

IBM i
7.5

*Programming
ILE Concepts*



Note

Before using this information and the product it supports, read the information in [“Notices” on page 193](#).

This edition applies to IBM® i 7.3 (product number 5770-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

© **Copyright International Business Machines Corporation 1997, 2022.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

ILE Concepts.....	1
What's new for IBM i 7.5.....	3
PDF file for ILE Concepts.....	5
Integrated Language Environment Introduction.....	7
What Is ILE?.....	7
What Are the Benefits of ILE?.....	7
Binding.....	7
Modularity.....	7
Reusable Components.....	8
Common Runtime Services.....	8
Coexistence with Existing Applications.....	8
Source Debugger.....	8
Better Control over Resources.....	8
Shared Open Data Path—Scenario.....	9
Commitment Control—Scenario.....	9
Better Control over Language Interactions.....	10
Mixed Languages—Scenario.....	10
Better Code Optimization.....	11
What Is the History of ILE?.....	11
Original Program Model Description.....	11
Principal Characteristics of OPM.....	12
Extended Program Model Description.....	12
Integrated Language Environment Description.....	12
Principal Characteristics of Procedure-Based Languages.....	13
ILE Basic Concepts.....	15
Structure of an ILE Program.....	15
Procedure.....	16
Module Object.....	16
ILE Program.....	17
Service Program.....	19
Binding Directory.....	21
Binding Directory Processing.....	22
Binder Functions.....	22
Calling a Program or a Procedure.....	24
Dynamic Program Calls.....	24
Static Procedure Calls.....	24
Procedure Pointer Calls.....	25
Activation.....	25
Error Handling Overview.....	26
Optimizing Translator.....	27
Debugger.....	28
ILE Advanced Concepts.....	29
Program Activation.....	29
Program Activation Creation.....	29
Activation Group.....	30
Activation Group Creation.....	31

Default Activation Groups.....	32
Non-Default Activation Group Deletion.....	33
Service Program Activation.....	35
Control Boundaries.....	37
Control Boundaries for Activation Groups.....	37
Control Boundaries between OPM and ILE Call Stack Entries.....	38
Control Boundary Use.....	39
Error Handling.....	39
Job Message Queues.....	40
Exception Messages and How They Are Sent.....	40
How Exception Messages Are Handled.....	41
Exception Recovery.....	41
Default Actions for Unhandled Exceptions.....	42
Types of Exception Handlers.....	43
ILE Conditions.....	45
Data Management Scoping Rules.....	45
Call-Level Scoping.....	46
Activation-Group-Level Scoping.....	46
Job-Level Scoping.....	47
Teraspace and Single-Level Storage.....	49
Teraspace Characteristics.....	49
Using Teraspace for Storage.....	49
Choosing a Program Storage Model.....	50
Specifying the Teraspace Storage Model.....	50
Selecting a Compatible Activation Group.....	51
How the Storage Models Interact.....	51
Rules for Binding Modules.....	51
Rules for Binding to Service Programs.....	52
Rules for Activating Programs and Service Programs.....	52
Rules for Program and Procedure Calls.....	52
Converting Your Program or Service Program to Inherit a Storage Model.....	52
Updating Your Programs: Teraspace Considerations.....	53
Taking Advantage of 8-byte Pointers in Your C and C++ Code.....	53
Pointer Support in C and C++ Compilers.....	54
Pointer Conversions.....	54
Using the Teraspace Storage Model.....	55
Using Teraspace: Best Practices.....	55
System Controls over Teraspace Programs When They are Created.....	56
System Controls over Teraspace Programs When They are Activated.....	56
IBM i Interfaces and Teraspace.....	56
Potential Problems that Can Arise When You Use Teraspace.....	56
Teraspace Usage Tips.....	57
Program Creation Concepts.....	61
Create Program and Create Service Program Commands.....	61
Use Adopted Authority (QUSEADPAUT).....	62
Using optimization parameters.....	62
Stored data in modules and programs.....	62
Symbol Resolution.....	63
Resolved and Unresolved Imports.....	64
Binding by Copy.....	64
Binding by Reference.....	64
Binding Large Numbers of Modules.....	64
Duplicate Symbols.....	65
Importance of the Order of Exports.....	65
Program Creation Example 1.....	66
Program Creation Example 2.....	68

Program Access.....	70
Program Entry Procedure Module Parameter on the CRTPGM Command.....	70
Export Parameter on the CRTSRVPGM Command.....	71
Export Parameter Used with Source File and Source Member Parameters.....	71
Maximum width of a file for the SRCFILE parameter.....	72
Import and Export Concepts.....	72
Binder Language.....	73
Signature.....	74
Start Program Export and End Program Export Commands.....	75
Program Level Parameter on the STRPGMEXP Command.....	75
Signature Parameter on the STRPGMEXP Command.....	75
Level Check Parameter on the STRPGMEXP Command.....	75
Export Symbol Command.....	76
Wildcard Export Symbol Examples.....	76
Binder Language Examples.....	77
Binder Language Example 1.....	77
Binder Language Example 2.....	78
Binder Language Example 3.....	79
Binder Language Example 4.....	81
Changing Programs.....	84
Program Updates.....	85
Parameters on the UPDPGM and UPDSRVPGM Commands.....	86
Module Replaced by a Module with Fewer Imports.....	87
Module Replaced by a Module with More Imports.....	87
Module Replaced by a Module with Fewer Exports.....	87
Module Replaced by a Module with More Exports.....	88
Tips for Creating Modules, Programs, and Service Programs.....	88
Activation Group Management.....	91
Multiple Applications Running in the Same Job.....	91
Reclaim Resources Command.....	92
Reclaim Resources Command for OPM Programs.....	93
Reclaim Resources Command for ILE Programs.....	94
Reclaim Activation Group Command.....	94
Service Programs and Activation Groups.....	94
Calls to Procedures and Programs.....	97
Call Stack.....	97
Call Stack Example.....	97
Calls to Programs and Calls to Procedures.....	98
Static Procedure Calls.....	98
Procedure Pointer Calls.....	99
Passing Arguments to ILE Procedures.....	99
Function Results.....	100
Omitted Arguments.....	101
Dynamic Program Calls.....	101
Passing Arguments on a Dynamic Program Call.....	101
Interlanguage Data Compatibility.....	102
Syntax for Passing Arguments in Mixed-Language Applications.....	102
Operational Descriptors.....	102
Requirements of Operational Descriptors.....	102
Absence of a Required Descriptor.....	103
Presence of an Unnecessary Descriptor.....	103
Bindable APIs for Operational Descriptor Access.....	103
Support for OPM and ILE APIs.....	103
Storage Management.....	105
Single-Level Storage Heap.....	105

Heap Characteristics.....	105
Default Heap.....	105
User-Created Heaps.....	106
Single-Heap Support.....	106
Heap Allocation Strategy.....	106
Single-Level Storage Heap Interfaces.....	107
Heap Support.....	107
Thread Local Storage.....	108
Exception and Condition Management.....	111
Handle Cursors and Resume Cursors.....	111
Exception Handler Actions.....	112
How to Resume Processing.....	112
How to Percolate a Message.....	113
How to Promote a Message.....	113
Default Actions for Unhandled Exceptions.....	114
Nested Exceptions.....	114
Condition Handling.....	115
How Conditions Are Represented.....	115
Layout of a Condition Token.....	115
Condition Token Testing.....	117
Relationship of ILE Conditions to Operating System Messages.....	117
IBM i Messages and the Bindable API Feedback Code.....	117
Debugging Considerations.....	119
Debug Mode.....	119
Debug Environment.....	119
Addition of Programs to Debug Mode.....	119
How Observability and Optimization Affect Debugging.....	120
Optimization Levels.....	120
Debug Data Creation and Removal.....	120
Module Views.....	120
Debugging across Jobs.....	120
OPM and ILE Debugger Support.....	121
Watch Support.....	121
Unmonitored Exceptions.....	121
Globalization Restriction for Debugging.....	121
Data Management Scoping.....	123
Common Data Management Resources.....	123
Commitment Control Scoping.....	124
Commitment Definitions and Activation Groups.....	124
Ending Commitment Control.....	125
Commitment Control during Activation Group End.....	125
ILE Bindable Application Programming Interfaces.....	127
ILE Bindable APIs Available.....	127
Dynamic Screen Manager Bindable APIs.....	130
Advanced Optimization Techniques.....	131
Program Profiling.....	131
Types of Profiling.....	131
How to Profile a Program.....	132
Enabling the program to collect profiling data.....	132
Collect Profiling Data.....	133
Applying the Collected Profiling Data.....	133
Managing Programs Enabled to Collect Profiling Data.....	134
Managing Programs with Profiling Data Applied to Them.....	135

How to Tell if a Program or Module is Profiled or Enabled for Collection.....	135
Interprocedural analysis (IPA).....	136
How to optimize your programs with IPA.....	138
IPA control file syntax.....	138
IPA usage notes.....	140
IPA restrictions and limitations.....	140
Partitions created by IPA.....	141
Advanced Argument Optimization.....	142
How to Use Advanced Argument Optimization.....	142
Considerations and Restrictions When Using Advanced Argument Optimization.....	142
Licensed Internal Code Options.....	143
Currently Defined Options.....	143
Application.....	147
Restrictions.....	147
Syntax.....	147
Release Compatibility.....	148
Displaying Module and ILE Program Licensed Internal Code Options.....	148
Adaptive Code Generation.....	149
ACG Concepts.....	149
Normal Operation.....	150
Restore Options.....	150
QFRCCVNRST System Value.....	150
FRCOBJCVN Parameter.....	151
Create Options.....	151
The CodeGenTarget LICOPT.....	152
QIBM_BN_CREATE_WITH_COMMON_CODEGEN Environment Variable.....	153
Displaying ACG Information.....	153
Object compatibility.....	153
Release-to-Release Considerations.....	154
Optimizing Compatible Programs.....	154
ACG and Logical Partitions.....	155
Shared Storage Synchronization.....	157
Shared Storage.....	157
Shared Storage Pitfalls.....	157
Shared Storage Access Ordering.....	157
Example Problem 1: One Writer, Many Readers	158
Example 1 Solution.....	158
Storage Synchronizing Actions.....	159
Example Problem 2: Two Contending Writers or Readers.....	160
Example 2 Solution.....	161
Alternate Solution: Using Check Lock Value / Clear Lock Value.....	161
Output Listing from CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM Command.....	163
Binder Listing.....	163
Basic Listing.....	163
Extended Listing.....	165
Full Listing.....	166
IPA Listing Components.....	167
Object File Map.....	167
Compiler Options Map.....	168
Inline Report.....	168
Global Symbols Map.....	169
Partition Map.....	169
Source File Map.....	169
Messages.....	169
Message Summary.....	169
Listing for Example Service Program.....	169

Binder Information Listing for Example Service Program.....	169
Cross-Reference Listing for Example Service Program.....	170
Binding Statistics for Example Service Program.....	171
Binder Language Errors.....	171
Signature Padded.....	171
Suggested Changes.....	172
Signature Truncated.....	172
Suggested Changes.....	172
Current Export Block Limits Interface.....	172
Suggested Changes.....	173
Duplicate Export Block.....	173
Suggested Changes.....	173
Duplicate Symbol on Previous Export.....	174
Suggested Changes.....	174
Level Checking Cannot Be Disabled More than Once, Ignored.....	174
Suggested Changes.....	174
Multiple Current Export Blocks Not Allowed, Previous Assumed.....	175
Suggested Changes.....	175
Current Export Block Is Empty.....	175
Suggested Changes.....	176
Export Block Not Completed, End-of-File Found before ENDPGMEXP.....	176
Suggested Changes.....	176
Export Block Not Started, STRPGMEXP Required.....	176
Suggested Changes.....	177
Export Blocks Cannot Be Nested, ENDPGMEXP Missing.....	177
Suggested Changes.....	177
Exports Must Exist inside Export Blocks.....	177
Suggested Changes.....	178
Identical Signatures for Dissimilar Export Blocks, Must Change Exports.....	178
Suggested Changes.....	178
Multiple Wildcard Matches.....	178
Suggested Changes.....	179
No Current Export Block.....	179
Suggested Changes.....	179
No Wildcard Matches.....	179
Suggested Changes.....	180
Previous Export Block Is Empty.....	180
Suggested Changes.....	180
Signature Contains Variant Characters.....	180
Suggested Changes.....	181
SIGNATURE(*GEN) Required with LVLCHK(*NO).....	181
Suggested Changes.....	181
Signature Syntax Not Valid.....	181
Suggested Changes.....	182
Symbol Name Required.....	182
Suggested Changes.....	182
Symbol Not Allowed as Service Program Export.....	182
Suggested Changes.....	183
Symbol Not Defined.....	183
Suggested Changes.....	183
Syntax Not Valid.....	184
Suggested Changes.....	184
Exceptions in Optimized Programs.....	185
CL Commands Used with ILE Objects.....	187
CL Commands Used with Modules.....	187
CL Commands Used with Program Objects.....	187

CL Commands Used with Service Programs.....	188
CL Commands Used with Binding Directories.....	188
CL Commands Used with Structured Query Language.....	188
CL Commands Used with CICS.....	188
CL Commands Used with Source Debugger.....	189
CL Commands Used to Edit the Binder Language Source File.....	189
Related information.....	191
Notices.....	193
Programming interface information.....	194
Trademarks.....	194
Terms and conditions.....	195
Index.....	197

ILE Concepts

This information describes concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the IBM i licensed program. Subjects covered include module creation, binding, message handling, the running and debugging of programs, and exception handling.

The concepts described in this information pertain to all ILE languages. Each ILE language may implement the ILE architecture somewhat differently. To determine exactly how each language enables the concepts described here, refer to the programmer's guide for that specific ILE language.

This information also describes IBM i functions that directly pertain to all ILE languages.

This information does not describe migration from an existing IBM i language to an ILE language. That information is contained in each ILE high-level language (HLL) programmer's guide.

What's new for IBM i 7.5

Here are the major changes to this information for this edition.

- Support for new synchronization MI instructions.

PDF file for ILE Concepts

You can view and print a PDF file of this information.

To view or download the PDF version of this document, select [ILE Concepts](#).

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the [Adobe Web site](http://www.adobe.com/products/acrobat/readstep.html) (www.adobe.com/products/acrobat/readstep.html) .

Integrated Language Environment Introduction

This topic defines the Integrated Language Environment (ILE) model, describes the benefits of ILE, and explains how ILE evolved from previous program models.

What Is ILE?

ILE is a set of tools and associated system support designed to enhance program development on the IBM i operating system.

The capabilities of this model can be used only by programs produced by the ILE family of compilers. That family includes ILE RPG, ILE COBOL, ILE C, ILE C++, and ILE CL.

What Are the Benefits of ILE?

ILE offers numerous benefits over previous program models. Those benefits include binding, modularity, reusable components, common runtime services, coexistence, and a source debugger. They also include better control over resources, better control over language interactions, better code optimization, a better environment for C, and a foundation for the future.

Binding

The benefit of binding is that it helps reduce the overhead associated with call operations. Binding the modules together speeds up the call. The previous call mechanism is still available, but there is also a faster alternative. To differentiate between the two types of calls, the previous method is referred to as a dynamic or external program call, and the ILE method is referred to as a static or bound procedure call.

The binding capability, together with the resulting improvement in call performance, makes it far more practical to develop applications in a highly modular fashion. An ILE compiler does not produce a program that can be run. Rather, it produces a module object (*MODULE) that can be combined (bound) with other modules to form a single runnable unit; that is, a program object (*PGM).

Just as you can call an RPG program from a COBOL program, ILE allows you to bind modules written in different languages. Therefore, it is possible to create a single runnable program that consists of modules written separately in RPG, COBOL, C, C++, and CL.

Modularity

The benefits from using a modular approach to application programming include the following:

- Faster compile time

The smaller the piece of code we compile, the faster the compiler can process it. This benefit is particularly important during maintenance, because often only a line or two needs to be changed. When we change two lines, we may have to recompile 2000 lines. That is hardly an efficient use of resources.

If we modularize the code and take advantage of the binding capabilities of ILE, we may need to recompile only 100 or 200 lines. Even with the binding step included, this process is considerably faster.

- Simplified maintenance

When updating a very large program, it is very difficult to understand exactly what is going on. This is particularly true if the original programmer wrote in a different style from your own. A smaller piece of code tends to represent a single function, and it is far easier to grasp its inner workings. Therefore, the logical flow becomes more obvious, and when you make changes, you are far less likely to introduce unwanted side effects.

- Simplified testing

Smaller compilation units encourage you to test functions in isolation. This isolation helps to ensure that test coverage is complete; that is, that all possible inputs and logic paths are tested.

- Better use of programming resources

Modularity lends itself to greater division of labor. When you write large programs, it is difficult (if not impossible) to subdivide the work. Coding all parts of a program may stretch the talents of a junior programmer or waste the skills of a senior programmer.

- Easier migration of code from other operating systems

Reusable Components

With ILE, you can select packages of routines that can be blended into your own programs. Routines written in any ILE language can be used by all ILE compiler users. The fact that programmers can write in the language of their choice ensures that you have the widest possible selection of routines.

The same mechanisms that IBM and other vendors use to deliver these packages to you are available for you to use in your own applications. Your installation can develop its own set of standard routines, and do so in any language it chooses.

Not only can you use off-the-shelf routines in your own applications. You can also develop routines in the ILE language of your choice and market them to users of any ILE language.

Common Runtime Services

A selection of off-the-shelf components (**bindable APIs**) is supplied as part of ILE, ready to be incorporated into your applications. These APIs provide services such as:

- Date and time manipulation
- Message handling
- Math routines
- Greater control over screen handling
- Dynamic storage allocation

Over time, additional routines will be added to this set and others will be available from third-party vendors.

IBM has online information that provides further details of the APIs supplied with ILE. Refer to the APIs topic that is found in the Programming category of the IBM i Information Center.

Coexistence with Existing Applications

ILE programs can coexist with existing OPM programs. ILE programs can call OPM programs and other ILE programs. Similarly, OPM programs can call ILE programs and other OPM programs. Therefore, with careful planning, it is possible to make a gradual transition to ILE.

Source Debugger

The source debugger allows you to debug ILE programs and service programs. For information about the source debugger, see [“Debugging Considerations” on page 119](#).

Better Control over Resources

Before the introduction of ILE, resources (for example, open files) used by a program could be scoped to (that is, owned by) only:

- The program that allocated the resources
- The job

In many cases, this restriction forces the application designer to make tradeoffs.

ILE offers a third alternative. A portion of the job can own the resource. This alternative is achieved through the use of an ILE construct, the **activation group**. Under ILE, a resource can be scoped to any of the following:

- A program
- An activation group
- The job

Shared Open Data Path—Scenario

Shared open data paths (ODPs) are an example of resources you can better control with ILE's new level of scoping.

To improve the performance of an application, a programmer decides to use a shared ODP for the customer master file. That file is used by both the Order Entry and the Billing applications.

Because a shared ODP is scoped to the job, it is quite possible for one of the applications to inadvertently cause problems for the other. In fact, avoiding such problems requires careful coordination among the developers of the applications. If the applications were purchased from different suppliers, avoiding problems may not even be possible.

What kind of problems can arise? Consider the following scenario:

1. The customer master file is keyed on account number and contains records for account numbers A1, A2, B1, C1, C2, D1, D2, and so on.
2. An operator is reviewing the master file records, updating each as required, before requesting the next record. The record currently displayed is for account B1.
3. The telephone rings. Customer D1 wants to place an order.
4. The operator presses the Go to Order Entry function key, processes the order for customer D1, and returns to the master file display.
5. The program still correctly displays the record for B1, but when the operator requests the next record, which record is displayed?

If you said D2, you are correct. When the Order Entry application read record D1, the current file position changed because the shared ODP was scoped to the job. Therefore, the request for the next record means the next record after D1.

Under ILE, this problem could be prevented by running the master file maintenance in an activation group dedicated to Billing. Likewise, the Order Entry application would run in its own activation group. Each application would still gain the benefits of a shared ODP, but each would have its own shared ODP, owned by the relevant activation group. This level of scoping prevents the kind of interference described in this example.

Scoping resources to an activation group allows programmers the freedom to develop an application that runs independently from any other applications running in the job. It also reduces the coordination effort required and enhances the ability to write drop-in extensions to existing application packages.

Commitment Control—Scenario

The ability to scope a shared open data path (ODP) to the application is useful in the area of commitment control.

Assume that you want to use a file under commitment control but that you also need it to use a shared ODP. Without ILE, if one program opens the file under commitment control, all programs in the job have to do so. This is true even if the commitment capability is needed for only one or two programs.

One potential problem with this situation is that, if any program in the job issues a commit operation, all updates are committed. The updates are committed even if logically they are not part of the application in question.

These problems can be avoided by running each part of the application that requires commitment control in a separate activation group.

Better Control over Language Interactions

Without ILE, the way a program runs on the operating system depends on a combination of the following factors:

- The language standard (for example, the ANSI standards for COBOL and C)
- The developer of the compiler

This combination can cause problems when you mix languages.

Mixed Languages—Scenario

Without activation groups, which are introduced by ILE, interactions among OPM languages are difficult to predict. ILE activation groups can solve that difficulty.

For example, consider the problems caused by mixing COBOL with other languages. The COBOL language standard includes a concept known as a **run unit**. A run unit groups programs together so that under certain circumstances they behave as a single entity. This can be a very useful feature.

Assume that three ILE COBOL programs (PRGA, PRGB, and PRGC) form a small application in which PRGA calls PRGB, which in turn calls PRGC (see [Figure 1 on page 10](#)). Under the rules of ILE COBOL, these three programs are in the same run unit. As a result, if any of them ends, all three programs should be ended and control should return to the caller of PRGA.

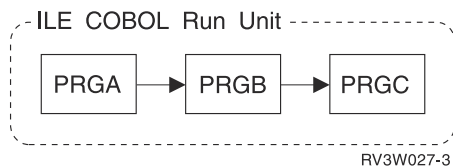


Figure 1. Three ILE COBOL Programs in a Run Unit

Suppose that we now introduce an RPG program (RPG1) into the application and that RPG1 is also called by the COBOL program PRGB (see [Figure 2 on page 10](#)). An RPG program expects that its variables, files, and other resources remain intact until the program returns with the last-record (LR) indicator on.

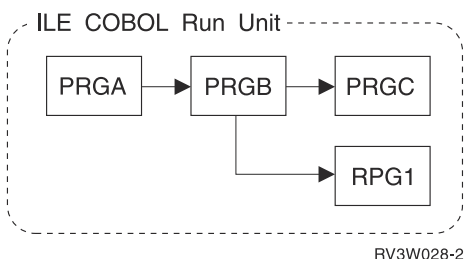


Figure 2. Three ILE COBOL Programs and One ILE RPG Program in a Run Unit

However, the fact that program RPG1 is written in RPG does not guarantee that all RPG semantics apply when RPG1 is run as part of the COBOL run unit. If the run unit ends, RPG1 disappears regardless of its LR indicator setting. In many cases, this situation may be exactly what you want. However, if RPG1 is a utility program, perhaps controlling the issue of invoice numbers, this situation is unacceptable.

We can prevent this situation by running the RPG program in a separate activation group from the COBOL run unit (see [Figure 3 on page 11](#)). An ILE COBOL run unit itself is an activation group.

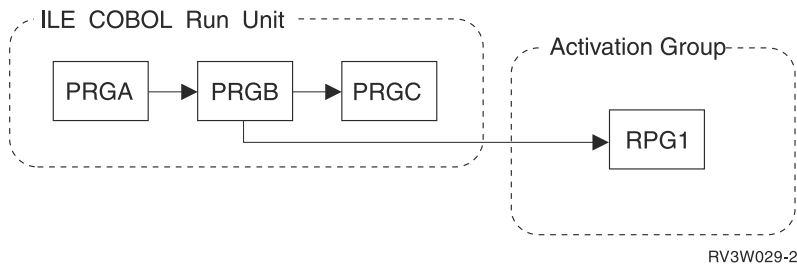


Figure 3. ILE RPG Program in a Separate Activation Group

For information about the differences between an OPM run unit and an ILE run unit, see the [ILE COBOL Programmer's Guide](#) .

Better Code Optimization

The translator can do many more types of optimization for ILE programs than it can for OPM programs.

An ILE-enabled compiler does not directly produce a module. First, it produces an intermediate form of the module, and then it calls the ILE translator to translate the intermediate code into instructions that can be run. By using an intermediate code that is used as input to the common ILE translator, an optimization added to the translator for one ILE language might benefit all ILE languages.

What Is the History of ILE?

ILE is a stage in the evolution of IBM i program models. Each stage evolved to meet the changing needs of application programmers.

The programming environment provided when the AS/400 system was first introduced is called the original program model (OPM). In OS/400® Version 1 Release 2, the Extended Program Model (EPM) was introduced.

Original Program Model Description

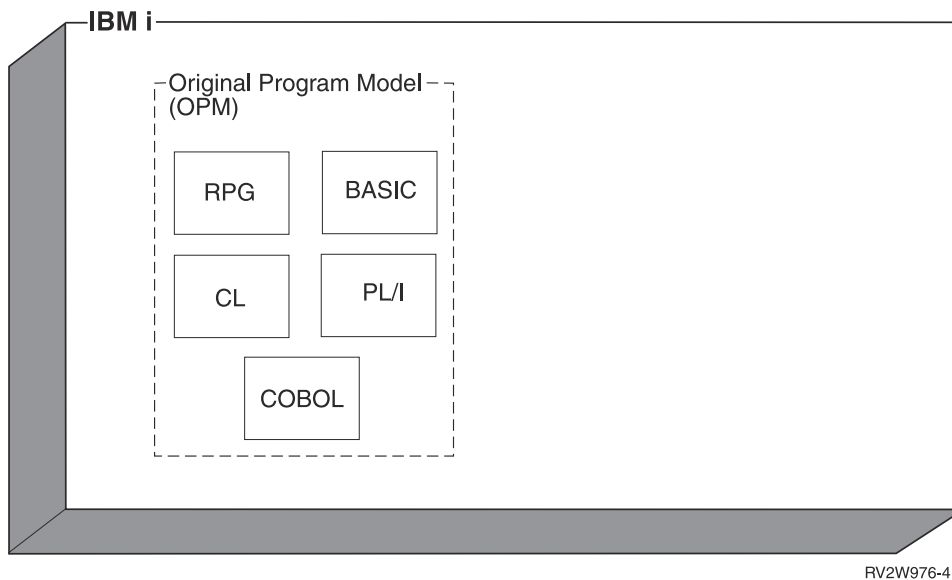
Application developers enter source code into a source file and compile that source. If the compilation is a success, a program object is created. The set of functions, processes, and rules that are used to directly create and run a program object is known as the *original program model (OPM)*.

As an OPM compiler generates the program object, it generates additional code. The additional code initializes program variables and provides any necessary code for special processing that is needed by the particular language. The special processing could include processing any input parameters expected by this program. When a program is to start running, the additional compiler-generated code becomes the starting point (entry point) for the program.

A program is typically activated when the operating system encounters a call request. At runtime, the call to another program is a dynamic program call. The resources needed for a dynamic program call can be significant. Application developers often design an application to consist of a few large programs that minimize the number of dynamic program calls.

Figure 4 on page 12 illustrates the relationship between OPM and the operating system. As you can see, RPG, COBOL, CL, BASIC, and PL/I all operate in this model. As of release 6.1, the BASIC compiler is no longer available.

The broken line forming the OPM boundary indicates that OPM is an integral part of IBM i. This integration means that many functions normally provided by the compiler writer are built into the operating system. The resulting standardization of calling conventions allows programs written in one language to freely call those written in another. For example, an application written in RPG typically includes a number of CL programs to issue file overrides or to send messages.



RV2W976-4

Figure 4. Relationship of OPM to IBM i

Principal Characteristics of OPM

The following list identifies the principal characteristics of OPM:

- Dynamic binding

When program A wants to call program B, it just does so. This dynamic program call is a simple and powerful capability. At runtime, the operating system locates program B and ensures that the user has the right to use it.

An OPM program has only a single entry point, whereas, each procedure in an ILE program can be an entry point.

- Limited data sharing

In OPM, an internal procedure has to share variables with the entire program, whereas, in ILE, each procedure can have its own locally-scoped variables.

Extended Program Model Description

OPM continues to serve a useful purpose. However, OPM does not provide direct support for procedures as defined in languages like C. A *procedure* is a set of self-contained high-level language (HLL) statements that performs a particular task and then returns to the caller. Individual languages vary in the way that a procedure is defined. In C, a procedure is called a function.

To allow languages that define procedure calls between compilation units or languages that define procedures with local variables to run on the operating system, OPM was enhanced. These enhancements are called the Extended Program Model (EPM). Before ILE, EPM served as an interim solution for procedure-based languages like Pascal and C.

The IBM i operating system no longer provides any EPM compilers.

Integrated Language Environment Description

As [Figure 5 on page 13](#) shows, ILE is tightly integrated into IBM i, just as OPM is. It provides the same type of support for procedure-based languages that EPM does, but it does so far more thoroughly and consistently. Its design provides for the more traditional languages, such as RPG and COBOL, and for future language developments.

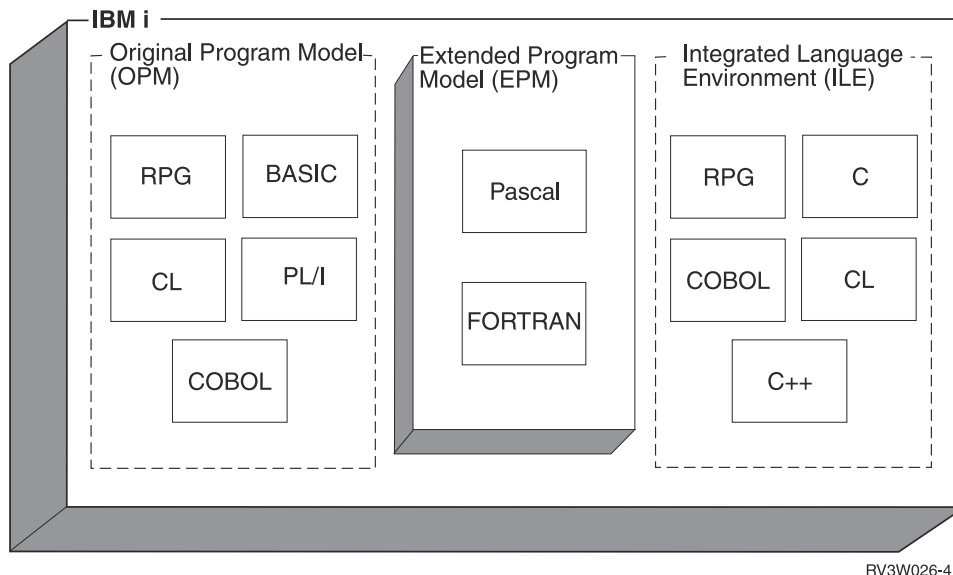


Figure 5. Relationship of OPM, EPM, and ILE to IBM i

Principal Characteristics of Procedure-Based Languages

Procedure-based languages have the following characteristics:

- Locally scoped variables

Locally scoped variables are known only within the procedure that defines them. The equivalent of locally scoped variables is the ability to define two variables with the same name that refer to two separate pieces of data. For example, the variable COUNT might have a length of 4 digits in subroutine CALCYR and a length of 6 digits in subroutine CALCDAY.

Locally scoped variables provide considerable benefit when you write subroutines that are intended to be copied into several different programs. Without locally scoped variables, the programmers must use a scheme such as naming variables based on the name of the subroutine.

- Automatic variables

Automatic variables are created whenever a procedure is entered. Automatic variables are destroyed when the procedure is exited.

- External variables

External data is one way of sharing data between programs. If program A declares a data item as external, program A is said to **export** that data item to other programs that want to share that data. Program D can then **import** the item without programs B and C being involved at all. For more information about imports and exports, see [“Module Object” on page 16](#).

- Multiple entry points

OPM COBOL and RPG programs have only a single entry point. In a COBOL program, it is the start of the PROCEDURE DIVISION. In an RPG program, it is the first-page (1P) output. This is the model that OPM supports.

Procedure-based languages, on the other hand, may have multiple entry points. For example, a C program may consist entirely of subroutines to be used by other programs. These procedures can be exported, along with relevant data if required, for other programs to import.

In ILE, programs of this type are known as service programs. They can include modules from any of the ILE languages. Service programs are similar in concept to dynamic link libraries (DLLs) in Microsoft Windows. Service programs are discussed in greater detail in [“Service Program” on page 19](#).

- Frequent calls

Programs written in procedure-based languages can be very call intensive.

ILE Basic Concepts

Table 1 on page 15 compares and contrasts the original program model (OPM) and the Integrated Language Environment (ILE) model. This topic briefly explains the similarities and differences listed in the table.

Table 1. Similarities and Differences between OPM and ILE	
OPM	ILE
Program	Program Service program
Compilation results in a runnable program	Compilation results in a nonrunnable module object
Compile, run	Compile, bind, run
Run units simulated for each language	Activation groups
Dynamic program call	Dynamic program call Static procedure call
Single-language focus	Mixed-language focus
Language-specific error handling	Common error handling Language-specific error handling
OPM debuggers	Source-level debugger

Structure of an ILE Program

An ILE program contains one or more modules. A module, in turn, contains one or more procedures (see Figure 6 on page 15).

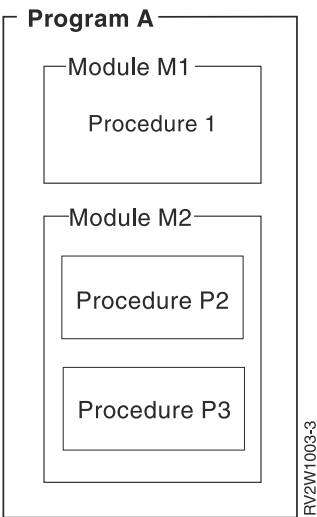


Figure 6. Structure of an ILE Program

Procedure

A *procedure* is a set of self-contained high-level language statements that performs a particular task and then returns to the caller. For example, an ILE C function is an ILE procedure.

Module Object

A **module object** is a *nonrunnable* object that is the output of an ILE compiler. A module object is represented to the system by the symbol *MODULE. A module object is the basic building block for creating runnable ILE objects. This is a significant difference between ILE and OPM. The output of an OPM compiler is a *runnable* program.

A module object can consist of one or more procedures and data item specifications. It is possible to directly access the procedures or data items in one module from another ILE object. See the ILE HLL programmer's guides for details on coding the procedures and data items that can be directly accessed by other ILE objects.

ILE RPG, ILE COBOL, ILE C, and ILE C++ all have the following common concepts:

- Exports

An **export** is the name of a procedure or data item, coded in a module object, that is available for use by other ILE objects. The export is identified by its name and its associated type, either procedure or data.

An export can also be called a **definition**.

- Imports

An **import** is the use of or reference to the name of a procedure or data item not defined in the current module object. The import is identified by its name and its associated type, either procedure or data.

An import can also be called a **reference**.

A module object is the basic building block of an ILE runnable object. Therefore, when a module object is created, the following may also be generated:

- Debug data

Debug data is the data necessary for debugging a running ILE object. This data is optional.

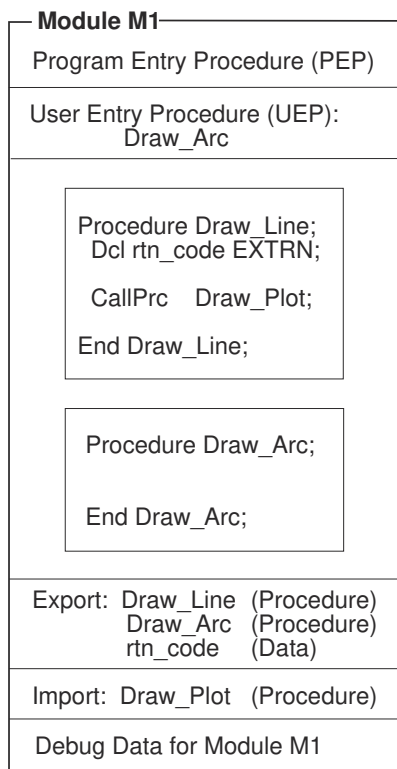
- Program entry procedure (PEP)

A **program entry procedure** is the compiler-generated code that is the entry point for an ILE program on a dynamic program call. It is similar to the code provided for the entry point in an OPM program.

- User entry procedure (UEP)

A **user entry procedure**, written by a programmer, is the target of the dynamic program call. It is the procedure that gets control from the PEP. The main() function of a C program becomes the UEP of that program in ILE.

Figure 7 on page 17 shows a conceptual view of a module object. In this example, module object M1 exports two procedures (Draw_Line and Draw_Arc) and a data item (rtn_code). Module object M1 imports a procedure called Draw_Plot. This particular module object has a PEP, a corresponding UEP (the procedure Draw_Arc), and debug data.



RV3W104-0

Figure 7. Conceptual View of a Module

Characteristics of a *MODULE object:

- A *MODULE object is the output from an ILE compiler.
- It is the basic building block for ILE runnable objects.
- It is not a runnable object.
- It may have a PEP defined.
- If a PEP is defined, a UEP is also defined.
- It can export procedure and data item names.
- It can import procedure and data item names.
- It can have debug data defined.

ILE Program

An ILE program shares the following characteristics with an OPM program:

- The program gets control through a dynamic program call.
- There is only one entry point to the program.
- The program is identified to the system by the symbol *PGM.

An ILE program has the following characteristics that an OPM program does not have:

- An ILE program is created from one or more copied module objects.
- One or more of the copied modules can contain a PEP.
- You have control over which module's PEP is used as the PEP for the ILE program object.

When the Create Program (CRTPGM) command is specified, the ENTMOD parameter allows you to select which module containing a PEP is the program's entry point.

A PEP that is associated with a module that is not selected as the entry point for the program is ignored. All other procedures and data items of the module are used as specified. Only the PEP is ignored.

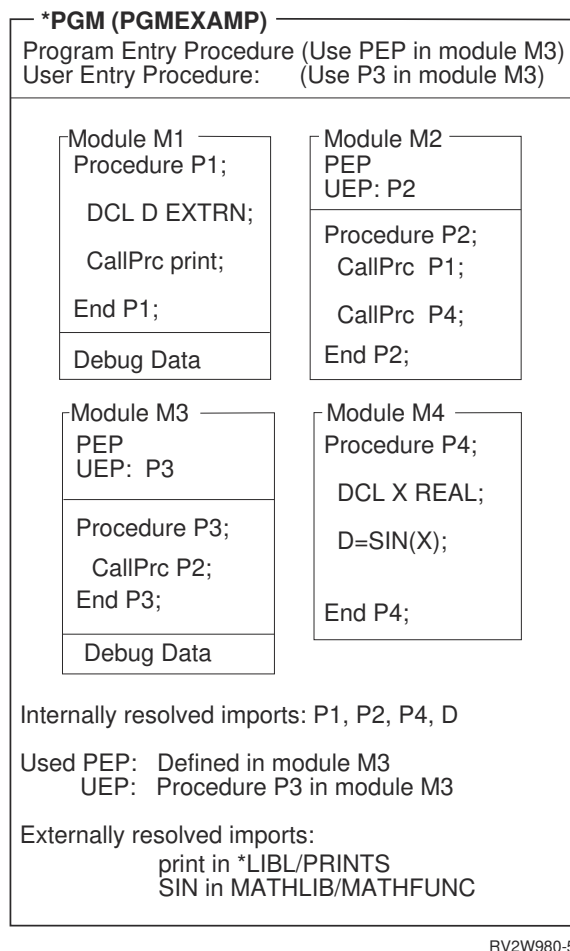
When a dynamic program call is made to an ILE program, the module's PEP that was selected at program-creation time is given control. The PEP calls the associated UEP.

When an ILE program object is created, only those procedures associated with the copied modules containing debug data can be debugged by the ILE debugger. The debug data does not affect the performance of a running ILE program.

Figure 8 on page 18 shows a conceptual view of an ILE program object. When the program PGMEXAMP is called, the PEP of the program, which was defined in the copied module object M3, is given control. The copied module M2 also has a PEP defined, but it is ignored and never used by the program.

In this program example, only two modules, M1 and M3, have the necessary data for the ILE debugger. Procedures from modules M2 and M4 cannot be debugged with the ILE debugger.

The imported procedures `print` and `SIN` are resolved to exported procedures from service programs `PRINTS` and `MATHFUNC`, respectively.



RV2W980-5

Figure 8. Conceptual View of an ILE Program

Characteristics of an ILE *PGM object:

- One or more modules from any ILE language are copied to make the *PGM object.
- The person who creates the program has control over which module's PEP becomes the only PEP for the program.
- On a dynamic program call, the module's PEP that was selected as the PEP for the program gets control to run.
- The UEP associated with the selected PEP is the user's entry point for the program.
- Procedures and data item names cannot be exported from the program.

- Procedures or data item names can be imported from modules and service programs but not from program objects. For information on service programs, see [“Service Program” on page 19](#).
- Modules can have debug data.
- A program is a runnable object.

Service Program

A **service program** is a collection of runnable procedures and available data items easily and directly accessible by other ILE programs or service programs. In many respects, a service program is similar to a subroutine library or procedure library.

Service programs provide common services that other ILE objects might need; hence the name service program. An example of a set of service programs provided by the operating system are the runtime procedures for a language. These runtime procedures often include such items as mathematical procedures and common input and output procedures.

The **public interface** of a service program consists of the names of the exported procedures and data items accessible by other ILE objects. Only those items that are exported from the module objects making up a service program are eligible to be exported from a service program.

The programmer can specify which procedures or data items can be known to other ILE objects. Therefore, a service program can have hidden or private procedures and data that are not available to any other ILE object.

It is possible to update a service program without having to re-create the other ILE programs or service programs that use the updated service program. The programmer making the changes to the service program controls whether the change is compatible with the existing support.

The way that ILE provides for you to control compatible changes is by using the **binder language**. The binder language allows you to define the list of procedure names and data item names that can be exported. A **signature** is generated from the names of procedures and data items and from the order in which they are specified in the binder language. To make compatible changes to a service program, new procedure or data item names should be added to the end of the export list. For more information on signatures, the binder language, and protecting your customers' investment in your service programs, see [“Binder Language” on page 73](#).

[Figure 9 on page 20](#) shows a conceptual view of a service program. Notice that the modules that make up that service program are the same set of modules that make up ILE program object PGMEXAMP in [Figure 8 on page 18](#). The previous signature, Sigyy, for service program SPGMEXAMP contains the names of procedures P3 and P4. After an upward-compatible change is made to the service program, the current signature, Sigxx, contains not only the names of procedures P3 and P4; it also contains the name of data item D. Other ILE programs or service programs that use procedures P3 or P4 do not have to be re-created.

Although the modules in a service program may have PEPs, these PEPs are ignored. The service program itself does not have a PEP. Therefore, unlike a program object, a service program cannot be the target of a dynamic program call.

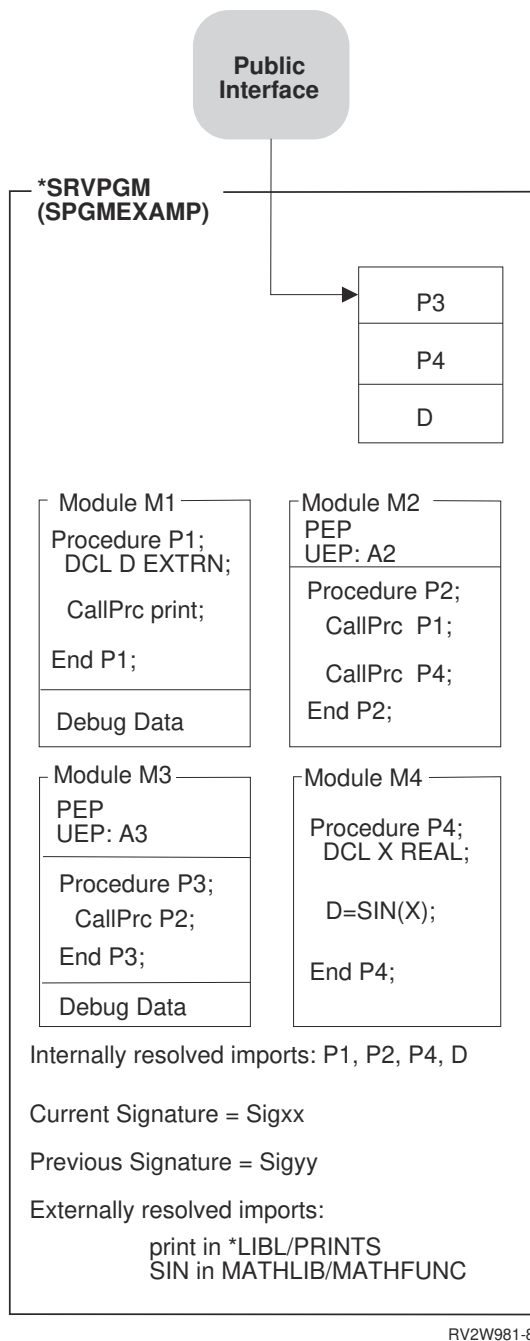


Figure 9. Conceptual View of an ILE Service Program

Characteristics of an ILE *SRVPGM object:

- One or more modules from any ILE language are copied to make the *SRVPGM object.
- No PEP is associated with the service program. Because there is no PEP, a dynamic program call to a service program is not valid. A module's PEP is ignored.
- Other ILE programs or service programs can use the exports of this service program identified by the public interface.
- Signatures are generated from the procedure and data item names that are exported from the service program.
- Service programs can be replaced without affecting the ILE programs or service programs that use them, as long as previous signatures are still supported.
- Modules can have debug data.

- A service program is a collection of data items and runnable procedures.
- Weak data can be exported only to an activation group. It cannot be made part of the public interface that is exported from the service program. For information about weak data, see Export in [“Import and Export Concepts”](#) on page 72.

Binding Directory

A **binding directory** lists the names of modules and service programs that you may need when creating an ILE program or service program. 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. A binding directory is a system object that is identified to the system by the symbol *BNDDIR.

Binding directories are optional. The reasons for using binding directories are convenience and program size.

- They offer a convenient method of listing the modules or service programs that you may need when creating your own ILE program or service program. For example, one binding directory may list all the modules and service programs that provide math functions. If you want to use some of those functions, you specify only the one binding directory, not each module or service program you use.

Note: The more modules or service programs a binding directory lists, the longer it may take to bind the programs. Therefore, you should include only the necessary modules or service programs in your binding directory.

- Binding directories can reduce program size because you do not specify modules or service programs that do not get used.

Very few restrictions are placed on the entries in a binding directory. The name of a module or service program can be added to a binding directory even if that object does not yet exist.

For a list of CL commands used with binding directories, see [“CL Commands Used with ILE Objects”](#) on page 187.

Figure 10 on page 21 shows a conceptual view of a binding directory.

Binding Directory (ABD)

Object Name	Object Type	Object Library
QALLOC	*SRVPGM	*LIBL
QMATH	*SRVPGM	QSYS
QFREE	*MODULE	*LIBL
QHFREE	*SRVPGM	ABC
▪	▪	▪
▪	▪	▪
▪	▪	▪

RV2W982-0

Figure 10. Conceptual View of a Binding Directory

Characteristics of a *BNDDIR object:

- Convenient method of grouping the names of service programs and modules that may be needed to create an ILE program or service program.
- Because binding directory entries are just names, the objects listed do not have to exist yet on the system.
- The only valid library names are *LIBL or a specific library.
- The objects in the list are optional. The named objects are used only if any unresolved imports exist and if the named object provides an export to satisfy the unresolved import request.

Binding Directory Processing

During binding, processing happens in this order:

1. All of the modules specified on the MODULE parameter are examined. The binder determines the list of symbols imported and exported by the object. After being examined, modules are bound, in the order listed, into the program being created.
2. All of the service programs on the BNDSRVPGM parameter are examined in the order listed. The service programs are bound only if needed to resolve an import.
3. All of the binding directories on the BNDDIR parameter are processed in the order listed. All the objects listed in these binding directories are examined in the order listed, but they are bound only if needed to resolve imports. Duplicate entries in binding directories are silently ignored.
4. Each module has a list of **reference system objects**. This list is simply a list of binding directories. The reference system objects from bound modules are processed in order such that all the reference system objects from the first module are processed first, then the objects from the second module, and so on. The objects listed in these binding directories are examined in the order listed, only as needed, and bound only if needed. This processing continues only as long as unresolved imports exist, even if OPTION(*UNRSLVREF) is used. In other words, processing objects stops when all imports are resolved.

While objects are examined, message CPD5D03, "Definition supplied multiple times for symbol", may be signalled even if the object is not ultimately bound into the program being created.

Note that modules usually have imports that are not apparent from the module's source code. These are added by the compiler to implement various language features that require runtime support from service programs. Use DSPMOD DETAIL(*IMPORT) to see these imports.

To see the list of imported and exported symbols for a module or service program, look at the Binder Information Listing section of a CRTPGM or CRTSRVPGM DETAIL(*EXTENDED) listing. It lists the objects that are examined during the binding.

Module or service program objects that are bound into the program or service program being created are indicated in the Binder Information Listing section of a CRTPGM or CRTSRVPGM DETAIL(*EXTENDED) listing. Once an object is created, you can also use the DSPPGM or DSPSRVPGM command DETAIL(*MODULE) to see the bound *MODULE objects, and DETAIL(*SRVPGM) to see the list of bound *SRVPGM objects.

You can use DSPMOD DETAIL(*REFSYSOBJ) to see the list of reference system objects, which are binding directories. These binding directories typically list the names of service program APIs supplied by the operating system or language runtime support. In this way, a module can be bound to its language runtime support and system APIs without the programmer having to specify anything special on the command.

Binder Functions

The function of the binder is similar to, but somewhat different from, the function provided by a linkage editor. The **binder** processes import requests for procedure names and data item names from specified modules. The binder then tries to find matching exports in the specified modules, service programs, and binding directories.

In creating an ILE program or service program, the binder performs the following types of binding:

- Bind by copy

To create the ILE program or service program, the following are copied:

The modules specified on the module parameter

Any modules selected from the binding directory that provide an export for an unresolved import

Physical addresses of the needed procedures and data items used within the copied modules are established when the ILE program or service program is created.

For example, in [Figure 9 on page 20](#), procedure P3 in module M3 calls procedure P2 in module M2. The physical address of procedure P2 in module M2 is made known to procedure M3 so that address can be directly accessed.

- Bind by reference

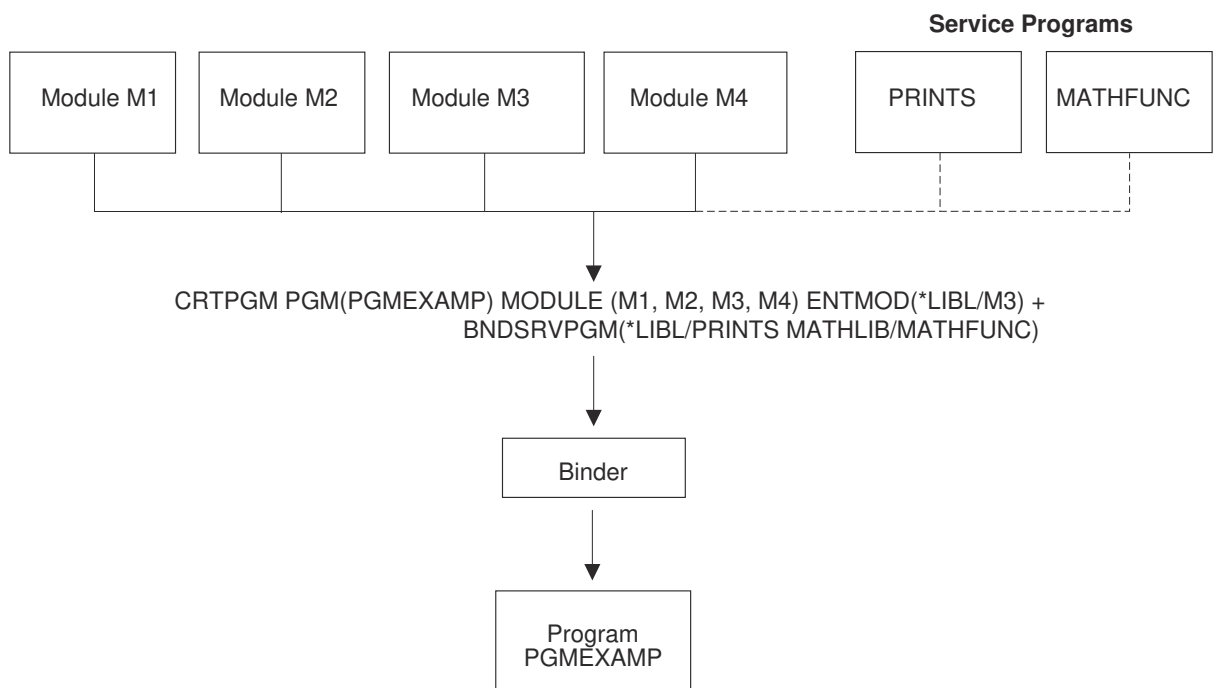
Symbolic links to the service programs that provide exports for unresolved import requests are saved in the created program or service program. The symbolic links refer to the service programs providing the exports. The links are converted to physical addresses when the program object to which the service program is bound is activated.

