Generally, the higher the optimization level, the more efficiently the procedures in the module run. However, higher optimization levels adversely affect breakpoints that are set with the debugger.

While debugging your code, set the optimization level to the minimum level (**NONE*). This allows you to accurately display and change variables. After you have completed your debugging session, set the optimization level to the maximum level. This provides the highest levels of performance for the procedures in the module.

Note: Even at optimization level **NONE*, some optimization may be done in certain cases that could affect the debugger's ability to accurately display the program's stopped location.

Starting a Debugging Session

To start the debugger from the Windows command prompt, perform the following steps:

1. Ensure that the debug server is running on the AS/400.
2. Start the debugger on your Windows workstation by entering the debugger invocation command **idebugas**. To use parameters with this command, typing

   ```
   idebugas /x program-name
   ```

   where `/x` represents any number of debugger parameters. Choose from the following parameters:

   /a Run the program, but do not stop at main. The program runs until it completes or comes to a breakpoint. This parameter has the same function as the shortcut keys Ctrl-R.

   /e*hostname* Specify the name of the host system that you want to connect to for your debugging session. If you do not specify the `/e`*`hostname`* command-line parameter, the hostname specified in the ICCASDEBUGHOST environment variable is used. If you do not specify the `/e`*`hostname`* parameter and the ICCASDEBUGHOST environment variable is not set, you must specify the hostname in the AS/400 Logon window.

   /i Step into the program, but do not run to main. This parameter has the same function as the **Step into program** check box on the Startup Information window.

   /j*jobname* Specify a job name, which requires the job-name operand. Job name is specified as it is on the Startup Information window. No spaces should be placed between the `/j` parameter and the job-name operand.

   If you specify the `/j` parameter and the program name, you will bypass the Startup Information window.

   /p+ Use program profile information, if program profile information is available. This is the default.

   /p- Do not use any program profile information.

   The AS/400 Logon window opens as shown in Figure 22 on page 180 if one of the following is true:

- You have not specified the */ehostname* parameter and the ICCASDEBUGHOST environment variable is not set.
- You are connecting to this particular AS/400 host for the first time with the */ehostname* parameter after restarting your system.

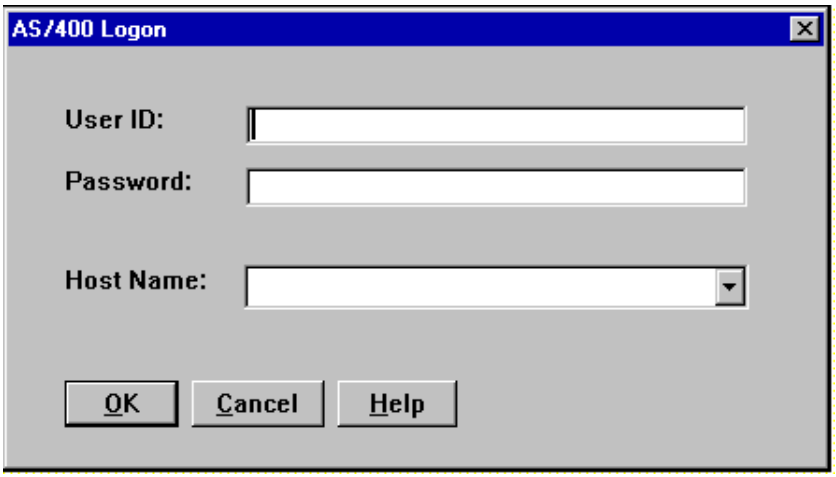


Figure 22. AS/400 Logon Window

3. Enter your user id, password, and the name of your AS/400 host in the AS/400 Logon window and click on **OK**. Next, the Debug Session Control window and the Startup Information window shown in Figure 23 open.

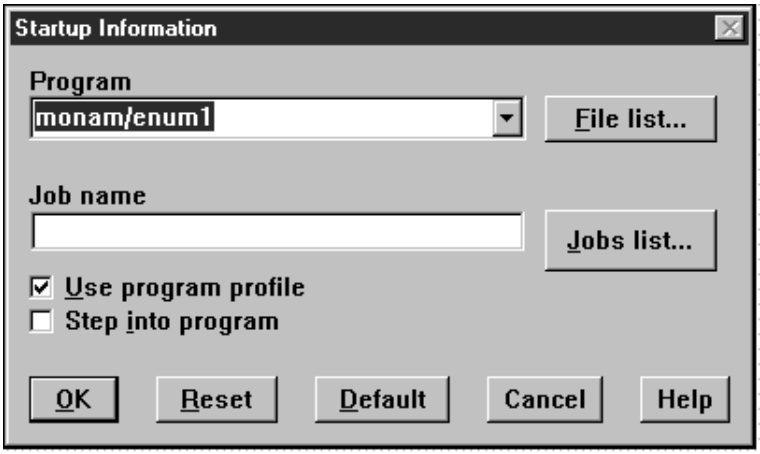


Figure 23. Startup Information Window

4. In the Startup Information window enter the following:

- In the **Program** entry field , either type the name of the program you want to debug, or click on the down arrow on the **Program** entry field and select a program.
- In the **Job name** entry field, type the name of the AS/400 job you will use when the program to be debugged is run.

  Often, this is the job associated with a 5250 terminal session, but it can be any AS/400 server job, a batch job, a prestarted job, or an invoked job. If you select **Jobs list**, the Jobs List window is shown. From this window, select the job you want and select **OK**.

  The **Startup Information** window is updated and the job name you selected now displays in the **Job name** entry field.
- Enable the **Use program profile** check box to restore the debugger windows and breakpoints when debugging a program more than once. The program profile is stored separately for each program debugged.
- Enable the **Step into program** check box to stop your program at the next runnable statement encountered. Use this option to debug C++ constructors for objects that are located in static storage or to debug a program that is already running. You should also enable this check box for ILE and OPM languages other than C and C++. Otherwise, programs that do not contain a main() function generate a debugger error message.
- Select **OK** to accept the information you have entered. The Startup Information window closes, and the debugger is started. A debugger message appears that asks you to start the program to be debugged on AS/400.

5. Start the program on AS/400
6. Press the **OK** button in the debugger message window to continue. The Debug Session Control window opens, and the source of your program is displayed.

Ending the Debugging Session

End the debugging session in one of the following ways:

- Press F3 in any of the debugger windows
- Select **Close debugger** from the **File** menu-bar choice in a debugger window
- If no debugger window is open, press Ctrl-C in the Windows command window from which you started the debugger. (If you started the debugger using an icon, click on that same icon again to bring up the Windows command window.)

  This may happen after you pressed the **OK** push button on the **Startup Information** window, but before the **Debug Session Control** window displays.

You may want to end the debugger session in one of the following situations:

- In the **Startup Information** window, you specify the name of a job that is an active AS/400 job, but it is not the job that you wanted to use.

- You specify the name of a program on the **Startup Information** window, but the program fails when you call it (for example, a signature violation occurs when you call the program).
- The job ends on the AS/400 before the startup program is called.

Locating Source Code

How the debugger finds source code depends on whether this source code is located on the Windows workstation, or on AS/400.

C++ Source Code

When you compile a C++ program and specify the source debug view by using the `/Ti+` option, the compiler stores the name of the source file and its directory path in the module object. When you try to display the source file, the debugger tries to find the file using the directory path and file name that was captured when the program was compiled. The debugger searches for the file in this order:

1. In the directory path that was stored when the module was compiled.
2. In the directory where the last file, if any, was found.
3. In the directories defined in the ICCASDEBUGPATH environment variable.

If the file is not found in any of these directories, you will be prompted for the name of the file. If the source file is not available, press the **Cancel** push button on the file prompt and a different view will be used.

Source Code Written in Other ILE or OPM Languages

When you compile other ILE (C, RPG, COBOL, and CL) modules, or OPM (CL, COBOL, RPG) programs, the name of the AS/400 source member is stored in the module or program object. When you try to display the source for a module or program, the debugger attempts to read the information from the AS/400 source member used to create the module or program.

If the source member is not found, the debugger searches your workstation for a source file using the following steps:

1. The debugger builds a name using the module name or program name and a file extension based on the language. The file extensions used are as follows:

   RPG For RPG language source

   C For C language source

   CBL For COBOL language source

   CL For CL language source.

2. The debugger searches for the file in this order:

   a. In the directory where the last file, if any, was found

b. In the directories listed in the ICCASDEBUGPATH environment variable.

If the file is not found, you are prompted to enter the name of the file. If you do not have the source available, press the **Cancel** push button on the file prompt and a different view will be used.

Frequently Used Features of the Debugger

This section introduces

- The title bar buttons
- Ways to run your program
- How to set breakpoints
- The use of multiple statements in program code
- Debugger performance considerations
- Debug limits.

Using the Tool Buttons

Buttons provide easy access to frequently used features. Check the tool bar on each window to determine which buttons are active for the window.

Step over runs the current, highlighted line in the called procedure, but does not stop in it. If the current line is a call, the program is halted when the call is completed.

Step debug runs the current, highlighted line in the program. The debugger steps over any procedure for which debugging information is not available (for example, library and system routines), and steps into any procedure for which debugging information is available.

Run runs the program. Control returns to the debugger when:

- The program stops at an enabled breakpoint or watch.
- The program encounters an exception
- The program ends.

When the debugger is running, the run button is grayed out.

 **View** changes the current program view window to one of the other program view windows, depending on the views available for the module or program. For example, you can change from the *LISTING view to the *STATEMENT view.

 **Call stack** displays the **Call Stack** window, which allows you to view all of the call stack entries. The procedures are displayed in the order that they were called.

 **Monitor**, from a program view window, displays the Monitor Expression window. This button also appears on the left side of the title bar on the Program Monitor window and the **Local Variables** window.

 **Breakpoints** displays the Breakpoints List window, which allows you to view all the breakpoints that have been set.

 **Debug Session Control** transfers control to the Debug Session Control window.

 **Next Representation** displays the next representation of the selected variable.

 **Delete** deletes the highlighted expression or variable.

 **Delete All** deletes all expressions and variables.

 **Growth Direction** changes the order of the entries on the call stack.

Running a Program

You can run your program by using Step commands, the Run command, or the Run to location command:

- Step commands control how the program runs. The execution of a line of code is reflected in all open views.

The step commands are located in the title bars and under the **Run** menu-bar choice of the program view windows.

- The **Run** command runs the program until a breakpoint or watch is encountered, the program is halted, or the program ends.

  The **Run** command is located in the title bars and under the **Run** menu-bar choice of the program view windows and the **Debug Session Control** window.

- The **Run to location** command runs the program to the current position in the current program view window. This command performs the following steps:

  – Add a breakpoint at the target line

  – Run the program

  – Remove the breakpoint.

  To select a position, press mouse button 1 once on the prefix area of a runnable statement. The prefix area is the area on the left of the program view window where the line numbers are displayed.

  The **Run to location** command is located under the **Run** menu-bar choice of the program view windows. It is also available in the popup menu for the line number, if mouse button 2 is configured for popup.

Setting Breakpoints

You can control how your program runs by setting breakpoints. A breakpoint stops your program at a specific location.

To set breakpoints in one of the following ways:

- Select the **Breakpoints** menu from the Debug Session Control window or from any of the program views windows.
- Double-click in the prefix area of a runnable statement in any of the program view windows.

  The prefix area is the area on the left of the program view window where line numbers are displayed.

  The prefix area turns red indicating that the breakpoint has been set. (The prefix area color is configurable.)

Setting Watches

You can monitor the values of expressions or variables while your program runs by setting watches. A **watch** monitors the content of the storage location of an expression or variable.

When the content of the watched location changes, the program stops at the statement following the change, and this line is highlighted. If the program that caused this expression or variable to be changed has not been added to the debugger, it is automatically added if:

- The program contains debug data

- You have authorization to the program.

You set watches in one of the following ways:

- Highlight a variable in the source window. Click on Breakpoints and select the Set watch action for the selected variable. The Watch window opens. It contains the name of the highlighted variable.
- Select the Set watch action on the Breakpoints pull-down which is available from several debugger windows.

  If the watch dialog is invoked from a window other than the source window (in which case default values are provided by the debugger), you must fill in the variable or expression to be watched. Otherwise, the watch will fail. If you do not specify values for the other entry fields in this dialog, default values are used. These defaults are:

  Bytes to Watch The number of bytes defined by the variable length are watched, up to 128 bytes. The default value is 0 which means that the declared type length of the variable is watched.

  Thread Every

  Frequency

  From 1

  To Infinity

  Every 1

It is possible that a watch condition is encountered within a program which you do not have sufficient authority to debug, or which does not contain debug data. In these circumstances, you receive a message specifying the program and location where the watch condition was encountered. Program execution resumes when you click on the message's **OK** button.

See “Characteristics of Watches” on page 204 for additional information on watches.

Writing Code That the Debugger Supports

Using VisualAge for C++ for AS/400, 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.

  Because individual statements cannot be accessed separately when you set breakpoints or when you use step commands, you may want to avoid the use of multiple statements on the same line.
- Although it is possible to create programs that contain more than one module with the same name (if the modules are in different libraries), the debugger does not support them.

  Attempting to debug programs that contain two or more modules with the same name will result in an error message being displayed.

Debugger Performance Considerations

In general, you should not have to worry about debugger performance, but if you find that the debugger is stepping slowly, you may want to consider the following points:

Expression evaluations

- Complex expressions take longer to evaluate than simple expressions.

  Performance is only an issue when you are monitoring an expression, since the expression must be evaluated each time the debugger stops.

- The settings of the **Default Data Representation** window affect the performance of expression evaluation:
  - Representing character pointers, arrays, and character arrays as hexadecimal pointers gives the best performance.
  - Representing structures using **System Defaults** performs better than using **User Defaults**.
  - Representing a character array as a string is faster than representing it as an array.
- Evaluating all of the elements of a large array takes longer than evaluating single elements. Use the **Monitor Expression** window to evaluate a single element.

Step performance

Step performance is affected by the number of enabled monitor windows, the numbers of expressions in the windows, and the complexity of the expression. Step performance can be improved by:

- Disabling or deleting expressions that no longer need to be monitored.
- Displaying only single elements of an array.
- After following a chain of pointers to a variable, disabling the pointers used and leaving only the variable active in the monitor.
- Not stepping with the Local Variables window enabled.
- Not stepping with the Call Stack window enabled. Minimizing the Call Stack window will disable it.

Expanding the procedures within a module

This function requires a lot of interaction with the AS/400. If a large number of procedures are in a module, searching for the procedure name by using the **Find Procedure** choice on the **File** menu is faster.

Using PC files instead of AS/400 source members

For non-C++ programs, performance can be improved by copying the files to the PC and using the **Change text file** choice from the **View** menu to specify the path name of the PC file. This technique is especially useful when debugging from remote sites.

Searching for a string in the text view

String searches can be speeded up by the following:

- Keeping the source file on the workstation.
- Using the **Find Procedure** choice to search for procedures
- Searching the ***LISTING** view instead of a source view that is on the AS/400.

Using the From/To/Every entry fields on line breakpoints

Large values specified for these options will significantly slow down your program, because the debugger must stop for the breakpoint and evaluate the From/To/Every clause each time. Even though you do not see the program stop, it is in fact stopping so that the debugger can evaluate the stop conditions.

If possible, an alternative is to set a conditional breakpoint by specifying an expression.

Setting a large number of watches

When a watch is set, the system checks after each instruction whether the value of the monitored variable or expression has changed. Setting many watches may lead to slower performance.

Debug Limits of the Cooperative Debugger

The following applies when you debug applications with the VisualAge for C++ for AS/400 cooperative debugger:

Limits

- The largest string that can be displayed is 4080 characters.
- The largest number of bytes that can be displayed in a hexadecimal dump is 1024.
- The largest number of elements displayed in a COBOL array or an RPG array is 500.
- The largest number of fields displayed in a COBOL record or an RPG structure is 500.
- Only 256 elements per dimension are returned for C and C++arrays.
- Only 140 characters per line are displayed for the text views retrieved from the AS/400. Files residing on the workstation will have the entire line displayed regardless of the line length.
- A maximum of 128 bytes can be watched for any variable.
- A maximum of 256 watches can be set system-wide. This number may be lower, because watching a variable can sometimes require more than one system watch.

Source files with a record size greater than 240

Source physical files that have a record size greater than 240 characters cause a message to be written to the job log for each record read. (The job log that the messages are written to is the job log of the debug server job that is serving your debug session.) This behavior slows down processing and may cause your debug session to end if the job log grows too large.

Chapter 14. Using the ILE Source Debugger

This section introduces the ILE source debugger you can use to debug ILE and OPM applications from AS/400.

See the section on debugging in *ILE Concepts* for more information on the ILE source debugger, including authority required to debug a program or service program and the effects of optimization levels.

Introducing the ILE Source Debugger

The Integrated Language Environment (ILE) source debugger is used to detect errors in and eliminate errors from programs and service programs

Using debug commands with any ILE or OPM program compiled with debug data, you can:

- View the program source or change the debug view
- Set and remove breakpoints and watch conditions
- Step through a specified number of statements
- Display the value of variables, expressions, structures, unions, records, arrays, pointers, bit fields, enumerations, and classes
- Change the value of variables
- Equate a shorthand name with a variable, expression, or debug command.

Before you can use the ILE source debugger, you must compile your source with one of the `/Ti+`, `/Til`, `/Tin`, or `/Tis` debug options on the **iccas** compiler invocation command.

After starting the source debugger, you can set breakpoints and call the program. When the program stops because of a breakpoint or a step command, the a module's view is shown on the display at the point where the program stopped. At this point, you can perform other actions such as displaying or changing the value of variables.

Note: If your program is optimized, you can still display variables, but their values may not be reliable. To ensure that the content of variables or data structures contain their correct (current) values, specify the `/O-` option with the **iccas** command when compiling your source file, to set the optimization level to none.

Avoiding Modification of Production Files

While debugging and testing your programs, ensure that your library list is changed to direct the programs to a test library containing test data so that any existing real data is not affected.

To prevent database files in production libraries from being modified unintentionally, do one of the following:

- Specify Start Debug (STRDBG) with the default **NO* for the *UPDPROD* parameter
- Use Change Debug (CHGDBG)
- Specify `SET UPPROD NO` in the Display Module Source display.

See the appendix on debugging in the *CL Reference* for more information on preventing the unintended modification of production files.

Debug Commands for the ILE Source Debugger

Debug commands and their parameters are entered on the debug command line displayed on the bottom of the Display Module Source and Evaluate Expression displays. They can be entered in uppercase, lowercase, or mixed case.

Note: The debug commands entered on the debug command line are not CL commands.

Command	Description
ATTR	Display the attributes of a variable. The attributes are the size and type of the variable as recorded in the debug symbol table.
BREAK	Set an unconditional or conditional breakpoint in the program being tested. Use BREAK *line-number* WHEN *expression* to enter a conditional breakpoint.
CLEAR	Remove conditional and unconditional breakpoints, or to remove one or all active watches.
DISPLAY	Display the names and definitions assigned by using the EQUATE command, and display a different source module than the one currently shown on the Display Module Source display. The module must exist in the current program.
EQUATE	Assign an expression, variable, or debug command to a name for shorthand use.
EVAL	Display or change the value of a variable or to display the value of expressions, records, structures, classes, arrays, pointers, bit fields, or enumerations.
FIND	Search for a specified string of text on the Display Module Source or Evaluate Expression display. A specified line number can also be found on the Display Module display.
QUAL	Define the scope of variables that appear in subsequent `EVAL` commands.
SET	Change debug options, such as the ability to update production files or to enable OPM source debug support.
STEP	Run one or more statements of the program being debugged.
WATCH	Request a breakpoint when the contents of a specified storage location is changed from its current value.

UP Move the displayed window showing the program source towards the beginning of the view by the amount entered.

DOWN Move the displayed window showing the program source towards the end of the view by the amount entered.

LEFT Move the displayed window showing the program source to the left by the number of characters entered.

RIGHT Move the displayed window showing the program source to the right by the number of characters entered.

TOP Position the view to show the first line.

BOTTOM Position the view to show the last line.

NEXT Position the view to the next breakpoint in the source currently displayed.

PREVIOUS Position the view to the previous breakpoint in the source currently displayed.

HELP Show the online help information for the available source debugger commands.

The online help for the ILE source debugger describes the debug commands, explains their allowed abbreviations, and provides syntax diagrams for each command. It also provides examples in each of the ILE languages of displaying and changing variables using the source debugger. You can access Help while in a debug session by pressing F1 (Help).

Debug Limits of the ILE Source Debugger

The ILE source debugger has the following limits:

- Function calls cannot be used in debug expressions. This is a limitation of the debug expression grammar.
- Precedence of operators and type conversion of mixed types conform to the C++ language.
- The maximum size of variables that can be displayed is 65535 characters:
  - With the `:c` and `:x` formatting overrides there are no exceptions.
  - With the `:s` formatting override, if no count is entered, the command stops after 30 bytes or a NULL, whichever is encountered first.
  - With the `:f` formatting override, if no count is entered, the command stops after 1024 bytes or a NULL, whichever is encountered first.
- The maximum number of classes that can be inherited as virtual base classes in a single compilation unit is 512.

Preparing a Program for Debugging

A program or module must have debug data available if you are to debug it. Since debug data is created during compilation, you specify one of the `/Ti` debug options on the **iccas** compiler invocation command, to indicate what type of data (if any) is to be created during compilation.