[Figure 9 on page 20](#) shows an example of a symbolic link to SIN in service program *MATHLIB/MATHFUNC. The symbolic link to SIN is converted to a physical address when the program object to which service program SPGMEXAMP is bound is activated.

At runtime, with physical links established to the procedures and data items being used, there is little performance difference between the following:

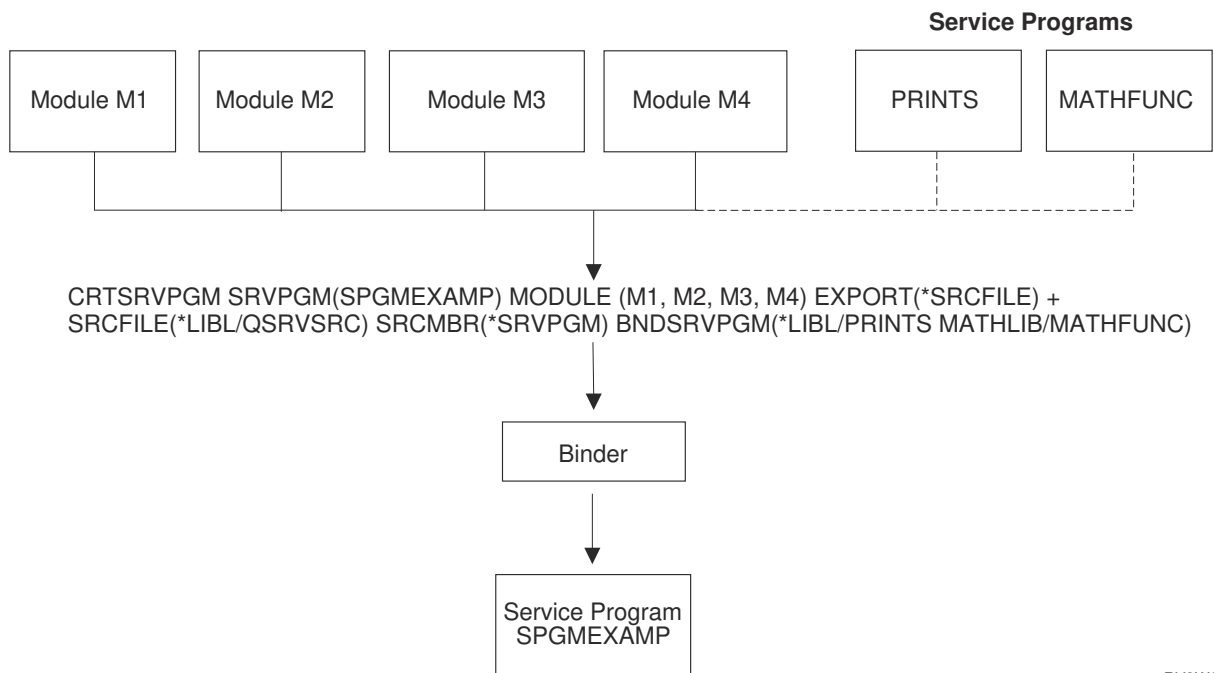
- Accessing a local procedure or data item
- Accessing a procedure or data item in a different module or service program bound to the same program

[Figure 11 on page 23](#) and [Figure 12 on page 24](#) show conceptual views of how the ILE program PGMEXAMP and service program SPGMEXAMP were created. The binder uses modules M1, M2, M3, and M4 and service programs PRINTS and MATHFUNC to create ILE program PGMEXAMP and service program SPGMEXAMP.



RV2W983-3

Figure 11. Creation of an ILE Program



RV3W030-1

Figure 12. Creation of a Service Program

For additional information on creating an ILE program or service program, see [“Program Creation Concepts”](#) on page 61.

Calling a Program or a Procedure

In ILE you can call either a program or a procedure. ILE requires that the caller identify whether the target of the call statement is a program or a procedure. ILE languages communicate this requirement by having separate call statements for programs and for procedures. Therefore, when you write your ILE program, you must know whether you are calling a program or a procedure.

Each ILE language has unique syntax that allows you to distinguish between a dynamic program call and a static procedure call. The standard call statement in each ILE language defaults to either a dynamic program call or a static procedure call. For RPG and COBOL the default is a dynamic program call, and for C the default is a static procedure call. Thus, the standard language call performs the same type of function in either OPM or ILE. This convention makes migrating from an OPM language to an ILE language relatively easy.

To determine how long your procedure names can be, see your ILE HLL programmer’s guide.

Dynamic Program Calls

A dynamic program call transfers control to either an ILE program object or an OPM program object, but not to an ILE service program. Dynamic program calls include the following:

- An OPM program can call another OPM program or an ILE program
- An ILE program can call an OPM program or another ILE program
- A service program can call an OPM program or an ILE program

Static Procedure Calls

A static procedure call transfers control to an ILE procedure. Static procedure calls can be coded only in ILE languages. A static procedure call can be used to call any of the following:

- A procedure within the same module
- A procedure in a separate module within the same ILE program or service program

- A procedure in a separate ILE service program

Figure 13 on page 25 shows examples of static procedure calls. The figure shows that:

- A procedure in an ILE program can call an exported procedure in the same program or in a service program. Procedure P1 in program A calls procedure P2 in another copied module. Procedure P3 in program C calls procedure P4 in service program D.
- A procedure in a service program can call an exported procedure in the same service program or in another service program. Procedure P6 in service program B calls procedure P7 in another copied module. Procedure P5 in service program E calls procedure P4 in service program F.

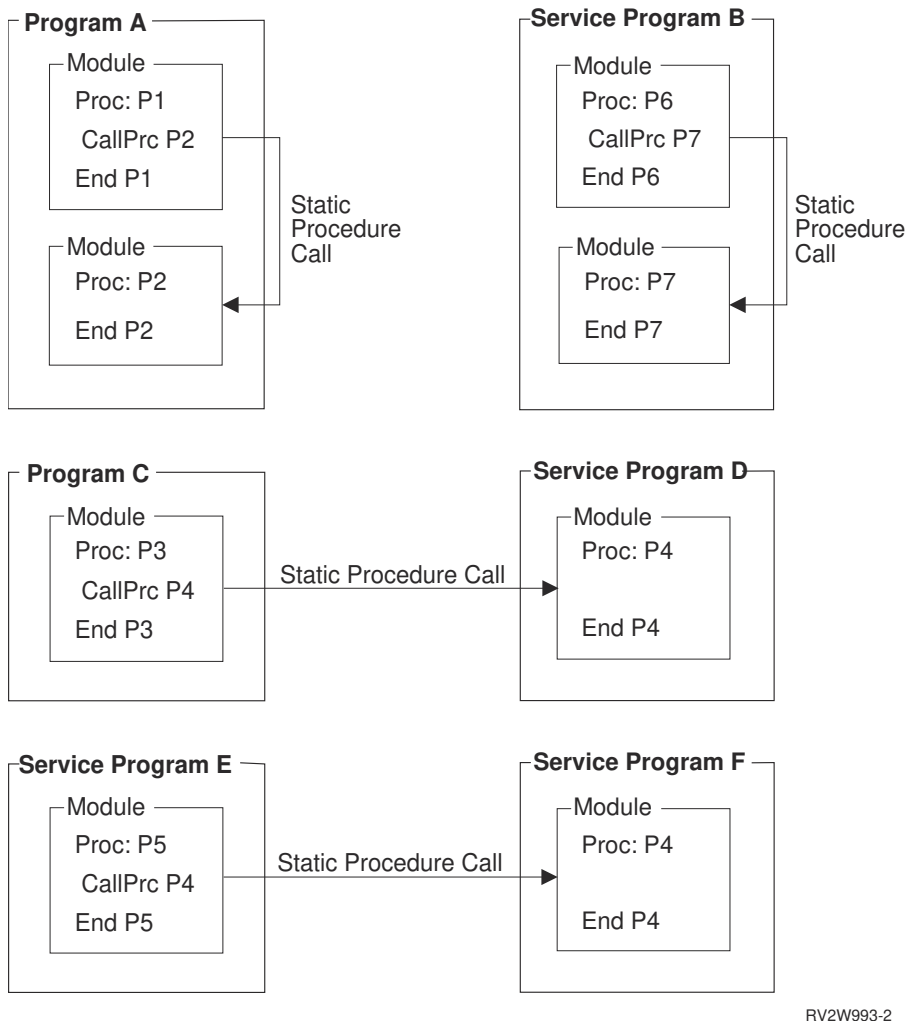


Figure 13. Static Procedure Calls

Procedure Pointer Calls

See [“Calls to Procedures and Programs” on page 97](#) for information on procedure pointer calls.

Activation

After successfully creating an ILE program, you will want to run your code. The process of getting a program or service program ready to run is called *activation*. You do not have to issue a command to activate a program. Activation is done by the system when a program is called.

Activation performs the following functions:

- Allocates and initializes the static data needed by the program or service program
- Resolves imports to runtime addresses of the corresponding service program exports.

A program or service program can be activated in more than one activation group, even within the same job. Each activation is local to a particular activation group, and each activation has its own static storage. When a program or service program is used concurrently by many jobs, only one copy of that object's instructions resides in storage, but the static variables are separate for each activation.

By default, service programs are immediately activated during the call to a program that directly or indirectly requires their services. You can request deferred activation for a service program when you are binding an ILE program or a service program that is created for V6R1, or later. If you request deferred activation for a service program, the activation of a service program can be deferred until one of its imported procedures is called. To minimize activation costs both at program startup and throughout program execution, it is suggested that you specify deferred activation for the service programs that satisfy procedure imports and that are used only on infrequently traveled code paths.

Note:

1. If you request deferred activation for a service program that satisfies a data import, partial immediate activation is required to initialize the static data.
2. If you request deferred activation for a service program that satisfies a procedure import for a procedure pointer call, partial immediate activation is required to provide the binding for the procedure pointer call.

To specify the activation mode for a service program as either immediate or deferred, use `*IMMED` or `*DEFER` on the `BNDSRVPGM` parameter of the following CL commands:

- Create Program (CRTPGM)
- Create Service Program (CRTSRVPGM)
- Update Program (UPDPGM)
- Update Service Program (UPDSRVPGM)

The Add Binding Directory (ADDBNDDIRE) command provides a similar input field for a service program entry, and the Work with Binding Directory Entries (WRKBNDDIRE) command provides output of the activation mode of service program entries in a binding directory.

If either of the following is true:

- Activation cannot find the needed service program
- The service program no longer supports the procedures or data items represented by the signature

an error occurs and you cannot run your application.

For more details on program activation, refer to [“Program Activation Creation” on page 29](#).

When activation allocates the storage necessary for the static variables used by a program, the space is allocated from an activation group. At the time the program or service program is created, you can specify the activation group that should be used at runtime.

For more information on activation groups, refer to [“Activation Group” on page 30](#).

Error Handling Overview

Figure 14 on page 27 shows the complete error-handling structure for both OPM and ILE programs. This figure is used throughout this manual to describe advanced error-handling capabilities. This topic gives a brief overview of the standard language error-handling capabilities. For additional information on error handling, refer to [“Error Handling” on page 39](#).

The figure shows a fundamental layer called exception-message architecture. An exception message may be generated by the system whenever an OPM program or an ILE program encounters an error. Exception messages are also used to communicate status information that may not be considered a program error. For example, a condition that a database record is not found is communicated by sending a status exception message.

Each high-level language defines language-specific error-handling capabilities. Although these capabilities vary by language, in general it is possible for each HLL user to declare the intent to handle

specific error situations. The declaration of this intent includes identification of an error-handling routine. When an exception occurs, the system locates the error-handling routine and passes control to user-written instructions. You can take various actions, including ending the program or recovering from the error and continuing.

Figure 14 on page 27 shows that ILE uses the same exception-message architecture that is used by OPM programs. Exception messages generated by the system initiate language-specific error handling within an ILE program just as they do within an OPM program. The lowest layer in the figure includes the capability for you to send and receive exception messages. This can be done with message handler APIs or commands. Exception messages can be sent and received between ILE and OPM programs.

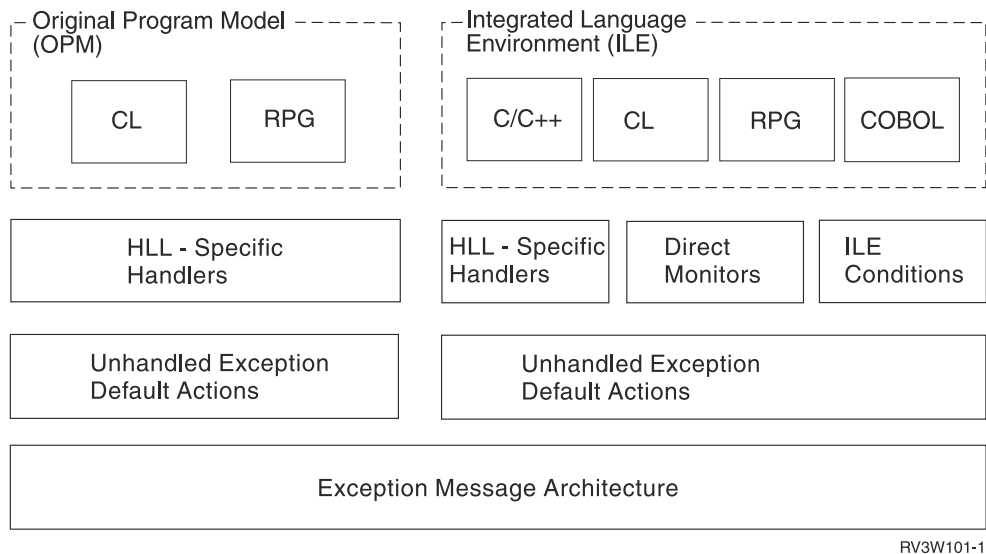


Figure 14. Error Handling for OPM and ILE

Language-specific error handling works similarly for ILE programs as for OPM programs, but there are basic differences:

- When the system sends an exception message to an ILE program, the procedure and module name are used to qualify the exception message. If you send an exception message, these same qualifications can be specified. When an exception message appears in the job log for an ILE program, the system normally supplies the program name, module name, and procedure name.
- Extensive optimization for ILE programs can result in multiple HLL statement numbers associated with the same generated instructions. As the result of optimization, exception messages that appear in the job log may contain multiple HLL statement numbers.

Additional error-handling capabilities are described in [“Error Handling” on page 39](#).

Optimizing Translator

Optimization means maximizing the runtime performance of the object. All ILE languages have access to the optimization techniques provided by the ILE optimizing translator. Generally, the higher the optimization level, the longer it takes to create the object. At runtime, highly optimized programs or service programs should run faster than corresponding programs or service programs created with a lower level of optimization.

Although optimization can be specified for a module, program object, and service program, the optimization techniques apply to individual modules. The levels of optimization are:

- 10 or *NONE
- 20 or *BASIC
- 30 or *FULL
- 40 (more optimization than level 30)

For performance reasons, it is probably desirable to use a high level of optimization when a program is put in production. For initial testing, it might be necessary to use a lower optimization level because of debugging limitations. However, it is strongly suggested that you use the optimization level at which a program is released for final testing because some bugs, such as uninitialized data, might only be exposed at higher optimization levels.

Because optimization at level 30 (*FULL) or level 40 can significantly affect your program instructions, you may need to be aware of certain debugging limitations and different addressing exception detection. Refer to [“Debugging Considerations” on page 119](#) for debug considerations. Refer to [“Exceptions in Optimized Programs” on page 185](#) for addressing error considerations.

Debugger

ILE provides a debugger that allows source-level debugging. The debugger can work with a listing file and allow you to set breakpoints, display variables, and step into or over an instruction. You can do these without ever having to enter a command from the command line. A command line is also available while working with the debugger.

The source-level debugger uses system-provided APIs to allow you to debug your program or service program. These APIs are available to everyone and allow you to write your own debugger.

The debuggers for OPM programs continue to exist on the operating system but can be used to debug only OPM programs. However, the ILE debugger can debug OPM programs that are compiled with either `OPTION(*SRCDBG)` or `OPTION(*LSTDBG)`.

Debugging an optimized program can be difficult. When you use the ILE debugger to view or change a variable that is used by a running program or procedure, the debugger retrieves or updates the data in the storage location for that variable. At level 20 (*BASIC), 30 (*FULL), or 40 optimization, the current value of a data variable might not be in storage, so the debugger cannot access it. Thus, the value displayed for a variable might not be the current value. For this reason, you should use optimization level 10 (*NONE) to create modules during development. Then, for best performance, you should use optimization level 30 (*FULL) or 40 when you create modules for final testing before a program is put into production.

For more information on the ILE debugger, see [“Debugging Considerations” on page 119](#).

ILE Advanced Concepts

This topic describes advanced concepts for the ILE model. Before reading this, you should be familiar with the concepts described in [“ILE Basic Concepts”](#) on page 15.

Program Activation

Activation is the process used to prepare a program to run. Both ILE programs and service programs must be activated by the system before they can be run.

Program activation includes two major steps:

1. Allocate and initialize static storage for the program.
2. Complete the binding of programs to service programs.

This topic concentrates on step 1. Step 2 is explained in [“Service Program Activation”](#) on page 35.

Figure 15 on page 29 shows a program object that is stored in permanent disk storage. As with all IBM i objects, program objects can be shared by multiple concurrent users running in different jobs and only one copy of the program's instructions exists. However, when a program is activated, storage for program variables must be allocated and initialized.

As shown in [Figure 15 on page 29](#), each program activation has its own copy of these variables.

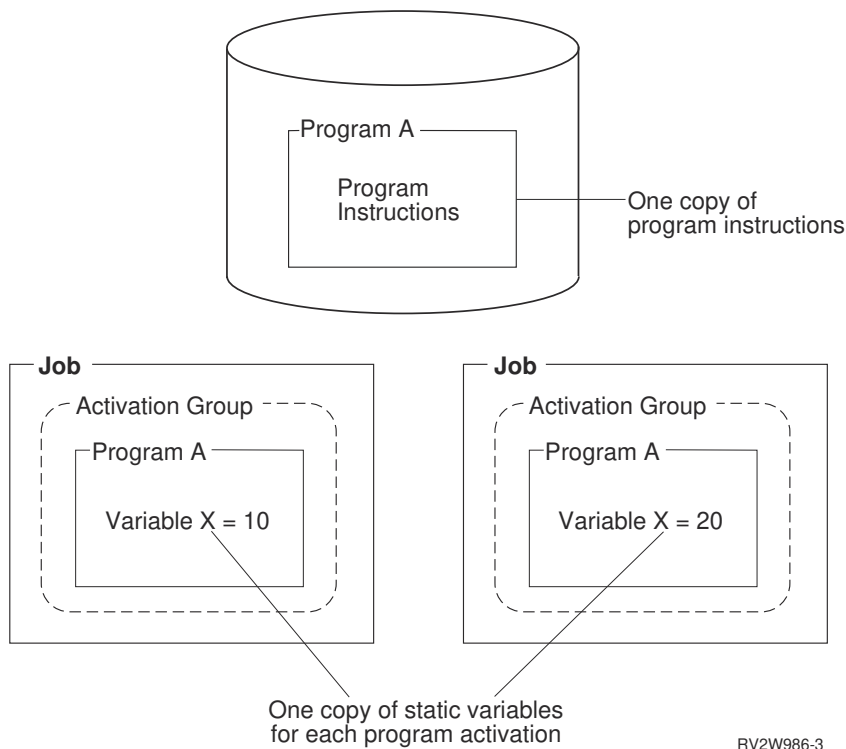


Figure 15. One Copy of Static Variables for Each Program Activation

Program Activation Creation

ILE manages the process of program activation by keeping track of program activations within an activation group. Refer to [“Activation Group”](#) on page 30 for a definition of an activation group. Only one activation for a particular program object is in an activation group. Programs of the same name residing in different libraries are considered different program objects when this rule is applied.

When you use a dynamic program call statement in your HLL program, ILE uses the activation group that was specified when the program was created. This attribute is specified by using the activation group (ACTGRP) parameter on either the Create Program (CRTPGM) command or the Create Service Program (CRTSRVPGM) command. If a program activation already exists within the activation group indicated with this parameter, it is used. If the program has never been activated within this activation group, it is activated first and then run. If there is a named activation group, the name can be changed with the ACTGRP parameter on the UPDPM and UPDSRVPM commands

Once a program is activated, it remains activated until the activation group is deleted. As a result of this rule, it is possible to have active programs that are not on the call stack. Figure 16 on page 30 shows an example of three active programs within an activation group, but only two of the three programs have procedures on the call stack. In this example, program A calls program B, causing program B to be activated. Program B then returns to program A. Program A then calls program C. The resulting call stack contains procedures for programs A and C but not for program B. For a discussion of the call stack, see “Call Stack” on page 97.

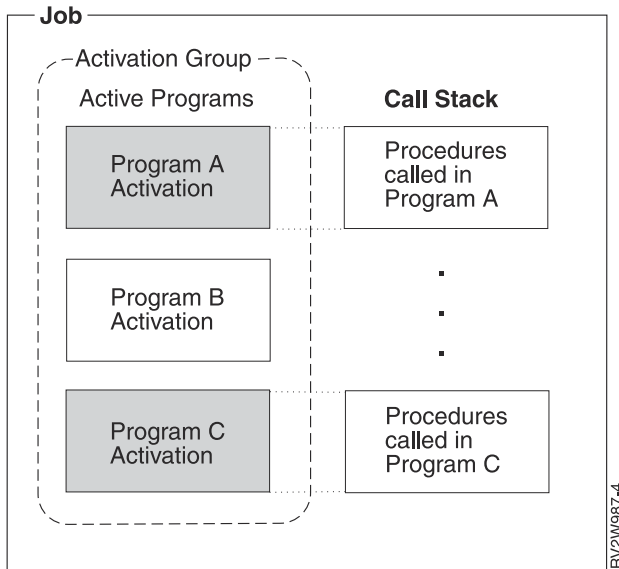


Figure 16. Program May Be Active But Not on the Call Stack

Activation Group

All ILE programs and service programs are activated within a substructure of a job called an **activation group**. This substructure contains the resources necessary to run the programs. These resources fall into the following general categories:

- Static program variables
- Dynamic storage
- Temporary data management resources
- Certain types of exception handlers and ending procedures

Activation groups use either single-level storage or teraspace for supplying storage for static program variables. For more information, see “Teraspace and Single-Level Storage” on page 49. When single-level storage is used, the static program variables and dynamic storage are assigned separate address spaces for each activation group, which provides some degree of program isolation and protection from accidental access. When teraspace is used, the static program variables and dynamic storage may be assigned separate address ranges within teraspace, which provides a lesser degree of program isolation and protection from accidental access.

The temporary data management resources include the following:

- Open files (open data path or ODP)
- Commitment definitions

- Local SQL cursors
- Remote SQL cursors
- Hierarchical file system (HFS)
- User interface manager
- Query management instances
- Open communications links
- Common Programming Interface (CPI) communications

The separation of these resources among activation groups supports a fundamental concept. That is, the concept that all programs activated within one activation group are developed as one cooperative application.

Software vendors may select different activation groups to isolate their programs from other vendor applications running in the same job. This vendor isolation is shown in [Figure 17 on page 31](#). In this figure, a complete customer solution is provided by integrating software packages from four different vendors. Activation groups increase the ease of integration by isolating the resources associated with each vendor package.

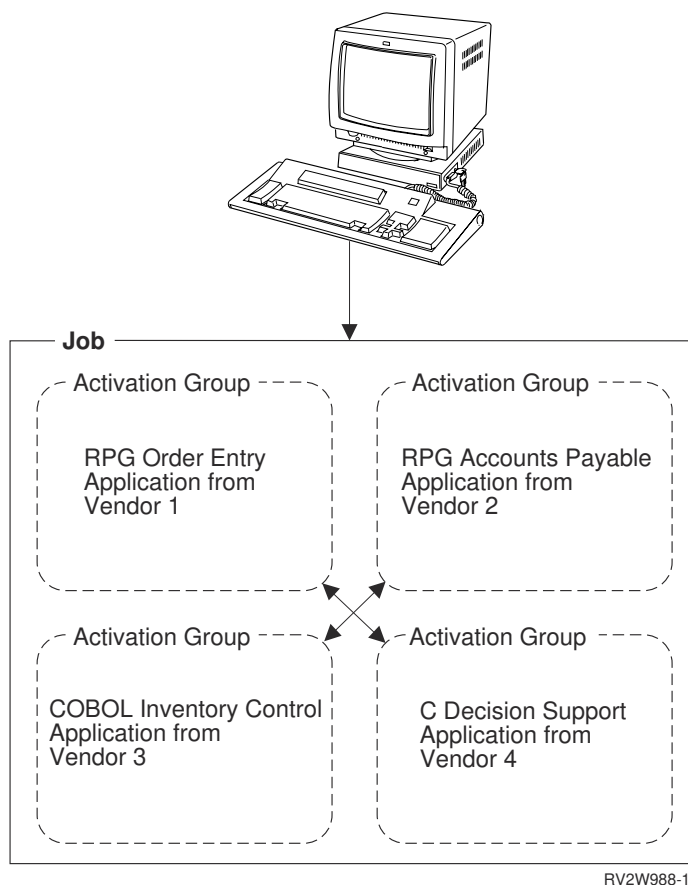


Figure 17. Activation Groups Isolate Each Vendor's Application

There is a significant consequence of assigning the above resources to an activation group. The consequence is that when an activation group is deleted, all of the above resources are returned to the system. The temporary data management resources left open at the time the activation group is deleted are closed by the system. The storage for static and dynamic storage that has not been deallocated is returned to the system.

Activation Group Creation

You can control the runtime creation of a non-default activation group by specifying an activation group attribute when you create your program or service program. The attribute is specified by using the

ACTGRP parameter on the CRTPGM command or CRTSRVPGM command. There is no Create Activation Group command.

All ILE programs have one of the following activation group attributes:

- A user-named activation group

Specified with the ACTGRP(name) parameter. This attribute allows you to manage a collection of ILE programs and service programs as one application. The activation group is created when it is first needed. It is then used by all programs and service programs that specify the same activation group name.

- A system-named activation group

Specified with the ACTGRP(*NEW) parameter on the CRTPGM command. This attribute allows you to create a new activation group whenever the program is called. ILE selects a name for this activation group. The name assigned by ILE is unique within your job. The name assigned to a system-named activation group does not match any name you choose for a user-named activation group. Service programs do not support this attribute.

- An attribute to use the activation group of the calling program

Specified with the ACTGRP(*CALLER) parameter. This attribute allows you to create an ILE program or service program that will be activated within the activation group of the calling program. With this attribute, a new activation group is never created when the program or service program is activated.

- An attribute to choose the activation group appropriate to the programming language and storage model.

Specified with the ACTGRP(*ENTMOD) parameter on the CRTPGM command. When ACTGRP(*ENTMOD) is specified, the program entry procedure module specified by the ENTMOD parameter is examined. One of the following occurs:

- If the module attribute is RPGLE, CBLLE, or CLLE, and
 - if STGM DL(*SNG LVL) is specified, then QILE is used as the activation group.
 - if STGM DL(*TERASPACE) is specified, then QILETS is used as the activation group.
- If the module attribute is not RPGLE, CBLLE, or CLLE, then *NEW is used as the activation group.

ACTGRP(*ENTMOD) is the default value for this parameter of the CRTPGM command.

All activation groups within a job have a name. Once an activation group exists within a job, it is used to activate programs and service programs that specify that name. As a result of this design, duplicate activation group names cannot exist within one job.

The ACTGRP parameter on the UPDPGM and UPDSRVPGM commands can be used to change the activation group into which the program or service program is activated.

Default Activation Groups

When a job is started, the system creates two activation groups to be used by all OPM programs. One of these activation groups is used for application programs. The other is used for operating system programs. These OPM default activation groups use single-level storage for static program variables. You cannot delete the OPM default activation groups. They are deleted by the system when your job ends.

ILE programs and service programs can be activated in the OPM default activation groups if the following conditions are satisfied:

- The ILE programs or service programs were created with the activation group *CALLER option or with the DFTACTGRP(*YES) option.

Note: The DFTACTGRP(*YES) option is only available on the CRTBNDCL (ILE CL) and CRTBNDRPG (ILE RPG) commands.

- The call to the ILE programs or service programs originates in the OPM default activation groups.
- The ILE program or service program does not use the teraspace storage model.

The operating system will also create a teraspace default activation group when it determines one is needed. The teraspace default activation group uses teraspace storage for static program variables. You cannot delete the teraspace default activation group. It will be deleted by the system when your job ends. ILE programs and service programs can be activated in the teraspace default activation group if the following conditions are satisfied:

- The ILE program or service program was created with the activation group *CALLER option.
- The state of the ILE program or service program is *USER.

One of the following conditions must also be satisfied for the ILE program or service program to be activated into the teraspace default activation group:

- The call to the ILE program or service program originates in the teraspace default activation group and the ILE program or service program was created with either the storage model *INHERIT or the storage model *TERASPACE option.
- The ILE program or service program was created with the storage model *INHERIT option, there are no application entries on the call stack associated with a different activation group, and the activation occurs in preparation for one of these invocations:
 - SQL stored procedure
 - SQL function
 - SQL trigger

Note: Starting with IBM i 7.1, SQL procedures, functions and triggers are created with storage model *INHERIT. On previous releases, SQL procedures, functions, and triggers were created with storage model *SNGLVL.

- The ILE program or service program was created with the storage model *TERASPACE option and there are no call stack entries associated with a teraspace storage model activation group. See [“Selecting a Compatible Activation Group”](#) on page 51 for additional information.

The static and heap storage used by ILE programs activated in one of the default activation groups are not returned to the system until the job ends. Similarly, temporary data management resources associated with any default activation group are normally scoped to a job. For example, normally, open files are not closed by the system until the job ends; see [“Reclaim Resources Command for ILE Programs”](#) on page 94 for more information.

Non-Default Activation Group Deletion

Activation groups require resources to be created within a job. Processing time may be saved if an activation group can be reused by an application. ILE provides several options to allow you to return from an invocation without ending or deleting the associated activation group. Whether the activation group is deleted depends on the type of activation group and the method in which the application ended.

An application may return to a call stack entry (see [“Call Stack”](#) on page 97) associated with another activation group in the following ways:

- HLL end verbs

For example, STOP RUN in COBOL or exit() in C.

- Call to API CEETREC
- Unhandled exceptions

Unhandled exceptions can be moved by the system to a call stack entry in another activation group.

- Language-specific HLL return statements

For example, a return statement in C, an EXIT PROGRAM statement in COBOL, or a RETURN statement in RPG.

- Skip operations

For example, sending an exception message or branching to a call stack entry that is not associated with your activation group.

You can delete an activation group from your application by using HLL end verbs or by calling API CEETREC. An unhandled exception can also cause your activation group to be deleted. These operations will always delete your activation group, provided the nearest control boundary is the oldest call stack entry associated with the activation group (sometimes called a *hard control boundary*). If the nearest control boundary is not the oldest call stack entry (sometimes called a *soft control boundary*), control passes to the call stack entry prior to the control boundary. However, the activation group is not deleted.

A control boundary is a call stack entry that represents a boundary to your application. ILE defines control boundaries whenever you call between activation groups. Refer to [“Control Boundaries”](#) on page 37 for a definition of a control boundary.

A user-named activation group may be left in the job for later use. For this type of activation group, any normal return or skip operation past a hard control boundary does not delete the activation group. In contrast, use of those same operations within a system-named activation group causes the activation group to be deleted. System-named activation groups are deleted because you cannot reuse them by specifying the system-generated name. For language-dependent rules about a normal return from the oldest call stack entry associated with an activation group, refer to the ILE HLL programmer’s guides.

Figure 18 on page 34 shows examples of how to leave an activation group. In the figure, procedure P1 is the oldest call stack entry. For the system-named activation group (created with the ACTGRP(*NEW) option), a normal return from P1 deletes the associated activation group. For the user-named activation group (created with the ACTGRP(name) option), a normal return from P1 does not delete the associated activation group.

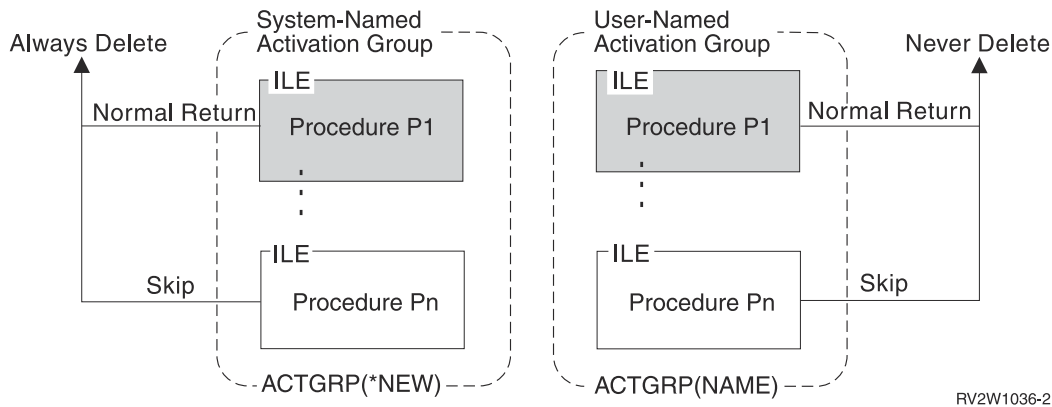


Figure 18. Leaving User-Named and System-Named Activation Groups

If a user-named activation group is left in the job, you can delete it by using the Reclaim Activation Group (RCLACTGRP) command. This command allows you to delete named activation groups after your application has returned. Only activation groups that are not in use can be deleted with this command.

Figure 19 on page 35 shows a job with one activation group that is not in use and one activation group that is currently in use. An activation group is considered in use if there are call stack entries associated with programs activated within that activation group. Using the RCLACTGRP command in program A or program B deletes the activation group for program C and program D.

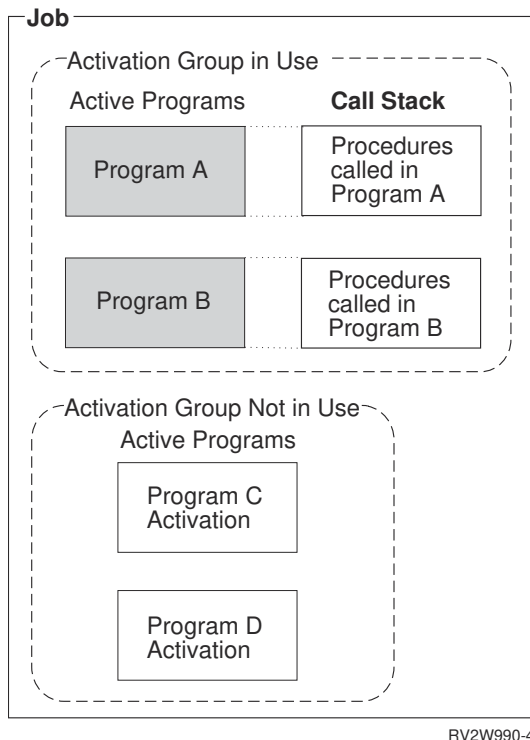


Figure 19. Activation Groups In Use Have Entries on the Call Stack

When an activation group is deleted by ILE, certain end-operation processing occurs. This processing includes calling user-registered exit procedures, data management cleanup, and language cleanup (such as closing files). Refer to [“Data Management Scoping Rules”](#) on page 45 for details on the data management processing that occurs when an activation group is deleted.

Service Program Activation

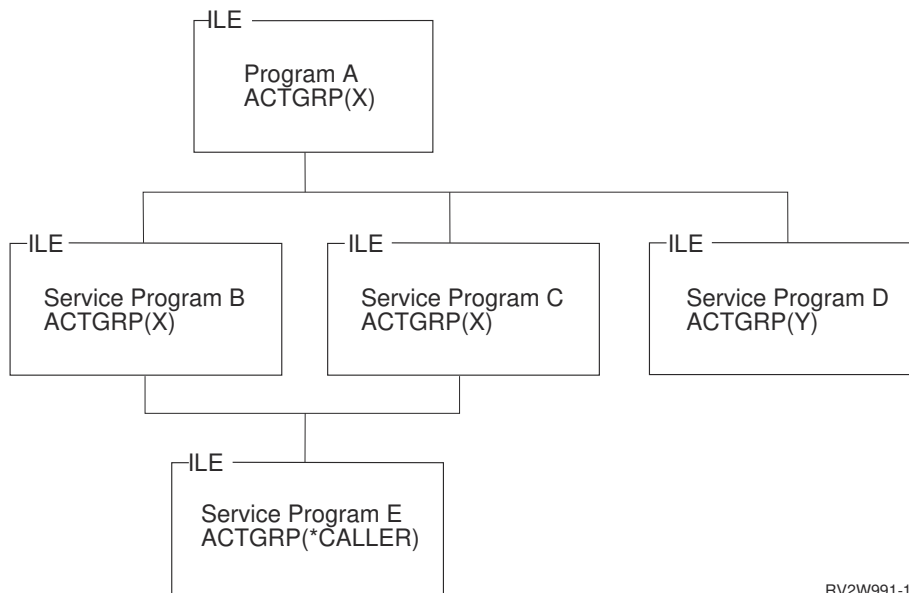
The system follows unique steps to activate a service program. The common steps used for programs and service programs are described in [“Program Activation”](#) on page 29. The following activation activities are unique for service programs that are bound for immediate activation:

- Service program activation starts indirectly as part of a dynamic program call to an ILE program.
- Service program activation includes completion of interprogram binding connections by mapping the symbolic links into physical links.
- Service program activation includes signature check processing.

These activation activities are performed for service programs that are bound for deferred activation when one of their imported procedures runs.

A program activated for the first time within an activation group is checked for binding to any service programs. If service programs are bound for immediate activation to the program that is activated, they are also activated as part of the same dynamic call processing. If service programs are bound for deferred activation to the program that is activated, those that satisfy procedure imports might not be activated until one of its imported procedures is called. Those that satisfy data imports are, at least, partially activated to initialize the static data. This process is repeated until all necessary service programs are activated.

[Figure 20 on page 36](#) shows ILE program A bound to service programs B, C, and D. Service programs B and C are also bound to service program E. The activation group attribute for each program and service program is shown.



RV2W991-1

Figure 20. Service Program Activation

Consider the case that all of these service programs are bound for immediate activation. When ILE program A is activated, the following actions take place:

- The service programs are located by using an explicit library name or by using the current library list. This option is controlled by you at the time the programs and service programs are created.
- Just like programs, a service program activation occurs only once within an activation group. In [Figure 20 on page 36](#), service program E is activated only one time, even though it is used by service programs B and C.
- A second activation group (Y) is created for service program D.
- Signature checking occurs among all of the programs and service programs.

Conceptually this process may be viewed as the completion of the binding process started when the programs and service programs were created. The CRTPGM command and CRTSRVPGM command saved the name and library of each referenced service program. An index into a table of exported procedures and data items was also saved in the client program or service program at program creation time. The process of service program activation completes the binding step by changing these symbolic references into addresses that can be used at runtime.

Once a service program is activated static procedure calls and static data item references to a module within a different service program are processed. The amount of processing is the same as would be required if the modules had been bound by copy into the same program. However, modules bound by copy require less activation time processing than service programs.

The activation of programs and service programs requires execute authority to the ILE program and all bound service program objects. In [Figure 20 on page 36](#), the current authority of the caller of program A is used to check authority to program A and all of the bound service programs. The authority of program A is also used to check authority to all of the bound service programs. Note that the authority of service program B, C, or D is not used to check authority to service program E.

Reconsider [Figure 20 on page 36](#) for the case that program A binds to service programs B and D for deferred activation, and C for immediate activation. D satisfies a data import for A. B satisfies procedure imports for A for static procedure calls only, but not for any procedure pointer calls. Then, B binds to service program E for deferred activation while C binds to E for immediate activation. When program A is activated, the following actions take place:

- Service program C, D, and E are located by using an explicit library name or by using the current library list. You can specify this option when you create the programs and service programs. D is located because it satisfies a data import and must be activated at least to the point that its static data is initialized.

- E is activated on behalf of C. When B runs and calls a procedure in E, E is not activated again because a service program activation only occurs once with an activation group.
- A second activation group (Y) is created for service program D.
- Signature checking occurs among all of the programs and the service programs that are bound for immediate activation or that require partial or full, immediate activation when A is activated. In this example, signature checking is done for C, D, and E.

Locating service program B and performing a signature check for B does not take place until one of its imported procedures is called.

The current authority of the caller of program A is used to check authority to program A and service programs C, D, and E. The current authority of the caller of service program B is used to check authority to B. The authority check of B might produce different results than for the case when B is bound for immediate activation.

Control Boundaries

ILE takes the following action when an unhandled function check occurs, an HLL end verb is used, or API CEETREC is called. ILE transfers control to the caller of the call stack entry that represents a boundary for your application. This call stack entry is known as a **control boundary**.

There are two definitions for a control boundary. [“Control Boundaries for Activation Groups” on page 37](#) and [“Control Boundaries between OPM and ILE Call Stack Entries” on page 38](#) illustrate the following definitions.

A control boundary can be either of the following:

- Any ILE call stack entry for which the immediately preceding call stack entry is associated with a different activation group.
- Any ILE call stack entry for which the immediately preceding call stack entry is for an OPM program.

Control Boundaries for Activation Groups

This example shows how control boundaries are defined between activation groups.

Figure 21 on page 38 shows two activation groups and the control boundaries established by the various calls. Invocations of procedures P2, P3, and P6 are potential control boundaries. For example, when you are running procedure P7, procedure P6 is the control boundary. When you are running procedures P4 or P5, procedure P3 becomes the control boundary.

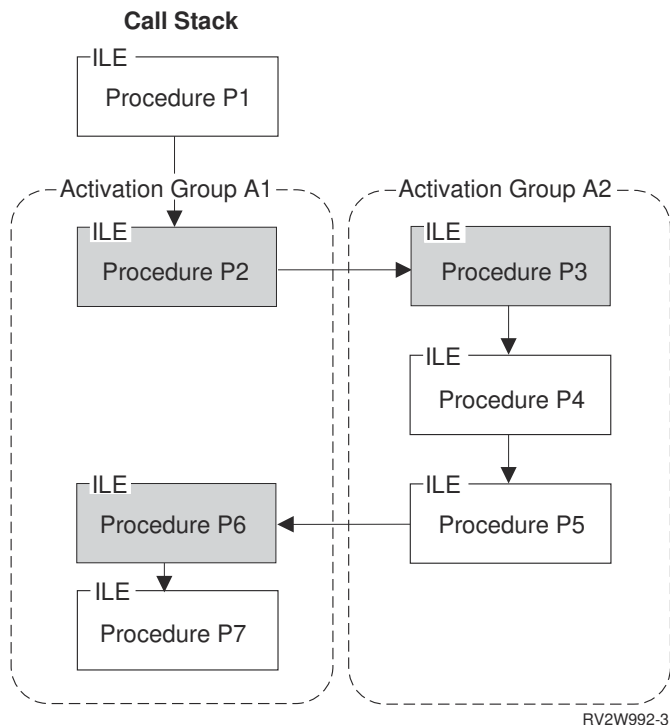


Figure 21. Control Boundaries

Note: Control boundaries are defined between all activation groups, including the default activation groups. For example, if a call stack entry associated with the teraspace default activation group immediately precedes a call stack entry associated with an OPM default activation group there will be a control boundary between the two call stack entries.

Control Boundaries between OPM and ILE Call Stack Entries

This example shows how control boundaries are defined between ILE call stack entries and OPM call stack entries.

Figure 22 on page 39 shows three ILE procedures (P1, P2, and P3) running in an OPM default activation group. This example could have been created by using the CRTPGM command or CRTSRVPGM command with the ACTGRP(*CALLER) parameter value. Invocations of procedures P1 and P3 are potential control boundaries because the preceding call stack entries are associated with OPM programs A and B.

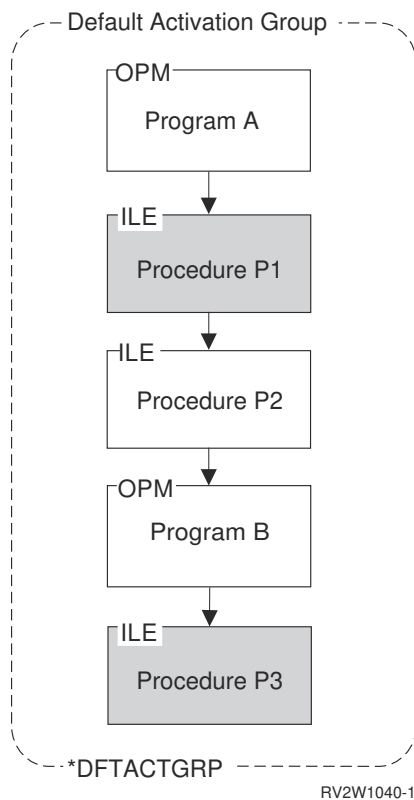


Figure 22. Control Boundaries between ILE call stack entries and OPM call stack entries

Control Boundary Use

When you use an ILE HLL end verb or call API CEETREC, ILE uses the most recent control boundary on the call stack to determine where to transfer control. The call stack entry just prior to the control boundary receives control after ILE completes all end processing.

The control boundary is used when an unhandled function check occurs within an ILE procedure. The control boundary defines the point on the call stack at which the unhandled function check is **promoted** to the generic ILE failure condition. For additional information, refer to [“Error Handling” on page 39](#).

When the nearest control boundary is the oldest call stack entry associated with a non-default activation group, any HLL end verb, call to the CEETREC API, or unhandled function check causes the activation group to be deleted. When the nearest control boundary is not the oldest call stack entry associated with a non-default activation group, control returns to the call stack entry just prior to the control boundary. The activation group is not deleted because earlier call stack entries exist for the same activation group.

Figure 21 on page 38 shows procedure P2 and procedure P3 as the oldest call stack entries associated with their activation groups. Using an HLL end verb or calling API CEETREC in procedure P2, P3, P4, or P5 (but not P6 or P7) would cause activation group A2 to be deleted.

Error Handling

This topic explains advanced error handling capabilities for OPM and ILE programs. To understand how these capabilities fit into the exception message architecture, refer to [Figure 23 on page 40](#). Specific reference information and additional concepts are found in [“Exception and Condition Management” on page 111](#). [Figure 23 on page 40](#) shows an overview of error handling. This topic starts with the bottom layer of this figure and continues to the top layer. The top layer represents the functions you may use to handle errors in an OPM or ILE program.

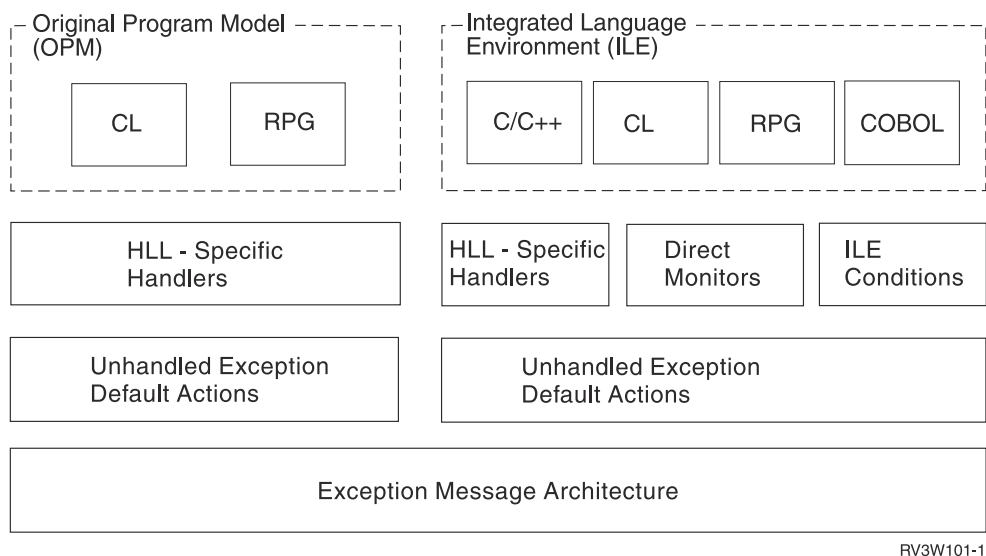


Figure 23. ILE and OPM Error Handling

Job Message Queues

A message queue exists for every call stack entry within each job. This message queue facilitates the sending and receiving of informational messages and exception messages between the programs and procedures running on the call stack. The message queue is referred to as the **call message queue**.

The call message queue is identified by the name of the OPM program or ILE procedure that is on the call stack. The procedure name or program name can be used to specify the target call stack entry for the message that you send. Because ILE procedure names are not unique, the ILE module name and ILE program or service program name can optionally be specified. When the same program or procedure has multiple call stack entries, the nearest call message queue is used.

In addition to the call message queues, each job contains one external message queue. All programs and procedures running within the job can send and receive messages between an interactive job and the workstation user by using this queue.

For information about sending and receiving exception messages by using APIs, see the Message Handling APIs in the API topic collection of the Programming category in the IBM i Information Center.

Exception Messages and How They Are Sent

This topic describes the different exception message types and the ways in which an exception message may be sent.

Error handling for ILE and OPM is based on exception message types. Unless otherwise qualified, the term **exception message** indicates any of these message types:

Escape (*ESCAPE)

Indicates an error causing a program to end abnormally, without completing its work. You will not receive control after sending an escape exception message.

Status (*STATUS)

Describes the status of work being done by a program. You may receive control after sending this message type. Whether you receive control depends on the way the receiving program handles the status message.

Notify (*NOTIFY)

Describes a condition requiring corrective action or a reply from the calling program. You may receive control after sending this message type. Whether you receive control depends on the way the receiving program handles the notify message.

Function Check

Describes an ending condition that has not been expected by the program. An ILE function check, CEE9901, is a special message type that is sent only by the system. An OPM function check is an escape message type with a message ID of CPF9999.

For information about these message types and other message types, see the API topic collection of the Programming category of the IBM i Information Center.

An exception message is sent in the following ways:

- Generated by the system

The operating system (including your HLL) generates an exception message to indicate a programming error or status information.

- Message handler API

The Send Program Message (QMHSNDPM) API can be used to send an exception message to a specific call message queue.

- ILE API

The Signal a Condition (CEESGL) bindable API can be used to raise an ILE condition. This condition results in an escape exception message or status exception message.

- Language-specific verbs

For ILE C and ILE C++, the `raise()` function generates a C signal. Neither ILE RPG nor ILE COBOL has a similar function.

How Exception Messages Are Handled

When you or the system send an exception message, exception processing begins. This processing continues until the exception is handled, which is when the exception message is modified to indicate that it has been handled.

The system modifies the exception message to indicate that it has been handled when it calls an exception handler for an OPM call message queue. Your ILE HLL modifies the exception message before your exception handler is called for an ILE call message queue. As a result, HLL-specific error handling considers the exception message handled when your handler is called. If you do not use HLL-specific error handling, your ILE HLL can either handle the exception message or allow exception processing to continue. Refer to your ILE HLL reference manual to determine your HLL default actions for unhandled exception messages.

With additional capabilities defined for ILE, you can bypass language-specific error handling. These capabilities include direct monitor handlers and ILE condition handlers. When you use these capabilities, you are responsible for changing the exception message to indicate that the exception is handled. If you do not change the exception message, the system continues exception processing by attempting to locate another exception handler. The topic [“Types of Exception Handlers” on page 43](#) contains details about direct monitor handlers and ILE condition handlers. For information that explains how to change an exception message, see the Change Exception Message (QMCHGEM) API in the API topic collection of the Programming category of the IBM i Information Center.

Exception Recovery

You may want to continue processing after an exception has been sent. Recovering from an error can be a useful application tool that allows you to deliver applications that tolerate errors. For ILE and OPM programs, the system has defined the concept of a **resume point**. The resume point is initially set to an instruction immediately following the occurrence of the exception. After handling an exception, you may continue processing at a resume point. For more information on how to use and modify a resume point, refer to [“Exception and Condition Management” on page 111](#).