The type of debug data that can be associated with a module is referred to as a *debug view*. With the ILE source debugger you can create two views for each module you want to debug:

- A listing view
- A statement view.

Note: The default value for the debug option on **iccas** is `/Ti-`, which means that the module being created does not contain debug data. Specify this option if you do not want debug data to be included with the module, or if you want faster compilation time.

The storage requirements for a module or program vary somewhat, depending on the type of debug data included. The debug options are listed below. Secondary storage requirements increase as you work down the list:

1. `/Ti-` (No debug data)
2. `/Tin` (Statement view)
3. `/Til` (Listing view)
4. `/Ti+` (All views).

Once you have created a module with debug data and bound it into a program (*PGM), you can start debugging.

Note: You can specify the `/Tis` compiler option to create a source view, but in order to view the source you must use the cooperative debugger, because the C++ source resides on the Windows workstation.

Creating a Listing View

A **listing view** contains text similar to the text in the compiler listing produced by the compiler. You create a listing view to debug a module by using the `/Til` or `/Ti+` options for the **iccas** command when you create a module.

The compiler creates the listing view while the module (*MODULE) is being generated. The listing view is created by copying the text of the appropriate source members into the module. There is no dependency on the source members upon which it is based, once the listing view is created.

For example, to create a listing view to debug a program created from the source file `myfile.cpp`, type:

```
iccas /Til myfile.cpp
```

Note: The maximum line length for a listing view is 255 characters.

Creating a Statement View

A statement view allows the module to be debugged using debug commands. Create a statement view by using the `/Tin` or `/Ti+` option for the **iccas** command when you create a module. Use this view when:

- You have storage constraints, but do not want to recompile the module or program if you need to debug it.
- You are sending compiled objects to other users and want to be able to diagnose problems in your code, without these users being able to see your actual code.

To create a statement view to debug a program created from the source file `myfile.cpp`, type

```
iccas /Tin /L+ myfile.cpp
```

Debug the program using the debug commands.

Starting the ILE Source Debugger

Start the ILE source debugger using the Start Debug (STRDBG) command. Once the debugger is started, it remains active until you enter the End Debug (ENDDBG) command.

You must have **CHANGE* authority to a program object to include it in a debug session. Initially you can add up to 20 programs to a debug session by using the Program (*PGM*) parameter on the STRDBG command. They can be any combination of OPM or ILE programs. (Depending on how the OPM programs are compiled and on the debug environment settings, you may be able to debug them by using the ILE source debugger.)

The first program specified on the STRDBG command is shown, if it has debug data and, if it is an OPM program, the *OPMSRC* parameter is **YES*. If it is an ILE program, the entry module is shown, if it has debug data; otherwise, the first module bound to the ILE program that contains debug data is shown.

If an OPM program is in a debug session, you can use the ILE source debugger if the following conditions are met:

1. The OPM program is a CL, COBOL, or RPG program.
2. It was compiled with OPTION(*LSTDBG) or OPTION(*SRCDBG). COBOL and RPG programs can be compiled with **LSTDBG* or **SRCDBG*. CL programs must be compiled with **SRCDBG*.
3. The ILE debug environment can accept OPM programs. To activate this setting, specify OPMSRC(*YES) on the STRDBG command. (The system default is OPMSRC(*NO).)

To start a debug session for a program `DBGEXMP` that calls an OPM program `RPGPGM`, type:

```
STRDBG PGM(MYLIB/DBGEXMP MYLIB/RPGPGM) OPMSRC(*YES)
```

The Display Module Source display appears. If the entry module has a listing view, the display shows text similar to the compiler listing of the entry module of the first program. If the program DBGEXMP was compiled with the `/Ti+` option, the listing for the main module `DBGEXMP` appears.

Note: ILE service programs cannot be specified on the STRDBG command. You can add ILE service programs to a debug session by using option 1 (Add) on the Work with Module List display (F14), or by letting the debugger add it as part of a Step Into debug command.

See “Setting Debug Options” for additional information.

Setting Debug Options

After you start a debug session, you can set debug options using the SET command on the debug command line:

- You can indicate whether database files can be updated while debugging your program.

  This option corresponds to the *UPDPROD* parameter of the STRDBG command. This parameter specifies whether database files in a production library can be opened for updating records or for adding new records when the job is in debug mode. If not, the files must be copied into a test library before you try to run a program that uses the files.

- You can also indicate if `find` operations are to be case- sensitive or case- insensitive during a search request. When you start a debug session, the default is case insensitive searching.
- You can specify whether OPM programs are to be debugged using the ILE source debugger. (This option corresponds to the *OPMSRC* parameter.)

Changing the debug options using the SET debug command affects the value for the corresponding parameter, if any, specified on the STRDBG command. You can also use the Change Debug command (CHGDBG) to set debug options.

To set a debug option, type:

```
SET option value
```

on the debug command line, where *option* is an option that you want to set, and *value* identifies the value of the *option*.

- Specify *UPDPROD(*YES)* or *UPDPROD(*NO)* to indicate whether you want to update production files or not.
- Specify *CASE(*MATCH)* or *CASE(*IGNORE)* to define case- sensitive or case-insensitive searching.

The command:

```
SET UPDPROD YES
SET CASE MATCH
```

allows production files to be updated in debug mode, and `find` operations to be case sensitive.

Note: If you enter the SET debug command without parameters, the Set Debug Options display is shown.

Suppose you are in a debug session working with an ILE program and you decide you should also debug an OPM program that has debug data available. To enable the ILE source debugger to accept OPM programs, follow these steps:

1. After entering STRDBG, if the current display is not the Display Module Source display, type `DSPMODSRC`. The Display Module Source display appears.
2. Type `SET`. The Set Debug Options display appears.
3. Type Y (Yes) for the **OPM source debug support** field, and press Enter to return to the Display Module Source display.

You can now add the OPM program, either by using the Work with Module List display, or by processing a call statement to that program.

Adding and Removing Programs from a Debug Session

You can add more programs and remove programs to be debugged after starting a debug session. You must have **CHANGE* authority to a program to add or remove it from a debug session.

For ILE programs

Use option 1 (Add program) on the Work with Module List display entered by using F14 from the Display Module Source display.

To remove an ILE program or service program, use option 4 (Remove program) on the same display. When an ILE program or service program is removed, all breakpoints for that program are removed. There is no limit to the number of ILE programs or service programs that can be in a debug session at one time.

For OPM programs

You have two choices depending on the value specified for OPMSRC.

- If you specified OPMSRC(*YES), by using either STRDBG, the SET debug command, or CHGDBG, you add or remove an OPM program using the Work with Module List display. (A module name is not listed for an OPM program.) There is no limit to the number of OPM programs that can be included in a debug session when OPMSRC(*YES) is specified.
- If you specified OPMSRC(*NO), you must use the Add Program (ADDPGM) command or the Remove Program (RMVPGM) command. Only 20 OPM programs can be in a debug session at one time when OPMSRC(*NO) is specified.

Note: You cannot debug an OPM program with debug data from both an ILE and an OPM debug session. If an OPM program is already in an OPM debug session, you must first remove it from that session before adding it to the ILE debug session or stepping into it from a call statement. If you want to debug an OPM program from an OPM debug session, you must first remove it from the ILE debug session.

Adding a Service Program to a Debug Session

To add the service program `SERVEPIO` to the debug session which is already started, do the following.

1. If the current display is not the Display Module Source display, type `DSPMODSRC` and press Enter. The Display Module Source display appears.
2. Press F14 (Work with module list) to show the Work with Module List display.
3. To add service program `SERVEPIO`:
   a. Select menu item `1` (Add program)
   b. Enter `SERVEPIO` in the **Program/module** field
   c. Change the type field from **PGM* to **SRVPGM* and press Enter.
4. Press F12 (Cancel) to return to the Display Module Source display.

Removing a Program from a Debug Session

To remove the ILE program `CVTIOPGM` and the service program `SERVEPIO` from a debug session do the following:

1. If the current display is not the Display Module Source display, type `DSPMODSRC`. The Display Module Source display appears.
2. Press F14 (Work with module list) to show the Work with Module List display.
3. On this display, type `4` (Remove program) on the lines next to `CVTIOPGM` and `SERVEPIO`, and press Enter.
4. Press F12 (Cancel) to return to the Display Module Source display.

Viewing the Program Source

The Display Module Source display shows the source of an ILE program one module at a time. A module's source can be shown if the module was compiled using one of the following debug view compiler options:

- `/Til` (Listing view)
- `/Ti+` (All views).

The source of an OPM program can be shown if the following conditions are met:

1. The OPM program was compiled with OPTION(*LSTDBG) or OPTION(*SRCDBG). (Only RPG and COBOL programs can be compiled with (*LSTDBG).)

2. The ILE debug environment is set to accept OPM programs; That is the value of OPMSRC is (*YES). (The system default is OPMSRC(*NO).)

Note: When you compile your source files with the `/Ti+` compiler option, the resulting module object contains a source view. By default, the STRDBG command displays the source view. Instead, the listing view is shown, because source files are located on your Windows workstation, and the source view results in a blank debugger screen.

You can change what is shown on the Display Module Source display by changing to a different module.

When you change the module, the executable statement on the displayed view is stored in memory and is viewed when the module is displayed again. Line numbers that have breakpoints set are highlighted. When a breakpoint, step, or message causes the program to stop and the display to be shown, the statement where the breakpoint occurred is highlighted.

Use the Top, Bottom, Up, Left, Right, Next, Previous, and Find commands for positioning in the Display Module Source display of your module source. See “Debug Commands for the ILE Source Debugger” on page 192 for a description of these commands.

Viewing a Different Module

To change the module that is shown on the Display Module Source display, use option 5 (Display module source) on the Work with Module List display. You access the Work with Module List display from the Display Module Source display by pressing F14 (Work with Module List display).

If you use this option with an ILE or OPM program object, the entry module with a source or listing view is shown (if it exists). Otherwise the first module with debug data bound to the program is shown.

For example, to change from a module `MOD1` to a module `MOD2` using the `Display module source` option, follow these steps:

1. To work with modules, type `DSPMODSRC`, and press Enter. The Display Module Source display is shown.
2. Press F14 (Work with module list) to show the Work with Module List display.
3. To select `MOD2`, type 5 (Display module source) next to it and press Enter. If a listing view is available, it is shown. Otherwise, you see a blank Display Module Source display with the message "Source not available".

An alternate method of viewing a different module is to use the DISPLAY debug command. On the debug command line, type `DISPLAY MODULE` *`name`*. The module *name* is shown if the module exists in a program already added to the debug session.

Setting and Removing Breakpoints

You can use breakpoints to halt a program at a specific point when it is running.