Default Actions for Unhandled Exceptions

If you do not handle an exception message in your HLL, the system takes a default action for the unhandled exception.

Figure 23 on page 40 shows the default actions for unhandled exceptions based on whether the exception was sent to an OPM or ILE program. Different default actions for OPM and ILE create a fundamental difference in error handling capabilities.

For OPM, an unhandled exception generates a special escape message known as a function check message. This message is given the special message ID of CPF9999. It is sent to the call message queue of the call stack entry that incurred the original exception message. If the function check message is not handled, the system removes that call stack entry. The system then sends the function check message to the previous call stack entry. This process continues until the function check message is handled. If the function check message is never handled, the job ends.

For ILE, an unhandled exception message is percolated to the previous call stack entry message queue. **Percolation** occurs when the exception message is moved to the previous call message queue. This creates the effect of sending the same exception message to the previous call message queue. When this happens, exception processing continues at the previous call stack entry.

Percolation of an unhandled exception continues until either a control boundary is reached or the exception message is handled. The processing steps taken when an unhandled exception percolates to a control boundary depend on the type of exception:

1. If it is a status exception, the exception is handled and the sender of the status is allowed to continue.
2. If it is a notify exception, the default reply is sent, the exception is handled, and the sender of the notify is allowed to continue.
3. If it is an escape exception, a special function check message is sent to the resume point. This function check may be handled, or it may percolate to a control boundary (see below).
4. If it is a function check, ILE considers the application to have ended with an unexpected error. All call stack entries up to and including the control boundary are cancelled. If the control boundary is the oldest invocation associated with a non-default activation group, the activation group is ended. A generic failure exception message is defined by ILE for all languages. The message identifier for this message is CEE9901, and is sent by ILE to the call stack entry immediately preceding the control boundary.

Figure 24 on page 43 shows an unhandled escape exception within ILE. In this example, the invocation of procedure P1 is a control boundary and is also the oldest invocation entry for the associated activation group. Procedure P4 incurs an escape exception. The escape exception percolates to the P1 control boundary and as a result the special function check message is sent to procedure P3's resume point. The function check is not handled and percolates to the P1 control boundary. The application is considered to have ended and the activation group is destroyed. Finally, the CEE9901 escape exception is sent to the call stack entry immediately preceding the control boundary (program A).

The default action for unhandled exception messages defined in ILE allows you to recover from error conditions that occur within a mixed-language application. For unexpected errors, ILE enforces a consistent failure message for all languages. This improves the ability to integrate applications from different sources.

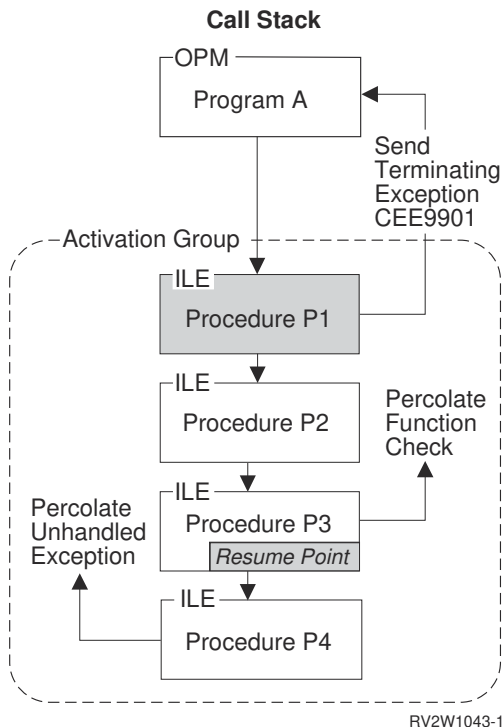


Figure 24. Unhandled Exception Default Action

Types of Exception Handlers

This topic provides an overview of the exception handler types provided for both OPM and ILE programs. As shown in Figure 23 on page 40, this is the top layer of the exception message architecture. ILE provides additional exception-handling capabilities when compared to OPM.

For OPM programs, HLL-specific error handling provides one or more handling routines for each call stack entry. The appropriate routine is called by the system when an exception is sent to an OPM program.

HLL-specific error handling in ILE provides the same capabilities. ILE, however, has additional types of exception handlers. These types of handlers give you direct control of the exception message architecture and allow you to bypass HLL-specific error handling. The additional types of handlers for ILE are:

- Direct monitor handler
- ILE condition handler

To determine if these types of handlers are supported by your HLL, refer to your ILE HLL programmer's guide.

Direct monitor handlers allow you to directly declare an exception monitor around limited HLL source statements. For ILE C, this capability is enabled through a `#pragma` directive. ILE COBOL does not directly declare an exception monitor around limited HLL source statements in the same sense that ILE C does. An ILE COBOL program cannot directly code the enablement and disablement of handlers around arbitrary source code. However, a statement such as

```
ADD a TO b ON SIZE ERROR imperative
```

is internally mapped to use the direct monitor mechanism despite being an HLL-specific handler. Thus, in terms of the priority of which handler gets control first, such a statement-scoped conditional imperative gets control before the ILE condition handler (registered through CEEHDLR). Control then proceeds to the USE declaratives in COBOL.

ILE condition handlers allow you to **register** an exception handler at runtime. ILE condition handlers are registered for a particular call stack entry. To register an ILE condition handler, use the Register a User-Written Condition Handler (CEEHDLR) bindable API. This API allows you to identify a procedure at runtime that should be given control when an exception occurs. The CEEHDLR API requires the ability

to declare and set a procedure pointer within your language. CEEHDLR is implemented as a built-in function. Therefore, its address cannot be specified and it cannot be called through a procedure pointer. ILE condition handlers may be **unregistered** by calling the Unregister a User-Written Condition Handler (CEEHDLU) bindable API.

OPM and ILE support HLL-specific handlers. **HLL-specific handlers** are the language features defined for handling errors. For example, the ILE C signal function can be used to handle exception messages. HLL-specific error handling in RPG includes the ability to handle exceptions for a single statement (E extender), a group of statements (MONITOR), or an entire procedure (*PSSR and INFSR subroutines). HLL-specific error handling in COBOL includes USE declarative for I/O error handling and imperatives in statement-scoped condition phrases such as ON SIZE ERROR and AT INVALID KEY.

Exception handler priority becomes important if you use both HLL-specific error handling and additional ILE exception handler types.

Figure 25 on page 45 shows a call stack entry for procedure P2. In this example, all three types of handlers have been defined for a single call stack entry. Though this may not be a typical example, it is possible to have all three types defined. Because all three types are defined, an exception handler priority is defined. The figure shows this priority. When an exception message is sent, the exception handlers are called in the following order:

1. Direct monitor handlers

First the invocation is chosen, then the relative order of handlers in that invocation. Within an invocation, all direct monitor handlers; RPG (E), MONITOR, INFSR, and error indicators; and COBOL statement-scoped conditional imperatives get control before the ILE condition handlers. Similarly, the ILE condition handlers get control before other HLL-specific handlers.

If direct monitor handlers have been declared around the statements that incurred the exception, these handlers are called before HLL-specific handlers. For example, if procedure P2 in Figure 25 on page 45 has a HLL-specific handler and procedure P1 has a direct monitor handler, P2's handler is considered before P1's direct monitor handler.

Direct monitors can be lexically nested. The handler specified in the most deeply nested direct monitor is chosen first within the multiply nested monitors that specify the same priority number.

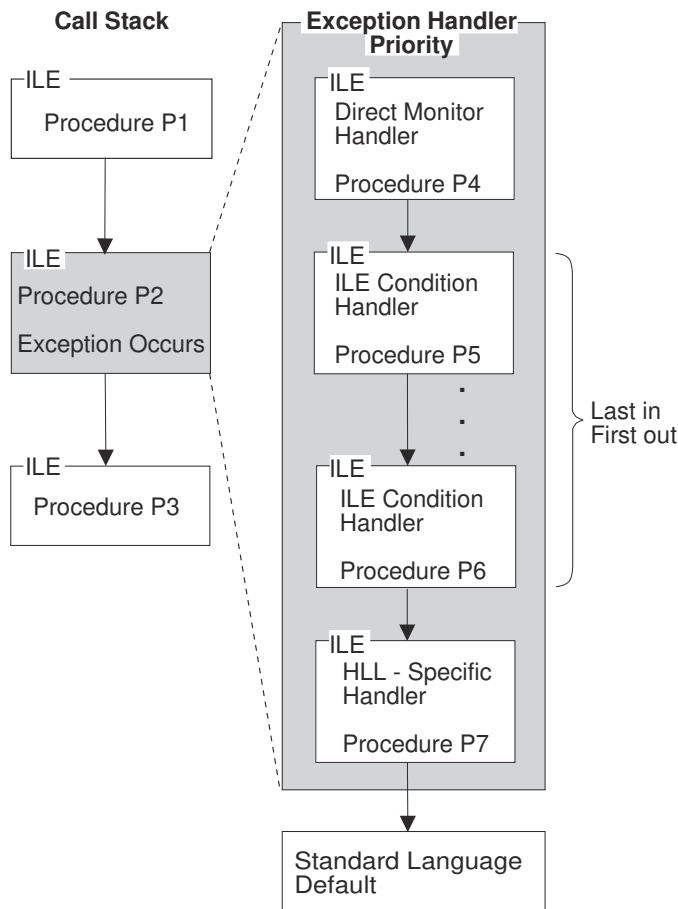
2. ILE condition handler

If an ILE condition handler has been registered for the call stack entry, this handler is called second. Multiple ILE condition handlers can be registered. In the example, procedure P5 and procedure P6 are ILE condition handlers. When multiple ILE condition handlers are registered for the same call stack entry, the system calls these handlers in last-in-first-out (LIFO) order. In general, HLL-specific handlers have the lowest priority, after direct monitor handlers and condition handlers. Exceptions are the HLL-specific handlers mentioned in the discussion of direct monitor handlers.

3. HLL-specific handler

HLL-specific handlers are called last.

The system ends exception processing when an exception message is modified to show that it has been handled. If you are using direct monitor handlers or ILE condition handlers, modifying the exception message is your responsibility. Several control actions are available. For example, you can specify handle as a control action. As long as the exception message remains unhandled, the system continues to search for an exception handler using the priorities previously defined. If the exception is not handled within the current call stack entry, percolation to the previous call stack entry occurs. If you do not use HLL-specific error handling, your ILE HLL can choose to allow exception handling to continue at the previous call stack entry.



RV2W1041-3

Figure 25. Exception Handler Priority

ILE Conditions

To allow greater cross-system consistency, ILE has defined a feature that you can use to work with error conditions. An ILE condition is a system-independent representation of an error condition within an HLL. Each ILE condition has a corresponding exception message. An ILE condition is represented by a condition token. A *condition token* is a 12-byte data structure that is consistent across multiple participating systems. This data structure contains information through which you can associate the condition with the underlying exception message.

To write programs that are consistent across systems, you need to use ILE condition handlers and ILE condition tokens. For more information on ILE conditions refer to [“Exception and Condition Management”](#) on page 111.

Data Management Scoping Rules

Data management scoping rules control the use of data management resources. These resources are temporary objects that allow a program to work with data management. For example, when a program opens a file, an object called an open data path (ODP) is created to connect the program to the file. When a program creates an override to change how a file should be processed, the system creates an override object.

Data management scoping rules determine when a resource can be shared by multiple programs or procedures running on the call stack. For example, open files created with the SHARE(*YES) parameter value or commitment definition objects can be used by many programs at the same time. The ability to share a data management resource depends on the level of scoping for the data management resource.

Data management scoping rules also determine the existence of the resource. The system automatically deletes unused resources within the job, depending on the scoping rules. As a result of this automatic cleanup operation, the job uses less storage and job performance improves.

ILE formalizes the data management scoping rules for both OPM and ILE programs into the following scoping levels:

- Call
- Activation group
- Job

Depending on the data management resource you are using, one or more of the scoping levels may be explicitly specified. If you do not select a scoping level, the system selects one of the levels as a default.

Refer to [“Data Management Scoping” on page 123](#) for information on how each data management resource supports the scoping levels.

Call-Level Scoping

Call-level scoping occurs when the data management resource is connected to the call stack entry that created the resource. Figure 26 on page 46 shows an example. Call-level scoping is usually the default scoping level for programs that run in the default activation group. In this figure, OPM program A, OPM program B, or ILE procedure P2 may choose to return without closing their respective files F1, F2, or F3. Data management associates the ODP for each file with the call-level number that opened the file. The RCLRSC command may be used to close the files based on a particular call-level number passed to that command.

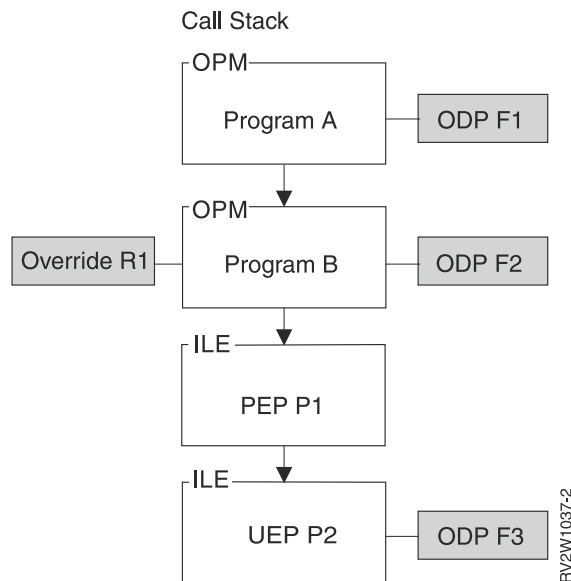


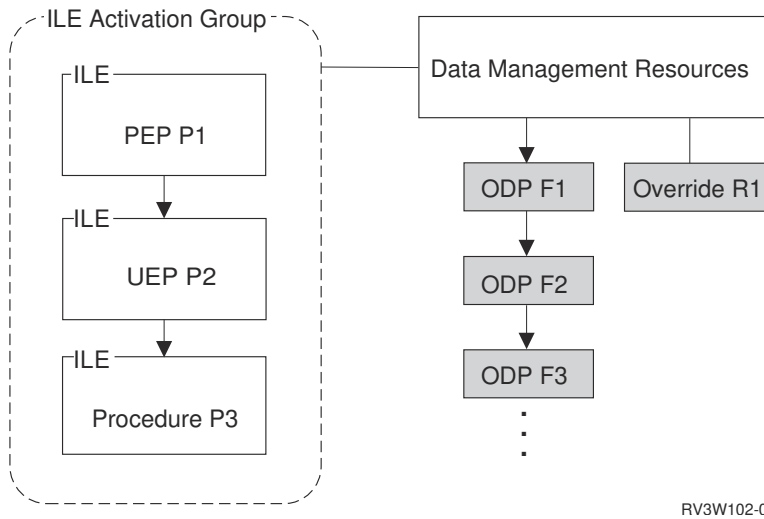
Figure 26. Call-Level Scoping

Overrides that are scoped to a particular call level are deleted when the corresponding call stack entry returns. Overrides may be shared by any call stack entry that is below the call level that created the override.

Activation-Group-Level Scoping

Activation-group-level scoping occurs when the data management resource is connected to the activation group associated with the ILE program or procedure invocation that created the resource. When the activation group is deleted, data management closes all resources associated with the activation group that have been left open. Figure 27 on page 47 shows an example of activation-group-level scoping. Activation-group-level scoping is the default scoping level for most types of data management resources

used by ILE procedures not running in a default activation group. For example, the figure shows ODPs for files F1, F2, and F3 and override R1 scoped to the activation group.



RV3W102-0

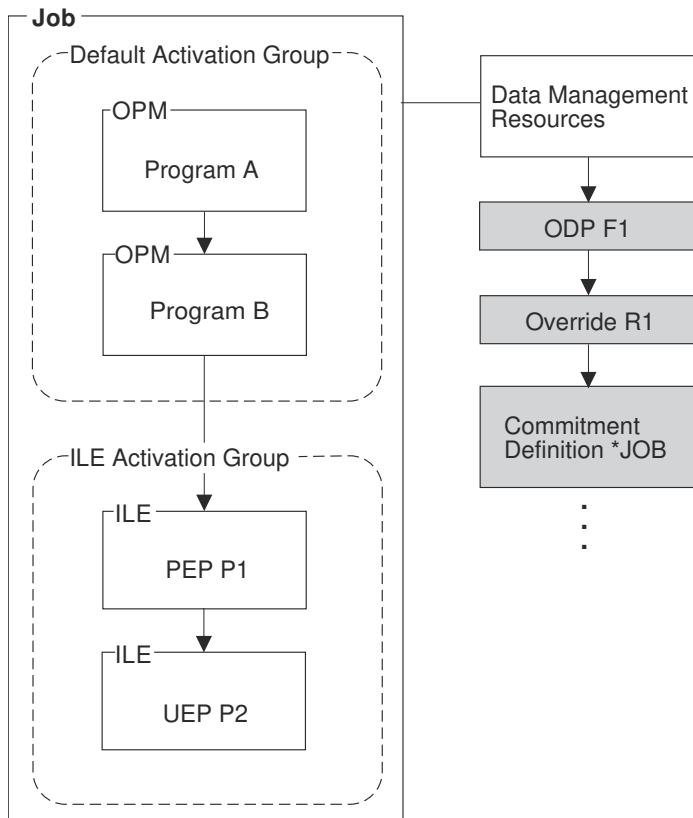
Figure 27. Activation Group Level Scoping

The ability to share a data management resource scoped to an activation group is limited to programs running in that activation group. This provides application isolation and protection. For example, assume that file F1 in the figure was opened with the SHARE(*YES) parameter value. File F1 could be used by any ILE procedure running in the same activation group. Another open operation for file F1 in a different activation group results in the creation of a second ODP for that file.

Job-Level Scoping

Job-level scoping occurs when the data management resource is connected to the job. Job-level scoping is available to both OPM and ILE programs. Job-level scoping allows for sharing data management resources between programs running in different activation groups. As described in the previous topic, scoping resources to an activation group limits the sharing of that resource to programs running in that activation group. Job-level scoping allows the sharing of data management resources between all ILE and OPM programs running in the job.

Figure 28 on page 48 shows an example of job-level scoping. Program A may have opened file F1, specifying job-level scoping. The ODP for this file is connected to the job. The file is not closed by the system unless the job ends. If the ODP has been created with the SHARE(YES) parameter value, any OPM program or ILE procedure could potentially share the file.



RV2W1039-2

Figure 28. Job-Level Scoping

Overrides scoped to the job level influence all open file operations in the job. In this example, override R1 could have been created by procedure P2. A job-level override remains active until it is either explicitly deleted or the job ends. The job-level override is the highest priority override when merging occurs. This is because call-level overrides are merged together when multiple overrides exist on the call stack.

Data management scoping levels may be explicitly specified by the use of scoping parameters on override commands, commitment control commands, and through various APIs. The complete list of data management resources that use the scoping rules are in [“Data Management Scoping”](#) on page 123.

Teraspace and Single-Level Storage

When you create ILE programs, you can select one of the following storage models in some compilers:

- Single-level storage
- Teraspace
- Inherit

The inherit storage model indicates that programs will adopt the storage model, either teraspace or single-level storage, of the activation group into which they are activated. ILE programs use single-level storage by default.

This topic focuses on the teraspace options.

Teraspace Characteristics

Teraspace is a large temporary space that is local to a job. A teraspace provides a contiguous address space but may consist of many individually allocated areas, with unallocated areas in between. Teraspace exists no longer than the time between job start and job end.

A teraspace is not a space object. This means that it is not a system object, and that you cannot refer to it by using a system pointer. However, teraspace is addressable with space pointers within the same job.

The following table shows how teraspace compares to single-level storage.

Table 2. Comparing teraspace to single-level storage		
Attributes	Teraspace	Single-level storage
Locality	Process local: normally accessible only to the owning job.	Global: accessible to any job that has a pointer to it.
Size	Approximately 100 TB total	Many 16 MB units.
Supports memory mapping?	Yes	No
Addressed by 8-byte pointers?	Yes	No
Supports sharing between jobs?	Must be done using shared memory APIs (for example, shmat or mmap).	Can be done by passing pointers to other jobs or using shared memory APIs.

Using Teraspace for Storage

A program's *storage model* determines the type of storage used for automatic, static and constant storage.

By default, compilers also use heap storage interfaces that match the storage model. However, some compilers allow the type of heap storage to be chosen independently of a program's storage model.

The ILE C and C++ compilers provide the TERASPACE (*YES *TSIFC) create command option to allow the use of teraspace versions of heap storage interfaces from single-level store programs without source code changes. For example, malloc() is mapped to _C_TS_malloc().

The ILE RPG compiler allows the type of heap storage to be explicitly set, independent of the storage model, using the ALLOC keyword on the Control Specification.

See the [ILE C/C++ Programmer's Guide](#)  or [ILE RPG Programmer's Guide](#)  for details on these compiler options.

Choosing a Program Storage Model

You can optionally create your modules and ILE programs so that they use the *teraspaces storage model*. Teraspace storage model programs use teraspace for automatic, static, and constant storage. When you choose the teraspace storage model, you can use larger areas for some of these types of storage. See [“Using the Teraspace Storage Model” on page 55](#) for more information about the teraspace storage model.

For modules, programs, and service programs, you have the option of specifying one of the following storage models for some compilers:

- Single-level storage (*SNGLVL)
- Teraspace (*TERASPACE)
- Inherit (*INHERIT)

This topic discusses the teraspace storage model.

Specifying the Teraspace Storage Model

To choose the teraspace storage model for your RPG, COBOL, C or C++ program, specify the following options when you compile your code:

1. For C and C++, specify *YES on the TERASPACE parameter when you create your modules.
2. Specify *TERASPACE or *INHERIT on the Storage model (STGMDL) parameter of the create module command for your ILE programming language.
3. Specify *TERASPACE on the STGMDL parameter of the Create Program (CRTPGM) or Create Service Program (CRTSRVPGM) command. This choice must be compatible with the storage model of the modules that you bind with the program. See [“Rules for Binding Modules” on page 51](#) for details.

You can also specify *TERASPACE on the STGMDL parameter of the Create Bound C Program (CRTBNDC), Create Bound C++ Program (CRTBNDCPP), Create Bound RPG Program (CRTBNDRPG), and Create Bound COBOL Program (CRTBNDCBL) commands, which create in one step a bound program that contains only one module.

On the CRTPGM and CRTSRVPGM commands, you can also specify *INHERIT on the STGMDL parameter. This causes the program or service program to be created in such a way that it can use either single-level storage or teraspace, depending on the type of storage in use in the activation group into which the program or service program is activated.

The use of the *INHERIT attribute provides the greatest flexibility, but then you must also specify *CALLER on the ACTGRP parameter. In this case, remember that your program or service program can get activated with either single-level storage or teraspace, and you must take care that your code can effectively handle both situations. For example, the total size of all static variables must be no larger than the smaller limits imposed for single-level storage.

Table 3. Allowed storage model for particular types of programs			
Program storage model	Program type		
	OPM *PGM	ILE *PGM	ILE *SRVPGM
*TERASPACE	No	Yes	Yes
*INHERIT	No	Yes, but only with ACTGRP(*CALLER)	Yes, but only with ACTGRP(*CALLER)
*SNGLVL	Yes	Yes	Yes

Selecting a Compatible Activation Group

An activation group reflects the storage model of the root program that caused the activation group to be created. The storage model determines the type of automatic, static, and constant storage that is provided to the program.

Single-level storage model programs receive single-level automatic, static, and constant storage. By default, these programs will also use single-level storage for heap storage.

Teraspace storage model programs receive teraspace automatic, static, and constant storage. By default, these programs will also use teraspace for heap storage.

Programs that use the teraspace storage model cannot be activated into an activation group whose root program uses the single-level storage model (this includes the OPM default activation groups). Programs that use the single-level storage model cannot be activated into an activation group whose root program uses the teraspace storage model.

The following table summarizes the relationship between storage models and the activation group type.

<i>Table 4. Relationship of storage models to activation groups</i>				
Program storage model	Activation group attribute			
	*CALLER	*DFTACTGRP	*NEW	Named
*TERASPACE	Yes	Not allowed.	Yes	Yes
*INHERIT	Yes	Not allowed.	Not allowed.	Not allowed.
*SINGLVL	Yes	Yes	Yes	Yes

When you choose the activation group in which your program or service program runs, consider the following guidelines:

- If your service program specifies STGMDL(*INHERIT), you must specify ACTGRP(*CALLER).
- If your program specifies STGMDL(*TERASPACE):
 - Specify ACTGRP(*NEW) or a named activation group.
 - Specify ACTGRP(*CALLER) only if you can assure that every program that calls your program uses the teraspace storage model.

How the Storage Models Interact

Consistency is required among the modules and programs that use a storage model. Here are rules to insure that programs interact properly.

- [“Rules for Binding Modules” on page 51](#)
- [“Rules for Binding to Service Programs” on page 52](#)
- [“Rules for Activating Programs and Service Programs” on page 52](#)
- [“Rules for Program and Procedure Calls” on page 52](#)

Rules for Binding Modules

The following table shows the rules for binding modules:

Binding rules: Binding module M into a program with a specified storage model.		The storage model of the program that is being created.		
		Teraspace	Inherit	Single-level storage
M	Teraspace	Teraspace	Error	Error
	Inherit	Teraspace	Inherit	Single-level storage
	Single-level storage	Error	Error	Single-level storage

Rules for Binding to Service Programs

The following table shows the rules for binding programs to target service programs.

Service program binding rules: Can the calling program or service program bind to a target service program?		Target service program storage model		
		Teraspace	Inherit	Single-level storage
Storage model of the calling program or service program	Teraspace	Yes	Yes	Yes ¹
	Inherit ¹	Yes ²	Yes	Yes ²
	Single-level storage	Yes ¹	Yes	Yes

Note:

1. The target service program must run in a distinct activation group. For example, the target service program cannot have the ACTGRP(*CALLER) attribute. It is not possible to mix storage models within a single activation group.
2. If the calling program or service program uses the inherit storage model and the target service program uses single-level storage or the teraspace storage model, then you must ensure that the activation group that the target service program is activated into has the same storage model as the target service program. Here is an example: service program A is created with the inherit storage model. Service program B is created with teraspace storage model and has the *CALLER activation group attribute. Service program A is bound to service program B. In this case, service program A should always activate into an activation group with the teraspace storage model.

Rules for Activating Programs and Service Programs

A service program that specifies the inherit storage model can activate into an activation group that runs programs that use single-level storage or teraspace storage models. Otherwise, the storage model of the service program must match the storage model of other programs that run in the activation group.

Rules for Program and Procedure Calls

Programs and service programs that use different storage models can interoperate. They can be bound together and share data as long as they conform to the rules and restrictions described in this topic.

Converting Your Program or Service Program to Inherit a Storage Model

By converting your programs or service programs to inherit a storage model (specifying *INHERIT on the STGM DL parameter), you enable them for use in either teraspace or single-level storage environments. Make sure that your code anticipates and effectively manages pointers to and from teraspace and single-level storage. See [“Using Teraspace: Best Practices”](#) on page 55 for more information.

There are two ways to enable your existing programs or service programs for the teraspace storage model: you can re-create them, or in some circumstances, you can change the storage model of existing

programs. When creating a program, first create all of your modules with the *INHERIT storage model. Then create your ILE program or service program with the *INHERIT storage model.

Alternatively, you can change the storage model of an existing program or service program from single-level storage (*SNGLVL) to *INHERIT, using the CHGPGM or CHGSRVPGM command, if the following are true:

1. The object is an ILE program or service program.
2. The object uses single-level storage model.
3. The object uses the activation group of its caller.
4. The object is a program whose target release is V6R1M0 or later.
5. The object is a service program whose target release is V5R1M0 or later.
6. All bound modules of the object have a target release of V5R1M0 or later.

Updating Your Programs: Teraspace Considerations

You can add and replace modules within a program as long as they use the same storage model. However, you cannot use the update commands to change the storage model of the bound module or the program.

Taking Advantage of 8-byte Pointers in Your C and C++ Code

An 8-byte pointer can point only to teraspace. An 8-byte procedure pointer refers to an active procedure through teraspace. The only types of 8-byte pointers are space and procedure pointers.

In contrast, there are many types of 16-byte pointers. The following table shows how 8-byte and 16-byte pointers compare.

Table 5. Pointer comparison		
Property	8-byte pointer	16-byte pointer
Length (memory required)	8 bytes	16 bytes
Tagged	No	Yes
Alignment	Byte alignment is permitted (that is, a packed structure). "Natural" alignment (8-byte) is preferred for performance.	Always 16-byte.
Atomicity	Atomic load and store operations when 8-byte aligned. Does not apply to aggregate copy operations.	Atomic load and store operations. Atomic copy when part of an aggregate.
Addressable range	Teraspace storage	Teraspace storage + single-level storage
Pointer content	A 64-bit value which represents an offset into teraspace. It does not contain an effective address.	16-byte pointer type bits and a 64-bit effective address.
Locality of reference	Process local storage reference. (An 8-byte pointer can only reference the teraspace of the job in which the storage reference occurs.)	Process local or single-level storage reference. (A 16-byte pointer can reference storage that is logically owned by another job.)

Table 5. Pointer comparison (continued)		
Property	8-byte pointer	16-byte pointer
Operations permitted	Pointer-specific operations allowed for space pointers and procedure pointers, and using a non-pointer view, all arithmetic and logical operations appropriate to binary data can be used without invalidating the pointer.	Only pointer-specific operations.
Fastest storage references	No	Yes
Fastest loads, stores, and space pointer arithmetic	Yes, including avoiding EAO overhead.	No
Size of binary value preserved when cast to pointer	8 bytes	4 bytes
Can be accepted as a parameter by a procedure that is an exception handler or cancel handler.	No	Yes

Pointer Support in C and C++ Compilers

To take full advantage of 8-byte pointers when you compile your code with the IBM C or C++ compiler, specify STGMDDL(*TERASPACE) and DTAMDLL(*LLP64).

The C and C++ compilers also provide the following pointer support:

- Syntax for explicitly declaring 8- or 16-byte pointers:
 - Declare a 8-byte pointer as `char * __ptr64`
 - Declare a 16-byte pointer as `char * __ptr128`
- A compiler option and pragma for specifying the *data model*, which is unique to the C and C++ programming environment. The data model affects the default size of pointers in the absence of one of the explicit qualifiers. You have two choices for the data model:
 - P128, also known as 4-4-16¹
 - LLP64, also known as 4-4-8²

Pointer Conversions

The IBM C and C++ compilers convert `__ptr128` to `__ptr64` and vice versa as needed, based on function and variable declarations. However, interfaces with pointer-to-pointer parameters require special handling.

The compilers automatically insert pointer conversions to match pointer lengths. For example, conversions are inserted when the pointer arguments to a function do not match the length of the pointer parameters in the prototype for the function. Or, if pointers of different lengths are compared, the compiler will implicitly convert the 8-byte pointer to a 16-byte pointer for the comparison. The compilers also allow explicit conversions to be specified, as casts. Keep these points in mind if adding pointer casts:

- A conversion from a 16-byte pointer to an 8-byte pointer works only if the 16-byte pointer contains a teraspace address or a null pointer value. Otherwise, an MCH0609 exception is signalled.

¹ Where 4-4-16 = sizeof(int) – sizeof(long) – sizeof(pointer)

² Where 4-4-8 = sizeof(int) – sizeof(long) – sizeof(pointer)

- 16-byte pointers cannot have types converted from one to another, but a 16-byte OPEN pointer can contain any pointer type. In contrast, no 8-byte OPEN pointer exists, but 8-byte pointers can be logically converted between a space pointer and a procedure pointer. Even so, an 8-byte pointer conversion is just a view of the pointer type, so it doesn't allow a space pointer to actually be used as a procedure pointer unless the space pointer was set to point to a procedure.

When adding explicit casts between pointers and binary values, remember that 8-byte and 16-byte pointers behave differently. An 8-byte pointer can retain a full 8-byte binary value, while a 16-byte pointer can only retain a 4-byte binary value. While holding a binary value, the only operation defined for a pointer is a conversion back to a binary field. All other operations are undefined, including use as a pointer, conversion to a different pointer length and pointer comparison. So, for example, if the same integer value were assigned to an 8-byte pointer and to a 16-byte pointer, then the 8-byte pointer were converted to a 16-byte pointer and a 16-byte pointer comparison were done, the comparison result would be undefined and likely would not produce an equal result.

Mixed-length pointer comparisons are defined only when a 16-byte pointer holds a teraspace address and an 8-byte pointer does, too (that is, the 8-byte pointer does not contain a binary value). Then it is valid to convert the 8-byte pointer to a 16-byte pointer and compare the two 16-byte pointers. In all other cases, comparison results are undefined. So, for example, if a 16-byte pointer were converted to an 8-byte pointer and then compared with an 8-byte pointer, the result is undefined.

Using the Teraspace Storage Model

In an ideal teraspace environment, all of your modules, programs, and service programs would use the teraspace storage model. On a practical level, however, you will need to manage an environment that combines modules, programs, and service programs that use both storage models.

This topic describes the practices you can implement to move toward an ideal teraspace environment. This topic also discusses how you can minimize the potential problems of an environment that mixes programs that use single-level storage and teraspace.

Using Teraspace: Best Practices

- *Use only teraspace storage model modules*

Create your modules such that they use the teraspace or inherit storage model. Single-level storage modules are not suitable for a teraspace environment because you cannot bind them into your program. If you absolutely have to use them (for instance, if you do not have access to the source code for the module), see scenario 9 in [“Teraspace Usage Tips” on page 57](#).

- *Bind only to service programs that use the teraspace or inherit storage model*

Your teraspace storage model program can bind to almost any kind of service program. However, it normally binds only to inherit or teraspace storage model service programs. If you control the service programs, you must create all of your service programs such that they can inherit the storage model of the program that binds them. In general, IBM service programs are created in this manner. You might need to do the same, especially if you plan to provide your service programs to third-party programmers. See scenario 10 in [“Teraspace Usage Tips” on page 57](#) if you absolutely have to bind to a single-level storage service program.

If you have followed the guidelines described in this topic, you can use teraspace in your programs. However, the use of teraspace requires that you pay careful attention to your coding practices, because single-level storage is used by default. The following topics describe the things you cannot do with teraspace, and some things you should not do. In some cases, the system prevents you from performing certain actions, but at other times you must manage potential teraspace and single-level storage interactions on your own.

- [“System Controls over Teraspace Programs When They are Created” on page 56](#)
- [“System Controls over Teraspace Programs When They are Activated” on page 56](#)

Note: Service programs that use the inherit storage model must also follow these practices because they may be activated to use teraspace.

System Controls over Teraspace Programs When They are Created

In most cases, the system prevents you from doing any of the following actions:

- Combining single-level storage and teraspace storage model modules into the same program or service program.
- Creating a teraspace storage model program or service program that also specifies a default activation group (ACTGRP(*DFTACTGRP)).
- Binding a single-level storage program to a teraspace storage model service program that also specifies an activation group of *CALLER.

System Controls over Teraspace Programs When They are Activated

In some cases at activation time, the system will determine that you have created your programs and service programs in such a way that both single-level storage and teraspace storage model programs or service programs would attempt to activate into the same activation group. The system will then send the activation access violation exception and fail the activation.

IBM i Interfaces and Teraspace

Interfaces that have pointer parameters typically expect tagged 16 byte (__ptr128) pointers:

- You can call interfaces with only a single level of pointer (for example, void f(char*p);) directly using 8-byte (__ptr64) pointers since the compiler will convert the pointer as required. Be sure to use the system header files.
- Interfaces with multiple levels of pointers (for example, void g(char**p);) ordinarily require that you explicitly declare a 16 byte pointer for the second level. However, versions that accept 8-byte pointers are provided for most system interfaces of this type, to allow them to be called directly from code that uses only 8-byte pointers. These interfaces are enabled through the standard header files when you select the datamodel(LLP64) option.

Bindable APIs for using teraspace:

IBM provides bindable APIs for allocating and discarding teraspace.³

- _C_TS_malloc() allocates storage within a teraspace.
- _C_TS_malloc64() allows a teraspace storage allocation to be specified using an 8-byte size value.
- _C_TS_free() frees one previous allocation of teraspace.
- _C_TS_realloc() changes the size of a previous teraspace allocation.
- _C_TS_calloc() allocates storage within a teraspace and sets it to 0.

malloc(), free(), calloc(), and realloc() allocate or deallocate single-level storage or teraspace storage according to the storage model of their calling program, unless it was compiled with the TERASPACE(*YES *TSIFC) compiler option.

POSIX shared memory and memory mapped file interfaces can use teraspace. For more information about Interprocess Communication APIs and the shmget() interface, see the *UNIX-type APIs* topic in the IBM i Information Center (under the Programming category and API topic).

Potential Problems that Can Arise When You Use Teraspace

When you use teraspace in your programs, you should be aware of the potential problems that can arise.

- *If you create programs using the TGTRLS(V5R4M0) parameter, be careful not to develop a dependency on passing teraspace addresses to other programs unless all these other programs can handle teraspace addresses.* All programs that run on IBM i 6.1 or later can use teraspace addresses.

³ The teraspace compiler option TERASPACE(*YES *TSIFC) is available from ILE C and C++ compilers to automatically map malloc(), free(), calloc() and realloc() to their teraspace versions when STGMDL(*SINGLVL) is specified.

- *Some MI instructions cannot process a teraspace address.* An attempt to use a teraspace address in these instructions causes an MCH0607 exception.
 - MATBPGM
 - MATPG
 - SCANX (only some options are limited)
- *Effective Address Overflow (EAO) can impair performance.* This situation can occur in certain code generated for a processor prior to POWER6. See [“Adaptive Code Generation”](#) on page 149. Specifically, an EAO can occur during an address computation on a 16-byte pointer, when the result is a teraspace address value that is less than the 16 MB boundary lower than the start address. A hardware interrupt is generated that is handled by the system software. Many such interrupts can affect performance. Avoid frequent teraspace address calculations that compute a smaller value in a different 16 MB area. Or, create your program with the MinimizeTeraspaceFalseEAOs LICOPT, as described in [“Licensed Internal Code Options”](#) on page 143.

Teraspace Usage Tips

You might encounter the following scenarios as you work with the teraspace storage model. Recommended solutions are provided.

- *Scenario 1: You need more than 16 MB of dynamic storage in a single allocation*
Use `_C_TS_malloc` or create your programs so that teraspace will be used for heap storage, as described in [“Using Teraspace for Storage”](#) on page 49.
- *Scenario 2: You need more than 16 MB of shared memory*
Use shared memory (`shmget`) with the teraspace option.
- *Scenario 3: You need to access large byte-stream files efficiently*
Use memory mapped files (`mmap`).

You can access memory-mapped files from any program, but for best performance, use the teraspace storage model and, if your language supports it, the 8-byte pointer data model.
- *Scenario 4: You need greater than 16 MB of contiguous automatic or static storage*
Use teraspace storage model.
- *Scenario 5: Your application makes heavy use of space pointers*
Use the teraspace storage model and a language that supports the 8-byte pointer data model, to reduce memory footprint and speed up pointer operations.
- *Scenario 6: You need to port code from another system and want to avoid issues that are unique to 16-byte pointer usage*
Use the teraspace storage model and a language that supports the 8-byte pointer data model.
- *Scenario 7: You need to use single-level storage in your teraspace program*
Sometimes your only choice is to use single-level storage in your teraspace storage model programs. For example, you might need it to store user data for interprocess communication. You can get single-level storage from any of the following sources:
 - Storage in a user space, obtained from the QUSCRTUS API or the CRTS MI instruction
 - Single-level storage heap interfaces in your programming language
 - Single-level storage reference that was passed to your program
 - Single-level storage heap space obtained from the ALCHS MI instruction
- *Scenario 8: Take advantage of 8-byte pointers in your code*
Create your module and program with `STGMDL(*TERASPACE)`. Use `DTAMD(*LLP64)` or explicit declarations (`__ptr64`) to get 8-byte pointers to refer to teraspace (as opposed to 16-byte pointers)

pointing into teraspace). Then you will get the advantages listed in [“Taking Advantage of 8-byte Pointers in Your C and C++ Code”](#) on page 53.

- *Scenario 9: Incorporating a single-level storage model module*

You cannot bind a single-level storage module with a teraspace storage model module. If you need to do this, first try to get a version of the module that uses (or inherits) the teraspace storage model, then simply use it as described in [“Using Teraspace: Best Practices”](#) on page 55. Otherwise, you have two options:

- Package the module into a separate service program. The service program will use the single-level storage model, so use the approach given in scenario 10, below, to call it.
- Package the module into a separate program. This program will use the single-level storage model. Use the approach outlined in scenario 11, below, to call it.

- *Scenario 10: Binding to a single-level storage model service program*

You can bind your teraspace program to a service program that uses single-level storage if the two service programs activate into separate activation groups. You cannot do this if the single-level storage service program specifies the ACTGRP(*CALLER) option.

- *Scenario 11: Calling functions that have pointer-to-pointer parameters*

Calls to some functions that have pointer-to-pointer parameters require special handling from modules compiled with the DTMDL(*LLP64 option). Implicit conversions between 8- and 16-byte pointers apply to pointer parameters. They do not apply to the data object pointed to by the pointer parameter, even if that pointer target is also a pointer. For example, the use of a **char**** interface declared in a header file that asserts the commonly used P128 data model will require some code in modules that are created with data model LLP64. Be sure to pass the address of a 16-byte pointer for this case. Here are some examples:

- In this example, you have created a teraspace storage model program using 8-byte pointers with the STGMDL (*TERASPACE) DTAMD (*LLP64) options on a create command, such as CRTCMOD. You now want to pass a pointer to a pointer to a character in an array from your teraspace storage model program to a P128 **char**** interface. To do so, you must explicitly declare a 16-byte pointer:

```
#pragma datamodel(P128)
void func(char **);
#pragma datamodel(pop)

char myArray[32];
char *_ptr128 myPtr;

myPtr = myArray; /* assign address of array to 16-byte pointer */
func(&myPtr);    /* pass 16-byte pointer address to the function */
```

- One commonly used application programming interface (API) with pointer-to-pointer parameters is **iconv**. It expects only 16-byte pointers. Here is part of the header file for **iconv**:

```
...
#pragma datamodel(P128)
...
size_t iconv(iconv_t cd,
             char **inbuf,
             size_t *inbytesleft,
             char **outbuf,
             size_t *outbytesleft);
...
#pragma datamodel(pop)
...
```

The following code calls **iconv** from a program compiled with the DTAMD(*LLP64) option:

```
...
iconv_t myCd;
size_t myResult;
char *_ptr128 myInBuf, myOutBuf;
size_t myInLeft, myOutLeft;
...
```

```
myResult = iconv(myCd, &myInBuf, &myInLeft, &myOutBuf, &myOutLeft);  
...
```

You should also be aware that the header file of the Retrieve Pointer to User Space (QUSPTRUS) interface specifies a `void*` parameter where a pointer to a pointer is actually expected. Be sure to pass the address of a 16-byte pointer for the second operand.

- *Scenario 12: Redefining functions*

Avoid redeclaring functions that are already declared in header files supplied by IBM. Usually, the local declarations do not have the correct pointer lengths specified. One such commonly used interface is **errno**, which is implemented as a function call in IBM i.

- *Scenario 13: Using data model *LLP64 with programs and functions that return a pointer*

If you are using data model *LLP64, look carefully at programs and functions that return a pointer. If the pointer points to single-level storage, its value cannot be correctly assigned to an 8-byte pointer, so clients of these interfaces must maintain the returned value in a 16-byte pointer. One such API is QUSPTRUS. User spaces reside in single-level storage.

Program Creation Concepts

The process for creating ILE programs or service programs gives you greater flexibility and control in designing and maintaining applications. The process includes two steps:

1. Compiling source code into modules.
2. Binding modules into an ILE program or service program. Binding occurs when the Create Program (CRTPGM) or Create Service Program (CRTSRVPGM) command is run.

This topic explains concepts associated with the binder and with the process of creating ILE programs or service programs. Before reading this, you should be familiar with the binding concepts described in [“ILE Basic Concepts”](#) on page 15.

Create Program and Create Service Program Commands

The Create Program (CRTPGM) and Create Service Program (CRTSRVPGM) commands look similar and share many of the same parameters. Comparing the parameters used in the two commands helps to clarify how each command can be used.

Table 6 on page 61 shows the commands and their parameters with the default values supplied.

Table 6. Parameters for CRTPGM and CRTSRVPGM Commands

Parameter Group	CRTPGM Command	CRTSRVPGM Command
Identification	PGM(*CURLIB/WORK) MODULE(*PGM)	SRVPGM(*CURLIB/UTILITY) MODULE(*SRVPGM)
Program access	ENTMOD(*FIRST)	EXPORT(*SRCFILE) SRCFILE(*LIBL/ QSRVSR) SRCMBR(*SRVPGM)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Runtime	ACTGRP(*ENTMOD)	ACTGRP(*CALLER)
Optimization	IPA(*NO) IPACTLFILE(*NONE) ARGOPT(*NO)	IPA(*NO) IPACTLFILE(*NONE) ARGOPT(*NO)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF *NOEVENTF) DETAIL(*NONE) ALWUPD(*YES) ALWLIBUPD(*NO) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER) STGMDL(*SNGLVL)	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF *NOEVENTF) DETAIL(*NONE) ALWUPD(*YES) ALWLIBUPD(*NO) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*BLANK) TGTRLS(*CURRENT) USRPRF(*USER) STGMDL(*SNGLVL)

The identification parameters for both commands name the object to be created and the modules copied. The only difference in the two parameters is in the default module name to use when creating the object. For CRTPGM, use the same name for the module as is specified on the program (*PGM) parameter. For CRTSRVPGM, use the same name for the module as is specified on the service program (*SRVPGM) parameter. Otherwise, these parameters look and act the same.

The most significant similarity in the two commands is how the binder resolves symbols between the imports and exports. In both cases, the binder processes the input from the module (MODULE), bound service program (BNDSRVPGM), and binding directory (BNDDIR) parameters.

The most significant difference in the commands is with the program-access parameters (see “[Program Access](#)” on page 70). For the CRTPGM command, all that needs to be identified to the binder is which module has the program entry procedure. Once the program is created and a dynamic program call is made to this program, processing starts with the module containing the program entry procedure. The CRTSRVPGM command needs more program-access information because it can supply an interface of several access points for other programs or service programs.

Use Adopted Authority (QUSEADPAUT)

The QUSEADPAUT system value defines which users can create programs with the use adopted authority (USEADPAUT(*YES)) attribute. All users authorized by the QUSEADPAUT system value can create or change programs and service programs to use adopted authority if they have the necessary authorities. See [Security reference](#) to find out what authorities are required.

The system value can contain the name of an authorization list. The user's authority is checked against this list. If the user has at least *USE authority to the named authorization list, the user can create, change, or update programs or service programs with the USEADPAUT(*YES) attribute. The authority to the authorization list cannot come from adopted authority.

If an authorization list is named in the system value and the authorization list is missing, the function being attempted will not complete. A message is sent indicating this. However, if the program is created with the QPRCPTPG API, and the *NOADPAUT value is specified in the option template, the program will create successfully even if the authorization list does not exist. If more than one function is requested on the command or API, and the authorization list is missing, the function is not performed.

Table 7. Possible Values for QUSEADPAUT

Values	Description
<i>authorizationlist name</i>	A diagnostic message is signaled to indicate that the program is created with USEADPAUT(*NO) if all of the following are true: • An authorization list is specified for the QUSEADPAUT system value. • The user does not have authority to the authorization list mentioned above. • There are no other errors when the program or service program is created. If the user has authority to the authorization list, the program or service program is created with USEADPAUT(*YES).
<u>*NONE</u>	All users authorized by the QUSEADPAUT system value can create or change programs and service programs to use adopted authority if the users have the necessary authorities.

Using optimization parameters

Specify optimization parameters to further optimize your ILE bound programs or service programs. For more information about optimization parameters to use when you create or modify programs, see “[Advanced Optimization Techniques](#)” on page 131.

Stored data in modules and programs

When a module is created, it contains useful data in addition to the compiled code. Depending on the compiler that was used, modules might contain the following types of data:

- Creation Data (*CRTDTA)

This is required to re-create or convert the module (for example, to change its optimization level). Newly created modules always contain creation data.

- Debug Data (*DBGDTA)

This is required to debug the program that the module is bound into. See [“Debugging Considerations” on page 119](#) and [“Interprocedural analysis \(IPA\)” on page 136](#) for more information about debugging.

- Intermediate Language Data (*ILDTA)

This is required for the advanced optimization technique of interprocedural analysis (IPA). See [“Advanced Optimization Techniques” on page 131](#) for more information.

Use the Display Module (DSPMOD) command to find out what kinds of data are stored in a module. The process of binding copies module data into the program or service program that is created. Intermediate Language Data (*ILDTA) is not copied into programs or service programs.

Use the Display Program (DSPPGM) or Display Service Program (DSPSRVPGM) commands with the DETAIL(*MODULE) parameter to see the details. Programs can contain the following kinds of data in addition to the module data:

- Creation Data (*CRTDTA)

This is required for re-creating the program or service program. Newly created programs always contain this creation data.

- Block Order Profiling Data (*BLKORD)

This is generated for application profiling. See [“Advanced Optimization Techniques” on page 131](#) for more information.

- Procedure Order Profiling Data (*PRCORD)

This is generated for application profiling. See [“Advanced Optimization Techniques” on page 131](#) for more information.

Creation Data (*CRTDTA) can exist for the program itself and for each bound module. When the data is created and stored with the module or program, it is observable. The operating system can use the data for operations, such as running the Change Module (CHGMOD) and Change Program (CHGPGM) commands, debugging your program, and using IPA.

You can remove observability with the Remove observable information (RMVOBS) parameter on the Change Module (CHGMOD), Change Program (CHGPGM), and Change Service Program (CHGSRVPGM) commands. If you remove observability, MI programs can no longer access the data.

Note: You cannot undo removing observability.

For most kinds of data, removing observability removes that data from the object. If you no longer need the corresponding function and want to make your object smaller, you can remove the data. However, Creation Data (*CRTDTA) is not removed from the object; it is transformed into an unobservable form. The operating system cannot use the data, but the machine can use unobservable creation data to convert the object.

Symbol Resolution

Symbol resolution is the process the binder goes through to match the following:

- The import requests from the set of modules to be bound by copy
- The set of exports provided by the specified modules and service programs

The set of exports to be used during symbol resolution can be thought of as an ordered (sequentially numbered) list. The order of the exports is determined by the following:

- The order in which the objects are specified on the MODULE, BNDSRVPGM, and BNDDIR parameters of the CRTPGM or CRTSRVPGM command
- The exports from the language runtime routines of the specified modules

Resolved and Unresolved Imports

An import and export each consist of a procedure or data type and a name. An **unresolved import** is one whose type and name do not yet match the type and name of an export. A **resolved import** is one whose type and name exactly match the type and name of an export.

Only the imports from the modules that are bound by copy go into the unresolved import list. During symbol resolution, the next unresolved import is used to search the ordered list of exports for a match. If an unresolved import exists after checking the set of ordered exports, the program object or service program is normally not created. However, if *UNRSLVREF is specified on the option parameter, a program object or service program with unresolved imports can be created. If such a program object or service program tries to use an unresolved import at runtime, error message MCH4439 is issued. That message says, "Attempt to use an import that was not resolved."

Binding by Copy

The modules specified on the MODULE parameter are always bound by copy. Modules named in a binding directory specified by the BNDDIR parameter are bound by copy if they are needed. A module named in a binding directory is needed in either of the following cases:

- The module provides an export for an unresolved import
- The module provides an export named in the current export block of the binder language source file being used to create a service program

If an export found in the binder language comes from a module object, that module is always bound by copy, regardless of whether it was explicitly provided on the command line or comes from a binding directory. For example,

```
Module M1:  imports P2
Module M2:  exports P2
Module M3:  exports P3
Binder language S1:  STRPGMEXP PGMLVL(*CURRENT)
                     EXPORT P3
                     ENDPGMEXP
Binding directory BNDDIR1:  M2
                           M3
CRTSRVPGM SRVPGM(MYLIB/SRV1) MODULE(MYLIB/M1) SRCFILE(MYLIB/S1)
                     SRCMBR(S1) BNDDIR(MYLIB/BNDDIR1)
```

Service program SRV1 will have three modules: M1, M2, and M3. M3 is copied because P3 is in the current export block.

Binding by Reference

Service programs specified on the BNDSRVPGM parameter are bound by reference. If a service program named in a binding directory provides an export for an unresolved import, that service program is bound by reference. A service program bound in this way does not add new imports.