- An *unconditional breakpoint* stops the program at a specific statement.
- A *conditional breakpoint* stops the program when a specific condition at a specific statement is met.

You set the breakpoints prior to running the program. When the program stops, the Display Module Source display is shown. The appropriate module is shown with the source positioned at the line where the breakpoint occurred. This line is highlighted. At this point, you can evaluate fields, set more breakpoints, and run any of the debug commands.

Consider the following characteristics of breakpoints:

- When a breakpoint is set on a statement, the breakpoint occurs before that statement is processed.
- When a statement with a conditional breakpoint is reached, the conditional expression associated with the breakpoint is evaluated before the statement is processed. If the expression is true, the breakpoint takes effect, and the program stops on that line.
- If the line on which you want to set a breakpoint is not an executable statement, the breakpoint is set on the next executable statement.
- A breakpoint is not processed when the corresponding statement is bypassed.
- Breakpoint functions are specified through debug commands. These functions include:
  - Adding breakpoints to programs
  - Removing breakpoints from programs
  - Displaying breakpoint information
  - Resuming the running of a program after a breakpoint has been reached.
- To set a breakpoint on the first statement of a multi-statement macro, the cursor should be on the line containing the macro invocation, not the macro expansion.

Setting and Removing Unconditional Breakpoints

You can set or remove an unconditional breakpoint by using:

- F6 (Add/Clear breakpoint) from the Display Module Source display
- F13 (Work with module breakpoints) from the Display Module Source display
- The BREAK debug command to set a breakpoint
- The CLEAR debug command to remove a breakpoint

The simplest way to set and remove an unconditional breakpoint is to use F6 (Add/Clear breakpoint). The function key acts as a toggle. If a breakpoint is already set

on the current line, pressing F6 removes it while the cursor is positioned on the current line.

To set or remove an unconditional breakpoint from the Display Module Source display press F13 (Work with module breakpoints). A list of options appear that allow you to set or remove breakpoints. If you select 4 (Clear), a breakpoint is removed from the line.

An alternate method of setting and removing unconditional breakpoints is to use the BREAK and CLEAR debug commands. To set an unconditional breakpoint using the BREAK debug command, type `BREAK line-number` on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module on which you want to set a breakpoint.

To remove an unconditional breakpoint using the CLEAR debug command, type:

```
CLEAR line-number
```

on the debug command line.

Setting an Unconditional Breakpoint

You can set an unconditional breakpoint using F6 (Add/Clear breakpoint).

To work with a module, type `DSPMODSRC` and press Enter. The Display Module Source display is shown.

If you want to set a breakpoint in a different module of the current program, type `DISPLAY MODULE name` on the debug command line where *name* is the name of the module that you want to display.

If you want to set the breakpoint in the module shown, place the cursor on a statement line number where you want to set the breakpoint, and press F6 (Add/Clear breakpoint):

- If there is no breakpoint on the line you specify, an unconditional breakpoint is set on that line.
- If there is a breakpoint on the line, it is removed.

After the breakpoint is set, press F3 (Exit) to leave the Display Module Source display. The breakpoint is not removed.

Call the program. When a breakpoint is reached, the program stops and the Display Module Source display is shown again, with the line containing the breakpoint highlighted. At this point, you can step through the program, resume processing, evaluate variables, or issue any other debug command.

Setting and Removing Conditional Breakpoints

You can set or remove a conditional breakpoint by using:

- The Work with Module Breakpoints display
- The BREAK debug command to set a breakpoint

- The CLEAR debug command to remove a breakpoint

Note: The relational operators supported for conditional breakpoints are <, >, ==, <=, >=, and !=.

Using the Work with Module Breakpoints Display

You access the Work with Module Breakpoints display from the Display Module Source display by pressing F13. The display provides you with a list of options that allow you to either add or remove conditional and unconditional breakpoints:

Making a Breakpoint Conditional

To make the breakpoint conditional, specify a conditional expression in the **Condition** field. If the line on which you want to set a breakpoint is not a executable statement, the breakpoint is set at the next executable statement.

Calling the Program

Once you have finished specifying all of the breakpoints, call the program. You can use F21 (Command Line) from the Display Module Source display to call the program from a command line or call the program after exiting from the display.

Evaluating a Conditional Expression

When a statement with a conditional breakpoint is reached, the conditional expression associated with the breakpoint is evaluated before the statement is run.

- If the result is false, the program continues to run.
- If the result is true, the program stops, and the Display Module Source display is shown.

At this point, you can evaluate fields, set more breakpoints, and run any of the debug commands.

Using the BREAK and CLEAR Debug Commands

An alternate method of setting and removing conditional breakpoints is to use the BREAK and CLEAR debug commands.

To set a conditional breakpoint using the BREAK debug command, type:

```
BREAK line-number WHEN expression
```

on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module on which you want to set a breakpoint. The variable *expression* is the conditional expression that is evaluated when the breakpoint is encountered.

To remove a conditional breakpoint using the CLEAR debug command, type:

```
CLEAR line-number
```

on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module from which you want to remove a breakpoint.

Setting a Conditional Breakpoint Using F13

To set a conditional breakpoint using F13 (Work with module breakpoints) do the following:

1. Press F13 (Work with module breakpoints) from the Display Module Source window. The Work with Module Breakpoints display is shown.
2. On this display, type 1 (Add) on the first line of the list to add a conditional breakpoint.
3. To set a conditional breakpoint at line 35 when the variable `i` is equal to 21, type 35 for the **Line** field, i==21 for the **Condition** field, and press Enter.

   A conditional breakpoint is set on line 35. The expression is evaluated before the statement is run. If the result is true (in the example, if i==21), the program stops, and the Display Module Source display is shown. If the result is false, the program continues to run.

   An existing breakpoint is always replaced by a new breakpoint entered at the same location.
4. After the breakpoint is set, press F12 (Cancel) to leave the Work with Module Breakpoints display. Press F3 (End Program) to leave the ILE source debugger. Your breakpoint is not removed.
5. Call the program. When the conditional breakpoint is reached and the condition is true, the program stops, and the Display Module Source display is shown again. At this point, you can step through the program or resume processing.

Setting a Conditional Breakpoint Using the BREAK Command

Assume you want to stop the program when the variable `j` has a certain value. To specify the conditional breakpoint using the BREAK command do the following:

1. From the Display Module Source display, enter `break 56 when j==5` to set a conditional breakpoint on line 56.
2. After the breakpoint is set, press F3 (End Program) to leave the ILE source debugger. Your breakpoint is not removed.
3. Call the program. When a breakpoint is reached, the program stops, and the Display Module Source display is shown again.

Removing All Breakpoints

You can remove all breakpoints, conditional and unconditional, from a program that has a module shown on the Display Module Source display by using the `CLEAR PGM` debug command. To use the debug command, type `CLEAR PGM` on the debug command line. The breakpoints are removed from all of the modules bound to the program.

Setting and Removing Watch Conditions

You use a *watch condition* to monitor if the current value of an expression or a variable changes while your program runs. In order for an expression to be watched, it must resolve to a value that can be assigned to.

Setting watch conditions is similar to setting conditional breakpoints, with one difference: Watch conditions stop the program as soon as the value of a watched expression or variable changes from its current value. Conditional breakpoints stop the program only if a variable changes to the value specified in the condition.

The debugger watches an expression or a variable through the content of a *storage address*, computed at the time the watch condition is set. When the content at the storage address changes from the value it had when the watch condition was set or when the last watch condition occurred, a breakpoint is set, and the program stops.

Note: After a watch condition has been registered, the new content at the watched storage is saved as the new current value of the corresponding expression or variable. The next watch condition will be registered if the new contents at the watched storage location change subsequently.

Characteristics of Watches

Consider the following characteristics of watches:

- Watches are monitored on a system-wide basis, with a maximum number of 256 watches that can be active simultaneously. This number includes watches set by the system. If an expression or a variable crosses a page boundary, two watches are used internally to monitor the storage locations. The maximum number of expressions or variables that can be watched simultaneously on a system-wide basis ranges from 128 to 256.

  Depending on overall system use, you may be limited in the number of watch conditions you can set at a given time. If you try to set a watch condition while the maximum number of active watches across the system is exceeded, you will receive an error message and the watch condition is not set.

- Watch conditions can only be set when a program is stopped under debug, and the expression or variable to be watched is in scope. If this is not the case, an error message is issued when a watch is requested, indicating that the corresponding call stack entry does not exist.

- Once the watch condition is set, the address of a storage location watched does not change. If a watch is set on a temporary location, such as the automatic storage of an ILE C or C++ procedure, which can be reused after the procedure ends, it may result in invalid watch-condition notifications.

  A watch condition may be registered although the watched variable is no longer in scope. You must not assume that a variable is in scope just because a watch condition is reported.

- Two watch locations in the same job must not overlay in any way. Two watch locations in different jobs must not start at the same storage address; otherwise, overlap is allowed. If these restrictions are violated, an error message is issued.

  Note: Changes made to a watched storage location are ignored if they are made by a job other than the one that set the watch condition.

- After the command is successfully run, your application is stopped if a program in your session changes the contents of the watched storage location, and the **Display Module Source** display is shown.

  If the program has debug data, and a source text view is available, it is shown. The source line of the statement that is about to be run when the content change at the storage location is detected is highlighted. A message indicates which watch condition is satisfied.

  If the program cannot be debugged, the text area of the display remains blank.
- Eligible programs are automatically added to the debug session if they cause the watch stop.
- You can also set watch conditions when you are using service jobs for debugging (debugging one job from another job).

Setting Watch Conditions

Before you can set a watch condition, your program must be stopped under debug, and the expression or variable you want to watch must be in scope:

- To watch a global variable, you must ensure that the program in which the variable is defined is active before setting the watch condition.
- To watch a local variable, you must step into the function in which the variable is defined before setting the watch condition.

You can set a watch condition by using:

- F18 (Work with Watch) to show the Work with Watch display from which you can clear or display watches.
- F17 (Watch Variable) to set a watch condition for a variable on which the cursor is positioned.
- The WATCH debug command.

Example of Setting a Watch Condition

To watch a variable `salary` in program `MYLIB/PAYROLL`, you set the watch condition by typing `WATCH salary` on a debug line, accepting the default value for the watch-length.