Note: To better control what gets bound to your program, specify the generic service program name or specific libraries. The value *LIBL should only be specified in a user-controlled environment when you know exactly what is getting bound to your program. Do not specify BNDSRVPGM(*LIBL/*ALL) with OPTION(*DUPPROC *DUPVAR). Specifying *LIBL with *ALL may give you unpredictable results at program runtime.

Binding Large Numbers of Modules

For the module (MODULE) parameter on the CRTPGM and CRTSRVPGM commands, there is a limit on the number of modules you can specify. If the number of modules you want to bind exceeds the limit, you can use one of the following methods:

1. Use binding directories to bind a large number of modules that provide exports that are needed by other modules.

2. Use a module naming convention that allows generic module names to be specified on the MODULE parameter on the CRTPGM and CRTSRVPGM commands. For example, CRTPGM PGM(mylib/payroll) MODULE(mylib/pay*). All modules with names starting with pay are unconditionally included in the program mylib/payroll. Therefore, pick your naming convention carefully so that the generic names specified on the CRTPGM or CRTSRVPGM commands do not bind unwanted modules.
3. Group the modules into separate libraries so that the value *ALL can be used with specific library names on the MODULE parameter. For example, CRTPGM PGM(mylib/payroll) MODULE(payload/*ALL). Every module in the library payroll is unconditionally included in the program mylib/payroll.
4. Use a combination of generic names and specific libraries that are described in method 2 and 3.
5. For service programs, use the binding source language. An export specified in the binding source language causes a module to be bound if it satisfies the export. The RTVBNDSRC command can help you create your binding source language. Although the MODULE parameter on the RTVBNDSRC command limits the number of modules that can be explicitly specified on the MODULE parameter, you can use generic module names and the value *ALL with specific library names. You can use the RTVBNDSRC command multiple times with output directed to the same source file. However, you may need to edit the binding source language in this case.

Duplicate Symbols

The symbols for variable names and procedure names can be exported from modules as either weak exports or strong exports.

Weak exports are supported by various programming languages as described in section [“Import and Export Concepts”](#) on page 72. When a module defines and references a weak symbol, the symbol is both weakly exported and imported. Several modules can define the same symbol as a weak export. During symbol resolution, the export from one module is chosen to satisfy all the matching imports so all modules refer to the same item.

Strong exports are generally defined by a single module. If more than one module defines the symbol as a strong export, an error message is issued and program creation will fail. On the program creation commands (CRTPGM, CRTSRVPGM, UPDPGM, UPDSRVPGM), you can use OPTION(*DUPPROC *DUPVAR *WARN) to allow the command to continue despite the duplicate symbol export warning messages. These error messages are important: when a module refers to a symbol that it exports, the reference is made as a local reference, not as an import reference. If two modules define the same symbol as a strong export, and each module refers to the symbol, they are referring to separate items. Carefully examine these messages to ensure your application will function correctly. You may need to change the strong exports to weak exports or change the code so that only a single module exports the symbol and all other modules import it.

Importance of the Order of Exports

With only a slight change to the command, you can create a different, but potentially equally valid, program. The order in which objects are specified on the MODULE, BNDSRVPGM, and BNDDIR parameters is usually important only if both of the following are true:

- Multiple modules or service programs are exporting duplicate symbol names
- Another module needs to import the symbol name

Most applications do not have duplicate symbols, and programmers seldom need to worry about the order in which the objects are specified. For those applications that have duplicate symbols exported that are also imported, consider the order in which objects are listed on CRTPGM or CRTSRVPGM commands.

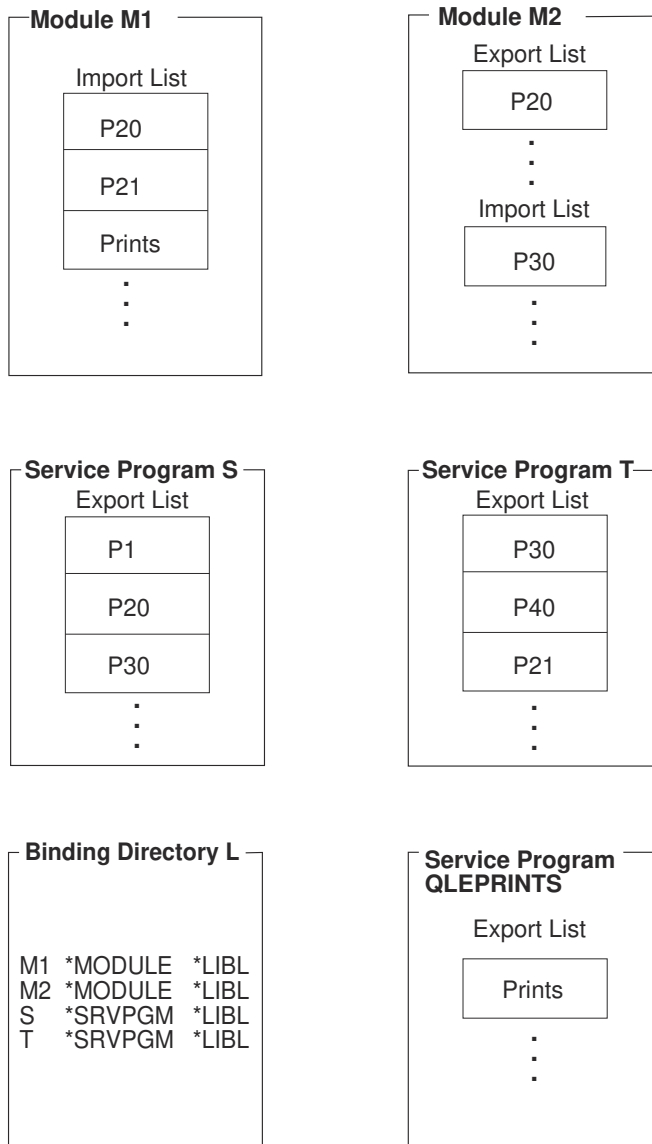
The following examples show how symbol resolution works. The modules, service programs, and binding directories in [Figure 29](#) on page 66 are used for the CRTPGM requests in [Figure 30](#) on page 67 and [Figure 31](#) on page 69. Assume that all the identified exports and imports are procedures.

The examples also show the role of binding directories in the program-creation process. Assume that library MYLIB is in the library list for the CRTPGM and CRTSRVPGM commands. The following command creates binding directory L in library MYLIB:

```
CRTBNDDIR BNDDIR(MYLIB/L)
```

The following command adds the names of modules M1 and M2 and of service programs S and T to binding directory L:

```
ADDBNDDIRE BNDDIR(MYLIB/L) OBJ((M1 *MODULE) (M2 *MODULE) (S) (T))
```



RV2W1054-3

Figure 29. Modules, Service Programs, and Binding Directory

Program Creation Example 1

Assume that the following command is used to create program A in [Figure 30 on page 67](#):

```
CRTPGM PGM(TEST/A)
MODULE(*LIBL/M1)
BNDSRVPGM(*LIBL/S)
BNDDIR(*LIBL/L)
OPTION(*DUPPROC)
```

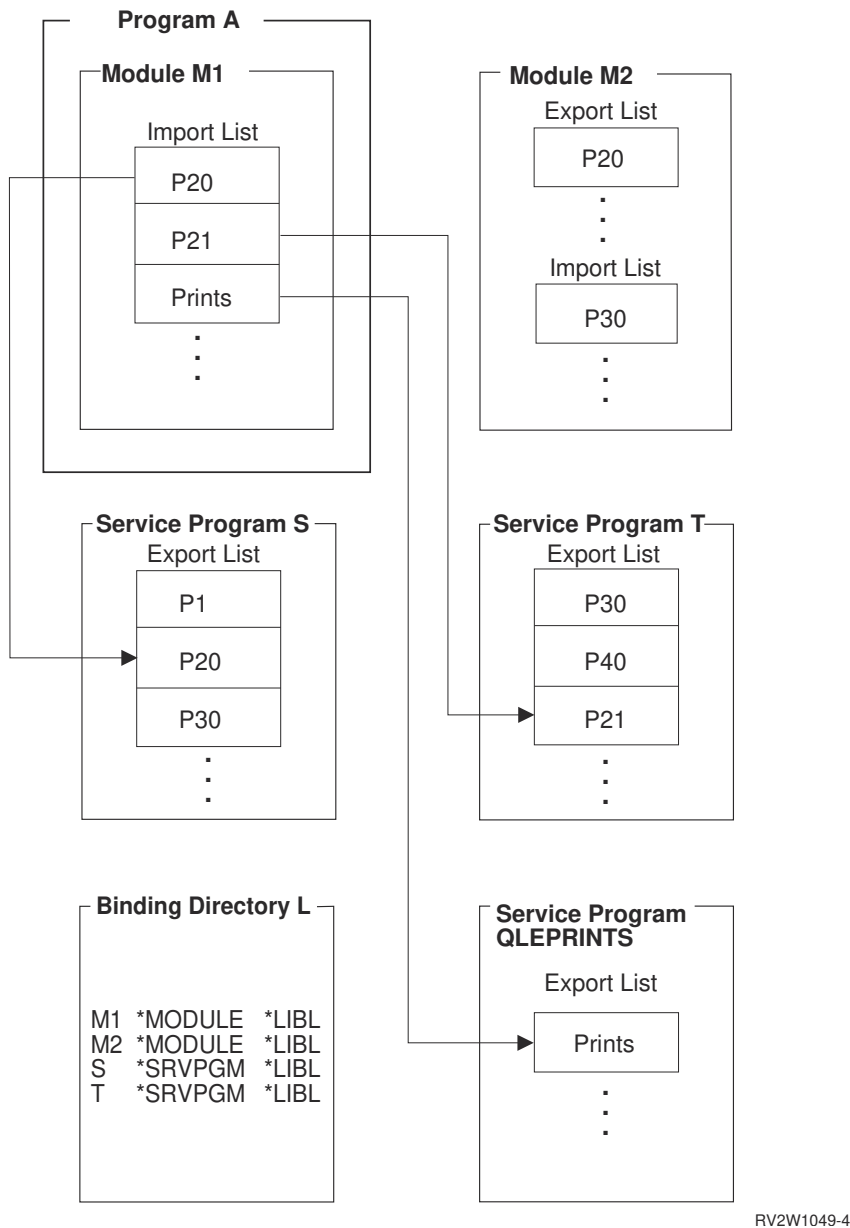


Figure 30. Symbol Resolution and Program Creation: Example 1

To create program A, the binder processes objects specified on the CRTPGM command parameters in the order specified:

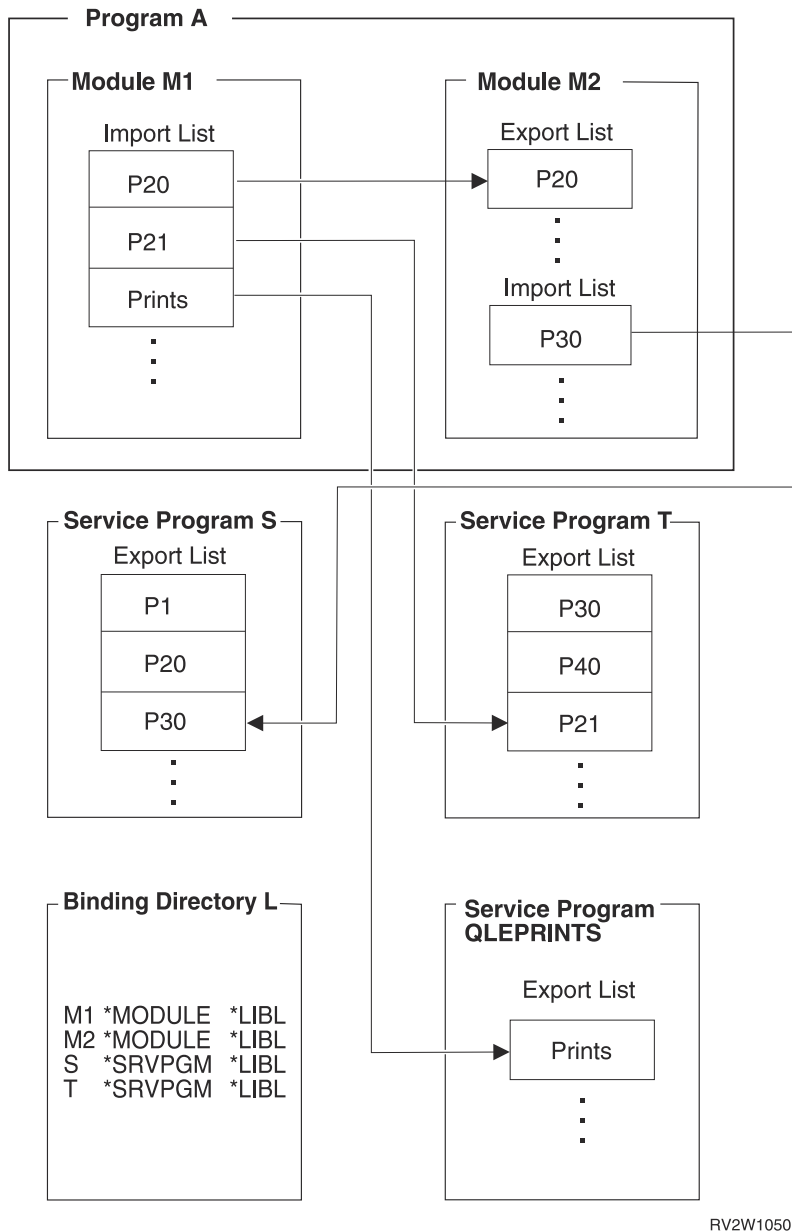
1. The value specified on the first parameter (PGM) is A, which is the name of the program to be created.
2. The value specified on the second parameter (module) is M1. The binder starts there. Module M1 contains three imports that need to be resolved: P20, P21, and Prints.
3. The value specified on the third parameter (BNDSRVPGM) is S. The binder scans the export list of service program S for any procedures that resolve any unresolved import requests. Because the export list contains procedure P20, that import request is resolved.
4. The value specified on the fourth parameter (BNDDIR) is L. The binder next scans binding directory L.
 - a. The first object specified in the binding directory is module M1. Module M1 is currently known because it was specified on the module parameter, but it does not provide any exports.
 - b. The second object specified in the binding directory is module M2. Module M2 provides exports, but none of them match any currently unresolved import requests (P21 and Prints).

- c. The third object specified in the binding directory is service program S. Service program S was already processed in step “3” on page 67 and does not provide any additional exports.
 - d. The fourth object specified in the binding directory is service program T. The binder scans the export list of service program T. Procedure P21 is found, which resolves that import request.
5. The final import that needs to be resolved (Prints) is not specified on any parameter. Nevertheless, the binder finds the Prints procedure in the export list of service program QLEPRINTS, which is a common runtime routine provided by the compiler in this example. When compiling a module, the compiler specifies as the default the binding directory containing its own runtime service programs and the ILE runtime service programs. That is how the binder knows that it should look for any remaining unresolved references in the runtime service programs provided by the compiler. If, after the binder looks in the runtime service programs, there are references that cannot be resolved, the bind normally fails. However, if you specify OPTION(*UNRSLVREF) on the create command, the program is created.

Program Creation Example 2

Figure 31 on page 69 shows the result of a similar CRTPGM request, except that the service program on the BNDSRVPGM parameter has been removed:

```
CRTPGM  PGM(TEST/A)
        MODULE(*LIBL/M1)
        BNDDIR(*LIBL/L)
        OPTION(*DUPPROC)
```



RV2W1050-4

Figure 31. Symbol Resolution and Program Creation: Example 2

The change in ordering of the objects to be processed changes the ordering of the exports. It also results in the creation of a program that is different from the program created in example 1. Because service program S is not specified on the BNDSRVPGM parameter of the CRTPGM command, the binding directory is processed. Module M2 exports procedure P20 and is specified in the binding directory ahead of service program S. Therefore, module M2 gets copied to the resulting program object in this example. When you compare Figure 30 on page 67 with Figure 31 on page 69 you see the following:

- Program A in example 1 contains only module M1 and uses procedures from service programs S, T, and QLEPRINTS.
- In program A of example 2, two modules called M1 and M2 use service programs T and QLEPRINTS.

The program in example 2 is created as follows:

1. The first parameter (PGM) specifies the name of the program to be created.
2. The value specified on the second parameter (MODULE) is M1, so the binder again starts there. Module M1 contains the same three imports that need to be resolved: P20, P21, and Prints.

3. This time, the third parameter specified is not BNDSRVPGM. It is BNDDIR. Therefore, the binder first scans the binding directory specified (L).
 - a. The first entry specified in the binding directory is module M1. Module M1 from this library was already processed by the module parameter.
 - b. The second entry specified in the binding directory is for module M2. The binder scans the export list of module M2. Because that export list contains P20, that import request is resolved. Module M2 is bound by copy and its imports must be added to the list of unresolved import requests for processing. The unresolved import requests are now P21, Prints, and P30.
 - c. Processing continues to the next object that is specified in the binding directory, the 'S' service program. Here, the service program S provides the P30 export for currently unresolved import requests of P21 and Prints. Processing continues to the next object that is listed in the binding directory, service program T.
 - d. Service program T provides export P21 for the unresolved import.
4. As in example 1, import request Prints is not specified. However, the procedure is found in the runtime routines provided by the language in which module M1 was written.

Symbol resolution is also affected by the strength of the exports. For information about strong and weak exports, see Export in [“Import and Export Concepts”](#) on page 72.

Program Access

When you create an ILE program object or service program object, you need to specify how other programs can access that program. On the CRTPGM command, you do so with the entry module (ENTMOD) parameter. On the CRTSRVPGM command, you do so with the export (EXPORT) parameter (see [Table 6](#) on page 61).

Program Entry Procedure Module Parameter on the CRTPGM Command

The program entry procedure module (ENTMOD) parameter tells the binder the name of the module in which the following are located:

- Program entry procedure (PEP)
- User entry procedure (UEP)

This information identifies which module contains the PEP that gets control when making a dynamic call to the program that is created.

The default value for the ENTMOD parameter is *FIRST. This value specifies that the binder uses as the entry module the first module it finds in the list of modules specified on the module parameter that contains a PEP.

If the following conditions exist:

- *FIRST is specified on the ENTMOD parameter
- A second module with a PEP is encountered

the binder copies this second module into the program object and continues the binding process. The binder ignores the additional PEP.

If *ONLY is specified on the ENTMOD parameter, only one module in the program can contain a PEP. If *ONLY is specified and a second module with a PEP is encountered, the program is not created.

For explicit control, you can specify the name of the module that contains the PEP. Any other PEPs are ignored. If the module explicitly specified does not contain a PEP, the CRTPGM request fails.

To see whether a module has a program entry procedure, you use the display module (DSPMOD) command. The information appears in the *Program entry procedure name* field of the Display Module Information display. If *NONE is specified in the field, this module does not have a PEP. If a name is specified in the field, this module has a PEP.

Export Parameter on the CRTSRVPGM Command

The export (EXPORT), source file (SRCFILE), source member (SRCMBR), and source stream file (SRCSTMF) parameters identify the public interface to the service program being created. The parameters specify the exports (procedures and data) that a service program makes available for use by other ILE programs or service programs.

The default value for the export parameter is *SRCFILE. That value directs the binder to the SRCFILE or SRCSTMF parameter for a reference to information about exports of the service program. This additional information is a source file with binder language source in it (see [“Binder Language”](#) on page 73). The binder locates the binder language source and, from the specified names to be exported, generates one or more signatures. The binder language also allows you to specify a signature of your choice instead of having the binder generate one.

The Retrieve Binder Source (RTVBNDSRC) command can be used to create a source file that contains binder language source. The source can be based on either an existing service program or a set of modules. If based on a service program, the source is appropriate for recreating or updating that service program. If based on a set of modules, the source contains all symbols eligible to be exported from the modules. In either case, you can edit this file to include only the symbols you want to export, then you can specify this file using the SRCFILE or SRCSTMF parameter of the CRTSRVPGM or UPDSRVPGM commands.

The other possible value for the export parameter is *ALL. When EXPORT(*ALL) is specified, all of the symbols exported from the copied modules are exported from the service program. The signature that gets generated is determined by the following:

- The number of exported symbols
- Alphabetical order of exported symbols

If EXPORT(*ALL) is specified, no binder language is needed to define the exports from a service program. By specifying this value, you do not need to generate the binder language source. However, a service program with EXPORT(*ALL) specified can be difficult to update or correct if the exports are used by other programs. If the service program is changed, the order or number of exports might change. Therefore, the signature of that service program might change. If the signature changes, all programs or service programs that use the changed service program have to be re-created.

EXPORT(*ALL) indicates that all symbols exported from the modules used in the service program are exported from the service program. ILE C can define exports as global or static. Only external variables declared in ILE C as global are available with EXPORT(*ALL). In ILE RPG, the following are available with EXPORT(*ALL):

- The RPG main procedure name
- The names of any exported subprocedures
- Variables defined with the keyword EXPORT

In ILE COBOL, the following language elements are module exports:

- The name in the PROGRAM-ID paragraph in the lexically outermost COBOL program (not to be confused with *PGM object) of a compilation unit. This maps to a strong procedure export.
- The COBOL compiler-generated name derived from the name in the PROGRAM-ID paragraph in the preceding bullet if that program does not have the INITIAL attribute. This maps to a strong procedure export. For information about strong and weak exports, see Export in [“Import and Export Concepts”](#) on page 72.
- Any data item or file item declared as EXTERNAL. This maps to a weak export.

Export Parameter Used with Source File and Source Member Parameters

The default value on the export parameter is *SRCFILE. If *SRCFILE is specified on the export parameter, the binder must also use either the SRCFILE and SRCMBR parameters or the SRCSTMF parameter to locate the binder language source.

The following example command binds a service program named UTILITY by using the defaults to locate the binder language source:

```
CRTSRVPGM SRVPGM(*CURLIB/UTILITY)
          MODULE(*SRVPGM)
          EXPORT(*SRCFILE)
          SRCFILE(*LIBL/QSRVSRCL)
          SRCMBR(*SRVPGM)
```

For this command to create the service program, a member named UTILITY must be in the source file QSRVSRCL. This member must then contain the binder language source that the binder translates into a signature and set of export identifiers. The default is to get the binder language source from a member with the same name as the name of the service program, UTILITY. If a file, member, or binder language source with the values supplied on these parameters is not located, the service program is not created.

Maximum width of a file for the SRCFILE parameter

In V3R7 or later releases, the maximum width of a file for the Source File (SRCFILE) parameter on the CRTSRVPGM or UPDSRVPGM command is 240 characters. If the file is larger than the maximum width, message CPF5D07 appears. For V3R2, the maximum width is 80 characters. For V3R6, V3R1 and V2R3, there is no limit on the maximum width.

Starting in IBM i 7.3, the binder source can be processed out of stream files (SRCSTMF parameter). Even though there is no record length for a stream file since a stream file by definition is a stream of bytes, the maximum length of a line (the number of characters between newline characters) that can be processed by the binder language compiler is 240 characters.

Import and Export Concepts

ILE languages support the following types of exports and imports:

- Weak data exports
- Weak data imports
- Strong data exports
- Strong data imports
- Strong procedure exports
- Weak procedure exports
- Procedure imports

An ILE module object can export procedures or data items to other modules. And an ILE module object can import (reference) procedures or data items from other modules. When using a module object on CRTSRVPGM command to create a service program, its exports optionally export from the service program. (See [“Export Parameter on the CRTSRVPGM Command”](#) on page 71.) The strength (strong or weak) of an export depends on the programming language. The strength determines when enough is known about an export to set its characteristics, such as the size of a data item. A strong export's characteristics are set at bind time. The strength of the exports affects symbol resolution.

- The binder uses the characteristics of the strong export, if one or more weak exports have the same name.
- If a weak export does not have the same name as a strong export, you cannot set its characteristics until activation time. At activation time, if multiple weak exports with the same name exist, the program uses the largest one. This is true, unless an already activated weak export with the same name has already set its characteristics.
- At bind time, if a binding directory is used, and weak exports are found to match weak imports, they will be bound. However, the binding directory is searched only as long as there are unresolved imports to be resolved. Once all imports are resolved, the search through the binding directory entries stops. Duplicate weak exports are not flagged as duplicate variables or procedures. The order of items in the binding directory is very important.

You can export weak exports outside a program object or service program for resolution at activation time. This is opposed to strong exports that you export only outside a service program and only at bind time.

You cannot, however, export strong exports outside a program object. You can export strong procedure exports outside a service program to satisfy either of the following at bind time:

- Imports in a program that binds the service program by reference.
- Imports in other service programs that are bound by reference to that program.

Service programs define their public interface through binding source language.

You can make weak procedure exports part of the public interface for a service program through the binding source language. However, exporting a weak procedure export from the service program through the binding source language no longer marks it as weak. It is handled as a strong procedure export.

You can only export weak data to an activation group. You cannot make it part of the public interface that is exported from the service program through the use of binder source language. Specifying weak data in the binder source language causes the bind to fail.

Table 8 on page 73 summarizes the types of imports and exports that are supported by some of the ILE languages:

Table 8. Imports and Exports Supported by ILE Languages							
ILE Languages	Weak Data Exports	Weak Data Imports	Strong Data Exports	Strong Data Imports	Strong Procedure Exports	Weak Procedure Exports	Procedure Imports
RPG IV	No	No	Yes	Yes	Yes	No	Yes
COBOL ²	Yes ³	Yes ³	No	No	Yes ¹	No	Yes
CL	No	No	No	No	Yes ¹	No	Yes
C	No	No	Yes	Yes	Yes	No	Yes
C++	No	No	Yes	Yes	Yes	Yes	Yes
Note: 1. COBOL and CL allow only one procedure to be exported from the module. 2. COBOL uses the weak data model. Data items that are declared as external become both weak exports and weak imports for that module. 3. COBOL requires the NOMONOPRC option. Without this option, the lowercase letters are automatically converted to uppercase.							

For information on which declarations become imports and exports for a particular language, see one of the following books:

- [ILE RPG Programmer's Guide](#) 
- [ILE COBOL Programmer's Guide](#) 
- [ILE C/C++ Programmer's Guide](#) 

Binder Language

The **binder language** is a small set of nonrunnable commands that defines the exports for a service program. The binder language enables the source entry utility (SEU) syntax checker to prompt and validate the input when a BND source type is specified.

Note: You cannot use the SEU syntax checking type BND for a binder source file that contains wildcarding. You also cannot use it for a binder source file that contains names longer than 254 characters.

The binder language consists of a list of the following commands:

1. Start Program Export (STRPGMEXP) command, which identifies the beginning of a list of exports from a service program
2. Export Symbol (EXPORT) commands, each of which identifies a symbol name available to be exported from a service program
3. End Program Export (ENDPGMEXP) command, which identifies the end of a list of exports from a service program

Figure 32 on page 74 is a sample of the binder language in a source file:

```
STRPGMEXP PGMLVL(*CURRENT) LVLCHK(*YES)
.
.
EXPORT SYMBOL(p1)
EXPORT SYMBOL('p2')
EXPORT SYMBOL('P3')
.
.
ENDPGMEXP
.
.
.
STRPGMEXP PGMLVL(*PRV)
.
.
EXPORT SYMBOL(p1)
EXPORT SYMBOL('p2')
.
.
ENDPGMEXP
```

Figure 32. Example of Binder Language in a Source File

Using the Retrieve Binder Source (RTVBNDSRC) command, you can generate the binder language source based on exports from one or more modules or service programs.

Signature

The symbols identified between a STRPGMEXP PGMLVL(*CURRENT) and ENDPGMEXP pair define the public interface to a service program. That public interface is represented by a *signature*. A signature is a value that identifies the interface supported by a service program.

Note: Do not confuse the signatures discussed in this topic with *digital object signatures*. Digital signatures on IBM i objects ensure the integrity of software and data. They also act as a deterrent to data tampering or the unauthorized modification to an object. The signature also provides positive identification of the data's origin. For more information about digital object signatures, see the Security category of information in the IBM i Information Center.

If you choose to specify an explicit signature, your binder language source only needs to have one export block; new exports can be added to the end of the list of exports. If you choose not to specify an explicit signature, the binder generates a signature from the list of procedure and data item names to be exported and from the order in which they are specified. You must add a new export block to your binder source every time you add new exports to your service program.

Note: To avoid making incompatible changes to a service program, existing procedure and data item names must not be removed or rearranged in the binder language source. Additional export blocks must contain the same symbols in the same order as existing export blocks. Additional symbols must be added only to the end of the list. This rule applies whether you specify an explicit signature, or whether you allow the binder to generate a new signature.

There is no way to remove a service program export in a way compatible with existing programs and service programs because that export might be needed by programs or service programs bound to that service program.

If an incompatible change is made to a service program, existing programs that remain bound to it might no longer work correctly. An incompatible change to a service program can be made only if it can be guaranteed that all programs and service programs bound to it are re-created with the CRTPGM or CRTSRVPGM command after the incompatible change is made.

A signature does not validate the interface to a particular procedure within a service program. An incompatible change to the interface of a particular procedure requires that all modules that call the procedure be recompiled, and all programs and service programs that contain those modules be re-created with CRTPGM or CRTSRVPGM.

Start Program Export and End Program Export Commands

The Start Program Export (STRPGMEXP) command identifies the beginning of a list of exports from a service program. The End Program Export (ENDPGMEXP) command identifies the end of a list of exports from a service program.

Multiple STRPGMEXP and ENDPGMEXP pairs specified within a source file cause multiple signatures to be created. The order in which the STRPGMEXP and ENDPGMEXP pairs occur is not significant.

Program Level Parameter on the STRPGMEXP Command

Only one STRPGMEXP command can specify PGMLVL(*CURRENT), but it does not have to be the first STRPGMEXP command. All other STRPGMEXP commands within a source file must specify PGMLVL(*PRV). The current signature represents whichever STRPGMEXP command has PGMLVL(*CURRENT) specified.

Signature Parameter on the STRPGMEXP Command

The signature (SIGNATURE) parameter allows you to explicitly specify a signature for a service program. The explicit signature can be a hexadecimal string or a character string. You may want to consider explicitly specifying a signature for either of the following reasons:

- The binder could generate a compatible signature that you do not want. A signature is based on the names of the specified exports and on their order. Therefore, if two export blocks have the same exports in the same order, they have the same signature. As the service program provider, you may know that the two interfaces are not compatible (because, for example, their parameter lists are different). In this case, you can explicitly specify a new signature instead of having the binder generate the compatible signature. If you do so, you create an incompatibility in your service program, forcing some or all clients to recompile.
- The binder could generate an incompatible signature that you do not want. If two export blocks have different exports or a different order, they have different signatures. If, as the service program provider, you know that the two interfaces are really compatible (because, for example, a function name has changed but it is still the same function), you can explicitly specify the same signature as previously generated by the binder instead of having the binder generate an incompatible signature. If you specify the same signature, you maintain a compatibility in your service program, allowing your clients to use your service program without rebinding.

The default value for the signature parameter, *GEN, causes the binder to generate a signature from exported symbols.

You can determine the signature values for a service program by using the Display Service Program (DSPSRVPGM) command and specifying DETAIL(*SIGNATURE).

Level Check Parameter on the STRPGMEXP Command

The level check (LVLCHK) parameter on the STRPGMEXP command specifies whether the binder will automatically check the public interface to a service program. Specifying LVLCHK(*YES), or using the

default value LVLCHK(*YES), causes the binder to examine the signature at runtime. The system verifies that the value matches the value known to the service program's clients. If the values match, clients of the service program can use the public interface without rebinding to the service program.

Use the LVLCHK(*NO) value with caution. If you cannot control the public interface, runtime or activation errors might occur. See [“Binder Language Errors” on page 171](#) for an explanation of the common errors that might occur from using the binder language.

Export Symbol Command

The Export Symbol (EXPORT) command identifies a symbol name available to be exported from a service program.

If the exported symbols contain lowercase letters, the symbol name should be enclosed within apostrophes as in [Figure 32 on page 74](#). If apostrophes are not used, the symbol name is converted to all uppercase letters. In the example, the binder searches for an export named P1, not p1.

Symbol names can also be exported through the use of wildcard characters (<<< or >>>). If a symbol name exists and matches the wildcard specified, the symbol name is exported. If any of the following conditions exists, an error is signaled and the service program is not created:

- No symbol name matches the wildcard specified
- More than one symbol name matches the wildcard specified
- A symbol name matches the wildcard specified but is not available for export

Substrings in the wildcard specification must be enclosed within quotation marks.

Signatures are determined by the characters in wildcard specifications. Changing the wildcard specification changes the signature even if the changed wildcard specification matches the same export. For example, the two wildcard specifications "r">>> and "ra">>> both export the symbol "rate" but they create two different signatures. Therefore, it is strongly recommended that you use a wildcard specification that is as similar to the export symbol as possible.

Note: You cannot use the SEU syntax checking type BND for a binder source file that contains wildcarding.

Wildcard Export Symbol Examples

For the following examples, assume that the symbol list of possible exports consists of:

```
interest_rate  
international  
prime_rate
```

The following examples show which export is chosen or why an error occurs:

EXPORT SYMBOL ("interest">>>)

Exports the symbol "interest_rate" because it is the only symbol that begins with "interest".

EXPORT SYMBOL ("i">>>"rate">>>)

Exports the symbol "interest_rate" because it is the only symbol that begins with "i" and subsequently contains "rate".

EXPORT SYMBOL (<<<"i">>>"rate")

Results in a "Multiple matches for wildcard specification" error. Both "prime_rate" and "interest_rate" contain an "i" and subsequently end in "rate".

EXPORT SYMBOL ("inter">>>"prime")

Results in a "No matches for wildcard specification" error. No symbol begins with "inter" and subsequently ends in "prime".

EXPORT SYMBOL (<<<)

Results in a "Multiple matches for wildcard specification" error. This symbol matches all three symbols and therefore is not valid. An export statement can result in only one exported symbol.

Binder Language Examples

As an example of using the binder language, assume that you are developing a simple financial application with the following procedures:

- Rate procedure

Calculates an Interest_Rate, given the values of Loan_Amount, Term_of_Payment, and Payment_Amount.

- Amount procedure

Calculates the Loan_Amount, given the values of Interest_Rate, Term_of_Payment, and Payment_Amount.

- Payment procedure

Calculates the Payment_Amount, given the values of Interest_Rate, Term_of_Payment, and Loan_Amount.

- Term procedure

Calculates the Term_of_Payment, given the values of Interest_Rate, Loan_Amount, and Payment_Amount.

Some of the output listings for this application are shown in [“Output Listing from CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM Command”](#) on page 163.

In the binder language examples, each module contains more than one procedure. The examples apply even to modules that contain only one procedure.

Binder Language Example 1

The binder language for the Rate, Amount, Payment, and Term procedures looks like the following:

```
FILE: MYLIB/QSRVSRC  MEMBER: FINANCIAL

STRPGMEXP  PGMLVL(*CURRENT)
EXPORT SYMBOL('Term')
EXPORT SYMBOL('Rate')
EXPORT SYMBOL('Amount')
EXPORT SYMBOL('Payment')
ENDPGMEXP
```

Some initial design decisions have been made, and three modules (MONEY, RATES, and CALCS) provide the necessary procedures.

To create the service program pictured in [Figure 33 on page 78](#), the binder language is specified on the following CRTSRVPGM command:

```
CRTSRVPGM  SRVPGM(MYLIB/FINANCIAL)
           MODULE(MYLIB/MONEY MYLIB/RATES MYLIB/CALCS)
           EXPORT(*SRCFILE)
           SRCFILE(MYLIB/QSRVSRC)
           SRCMBR(*SRVPGM)
```

Note that source file QSRVSRC in library MYLIB, specified in the SRCFILE parameter, is the file that contains the binder language source.

Also note that no binding directory is needed because all the modules needed to create the service program are specified on the MODULE parameter.

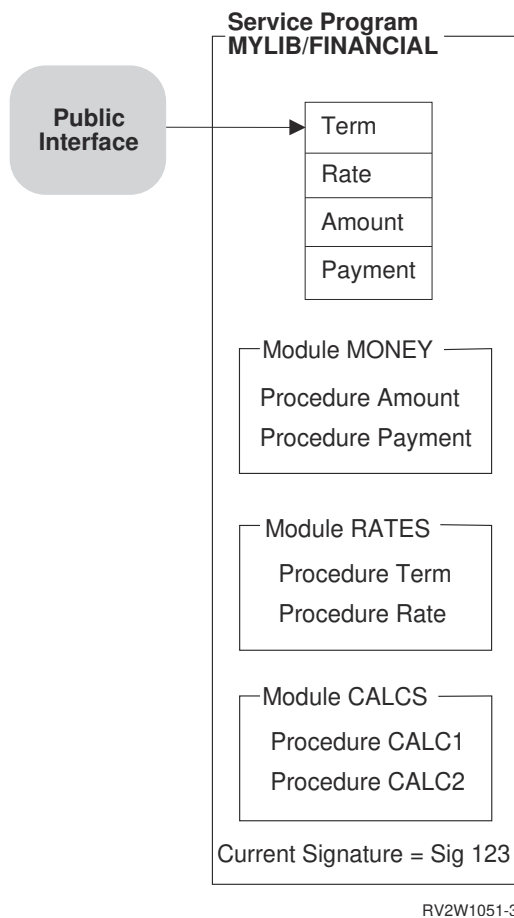


Figure 33. Creating a Service Program by Using the Binder Language

Binder Language Example 2

As progress is made in developing the application, a program called **BANKER** is written. **BANKER** needs to use the procedure called **Payment** in the service program called **FINANCIAL**. The resulting application with the **BANKER** program is shown in [Figure 34 on page 79](#).

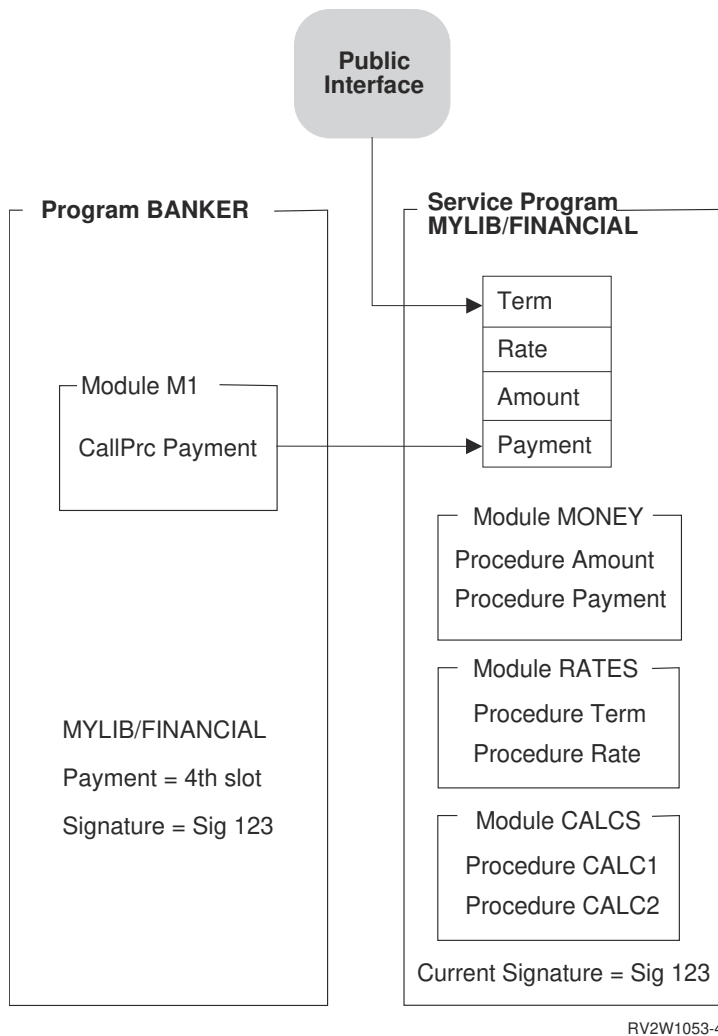


Figure 34. Using the Service Program FINANCIAL

When the BANKER program was created, the MYLIB/FINANCIAL service program was provided on the BNDSRVPGM parameter. The symbol Payment was found to be exported from the fourth slot of the public interface of the FINANCIAL service program. The current signature of MYLIB/FINANCIAL along with the slot associated with the Payment interface is saved with the BANKER program.

During the process of getting BANKER ready to run, activation verifies the following:

- Service program FINANCIAL in library MYLIB can be found.
- The service program still supports the signature (SIG 123) saved in BANKER.

This signature checking verifies that the public interface used by BANKER when it was created is still valid at runtime.

As shown in Figure 34 on page 79, at the time BANKER gets called, MYLIB/FINANCIAL still supports the public interface used by BANKER. If activation cannot find either a matching signature in MYLIB/FINANCIAL or the service program MYLIB/FINANCIAL, the following occurs:

BANKER fails to get activated.
An error message is issued.

Binder Language Example 3

As the application continues to grow, two new procedures are needed to complete our financial package. The two new procedures, OpenAccount and CloseAccount, open and close the accounts, respectively. The

following steps need to be performed to update MYLIB/FINANCIAL such that the program BANKER does not need to be re-created:

1. Write the procedures OpenAccount and CloseAccount.
2. Update the binder language to specify the new procedures.

The updated binder language supports the new procedures. It also allows the existing ILE programs or service programs that use the FINANCIAL service program to remain unchanged. The binder language looks like this:

```
FILE: MYLIB/QSRVSRC  MEMBER: FINANCIAL

STRPGMEXP  PGMLVL(*CURRENT)
EXPORT SYMBOL('Term')
EXPORT SYMBOL('Rate')
EXPORT SYMBOL('Amount')
EXPORT SYMBOL('Payment')
EXPORT SYMBOL('OpenAccount')
EXPORT SYMBOL('CloseAccount')
ENDPGMEXP

STRPGMEXP  PGMLVL(*PRV)
EXPORT SYMBOL('Term')
EXPORT SYMBOL('Rate')
EXPORT SYMBOL('Amount')
EXPORT SYMBOL('Payment')
ENDPGMEXP
```

When an update operation to a service program is needed to do both of the following:

- Support new procedures or data items
- Allow the existing programs and service programs that use the changed service program to remain unchanged

one of two alternatives must be chosen. The first alternative is to perform the following steps:

1. Duplicate the STRPGMEXP, ENDPGMEXP block that contains PGMLVL(*CURRENT).
2. Change the duplicated PGMLVL(*CURRENT) value to PGMLVL(*PRV).
3. In the STRPGMEXP command that contains PGMLVL(*CURRENT), add to the end of the list the new procedures or data items to be exported.
4. Save the changes to the source file.
5. Create or re-create the new or changed modules.
6. Create the service program from the new or changed modules by using the updated binder language.

The second alternative is to take advantage of the signature parameter on the STRPGMEXP command and to add new symbols at the end of the export block:

```
STRPGMEXP  PGMVAL(*CURRENT) SIGNATURE('123')
EXPORT SYMBOL('Term')
.
.
.
EXPORT SYMBOL('OpenAccount')
EXPORT SYMBOL('CloseAccount')
ENDPGMEXP
```

To create the enhanced service program shown in [Figure 35 on page 81](#), the updated binder language shown in [Binder Language Example 3](#) is used on the following CRTSRVPGM command:

```
CRTSRVPGM  SRVPGM(MYLIB/FINANCIAL)
           MODULE(MYLIB/MONEY MYLIB/RATES MYLIB/CALCS MYLIB/ACCOUNTS))
           EXPORT(*SRCFILE)
           SRCFILE(MYLIB/QSRVSRC)
           SRCMBR(*SRVPGM)
```

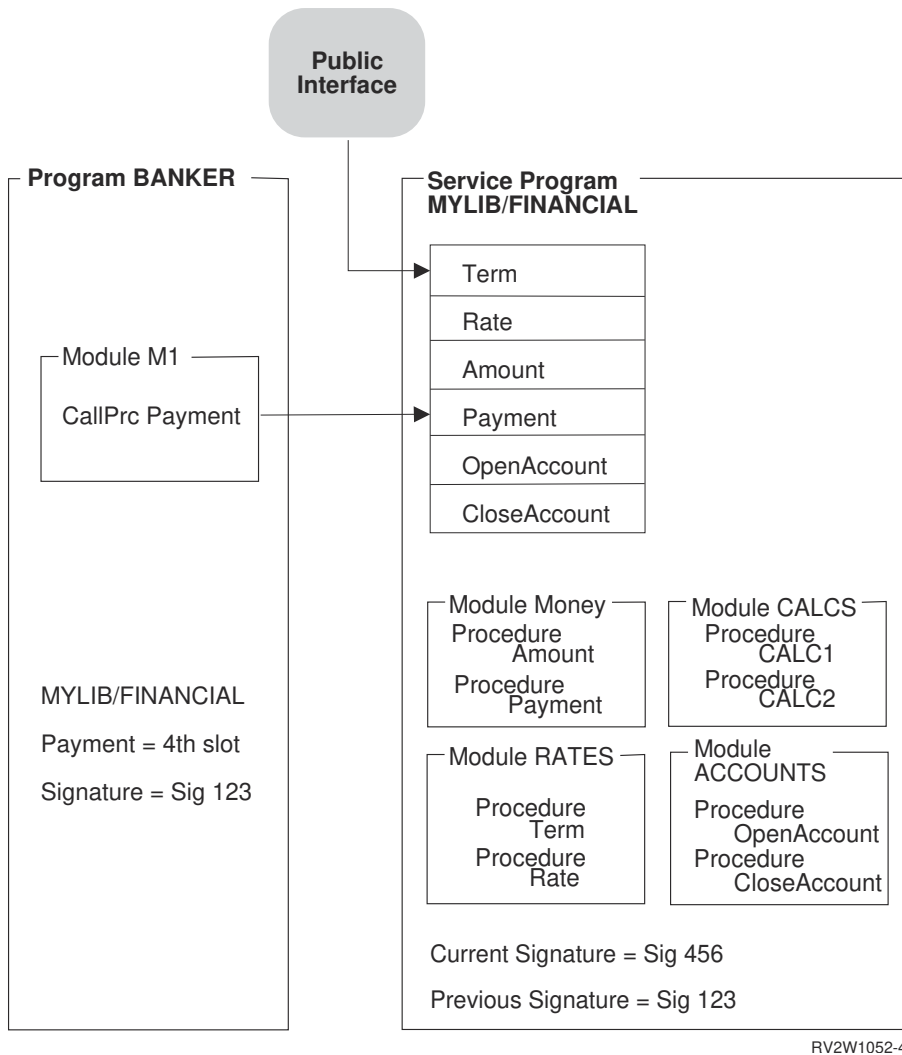


Figure 35. Updating a Service Program by Using the Binder Language

The BANKER program does not have to change because the previous signature is still supported. (See the previous signature in the service program MYLIB/FINANCIAL and the signature saved in BANKER.) If BANKER were re-created by the CRTPGM command, the signature that is saved with BANKER would be the current signature of service program FINANCIAL. The only reason to re-create the program BANKER is if the program used one of the new procedures provided by the service program FINANCIAL. The binder language allows you to enhance the service program without changing the programs or service programs that use the changed service program.

Binder Language Example 4

After shipping the updated FINANCIAL service program, you receive a request to create an interest rate based on the following:

- The current parameters of the Rate procedure
- The credit history of the applicant

A fifth parameter, called **Credit_History**, must be added on the call to the **Rate** procedure. **Credit_History** updates the **Interest_Rate** parameter that gets returned from the **Rate** procedure. Another requirement is that existing ILE programs or service programs that use the FINANCIAL service program must not have to be changed. If the language does not support passing a variable number of parameters, it seems difficult to do both of the following:

- Update the service program

- Avoid re-creating all the other objects that use the FINANCIAL service program

Fortunately, however, there is a way to do this. The following binder language supports the updated Rate procedure. It still allows existing ILE programs or service programs that use the FINANCIAL service program to remain unchanged.

```
FILE: MYLIB/QSRVSRC  MEMBER: FINANCIAL

STRPGMEXP  PGMLVL(*CURRENT)
EXPORT SYMBOL('Term')
EXPORT SYMBOL('Old_Rate') /* Original Rate procedure with four parameters */
EXPORT SYMBOL('Amount')
EXPORT SYMBOL('Payment')
EXPORT SYMBOL('OpenAccount')
EXPORT SYMBOL('CloseAccount')
EXPORT SYMBOL('Rate') /* New Rate procedure that supports +
                        a fifth parameter, Credit_History */
ENDPGMEXP

STRPGMEXP  PGMLVL(*PRV)
EXPORT SYMBOL('Term')
EXPORT SYMBOL('Rate')
EXPORT SYMBOL('Amount')
EXPORT SYMBOL('Payment')
EXPORT SYMBOL('OpenAccount')
EXPORT SYMBOL('CloseAccount')
ENDPGMEXP

STRPGMEXP  PGMLVL(*PRV)
EXPORT SYMBOL('Term')
EXPORT SYMBOL('Rate')
EXPORT SYMBOL('Amount')
EXPORT SYMBOL('Payment')
```

The original symbol Rate was renamed Old_Rate but remains in the same relative position of symbols to be exported. This is important to remember.

A comment is associated with the Old_Rate symbol. A comment is everything between /* and */. The binder ignores comments in the binder language source when creating a service program.

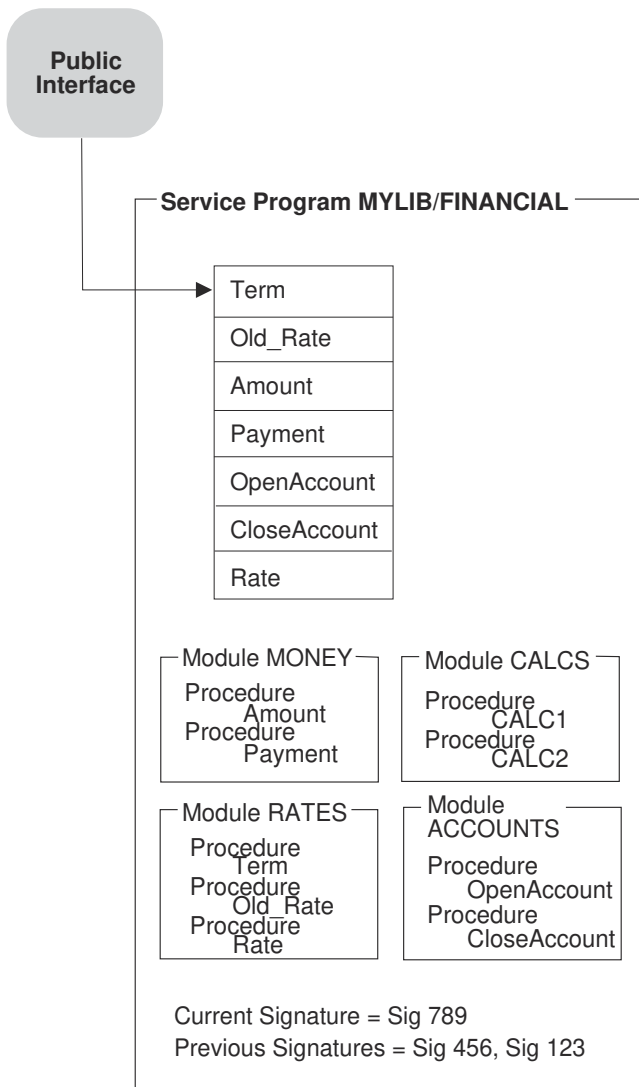
The new procedure Rate, which supports the additional parameter of Credit_History, must also be exported. This updated procedure is added to the end of the list of exports.

The following two ways can deal with the original Rate procedure:

- Rename the original Rate procedure that supports four parameters as Old_Rate. Duplicate the Old_Rate procedure (calling it Rate). Update the code to support the fifth parameter of Credit_History.
- Update the original Rate procedure to support the fifth parameter of Credit_History. Create a new procedure called Old_Rate. Old_Rate supports the original four parameters of Rate. It also calls the new updated Rate procedure with a dummy fifth parameter.

This is the preferred method because maintenance is simpler and the size of the object is smaller.

Using the updated binder language and a new RATES module that supports the procedures Rate, Term, and Old_Rate, you create the following FINANCIAL service program:



RV2W1055-2

Figure 36. Updating a Service Program by Using the Binder Language

The ILE programs and service programs that use the original Rate procedure of the FINANCIAL service program go to slot 2. This directs the call to the Old_Rate procedure, which is advantageous because Old_Rate handles the original four parameters. If any of the ILE programs or service programs that used the original Rate procedure need to be re-created, do one of the following:

- To continue to use the original four-parameter Rate procedure, call the Old_Rate procedure instead of the Rate procedure.
- To use the new Rate procedure, add the fifth parameter, Credit_History, to each call to the Rate procedure.