If the value of the variable `salary` changes subsequently, the application stops and the Display Module Source display shown in Figure 24 on page 206 displays.

```
                              Display Module Source
Program:   PAYROL          Library:   MYLIB       Module:  PAYROLL
    52  for  (cnt=0;
    53        cnt<EMPMAX &&;
    54          scanf("%s%s%f%d%d", payptr->first, payptr->last,
    55                 &(payptr->wage), &eflag, &(payptr->hrs))!=EOF;
    56        cnt++, payptr++)
    57  {
    58     payptr->exempt=eflag;
    59  }
    60  empsort(payfile, cnt);
    61  for  (index=1, payptr=payfile; index<=cnt; index++,payptr++) {
    62    if  (payptr->exempt==1) {
    63        salary = 40*(payptr->wage);
    64        numexempt++;  }
    65    else
    66        salary = (payptr->hours)*(payptr->wage);
                                                                    More...
Debug . . .  ___________________________________________________________
_______________________________________________________________________
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable   F18=Work with watch   F24=More keys
Watch number  1 at line  64, variable: salary
```

Figure 24. Display Module Source Display After a Watch Condition Is Encountered

1. The line number of the statement where the change to the watch variable was detected is highlighted. This is the first executable line *following* the statement that changed the variable.
2. A message indicates that the watch condition is satisfied.

   Note: If a text view is not available, a blank Display Module Source display appears, as shown in Figure 25 on page 207, with a message that the watch condition was satisfied.

   The following programs cannot be added to the ILE debug environment:

   - ILE programs without debug data
   - OPM programs with non-source debug data only
   - OPM programs without debug data.

   In the first two cases, the stopped statement number is passed. In the third case, the stopped MI instruction is passed. In all cases, the information is displayed at the bottom of a blank Display Module Source display as shown in Figure 25 on page 207. Instead of the line number, the statement or the instruction number is given.

```
                              Display Module Source

(Source not available)









F3=End program  F12=Resume  F14=Work with module list  F18 Work with watch
F21=Command entry  F22=Step into  F23=Display output
Watch number  1 at instruction 18,  variable: salary
```

Figure 25. Example of a Display Module Source when Text View Is Not Available

Using the WATCH Command

If you use the WATCH command, it must be entered as a single command; no other debug commands are allowed on the same command line. To access the Work with Watch display shown below, type `WATCH` on the debug command line, without any parameters.

```
                              Work with Watch

                                                  System:  DEBUGGER
Type options, press Enter.
  4=Clear 5=Display

Opt    Num      Variable                 Address            Length
 -     1        salary                   080090506F027004   4









                                                                Bottom
Command
===>_____________________________________________________________________
F3=Exit  F4=Prompt  F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 26. Example of a Work with Watch Display

The Work with Watch display shows all watches currently active in the debug session. You can clear and display watches from this display. When you select Option 5 `Display`, the Display Watch window displays information about the currently active watch:

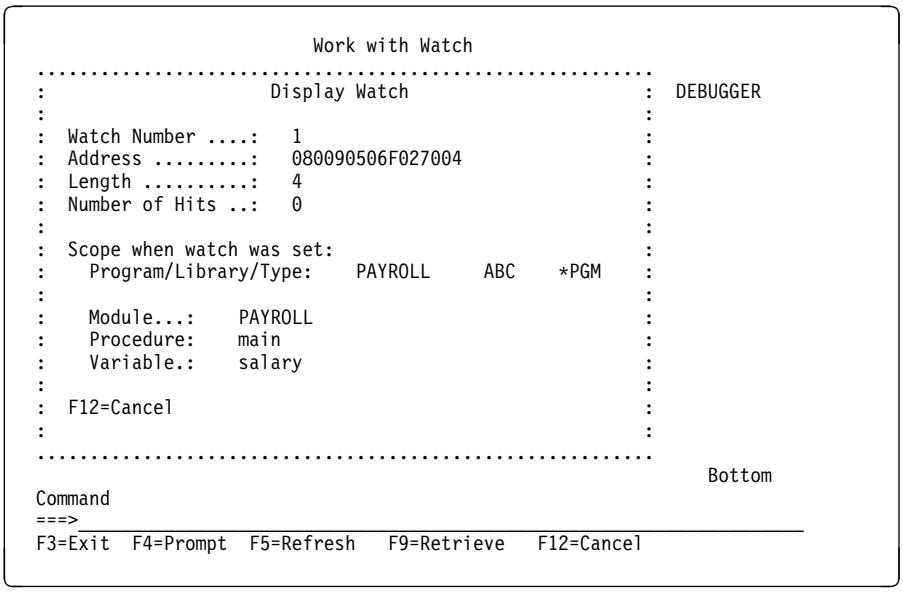


Figure 27. Example of a Display Watch Window

- To specify a variable `var`to be watched, type `WATCH` *`var`* on the debug command line.

  This command requests a breakpoint to be set if the value of `var` is changed from its current value.

- To specify an expression to be watched, type `WATCH` *`expression`* on the debug command line.

  This command requests a breakpoint to be set if the value of `expression` is changed from its current value.

  Note: *expression* is used to determine the address of the storage location to watch and must resolve to a location that can be assigned to, such as `(p+2)`, where `p` is a pointer. The scope of the expression variables in a watch is defined by the most recently issued QUAL command.

- To set a watch condition and specify a watch length, type:

  `WATCH` *`expression`* `:` *`watch length`*

  on a debug command line.

Each watch allows you to monitor and compare a maximum of 128 bytes of contiguous storage. If the maximum length of 128 bytes is exceeded, the watch will not be set, and the debugger issues an error message.

By default, the length of the expression type is also the length of the watch-comparison operation. The *watch length* parameter overrides this default. It determines the number of bytes of an expression to be compared to determine if a change in value occurred.

- If a 4-byte binary integer is specified as the variable, without the watch-length parameter, the comparison length is 4 bytes. However, if the watch-length parameter is specified, it overrides the length of the expression in determining the watch length.

Displaying Active Watches

To display a system-wide list of active watches and show which job set them, type: `DSPDBGWCH` on a CL command line. This command brings up the following Display Debug Watches display:

```
                          Display Debug Watches

                                                      System:  DEBUGGER
------------Job---------------    NUM     LENGTH    ADDRESS
MYJOBNAME1  MYUSERPRF1  123456      1       4       080090506F027004
JOB4567890  PRF4567890  222222      1       4       09849403845A2C32
JOB4567890  PRF4567890  222222      2       4       098494038456AA00
JOB         PROFILE     333333     14       4       040689578309AF09
SOMEJOB     SOMEPROFIL  444444      3       4       005498348048242A









                                                                  More...
Press Enter to continue

F3=Exit  F5=Refresh   F12=Cancel
```

Figure 28. Display Debug Watches Display

Note: This display does not show watch conditions set by the system.

Removing Watch Conditions

Watches can be removed in the following ways:

- The CLEAR command used with the WATCH keyword selectively ends one or all watches. To clear the watch identified by `watch-number`, type:

  `CLEAR WATCH` *`watch-number`*

  The watch number can be obtained from the Work with Watches display.

To clear all watches for your session, type `CLEAR WATCH ALL` on a debug command line.

Note: While the CLEAR PGM command removes all breakpoints in the program that contains the module being displayed, it has no effect on watches. You must explicitly use the WATCH keyword with the CLEAR command to remove watch conditions.

- The CL End Debug (ENDDBG) command removes watches set in the local job or in a service job.

  Note: ENDDBG is called automatically in abnormal situations to ensure that all affected watches are removed.

- The initial program load (IPL) of your AS/400 system removes all watch conditions system-wide.

Stepping through the Program

After a breakpoint is encountered, you can run a specified number of statements of the program, then stop the program again and return to the Display Module Source display. You do this by using the step function of the ILE source debugger. The program resumes running on the next statement of the module in which the program stopped. A breakpoint is used to stop a program.

You can step *into* an OPM program if it has debug data available and if the debug session accepts OPM programs for debugging. You can step *through* a program by using:

- F10 (Step) or F22 (Step into) on the Display Module Source display
- The STEP debug command.

One way to step through a program one statement at a time is to use F10 (Step) or F22 (Step into) on the Display Module Source display. When you press F10 or F22, the next statement of the module shown in the Display Module Source display is run, and the program is stopped again.

Note: You cannot specify the number of statements to step through when you use F10 (Step) or F22 (Step into). Pressing F10 or F22 performs a single step.

Another way to step through a program is to use the STEP debug command. The STEP debug command allows you to run more than one statement in a single step. The default number of statements to run is one. To use the STEP debug command, type `STEP number-of-statements` on the debug command line.

The variable *number-of-statements* is the number of statements of the program that you want to run in the next step before the program is halted again. If you type `STEP 5` on the debug command line, the next five statements of your program are run, the program is stopped again and the Display Module Source display is shown.

When a CALL statement to another program or procedure is encountered in a debug session, you can do one of the following:

- Step over the called program or procedure
- Step into the called program or procedure.

If you choose to *step over* the called program, the CALL statement and the called program are run as a single step. The called program is run to completion before the calling program is stopped at the next step. Step over is the default step mode.

If you choose to *step into* the called program, each statement in the called program is run as a single step. If the next step at which the running program is to stop falls within the called program, the called program is halted at this point and the called program is shown in the Display Module Source display.

Stepping over Programs

You can step over programs by using:

- F10 (Step) on the Display Module Source display
- The STEP OVER debug command.

Using F10 to Step over Programs

Use F10 (Step) on the Display Module Source display to step over a called program in a debug session. If the next statement to be run is a CALL statement to another program, pressing F10 (Step) causes the called program to run to completion before the calling program is stopped again.

Using the STEP OVER Debug Command

Use the STEP OVER debug command to step over a called program in a debug session. To use the STEP OVER debug command, type `STEP` *`number-of-statements`* `OVER` on the debug command line. The variable *number-of-statements* is the number of statements of the program that you want to run in the next step before the program is halted again.

If this variable is omitted, the default is 1. If one of the statements that are run contains a call to another program, the ILE source debugger steps over the called program.

Stepping into Programs

Step into programs by using:

- F22 (Step into) on the Display Module Source display
- The STEP INTO debug command.

Using F22 to Step into Programs

Use F22 (Step into) on the Display Module Source display to step into a called program in a debug session. If the next statement to be run is a CALL statement to another program, pressing F22 causes the first executable statement in the called program to be run. The called program is then shown in the Display Module Source display.

Note: The called program must have debug data associated with it in order for it to be shown in the Display Module Source display.

Using the STEP INTO Debug Command

Use the STEP INTO debug command to step into a called program in a debug session. To use the STEP INTO debug command, type:

```
STEP number-of-statements INTO
```

on the debug command line. The variable *number-of-statements* is the number of statements of the program that you want to run in the next step before the program is halted again. If this variable is omitted, the default is 1.

Stepping into Called Programs

If one of the statements that are run contains a CALL statement to another program, the source debugger steps into the called program. Each statement in the called program is counted in the step. If the step ends in the called program, the called program is shown in the Display Module Source display. For example, if you type `STEP 5 INTO` on the debug command line, the next five statements of the program are run. If the third statement is a CALL statement to another program, two statements of the calling program are run and the first three statements of the called program are run.

The STEP INTO command works with the CL CALL command as well. You can take advantage of this to step through your program after calling it. After starting the source debugger, from the initial Display Module Source display, enter `STEP 1 INTO`. This sets the step count to 1. Use the F12 key to return to the command line and then call the program. The program stops at the first statement with debug data.

Stepping Into a Program Using F22

Use F22 (Step Into) to step into program `CPGM` from the program `DEBUGEX`.

1. Assume that the Display Module Source Display shows the source for `DEBUGEX`
2. To set an unconditional breakpoint at line 92, which is the last executable statement before the call to function `CalcTax()` in program `CPPPGM`, type `Break` and press Enter.
3. Press F3 (End Program) to leave the Display Module Source display.
4. Call the program. The program stops at breakpoint 92, as shown in Figure 29 on page 213.

DEBUGEX Before Stepping Into CPGM

```
                          Display Module Source
Program:   DEBUGEX        Library:   MYLIB          Module:   DEBUGEX
    88          cout << "Please enter amount" << endl;
    89          cin >> input;
    90          if (input > MINIMUM) {
    91          // call function CalcTax in separate program CPPPGM
    92             retval1 = CalcTax(input);
    93             if (retval1 > LIMIT)
    94                retval2 = CalcSurtax(input)
    95          }
    96          cout << "Total tax is " << retval1 = retval2 << endl;
    97       }
    98
    99
   100
   101
   102
                                                                    More...
Debug . . .
F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable    F18=Work with watch   F24=More keys
Breakpoint at line 90
```

Figure 29. Module Source Display for DEBUGEX

5. Press F22 (Step into). One statement of the program runs, and then the Display Module Source display of CPGM is shown, as in Figure 30 on page 214.

   The first executable statement of CPGM is processed (line 13) and then the program stops.

   Note: You cannot specify the number of statements to step through when you use F22. Pressing F22 performs a single step.

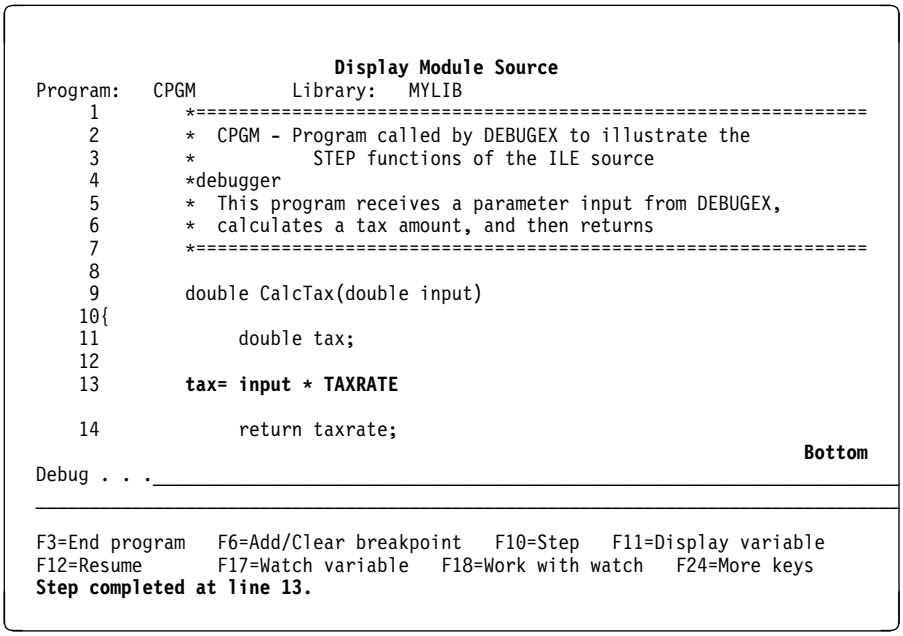


Figure 30. Module Source Display After Stepping Into CPGM

If there is no debug data available, you see a blank Display Module Source display with a message indicating that the source is not available.

Stepping Over Procedures

If you specify *over* on the STEP debug command, calls to procedures, and functions count as single statements. This is the default step mode. You can start the step-over function by using:

- The STEP OVER debug command
- F10 (Step).

Stepping Into Procedures

If you specify *into* on the STEP debug command, each statement in a procedure or function that is called counts as a single statement. Stepping through four statements of a program may result in running 20 statements if one of the four is a call to a procedure with 16 statements. You can start the step into function by using:

- The STEP INTO debug command
- F22 (Step into).

The called procedure must have debug data associated with it to be shown in the Display Module Source display.

You can automatically put a program or service program into debug. This happens if the program or service program:

- Has debug data
- Is not in debug
- Contains a procedure that is stepped into from another program in debug.

The program or service program is added to debug for you and the Display Module Source display shows the procedure in the program or service program. From this point, modules in the program or service program can be accessed using the Work with Modules List display just like modules in programs you added to debug.

Displaying the Value of Variables and Expressions

You can display the value of scalar variables, expressions, records, structures, classes, arrays, pointers, enumerations, bit fields, unions, and functions using the EVAL debug command.

There are two ways to display or evaluate:

- F11 (Display variable)
- The EVAL debug command.

The scope of the variables used in the EVAL debug command is defined by using the QUAL debug command.

Using F11 to Display Variables

The easiest way to display data or an expression is to use F11 (Display variable) on the Display Module Source display. Place your cursor on the variable that you want to display and press F11. The current value of the variable is shown on the message line at the bottom of the Display Module Source display.

In cases where you are evaluating structures, records, classes, arrays, pointers, enumerations, bit fields, unions or functions, the message returned when you press F11 (Display variable) may span several lines. Messages that span several lines are shown on the Evaluate Expression display to show the entire text of the message. Once you have finished viewing the message on the Evaluate Expression display, press Enter to return to the Display Module Source display.

The Evaluate Expression display also shows all the past debug commands that you entered and the results from these commands. You can search forward or backward on the Evaluate Expression display for a specified string, or text and retrieve or reissue debug commands.

Using the EVAL Debug Command to Display Variables

To display data using the EVAL debug command, type EVAL *variable* on the debug command line. *Variable* is the name of the variable, expression, data structure, class, array, pointer, bit-field, union, function or array that you want to display or evaluate. The

value is shown on the message line if the EVAL debug command is entered from the Display Module Source display and the value can be shown on a single line. Otherwise, it is shown on the Evaluate Expression display.

Figure 31 shows the result of using the debug command:

```
EVAL hold_formatted_cost
```

to display the contents of a variable `hold_formatted_cost`.

```
                            Display Module Source
 Program:   CPPPG1         Library:   CURLIB
    Module:   CPPMOD2
     47      if (j<0) return(0);
     48      if (hold_formatted_cost[i] == '$')
     49      {
     50        formatted_cost[j] = hold_formatted_cost[i];
     51        break;
     52      }
     53      if (i<16 &&; !((i-2)%3))
     54      {
     55        formatted_cost[j] = ',';
     56        --j;
     57      }
     58      formatted_cost[j] = hold_formatted_cost[i];
     59      --j;
     60    }
     61

                   More...
 Debug . . .   _____________________________________________________________
 ______________________________________________________________________________

 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F17=Watch variable     F18=Work with watch  F24=More Keys
hold_formatted_cost = SPP:0000C048BD0003F0
```

Figure 31. Displaying a Variable Using the EVAL Debug Command

Use the EVAL debug command to determine the value of an expression. If variable `j` has a value of 1024 and you type `eval (j * j)/512` on the debug command line, the message line shows `(j * j)/512 = 2048`.

Sample EVAL Commands for C Pointers, Variables, and Bit Fields

Figure 32 on page 217 shows the use of the EVAL command with pointers, variables, and bit fields. The pointers, variables, and bit fields are based on the source in “Sample Source for EVAL Commands” on page 228.

```
 Pointers
// Display a pointer
>eval pc1
 pc1 = SPP:0000C0260900107C

// Assign a value to a pointer
>eval pc2=pc1
 pc2=pc1 = SPP:0000C0260900107C

// Dereference a pointer
>eval *pc1
 *pc1 = 'C'

// Take the address of a pointer
>eval &pc1
 &pc1 = SPP:0000C02609001040

// Build an expression with normal C precedence
>eval *&pc1
 *&pc1 = SPP:0000C0260900107C

// Casting a pointer
>eval *(short *)pc1
 *(short *)pc1 = -15616

// Treat an unqualified array as a pointer
>eval arr1
 arr1 = SPP:0000C02609001070

// Apply the array type through dereferencing
// (character in this example)
>eval *arr1
 *arr1 = 'A'

// Override the formatting of an expression that is an lvalue
>eval *arr1:s
 *arr1:s = "ABC"
```

Figure 32 (Part 1 of 3). Sample EVAL Commands for C Pointers, Variables, and Bit Fields

```
// Set a pointer to null by assigning 0
>eval pc1=0
 pc1=0 = SYP:*NULL

// Evaluate a function pointer
>eval fncptr
 fncptr = PRP:0000A0CD0004F010

// Use the arrow operator
>eval *pY->x.p
 *pY->x.p = ' '
```

Simple Variables

```
// Perform logical operations
>eval i1==u1 || i1<u1
 i1==u1 || i1<u1 = 0

// Unary operators occur in proper order
>eval i1++
 i1++ = 100

// i1 is incremented after being used
>eval i1
 i1 = 101

// i1 is incremented before being used
>eval ++i1
 ++i1 = 102

// Implicit conversion
>eval u1 = -10
 u1 = -10 = 4294967286

// Implicit conversion
>eval (int)u1
 (int)u1 = -10
```

Figure 32 (Part 2 of 3). Sample EVAL Commands for C Pointers, Variables, and Bit Fields

```
Bit Fields
// Display an entire structure
>eval bits
 bits.b1 = 1
 bits.b4 = 2

// Work with a single member of a structure
>eval bits.b4 = bits.b1
 bits.b4 = bits.b1 = 1

// Bit fields are fully supported
>eval bits.b1 << 2
 bits.b1 << 2 = 4

// You can overflow bit fields, but no warning is generated
>eval bits.b1 = bits.b1 << 2
 bits.b1 = bits.b1 << 2 = 4
>eval bits.b1
 bits.b1 = 0
```

Figure 32 (Part 3 of 3). Sample EVAL Commands for C Pointers, Variables, and Bit Fields

Figure 33 on page 220 shows the use of the EVAL command with structures, unions, and enumerations. The structures, unions, and enumerations are based on the source in "Sample Source for EVAL Commands" on page 228.