When an update to a service program must meet the following requirements:

- Support a procedure that changed the number of parameters it can process
- Allow existing programs and service programs that use the changed service program to remain unchanged

the following steps need to be performed:

1. Duplicate the STRPGMEXP, ENDPGMEXP block that contains PGMLVL(*CURRENT).
2. Change the duplicated PGMLVL(*CURRENT) value to PGMLVL(*PRV).

3. In the STRPGMEXP command that contains PGMLVL(*CURRENT), rename the original procedure name, but leave it in the same relative position.
In this example, Rate was changed to Old_Rate but left in the same relative position in the list of symbols to be exported.
4. In the STRPGMEXP command that has PGMLVL(*CURRENT), place the original procedure name at the end of the list that supports a different number of parameters.
In this example, Rate is added to the end of the list of exported symbols, but this Rate procedure supports the additional parameter Credit_History.
5. Save the changes to the binder language source file.
6. In the file containing the source code, enhance the original procedure to support the new parameter.
In the example, this means changing the existing Rate procedure to support the fifth parameter of Credit_History.
7. A new procedure is created that handles the original parameters as input and calls the new procedure with a dummy extra parameter.
In the example, this means adding the Old_Rate procedure that handles the original parameters and calling the new Rate procedure with a dummy fifth parameter.
8. Save the binder language source code changes.
9. Create the module objects with the new and changed procedures.
10. Create the service program from the new and changed modules using the updated binder language.

Changing Programs

The Change Program (CHGPGM) and Change Service Program (CHGSRVPGM) commands change the attributes of a program and service program without requiring recompiling. Some of the changeable attributes follow:

- The optimization attribute.
- The user profile attribute.
- Use adopted authority attribute.
- The profiling data attribute.
- The program text.
- Licensed Internal Code options.
- Storage model (only from *SNGLVL to *INHERIT).

The user can also force recreation of a program even if the specified attributes are the same as the current attributes. Do this by specifying the force program recreation (FRCCRT) parameter with a value of *YES.

The force program recreation (FRCCRT) parameter can also be specified with the values of *NO and *NOCRT. These values determine whether the requested program attributes are actually changed when the change requires that the program be re-created. Modifying the following program attributes may cause the program to be re-created:

- The Optimize program prompt (OPTIMIZE parameter)
- The Use adopted authority prompt (USEADPAUT parameter)
- The Profiling data prompt (PRFDTA parameter)
- The User profile prompt (USRPRF parameter)
- Licensed Internal Code options prompt (LICOPT parameter)
- Storage model prompt (STGMDL parameter)

A value of *NO for the force program recreation (FRCCRT) parameter means that the recreation is not forced, but if one of the program attributes requiring recreation has changed, the program is recreated. This option allows the system to determine whether a change is required.

Recreating a program with CHGPGM or CHGSRVPGM while one or more jobs is using the program causes an "Object Destroyed" exception to occur, and these jobs may fail. By changing the command default for the force program recreation (FRCCRT) parameter to *NOCRT, you can prevent this from inadvertently happening.

You can use the number of threads (NBRTHD) parameter to take advantage of available processing unit cycles, especially on a multiprocessor system.

Program Updates

After an ILE program object or service program is created, you may have to correct an error in it or add an enhancement to it. However, after you service the object, it may be so large that shipping the entire object to your customers is difficult or expensive.

You can reduce the shipment size by using the Update Program (UPDPGM) or Update Service Program (UPDSRVPGM) command. These commands replace only the specified modules, and only the changed or added modules have to be shipped to your customers.

If you use the PTF process, an exit program containing one or more calls to the UPDPGM or UPDSRVPGM commands can be used to do the update functions. Binding the same module to multiple program objects or service programs requires running the UPDPGM or UPDSRVPGM command against each *PGM and *SRVPGM object.

For example, refer to [Figure 37 on page 86](#).

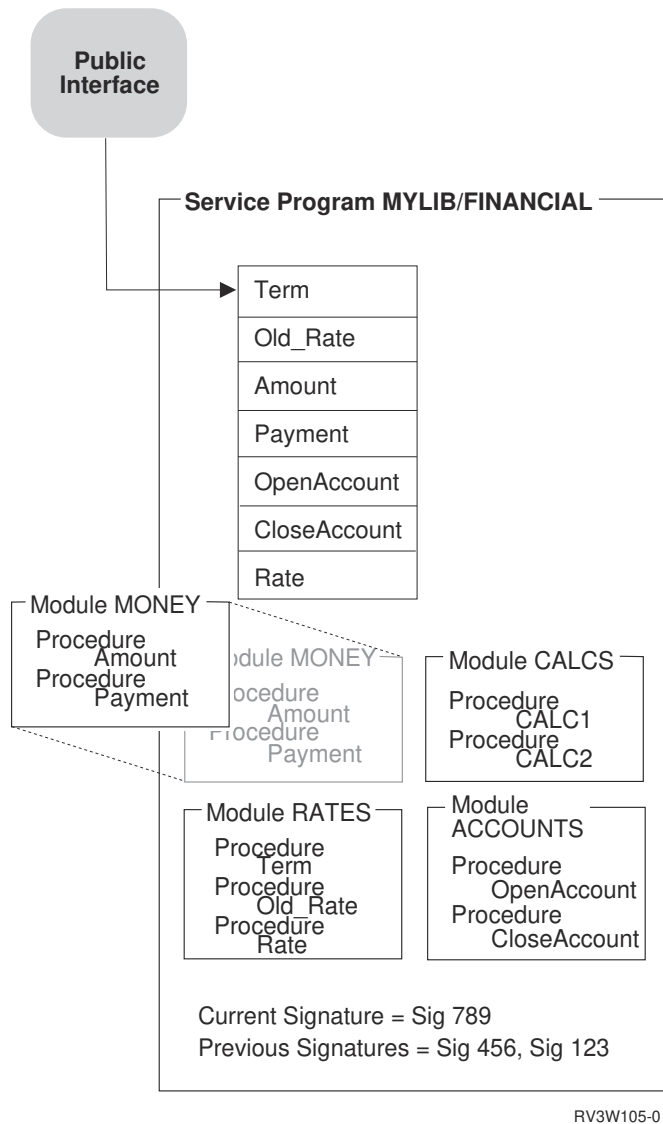


Figure 37. Replacing a Module in a Service Program

If a program or service program is updated while it remains activated in another job, the job will continue to use the old version of the program or service program. New activations will use the updated version of the program or service program.

The allow update (ALWUPD) and allow *SRVPGM library update (ALWLIBUPD) parameters on the CRTPGM or CRTSRVPGM command determine whether a program object or service program can be updated. By specifying ALWUPD(*NO), the modules in a program object or service program cannot be replaced by the UPDPGM or UPDSRVPGM command. By specifying ALWUPD(*YES) and ALWLIBUPD(*YES), you can update your program to use a service program from a library that was not previously specified. By specifying ALWUPD(*YES) and ALWLIBUPD(*NO), you can update the modules, but not the bound service program library. You cannot specify ALWUPD(*NO) and ALWLIBUPD(*YES) at the same time.

Parameters on the UPDPGM and UPDSRVPGM Commands

Each module specified on the module parameter replaces a module with the same name that is bound into a program object or service program. If more than one module bound into a program object or service program has the same name, the replacement library (RPLLIB) parameter is used. This parameter specifies which method is used to select the module to be replaced. If no module with the same name is already bound into a program object or service program, the program object or service program is not updated.

The bound service program (BNDSRVPGM) parameter specifies additional service programs beyond those that the program object or service program is already bound to. If a replacing module contains more imports or fewer exports than the module it replaces, these service programs may be needed to resolve those imports.

With the service program library (SRVPGMLIB) parameter, you can specify the library that stores the bound service programs. Each time you run the UPDPGM or UPDSRVPGM commands, the bound service programs from the specified library are used. The UPDPGM or UPDSRVPGM command allows you to change library if ALWLIBUPD(*YES) is used.

The binding directory (BNDDIR) parameter specifies binding directories that contain modules or service programs that also may be required to resolve extra imports.

The activation group (ACTGRP) parameter specifies the activation group name to be used when a program or service program is activated. This parameter also allows you to change the activation group name of a named activation group.

Module Replaced by a Module with Fewer Imports

If a module is replaced by another module with fewer imports, the new program object or service program is always created. However, the updated program object or service program contains an isolated module if the following conditions exist:

- Because of the now missing imports, one of the modules bound into a program object or service program no longer resolves any imports
- That module originally came from a binding directory used on the CRTPGM or CRTSRVPGM command

Programs with isolated modules may grow significantly over time. To remove modules that no longer resolve any imports and that originally came from a binding directory, you can specify OPTION(*TRIM) when updating the objects. However, if you use this option, the exports that the modules contain are not available for future program updates.

Module Replaced by a Module with More Imports

If a module is replaced by a module with more imports, the program object or service program can be updated if those extra imports are resolved, given the following:

- The existing set of modules bound into the object.
- Service programs bound to the object.
- Binding directories specified on the command. If a module in one of these binding directories contains a required export, the module is added to the program or service program. If a service program in one of these binding directories contains a required export, the service program is bound by reference to the program or service program.
- Implicit binding directories. An **implicit binding directory** is a binding directory that contains exports that may be needed to create a program that contains the module. Every ILE compiler builds a list of implicit binding directories into each module it creates.

If those extra imports cannot be resolved, the update operation fails unless OPTION(*UNRSLVREF) is specified on the update command.

Module Replaced by a Module with Fewer Exports

If a module is replaced by another module with fewer exports, the update occurs if the following conditions exist:

- The missing exports are not needed for binding.
- The missing exports are not exported out of the service program in the case of UPDSRVPGM.

If a service program is updated with EXPORT(*ALL) specified, a new export list is created. The new export list will be different from the original export list.

The update does not occur if the following conditions exist:

- Some imports cannot be resolved because of the missing exports.
- Those missing exports cannot be found from the extra service programs and binding directories specified on the command.
- The binder language indicates to export a symbol, but the export is missing.

Module Replaced by a Module with More Exports

If a module is replaced by another module with more exports, the update operation occurs if all the extra exports are uniquely named. The service program export is different if EXPORT(*ALL) is specified.

However, if one or more of the extra exports are not uniquely named, the duplicate names may cause a problem:

- If OPTION(*NODUPPROC) or OPTION(*NODUPVAR) is specified on the update command, the program object or service program is not updated.
- If OPTION(*DUPPROC) or OPTION(*DUPVAR) is specified, the update occurs, but the extra export may be used rather than the original export of the same name.

Tips for Creating Modules, Programs, and Service Programs

To create and maintain modules, ILE programs, and service programs conveniently, consider the following:

- Follow a naming convention for the modules that will get copied to create a program or service program.

A naming strategy with a common prefix makes it easier to specify modules generically on the module parameter.

- For ease of maintenance, include each module in only one program or service program. If more than one program needs to use a module, put the module in a service program. That way, if you have to redesign a module, you only have to redesign it in one place.
- To ensure your signature, use the binder language whenever you create a service program.

The binder language allows the service program to be easily updated without having to re-create the using programs and service programs.

The Retrieve Binder Source (RTVBNDSRC) command can be used to help generate the binder language source based on exports from one or more modules or service programs.

If either of the following conditions exists:

- A service program will never change
- Users of the service program do not mind changing their programs when a signature changes

you do not need to use the binder language. Because this situation is not likely for most applications, consider using the binder language for all service programs.

- If you get a CPF5D04 message when using a program creation command such as CRTPGM, CRTSRVPGM, or UPDPM, but your program or service program is still created, there are two possible explanations:
 1. Your program is created with OPTION(*UNRSLVREF) and contains unresolved references.
 2. You are binding to a *SRVPGM listed in *BNDDIR QSYS/QUSAPIBD that is shipped with *PUBLIC *EXCLUDE authority, and you do not have authority. To see who is authorized to an object, use the DSPOBJAUT command. System *BNDDIR QUSAPIBD contains the names of *SRVPGMs that provide system APIs. Some of these APIs are security-sensitive, so the *SRVPGMs they are in are shipped with *PUBLIC *EXCLUDE authority. These *SRVPGMs are grouped at the end of QUSAPIBD. When you are using a *PUBLIC *EXCLUDE service program in this list, the binder usually has to examine other *PUBLIC *EXCLUDE *SRVPGMs ahead of yours, and it takes the CPF5D04.

To avoid getting the CPF5D04 message, use one of the following methods:

- Explicitly specify any *SRVPGMs your program or service program is bound to. To see the list of *SRVPGMs your program or service is bound to, use DSPPGM or DSPSRVPGM DETAIL(*SRVPGM). These *SRVPGMs can be specified on the CRTPGM or CRTSRVPGM BNDSRVPGM parameter. They can also be placed into a binding directory given on the CRTBNDRPG, CRTRPGMOD, CRTBNDCBL, CRTPGM, or CRTSRVPGM BNDDIR parameter, or from an RPG H-spec. Taking this action ensures that all references are resolved before the *PUBLIC *EXCLUDE *SRVPGMs in *BNDDIR QUSAPIBD need to be examined.
- Grant *PUBLIC or individual authority to the *SRVPGMs listed in the CPF5D04 messages. This has the drawback of authorizing users to potentially security-sensitive interfaces unnecessarily.
- If OPTION(*UNRSLVREF) is used and your program contains unresolved references, make sure all references are resolved.
- If other people will use a program object or service program that you create, specify OPTION(*RSLVREF) when you create it. When you are developing an application, you may want to create a program object or service program with unresolved imports. However, when in production, all the imports should be resolved.

If OPTION(*WARN) is specified, unresolved references are listed in the job log that contains the CRTPGM or CRTSRVPGM request. If you specify a listing on the DETAIL parameter, they are also included on the program listing. You should keep the job log or listing.

- When designing new applications, determine if common procedures that should go into one or more service programs can be identified.

It is probably easiest to identify and design common procedures for new applications. If you are converting an existing application to use ILE, it may be more difficult to determine common procedures for a service program. Nevertheless, try to identify common procedures needed by the application and try to create service programs containing the common procedures.

- When converting an existing application to ILE, consider creating a few large programs.

With a few, usually minor changes, you can easily convert an existing application to take advantage of the ILE capabilities. After you create the modules, combining them into a few large programs may be the easiest and least expensive way to convert to ILE.

- Try to limit the number of service programs your application uses.

This may require a service program to be created from more than one module. The advantages are a faster activation time and a faster binding process.

There are no simple answers for the number of service programs an application should use. If a program uses hundreds of service programs, it is probably using too many. On the other hand, one service program may not be practical either.

Activation Group Management

This topic contains examples of how to structure an application using activation groups. Topics include:

- Supporting multiple applications
- Using the Reclaim Resources (RCLRSC) command with OPM and ILE programs
- Deleting activation groups with the Reclaim Activation Group (RCLACTGRP) command
- Service programs and activation groups

Multiple Applications Running in the Same Job

User-named activation groups allow you to leave an activation group in a job for later use. A normal return operation or a skip operation (such as `longjmp()` in ILE C) past the control boundary does not delete your activation group.

This allows you to leave your application in its last-used state. Static variables and open files remain unchanged between calls into your application. This can save processing time and may be necessary to accomplish the function you are trying to provide.

You should be prepared, however, to accept requests from multiple independent clients running in the same job. The system does not limit the number of ILE programs that can be bound to your ILE service program. As a result, you may need to support multiple clients.

Figure 38 on page 91 shows a technique that you may use to share common service functions while keeping the performance advantages of a user-named activation group.

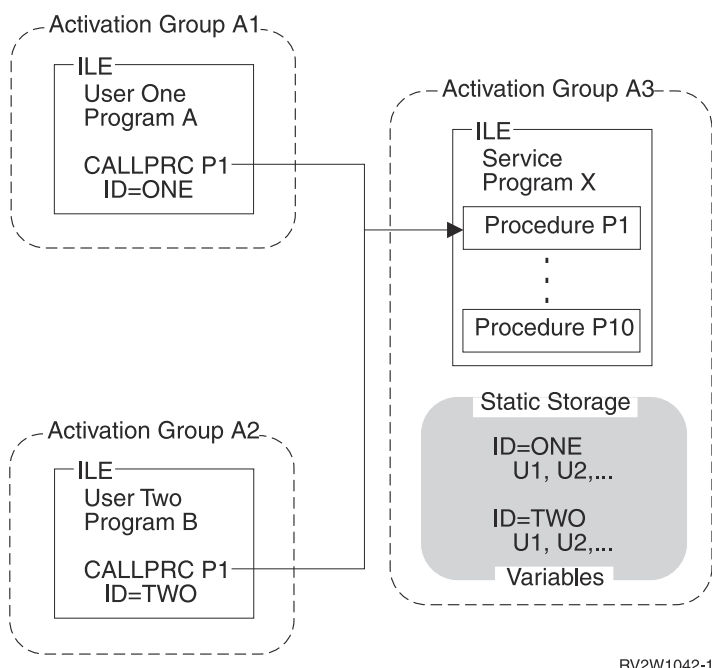


Figure 38. Multiple Applications Running in the Same Job

Each call to a procedure in service program X requires a user handle. The field ID represents a user handle in this example. Each user is responsible for providing this handle. You do an initialization routine to return a unique handle for each user.

When a call is made to your service program, the user handle is used to locate the storage variables that relate to this user. While saving activation group creation time, you can support multiple clients at the same time.

Reclaim Resources Command

The Reclaim Resources (RCLRSC) command depends on a system concept known as a **level number**. A level number is a unique value assigned by the system to certain resources you use within a job. Three level numbers are defined as follows:

Call level number

Each call stack entry is given a unique level number

Program-activation level number

Each OPM and ILE program activation is given a unique level number

Activation-group level number

Each activation group is given a unique level number

As your job runs, the system continues to assign unique level numbers for each new occurrence of the resources just described. The level numbers are assigned in increasing value. Resources with higher level numbers are created after resources with lower level numbers.

[Figure 39 on page 93](#) shows an example of using the RCLRSC command on OPM and ILE programs. Call-level scoping has been used for the open files shown in this example. When call-level scoping is used, each data management resource is given the same level numbers as the call stack entry that created that resource.

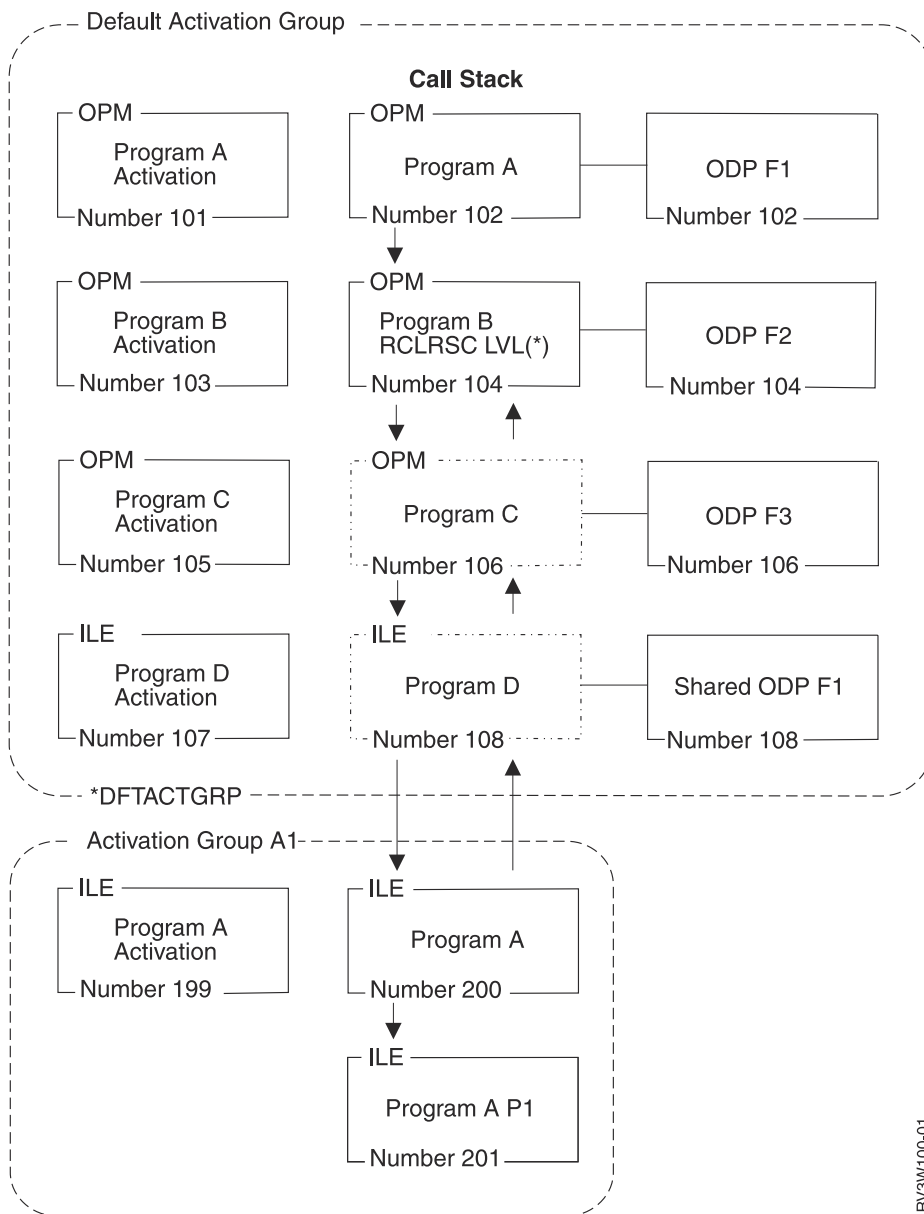


Figure 39. Reclaim Resources

In this example, the calling sequence is programs A, B, C, and D. Programs D and C return to program B. Program B is about to use the RCLRSC command with an option of LVL(*). The RCLRSC command uses the level (LVL) parameter to clean up resources. All resources with a call-level number greater than the call-level number of the current call stack entry are cleaned up. In this example, call-level number 104 is used as the starting point. All resources greater than call-level number 104 are deleted. Note that resources in call level 200 and 201 are unaffected by RCLRSC because they are in an ILE activation group. RCLRSC works only in the default activation groups.

In addition, the storage from programs C and D and the open data path (ODP) for file F3 is closed. File F1 is shared with the ODP opened in program A. The shared ODP is closed, but file F1 remains open.

Reclaim Resources Command for OPM Programs

The Reclaim Resources (RCLRSC) command may be used to close open files and free static storage for OPM programs that have returned without ending. Some OPM languages, such as RPG, allow you to return without ending the program. If you later want to close the program's files and free its storage, you may use the RCLRSC command.

Reclaim Resources Command for ILE Programs

For ILE programs that are created by the CRTBNDRPG and CRTBNDCL commands with DFTACTGRP(*YES) specified, the RCLRSC command frees static storage just as it does for OPM programs. For ILE programs that are **not** created by the CRTBNDRPG or CRTBNDCL commands with DFTACTGRP(*YES) specified, the RCLRSC command reinitializes any activations that have been created in a default activation group but does not free static storage. ILE programs that use large amounts of static storage should be activated in an ILE activation group. Deleting the activation group returns this storage to the system. The RCLRSC command closes files opened by service programs or ILE programs running in a default activation group. The RCLRSC command does not reinitialize static storage of service programs and does not affect non-default activation groups.

To use the RCLRSC command directly from ILE, you can use either the QCAPCMD API or an ILE CL procedure. Using the QCAPCMD API, you can directly call system commands without the use of a CL program. In [Figure 39 on page 93](#), directly calling system commands is important because you might want to use the call-level number of a particular ILE procedure. Certain languages, such as ILE C, also provide a system function that allows direct running of IBM i commands.

Reclaim Activation Group Command

The Reclaim Activation Group (RCLACTGRP) command can be used to delete a non-default activation group that is not in use. This command allows options to either delete all eligible activation groups or to delete an activation group by name.

Service Programs and Activation Groups

When you create an ILE service program, decide whether to specify an option of *CALLER or a name for the ACTGRP parameter. This option determines whether your service program will be activated into the caller's activation group or into a separately named activation group. Either choice has advantages and disadvantages. This topic discusses what each option provides.

For the ACTGRP(*CALLER) option, the service program functions as follows:

- Static procedure calls are fast

Static procedure calls into the service program are optimized when running in the same activation group.

- Shared external data

Service programs may export data to be used by other programs and service programs in the same activation group.

- Shared data management resources

Open files and other data management resources may be shared between the service program and other programs in the activation group. The service program may issue a commit operation or a rollback operation that affects the other programs in the activation group.

- No control boundary

Unhandled exceptions within the service program percolate to the client programs. HLL end verbs used within the service program can delete the activation group of the client programs.

For the ACTGRP(name) option, the service program functions as follows:

- Separate address space for variables, if using single-level storage model.

The client program cannot manipulate pointers to address your working storage. This may be important if your service program is running with adopted authority.

- Separate data management resources

You have your own open files and commitment definitions. The accidental sharing of open files is prevented.

- State information controlled

You control when the application storage is deleted. By using HLL end verbs or normal language return statements, you can decide when to delete the application. You must, however, manage the state information for multiple clients.

Calls to Procedures and Programs

The ILE call stack and argument-passing methods facilitate interlanguage communication, making it easier for you to write mixed-language applications. This topic discusses different examples of dynamic program calls and static procedure calls, which were introduced in [“Calling a Program or a Procedure” on page 24](#). A third type of call, the procedure pointer call, is introduced.

In addition, this topic discusses how to support OPM application programming interfaces (APIs) using new ILE functions or OPM-to-ILE conversions.

Call Stack

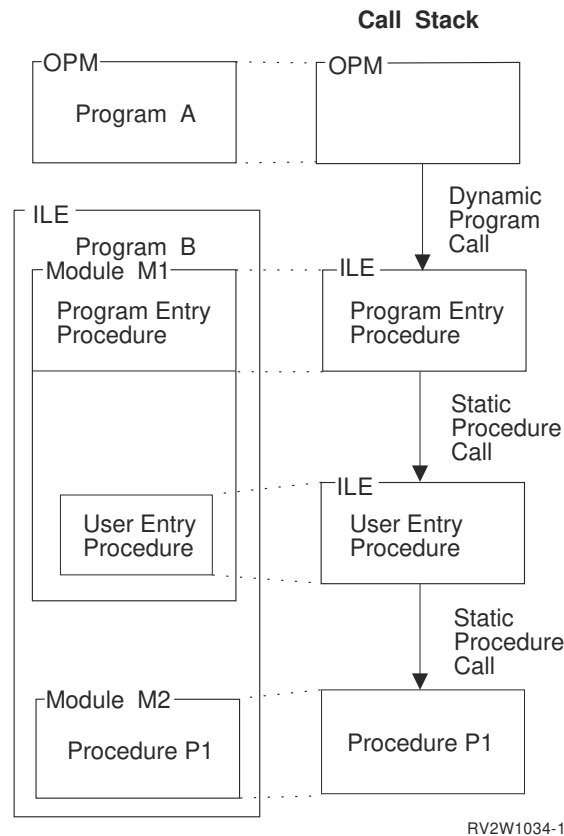
The **call stack** is a last-in-first-out (LIFO) list of **call stack entries**, one entry for each called procedure or program. Each call stack entry has information about the automatic variables for the procedure or program and about other resources scoped to the call stack entry, such as condition handlers and cancel handlers.

There is one call stack per thread. A call adds a new entry on the call stack for the called procedure or program and passes control to the called object. A return removes the stack entry and passes control back to the calling procedure or program. See [Threads on IBM i](#) for more information.

Call Stack Example

[Figure 40 on page 98](#) contains a segment of a call stack with two programs: an OPM program (Program A) and an ILE program (Program B). Program B contains three procedures: its program entry procedure, its user entry procedure, and another procedure (P1). The concepts of program entry procedure (PEP) and user entry procedure (UEP) are defined in [“Module Object” on page 16](#). The call flow includes the following steps:

1. A dynamic program call to Program A.
2. Program A calls Program B, passing control to its PEP. This call to Program B is a dynamic program call.
3. The PEP calls the UEP. This is a static procedure call.
4. The UEP calls procedure P1. This is a static procedure call.



RV2W1034-1

Figure 40. Dynamic Program Calls and Static Procedure Calls on the Call Stack

Figure 40 on page 98 illustrates the call stack for this example. The most recently called entry on the stack is depicted at the bottom of the stack. It is the entry for the program or procedure that is currently processing. The current program or procedure may do either of the following:

- Call another procedure or program, which adds another entry to the bottom of the stack.
- Return control to its caller after it is done processing, which causes its call stack entry to be removed from the stack.

Assume that, after procedure P1 is done, no more processing is needed from Program B. Procedure P1 returns control to the UEP, and the entry for P1 is removed from the stack. Then the UEP returns control to the PEP, and the UEP entry is removed from the stack. Finally, the PEP returns control to Program A, and the PEP entry is removed from the stack. Only Program A's entry is left on this segment of the call stack. Program A continues processing from the point where it made the dynamic program call to Program B.

Calls to Programs and Calls to Procedures

Three types of calls can be made during ILE runtime: dynamic program calls, static procedure calls, and procedure pointer calls.

When an ILE program is activated, all of its procedures except its PEP become available for static procedure calls and procedure pointer calls. Program activation occurs when the program is called by a dynamic program call and the activation does not already exist. When a program is activated, the service programs that are bound to this program, for which deferred activation does not apply, are also activated. The procedures in an ILE service program can be accessed only by static procedure calls or by procedure pointer calls (not by dynamic program calls).

Static Procedure Calls

A call to an ILE procedure adds a new call stack entry to the bottom of the stack and passes control to a specified procedure. Examples include any of the following:

1. A call to a procedure in the same module
2. A call to a procedure in a different module in the same ILE program or service program
3. A call to a procedure that has been exported from an ILE service program in the same activation group
4. A call to a procedure that has been exported from an ILE service program in a different activation group

In examples [“1” on page 99](#), [“2” on page 99](#), and [“3” on page 99](#), the static procedure call does not cross an activation group boundary. This call path is much shorter than the path for a dynamic program call to an ILE or OPM program. In example [“4” on page 99](#), the call crosses an activation group boundary, and additional processing is done to switch activation group resources. The call path length is longer than the path length of a static procedure call within an activation group, but still shorter than for a dynamic program call.

For a static procedure call, the called procedure has been bound to the calling procedure. The call always accesses the same procedure. In contrast, the target of a procedure pointer call can vary with each call.

Procedure Pointer Calls

Procedure pointer calls provide a way to call a procedure dynamically. For example, by manipulating arrays, or tables, of procedure names or addresses, you can dynamically route a procedure call to different procedures.

Procedure pointer calls add entries to the call stack in exactly the same manner as static procedure calls. Any procedure that can be called using a static procedure call can also be called through a procedure pointer. If the called procedure is in the same activation group, the cost of a procedure pointer call is almost identical to the cost of a static procedure call.

Passing Arguments to ILE Procedures

In an ILE procedure call, an **argument** is an expression that represents a value that the calling procedure passes to the procedure specified in the call. ILE languages use three methods for passing arguments:

by value, directly

The value of the data object is placed directly into the argument list.

by value, indirectly

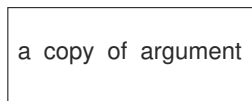
The value of the data object is copied to a temporary location. The address of the copy (a pointer) is placed into the argument list.

by reference

A pointer to the data object is placed into the argument list. Changes made by the called procedure to the argument are reflected in the calling procedure.

[Figure 41 on page 100](#) illustrates these argument passing styles. Not all ILE languages support passing by value, directly. The available passing styles are described in the ILE HLL programmer's guides.

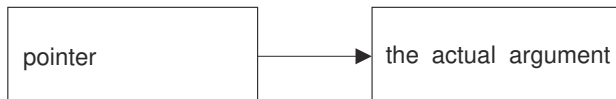
By value, directly



By value, indirectly



By reference



RV2W1027-1

Figure 41. Methods for Passing Arguments to ILE Procedures

HLL semantics usually determine when data is passed by value and when it is passed by reference. For example, ILE C passes and accepts arguments by value, directly, while for ILE COBOL and ILE RPG, arguments are usually passed by reference. You must ensure that the calling program or procedure passes arguments in the manner expected by the called procedure. The ILE HLL programmer's guides contain more information on passing arguments to different languages.

Prior to IBM i 7.3 a maximum of 400 arguments were allowed on a static procedure call. In IBM i 7.3 and later a maximum of 16,383 arguments are allowed on a static procedure call. Each ILE language may further restrict the maximum number of arguments. The ILE languages support the following argument-passing styles:

- ILE C passes and accepts arguments by value directly, widening integers and floating-point values by default. Arguments can be passed unwidened or by value indirectly if you specify the appropriate values on the `#pragma` argument directive for the called function.
- ILE C++ passes and accepts arguments by value directly. C++ does not widen parameters and floating-point values by default. Arguments can be widened or passed by value indirectly if you specify the appropriate values on the extern linkage specifier for the declaration of the called function.
- ILE COBOL passes and accepts arguments by value, by reference, or by value indirectly. Parameters that are passed by value are not widened.
- ILE RPG passes and accepts arguments by value, or by reference. RPG does not widen integers and floating point values by default, but this is available for parameters passed by value, by coding `EXTPROC(*CWIDEN)`.
- ILE CL passes and accepts arguments by reference and by value. Parameters that are passed by value are not widened.

Function Results

To support HLLs that allow the definition of functions (procedures that return a result argument), the model assumes that a special function result argument may be present, as shown in [Figure 42 on page 101](#). As described in the ILE HLL programmer's guides, ILE languages that support function results use a common mechanism for returning function results.

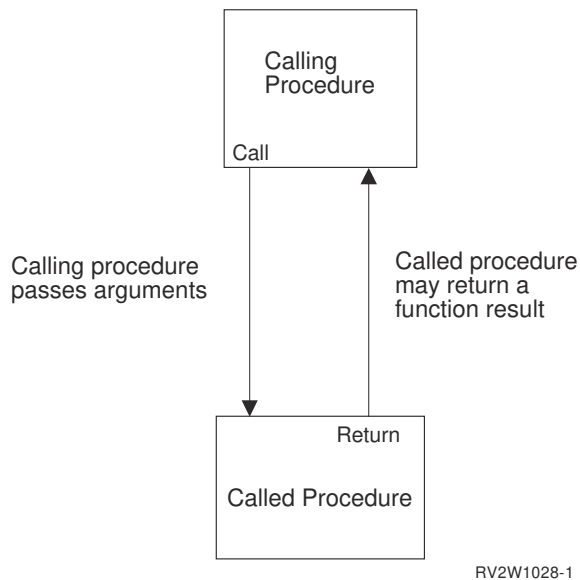


Figure 42. Program Call Argument Terminology

Omitted Arguments

All ILE languages can simulate omitted arguments, which allows the use of the feedback code mechanism for ILE condition handlers and other runtime procedures. For example, if an ILE C procedure or an ILE bindable API is expecting an argument passed by reference, you can sometimes omit the argument by passing a null pointer in its place. For information about how to specify an omitted argument in a specific ILE language, refer to the programmer's guide for that language. The API topic of the Programming category of the IBM i Information Center specifies which arguments can be omitted for each API.

For ILE languages that do not provide an intrinsic way for a called procedure to test if an argument has been omitted, the Test for Omitted Argument (CEETSTA) bindable API is available.

Dynamic Program Calls

A dynamic program call is a call made to a program object. For example, when you use the CL command CALL, you are making a dynamic program call.

OPM programs are called by using dynamic program calls. OPM programs are additionally limited to making only dynamic program calls.

ILE programs are also called by dynamic program calls. The procedures within an activated ILE program can be accessed by using static procedure calls or procedure pointer calls.

In contrast to static procedure calls, which are bound at compile time, symbols for dynamic program calls are resolved to addresses when the call is performed. As a result, a dynamic program call uses more system resources than a static procedure call. Examples of a dynamic program call include:

- A call to an ILE program or an OPM program
- A call to a non-bindable API

A dynamic program call to an ILE program passes control to the PEP of the identified program, which then passes control to the UEP of the program. After the called program is done processing, control is passed back to the instruction following the call program instruction.

Passing Arguments on a Dynamic Program Call

Calls to ILE or OPM programs (in contrast to calls to ILE procedures) pass arguments by reference, meaning that the called program receives the address of each argument.

When using a dynamic program call, you need to know the method of argument passing that is expected by the called program and how to simulate it if necessary. A maximum of 255 arguments are allowed on a dynamic program call. Each ILE language may further restrict the maximum number of arguments. Some ILE languages support the built-in function CALLPGMV, which allows a maximum of 16383 arguments. Information on how to use the different passing methods is contained in the ILE HLL programmer's guides.

Interlanguage Data Compatibility

ILE calls allow arguments to be passed between procedures that are written in different HLLs. To facilitate data sharing between the HLLs, some ILE languages have added data types. For example, ILE COBOL added USAGE PROCEDURE-POINTER as a new data type.

To pass arguments between HLLs, you need to know the format each HLL expects of arguments it is receiving. The calling procedure is required to make sure the arguments are the size and type expected by the called procedure. For example, an ILE C function may expect a 4-byte integer, even if a short integer (2 bytes) is declared in the parameter list. Information on how to match data type requirements for passing arguments is contained in the ILE HLL programmer's guides.

Syntax for Passing Arguments in Mixed-Language Applications

Some ILE languages provide syntax for passing arguments to procedures in other ILE languages. For example, ILE C provides a `#pragma` argument to pass value arguments to other ILE procedures by value indirectly; RPG has special values for the EXTPROC prototype keyword.

Operational Descriptors

Operational descriptors may be useful to you if you are writing a procedure or API that can receive arguments from procedures written in different HLLs. **Operational descriptors** provide descriptive information to the called procedure in cases where the called procedure cannot precisely anticipate the form of the argument (for example, different types of strings). The additional information allows the procedure to properly interpret the arguments.

The argument supplies the value; the operational descriptor supplies information about the argument's size and type. For example, this information may include the length of a character string and the type of string.

With operational descriptors, services such as bindable APIs are not required to have a variety of different bindings for each HLL, and HLLs do not have to imitate incompatible data types. A few ILE bindable APIs use operational descriptors to accommodate the lack of common string data types between HLLs. The presence of the operational descriptor is transparent to the API user.

Operational descriptors support HLL semantics while being invisible to procedures that do not use or expect them. Each ILE language can use data types that are appropriate to the language. Each ILE language compiler provides at least one method for generating operational descriptors. For more information on HLL semantics for operational descriptors, refer to the ILE HLL reference manual.

Operational descriptors are distinct from other data descriptors with which you may be familiar. For instance, they are unrelated to the descriptors associated with distributed data or files.

Requirements of Operational Descriptors

You need to use operational descriptors when they are expected by a called procedure written in a different ILE language and when they are expected by an ILE bindable API. Generally, bindable APIs require descriptors for most string arguments. Information about bindable APIs in the API topic of the Programming category of the IBM i Information Center specifies whether a given bindable API requires operational descriptors.

Absence of a Required Descriptor

The omission of a required descriptor is an error. If a procedure requires a descriptor for a specific parameter, this requirement forms part of the interface for that procedure. If a required descriptor is not provided, the called procedure will fail at runtime.

Presence of an Unnecessary Descriptor

The presence of a descriptor that is not required does not interfere with the called procedure's access to arguments. If an operational descriptor is not needed or expected, the called procedure simply ignores it.

Note: However, generating unneeded descriptors can diminish performance, because descriptor use effectively increases the length of the call path.

Bindable APIs for Operational Descriptor Access

Descriptors are normally accessed directly by a called procedure according to the semantics of the HLL in which the procedure is written. Once a procedure is programmed to expect operational descriptors, no further handling is usually required by the programmer. However, sometimes a called procedure needs to determine whether the descriptors that it requires are present before accessing them. For this purpose the following bindable APIs are provided:

- Retrieve Operational Descriptor Information (CEEDOD) bindable API
- Get String Information (CEEGSI) bindable API

Support for OPM and ILE APIs

When you develop new functions in ILE or convert an existing application to ILE, you may want to continue to support call-level APIs from OPM. This topic explains one technique that may be used to accomplish this dual support while maintaining your application in ILE.

ILE service programs provide a way for you to develop and deliver bindable APIs that may be accessed from all ILE languages. To provide the same functions to OPM programs, you need to consider the fact that a procedure in an ILE service program cannot be called directly from an OPM program.

The technique to use is to develop ILE program stubs for each bindable API that you plan to support. You may want to name the bindable APIs the same as the ILE program stubs, or you may choose different names. Each ILE program stub contains a static procedure call to the actual bindable API.

An example of this technique is shown in [Figure 43 on page 104](#).

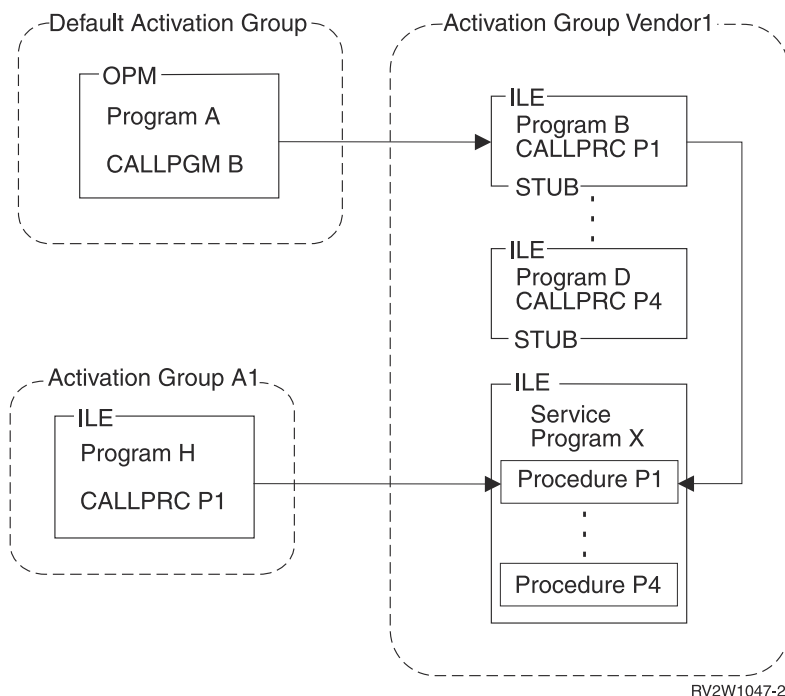


Figure 43. Supporting OPM and ILE APIs

Programs B through D are the ILE program stubs. Service program X contains the actual implementation of each bindable API. Each program stub and the service program are given the same activation group name. In this example, the activation group name VENDOR1 is chosen.

Activation group VENDOR1 is created by the system when necessary. The dynamic program call from OPM program A creates the activation group on the first call from an OPM program. The static procedure call from ILE program H creates the activation group when ILE program H is activated. Once the activation group exists, it may be used from either program A or program H.

Write the implementation of your API in an ILE procedure (procedure P1 in this example). This procedure can be called either directly through a procedure call or indirectly through a dynamic program call to the program stub. You must be careful when you take actions that are dependent on the call stack structure, such as sending exception messages. A normal return from either the program stub or the implementing procedure leaves the activation group in the job for later use. You can implement your API procedure with the knowledge that a control boundary is established for either the program stub or the implementing procedure on each call. HLL ending verbs delete the activation group whether the call originated from an OPM program or an ILE program.

Storage Management

The operating system provides storage support for the ILE high-level languages. This storage support removes the need for unique storage managers for the runtime environment of each language. It avoids incompatibilities between different storage managers and mechanisms in high-level languages.

The operating system provides the automatic, static, and dynamic storage used by programs and procedures at runtime. Automatic and static storage are managed by the operating system. That is, the need for automatic and static storage is known at compilation time from program variable declarations. Dynamic storage is managed by the program or procedure. The need for dynamic storage is known only at runtime.

When program activation occurs, static storage for program variables is allocated and initialized.

When a program or procedure begins to run, automatic storage is allocated. The automatic storage stack is extended for variables as the program or procedure is added to the call stack.

As a program or procedure runs, dynamic storage is allocated under program control. This storage is extended as additional storage is required. You have the ability to control dynamic storage. The remainder of this topic concentrates on dynamic storage and the ways in which it can be controlled.

Single-Level Storage Heap

A **heap** is an area of storage that is used for allocations of dynamic storage. The amount of dynamic storage that is required by an application depends on the data being processed by the program and procedures that use a heap. The operating system allows the use of multiple single-level storage heaps that are dynamically created and discarded. The ALCHSS instruction always uses single-level storage. Some languages also support the use of teraspace for dynamic storage.

Heap Characteristics

Each heap has the following characteristics:

- The system assigns a unique heap identifier to each heap within the activation group.

The heap identifier for the default heap is always zero.

A storage management-bindable API, called by a program or procedure, uses the heap identifier to identify the heap on which it is to act. The bindable API must run within the activation group that owns the heap.

- The activation group that creates a heap also owns it.

Because activation groups own heaps, the lifetime of a heap is no longer than that of the owning activation group. The heap identifier is meaningful and unique only within the activation group that owns it.

- The size of a heap is dynamically extended to satisfy allocation requests.

The maximum size of the heap is 4 gigabytes minus 512K bytes. This is the maximum heap size if the total number of allocations (at any one time) does not exceed 128 000.

- The maximum size of any single allocation from a heap is limited to 16 megabytes minus 64K bytes.

Default Heap

The first request for dynamic storage from the default heap within an activation group that is using single-level storage results in the creation of a default heap from which the storage allocation takes place. If there is insufficient storage in the heap to satisfy any subsequent requests for dynamic storage, the system extends the heap and allocates additional storage.

Allocated dynamic storage remains allocated until explicitly freed or until the system discards the heap. The default heap is discarded only when the owning activation group ends.

Programs in the same activation group automatically share dynamic storage provided the default heap allocated the storage. However, you can isolate the dynamic storage that is used by some programs and procedures within an activation group. You do this by creating one or more heaps.

User-Created Heaps

You can explicitly create and discard one or more heaps by using ILE bindable APIs. This gives you the capability of managing the heaps and the dynamic storage that is allocated from those heaps.

For example, the system may or may not share dynamic storage that is allocated in user-created heaps for programs within an activation group. The sharing of dynamic storage depends on the heap identifier that is referred to by the programs. You can use more than one heap to avoid automatic sharing of dynamic storage. In this way you can isolate logical groups of data. Following are some additional reasons for using one or more user-created heaps:

- You can group certain storage objects together to meet a one-time requirement. Once you meet that requirement, you can free the dynamic storage that was allocated by a single call to the Discard Heap (CEEDSHP) bindable API. This operation frees the dynamic storage and discards the heap. In this way, dynamic storage is available to meet other requests.
- You can free multiple dynamic storage that is allocated at once by using the Mark Heap (CEEMKHP) and Release Heap (CEERLHP) bindable APIs. The CEEMKHP bindable API allows you to mark a heap. When you are ready to free the group of allocations that were made since the heap was marked, use the CEERLHP bindable API. Using the mark and release functions leaves the heap intact, but frees the dynamic storage that is allocated from it. In this way, you can avoid the system overhead that is associated with heap creation by re-using existing heaps to meet dynamic storage requirements.
- Your storage requirements may not match the storage attributes that define the default heap. For example, the initial size of the default heap is 4K bytes. However, you require a number of dynamic storage allocations that together exceed 4K bytes. You can create a heap with a larger initial size than 4K bytes. This reduces the system overhead which would otherwise occur both when implicitly extending the heap and subsequently accessing the heap extensions. Similarly, you can have heap extensions larger than 4K bytes. For information about defining heap sizes, see [“Heap Allocation Strategy” on page 106](#) and the discussion of heap attributes.

You might have other reasons for using multiple heaps rather than the default heap. The storage management-bindable APIs give you the capability to manage the heaps that you create and the dynamic storage that is allocated in those heaps. For information about the storage management APIs, see the API topic collection in the Programming category of the IBM i Information Center.

Single-Heap Support

Languages that do not have intrinsic multiple-heap storage support use the default single-level storage heap. You cannot use the Discard Heap (CEEDSHP), the Mark Heap (CEEMKHP), or the Release Heap (CEERLHP) bindable APIs with the default heap. You can free dynamic storage that is allocated by the default heap by using explicit free operations, or by ending the activation group that owns it.

These restrictions on the use of the default heap help prevent inadvertent release of allocated dynamic storage in mixed-language applications. Remember to consider release heap and discard heap operations as insecure for large applications that re-use existing code with potentially different storage support. Remember not to use release heap operations that are valid for the default heap. This causes multiple parts of an application that uses the mark function correctly when used separately to possibly fail when used together.

Heap Allocation Strategy

The attributes associated with the default heap are defined by the system through a default allocation strategy. This allocation strategy defines attributes such as a heap creation size of 4K bytes and an extension size of 4K bytes. You cannot change this default allocation strategy.

However, you can control heaps that you explicitly create through the Create a Heap (CEE4CRHP) bindable API. You also can define an allocation strategy for explicitly created heaps through the Define Heap Allocation Strategy (CEE4DAS) bindable API. Then, when you explicitly create a heap, the heap attributes are provided by the allocation strategy that you defined. In this way you can define separate allocation strategies for one or more explicitly created heaps.

You can use the CEE4CRHP bindable API without defining an allocation strategy. In this case, the heap is defined by the attributes of the _CEE4ALC allocation strategy type. The _CEE4ALC allocation strategy type specifies a heap creation size of 4K bytes and an extension size of 4K bytes. The _CEE4ALC allocation strategy type contains the following attributes:

```
Max_Sngl_Alloc = 16MB - 64K /* maximum size of a single allocation */
Min_Bdy       = 16         /* minimum boundary alignment of any allocation */
Crt_Size      = 4K         /* initial creation size of the heap */
Ext_Size      = 4K         /* the extension size of the heap */
Alloc_Strat   = 0          /* a choice for allocation strategy */
No_Mark       = 1          /* a group deallocation choice */
Blk_Xfer      = 0          /* a choice for block transfer of a heap */
PAG           = 0          /* a choice for heap creation in a PAG */
Alloc_Init    = 0          /* a choice for allocation initialization */
Init_Value    = 0x00       /* initialization value */
```

The attributes that are shown here illustrate the structure of the _CEE4ALC allocation strategy type. For information about all of the _CEE4ALC allocation strategy attributes, see the API topic collection in the Programming category of the IBM i Information Center.

Single-Level Storage Heap Interfaces

Bindable APIs are provided for all heap operations. Applications can be written using either the bindable APIs, language-intrinsic functions, or both.

The bindable APIs fall into the following categories:

- Basic heap operations. These operations can be used on the default heap and on user-created heaps.
 - The Free Storage (CEE4FRST) bindable API frees one previous allocation of heap storage.
 - The Get Heap Storage (CEE4GTST) bindable API allocates storage within a heap.
 - The Reallocate Storage (CEE4CZST) bindable API changes the size of previously allocated storage.
- Extended heap operations. These operations can be used only on user-created heaps.
 - The Create Heap (CEE4CRHP) bindable API creates a new heap.
 - The Discard Heap (CEE4DSHP) bindable API discards an existing heap.
 - The Mark Heap (CEE4MKHP) bindable API returns a token that can be used to identify heap storage to be freed by the CEE4RLHP bindable API.
 - The Release Heap (CEE4RLHP) bindable API frees all storage allocated in the heap since the mark was specified.
- Heap allocation strategies
 - The Define Heap Allocation Strategy (CEE4DAS) bindable API defines an allocation strategy that determines the attributes for a heap created with the CEE4CRHP bindable API.

For information about the storage management APIs, see the API topic collection in the Programming category of the IBM i Information Center.

Heap Support

By default, the dynamic storage provided by malloc, calloc, realloc and new is the same type of storage as the storage model of the root program in the activation group. However, when the single-level storage model is in use, then teraspace storage is provided by these interfaces if the TERASPACE(*YES *TSIFC) compiler option was specified. Similarly, a single-level storage model program can explicitly use bindable APIs to work with teraspace, such as _C_TS_malloc, _C_TS_free, _C_TS_realloc and _C_TS_calloc.

For details about how you can use teraspace storage, see [“Teraspace and Single-Level Storage”](#) on page 49.

If you choose to use both the CEExxxx storage management bindable APIs and the ILE C `malloc()`, `calloc()`, `realloc()`, and `free()` functions, the following rules apply:

- Dynamic storage allocated through the C functions `malloc()`, `calloc()`, and `realloc()`, cannot be freed or reallocated with the CEEFRST and the CEECZST bindable APIs.
- Dynamic storage allocated by the CEEGTST bindable API can be freed with the `free()` function.
- Dynamic storage initially allocated with the CEEGTST bindable API can be reallocated with the `realloc()` function.

Other languages, such as COBOL, have no heap storage model. These languages can access the ILE dynamic storage model through the bindable APIs for dynamic storage.

RPG has operation codes `ALLOC`, `REALLOC` and `DEALLOC`, and builtin functions `%ALLOC` and `%REALLOC` for accessing heap storage. An RPG module can use either single-level heap storage or teraspace heap storage. For modules with teraspace storage model, the default type of heap storage is teraspace. For modules with inherit or single-level storage model, the default type of heap storage is single-level. However, you can explicitly set the type of heap storage using the `ALLOC` keyword on the Control specification. The RPG support uses the CEEGTST, CEECZST, and CEEFRST bindable APIs for single-level heap storage operations and it uses the `_C_TS_malloc()`, `_C_TS_realloc()`, and `_C_TS_free()` functions for teraspace heap storage operations.

Thread Local Storage

The ILE C, ILE C++, and ILE RPG compilers all support thread local storage (TLS). The TLS variables for each program or service program are organized into a TLS frame. The *TLS frame* contains an initialized copy of each TLS variable that is associated with the program or service program. One copy of the TLS frame is created for each thread that runs the program or service program. For information about the support available in a particular compiler, refer to documentation of the specific high-level language (HLL).

A TLS variable is similar to a static variable, except that a unique copy of the TLS variable exists for each thread. See the following table for the differences between TLS variables and static variables.

Table 9. Differences between TLS variables and static variables		
	Static Variable	TLS Variable
When is storage for the variable allocated?	When the program or service program is activated.	When the thread first touches the TLS frame that contains the variable.
When is the variable initialized?	When the program or service program is activated. ²	When the thread first touches the TLS frame that contains the variable.
When is storage for the variable freed?	When the program or service program is deactivated.	When the thread is destroyed.
Does each thread have its own copy of the variable?	No, a single copy is shared by all threads.	Yes.
Is the variable stored in single-level storage or teraspace storage?	Depends on the activation group of the program or service program. ¹	TLS variables are always stored in teraspace storage. ¹
¹ See “Teraspace and Single-Level Storage” on page 49 for more information.		
² This represents the time when the variable is initialized directly by the system. The variable might be initialized indirectly by your HLL at a later time.		

When a reference is made to a TLS variable within a thread, the reference accesses the copy of the variable associated with that thread. It will not access or update a copy of the variable associated with any other thread.

Because each TLS variable is associated with one thread, synchronization (as described in [“Shared Storage Synchronization”](#) on page 157) is usually not a concern. Synchronization might become necessary, however, if the address of a TLS variable is passed to another thread.

Exception and Condition Management

This topic provides additional details on exception handling and condition handling. Before reading this, read the advanced concepts described in [“Error Handling”](#) on page 39.

The exception message architecture of the operating system is used to implement both exception handling and condition handling. There are cases in which exception handling and condition handling interact. For example, an ILE condition handler registered with the Register a User-Written Condition Handler (CEEHDLR) bindable API is used to handle an exception message sent with the Send Program Message (QMHSNDPM) API. These interactions are explained in this topic. The term *exception handler* is used to mean either an operating system exception handler or an ILE condition handler.

Handle Cursors and Resume Cursors

To process exceptions, the system uses two pointers called the handle cursor and resume cursor. These pointers keep track of the progress of exception handling. You need to understand the use of the handle cursor and resume cursor under certain advanced error-handling scenarios. These concepts are used to explain additional error-handling features in later topics.

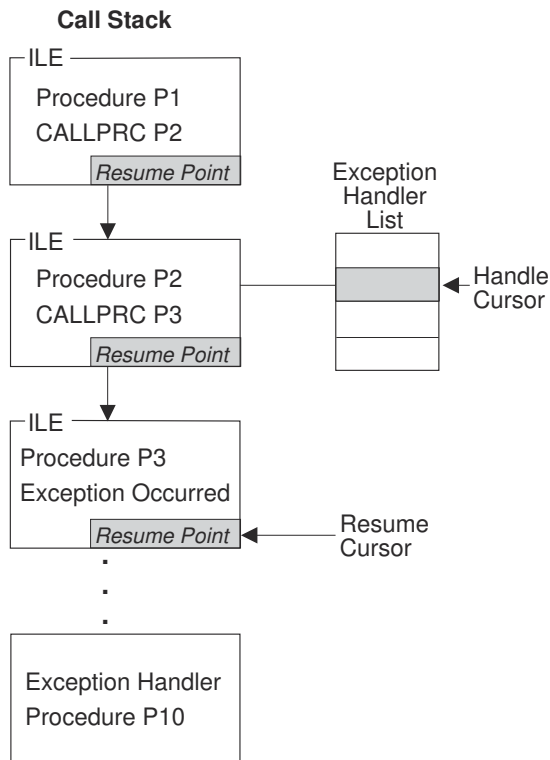
The **handle cursor** is a pointer that keeps track of the current exception handler. As the system searches for an available exception handler, it moves the handle cursor to the next handler in the exception handler list defined by each call stack entry. This list can contain:

- Direct monitor handlers
- ILE condition handlers
- HLL-specific handlers

The handle cursor moves down the exception handler list to lower priority handlers until the exception is handled. If the exception is not handled by any of the exception handlers that have been defined for a call stack entry, the handle cursor moves to the first (highest priority) handler for the previous call stack entry.

The **resume cursor** is a pointer that keeps track of the current location at which your exception handler can resume processing after handling the exception. Normally the system sets the resume cursor to the next instruction following the occurrence of an exception. For call stack entries above the procedure that incurred the exception, the resume point is directly after the procedure or program call that currently suspended the procedure or program. To move the resume cursor to an earlier resume point, use the Move Resume Cursor (CEEMRCR) bindable API.

[Figure 44 on page 112](#) shows an example of the handle cursor and resume cursor.



RV2W1044-0

Figure 44. Handle Cursor and Resume Cursor Example

The handle cursor is currently at the second exception handler defined in the exception handler priority list for procedure P2. The handler procedure P10 is currently called by the system. If procedure P10 handles the exception and returns, control goes to the current resume cursor location defined in procedure P3. This example assumes that procedure P3 percolated the exception to procedure P2.

The exception handler procedure P10 can modify the resume cursor with the Move Resume Cursor (CEEMRCR) bindable API. Two options are provided with this API. An exception handler can modify the resume cursor to either of the following:

- The call stack entry containing the handle cursor
- The call stack entry prior to the handle cursor

In [Figure 44 on page 112](#), you could modify the resume cursor to either procedure P2 or P1. After the resume cursor is modified and the exception is marked as handled, a normal return from your exception handler returns control to the new resume point.

Exception Handler Actions

When your exception handler is called by the system, you can take several actions to handle the exception. For example, ILE C extensions support control actions, branch point handlers, and monitoring by message ID. The possible actions described here pertain to any of the following types of handlers:

- Direct monitor handler
- ILE condition handler
- HLL-specific handler

How to Resume Processing

If you determine that processing can continue, you can resume at the current resume cursor location. Before you can resume processing, the exception message must be changed to indicate that it has been handled. Certain types of handlers require you to explicitly change the exception message to indicate that

the message has been handled. For other handler types, the system can change the exception message before your handler is called.

For a direct monitor handler, you may specify an action to be taken for the exception message. That action may be to call the handler, to handle the exception before calling the handler, or to handle the exception and resume the program. If the action is just to call the handler, you can still handle the exception by using the Change Exception Message (QMHCHGEM) API or the bindable API CEE4HC (Handle Condition). You can change the resume point within a direct monitor handler by using the Move Resume Cursor (CEEMRCR) bindable API. After making these changes, you continue processing by returning from your exception handler.

For an ILE condition handler, you continue the processing by setting a return code value and returning to the system. For information about actual return code values for the Register a User-Written Condition Handler (CEEHDLR) bindable API, see the API topic collection in the Programming category of the IBM i Information Center.

For an HLL-specific handler, the exception message is changed to indicate that it has been handled before your handler is called. To determine whether you can modify the resume cursor from an HLL-specific handler, refer to your ILE HLL programmer's guide.

How to Percolate a Message

If you determine that an exception message is not recognized by your handler, you can percolate the exception message to the next available handler. For percolation to occur, the exception message must not be considered as a handled message. Other exception handlers in the same or previous call stack entries are given a chance to handle the exception message. The technique for percolating an exception message varies depending on the type of exception handler.

For a direct monitor handler, do not change the exception message to indicate that it has been handled. A normal return from your exception handler causes the system to percolate the message. The message is percolated to the next exception handler in the exception handler list for your call stack entry. If your handler is at the end of the exception handler list, the message is percolated to the first exception handler in the previous call stack entry.

For an ILE condition handler, you communicate a percolate action by setting a return code value and returning to the system. For information about the actual return code values for the Register a User-Written Condition Handler (CEEHDLR) bindable API, see the API topic collection in the Programming category of the IBM i Information Center.

For an HLL-specific handler, it may not be possible to percolate an exception message. Whether you can percolate a message depends on whether your HLL marks the message as handled before your handler is called. If you do not declare an HLL-specific handler, your HLL can percolate the unhandled exception message. Please refer to your ILE HLL reference manual to determine the exception messages your HLL-specific handler can handle.

How to Promote a Message

Under certain limited situations, you can choose to modify the exception message to a different message. This action marks the original exception message as handled and restarts exception processing with a new exception message. This action is allowed only from direct monitor handlers and ILE condition handlers.

For direct monitor handlers, use the Promote Message (QMHPRMM) API to promote a message. The system can promote only status and escape message types. With this API, you have some control over the handle cursor placement that is used to continue exception processing. Refer to the API topic in the Programming category of the IBM i Information Center.

For an ILE condition handler, you communicate the promote action by setting a return code value and returning to the system. For information about the actual return code values for the Register a User-Written Condition Handler (CEEHDLR) bindable API, see the API topic collection in the Programming category of the IBM i Information Center.

Default Actions for Unhandled Exceptions

If an exception message is percolated to the control boundary, the system takes a default action. If the exception is a notify message, the system sends the default reply, handles the exception, and allows the sender of the notify message to continue processing. If the exception is a status message, the system handles the exception and allows the sender of the status message to continue processing. If the exception is an escape message, the system handles the escape message and sends a function check message back to where the resume cursor is currently positioned. If the unhandled exception is a function check, all entries on the stack up to the control boundary are cancelled and the CEE9901 escape message is sent to the next prior stack entry.

Table 10 on page 114 contains default responses that the system takes when an exception is unhandled at a control boundary.

Table 10. Default Responses to Unhandled Exceptions

Message Type	Severity of Condition	Condition Raised by the Signal a Condition (CEESGL) Bindable API	Exception Originated from Any Other Source
Status	0 (Informative message)	Return the unhandled condition.	Resume without logging the message.
Status	1 (Warning)	Return the unhandled condition.	Resume without logging the message.
Notify	0 (Informative message)	Not applicable.	Log the notify message and send the default reply.
Notify	1 (Warning)	Not applicable.	Log the notify message and send the default reply.
Escape	2 (Error)	Return the unhandled condition.	Log the escape message and send a function check message to the call stack entry of the current resume point.
Escape	3 (Severe error)	Return the unhandled condition.	Log the escape message and send a function check message to the call stack entry of the current resume point.
Escape	4 (Critical ILE error)	Log the escape message and send a function check message to the call stack entry of the current resume point.	Log the escape message and send a function check message to the call stack entry of the current resume point.
Function check	4 (Critical ILE error)	Not applicable	End the application, and send the CEE9901 message to the caller of the control boundary.

Note: When the application is ended by an unhandled function check, the activation group is deleted if the control boundary is the oldest call stack entry in the activation group.

Nested Exceptions

A **nested exception** is an exception that occurs while another exception is being handled. When this happens, processing of the first exception is temporarily suspended. The system saves all of the

associated information such as the locations of the handle cursor and resume cursor. Exception handling begins again with the most recently generated exception. New locations for the handle cursor and resume cursor are set by the system. Once the new exception has been properly handled, handling activities for the original exception normally resume.

When a nested exception occurs, both of the following are still on the call stack:

- The call stack entry associated with the original exception
- The call stack entry associated with the original exception handler

To reduce the possibility of exception handling loops, the system stops the percolation of a nested exception at the original exception handler call stack entry. Then the system promotes the nested exception to a function check message and percolates the function check message to the same call stack entry. If you do not handle the nested exception or the function check message, the system ends the application by calling the Abnormal End (CEE4ABN) bindable API. In this case, message CEE9901 is sent to the caller of the control boundary.

If you move the resume cursor while processing the nested exception, you can implicitly modify the original exception. To cause this to occur, do the following:

1. Move the resume cursor to a call stack entry earlier than the call stack entry that incurred the original exception
2. Resume processing by returning from your handler

Condition Handling

ILE conditions are operating system exception messages represented in a manner that is independent of the system. An ILE condition token is used to represent an ILE condition. *Condition handling* refers to the ILE functions that you can use to handle errors separately from language-specific error handling. Other systems have implemented these functions. You can use condition handling to increase the portability of your applications between systems that have implemented condition handling.

ILE condition handling includes the following functions:

- Ability to dynamically register an ILE condition handler
- Ability to signal an ILE condition
- Condition token architecture
- Optional condition token feedback codes for bindable ILE APIs

These functions are described in the topics that follow.

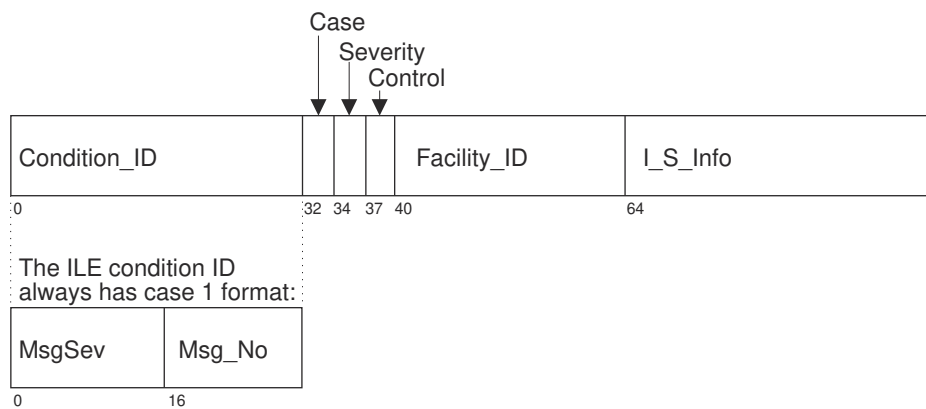
How Conditions Are Represented

The ILE **condition token** is a 12-byte compound data type that contains structured fields to convey aspects of a condition. Such aspects can be its severity, its associated message number, and information that is specific to the given instance of the condition. The condition token is used to communicate this information about a condition to the system, to message services, to bindable APIs, and to procedures. The information returned in the optional *fc* parameter of all ILE bindable APIs, for example, is communicated using a condition token.

If an exception is detected by the operating system or by the hardware, a corresponding condition token is automatically built by the system. You can also create a condition token using the Construct a Condition Token (CEENCOD) bindable API. Then you can signal a condition to the system by returning the token through the Signal a Condition (CEESGL) bindable API.

Layout of a Condition Token

Figure 45 on page 116 displays a map of the condition token. The starting bit position is shown for each field.



RV2W1032-2

Figure 45. ILE Condition Token Layout

Every condition token contains the components indicated in [Figure 45 on page 116](#):

Condition_ID

A 4-byte identifier that, with the Facility_ID, describes the condition that the token communicates. ILE bindable APIs and most applications produce case 1 conditions.

Case

A 2-bit field that defines the format of the Condition_ID portion of the token. ILE conditions are always case 1.

Severity

A 3-bit binary integer that indicates the severity of the condition. The Severity and MsgSev fields contain the same information. See [Table 10 on page 114](#) for a list of ILE condition severities. See [Table 12 on page 117](#) and [Table 13 on page 117](#) for the corresponding operating system message severities.

Control

A 3-bit field containing flags that describe or control various aspects of condition handling. The third bit specifies whether the Facility_ID has been assigned by IBM.

Facility_ID

A 3-character alphanumeric string that identifies the facility that generated the condition. The Facility_ID indicates whether the message was generated by the system or an HLL runtime. [Table 11 on page 116](#) lists the facility IDs used in ILE.

I_S_Info

A 4-byte field that identifies the instance specific information associated with a given instance of the condition. This field contains the reference key to the instance of the message associated with the condition token. If the message reference key is zero, there is no associated message.

MsgSev

A 2-byte binary integer that indicates the severity of the condition. MsgSev and Severity contain the same information. See [Table 10 on page 114](#) for a list of ILE condition severities. See [Table 12 on page 117](#) and [Table 13 on page 117](#) for the corresponding operating system message severities.

MsgNo

A 2-byte binary number that identifies the message associated with the condition. The combination of Facility_ID and MsgNo uniquely identifies a condition.

[Table 11 on page 116](#) contains the facility IDs used in ILE condition tokens and in the prefix of messages.

Table 11. Facility IDs Used in Messages and ILE Condition Tokens

Facility ID	Facility
CEE	ILE common library
CPF	IBM i message

Table 11. Facility IDs Used in Messages and ILE Condition Tokens (continued)

Facility ID	Facility
MCH	IBM i machine exception message

Condition Token Testing

You can test a condition token that is returned from a bindable API for the following:

Success

To test for success, determine if the first 4 bytes are zero. If the first 4 bytes are zero, the remainder of the condition token is zero, indicating a successful call was made to the bindable API.

Equivalent Tokens

To determine whether two condition tokens are equivalent (that is, the same *type* of condition token, but not the same *instance* of the condition token), compare the first 8 bytes of each condition token with one another. These bytes are the same for all instances of a given condition.

Equal Tokens

To determine whether two condition tokens are equal, (that is, they represent the same *instance* of a condition), compare all 12 bytes of each condition token with one another. The last 4 bytes can change from instance to instance of a condition.

Relationship of ILE Conditions to Operating System Messages

A message is associated with every condition that is raised in ILE. The condition token contains a unique ID that ILE uses to write a message associated with the condition to the message file.

The format of every runtime message is **FFFxxxx**:

FFF

The facility ID, a 3-character ID that is used by all messages generated under ILE and ILE languages. Refer to [Table 11 on page 116](#) for a list of IDs and corresponding facilities.

xxxx

The error message number. This is a hexadecimal number that identifies the error message associated with the condition.

[Table 12 on page 117](#) and [Table 13 on page 117](#) show how ILE condition severity maps to message severity.

Table 12. Mapping *ESCAPE Message Severities to ILE Condition Severities

From IBM i Message Severity	To ILE Condition Severity	To IBM i Message Severity
0-29	2	20
30-39	3	30
40-99	4	40

Table 13. Mapping *STATUS and *NOTIFY Message Severities to ILE Condition Severities

From IBM i Message Severity	To ILE Condition Severity	To IBM i Message Severity
0	0	0
1-99	1	10

IBM i Messages and the Bindable API Feedback Code

As input to a bindable API, you have the option of coding a feedback code, and using the feedback code as a return (or feedback) code check in a procedure. The feedback code is a condition token value that is provided for flexibility in checking returns from calls to other procedures. You can then use the feedback

code as input to a condition token. If the feedback code is omitted on the call to a bindable API and a condition occurs, an exception message is sent to the caller of the bindable API.

If you code the feedback code parameter in your application to receive feedback information from a bindable API, the following sequence of events occurs when a condition is raised:

1. An informational message is sent to the caller of the API, communicating the message associated with the condition.
2. The bindable API in which the condition occurred builds a condition token for the condition. The bindable API places information into the instance specific information area. The instance specific information of the condition token is the message reference key of the informational message. This is used by the system to react to the condition.
3. If a detected condition is critical (severity is 4), the system sends an exception message to the caller of the bindable API.
4. If a detected condition is not critical (severity less than 4), the condition token is returned to the routine that called the bindable API.
5. When the condition token is returned to your application, you have the following options:
 - Ignore it and continue processing.
 - Signal the condition using the Signal a Condition (CEESGL) bindable API.
 - Get, format, and dispatch the message for display using the Get, Format, and Dispatch a Message (CEEMSG) bindable API.
 - Store the message in a storage area using the Get a Message (CEEMGET) bindable API.
 - Use the Dispatch a Message (CEEMOUT) bindable API to dispatch a user-defined message to a destination that you specify.
 - When the caller of the API regains control, the informational message is removed and does not appear in the job log.

If you omit the feedback code parameter when you are calling a bindable API, the bindable API sends an exception message to the caller of the bindable API.

Debugging Considerations

The source debugger is used to debug OPM, ILE and service programs. CL commands can still be used to debug original program model (OPM) programs.

This topic presents several considerations about the source debugger. Information on how to use the source debugger can be found in the online information and in the programmer's guide for the ILE high-level language (HLL) you are using. Information on the commands to use for a specific task (for example, creating a module) can be found in your ILE HLL programmer's guide.

Debug Mode

To use the source debugger, your session must be in debug mode. **Debug mode** is a special environment in which program debug functions can be used in addition to normal system functions.

Your session is put into debug mode when you run the Start Debug (STRDBG) command.

Debug Environment

A program can be debugged in either of the two environments:

- The OPM debug environment. All OPM programs are debugged in this environment unless the OPM programs are explicitly added to the ILE debug environment.
- The ILE debug environment. All ILE programs are debugged in this environment. In addition, an OPM program is debugged in this environment if all of the following criteria are met:
 - It is a CL, COBOL or RPG program.
 - It is compiled with OPM source debug data.
 - The OPMSRC (OPM source level debug) parameter of the STRDBG command is set to *YES.

The ILE debug environment provides source level debug support. The debug capability comes directly from statement, source, or list views of the code.

Once an OPM program is in the ILE debug environment, the system will provide seamless debugging of both the ILE and OPM programs through the same user interface. For information on how to use the source debugger for OPM programs in the ILE debug environment, see online help or the programmer's guide for the equivalent ILE high-level language (HLL) you are using for the OPM language: CL, COBOL, or RPG.

Addition of Programs to Debug Mode

A program must be added to debug mode before it can be debugged. OPM programs, ILE programs, and ILE service programs can be in debug mode at the same time. As many as 20 OPM programs can be in debug mode at one time in the OPM debug environment. The number of ILE programs, service programs and OPM programs in the ILE debug environment that can be in debug mode at one time is not limited. However, the maximum amount of debug data that is supported at one time is 16MB per module.

You must have *CHANGE authority to a program or service program to add it to debug mode. A program or service program can be added to debug mode when it is stopped on the call stack.

ILE programs and service programs are accessed by the source debugger one module at a time. When you are debugging an ILE program or service program, you may need to debug a module in another program or service program. That second program or service program must be added to debug mode before the module in the second program can be debugged.

When debug mode ends, all programs are removed from debug mode.

How Observability and Optimization Affect Debugging

The optimization level and the debug data observability of bound modules affect your ability to debug a program.

Optimization Levels

With higher levels of optimization, you cannot change variables and might not be able to view the actual value of a variable during debugging. When you are debugging, set the optimization level to 10 (*NONE). This provides the lowest level of performance for the procedures in the module but allows you to accurately display and change variables. After you have completed your debugging, set the optimization level to 30 (*FULL) or 40. This provides the highest level of performance for the procedures in the module.

Debug Data Creation and Removal

Debug data is stored with each module and is generated when a module is created. To debug a procedure in a module that has been created without debug data, you must re-create the module with debug data, and then rebind the module to the ILE program or service program. You do not have to recompile all the other modules in the program or service program that already have debug data.

To remove debug data from a module, re-create the module without debug data or use the Change Module (CHGMOD) command.

Module Views

The levels of debug data available may vary for each module in an ILE program or service program. The modules are compiled separately and could be produced with different compilers and options. These debug data levels determine which **views** are produced by the compiler and which views are displayed by the source debugger. Possible values are:

***NONE**

No debug views are produced.

***STMT**

No source is displayed by the debugger, but breakpoints can be added using procedure names and statement numbers found on the compiler listing. The amount of debug data stored with this view is the minimum amount of data necessary for debugging.

***SOURCE**

The source debugger displays source if the source files used to compile the module are still present on the system.

***LIST**

The list view is produced and stored with the module. This allows the source debugger to display source even if the source files used to create the module are not present on the system. This view can be useful as a backup copy if the program will be changed. However, the amount of debug data may be quite large, especially if other files are expanded into the listing. The compiler options used when the modules were created determine whether the includes are expanded. Files that can be expanded include DDS files and include files (such as ILE C includes, ILE RPG /COPY files, and ILE COBOL COPY files).

***ALL**

All debug views are produced. As for the list view, the amount of debug data may be very large.

ILE RPG also has a debug option *COPY that produces both a source view and a copy view. The copy view is a debug view that has all the /COPY source members included.

Debugging across Jobs

You may want to use a separate job to debug programs running in your job or a batch job. This is very useful when you want to observe the function of a program without the interference of debugger panels. For example, the panels or windows that an application displays may overlay or be overlaid by the

debugger panels during stepping or at breakpoints. You can avoid this problem by starting a service job and starting the debugger in a different job from the one that is being debugged. For information on this, see the topic on testing in the CL Programming book.

OPM and ILE Debugger Support

The OPM and ILE debugger support enables source-level debugging of the OPM programs through the ILE Debugger APIs. For information about ILE Debugger APIs, see the API topic in the Programming category of the IBM i Information Center. The OPM and ILE debugger support provides seamless debugging of both the ILE and OPM programs through the same user interface. To use this support, you must compile an OPM program with the RPG, COBOL, or CL compiler. You must set the OPTION parameter to *SRCDBG or *LSTDBG for the compilation.

Watch Support

The Watch support provides the ability to stop program execution when the content of a specified storage location is changed. The storage location is specified by the name of a program variable. The program variable is resolved to a storage location and the content at this location is monitored for changes. If the content at the storage location is changed, execution stops. The interrupted program source is displayed at the point of interruption, and the source line that is highlighted will be run after the statement that changed the storage location.

Unmonitored Exceptions

When an unmonitored exception occurs, the program that is running issues a function check and sends a message to the job log. If you are in debug mode and the modules of the program were created with debug data, the source debugger shows the Display Module Source display. The program is added to debug mode if necessary. The appropriate module is shown on the display with the affected line highlighted. You can then debug the program.

Globalization Restriction for Debugging

If either of the following conditions exist:

- The coded character set identifier (CCSID) of the debug job is 290, 930, or 5026 (Japan Katakana)
- The code page of the device description used for debugging is 290, 930, or 5026 (Japan Katakana)

debug commands, functions, and hexadecimal literals should be entered in uppercase. For example:

```
BREAK 16 WHEN var=X'A1B2'  
EVAL var:X
```

The above restriction for Japan Katakana code pages does not apply when using identifier names in debug commands (for example, EVAL). However, when debugging ILE RPG, ILE COBOL, or ILE CL modules, identifier names in debug commands are converted to uppercase by the source debugger and therefore may be redisplayed differently.

Data Management Scoping

This topic contains information on the data management resources that may be used by an ILE program or service program. Before reading this, you should understand the data management scoping concepts described in [“Data Management Scoping Rules”](#) on page 45.

Details for each resource type are left to each ILE HLL programmer’s guide.

Common Data Management Resources

This topic identifies all the data management resources that follow data management scoping rules. Following each resource is a brief description of how to specify the scoping. Additional details for each resource can be found in the publications referred to.

Open file operations

Open file operations result in the creation of a temporary resource that is called an open data path (ODP). You can start the open function by using HLL open verbs, the Open Query File (OPNQRYF) command, or the Open Data Base File (OPNDBF) command. The ODP is scoped to the activation group of the program that opened the file. For OPM or ILE programs that run in a default activation group, the ODP is scoped to the call-level number. To change the scoping of HLL open verbs, you can use an override. You can specify scoping by using the open scope (OPNSCOPE) parameter on all override commands, the OPNDBF command, and the OPNQRYF command.

Overrides

Overrides are scoped to the call level, the activation group level, or the job level. To specify override scoping, use the override scope (OVRSCOPE) parameter on any override command. If you do not specify explicit scoping, the scope of the override depends on where the system issues the override. If the system issues the override from a default activation group, it is scoped to the call level. If the system issues the override from any other activation group, it is scoped to the activation group level.

Commitment definitions

Commitment definitions support scoping to the activation group level and scoping to the job level. The scoping level is specified with the control scope (CTLSCOPE) parameter on the Start Commitment Control (STRCMTCTL) command. For more information about commitment definitions, see the [Backup and Recovery](#) topic in the IBM i Information Center.

Local SQL cursors

SQL cursors can be declared and used within user built applications or within Database constructed ILE C programs or ILE C service programs built on behalf of CREATE PROCEDURE, CREATE FUNCTION, or CREATE TRIGGER LANGUAGE SQL statements.

For all types of SQL applications, there are SQL cursor scope controls. The cursor scope controls include an option to scope cursors to the activation group, as well as several other scope choices.

For complete details, refer to [IBM i DB2® for i SQL Reference](#).

Remote SQL connections

Remote connections used with SQL cursors are scoped to an activation group implicitly as part of normal SQL processing. This allows multiple conversations to exist among either one source job and multiple target jobs or multiple systems.

User interface manager

The Open Print Application (QUIOPNPA) and Open Display Application APIs support an application scope parameter. These APIs can be used to scope the user interface manager (UIM) application to either an activation group or the job. For more information about the user interface manager, see the API topic under the Programming category of the IBM i Information Center.

Open data links (open file management)

The Enable Link (QOLELINK) API enables a data link. If you use this API from within a non-default activation group, the data link is scoped to that activation group. If you use this API from within a

default activation group, the data link is scoped to the call level. For more information about open data links, see the API topic under the Programming category of the IBM i Information Center.

Common Programming Interface (CPI) Communications conversations

The activation group that starts a conversation owns that conversation. The activation group that enables a link through the Enable Link (QOLELINK) API owns the link. For information about Common Programming Interface (CPI) Communications conversations, see the API topic collection under the Programming category of the IBM i Information Center.

Hierarchical file system

The Open Stream File (OHFOPNSF) API manages hierarchical file system (HFS) files. You can use the open information (OPENINFO) parameter on this API to control scoping to either the activation group or the job level. For more information about the hierarchical file system, see the API topic under the Programming category of the IBM i Information Center.

Commitment Control Scoping

ILE introduces two changes for commitment control:

- Multiple, independent commitment definitions per job. Transactions can be committed and rolled back independently of each other. Before ILE, only a single commitment definition was allowed per job.
- If changes are pending when an activation group ends normally, the system implicitly commits the changes. Before ILE, the system did not commit the changes.

Commitment control allows you to define and process changes to resources, such as database files or tables, as a single transaction. A **transaction** is a group of individual changes to objects on the system that should appear to the user as a single atomic change. Commitment control ensures that one of the following occurs on the system:

- The entire group of individual changes occurs (a **commit** operation)
- None of the individual changes occur (a **rollback** operation)

Various resources can be changed under commitment control using both OPM programs and ILE programs.

The Start Commitment Control (STRCMTCTL) command makes it possible for programs that run within a job to make changes under commitment control. When commitment control is started by using the STRCMTCTL command, the system creates a **commitment definition**. Each commitment definition is known only to the job that issued the STRCMTCTL command. The commitment definition contains information pertaining to the resources being changed under commitment control within that job. The commitment control information in the commitment definition is maintained by the system as the commitment resources change. The commitment definition is ended by using the End Commitment Control (ENDCMTCTL) command. For more information about commitment control, see the [Backup and Recovery](#) topic.

Commitment Definitions and Activation Groups

Multiple commitment definitions can be started and used by programs running within a job. Each commitment definition for a job identifies a separate transaction that has resources associated with it. These resources can be committed or rolled back independently of all other commitment definitions started for the job.

Note: Only ILE programs can start commitment control for activation groups other than a default activation group. Therefore, a job can use multiple commitment definitions only if the job is running one or more ILE programs.

Original program model (OPM) programs run in the single-level storage default activation group. By default, OPM programs use the *DFTACTGRP commitment definition. For OPM programs, you can use the *JOB commitment definition by specifying CMTSCOPE(*JOB) on the STRCMTCTL command.

When you use the Start Commitment Control (STRCMTCTL) command, you specify the scope for a commitment definition on the commitment scope (CMTSCOPE) parameter. The **scope** for a commitment

definition indicates which programs that run within the job use that commitment definition. The default scope for a commitment definition is to the activation group of the program issuing the STRCMTCTL command. Only programs that run within that activation group will use that commitment definition, except that the default activation groups share one commitment definition. Commitment definitions that are scoped to an activation group are referred to as commitment definitions at the **activation group level**. The commitment definition started at the activation group level for a default activation group is known as the default activation group (*DFTACTGRP) commitment definition. Commitment definitions for many activation group levels can be started and used by programs that run within various activation groups for a job.

A commitment definition can also be scoped to the job. A commitment definition with this scope value is referred to as the **job-level** or *JOB commitment definition. Any program running in an activation group that does not have a commitment definition started at the activation group level uses the job-level commitment definition. This occurs if the job-level commitment definition has already been started by another program for the job. Only a single job-level commitment definition can be started for a job.

For a given activation group, only a single commitment definition can be used by the programs that run within that activation group. Programs that run within an activation group can use the commitment definition at either the job level or the activation group level. However, they cannot use both commitment definitions at the same time.

When a program performs a commitment control operation, the program does not directly indicate which commitment definition to use for the request. Instead, the system determines which commitment definition to use based on which activation group the requesting program is running in. This is possible because, at any point in time, the programs that run within an activation group can use only a single commitment definition.

Ending Commitment Control

Commitment control may be ended for either the job-level or activation-group-level commitment definition by using the End Commitment Control (ENDCMTCTL) command. The ENDCMTCTL command indicates to the system that the commitment definition for the activation group of the program making the request is to be ended. The ENDCMTCTL command ends one commitment definition for the job. All other commitment definitions for the job remain unchanged.

If the commitment definition at the activation group level is ended, programs running within that activation group can no longer make changes under commitment control. If the job-level commitment definition is started or already exists, any new file open operations specifying commitment control use the job-level commitment definition.

If the job-level commitment definition is ended, any program running within the job that was using the job-level commitment definition can no longer make changes under commitment control. If commitment control is started again with the STRCMTCTL command, changes can be made.

Commitment Control during Activation Group End

When the following conditions exist at the same time:

- An activation group ends
- The job is not ending

the system automatically ends a commitment definition at an activation group level. If both of the following conditions exist:

- Uncommitted changes exist for a commitment definition at an activation group level
- The activation group is ending normally

the system performs an implicit commit operation for the commitment definition before it ends the commitment definition. Otherwise, if either of the following conditions exist:

- The activation group is ending abnormally

- The system encountered errors when closing any files opened under commitment control scoped to the activation group

an implicit rollback operation is performed for the commitment definition at the activation group level before being ended. Because the activation group ends abnormally, the system updates the notify object with the last successful commit operation. Commit and rollback are based on pending changes. If there are no pending changes, there is no rollback, but the notify object is still updated. If the activation group ends abnormally with pending changes, the system implicitly rolls back the changes. If the activation group ends normally with pending changes, the system implicitly commits the changes.

An implicit commit operation or rollback operation is never performed during activation group end processing for the *JOB or *DFTACTGRP commitment definitions. This is because the *JOB and *DFTACTGRP commitment definitions are never ended because of an activation group ending. Instead, these commitment definitions are either explicitly ended by an ENDCMTCTL command or ended by the system when the job ends.

The system automatically closes any files scoped to the activation group when the activation group ends. This includes any database files scoped to the activation group opened under commitment control. The close operation for any such file occurs before any implicit commit operation that is performed for the commitment definition at the activation group level. Therefore, any records that reside in an I/O buffer are first forced to the database before any implicit commit operation is performed.

As part of the implicit commit operation or rollback operation, the system calls the API commit and rollback exit program for each API commitment resource. Each API commitment resource must be associated with the commitment definition at the activation group level. After the API commit and rollback exit program is called, the system automatically removes the API commitment resource.

The notify object is updated if the following conditions exist:

- An implicit rollback operation is performed for a commitment definition that is being ended because an activation group is being ended
- A notify object is defined for the commitment definition

ILE Bindable Application Programming Interfaces

ILE bindable application programming interfaces (bindable APIs) are an important part of ILE. In some cases they provide additional function beyond that provided by a specific high-level language. For example, not all HLLs offer intrinsic means to manipulate dynamic storage. In those cases, you can supplement an HLL function by using particular bindable APIs. If your HLL provides the same function as a particular bindable API, use the HLL-specific one.

Bindable APIs are HLL independent. This can be useful for mixed-language applications. For example, if you use only condition management bindable APIs with a mixed-language application, you will have uniform condition handling semantics for that application. This makes condition management more consistent than when using multiple HLL-specific condition handlers.

The bindable APIs provide a wide range of function including:

- Activation group and control flow management
- Condition management
- Date and time manipulation
- Dynamic screen management
- Math functions
- Message handling
- Program or procedure call management and operational descriptor access
- Source debugger
- Storage management

For reference information about the ILE bindable APIs, see the API topic under the Programming category of the IBM i Information Center.

ILE Bindable APIs Available

Most bindable APIs are available to any HLL that ILE supports. ILE provides the following bindable APIs:

Activation Group and Control Flow Bindable APIs

- Abnormal End (CEE4ABN)
- Find a Control Boundary (CEE4FCB)
- Register Activation Group Exit Procedure (CEE4RAGE)
- Register Call Stack Entry Termination User Exit Procedure (CEERTX)
- Signal the Termination-Imminent Condition (CEETREC)
- Unregister Call Stack Entry Termination User Exit Procedure (CEEUTX)

Condition Management Bindable APIs

- Construct a Condition Token (CEENCOD)
- Decompose a Condition Token (CEEDCOD)
- Handle a Condition (CEE4HC)
- Move the Resume Cursor to a Return Point (CEEMRCR)
- Register a User-Written Condition Handler (CEEHDLR)
- Retrieve ILE Version and Platform ID (CEEGPID)
- Return the Relative Invocation Number (CEE4RIN)
- Signal a Condition (CEESGL)
- Unregister a User Condition Handler (CEEHDLU)

Date and Time Bindable APIs

Calculate Day-of-Week from Lilian Date (CEEDYWK)
Convert Date to Lilian Format (CEEDAYS)
Convert Integers to Seconds (CEEISEC)
Convert Lilian Date to Character Format (CEEDATE)
Convert Seconds to Character Timestamp (CEEDATM)
Convert Seconds to Integers (CEESECI)
Convert Timestamp to Number of Seconds (CEESECS)
Get Current Greenwich Mean Time (CEEGMT)
Get Current Local Time (CEELOCT)
Get Offset from Universal Time Coordinated to Local Time (CEEUTCO)
Get Universal Time Coordinated (CEEUTC)
Query Century (CEEQCEN)
Return Default Date and Time Strings for Country or Region (CEEFMDT)
Return Default Date String for Country or Region (CEEFMDA)
Return Default Time String for Country or Region (CEEFTMT)
Set Century (CEESCEN)

Math Bindable APIs

The x in the name of each math bindable API refers to one of the following data types:

- I**
32-bit binary integer
- S**
32-bit single floating-point number
- D**
64-bit double floating-point number
- T**
32-bit single floating-complex number (both real and imaginary parts are 32 bits long)
- E**
64-bit double floating-complex number (both real and imaginary parts are 64 bits long)

Absolute Function (CEESxABS)
Arccosine (CEESxACS)
Arcsine (CEESxASN)
Arctangent (CEESxATN)
Arctangent2 (CEESxAT2)
Conjugate of Complex (CEESxCJG)
Cosine (CEESxCOS)
Cotangent (CEESxCTN)
Error Function and Its Complement (CEESxERx)
Exponential Base e (CEESxEXP)
Exponentiation (CEESxXPx)
Factorial (CEE4SIFAC)
Floating Complex Divide (CEESxDVD)
Floating Complex Multiply (CEESxMLT)
Gamma Function (CEESxGMA)
Hyperbolic Arctangent (CEESxATH)
Hyperbolic Cosine (CEESxCSH)
Hyperbolic Sine (CEESxSNH)
Hyperbolic Tangent (CEESxTNH)
Imaginary Part of Complex (CEESxIMG)

Log Gamma Function (CEESxLGM)
Logarithm Base 10 (CEESxLG1)
Logarithm Base 2 (CEESxLG2)
Logarithm Base e (CEESxLOG)
Modular Arithmetic (CEESxMOD)
Nearest Integer (CEESxNIN)
Nearest Whole Number (CEESxNWN)
Positive Difference (CEESxDIM)
Sine (CEESxSIN)
Square Root (CEESxSQT)
Tangent (CEESxTAN)
Transfer of Sign (CEESxSGN)
Truncation (CEESxINT)

Additional math bindable API:

Basic Random Number Generation (CEERANO)

Message Handling Bindable APIs

Dispatch a Message (CEEMOUT)
Get a Message (CEEMGET)
Get, Format, and Dispatch a Message (CEEMSG)

Program or Procedure Call Bindable APIs

Get String Information (CEEGSI)
Retrieve Operational Descriptor Information (CEEDOD)
Test for Omitted Argument (CEETSTA)

Source Debugger Bindable APIs

Allow a Program to Issue Debug Statements (QteSubmitDebugCommand)
Enable a Session to Use the Source Debugger (QteStartSourceDebug)
Map Positions from One View to Another (QteMapViewPosition)
Register a View of a Module (QteRegisterDebugView)
Remove a View of a Module (QteRemoveDebugView)
Retrieve the Attributes of the Source Debug Session (QteRetrieveDebugAttribute)
Retrieve the List of Modules and Views for a Program (QteRetrieveModuleViews)
Retrieve the Position Where the Program Stopped (QteRetrieveStoppedPosition)
Retrieve Source Text from the Specified View (QteRetrieveViewText)
Set the Attributes of the Source Debug Session (QteSetDebugAttribute)
Take a Job Out of Debug Mode (QteEndSourceDebug)

Storage Management Bindable APIs

Create Heap (CEECRHP)
Define Heap Allocation Strategy (CEE4DAS)
Discard Heap (CEEDSHP)
Free Storage (CEEFRST)
Get Heap Storage (CEEGTST)
Mark Heap (CEEMKHP)
Reallocate Storage (CEECZST)
Release Heap (CEERLHP)

Dynamic Screen Manager Bindable APIs

The dynamic screen manager (DSM) bindable APIs are a set of screen I/O interfaces that provide a dynamic way to create and manage display screens for the ILE high-level languages.

The DSM APIs fall into the following functional groups:

- **Low-level services**

The low-level services APIs provide a direct interface to the 5250 data stream commands. The APIs are used to query and manipulate the state of the display screen; to create, query, and manipulate input and command buffers that interact with the display screen; and to define fields and write data to the display screen.

- **Window services**

The window services APIs are used to create, delete, move, and resize windows; and to manage multiple windows concurrently during a session.

- **Session services**

The session services APIs provide a general paging interface that can be used to create, query, and manipulate sessions; and to perform input and output operations to sessions.

For information about the DSM bindable APIs, see the API topic collection under the Programming category of the IBM i Information Center.

Advanced Optimization Techniques

This topic describes the following techniques you can use to optimize your ILE programs and service programs:

- [“Adaptive Code Generation” on page 149](#)
- [“Advanced Argument Optimization” on page 142](#)
- [“Interprocedural analysis \(IPA\)” on page 136](#)
- [“Licensed Internal Code Options” on page 143](#)
- [“Program Profiling” on page 131](#)

Program Profiling

Program profiling is an advanced optimization technique to reorder procedures, or code within procedures, in ILE programs and service programs based on statistical data gathered while running the program. This reordering can improve instruction cache utilization and reduce paging required by the program, thereby improving performance. The semantic behavior of the program is not affected by program profiling.

The performance improvement realized by program profiling depends on the type of application. Generally speaking, you can expect more improvement from programs that spend the majority of time in the application code itself, rather than spending time in the runtime or doing input/output processing. The performance of program code produced when applying profile data depends upon the optimizing translator correctly identifying the most important portions of the program during typical use. Therefore, it is important to gather profile data while performing the tasks that will be performed by end users, and using input data that is similar to that expected in the environment in which the program will be run.

Program profiling is available only for ILE programs and service programs that meet the following conditions:

- The programs were created specifically for V4R2M0 or later releases.
- If the programs were created for a release prior to V5R2M0, the program's target release must be the same as the current system's release.
- The programs were compiled using an optimization level of *FULL (30) or above. On V5R2M0 and later systems, bound modules with less than optimization level 30 are allowed, but do not participate in application profiling.

Note: Because of the optimization requirements, you should fully debug your programs before using program profiling.

Types of Profiling

You can profile your programs in the following two ways:

- Block order
- Procedure order and block order

Block order profiling records the number of times each side of a conditional branch is taken. When block order profiling data is applied to a program, a variety of profile-based optimizations are performed within procedures by the optimizing translator. For one of these optimizations, code is ordered so that the most frequently executed code paths in a procedure are contiguous within the program object. This reordering improves performance by utilizing processor components such as the instruction cache and instruction prefetch unit more effectively.

Procedure order profiling records the number of times each procedure calls another procedure within the program. Procedures within the program are reordered so that the most frequently called procedures are packaged together. This reordering improves performance by reducing memory paging.

Even though you can choose to apply only block order profiling to your program, it is recommended that you apply both types for the largest performance gains.

How to Profile a Program

Profiling a program is a five step process:

1. Enable the program to collect profiling data.
2. Start the program profiling collection on the system with the Start Program Profiling (STRPGMPRF) command.
3. Collect profiling data by running the program through its high-use code paths. Because program profiling uses statistical data gathered while running the program to perform these optimizations, it is critical that this data be collected over typical uses of your application.
4. End the program profiling collection on the system with the End Program Profiling (ENDPGMPRF) command.
5. Apply the collected profiling data to the program by requesting that code be reordered for optimal performance based on the collected profiling data.

Enabling the program to collect profiling data

A program is enabled to collect profiling data if at least one of the modules bound into the program is enabled to collect profiling data. Enabling a program to collect profiling data can be done either by changing one or more *MODULE objects to collect profiling data and then creating or updating the program with these modules, or by changing the program after it is created to collect profiling data. Both techniques result in a program with bound modules enabled to collect profiling data.

Depending on the ILE language you are using, there may be an option on the compiler command to create the module as enabled to collect profiling data. The change module (CHGMOD) command can also be used by specifying *COL on the profiling data (PRFDTA) parameter to change any ILE module to collect profiling data, as long as the ILE language supports an optimization level of at least *FULL (30).

To enable a program to collect profiling data after creation through either the Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) commands, do the following for an observable program:

- Specify *COL on the profiling data (PRFDTA) parameter. This specification affects all modules bound in the program that:
 - Were created for a release of V4R2M0 or later. If you are using a system earlier than V5R2M0, the program must be on a system at the same release level for which the program was created in order to enable profiling data collection. The same restriction applies for bound modules.
 - Have an optimization level of 30 or above. In V5R2M0 and later releases, any bound modules with less than optimization level 30 will be allowed, but will not participate in application profiling.

Note: A program enabled to collect application profiling data on a system with release prior to V5R2M0 can have that data applied on a V5R2M0 or later system, but the results may not be optimal. If you intend to apply profiling data or use the resulting program on a V5R2M0 or later system, you should enable or re-enable profiling data collection for the program on a V5R2M0 or later system.

Enabling a module or program to collect profiling data requires that the object be re-created. Therefore, the time required to enable a module or program to collect profiling data is comparable to the time it takes to force recreate the object (FRCCRT parameter). Additionally, the size of the object will be larger due to the extra machine instructions generated by the optimizing translator.

Once you enable a program or module to collect profiling data, creation data observability cannot be removed until one of the following occurs:

- The collected profiling data is applied to the program.

- The program or module is changed so that it cannot collect profiling data.

Use the Display Module (DSPMOD), Display Program (DSPPGM) or Display Service Program (DSPSRVPGM) commands, specifying DETAIL(*BASIC), to determine if a module or program is enabled to collect profiling data. For programs or service programs use option 5 (display description) from the DETAIL(*MODULE) to determine which of the bound module(s) are enabled to collect profiling data. See topic [“How to Tell if a Program or Module is Profiled or Enabled for Collection”](#) on page 135 for more details.

Note: If a program already has profiling data collected (the statistical data gathered while the program is running), this data is cleared when a program is re-enabled to collect profiling data. See [“Managing Programs Enabled to Collect Profiling Data”](#) on page 134 for details.

Collect Profiling Data

Program profiling must be started on the machine that a program enabled to collect profiling data is to be run on in order for that program to update profiling data counts. This enables large, long-running applications to be started and allowed to reach a steady state before gathering profiling data. This gives you control over when data collection occurs.

Use the Start Program Profiling (STRPGMPRF) command to start program profiling on a machine. To end program profiling on a machine, use the End Program Profiling (ENDPGMPRF) command. Both commands are shipped with the public authority of *EXCLUDE. Program profiling is ended implicitly when a machine is IPLed.

Once program profiling is started, any program or service program that is run that is also enabled to collect profiling data will update its profiling data counts. This will happen regardless of whether or not the program was activated before the STRPGMPRF command was issued.

If the program you are collecting profiling data on can be called by multiple jobs on the machine, the profiling data counts will be updated by all of these jobs. If this is not desirable, a duplicate copy of the program should be made in a separate library and that copy should be used instead.

Note:

1. When program profiling is started on a machine, profiling data counts are incremented while a program that is enabled to collect profiling data is running. Therefore it is possible that "stale" profiling data counts are being added to if this program was previously run without subsequently clearing these counts. You can force the profiling data counts to be cleared in several ways. See [“Managing Programs Enabled to Collect Profiling Data”](#) on page 134 for details.
2. Profiling data counts are not written to DASD each time they are incremented because doing so would cause too great a degradation to the program's runtime. Profiling data counts are only written to DASD when the program is naturally paged out. To ensure profiling data counts are written to DASD, use the Clear Pool (CLRPOOL) command to clear the storage pool in which the program is running.

Applying the Collected Profiling Data

Applying collected profiling data does the following:

1. Instructs the machine to use the collected profiling data to reorder procedures (procedure order profiling data) in the program for optimal performance.
2. Instructs the machine to use the collected profiling data (basic block profiling data) to reorder the code within procedures in the program for optimal performance.
3. Removes the machine instructions from the program that were previously added when the program was enabled to collect profiling data. The program can then no longer collect profile data.
4. Stores the collected profiling data in the program as observable data:
 - *BLKORD (basic block profiling observability)
 - *PRCORD (procedure order profiling observability)

Once the collected data has been applied to the program, it cannot be applied again. To apply profiling data again requires you to go through the steps outlined in “How to Profile a Program” on page 132. **Any previously applied profiling data is discarded when a program is enabled to collect profiling data.**

If you want to apply the data you already collected again, you may want to make a copy of the program before applying profiling data. This may be desirable if you are experimenting with the benefits derived from each type of profiling (either block order or block and procedure ordered).

To apply profiling data, use the Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) command. For the profiling data (PRFDTA) parameter specify:

- Block order profiling data (*APYBLKORD)
- Both block order and procedure profiling data (*APYALL) or (*APYPCORD)

IBM recommends using *APYALL.

Applying profiling data to the program creates and saves two additional forms of observability with the program. You can remove these additional observabilities by using the Change Program (CHGPGM) and Change Service Program (CHGSRVPGM) commands.

- *BLKORD observability is implicitly added when block order profiling data is applied to the program. This allows the machine to preserve the applied block order profiling data for the program in cases where the program is recreated.
- Applying procedure order profiling data to the program implicitly adds *PCORD and *BLKORD observability. This allows the machine to preserve the applied procedure order profiling data for the program in cases where the program is either recreated or updated.

For example, you apply block order profiling data to your program and then subsequently remove *BLKORD observability. The program is still block order profiled. However, any change that causes your program to be recreated will also cause it to no longer be block order profiled.

Managing Programs Enabled to Collect Profiling Data

Changing a program that is enabled to collect profiling data by using the Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) commands will implicitly cause profiling data counts to be zeroed if the change requires the program be recreated. For example, if you change a program that is enabled to collect profiling data from optimization level *FULL to optimization level 40, any collected profiling data will be implicitly cleared. This is also true if a program that is enabled to collect profiling data is restored, and FRCOBJCVN(*YES *ALL) is specified on the Restore Object (RSTOBJ) command.

Likewise, updating a program that is enabled to collect profiling data by using the Update Program (UPDPGM) or Update Service Program (UPDSRVPGM) commands will implicitly cause profiling data counts to be cleared if the resulting program is still enabled to collect profiling data. For example, program P1 contains modules M1 and M2. Module M1 bound in P1 is enabled to collect profiling data but module M2 is not. So long as one of the modules is enabled, updating program P1 with module M1 or M2 will result in a program that is still enabled to collect profiling data. All profiling data counts will be cleared. However, if module M1 is changed to no longer be enabled to collect profiling data by specifying *NOCOL on the profiling data (PRFDTA) parameter of the Change Module (CHGMOD) command, updating program P1 with M1 will result in program P1 no longer being enabled to collect profiling data.