```
 Structures and Unions
// Cast with typedefs
>eval (struct z *)&zz
 (struct z *)&zz = SPP:0000C005AA0010D0

// Cast with tags
>eval *(c *)&zz
 (*(c *)&zz).a = 1
 (*(c *)&zz).b = SYP:*NULL

// Assign union members
>eval u.x = -10
 u.x = -10 = -10

// Display a union. The union is formatted for each definition
>eval u
 u.y = 4294967286
 u.x = -10
 Enumerations
// Display both the enumeration and its value
>eval Color
 Color = blue (2)
>eval Number
 Number = three (2)

// Cast to a different enumeration
>eval (enum color)Number
 (enum color)Number = blue (2)

// Assign by number
>eval Number = 1
 Number = 1 = two (1)

// Assign by enumeration
>eval Number = three
 Number = three = three (2)

// Use enums in an expression
>eval arr1[one]
 arr1[one] = 'A'
```

Figure 33. Sample EVAL Commands for C Structures, Unions and Enumerations

EVAL Commands for System and Space Pointers

Figure 34 on page 221 shows the use of the EVAL command with system and space pointers. The system and space pointers are based on the source in “Sample Source for Displaying System and Space Pointers” on page 230.

```
 System and Space Pointers
// System pointers are formatted
// :1934:QTEMP     :111111110
>eval pSYSptr
 pSYSptr =
           SYP:QTEUSERSPC
           0011100
// Space pointers return 8 bytes that can be used in
// System Service Tools
>eval pBuffer
 pBuffer = SPP:0000071ECD000200
```

Figure 34. Sample EVAL Commands for System and Space Pointers

You can use the EVAL command on C++ language features and constructs. The ILE source debugger can display a full class or structure but only with those fields defined in the derived class. You can display a base class in full by casting the derived class to the particular base class.

Figure 35 on page 222 shows the use of the EVAL command with C++ language constructs. The C++ language constructs are based on the source in "Sample Source for Displaying C++ Constructs" on page 231. Additional C++ examples are provided in the source debugger online help.

```
// Follow the class hierarchy (specifying class D is optional)
> EVAL *(class D *)this
  (*(class D *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(class D *)this).__vbp1D = SPP:C40F5E3D7F000440
  (*(class D *)this).d = 4

// Follow the class hierarchy (without specifying class D)
> EVAL *(D *)this
  (*(D *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(D *)this).__vbp1D = SPP:C40F5E3D7F000440
  (*(D *)this).d = 4

// Look at a local variable
> EVAL VAR
  VAR = 1

// Look at a global variable
> EVAL ::VAR
  ::VAR = 2

// Look at a class member (specifying this-> is optional)
> EVAL this->f
  this->f = 6

// Look at a class member (without specifying this->)
> EVAL f
  f = 6

// Disambiguate variable ac
> EVAL A::ac
  A::ac = 12

// Scope operator with template
> EVAL E<int>::ac
  E<int>::ac = 12

// Cast with template:
> EVAL *(E<int> *)this
  (*(E<int> *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(E<int> *)this).__vbp1EXTi_ = SPP:C40F5E3D7F000400
  (*(E<int> *)this).e = 5

// Assign a value to a variable
> EVAL f=23
  f=23 = 23

// See all local variables in a single EVAL statement
> EVAL %LOCALVARS
  local = 828
  this = SPP:C40F5E3D7F000400
  VAR = 1
```

Figure 35. Sample EVAL Commands for C++ Expressions

Displaying Data Structures

You display the contents of a data structure like any stand-alone variable. Use the data structure name after EVAL to see the entire contents.

Figure 36 shows the entire structure with two elements being displayed. Each element of the structure is formatted according to its type and displayed.

```
> EVAL test
  test.charValue = 'c'
  test.intValue = 10
 1  struct {               //Code evaluated at line 9 where a
 2    char charValue       //breakpoint was set.
 3    unsigned long intValue;
 4  } test;
 5
 6  main(){
 7    test.intValue = 10;
 8    test.charValue = 'c';
 9    test.charValue = 11;
10  }
```

Figure 36. Using EVAL with Data Structures

Displaying Variables as Hexadecimal Values

To display the value of a variable in hexadecimal format, type `EVAL` *`variable-name: x number-of-bytes`* on the debug command line, where *variable-name* is the name of the variable that you want to display in hexadecimal format, `'x'` specifies that the variable is to be display in hexadecimal format, and *number-of-bytes* indicates the number of bytes displayed.

If no length is specified after the `'x'`, the size of the variable is used as the length. A minimum of 16 bytes is always displayed. If the length of the variable is less than 16 bytes, the remaining space is filled with zeros until the 16 byte boundary is reached.

The variable `test` is defined as an eight-byte character variable. The EVAL command result length is specified as 32 bytes:

```
>  EVAL test: x 32
   00000     83000000 0000000A 00000000 00000000  - c......
   00010     00000000 00000000 00000000 00000000  - .......
 1  struct {          //Code evaluated at line 9 where a
 2    char charValue       //breakpoint was set.
 3    unsigned long intValue;
 4  } test;
 5
 6  main(){
 7    test.intValue = 10;
 8    test.charValue = 'c';
 9    test.charValue = 11;
10  }
```

Figure 37. Using EVAL with Hexadecimal Values

The result shows 32 bytes are formatted, although only the first 8 are meaningful. The left column is an offset in hexadecimal from the start of the variable. The right column is an EBCDIC character representation of the data.

Displaying Arrays in Character Format

To display an array in character format, type:

```
EVAL array-name:
c number-of-characters
```

on the debug command line. The variable *array-name* is the name of the array that you want to display in character format, `'c'` specifies the number of characters to display.

In Figure 38 the array must be dereferenced by the * operator. If the * operator is not entered, the array is displayed as a space pointer. If the dereferencing operator is used, but `':c'` is not appended to the expression, only the first array element is displayed. The default length of the display is 1.

```
> EVAL *array: c 11
  *array1: c 11 = '0123456789 '
1  #include <string.h>         //Code evaluated at line 6
2  char array1 [11]; //where a breakpoint was set
3  int i;
4  main(){
5    strcpy(array1,"0123456789");
6    i = 0;
7  }
```

Figure 38. Using EVAL with Arrays

The result illustrates displaying 11 characters, including a NULL character. The NULL character appears as a blank.

Displaying NULL Terminated Arrays (Strings) in Character Format

To display a NULL terminated array (string) in character format, type:

```
EVAL array-name: s length of string
```

on the debug command line. The variable `array-name` is the name of the array that you want to display in character format, and `'s'` specifies the length of the string to display.

In Figure 39 on page 225 the array must be dereferenced by the * operator. If the * operator is not entered, the array is displayed as a space pointer. If the dereferencing operator is used, but `':s'` is not appended to the expression, only the first array element is displayed.

```
> EVAL *array: s
  *array1: s = "0123456789 "
1  #include <string.h>         //Code evaluated at line 6
2  char array1[11]; //where a breakpoint was set
3  int i;
4  main() {
5    strcpy(array1,"0123456789");
6    i = 0;
7  }
```

Figure 39. Using EVAL with NULL Terminated Arrays

A string length of up to 65535 can follow the `s` character. Formatting stops at the first NULL character encountered. If no length is specified, formatting stops after 30 characters or the first NULL character, whichever is less.

To display a NULL terminated array (string) in multi-line character format, type:

`EVAL` *`array-name: f`*

on the debug command line. The variable `array-name` is the name of the array that you want to display in character format, and `'f'` specifies that the newline character (`x'15'`) is scanned while displaying the string output. When a newline character is found, the characters following the newline character are displayed at the beginning of the next line. If the end of the display line occurs the output is wrapped to the next display line.

Figure 40 shows how text is wrapped to the next line when you specify `s:` after the *array-name* in the EVAL debug command. When you specify `f:` after the *array-name* the characters following the newline character are displayed at the beginning of the next line.

```
> EVAL *testc: s 100
   *testc: s 100 =
       1 "This is the 1st line. This is the 2nd line. This"
       61 "is the 3rd line."
> EVAL *testc: f 100
   *testc: f 100 =
                           "This is the 1st line."
                           "This is the 2nd line."
                           "This is the 3rd line."
 1  #include <string.h>         //Code evaluated at line 6
                                 //where a breakpoint was set
 2  int main(){
 3    char testc[]= {"This is the 1st line."
 4                               "\nThis is the 2nd line."
 5                               "\nThis is the third line."};
 6    int i;
 7    i = 1;
 8  }
```

Figure 40. Using EVAL with NULL Terminated Arrays in Multi-Line Character Format

Displaying a Template Class and a Template Function

To display a template class or a template function, type `EVAL template-name` on the debug command line. The variable *template-name* is the name of the template class or template function you want to display.

Figure 41 shows the results of evaluating a template class. You must enter a template name that matches the demangled template name. Typedef names are not valid because the typedef information is removed when the template name is mangled.

```
> EVAL XX<int>::a
  XX<int>::= '1 '
> EVAL XX<inttype>::a
  Identifier not found
1  template < class A >        //Code evaluated at line 8
2  class XX {                  //where a breakpoint was set
3     static A a;
4     static B b;
5  };
6  XX<int> x;
7  typedef int inttype;
8  int XX<int>::a =1;          //mangled name a__2XXXTi_
9  int XX<inttype>::b = 2;     //mangled name b__2XXXTi_
```

Figure 41. Using EVAL with a Class Template

Figure 42 shows the results of evaluating a template function.

```
> EVAL XX<int,12>::sxa
  XX<int,12>::sxa = '1 '
> EVAL xxobj.xca[0]
  xxobj.xca[0] = '2 '
1  template < class A, int B>  //Code evaluated at lines 8 and 9
2  class XX {                  //where breakpoints were set
3     static A sxa;
4     char     xca[B];
5  public:
6    XX(void) { xca[0] = 2; }
7  };
8  XX<int,12> xxobj;
9  int XX<int,2*6>::sxa =1;
                               //same as intXX<int,12>::sxa
                               //mangled name sxa__2XXXTiSP12_
```

Figure 42. Using EVAL with a Function Template

Changing the Value of Scalar Variables

Change the value of scalar variables using the EVAL debug command with an assignment operator (=). The program must be called and stopped at a breakpoint or step location to change the value. To change the value of a variable, type:

`EVAL variable-name = value`

on the debug command line. `variable-name` is the name of the variable that you want to change and `value` is an identifier, literal, or constant value that you want to assign to variable *variable-name*.

```
EVAL COUNTER=3
```

changes the value of `COUNTER` to 3 and shows

```
COUNTER=3 = 3
```

on the message line of the **Display Module Source** display.

Use the EVAL debug command to assign numeric, alphabetic, and alphanumeric data to variables. When you assign values to a character variable, the following rules apply:

- If the length of the source expression is less than the length of the target expression, the data is left justified in the target expression and the remaining positions are filled with blanks
- If the length of the source expression is greater than the length of the target expression, the data is left justified in the target expression and truncated to the length of the target expression.

The scope of the variables used in the EVAL debug command is defined by using the QUAL debug command. To change a variable at line 48, type `QUAL 48`. Line 48 is the number within a function to which you want the variables scoped for the EVAL debug command.

Note: You do not always have to use the QUAL debug command before the `EVAL` debug command. An automatic `QUAL` is done when a breakpoint is encountered or a step is done. This establishes the default for the scoping rules to be the current stop location.

Figure 43 shows the results of changing the array element at 1 from $ to #.