You can explicitly clear profiling counts from the program by specifying the *CLR option on the profiling data (PRFDTA) parameter of the Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) commands. Note the program must not be activated to use the *CLR option.

If you no longer want the program to collect profiling data, you can take one of the following actions:

- Specify *NOCOL on the profiling data (PRFDTA) parameter of the Change Program (CHGPGM) command.
- Specify *NOCOL on the profiling data (PRFDTA) parameter of the Change Service Program (CHGSRVPGM) command.

Either action changes the program back to the state before it collected profiling data. You can also change the PRFDTA value of the modules to *NOCOL with the CHGMOD command or by recompiling the modules and rebinding the modules into the program.

Managing Programs with Profiling Data Applied to Them

If a program that has profiling data applied is changed by using the Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) commands, you will lose applied profiling data if both of these conditions are true:

- The change requires the program to be recreated.

Note: The optimization level of a program that has profiling data applied cannot be changed to be less than optimization level 30. This is because the profiling data is optimization level dependent.

- The required profiling observability has been removed.

Also all applied profiling data will be lost if the change request is to enable the program to collect profiling data, regardless of whether profiling observability has been removed or not. Such a request will result in a program that is enabled to collect profiling data.

Here are some examples:

- Program A has procedure order and block order profiling data applied. *BLKORD observability has been removed from the program but *PRCORD observability has not. A CHGPGM command is run to change the performance collection attribute of program A, which also requires the program to be recreated. This change request will cause program A to no longer be block order profiled. However, the procedure order profiling data will still be applied.
- Program A has procedure order and block order profiling data applied. *BLKORD and *PRCORD observability have been removed from the program. A CHGPGM command is run to change the user profile attribute of program A, which also requires the program to be recreated. This change request will cause program A to no longer be block order or procedure order profiled. Program A will go back to the state before the profiling data was applied.
- Program A has block order profiling data applied. *BLKORD observability has been removed from the program. A CHGPGM command is run to change the text of the program, which does **not** require the program to be recreated. After this change, program A will still be block order profiled.
- Program A has procedure order and block order profiling data applied. This does not remove *PRCORD and *BLKORD observability from the program. Run a CHGPGM command to enable the program to collect profiling data (this recreates the program). This causes program A to no longer be block order or procedure order profiled. This leaves the program in a state as if profiling data was never applied. This enables the program to collect profiling data with all profiling data counts cleared.

A program that has had profiling data applied to it (*APYALL, *APYBLKORD, or *APYPRCORD) cannot be immediately changed to an unprofiled program by specifying PRFDTA(*NOCOL) on the CHGPGM or CHGSRVPGM commands. This feature is intended as a safety net to avoid accidental loss of profiling data. If this is truly what is intended, the program must first be changed to PRFDTA(*COL), effectively removing the existing profiling data, and then changed to PRFDTA(*NOCOL).

How to Tell if a Program or Module is Profiled or Enabled for Collection

Use the Display Program (DSPPGM) or Display Service Program (DSPSRVPGM) commands, specifying DETAIL(*BASIC), to determine the program profiling data attribute of a program. The value of "Profiling data" will be one of the following values:

- *NOCOL - The program is not enabled to collect profiling data.
- *COL - One or more modules in the program are enabled to collect profiling data. This value does not indicate if profiling data was actually collected.
- *APYALL - Block order and procedure order profiling data are applied to this program. The collection of profiling data is no longer enabled.
- *APYBLKORD - Block order profiling data is applied to the procedures of one or more bound modules in this program. This applies to only the bound modules that were previously enabled to collect profiling data. The collection of profiling data is no longer enabled.
- *APYPRCORD - Procedure order program profiling data is applied to this program. The collection of profiling data is no longer enabled.

To have only procedure order profiling applied to a program:

- First profile it by specifying *APYALL or *APYPRCORD (which is the same as *APYALL).
- Then remove the *BLKORD observability and recreate the program.

To display the program profiling data attribute of a module bound within the program, use DSPPGM or DSPSRVPGM DETAIL(*MODULE). Specify option 5 on the modules bound into the program to see the value of this parameter at the module level. The value of "Profiling data" will be one of the following values:

- *NOCOL - This bound module is not enabled to collect profiling data.
- *COL - This bound module is enabled to collect profiling data. This value does not indicate if profiling data was actually collected.
- *APYBLKORD - Block order profiling data is applied to one or more procedures of this bound module. The collection of profiling data is no longer enabled.

In addition DETAIL(*MODULE) displays the following fields to give an indication of the number of procedures affected by the program profiling data attribute.

- Number of procedures - Total number of procedures in the module.
- Number of procedures block reordered - The number of procedures in this module that are basic block reordered.
- Number of procedures block order measured - Number of procedures in this bound module that had block order profiling data collected when block order profiling data was applied. When the benchmark was run, it could be the case that no data was collected for a specific procedure because the procedure was not executed in the benchmark. Thus this count reflects the actual number of procedures that were executed with the benchmark.

Use DSPMOD command to determine the profiling attribute of a module. The value of "Profiling data" will be one of the following. It will never show *APYBLKORD because basic block data can be applied only to modules bound into a program, never to stand-alone modules.

- *NOCOL - module is not enabled to collect profile data.
- *COL - module is enabled to collect profile data.

Interprocedural analysis (IPA)

This topic provides an overview of the Interprocedural Analysis (IPA) processing that is available through the IPA option on the CRTPGM and CRTSRVPGM commands.

At compile time, the optimizing translator performs both intraprocedural and interprocedural analysis. Intraprocedural analysis is a mechanism for performing optimization for each function within a compilation unit, using only the information available for that function and compilation unit. Interprocedural analysis is a mechanism for performing optimization across function boundaries. The optimizing translator performs interprocedural analysis, but only within a compilation unit. Interprocedural analysis that is performed by the IPA compiler option improves on the limited interprocedural analysis described above. When you run interprocedural analysis through the IPA option, IPA performs optimizations across the entire program. It also performs optimizations not otherwise available at compile time with the optimizing translator. The optimizing translator or the IPA option performs the following types of optimizations:

- *Inlining across compilation units.* Inlining replaces certain function calls with the actual code of the function. Inlining not only eliminates the overhead of the call, but also exposes the entire function to the caller and thus enables the compiler to better optimize your code.
- *Program partitioning.* Program partitioning improves performance by reordering functions to exploit locality of reference. Partitioning places functions that call each other frequently in closer proximity in memory. For more information on program partitioning, see [“Partitions created by IPA” on page 141](#).

- *Coalescing of global variables.* The compiler puts global variables into one or more structures and accesses the variables by calculating the offsets from the beginning of the structures. This lowers the cost of variable access and exploits data locality.
- *Code straightening.* Code straightening streamlines the flow of your program.
- *Unreachable code elimination.* Unreachable code elimination removes unreachable code within a function.
- *Call graph pruning of unreachable functions.* The call graph pruning of unreachable functions removes code that is 100% inlined or never referred to.
- *Intraprocedural constant propagation and set propagation.* IPA propagates floating point and integer constants to their uses and computes constant expressions at compile time. Also, variable uses that are known to be one of several constants can result in the folding of conditionals and switches.
- *Intraprocedural pointer alias analysis.* IPA tracks pointer definitions to their uses, resulting in more refined information about memory locations that a pointer dereference may use or define. This enables other parts of the compiler to better optimize code around such dereferences. IPA tracks data and function pointer definitions. When a pointer can only refer to a single memory location or function, IPA rewrites it to be an explicit reference to the memory location or function.
- *Intraprocedural copy propagation.* IPA propagates expressions, and defines some variables to the uses of the variable. This creates additional opportunities for the folding of constant expressions. It also eliminates redundant variable copies.
- *Intraprocedural unreachable code and store elimination.* IPA removes definitions of variables that it cannot reach, along with the computation that feeds the definition.
- *Conversion of reference (address) arguments to value arguments.* IPA converts reference (address) arguments to value arguments when the formal parameter is not written in the called procedure.
- *Conversion of static variables to automatic (stack) variables.* IPA converts static variables to automatic (stack) variables when their use is limited to a single procedure call.

The runtime for code that is optimized using IPA is normally faster than for code optimized only at compile time. Not all applications are suited for IPA optimization, however, and the performance gains that are realized from using IPA will vary. For certain applications, the performance of the application may not improve when using interprocedural analysis. In fact, in some rare cases, the performance of the application can actually degrade when you use interprocedural analysis. If this occurs, we suggest that you not use interprocedural analysis. The performance improvement realized by interprocedural analysis depends on the type of application. Applications that will most likely show performance gains are those that have the following characteristics:

- Contain a large number of functions
- Contain a large number of compilation units
- Contain a large number of functions that are not in the same compilation units as their callers
- Do not perform a large number of input and output operations

Interprocedural optimization is available only for ILE programs and service programs that meet the following conditions:

- You created the modules bound into the program or service program specifically for V4R4M0 or later releases.
- You compiled the modules bound into the program or service program with an optimization level of 20 (*BASIC) or higher.
- The modules bound into the program or service program have IL data that is associated with them. Use the create module option MODCRTOPT(*KEEPILDTA) to keep intermediate language (IL) data with the module.

Note: Because of the optimization requirements, you should fully debug your programs before you use interprocedural analysis.

How to optimize your programs with IPA

To use IPA to optimize your program or service program objects, perform the following steps:

1. Make sure that you compile all of the modules necessary for the program or service program with MODCROPT(*KEEPILDTA) and with an optimization level of 20 or greater (preferably 40). You can use the DSPMOD command with the DETAIL(*BASIC) parameter to verify that a single module is compiled with the correct options. The **Intermediate language data** field will have a value of *YES if IL data is present. The **Optimization level** field indicates the optimization level of the module.
2. Specify IPA(*YES) on the CRTPGM or CRTSRVPGM command. When the IPA portion of the bind runs, the system displays status messages to indicate IPA progress.

You can further define how IPA optimizes your program by using the following parameter:

- Specify IPACTLFILE(*IPA-control-file*) to provide additional IPA suboption information. See [“IPA control file syntax” on page 138](#) for a listing of the options you can specify in the control file.

When you specify IPA(*YES) on the CRTPGM command, you cannot also allow updates to the program (that is, you cannot specify ALWUPD(*YES)). This is also true for the ALWLIBUPD parameter on the CRTSRVPGM command. If specified along with IPA(*YES), the parameter must be ALWLIBUPD(*NO).

IPA control file syntax

The IPA control file is a stream file that contains additional IPA processing directives. The control file can be a member of a file, and uses the QSYS.LIB naming convention (for example, /qsys.lib/mylib.lib/xx.file/yy.mbr). The IPACTLFILE parameter identifies the path name of this file.

IPA issues an error message if the control file directives have syntax that is not valid.

You can specify the following directives in the control file:

exits=name[,name]

Specifies a list of functions, each of which always ends the program. You can optimize calls to these functions (for example, by eliminating save and restore sequences), because the calls never return to the program. These functions must not call any other parts of the program that have IL data associated with them.

inline=attribute

Specifies how you want the compiler to identify functions that you want it to process inline. You can specify the following attributes for this directive:

auto

Specifies that the inliner should determine if a function can be inlined on the basis of the inline-limit and inline-threshold values. The noline directive overrides automatic inlining. This is the default.

noauto

Specifies that IPA should consider for inlining only the functions that you have specified by name with the inline directive.

name[,name]

Specifies a list of functions that you want to inline. The functions may or may not be inlined.

name[,name] from name[,name]

Specifies a list of functions that are desirable candidates for inlining, if a particular function or list of functions calls the functions. The functions may or may not be inlined.

inline-limit=num

Specifies the maximum relative size (in abstract code units) to which a function can grow before inlining stops. Abstract code units are proportional in size to the executable code in the function. Larger values for this number allow the compiler to inline larger subprograms, more subprogram calls, or both. This directive is applicable only when inline=auto is on. The default value is 8192.

inline-threshold=size

Specifies the maximum size (in abstract code units) of a function that can be a candidate for automatic inlining. This directive is applicable only when `inline=auto` is on. The default size is 1024.

isolated=name[,name]

Specifies a list of "isolated" functions. Isolated functions are those that do not directly (or indirectly through another function within its call chain) refer to or change global variables that are accessible to visible functions. IPA assumes that functions that are bound from service programs are isolated.

lowfreq=name[,name]

Specifies names of functions that are expected to be called infrequently. These are typically error handling functions or trace functions. IPA can make other parts of the program faster by doing less optimization for calls to these functions.

missing=attribute

Specifies the interprocedural behavior of *missing* functions. Missing functions are those that do not have IL data associated with them, and that are not explicitly named in an `unknown`, `safe`, `isolated`, or `pure` directive. These directives specify how much optimization IPA can safely perform on calls to library routines that do not have IL data associated with them.

IPA has no visibility to the code within these functions. You must ensure that all user references are resolved with user libraries or runtime libraries.

The default setting for this directive is `unknown`. *Unknown* instructs IPA to make pessimistic assumptions about the data that may be used and changed through a call to such a missing function, and about the functions that may be called indirectly through it. You can specify the following attributes for this directive:

unknown

Specifies that the missing functions are "unknown". See the description for the `unknown` directive below. This is the default attribute.

safe

Specifies that the missing functions are "safe". See the description for the `safe` directive, below.

isolated

Specifies that the missing functions are "isolated". See the description for the `isolated` directive, above.

pure

Specifies that the missing functions are "pure". See the description for the `pure` directive, below.

noinline=name[,name]

Specifies a list of functions that the compiler will not inline.

noinline=name[,name] from name[,name]

Specifies a list of functions that the compiler will not inline, if the functions are called from a particular function or list of functions.

partition=small|medium|large|unsigned-integer

Specifies the size of each program partition that IPA creates. The size of the partition is directly proportional to the time required to link and the quality of the generated code. When the partition size is large, the time required to link is longer but the quality of the generated code is generally better.

The default for this directive is `medium`.

For a finer degree of control, you can use an unsigned-integer value to specify the partition size. The integer is in abstract code units, and its meaning may change between releases. You should only use this integer for very short term tuning efforts, or for those situations where the number of partitions must remain constant.

pure=name[,name]

Specifies a list of *pure* functions. These are functions that are `safe` and `isolated`. A pure function has no observable internal state. This means that the returned value for a given call of a function is independent of any previous or future calls of the function.

safe=name[,name]

Specifies a list of *safe* functions. These are functions that do not directly or indirectly call any function that has IL data associated with it. A safe function may refer to and change global variables.

unknown=name[,name]

Specifies a list of *unknown* functions. These are functions that are not safe, isolated, or pure.

IPA usage notes

- Use of IPA can increase bind time. Depending on the size of the application and the speed of your processor, the bind time can increase significantly.
- IPA can generate significantly larger bound program and service program objects than traditional binding.
- While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause functioning programs that contain errors to fail.
- Because IPA will compile functions inline, take care when using APIs that accept a relative stack frame offset (for example, QMHRCVPM).
- To compile functions inline, IPA uses its own inliner rather than the backend inliner. Any parameters given for the backend inliner, such as using the `INLINE` option on the compile command, are ignored. Parameters for the IPA inliner are given in the IPA control file.

IPA restrictions and limitations

- You cannot use either the UPDPGM or UPDSRVPGM on a bound program or service program that IPA has optimized.
- You cannot debug any programs or service programs that IPA has optimized with the normal source debug facilities. This is because IPA does not maintain debug information within the IL data and in fact throws away any debug information when it generates the output partitions. As such, the source debugger does not handle IPA programs or service programs.
- There is a limit of 10,000 output partitions. If you reach this limit, the bind will fail, and the system will send a message. If you reach this limit, you should run the CRTPGM or CRTSRVPGM command again, and specify a larger partition size. See the partition directive in [“IPA control file syntax” on page 138](#).
- There are certain IPA limitations that may apply to your program if that program contains SQL data. If the compiler that you use allows an option to keep the IL data, then these limitations do not apply. If the compiler you use does not allow an option to keep the IL data, you must perform the steps listed below to use IPA on a program containing SQL data. For example, consider a C program with embedded SQL statements. You would normally compile this source with the CRTSQLCI command; however, that command does not have a MODCRTOPT(*KEEPILDTA) option.

Perform the following steps to create a *MODULE that contains both embedded SQL data and IL data.

1. Compile an SQL C source file with the CRTSQLCI command. Specify the OPTION(*NOGEN) and the TOSRCFILE(QTEMP/QSQLTEMP) compiler options. This step precompiles the SQL statements and places the SQL precompiler data into the associated space of the original source file. It also places the C source into a member with the same name in temporary source physical file QTEMP/QSQLTEMP.
 2. Compile the C source file in QTEMP/QSQLTEMP with the MODCRTOPT(*KEEPILDTA) option on the compiler command. This action creates an SQL C *MODULE object, and propagates the preprocessor data from the associated space of the original source file into the module object. This *MODULE object also contains the IL data. At this point, you can specify the *MODULE object on the CRTPGM or CRTSRVPGM command with the IPA(*YES) parameter.
- IPA cannot optimize modules that you compile at optimization level 10 (*NONE). IPA requires information within the IL data that is available only at higher optimization levels.
 - IPA cannot optimize modules that do not contain IL data. Because of this, IPA can optimize only those modules that you create with compilers that offer the MODCRTOPT(*KEEPILDTA) option. Currently, this includes the C and C++ compilers.

- For a program, the module containing the program entry point, which is typically the main function, must have the correct attributes as noted above, or IPA will fail. For a service program, at least one of the modules containing exported functions must have the correct attributes as noted above, or IPA will fail. It is desirable that the other modules within the program or service program also have the correct attributes, but it is not required. Any modules without the correct attributes will be accepted by IPA, but they will not be optimized.
- IPA might not be able to correctly optimize modules compiled with the RTBND(*LLP64) option on either the Create C++ Module (CRTCPMOD) or Create Bound C++ Program (CRTBNDCPP) command. If virtual functions are not used in the module, then IPA can optimize the module. If virtual functions are used, then the MODCRTOPT(*NOKEEPILDTA) option should be specified.
- IPA might not be able to correctly optimize modules that contain decimal floating-point data or variables.
- IPA cannot correctly optimize modules that contain thread local storage variables.

Partitions created by IPA

The final program or service program created by IPA consists of partitions. IPA creates a *MODULE for each partition. Partitions have two purposes:

- They improve the locality of reference in a program by concentrating related code in the same region of storage.
- They reduce the memory requirements during object code generation for that partition.

There are three types of partitions:

- An initialization partition. This contains initialization code and data.
- The primary partition. This contains information for the primary entry point for the program.
- Secondary or other partitions.

IPA determines the number of each type of partition in the following ways:

- The 'partition' directive within the control file specified by the IPACTLFILE parameter. This directive indicates how large to make each partition.
- The connectivity within the program call graph. Connectivity refers to the volume of calls between functions in a program.
- Conflict resolution between compiler options specified for different compilation units. IPA attempts to resolve conflicts by applying a common option across all compilation units. If it cannot, it forces the compilation units for which the effects of the original option are to be maintained into separate partitions.

One example of this is the *Licensed Internal Code Options* (LICOPTs). If two compilation units have conflicting LICOPTs, IPA cannot merge functions from those compilation units into the same output partition. Refer to “Partition Map” on page 169 for an example of the Partition Map listing section. IPA creates the partitions in a temporary library, and binds the associated *MODULEs together to create the final program or service program. IPA creates the partition *MODULE names using a random prefix (for example, QD0068xxxx where xxxx ranges from 0000 to 9999).

Because of this, some of the fields within DSPPGM or DSPSRVPGM may not be as expected. The 'Program entry procedure module' shows the *MODULE partition name and not the original *MODULE name. The 'Library' field for that module shows the temporary library name rather than the original library name. In addition, the names of the modules bound into the program or service program will be the generated partition names. For any program or service program that has been optimized by IPA, the 'Program attribute' field displayed by DSPPGM or DSPSRVPGM will be IPA, as will the attribute field of all bound modules for that program or service program.

Note: When IPA is doing partitioning, IPA may prefix the function or data name with @nnn@ or XXXX@nnn@, where XXXX is the partition name, and where nnn is the source file number. This ensures that static function names and static data names remain unique.

Advanced Argument Optimization

Advanced argument optimization is a cross-module optimization that is used to improve the performance of programs that contain frequently run procedure calls, including C++ applications that make mostly nonvirtual method calls. Improved runtime performance is achieved by enabling the translator and binder to use the most efficient mechanisms to pass parameters and return results between procedures that are called within a program or service program.

How to Use Advanced Argument Optimization

The Argument optimization (ARGOPT) parameter, with *YES and *NO as possible values, is available on the CRTPGM and CRTSRVPGM commands to support advanced argument optimization. Specifying ARGOPT(*YES) causes the program or service program to be created with advanced argument optimization. The default is *NO.

Considerations and Restrictions When Using Advanced Argument Optimization

When ARGOPT(*YES) is specified during program creation, advanced argument optimization is applied. In general, this will improve the performance of most procedure calls within the program. However, you need to consider the following items before deciding to use advanced argument optimization:

- Interaction with pragma-based argument optimization:

Argument optimization enabled with ARGOPT(*YES) and argument optimization enabled by the `#pragma argopt` directive, which is supported by the C and C++ compilers, are both redundant and complementary.

If you already have `#pragma argopt` in your code, leave it in there, and also use ARGOPT(*YES). You can remove the redundant `#pragma argopt`'s later, or keep them in there.

If you do not have `#pragma argopt` in your code, using ARGOPT(*YES) will generally help. If you call procedures through a function pointer, you might want to think about using `#pragma argopt` for those cases, as advanced argument optimization does not optimize calls through a function pointer. Virtual function calls in C++ are examples of function pointer calls.

For more information about the `#pragma argopt` directive, see [ILE C/C++ Compiler Reference](#) .

However, unlike the pragma-based argument optimization, which requires manual insertion of `#pragma` directives into the source code, advanced argument optimization requires no source code changes and is applied automatically. In addition, advanced argument optimization can be applied to programs created in any language, while the pragma-based solution is only for C and C++.

While the `#pragma argopt` directive can be applied to function pointers, advanced argument optimization does not automatically optimize virtual function calls and calls through a function pointer. Therefore, for optimizing indirect calls, the `argopt` pragma is useful when used in this complementary way with advanced argument optimization.

- 16-byte pointers:

16-byte space pointer parameters benefit the most from argument optimization. Space pointers point to data object types, such as characters, numbers, classes and data structures. Examples of space pointers in C and C++ include `char*` and `int*`. However, parameters declared with other types of 16-byte pointers that are unique to IBM i, such as pointers that point to system objects, are not optimized by argument optimization. Open pointer parameters such as `void*` pointers in C and C++, which are based on incomplete types, are also not optimized.

- DTAMD(*LLP64):

C and C++ applications that consist of modules created with DTAMD(*LLP64) benefit less from argument optimization than those created with the default DTAMD(*P128). In the former case, pointers to data are 8 bytes long, and these are always passed between procedures using the most

efficient mechanisms. In the latter case, pointers to data are 16 bytes long, which are prime candidates for argument optimization.

- Target release:

Programs created with `ARGOPT(*YES)` must also be created with target release V6R1M0, or later.

To take full advantage of advanced argument optimization, modules bound into programs created with `ARGOPT(*YES)` should be created with target release V6R1M0 or later, as calls into or calls originated from functions defined in modules created before V6R1M0 are ignored by advanced argument optimization.

- Longer program creation time:

When `ARGOPT(*YES)` is specified during program creation, additional analysis is performed across all of the modules in the program. For programs that consist of hundreds or thousands of modules, creation time can be significantly longer.

Similarly, when you use the Update Program (UPDPGM) or Update Service Program (UPDSRVPGM) commands to update a program that is created with `ARGOPT(*YES)`, the amount of time to complete the update can be longer because of the extra analysis that might be required to ensure that all cross-module calls are updated. If there are no changes to procedure interfaces, this extra time is typically short.

- Interaction with special calling conventions:

Advanced argument optimization is not applicable to dynamic program calls. In addition, C and C++ functions that are defined with the `_System` keyword are not candidates for advanced argument optimization.

- Interaction with program profiling:

You can use advanced argument optimization and program profiling at the same time.

- Interaction with interprocedural analysis (IPA):

The cross-module analysis and optimizations performed by IPA are redundant with advanced argument optimization. Therefore, when you use IPA, it is not necessary to use advanced argument optimization.

Licensed Internal Code Options

Licensed Internal Code options (LICOPTs) are compiler options that are passed to the Licensed Internal Code in order to affect how code is generated or packaged. These options affect the code that is generated for a module. You can use some of the options to fine-tune the optimization of your code. Some of the options aid in debugging. This section discusses Licensed Internal Code options.

Currently Defined Options

The Licensed Internal Code options that are currently defined are:

[No]AlwaysTryToFoliate

Instructs the optimizing translator, when compiling at optimization level 40, to more aggressively attempt an optimization known as call foliation, which attempts to reduce the number of stack frames maintained on the runtime call stack. The advantage to this is that, in some cases, fewer stack frames may be required, which can improve locality of reference and, in rare circumstances, reduce the possibility of runtime stack overflow. The disadvantage is that, in the event of program failure, there may be fewer clues that are left behind in the call stack when you debug. This option is off by default.

[No]CallTracingAtHighOpt

Use this option to request that call and return traps be inserted into the procedure prologue and epilogue, respectively, of procedures that require a stack, even at optimization level 40. The advantage of inserting call and return traps is the ability to use job trace, while the disadvantage is potentially worse runtime performance. In releases before 6.1, this option is off by default, meaning that no call and return traps are inserted into any procedures at optimization level 40. Beginning in 6.1, this option is ignored and call and return traps are unconditionally inserted in procedures which require a stack, even at optimization level 40.

[No]Compact

Use this option to reduce code size where possible, at the expense of execution speed. This is done by inhibiting optimizations that replicate or expand code inline. This option is off by default.

CodeGenTarget=

The CodeGenTarget option specifies the creation target model for a program or module object. The creation target model indicates the hardware features that the code generated for that object can use. See the following table for the possible values of this LICOPT.

Value	Meaning
CodeGenTarget=Current	The optimizing translator can use all features available on the current machine.
CodeGenTarget=Common	The optimizing translator can use any feature that is available on every system supported by the target release.
CodeGenTarget=Legacy	The optimizing translator cannot use any features that are only available in the POWER6 level of the PowerPC® AS architecture and later.
CodeGenTarget=Power6	The optimizing translator can use all features available in the POWER6 level of the PowerPC AS architecture.
CodeGenTarget=Power7	The optimizing translator can use all features available in the POWER7 level of the PowerPC AS architecture.
CodeGenTarget=Power8	The optimizing translator can use all features available in the POWER8 level of the PowerPC AS architecture.
CodeGenTarget=Power9	The optimizing translator can use all features available in the POWER9 level of the PowerPC AS architecture.

For more information about this option, refer to [“The CodeGenTarget LICOPT” on page 152.](#)

[No]CreateSuperblocks

This option controls formation of superblocks, which are large extended basic blocks with no control flow entry except in the superblock header. It also controls certain optimizations performed on superblocks, such as trace unrolling and trace peeling. Superblock formation and optimizations can cause large amount of code duplication; this LICOPT can be used to disable these optimizations. This LICOPT is only effective when profile data is applied. This option is on by default.

[No]DetectConvertTo8BytePointerError

This option is ignored on systems running 6.1 or later. Every conversion from a 16-byte pointer to an 8-byte pointer signals an MCH0609 exception if the 16-byte pointer does not contain a teraspace address and does not contain a null pointer value.

[No]EnableInlining

This option controls procedure inlining by the optimizing translator. Procedure inlining means a call to a procedure is replaced by an inline copy of the procedure code. This option is on by default.

[No]FoldFloat

Specifies that the system may evaluate constant floating-point expressions at compile time. This LICOPT overrides the 'Fold float constants' module creation option. When this LICOPT isn't specified, the module creation option is honored.

LoopUnrolling=<option>

Use the LoopUnrolling option to control the amount of loop unrolling performed by the optimizing translator. Valid values are 0 to disable loop unrolling, 1 to unroll small loops with an emphasis on reducing code duplication, and 2 to aggressively unroll loops. Using option 2 may substantially increase the size of generated code. The default value is 1.

[No]Maf

Permit the generation of floating-point multiply-add instructions. These instructions combine a multiply and an add operation without an intermediate rounding operation. Execution performance

is improved, but computational results may be affected. This LICOPT overrides the 'Use multiply add' module creation option. When this LICOPT isn't specified, the module creation option is honored.

[No]MinimizeTeraspaceFalseEAOs

Effective address overflow (EAO) checking is done as part of address arithmetic operations for 16-byte pointers. The same generated code must handle both teraspace and single-level storage (SLS) addresses, so valid teraspace uses can incur false EAOs if the code was generated for a processor before POWER6. See [“Adaptive Code Generation”](#) on page 149 for more information. These EAO conditions do not indicate a problem, but handling them adds significant processing overhead. The `MinimizeTeraspaceFalseEAOs` LICOPT causes differences in the hardware instruction sequences generated for the program. Different address arithmetic instruction sequences are generated that are slightly slower in the usual case but eliminate most EAO occurrences. An example of when this LICOPT should be used is when most address arithmetic performed in a module computes teraspace address results to a lower-valued 16 MB area. This option is off by default.

[No]OrderedPtrComp

Use this option to compare pointers as unsigned integer values, and to always produce an ordered result (equal, less than, or greater than). When you use this option, pointers that refer to different spaces will not compare unordered. This option is off by default.

[No]PredictBranchesInAbsenceOfProfiling

When profile data is not provided, use this option to perform static branch prediction to guide code optimizations. If profile data is provided, the profile data will be used to predict branch probabilities instead, regardless of this option. This option is off by default.

[No]PtrDisjoint

This option enables an aggressive typed-based alias refinement that allows the optimizing translator to eliminate a larger set of redundant loads, which may improve runtime performance. An application can safely use this option if the contents of a pointer are not accessed through a non-pointer type. The following expression in C demonstrates an unsafe way to access the value of a pointer:

```
void* spp;
... = ((long long*) &spp) [1]; // Access low order 8 bytes of 16-byte pointer
```

Default: `NoPtrDisjoint`

[No]ReassocForIndexedLdSt

This option instructs the optimizing translator to reassociate addressing expressions that feed load or store instructions so that they meet the requirements for conversion into indexed load or store instructions. In most cases, this means searching for load or store instructions with a nonzero displacement that might be transformable into indexed loads or store instructions had the displacement been zero. In this case, reassociation generates an explicit addition of the displacement into the address expression, and the displacement in the load or store instructions is zero.

The default of this Licensed Internal Code is `ReassocForIndexedLdSt`.

[No]TailCallOptimizations

This option instructs the optimizing translator, when it compiles at optimization level 40, to perform tail call optimizations that attempt to reduce the number of stack frames that are maintained on the runtime call stack. The advantage to this is that, in some cases, fewer stack frames might be required, which can improve locality of reference and, in rare circumstances, reduce the possibility of runtime stack overflow. The disadvantage is that, if a program fails, fewer clues are left in the call stack that can be used for debugging. It is recommended that this LICOPT remain enabled.

A *tail call* is a call in the procedure that is done immediately before returning. In these cases, the optimizing translator attempts to perform tail call optimizations that eliminate the need for the caller to allocate a stack frame and leave the stack unaltered. Normally, a stack frame is created by the caller to allow the callee to save and restore the return address. With these optimizations, the tail call is changed to a simple branch operation that does not calculate a new return address. The original return address is left intact and therefore points back to the return location of the caller. The optimizations eliminate the intermediate stack frame between the caller and callee, and the callee returns to the caller's caller.

For example, if Function A calls Function B and Function B does a tail call to Function C, with tail call optimizations enabled, Function B does not allocate a stack frame and instead branches to Function C. When Function C is complete, it returns directly back to Function A following the call to Function B. Without tail call optimizations, Function C returns to Function B, which immediately returns back to Function A.

For frequently called procedures, tail call optimizations can improve performance by eliminating the operations needed to create and destroy an unnecessary stack frame. If you specify the LICOPT value NoTailCallOptimizations, these optimizations are not attempted.

The default of this LICOPT is TailCallOptimizations.

TargetProcessorModel=<option>

The TargetProcessorModel option instructs the translator to perform optimizations for the specified processor model. Programs created with this option run on all supported hardware models, but generally run faster on the specified processor model. See the following table for the TargetProcessorModel values, associated processor models, and target release-specific defaults for the option.

Target release of program	TargetProcessorModel value	Specified processor model
IBM i 7.3 and IBM i 7.2	6	POWER8
IBM i 7.1	5	POWER7
IBM i 6.1	4	POWER6
V5R4 and earlier	3	POWER5

The TargetProcessorModel LICOPT is one of several factors determining which processor model should be targeted when creating a module, changing a module or bound module, or re-creating a module or bound module. The following rules apply to both modules and bound modules.

- When a module is created or changed, and the TargetProcessorModel LICOPT is specified, the generated code for the module is tuned for best performance on the specified processor model, regardless of whether the CodeGenTarget LICOPT is also specified.
- When a module is created or changed, and the TargetProcessorModel LICOPT is not specified, but the CodeGenTarget LICOPT is specified to be POWER6, POWER7, POWER8, or POWER9, the generated code for the module is tuned for best performance on the specified processor model.
- When a module is created or changed, and the TargetProcessorModel LICOPT is not specified, but the CodeGenTarget LICOPT is specified to be Current, the generated code for the module is tuned for best performance on the processor in use by the partition on which the module resides.
- When a module is created or changed, and the TargetProcessorModel LICOPT is not specified, but the CodeGenTarget LICOPT is specified to be Common, then the module is tuned for best performance on a default processor model for the release. The default processor model is POWER8 for IBM i 7.3 and IBM i 7.2, POWER7 for IBM i 7.1, POWER6 for IBM i 6.1, and POWER5 for earlier supported releases.
- When a module is created or changed, and the TargetProcessorModel LICOPT is not specified, but the CodeGenTarget LICOPT is specified to be Legacy, then the module is tuned for best performance on the POWER5 processor model.
- When a module is re-created, the module is tuned for best performance on the processor in use by the partition on which the module resides, regardless of the values of any TargetProcessorModel and CodeGenTarget LICOPTs specified when the module was created or last changed.

Note that for most of these options there is a positive and a negative variation, the negative beginning with the prefix 'no'. The negative variant means that the option is not to be applied. There will always be two variants like this for Boolean options, in order to allow a user to explicitly turn off an option as well as

turn it on. The capability to do so is necessary to turn off an option for which the default option is on. The default for any option may change from release to release.

Application

You can specify Licensed Internal Code options (LICOPTs) when you create modules, and you can modify the options of an existing object with the Change Module (CHGMOD), Change Program (CHGPGM), and Change Service Program (CHGSRVPGM) commands. With these commands, you can replace the entire LICOPT string or add LICOPTs to an existing string. Adding LICOPTs to an existing string is useful when the objects being changed have different LICOPTs and you do not want to lose the existing LICOPTs.

An example of applying Licensed Internal Code options to a module is:

```
> CHGMOD MODULE(TEST) LICOPT('maf')
```

When used on CHGPGM or CHGSRVPGM, the system applies the specified Licensed Internal Code options to all modules that are contained in the ILE program object. An example of applying Licensed Internal Code options to an ILE program object is:

```
> CHGPGM PGM(TEST) LICOPT('nomaf')
```

An example of applying Licensed Internal Code options to a service program is:

```
> CHGSRVPGM SRVPGM(TEST) LICOPT('maf')
```

To add a LICOPT to an existing object, use the *ADD keyword on the LICOPT parameter. This example preserves the object's existing LICOPTs and adds a new one: LICOPT ('maf' , *ADD).

Restrictions

Several restrictions exist on the types of programs and modules to which you can apply Licensed Internal Code options.

- You cannot apply Licensed Internal Code options to OPM programs.
- The module or ILE program or service program object must have been originally created for release V4R5M0 or later.
- You cannot apply Licensed Internal Code options to pre-V4R5 bound modules within a V4R5 or later program or service program. This does not affect other bound modules within the program which can have LICOPTs applied.

Syntax

On the CHGMOD, CHGPGM, and CHGSRVPGM commands, the case of the LICOPT parameter value is not significant. For example, the following two command invocations would have the same effect:

```
> CHGMOD MODULE(TEST) LICOPT('nomaf')
> CHGMOD MODULE(TEST) LICOPT('NoMaf')
```

When specifying several Licensed Internal Code options together, you must separate the options by commas. Also, the system ignores all spaces that precede or that follow any option. Here are some examples:

```
> CHGMOD MODULE(TEST) LICOPT('Maf,NoFoldFloat')
> CHGMOD MODULE(TEST) LICOPT('Maf, NoFoldFloat')
> CHGMOD MODULE(TEST) LICOPT(' Maf , NoFoldFloat  ')
```

For Boolean options, the system does not allow specifying of the two opposite variants at the same time. For example, the system will not allow the following command:

```
> CHGMOD MODULE(TEST) LICOPT('Maf,NoMaf')    <- NOT ALLOWED!
```

However, you can specify the same option more than once. For example, this is valid:

```
> CHGMOD MODULE(TEST) LICOPT('Maf, NoFoldFloat, Maf')
```

Release Compatibility

The system will not allow users to move a module, program, or service program that has had Licensed Internal Code options applied to any release before V4R5M0. In fact, the system prevents the user from specifying an earlier target release when attempting to save the object to media or to a save file.

IBM i sometimes defines new Licensed Internal Code options in new releases (or within a given release through a PTF). You can use the new options on systems that have the first release that supports them, or any later release. You can move any module, program, or service program that has new options applied, to a release that does not support the options. The system ignores and no longer applies unsupported Licensed Internal Code options for converted objects if the LICOPT parameter of a command does not specify the options. Unsupported LICOPT values are ignored if re-creation occurs when you use LICOPT(*SAME) on the CHGMOD, CHGPGM, or CHGSRVPGM commands. They are also ignored when re-creation occurs when the system automatically converts the object. In contrast, any attempt to specify unsupported options in the LICOPT parameter of the CHGMOD, CHGPGM, or CHGSRVPGM commands will fail.

Displaying Module and ILE Program Licensed Internal Code Options

The DSPMOD, DSPPGM, and DSPSRVPGM CL commands and the QSYS2.BOUND_MODULE_INFO view display the Licensed Internal Code options that were applied.

DSPMOD displays them in the Module Information section. For example:

```
Licensed Internal Code options . . . . . : maf
```

DSPPGM and DSPSRVPGM display the Licensed Internal Code options that are applied to each individual module within the program in the Module Attributes section for each module.

By specifying the same Licensed Internal Code option more than once, all occurrences of that option except for the last one appear preceded by a '+' symbol. For example, assume that the command used to apply Licensed Internal Code options to a module object is as stated below:

```
> CHGMOD MODULE(TEST) LICOPT('maf, maf, Maf')
```

Then DSPMOD will show this:

```
Licensed Internal Code options . . . . . : +maf,+maf,Maf
```

The '+' means that the user specified redundant occurrences of the same option.

If any Licensed Internal Code options appear preceded by a '*' symbol, they no longer apply to a Module or ILE Program. This is because the system that performed the last recreation of the object did not support them. For more information, please see the [“Release Compatibility” on page 148](#) section. For example, assume that the new option was originally applied on a release N+1 system by using the following command:

```
> CHGMOD MODULE(TEST) LICOPT('NewOption')
```

The Module is taken back to a release N system that does not support that option, and then the Module object is recreated there using:

```
> CHGMOD MODULE(TEST) FRCCRT(*YES) LICOPT(*SAME)
```

The Licensed Internal Code options shown on DSPMOD will look like this:

```
Licensed Internal Code options . . . . . : *NewOption
```

The '*' means that the option no longer applies to the Module.

The `BOUND_MODULE_INFO` view can be used to see the Licensed Internal Code options for each module as well.

```
SELECT BOUND_MODULE_LIBRARY, BOUND_MODULE, LIC_OPTIONS, PROGRAM_LIBRARY, PROGRAM_NAME,
OBJECT_TYPE
FROM QSYS2.BOUND_MODULE_INFO
WHERE PROGRAM_LIBRARY = 'MYLIB';
```

Adaptive Code Generation

Generally, you do not need to understand the details of the underlying hardware architecture so that the architecture can change smoothly over time. Architecture changes range from adding a single processor instruction to changing the entire processor instruction set. To ensure that your programs continue to run correctly when the operating system moves between platforms that have different levels of underlying hardware, an abstract machine interface (MI) that represents programs in a hardware-independent format is used.

The optimizing translator is responsible for generating hardware instructions from the MI representation. Because the optimizing translator is a component of the operating system, there is a single version of the optimizing translator for each release. However, a given release might be supported on a number of different system models, each of which might have slightly different processor hardware from the others.

In releases before 6.1, the optimizing translator for a given release was designed to generate only instructions that would run on all system models supported by that release. The advantage of this policy is that a program compiled for a particular release can run unchanged on any system running the same release. This makes it easy to build and distribute software for each release. However, important new processor features, which often provide significant performance advantages, cannot be used until they are present on all systems supported by the current release. A gap of a number of years might occur between the availability of a processor feature and its use in your programs.

As of 6.1, you can take advantage of all processor features on your systems, regardless of whether those features are present on other system models supported by the same release. Furthermore, programs can be moved from one system model to another and continue to run correctly, even if the new machine does not have all the processor features available on the original one. The technology used to achieve this is called adaptive code generation. Adaptive code generation (ACG) can work without user intervention for most scenarios. However, if you build and distribute software to run on a variety of system models, you might want to exercise some control over which processor features are used by adaptive code generation.

ACG Concepts

To understand how adaptive code generation (ACG) works, it is helpful to understand the following concepts.

A *hardware feature* is a feature that has been added to the family of processors supported by IBM i. For instance, one new feature available in some processors supported by 6.1, is a new hardware decimal floating-point unit. A processor with this unit is considered by ACG to have the decimal floating-point feature, while a processor without it does not have the decimal floating-point feature. The aggregation of all added features present in a processor is called that processor's feature set.

A *target model* is an abstraction of all processors that have the same feature set. An example of a target model would be all processors that conform to the POWER6 level of the PowerPC AS architecture.

A module or program object also has a feature set that identifies the features required by the object so that the object runs correctly without change. An object is compatible with a target model if all features in the object's feature set are also present in the target model's feature set.

Code generation refers to the process of creating hardware instructions for your module and program objects. Code generation is performed by the optimizing translator.

A module or program object can be moved from one system to another. The system on which an object resides is called the *current machine*.

Normal Operation

When you compile a module object using a command such as Create C Module (CRTCMOD), the optimizing translator automatically detects which processor features are available on your system. The hardware instructions generated for your module object take advantage of any optional processor features that might be useful. The optimizing translator stores the feature set used in the module object as part of the object.

When you create a program object using a command such as Create Program (CRTPGM) or Create Service Program (CRTSRVPGM), the binder determines the feature set for the program object and stores it as part of the program object. A feature is included in the program's feature set if it is present in any of the feature sets of the program's modules.

The first time the program object is activated on your system, the system checks to be sure that the program object is compatible with the target model associated with your system; that is, it ensures that your program does not use any features that are not available on your system. Because you compiled the program object on this system, the program object always passes this compatibility check, and the program runs correctly.

Suppose you want to migrate this program object to another system that uses the same release, but has a different target model. The first time the program is activated on the system to which it is moved, the system performs the compatibility check against this system's target model. If the program is compatible with the system, the program runs correctly. However, if the program requires any processor feature that is not supported by the system to which it was moved, then the system automatically calls the optimizing translator to convert the program to be compatible. The translator detects which features are available on the new system, and takes advantage of all those that are applicable, just as it did when the original module objects were created. The converted program is then activated as requested.

Restore Options

To change the behavior of adaptive code generation when module and program objects are restored onto your system, set the Force Conversion on Restore (QFRCCVNRST) system value and Force Object Conversion (FRCOBJCVN) command parameter. The FRCOBJCVN parameter is present on the Restore (RST), Restore Object (RSTOBJ), and Restore Library (RSTLIB) commands.

QFRCCVNRST System Value

The Force Conversion on Restore (QFRCCVNRST) system value has the following possible values.

Value	Meaning
0	Do not convert anything.
1	Objects with validation errors are converted. (This is the default.)
2	Objects that require conversion on the current version of the operating system, or on the current machine, and objects with validation errors are converted.
3	Objects that are suspected of having been tampered with, objects that contain validation errors, and objects that require conversion on the current version of the operating system or on the current machine are converted.
4	Objects that contain sufficient creation data to be converted and do not have valid digital signatures are converted.
5	Objects that contain sufficient creation data are converted.
6	All objects that do not have valid digital signatures are converted.
7	All objects are converted.

By setting QFRCCVNRST to 2 or 3, you can ensure that any incompatible program and module objects are converted immediately during Restore command (RST, RSTOBJ, RSTLIB) processing, rather than on the

first activation. This might be preferable in some cases, because the time to convert a program can be long, particularly for large programs and programs that are compiled at high optimization levels. If you commonly restore programs onto a system with which they might be incompatible, consider changing this system value.

FRCOBJCVN Parameter

The Force Object Conversion (FRCOBJCVN) parameter on the Restore (RST), Restore Object (RSTOBJ), and Restore Library (RSTLIB) commands can be used to exercise control over adaptive code generation. The following table shows the possible values for these commands. The RSTOBJ command is used as an example.

Value	Meaning
RSTOBJ FRCOBJCVN(*SYSVAL)	The objects are converted based on the value of the Force Conversion on Restore (QFRCCVNRST) system value. (This is the default.)
RSTOBJ FRCOBJCVN(*NO)	The objects are not converted during the restore operation.
RSTOBJ FRCOBJCVN(*YES *RQD)	The objects are converted during the restore operation only if they require conversion to be used by the current operating system or require conversion to be compatible with the current machine.
RSTOBJ FRCOBJCVN(*YES *ALL)	All objects are converted regardless of their current format and machine compatibility, including compatible objects in the current format.

The option RSTOBJ FRCOBJCVN(*YES *RQD) causes any incompatible module and program objects that are restored to be immediately converted, rather than subsequently retranslated during the first activation. You should consider using this option if you do not want to change the QFRCCVNRST system value, but want to ensure that any incompatible objects that are restored by this command are converted immediately.

Create Options

When you create module and program objects, the default behavior of using all features available on the current machine is usually desirable. However, if you are building software for distribution onto a variety of systems, you might want to be more conservative in specifying which features should be used in your programs. For example, if you sell a software package that runs on 6.1, you might not want your programs and service programs to be converted on your customers' machines. This could happen if the system you build on contains more features than those of some of your customers. Conversion on customer machines might be undesirable because of the following reasons:

- Customers might prefer not to incur the time spent converting the objects (during the restoration or during the first activation)
- Converted programs contain different hardware instructions than your original programs do. Although the program functionality remains the same, this might affect your customer support processes.

To avoid conversion, you can specify that your programs be built only with features common to all systems supported by the target release. You can achieve this in one of the following ways:

- Specifying the CodeGenTarget Licensed Internal Code option (LICOPT) when creating your module and program objects.
- Specifying the CodeGenTarget LICOPT on a Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) command.
- Setting an environment variable to control which features are used.

The CodeGenTarget LICOPT

You can specify the Licensed Internal Code options (LICOPT) parameter on many of the module and program creation commands, such as Create C Module (CRTCMOD) and Create Bound C Program (CRTBNDC). The LICOPT parameter directs the optimizing translator to use or avoid certain options when it creates the hardware instructions for objects. You can use the LICOPT CodeGenTarget option to provide control over features that are selected by adaptive code generation. Refer to [CodeGenTarget](#) for all the possible values of this option.

By selecting the CodeGenTarget=Common option when creating your module and program objects, you can ensure that your software product does not require conversion when it is restored onto customer machines, or when it is first used by your customers. However, you might be giving up some potential performance improvements on machines with more hardware features.

After creating a program object, you might realize that you should have specified which features are allowed to be used in the object. Instead of re-creating the program, you can use the Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) command and specify the CodeGenTarget LICOPT parameter on that command. The program is converted, and the optimizing translator uses only those features that you specified.

New features are available for the POWER6, POWER7 and POWER8 levels of the PowerPC AS architecture. The following CodeGenTarget LICOPT values can be used to control which features are selected by adaptive code generation.

To select POWER8 features (POWER8 includes all features of POWER7 and POWER6), do one of the following:

- Specify CodeGenTarget=POWER8
- If running on POWER8 hardware, specify CodeGenTarget=Current

To select POWER7 features (POWER7 includes all features of POWER6 but does not include any POWER8 features), do one of the following:

- Specify CodeGenTarget=POWER7
- If running on POWER7 hardware, specify CodeGenTarget=Current
- If creating for IBM i 7.3, specify CodeGenTarget=Common

To select POWER6 features, but not POWER7 or POWER8 features, do one of the following:

- Specify CodeGenTarget=POWER6
- If running on POWER6 hardware, specify CodeGenTarget=Current
- If creating for IBM i 7.2, specify CodeGenTarget=Common

To prevent the use of POWER6, POWER7, and POWER8 features, do one of the following:

- Specify CodeGenTarget=Legacy
- If creating for IBM i 6.1 or IBM i 7.1, specify CodeGenTarget=Common

Features associated with POWER6 hardware include:

- A hardware decimal floating-point unit
- Efficient hardware support for ILE pointer handling

Features associated with POWER7 hardware include:

- A number of new instructions that may speed up certain computations, such as conversions between integer and floating-point values.

Features associated with POWER8 hardware include:

- New instructions that may speed up floating point operations.

For a module object, use the Display Module (DSPMOD) command with DETAIL(*BASIC) to see the LICOPT options that are applied to that module. For a program or service program object, LICOPT options

are associated with each module in the program. Use the Display Program (DSPPGM) or Display Service Program (DSPSRVPGM) command with DETAIL(*MODULE), and then specify option 5 for the modules you want to view. The value of Licensed Internal Code options on one of these display screens might contain CodeGenTarget=model for some models. This indicates that the LICOPT is used to override the default behavior when the module was created, either by directly specifying the LICOPT, or by setting the QIBM_BN_CREATE_WITH_COMMON_CODEGEN environment variable. If such a LICOPT is not present, then the default behavior was not overridden.

For more information about how to display Licensed Internal Code options, refer to [“Displaying Module and ILE Program Licensed Internal Code Options”](#) on page 148.

QIBM_BN_CREATE_WITH_COMMON_CODEGEN Environment Variable

For large builds, it might not be convenient to specify the CodeGenTarget LICOPT for all module and program creation commands. In addition, the command you use to create objects might not support the LICOPT parameter. In these situations, you can use the QIBM_BN_CREATE_WITH_COMMON_CODEGEN environment variable to set the adaptive code generation behavior for your builds.

To work with environment variables, use the Work with Environment Variable (WRKENVVAR) command.

The possible values for the QIBM_BN_CREATE_WITH_COMMON_CODEGEN environment variable are in the following table.

Value	Meaning
0	When you create new modules, use LICOPT CodeGenTarget as specified. If CodeGenTarget is not specified, use CodeGenTarget=Current. (This is the default.)
1	When you create new modules, use LICOPT CodeGenTarget as specified. If CodeGenTarget is not specified, use CodeGenTarget=Common.
2	When you create new modules, always use CodeGenTarget=Common, overriding any LICOPT CodeGenTarget that might have been supplied.

- Value 0 denotes the default behavior.
- Value 1 is useful when you want to use the common subset of features on most parts, but would like to override this for individual parts.
- Value 2 is useful when you need to ensure that all parts are built with the common subset of features.

Note: This environment variable applies only when you create new modules. It does not apply on change commands, such as Change Program (CHGPGM). Nor does it apply during object conversion.

Displaying ACG Information

You can use the Display Module (DSPMOD), Display Program (DSPPGM), and Display Service Program (DSPSRVPGM) commands to determine whether your module and program objects are compatible with the current machine, and whether the CodeGenTarget LICOPT is used to override default ACG behavior.

Object compatibility

Use the Display Program (DSPPGM), Display Service Program (DSPSRVPGM), or Display Module (DSPMOD) command with DETAIL(*BASIC) to determine whether your module or program object is able to run correctly without first being converted. The value of the Conversion required field is either *YES or *NO. If the value of Conversion required is *YES for a program object, the program is converted the first time it is activated. Alternately, you can use CHGPGM FRCCRT(*YES) or Start Object Conversion (STROBJCVN) to force conversion to occur at a more convenient time. See [“Optimizing Compatible Programs”](#) on page 154 for more information about when these two commands can be used.