```
EVAL hold_formatted_cost [1] = '#'
   hold_formatted_cost[1]= '#' = '#':

//Code evaluated before statement 51 where a breakpoint is set
47      if (j<0) return(0);
48      if (hold_formatted_cost[i] == '$')
49      {
50        formatted_cost[j] = hold_formatted_cost[i];
51        break;
52      }
53      if (i<16 && !((i-2)%3))
54      {
55        formatted_cost[j] = ',';
56        --j;
57      }
58      formatted_cost[j] = hold_formatted_cost[i];
59      --j;
60    }
61
```

Figure 43. Using EVAL to Change a Variable

Equating a Name with a Variable, Expression, or Command

You use the EQUATE debug command to equate a name with a variable, expression, or debug command for shorthand use. You can then use that name alone or within another expression. If you use it within another expression, the value of the name is determined before the expression is evaluated. These names stay active until a debug session ends or a name is removed.

To equate a name with a field, expression or debug command, type:

```
EQUATE shorthand-name definition
```

on the debug command line. *shorthand-name* is the character string that contains no blanks that you want to equate with a variable, expression, or debug command. *Definition* is a character string separated from *shorthand-name* by at least one blank that is the variable, expression, or debug command that you are equating with the name. The character strings can be in uppercase, lowercase, or mixed case.

For example, to define a shorthand name called `DC` which displays the contents of a variable called COUNTER, type:

```
EQUATE DC EVAL COUNTER
```

on the debug command line. Each time `DC` is typed on the debug command line, the command EVAL COUNTER is performed.

The maximum number of characters that can be typed in an EQUATE command is 144 which is the length of the command line. After any EQUATE commands are expanded, the length is limited to 150 characters, which is the maximum command length. If a definition is not supplied and a previous EQUATE command defined the name, the previous definition is removed. If the name was not previously defined, an error message is shown.

To see the names that have been defined with the EQUATE debug command for a debug session, type `DISPLAY EQUATE` on the debug command line. A list of the active names is shown on the Evaluate Expression display.

Sample Source for EVAL Commands

The sample `EVAL` commands presented in Figure 32 on page 217 and Figure 33 on page 220 are based on the following source:

```
#include <iostream.h>
#include <pointer.h>

/** POINTERS **/
_SYSPTR pSys;             //System pointer
_SPCPTR pSpace;           //Space pointer
int (*fncptr)(void);      //Function pointer
char *pc1;                //Character pointer
char *pc2;                //Character pointer
```

```
int  *pi1;                //Integer pointer
char arr1[] = "ABC";      //Array

/** SIMPLE VARIABLES **/
int i1;                   //Integer
unsigned u1;              //Unsigned Integer
char c1;                  //Character
float f1;                 //Float

/** STRUCTURES  **/
struct {              //Bit fields
  int b1 : 1;
  int b4 : 4;
 }bits;
struct x{                 // Tagged structure
   int x;
   char *p;
};
struct y{                 // Structure with
   int y;                      // structure member
   struct x x;
};
typedef struct z {        // Structure typedef
   int z;
   char *p;
} z;
z zz;                          // Structure using typedef
z *pZZ;                        // Same
typedef struct c {        // Structure typedef
   unsigned a;
   char *b;
} c;
c d;                           // Structure using typedef

/** UNIONS **/
union u{                  // Union
   int x;
   unsigned y;
};
union u u;                    // Variable using union
union u *pU;                  // Same

/** ENUMERATIONS **/
enum number {one, two, three};
enum color {red,yellow,blue};
enum number Number = one;
enum color Color = blue;

/** FUNCTION **/
int ret100(void) { return 100;}
main()
{
```

```
   float dec1;
   struct y y, *pY;
   bits.b1 = 1;
   bits.b4 = 2;
   i1 = ret100();
   c1 = 'C';
   f1 = 100e2;
   dec1 = 12.3;
   pc1 = &c1;
   pi1 = &i1;
   d.a = 1;
   pZZ = &zz;
   pZZ->z=1;
   pY = &y;
   pY->x.p=(char*)&y;
   pU=&u;
   pU->x=255;
   Number=(number)Color;
   fncptr = &ret100;
   pY->x.x=1;                    // Set breakpoint here
 }
```

Sample Source for Displaying System and Space Pointers

The sample EVAL command for displaying system and space pointers presented in Figure 34 on page 221 is based on the following source:

```
#include <iostream.h>
#include <mispace.h>
#include <pointer.h>
#include <mispcobj.h>
#include <except.h>
#include <lecond.h>
#include <leenv.h>
#include <qtedbgs.h>      // From qsysinc

// Link up the Create User Space API
#pragma linkage(CreateUserSpace,OS)
#pragma map(CreateUserSpace,"QUSCRTUS")
void CreateUserSpace(char[20],
                     char[10],
                     long int,
                     char,
                     char[10],
                     char[50],
                     char[10],
                     _TE_ERROR_CODE_T *
                     );

// Link up the Delete User Space API
#pragma linkage(DeleteUserSpace,OS)
```

```
#pragma map(DeleteUserSpace,"QUSDLTUS")
void DeleteUserSpace(char[20],
                     _TE_ERROR_CODE_T *
                     );

// Link up the Retrieve Pointer to User Space API
#pragma linkage(RetrievePointerToUserSpace,OS)
#pragma map(RetrievePointerToUserSpace,"QUSPTRUS")
void RetrievePointerToUserSpace(char[20],
                                char **,
                                _TE_ERROR_CODE_T *
                                );

int main (int argc, char *argv[])
{
    char *pBuffer;
    _SYSPTR pSYSptr;
    _TE_ERROR_CODE_T errorCode;
    errorCode.BytesProvided = 0;
    CreateUserSpace("QTEUSERSPCQTEMP     ",
                    "QTESSPC   ",
                    10,
                    0,
                    "*ALL      ",
                    "                                                    ",
                    "*YES      ",
                    &errorCode
                    );

    //! call RetrievePointerToUserSpace - Retrieve Pointer to User Space
    //!! (pass: Name and library of user space, pointer variable
    //!! return: nothing (pointer variable is left pointing to start
    //!!         of user space)
    RetrievePointerToUserSpace("QTEUSERSPCQTEMP     ",
                               &pBuffer,
                               &errorCode);

    // convert the space pointer to a system pointer
    pSYSptr = _SETSPFP(pBuffer);
    cout << "Space pointer: " << pBuffer << endl;
    cout << "System pointer: " << pSYSptr << endl;
    return 0;}
```

Sample Source for Displaying C++ Constructs

The sample EVAL command for displaying C++ constructs presented in Figure 35 on page 222 is based on the following source:

```
// Program demonstrates the EVAL debug command
class A {
  public:
     union {
       int a;
       int ua;
     };
  int ac;
  int amb;
  int not_amb;
  };

class B {
  public:
     int b;
};

class C {
  public:
     int ac;
     static int c;
     int amb;
     int not_amb;
};

int C::c = 45;
template <class T> class E : public A, public virtual B {
  public:
     T e;
};

class D : public C, public virtual B {
   public:
      int d;
};

class outter {
   public:
      static int static_i;
      class F : public E<int>, public D {
         public:
            int f;
            int not_amb;
            void funct();
       } inobj;
};

int outter :: static_i = 45;

int VAR = 2;
```

```
void outter::F::funct()
{
    int local;
    a=1;              //EVAL VAR  : Is VAR in global scope
    b=2;
    c=3;
    d=4;
    e=5;
    f=6;

    local = 828;
    int VAR;

    VAR=1;
    static_i=10;
    A::ac=12;
    C::ac=13;
    not_amb=32;

    not_amb=13;
    // Stop here and show:
    // EVAL VAR          : is VAR in local scope
    // EVAL ::VAR        : is VAR in global scope
    // EVAL %LOCALVARS   : see all local vars
    // EVAL *this        : fields of derived class
    // EVAL this->f      : show member f
    // EVAL f            : in derived class
    // EVAL a            : in base class
    // EVAL b            : in Virtual Base class
    // EVAL c            : static member
    // EVAL static_i     : static var made visible
                                          : by middle-end
    // EVAL au           : anonymous union members
    // EVAL a=49         :
    // EVAL au
    // EVAL ac           : show ambigous var
    // EVAL A::ac        : disambig with scope op
    // EVAL B::ac        : Scope op
    // EVAL E<int>::ac   : Scope op
    // EVAL this         : notice pointer values
    // EVAL (E<int>*)this      : change
    // EVAL (class D *)this    : class is optional
    // EVAL *(E<int> *)this    : show fields
    // EVAL *(D *) this       : show fields

void main()
{
   outter obj;
   int outter::F::*mptr = &outter::F::b;
   int i;
   int& r = i;
   obj.inobj.funct();
```

```
    i = 777;

    obj.static_i = 2;
    // Stop here
    // EVAL obj.inobj.*mptr  : member ptr
    // EVAL obj.inobj.b
    // EVAL i
    // EVAL r
    // EVAL r=1
    // EVAL i
    // EVAL (A &) (obj.inobj)  : reference cast
    // EVAL
}
```

Bibliography

This bibliography lists the publications that make up the IBM VisualAge for C++ for AS/400 library and publications of related IBM products referenced in this guide. The list of related publications is not exhaustive but should be adequate for most VisualAge for C++ for AS/400 users.

The IBM VisualAge for C++ for AS/400 Library

The following books are part of the IBM VisualAge for C++ for AS/400 library:

- *LPS: VisualAge for C++ for AS/400*, GC09-2414
- *VisualAge for C++ for AS/400 Installation Guide and Product Overview*, SC09-2415
- *VisualAge for C++ for AS/400 C++ User's Guide*, SC09-2416
- *VisualAge for C++ for AS/400 C++ Programming Guide*, SC09-2417
- *ILE C/C++ MI Library Reference*, SC09-2418
- *VisualAge for C++ for AS/400 IBM Open Class Library Reference*, SC09-2440
- *VisualAge for C++ for AS/400 C Library Reference*, SC09-2441
- *VisualAge for C++ for AS/400 C++ Language Reference*, SC09-2442
- *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*, SC09-2443
- *IBM Access Class Library for OS/400 Reference*, SC41-4620
- *IBM Access Class Library for Windows Reference*, SC41-4622
- *IBM Access Class Library User's Guide*, SC41-4623
- *ILE Concepts*, SC41-4606
- *CL Programming*, SC41-4721
- *CL Reference*, SC41-4722

Other IBM Publications

- *VisualAge for C++ for Windows User's Guide*, S33H-5031
- *Client Access for Windows -- Setup*, SC41-3512

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)*

Other Books You might Need

The following contains a list of books that you might find helpful.

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 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 Margret 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.

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W