If the value of the Conversion required field is *YES for a module object, and the module object is not in an older format (see *FORMAT in the following table), binding this module object into a program or service program results in a program object that also requires conversion. This outcome is useful when you are

building programs for deployment to a system that has more features than the build system. If the value of the Conversion required field is *NO, then your program or module object is ready for immediate use.

The reason that a module or program object requires conversion can be determined by viewing the Conversion details field. This field has one of the following values.

Value	Meaning
*FORMAT	The object is not compatible with the current machine. The object is in an older format. (For example, objects created for releases earlier than 6.1 are in a different format from objects created for 6.1.) Binding a module object in an older format causes the bound module to be converted.
*FEATURE	The object is not compatible with the current machine. The object format is compatible with the current machine, but the object uses at least one feature that is not supported by the current machine.
*COMPAT	The object is compatible with the current machine. The object format is compatible with the current machine, and all features that the object uses are implemented by the current machine. However, the object uses at least one feature that is not supported by the common level of hardware supported by version, release, and modification level that the object was created for.
*COMMON	The object is compatible with the current machine. The object format is compatible, and all features that the object uses are implemented by the common level of hardware supported by the version, release, and modification level that the object was created for.

Release-to-Release Considerations

The behavior of adaptive code generation is determined by the target version, release, and modification level you choose when you create a module or program object. ACG is only supported on systems running 6.1, or later. If the CodeGenTarget LICOPT is specified for an object with an earlier target release, the LICOPT is tolerated, but has no effect on the code generated for the object.

The target version, release, and modification level also governs the meaning of LICOPT('CodeGenTarget=Common'). When this LICOPT is selected, it means that the optimizing translator should use the set of features available on all machines supported by the target release, which might differ from the release on which the program or module object was created. Therefore, you can use a build machine running a more recent version of the operating system to create common code that runs on every machine running an earlier version of the operation system, provided that this earlier version is no earlier than 6.1.

Note: You cannot use the optimizing translator that is on an earlier version to take advantage of all features common to machines running on a more recent version.

Optimizing Compatible Programs

When compatible objects are restored onto a machine, they are not normally converted, unless you have specified the FRCOBJCVN(*YES *ALL) parameter on the Restore command. This means that the objects might not take full advantage of the machine's capabilities. Some features might not have been used by the optimizing translator. You might want to update your module and program objects to take full advantage of your machine.

You need to know whether your module and program objects have full creation data available. Use the Display Module (DSPMOD), Display Program (DSPPGM), or Display Service Program (DSPSRVPGM) command with DETAIL(*BASIC) to see this.

- If all creation data is available and observable, the All creation data field is *YES. You can use the Change Module (CHGMOD), Change Program (CHGPGM), and Change Service Program (CHGSRVPGM) commands with FRCRT(*YES) to force re-creation of the objects.

- If all creation data is available but not all of it is observable, the All creation data field is *UNOBS. You can force a conversion of these objects during a restore operation by specifying the FRCOBJCNV(*YES *ALL) parameter on the Restore command.
- If not all creation data is available, the All creation data field is *NO. You can do nothing to change the objects. If your programs lack creation data in this way, they must be recreated from source to run on 6.1 and later releases.

ACG and Logical Partitions

If you have a system with several logical partitions, you can configure a partition to imitate a system with a different processor than the one you have physically installed. For the purposes of adaptive code generation, such a partition is treated as if it is running on the imitated processor. Any features in a program that are not available on the imitated processor force the program to require conversion to run on that processor, even if those features are available on the underlying physical processor.

Shared Storage Synchronization

Shared storage provides an efficient means for communication between two or more concurrently running threads. This topic discusses a number of issues that are related to shared storage. The primary focus is on data synchronization problems that can arise when accessing shared storage, and how to overcome them.

Although not unique to ILE, the programming problems associated with shared storage are more likely to occur in ILE languages than in original MI languages. This is due to the broader support for multiprogramming Application Programming Interfaces in ILE.

Shared Storage

The term *shared storage*, as it pertains to this discussion, refers to any space data that is accessed from two or more threads. This definition includes any storage directly accessible down to its individual bytes, and can include the following classes of storage:

- MI space objects
- Primary associated spaces of other MI objects
- POSIX shared memory segments
- Implicit process spaces: Automatic storage, static storage, and activation-based heap storage
- Teraspace

The system considers these spaces, regardless of the longevity of their existence, as shared storage when accessed by multiple threads capable of concurrent processing.

Shared Storage Pitfalls

When creating applications that take advantage of shared storage, you need to avoid two types of problems that can result in unpredictable data values: *race conditions* and *storage access ordering problems*.

- A race condition exists when different program results are possible due solely to the relative timing of two or more cooperating threads.

You can avoid race conditions by synchronizing the processing of the competing threads so that they interact in a predictable, well-behaved manner. Although the focus of this document is on storage synchronization, the techniques for synchronizing thread execution and synchronizing storage overlap to a great extent. Because of this, the example problems discussed later in this topic briefly touch on race conditions.

- Storage access ordering problems are also known as storage synchronization or memory consistency problems. These problems result when two or more cooperating threads rely on a specific ordering of updates to shared storage, and their respective accesses to the storage accesses are not synchronized. For example, one thread might store values to two shared variables, and another thread has an implicit dependency on observing those updates in a certain order.

You can avoid shared storage access ordering problems by ensuring that the system performs storage synchronization actions for the threads that read from and write to shared storage. Some of these actions are described in the following topics.

Shared Storage Access Ordering

When threads share storage, no guarantee exists that shared storage accesses (reads and writes) performed by one thread will be observed in that particular order by other threads. You can prevent

this by having some form of explicit storage synchronization performed by the threads that are reading or writing to the shared storage.

Storage synchronization is required when two or more threads attempt concurrent access to shared storage, and the semantics of the threads' logic requires some ordering on the shared storage accesses. When the order in which shared storage updates are observed is not important, no storage synchronization is necessary. A given thread will always observe its own storage updates (to shared or non-shared storage) in order. All threads accessing *overlapping* shared storage locations will observe those accesses in the same order.

Consider the following simple example, which illustrates how both race conditions and storage access ordering problems can lead to unpredictable results.

```

volatile int X = 0;
volatile int Y = 0;

Thread A
-----
Y = 1;
X = 1;

Thread B
-----
print(X);
print(Y);

```

The table below summarizes the possible results that are printed by B.

X	Y	Type of Problem	Explanation
0	0	Race Condition	Thread B read the variables before the modifications of Thread A.
0	1	Race Condition	Thread B observed the update to Y but printed X before observing Thread A's update.
1	1	Race Condition	Thread B read both variables after the updates of Thread A.
1	0	Storage Access Ordering	Thread B observed the update to X but had yet to see Thread A's update to Y. With no explicit data synchronizing actions, this type of out-of-sequence storage access can occur.

Example Problem 1: One Writer, Many Readers

Usually, the potential for out-of-order shared storage accesses does not affect the correctness of multi-threaded program logic. However, in some cases, the order in which threads see the storage updates of other threads is vital to the correctness of the program.

Consider a typical scenario that requires some form of explicit data synchronization. This is when the state of one shared storage location is used (by convention in a program's logic) to control access to a second (non-overlapping) shared storage location. For example, assume that one thread initializes some shared data (DATA). Furthermore, assume that the thread then sets a shared flag (FLAG) to indicate to all other threads that the shared data is initialized.

```

Initializing Thread
-----
DATA = 10
FLAG = 1

All Other Threads
-----
loop until FLAG has value 1
use DATA

```

In this case, the sharing threads must enforce an order on the shared storage accesses. Otherwise, other threads might view the initializing thread's shared storage updates out of order. This could allow some or all of the other threads to read an uninitialized value from DATA.

Example 1 Solution

A preferred method for solving the problem in the example above is to avoid the dependency between the data and flag values. You can do this by using a more robust thread synchronization scheme. Although you

can employ many of the thread synchronization techniques, one that lends itself well to this problem is a semaphore.

In order for the following logic to be appropriate, you must assume the following:

- The program created the semaphore before starting the cooperating threads.
- The program initialized the semaphore to a count of 1.

Initializing Thread	All Other Threads
----- DATA = 10 Decrement semaphore	----- Wait for semaphore count to reach 0 use DATA

Storage Synchronizing Actions

When an ordering of shared storage accesses is required, all threads that require enforcement of an ordering must take explicit action to synchronize the shared storage accesses. These actions are called *storage synchronizing actions*.

A synchronizing action taken by a thread ensures that shared storage accesses that appear prior to the synchronizing action in the thread's logical flow complete before those accesses that appear in the logical flow of the code after the synchronizing action. This is from the viewpoint of other threads at their synchronizing actions. In other words, if a thread performs two writes to two shared locations and a synchronizing action separates those writes, the system accomplishes the following: The first write is guaranteed to be available to other threads at or before their next synchronizing actions, and no later than the point at which the second write becomes available.

When two reads from two shared locations are separated by a storage synchronizing action, the second read reads a value no less current than the first read. This is only true when other threads enforce an ordering when writing to the shared storage.

The following thread synchronization actions are also storage synchronization actions:

Mechanism	Synchronizing Action	First Available in VRM
Object Locks	Lock, Unlock	All
Space Location Locks	Lock, Unlock	All
Mutex	Lock, Unlock	V3R1M0
Semaphores	Post, Wait	V3R2M0
Pthread Conditions	Wait, Signal, Broadcast	V4R2M0
Data Queues	Enqueue, Dequeue	All
User Queues	Enqueue, Dequeue	All
Message Queues	Enqueue, Dequeue	V3R2M0
Compare-and-Swap	Successful store to target	V3R1M0
Check Lock Value (CHKLKVAL)	Successful store to target	V5R3M0
Clear Lock Value (CLRLKVAL)	Always	V5R3M0

The following MI instructions each constitute a storage synchronization action:

Mechanism	Synchronizing Action	First Available in VRM
SYNCADDF MI Instruction	Always	V7R5M0
SYNCSUBF MI Instruction	Always	V7R5M0
SYNCANDF MI Instruction	Always	V7R5M0

Mechanism	Synchronizing Action	First Available in VRM
SYNCORF MI Instruction	Always	V7R5M0
SYNCXORF MI Instruction	Always	V7R5M0
SYNCNANDF MI Instruction	Always	V7R5M0
SYNCFADD MI Instruction	Always	V7R5M0
SYNCFSUB MI Instruction	Always	V7R5M0
SYNCFAND MI Instruction	Always	V7R5M0
SYNCFOR MI Instruction	Always	V7R5M0
SYNCFXOR MI Instruction	Always	V7R5M0
SYNCFNAND MI Instruction	Always	V7R5M0
SYNCCMPSW MI Instruction	Always	V7R5M0
SYNCBCMPSW MI Instruction	Always	V7R5M0
SYNCSTO MI Instruction	Always	V7R5M0

Additionally, the following MI instruction constitutes a storage synchronization action, but is not usable for synchronizing threads:

Mechanism	Synchronizing Action	First Available in VRM
SYNCSTG MI Instruction	Always	V4R5M0

Remember: To completely enforce shared storage access ordering between two or more threads, all threads that are dependent on the access ordering must use the appropriate synchronizing actions. This is true for both readers and writers of the shared data. This agreement between readers and writers ensures that the order of accesses will remain unchanged by any optimizations that are employed by the underlying machine.

Example Problem 2: Two Contending Writers or Readers

Another common problem requiring additional synchronization is one in which two or more threads attempt to enforce an informal locking protocol, as in the example below. In this example, two threads manipulate data in shared storage. Both threads repeatedly attempt to read and write two shared data items, using a shared flag in an attempt to serialize accesses.

Thread A	Thread B
<pre> ----- /* Do some work on the shared data */ for (int i=0; i<10; ++i) { /* Wait until the locked flag is clear */ while (locked == 1) { sleep(1); } locked = 1; /* Set the lock */ /* Update the shared data */ data1 += 5; data2 += 10; locked = 0; /* Clear the lock */ } </pre>	<pre> ----- /* Do some work on the shared data */ for (int i=0; i<10; ++i) { /* Wait until the locked flag is clear while (locked == 1) { sleep(1); } locked = 1; /* Set the lock */ /* Update the shared data */ data1 += 4; data2 += 6; locked = 0; /* Clear the lock */ } </pre>

This example illustrates both of our shared memory pitfalls.

Race Conditions

The locking protocol used here has not circumvented the data race conditions. Both jobs could simultaneously see that the locked flag is clear, and thus both fall into the logic which updates the data. At that point, there is no guarantee of which data values will be read, incremented, and written — allowing many possible outcomes.

Storage Access Ordering Concerns

Ignore, for a moment, the race condition mentioned above. Notice that the logic used by both jobs to update the lock and the shared data contains assumptions about the implicit ordering of the field updates. Specifically, there is an assumption on the part of each thread that the other thread will observe that the locked flag has been set to 1 prior to observing changes to the data. Additionally, it is assumed that each thread will observe the changing of the data prior to observing the locked flag value of zero. As noted earlier in this discussion, these assumptions are not valid.

Example 2 Solution

To avoid the race condition, and to enforce storage ordering, you should serialize accesses to the shared data by one of the synchronization mechanisms that is enumerated above. This example, where multiple threads are competing for a shared resource, lends itself well to some form of lock. A solution employing a space location lock will be discussed, followed by an alternative solution using the Check Lock Value and Clear Lock Value.

THREAD A	THREAD B
<pre>----- for (i=0; i<10; ++i) { /* Get an exclusive lock on the shared data. We go into a wait state until the lock is granted. */ locks1(LOCK_LOC, _LENR_LOCK); /* Update the shared data */ data1 += 5; data2 += 10; /* Unlock the shared data */ unlocks1(LOCK_LOC, _LENR_LOCK); }</pre>	<pre>----- for (i=0; i<10; ++i) { /* Get an exclusive lock on the shared data. We go into a wait state until the lock is granted. */ locks1(LOCK_LOC, _LENR_LOCK); /* Update the shared data */ data1 += 4; data2 += 6; /* Unlock the shared data */ unlocks1(LOCK_LOC, _LENR_LOCK); }</pre>

Restricting access to the shared data with a lock guarantees that only one thread will be able to access the data at a time. This solves the race condition. This solution also solves the storage access ordering concerns, since there is no longer an ordering dependency between two shared storage locations.

Alternate Solution: Using Check Lock Value / Clear Lock Value

Space location locks, like those used in the first solution, are full of features that are not required in this simple example. For instance, space location locks support a time-out value which would allow processing to resume if unable to acquire the lock within some period of time. Space locations locks also support several combinations of shared locks. These are important features, but come at the price of some performance overhead.

An alternative is to use *Check Lock Value* and *Clear Lock Value*. Together, these two MI instructions provide a way to implement a very simple and fast locking protocol, particularly if there is not much contention on the lock.

In this solution, the system uses CHKLKVAL to attempt to acquire the lock. If that attempt fails (because the system found the lock already in use), the thread will wait for a while and then try again, repeating until the lock is acquired. After updating the shared data, the system will use CLRLKVAL to release the lock. In this example, assume that, in addition to the shared data items, the threads also share the address of an 8-byte location. This code refers to that location as variable LOCK. Furthermore, assume that the lock was initialized to zero, either through static initialization or some prior synchronized initialization.

THREAD A	THREAD B
<pre>----- /* Do some work on the shared data */</pre>	<pre>----- /* Do some work on the shared data</pre>

```

*/
for (i=0; i<10; ++i) {
    /* Attempt to acquire the lock using
    using CHKLKVAL. By convention, use value
    value 1 to indicate locked, 0 to indicate
    indicate unlocked. */
    while (_CHKLKVAL(&LOCK, 0, 1) == 1) {
1) {
        sleep(1); /* wait a bit and try again */
    try again */
    }

    /* Update the shared data */
    data1 += 5;
    data2 += 10;

    /* Unlock the shared data. Use of
    of CLRLKVAL ensures other jobs/threads
    threads see update to shared data prior to
    prior to release of the lock. */
    _CLRLKVAL(&LOCK, 0);
}

```

```

for (i=0; i<10; ++i) {
    /* Attempt to acquire the lock
    CHKLKVAL. By convention, use
    1 to indicate locked, 0 to
    unlocked. */
    while (_CHKLKVAL(&LOCK, 0, 1) ==
    sleep(1); /* wait a bit and
    }

    /* Update the shared data */
    data1 += 4;
    data2 += 6;

    /* Unlock the shared data. Use
    CLRLKVAL ensures other jobs/
    see update to shared data
    release of the lock. */
    _CLRLKVAL(&LOCK, 0);
}

```

Here, the threads use Check Lock Value to perform a race-free test and update of the lock variable, and Clear Lock Value to reset the lock variable to the unlocked state. This solves the race condition experienced in the original problem fragments. It also addresses the storage access ordering problem. As noted earlier, used in this fashion, Check Lock Value and Clear Lock Value are synchronizing actions. Using Check Lock Value to set the lock prior to reading the shared data ensures that the threads read the most recently updated data. Using Clear Lock Value to clear the lock after updating the shared data ensures that the updates are available for subsequent reads by any thread after its next synchronizing action.

Output Listing from CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM Command

This topic shows examples of binder listings and explains errors that could occur as a result of using the binder language.

Binder Listing

The binder listings for the Create Program (CRTPGM), Create Service Program (CRTSRVPGM), Update Program (UPDPGM), and Update Service Program (UPDSRVPGM) commands are almost identical. This topic presents a binder listing from the CRTSRVPGM command used to create the FINANCIAL service program in [“Binder Language Examples” on page 77](#).

Three types of listings can be specified on the detail (DETAIL) parameter of the CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM commands:

- *BASIC
- *EXTENDED
- *FULL

Basic Listing

If you specify DETAIL(*BASIC) on the CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM command, the listing consists of the following:

- The values specified on the CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM command
- A brief summary table
- Data showing the length of time some pieces of the binding process took to complete

[Figure 46 on page 164](#), [Figure 47 on page 164](#), and [Figure 48 on page 164](#) show this information.

1	Create Service Program	Page
<pre> Service program : FINANCIAL Library : MYLIB Export : *SRCFILE Export source file : QSRVSRC Library : MYLIB Export source member : *SRVPGM Activation group : *CALLER Allow update : *YES Allow bound *SRVPGM library name update : *NO Creation options : *GEN *NODUPPROC *NODUPVAR *DUPWARN Listing detail : *FULL User profile : *USER Replace existing service program : *YES Target release : *CURRENT Allow reinitialization : *NO Storage model : *SNGLVL Argument optimization : *NO Interprocedural analysis : *NO IPA control file : *NONE Authority : *LIBCRTAUT Text : Module Library Module Library MONEY MYLIB CALCS MYLIB RATES MYLIB ACCTS MYLIB Service Program Library Activation *NONE Binding Directory Library *NONE </pre>		

Figure 46. Values Specified on CRTSRVPGM Command

3	Create Service Program	Page
Brief Summary Table		
<pre> Program entry procedures : 0 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 * * * * * </pre>		

Figure 47. Brief Summary Table

23	Create Service Program	Page
Binding Statistics		
<pre> Symbol collection CPU time : .018 Symbol resolution CPU time : .006 Binding directory resolution CPU time : .403 Binder language compilation CPU time : .040 Listing creation CPU time : 1.622 Program/service program creation CPU time : .178 Total CPU time : 2.761 Total elapsed time : 11.522 * * * * * E N D O F B I N D I N G S T A T I S T I C S * * * * * *CPC5D0B - Service program FINANCIAL created in library MYLIB. * * * * * E N D O F C R E A T E S E R V I C E P R O G R A M L I S T I N G * * * * * </pre>		

Figure 48. Binding Statistics

Extended Listing

If you specify DETAIL(*EXTENDED) on the CRTPGM, CRTSRVPGM, UPDPM, or UPDSRVPGM command, the listing includes all the information provided by DETAIL(*BASIC) plus an extended summary table. The extended summary table shows the number of imports (references) that were resolved and the number of exports (definitions) processed. For the CRTSRVPGM or UPDSRVPGM command, the listing also shows the binder language used, the signatures generated, and which imports (references) matched which exports (definitions). The following listings show examples of the additional data.

Create Service Program							Page
2							
Extended Summary Table							
Valid definitions	:						418
Strong	:						418
Weak	:						0
Resolved references	:						21
To strong definitions	:						21
To weak definitions	:						0
***** END OF EXTENDED SUMMARY TABLE *****							

Create Service Program							Page
4							
Binder Information Listing							
Module	:	MONEY					
Library	:	MYLIB					
Bound	:	*YES					
Key	Number	Symbol	Ref	Identifier	Type	Scope	Export
	00000001	Def		main	Proc	Module	Strong
	00000002	Def		Amount	Proc	SrvPgm	Strong
	00000003	Def		Payment	Proc	SrvPgm	Strong
	00000004	Ref	0000017F	Q LE AG_prod_rc	Data		
	00000005	Ref	0000017E	Q LE AG_user_rc	Data		
	00000006	Ref	000000AC	_C_main	Proc		
	00000007	Ref	00000180	Q LE leDefaultEh	Proc		
	00000008	Ref	00000181	Q LE mhConversionEh	Proc		
	00000009	Ref	00000125	_C_exception_router	Proc		
Module	:	RATES					
Library	:	MYLIB					
Bound	:	*YES					
Key	Number	Symbol	Ref	Identifier	Type	Scope	Export
	0000000A	Def		Term	Proc	SrvPgm	Strong
	0000000B	Def		Rate	Proc	SrvPgm	Strong
	0000000C	Ref	0000017F	Q LE AG_prod_rc	Data		
	0000000D	Ref	0000017E	Q LE AG_user_rc	Data		
	0000000E	Ref	00000180	Q LE leDefaultEh	Proc		
	0000000F	Ref	00000181	Q LE mhConversionEh	Proc		
	00000010	Ref	00000125	_C_exception_router	Proc		
Module	:	CALCS					
Library	:	MYLIB					
Bound	:	*YES					
Key	Number	Symbol	Ref	Identifier	Type	Scope	Export
	00000011	Def		Calc1	Proc	Module	Strong
	00000012	Def		Calc2	Proc	Module	Strong
	00000013	Ref	0000017F	Q LE AG_prod_rc	Data		
	00000014	Ref	0000017E	Q LE AG_user_rc	Data		
	00000015	Ref	00000180	Q LE leDefaultEh	Proc		
	00000016	Ref	00000181	Q LE mhConversionEh	Proc		
	00000017	Ref	00000125	_C_exception_router	Proc		
Module	:	ACCTS					
Library	:	MYLIB					
Bound	:	*YES					
Key	Number	Symbol	Ref	Identifier	Type	Scope	Export
	00000018	Def		OpenAccount	Proc	SrvPgm	Strong
	00000019	Def		CloseAccount	Proc	SrvPgm	Strong
	0000001A	Ref	0000017F	Q LE AG_prod_rc	Data		
	0000001B	Ref	0000017E	Q LE AG_user_rc	Data		
	0000001C	Ref	00000180	Q LE leDefaultEh	Proc		

	0000001D	Ref	00000181	Q LE mhConversionEh	Proc		
	0000001E	Ref	00000125	_C_exception_router	Proc		
Service program :				QC2SYS			
Library :				*LIBL			
Bound :				*NO			
Key	Number	Symbol	Ref	Identifier	Type	Scope	Export
	0000001F	Def		system	Proc		Strong
Service program :				QLEAWI			
Library :				*LIBL			
Bound :				*YES			
Key	Number	Symbol	Ref	Identifier	Type	Scope	Export
	0000017E	Def		Q LE AG_user_rc	Data		Strong
	0000017F	Def		Q LE AG_prod_rc	Data		Strong
	00000180	Def		Q LE leDefaultEh	Proc		Strong
	00000181	Def		Q LE mhConversionEh	Proc		Strong
Create Service Program							Page
14							
Binder Language Listing							
STRPGMEXP PGMLVL(*CURRENT)							
EXPORT SYMBOL('Term')							
EXPORT SYMBOL('Rate')							
EXPORT SYMBOL('Amount')							
EXPORT SYMBOL('Payment')							
EXPORT SYMBOL('OpenAccount')							
EXPORT SYMBOL('CloseAccount')							
ENDPGMEXP							
***** Export signature: 00000000ADCEFE088738A98DBA6E723.							
STRPGMEXP PGMLVL(*PRV)							
EXPORT SYMBOL('Term')							
EXPORT SYMBOL('Rate')							
EXPORT SYMBOL('Amount')							
EXPORT SYMBOL('Payment')							
ENDPGMEXP							
***** Export signature: 00000000000000000000ADC89D09E0C6E7.							
* * * * * E N D O F B I N D E R L A N G U A G E L I S T I N G * * * * *							

Full Listing

If you specify DETAIL(*FULL) on the CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM command, the listing includes all the detail provided for DETAIL(*EXTENDED) plus a cross-reference listing. The following listing shows a partial example of the additional data provided.

Create Service Program			Page			
15						
Cross-Reference Listing						
Identifier	Defs	-----Refs-----		Type	Library	Object
.	.	Ref	Ref	.	.	.
.	.			.	.	.
.	.			.	.	.
xlatewt	000000DD			*SRVPGM	*LIBL	
QC2UTIL1						
yn	00000140			*SRVPGM	*LIBL	
QC2UTIL2						
y0	0000013E			*SRVPGM	*LIBL	
QC2UTIL2						
y1	0000013F			*SRVPGM	*LIBL	
QC2UTIL2						
Amount	00000002			*MODULE	MYLIB	MONEY
Calc1	00000011			*MODULE	MYLIB	CALCS
Calc2	00000012			*MODULE	MYLIB	CALCS
CloseAccount	00000019			*MODULE	MYLIB	ACCTS
CEECRHP	000001A0			*SRVPGM	*LIBL	QLEAWI
CEECZST	0000019F			*SRVPGM	*LIBL	QLEAWI
CEEDATE	000001A9			*SRVPGM	*LIBL	QLEAWI
CEEDATM	000001B1			*SRVPGM	*LIBL	QLEAWI
CEEDAYS	000001A8			*SRVPGM	*LIBL	QLEAWI
CEEDCOD	00000187			*SRVPGM	*LIBL	QLEAWI
CEEDSHP	000001A1			*SRVPGM	*LIBL	QLEAWI
CEEDYWK	000001B3			*SRVPGM	*LIBL	QLEAWI

CEEFMDA	000001AD			*SRVPGM	*LIBL	QLEAWI
CEEFMDT	000001AF			*SRVPGM	*LIBL	QLEAWI
CEEFTMT	000001AE			*SRVPGM	*LIBL	QLEAWI
CEEFRST	0000019E			*SRVPGM	*LIBL	QLEAWI
CEEGMT	000001B6			*SRVPGM	*LIBL	QLEAWI
CEEGPID	00000195			*SRVPGM	*LIBL	QLEAWI
CEEGTST	0000019D			*SRVPGM	*LIBL	QLEAWI
CEEISEC	000001B0			*SRVPGM	*LIBL	QLEAWI
CEELOCT	000001B4			*SRVPGM	*LIBL	QLEAWI
CEEMGET	000001B3			*SRVPGM	*LIBL	QLEAWI
CEEMKHP	000001A2			*SRVPGM	*LIBL	QLEAWI
CEEMOUT	000001B4			*SRVPGM	*LIBL	QLEAWI
CEEMRCR	000001B2			*SRVPGM	*LIBL	QLEAWI
CEEMSG	000001B5			*SRVPGM	*LIBL	QLEAWI
CEENCOD	000001B6			*SRVPGM	*LIBL	QLEAWI
CEEQCEN	000001AC			*SRVPGM	*LIBL	QLEAWI
CEERLHP	000001A3			*SRVPGM	*LIBL	QLEAWI
CEEScen	000001AB			*SRVPGM	*LIBL	QLEAWI
CEESECI	000001B2			*SRVPGM	*LIBL	QLEAWI
CEESECS	000001AA			*SRVPGM	*LIBL	QLEAWI
CEESGL	00000190			*SRVPGM	*LIBL	QLEAWI
CEETREC	00000191			*SRVPGM	*LIBL	QLEAWI
CEEUTC	000001B5			*SRVPGM	*LIBL	QLEAWI
CEEUTC0	000001B7			*SRVPGM	*LIBL	QLEAWI
CEE4ABN	00000192			*SRVPGM	*LIBL	QLEAWI
CEE4CpyDvfb	0000019A			*SRVPGM	*LIBL	QLEAWI
CEE4CpyIofb	00000199			*SRVPGM	*LIBL	QLEAWI
CEE4Cpy0fb	00000198			*SRVPGM	*LIBL	QLEAWI
CEE4DAS	000001A4			*SRVPGM	*LIBL	QLEAWI
CEE4FCB	0000018A			*SRVPGM	*LIBL	QLEAWI
CEE4HC	00000197			*SRVPGM	*LIBL	QLEAWI
CEE4RAGE	0000018B			*SRVPGM	*LIBL	QLEAWI
CEE4RIN	00000196			*SRVPGM	*LIBL	QLEAWI
OpenAccount	00000018			*MODULE	MYLIB	ACCTS
Payment	00000003			*MODULE	MYLIB	MONEY
Q LE leBdyCh	00000188			*SRVPGM	*LIBL	QLEAWI
Q LE leBdyEpilog	00000189			*SRVPGM	*LIBL	QLEAWI
Q LE leDefaultEh	00000180	00000007	0000000E	*SRVPGM	*LIBL	QLEAWI
	00000015		0000001C			
Q LE mhConversionEh	00000181	00000008	0000000F	*SRVPGM	*LIBL	QLEAWI
	00000016		0000001D			
Q LE AG_prod_rc	0000017F	00000004	0000000C	*SRVPGM	*LIBL	QLEAWI
	00000013	0000001A				
Q LE AG_user_rc	0000017E	00000005	0000000D	*SRVPGM	*LIBL	QLEAWI
		00000014	0000001B			
Q LE HdliRouterEh	0000018F			*SRVPGM	*LIBL	QLEAWI
Q LE RtxRouterCh	0000018E			*SRVPGM	*LIBL	QLEAWI
Rate	0000000B			*MODULE	MYLIB	RATES
Term	0000000A			*MODULE	MYLIB	RATES

IPA Listing Components

The following sections describe the IPA components of the listing:

- Object File Map
- Compiler Options Map
- Inline Report
- Global Symbols Map
- Partition Map
- Source File Map
- Messages
- Message Summary

The CRTPGM or CRTSRVPGM command generates all of these sections, except the inline report, if you specify IPA(*YES) and DETAIL(*BASIC or *EXTENDED). The CRTPGM or CRTSRVPGM command generates the inline report only if you specify IPA(*YES) and DETAIL(*FULL).

Object File Map

The Object File Map listing section displays the names of the object files that were used as input to IPA. Other listing sections, such as the Source File Map, use the FILE ID numbers that appear in this listing section.

Compiler Options Map

The Compiler Options Map listing section identifies the compiler options that were specified within the IL data for each compilation unit that is processed. For each compilation unit, it displays the options that are relevant to IPA processing. You can specify these options through a compiler option, a `#pragma` directive, or as default values.

Inline Report

The Inline Report listing section describes the actions that are performed by the IPA inliner. In this report, the term 'subprogram' is equivalent to a C/C++ function or a C++ method. The summary contains such information as:

- Name of each defined subprogram. IPA sorts subprogram names in alphabetical order.
- Reason for action on a subprogram:
 - You specified `#pragma noline` for the subprogram.
 - You specified `#pragma inline` for the subprogram.
 - IPA performed automatic inlining on the subprogram.
 - There was no reason to inline the subprogram.
 - There was a partition conflict.
 - IPA could not inline the subprogram because IL data did not exist.
- Action on a subprogram:
 - IPA inlined subprogram at least once.
 - IPA did not inline subprogram because of initial size constraints.
 - IPA did not inline subprogram because of expansion beyond size constraint.
 - The subprogram was a candidate for inlining, but IPA did not inline it.
 - Subprogram was a candidate for inlining, but was not referred to.
 - The subprogram is directly recursive, or some calls have mismatched parameters.
- Status of original subprogram after inlining:
 - IPA discarded the subprogram because it is no longer referred to and is defined as static internal.
 - IPA did not discard the subprogram, for various reasons:
 - Subprogram is external. (It can be called from outside the compilation unit.)
 - Subprogram call to this subprogram remains.
 - Subprogram has its address taken.
- Initial relative size of subprogram (in Abstract Code Units).
- Final relative size of subprogram (in Abstract Code Units) after inlining.
- The number of calls within the subprogram and the number of these calls that IPA inlined into the subprogram.
- The number of times the subprogram is called by others in the compile unit and the number of times IPA inlined the subprogram.
- The mode that is selected and the value of threshold and limit specified. Static functions whose names may not be unique within the application as a whole will have names prefixed with `@nnn@` or `XXXX@nnn@`, where `XXXX` is the partition name, and where `nnn` is the source file number.

The detailed call structure contains specific information of each subprogram such as:

- Subprograms that it calls.
- Subprograms that call it.
- Subprograms in which it is inlined.

The information can allow better analysis of the program if you want to use the inliner in selective mode. The counts in this report do not include calls from non-IPA to IPA programs.

Global Symbols Map

The Global Symbols Map listing section shows how global symbols are mapped into members of global data structures by the global variable coalescing optimization process. It includes symbol information and file name information (file name information may be approximate). In addition, line number information may be available.

Partition Map

The Partition Map listing section describes each of the object code partitions created by IPA. It provides the following information:

- The reason for generating each partition.
- The options used to generate the object code.
- The function and global data included in the partition.
- The source files that were used to create the partition.

Source File Map

The Source File Map listing section identifies the source files that are included in the object files.

Messages

If IPA detects an error, or the possibility of an error, it issues one or more diagnostic messages, and generates the Messages listing section. This listing section contains a summary of the messages that are issued during IPA processing. The messages are sorted by severity. The Messages listing section displays the listing page number where each message was originally shown. It also displays the message text, and optionally, information relating to a file name, line (if known), and column (if known).

Message Summary

The Message Summary listing section displays the total number of messages and the number of messages for each severity level.

Listing for Example Service Program

The listings in [“Basic Listing”](#) on page 163, [“Extended Listing”](#) on page 165, and [“Full Listing”](#) on page 166 show some of the listing data generated when DETAIL(*FULL) was specified to create the FINANCIAL service program in [Figure 35 on page 81](#). The figures show the binding statistics, the binder information listing, and the cross-reference listing.

Binder Information Listing for Example Service Program

The binder information listing ([“Extended Listing”](#) on page 165) includes the following data and column headings:

- The library and name of the module or service program that was processed.

If the *Bound* field shows a value of *YES for a module object, the module is marked to be bound by copy. If the *Bound* field shows a value of *YES for a service program, the service program is bound by reference. If the *Bound* field shows a value of *NO for either a module object or service program, that object is not included in the bind. The reason is that the object did not provide an export that satisfied an unresolved import.

- Number

For each module or service program that was processed, a unique identifier (ID) is associated with each export (definition) or import (reference).

- Symbol

This column identifies the symbol name as an export (Def) or an import (Ref).

- Ref

A number specified in this column (Ref) is the unique ID of the export (Def) that satisfies the import request. For example, in [“Extended Listing” on page 165](#) the unique ID for the import 00000005 matches the unique ID for the export 0000017E.

- Identifier

This is the name of the symbol that is exported or imported. The symbol name imported for the unique ID 00000005 is Q LE AG_user_rc. The symbol name exported for the unique ID 0000017E is also Q LE AG_user_rc.

- Type

If the symbol name is a procedure, it is identified as Proc. If the symbol name is a data item, it is identified as Data.

- Scope

For modules, this column identifies whether an exported symbol name is accessed at the module level or at the public interface to a service program. If a program is being created, the exported symbol names can be accessed only at the module level. If a service program is being created, the exported symbol names can be accessed at the module level or the service program (SrvPgm) level. If an exported symbol is a part of the public interface, the value in the *Scope* column must be SrvPgm.

- Export

This column identifies the strength of a data item that is exported from a module or service program.

- Key

This column contains additional information about any weak exports. Typically this column is blank.

Cross-Reference Listing for Example Service Program

The cross-reference listing in [“Full Listing” on page 166](#) is another way of looking at the data presented in the binder information. The cross-reference listing includes the following column headings:

- Identifier

The name of the export that was processed during symbol resolution.

- Defs

The unique ID associated with each export.

- Refs

A number in this column indicates the unique ID of the import (Ref) that was resolved to this export (Def).

- Type

Identifies whether the export came from a *MODULE or a *SRVPGM object.

- Library

The library name as it was specified on the command or in the binding directory.

- Object

The name of the object that provided the export (Def).

Binding Statistics for Example Service Program

Figure 48 on page 164 shows a set of statistics for creating the service program FINANCIAL. The statistics identify where the binder spent time when it was processing the create request. You have only indirect control over the data presented in this section. Some amount of processing overhead cannot be measured. Therefore, the value listed in the *Total CPU time* field is larger than the sum of the times listed in the preceding fields.

Binder Language Errors

While the system is processing the binder language during the creation of a service program, an error might occur. If DETAIL(*EXTENDED) or DETAIL(*FULL) is specified on the Create Service Program (CRTSRVPGM) command, you can see the errors in the spooled file.

The following information messages could occur:

- Signature padded
- Signature truncated

The following warning errors could occur:

- Current export block limits interface
- Duplicate export block
- Duplicate symbol on previous export
- Level checking cannot be disabled more than once, ignored
- Multiple current export blocks not allowed, previous assumed

The following serious errors could occur:

- Current export block is empty
- Export block not completed, end-of-file found before ENDPGMEXP
- Export block not started, STRPGMEXP required
- Export blocks cannot be nested, ENDPGMEXP missing
- Exports must exist inside export blocks
- Identical signatures for dissimilar export blocks, must change exports
- Multiple wildcard matches
- No current export block
- No wildcard match
- Previous export block is empty
- Signature contains variant characters
- SIGNATURE(*GEN) required with LVLCHK(*NO)
- Signature syntax not valid
- Symbol name required
- Symbol not allowed as service program export
- Symbol not defined
- Syntax not valid

Signature Padded

Figure 49 on page 172 shows a binder language listing that contains this message.

```

Binder Language Listing

STRPGMEXP SIGNATURE('Short signature')
***** Signature padded
EXPORT SYMBOL('Proc_2')
ENDPGMEXP

***** Export signature: E2889699A340A289879581A3A4998540.

* * * * * E N D O F B I N D E R L A N G U A G E L I S T I N G * * * * *

```

Figure 49. The Signature Provided Was Shorter than 16 Bytes, So It Is Padded

This is an information message.

Suggested Changes

No changes are required.

If you wish to avoid the message, make sure that the signature being provided is exactly 16 bytes long.

Signature Truncated

Figure 50 on page 172 shows a binder language listing that contains this message.

```

Binder Language Listing

STRPGMEXP SIGNATURE('This signature is very long')
***** Signature truncated
EXPORT SYMBOL('Proc_2')
ENDPGMEXP

***** Export signature: E38889A240A289879581A3A499854089.

* * * * * E N D O F B I N D E R L A N G U A G E L I S T I N G * * * * *

```

Figure 50. Only the First 16 Bytes of Data Provided Are Used for the Signature

This is an information message.

Suggested Changes

No changes are required.

If you wish to avoid the message, make sure that the signature being provided is exactly 16 bytes long.

Current Export Block Limits Interface

Figure 51 on page 172 shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
EXPORT SYMBOL(A)
EXPORT SYMBOL(B)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CD2.
STRPGMEXP PGMLVL(*PRV)
EXPORT SYMBOL(A)
EXPORT SYMBOL(B)
EXPORT SYMBOL(C)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CDE3.
***** Current export block limits interface.

* * * * * E N D O F B I N D E R L A N G U A G E L I S T I N G * * * * *

```

Figure 51. A PGMLVL(*PRV) Exported More Symbols than the PGMLVL(*CURRENT)

This is a warning error.

A PGMLVL(*PRV) export block has specified more symbols than the PGMLVL(*CURRENT) export block.

If no other errors occurred, the service program is created.

If both of the following are true:

- PGMLVL(*PRV) had supported a procedure named C
- Under the new service program, procedure C is no longer supported

any ILE program or service program that called procedure C in this service program gets an error at runtime.

Suggested Changes

1. Make sure that the PGMLVL(*CURRENT) export block has more symbols to be exported than a PGMLVL(*PRV) export block.
2. Run the CRTSRVPGM command again.

In this example, the EXPORT SYMBOL(C) was incorrectly added to the STRPGMEXP PGMLVL(*PRV) block instead of to the PGMLVL(*CURRENT) block.

Duplicate Export Block

Figure 52 on page 173 shows a binder language listing that contains this error.

```
Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(B)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CD2.
STRPGMEXP  PGMLVL(*PRV)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(B)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CD2.
***** Duplicate export block.

      * * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 52. Duplicate STRPGMEXP/ENDPGMEXP Blocks

This is a warning error.

More than one STRPGMEXP and ENDPGMEXP block exported all the same symbols in the exact same order.

If no other errors occurred, the service program is created. The duplicated signature is included only once in the created service program.

Suggested Changes

1. Make one of the following changes:
 - Make sure that the PGMLVL(*CURRENT) export block is correct. Update it as appropriate.
 - Remove the duplicate export block.
2. Run the CRTSRVPGM command again.

In this example, the STRPGMEXP command with PGMLVL(*CURRENT) specified needs to have the following source line added after EXPORT SYMBOL(B):

```
EXPORT  SYMBOL(C)
```

Duplicate Symbol on Previous Export

Figure 53 on page 174 shows a binder language listing that contains a duplicate symbol error.

```
Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
EXPORT SYMBOL(A)
EXPORT SYMBOL(B)
EXPORT SYMBOL(A)
***** Duplicate symbol on previous export
EXPORT SYMBOL(C)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CDED3.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 53. Duplicate Exported Symbols

This is a warning error.

A symbol to be exported from the service program was specified more than once in a STRPGMEXP and ENDPGMEXP block.

If no other errors occurred, the service program is created. Only the first duplicate symbol is exported from the service program. All duplicate symbols affect the signature that is generated.

Suggested Changes

1. Remove one of the duplicate source lines from the binder language source file.
2. Run the CRTSRVPGM command again.

In this example, remove the second EXPORT SYMBOL(A).

Level Checking Cannot Be Disabled More than Once, Ignored

Figure 54 on page 174 shows a binder language listing that contains this error.

```
Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT) LVLCHK(*NO)
EXPORT SYMBOL(A)
EXPORT SYMBOL(B)
ENDPGMEXP
***** Export signature: 00000000000000000000000000000000.
STRPGMEXP  PGMLVL(*PRV) LVLCHK(*NO)
***** Level checking cannot be disabled more than once, ignored
EXPORT SYMBOL(A)
ENDPGMEXP
***** Export signature: 000000000000000000000000000000C1.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 54. Multiple STRPGMEXP Commands Have LVLCHK(*NO) Specified

This is a warning error.

More than one STRPGMEXP blocks specified LVLCHK(*NO).

If no other errors occurred, the service program is created. The second and subsequent LVLCHK(*NO) are assumed to be LVLCHK(*YES).

Suggested Changes

1. Make sure that only one STRPGMEXP block has LVLCHK(*NO) specified.

2. Run the CRTSRVPGM command again.

In this example, the PGMLVL(*PRV) export block is the only export block that has LVLCHK(*NO) specified. The LVLCHK(*NO) value is removed from the PGMLVL(*CURRENT) export block.

Multiple Current Export Blocks Not Allowed, Previous Assumed

Figure 55 on page 175 shows a binder language listing that contains this error.

```
Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
EXPORT SYMBOL(A)
EXPORT SYMBOL(B)
EXPORT SYMBOL(C)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CDE3.
STRPGMEXP
EXPORT SYMBOL(A)
***** Multiple 'current' export blocks not allowed, 'previous' assumed.
EXPORT SYMBOL(B)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CD2.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 55. More than One PGMLVL(*CURRENT) Value Specified

This is a warning error.

A value of PGMLVL(*CURRENT) was specified or was allowed to default to PGMLVL(*CURRENT) on more than one STRPGMEXP command. The second and subsequent export blocks with a value of PGMLVL(*CURRENT) are assumed to be PGMLVL(*PRV).

If no other errors occurred, the service program is created.

Suggested Changes

1. Change the appropriate source text to STRPGMEXP PGMLVL(*PRV).
2. Run the CRTSRVPGM command again.

In this example, the second STRPGMEXP is the one to change.

Current Export Block Is Empty

Figure 56 on page 175 shows a binder language listing that contains this error.

```
Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
ENDPGMEXP
***** Export signature: 00000000000000000000000000000000.
***ERROR Current export block is empty.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 56. No Symbols to Be Exported from the STRPGMEXP PGMLVL(*CURRENT) Block

This is a serious error.

No symbols are identified to be exported from the *CURRENT export block.

The service program is not created.

Suggested Changes

1. Make one of the following changes:

- Add the symbol names to be exported.
- Remove the empty STRPGMEXP-ENDPGMEXP block, and make another STRPGMEXP-ENDPGMEXP block as PGMLVL(*CURRENT).

2. Run the CRTSRVPGM command.

In this example, the following source line is added to the binder language source file between the STRPGMEXP and ENDPGMEXP commands:

```
EXPORT SYMBOL(A)
```

Export Block Not Completed, End-of-File Found before ENDPGMEXP

Figure 57 on page 176 shows a binder language listing that contains this error.

```
Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
***ERROR Syntax not valid.
***ERROR Export block not completed, end-of-file found before ENDPGMEXP.

      * * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 57. No ENDPGMEXP Command Found, but the End of the Source File Was Found

This is a serious error.

No ENDPGMEXP was found before the end of the file was reached.

The service program is not created.

Suggested Changes

1. Make one of the following changes:

- Add the ENDPGMEXP command in the appropriate place.
- Remove any STRPGMEXP command that does not have a matching ENDPGMEXP command, and remove any symbol names to be exported.

2. Run the CRTSRVPGM command.

In this example, the following lines are added after the STRPGMEXP command:

```
EXPORT SYMBOL(A)
ENDPGMEXP
```

Export Block Not Started, STRPGMEXP Required

Figure 58 on page 176 shows a binder language listing that contains this error.

```
Binder Language Listing

ENDPGMEXP
***ERROR Export block not started, STRPGMEXP required.
***ERROR No 'current' export block

      * * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 58. STRPGMEXP Command Is Missing

This is a serious error.

No STRPGMEXP command was found prior to finding an ENDPGMEXP command.

The service program is not created.

Suggested Changes

1. Make one of the following changes:
 - Add the STRPGMEXP command.
 - Remove any exported symbols and the ENDPGMEXP command.
2. Run the CRTSRVPGM command.

In this example, the following two source lines are added to the binder language source file before the ENDPGMEXP command.

```
STRPGMEXP
EXPORT  SYMBOL(A)
```

Export Blocks Cannot Be Nested, ENDPGMEXP Missing

[Figure 59 on page 177](#) shows a binder language listing that contains this error.

```

                                     Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT  SYMBOL(A)
  EXPORT  SYMBOL(B)
STRPGMEXP  PGMLVL(*PRV)
***ERROR Export blocks cannot be nested, ENDPGMEXP missing.
  EXPORT  SYMBOL(A)
ENDPGMEXP
***** Export signature: 00000000000000000000000000000000C1.

      * * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *
```

Figure 59. ENDPGMEXP Command Is Missing

This is a serious error.

No ENDPGMEXP command was found prior to finding another STRPGMEXP command.

The service program is not created.

Suggested Changes

1. Make one of the following changes:
 - Add the ENDPGMEXP command prior to the next STRPGMEXP command.
 - Remove the STRPGMEXP command and any symbol names to be exported.
2. Run the CRTSRVPGM command.

In this example, an ENDPGMEXP command is added to the binder source file prior to the second STRPGMEXP command.

Exports Must Exist inside Export Blocks

[Figure 60 on page 178](#) shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
EXPORT SYMBOL(A)
EXPORT SYMBOL(B)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CD2.
EXPORT SYMBOL(A)
***ERROR Exports must exist inside export blocks.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *

```

Figure 60. Symbol Name to Be Exported Is outside the STRPGMEXP-ENDPGMEXP Block

This is a serious error.

A symbol to be exported is not defined within a STRPGMEXP-ENDPGMEXP block.

The service program is not created.

Suggested Changes

1. Make one of the following changes:
 - Move the symbol to be exported. Put it within a STRPGMEXP-ENDPGMEXP block.
 - Remove the symbol.
2. Run the CRTSRVPGM command.

In this example, the source line in error is removed from the binder language source file.

Identical Signatures for Dissimilar Export Blocks, Must Change Exports

This is a serious error.

Identical signatures have been generated from STRPGMEXP-ENDPGMEXP blocks that exported different symbols. This error condition is highly unlikely to occur. For any set of nontrivial symbols to be exported, this error should occur only once every 3.4E28 tries.

The service program is not created.

Suggested Changes

1. Make one of the following changes:
 - Add an additional symbol to be exported from the PGMLVL(*CURRENT) block.
The preferred method is to specify a symbol that is already exported. This would cause a warning error of duplicate symbols but would help ensure that a signature is unique. An alternative method is to add another symbol to be exported that has not been exported.
 - Change the name of a symbol to be exported from a module, and make the corresponding change to the binder language source file.
 - Specify a signature by using the SIGNATURE parameter on the Start Program Export (STRPGMEXP) command.
2. Run the CRTSRVPGM command.

Multiple Wildcard Matches

[Figure 61 on page 179](#) shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
EXPORT ("A"<<)
***ERROR Multiple matches of wildcard specification
EXPORT ("B"<<)
ENDPGMEXP
***** Export signature: 000000000000000000000000FFC2.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G

```

Figure 61. Multiple Matches of Wildcard Specification

This is a serious error.

A wildcard specified for export matched more than one symbol available for export.

The service program is not created.

Suggested Changes

1. Specify a wildcard with more detail so that the desired matching export is the only matching export.
2. Run the CRTSRVPGM command.

No Current Export Block

Figure 62 on page 179 shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP PGMLVL(*PRV)
EXPORT SYMBOL(A)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000C1.
***ERROR No 'current' export block

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *

```

Figure 62. No PGMLVL(*CURRENT) Export Block

This is a serious error.

No STRPGMEXP PGMLVL(*CURRENT) is found in the binder language source file.

The service program is not created.

Suggested Changes

1. Make one of the following changes:
 - Change a PGMLVL(*PRV) to PGMLVL(*CURRENT).
 - Add a STRPGMEXP-ENDPGMEXP block that is the correct *CURRENT export block.
2. Run the CRTSRVPGM command.

In this example, the PGMLVL(*PRV) is changed to PGMLVL(*CURRENT).

No Wildcard Matches

Figure 63 on page 180 shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
EXPORT ("Z"<<<)
***ERROR No matches of wildcard specification
EXPORT ("B"<<<)
ENDPGMEXP
***** Export signature: 000000000000000000000000FFC2.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G

```

Figure 63. No Matches of Wildcard Specification

This is a serious error.

A wildcard specified for export did not match any symbols available for export.

The service program is not created.

Suggested Changes

1. Specify a wildcard that matches the symbol desired for export.
2. Run the CRTSRVPGM command.

Previous Export Block Is Empty

Figure 64 on page 180 shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(B)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CD2.
STRPGMEXP PGMLVL(*PRV)
ENDPGMEXP
***** Export signature: 000000000000000000000000000000.
***ERROR Previous export block is empty.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *

```

Figure 64. No PGMLVL(*CURRENT) Export Block

This is a serious error.

A STRPGMEXP PGMLVL(*PRV) was found, and no symbols were specified.

The service program is not created.

Suggested Changes

1. Make one of the following changes:
 - Add symbols to the STRPGMEXP-ENDPGMEXP block that is empty.
 - Remove the STRPGMEXP-ENDPGMEXP block that is empty.
2. Run the CRTSRVPGM command.

In this example, the empty STRPGMEXP-ENDPGMEXP block is removed from the binder language source file.

Signature Contains Variant Characters

Figure 65 on page 181 shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP SIGNATURE('!\cdefghijklmnop')
***ERROR Signature contains variant characters
EXPORT SYMBOL('Proc_2')
ENDPGMEXP

***** Export signature: E05A8384858687888991929394959697.

* * * * * E N D O F B I N D E R L A N G U A G E L I S T I N G * * * * *

```

Figure 65. Signature Contains Variant Characters

This is a serious error.

The signature contains characters that are not in all coded character set identifiers (CCSIDs).

The service program is not created.

Suggested Changes

1. Remove the variant characters.
2. Run the CRTSRVPGM command.

In this specific case, it is the \! that needs to be removed.

SIGNATURE(*GEN) Required with LVLCHK(*NO)

[Figure 66 on page 181](#) shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP SIGNATURE('ABCDEFGHIJKLMNOP') LVLCHK(*NO)
EXPORT SYMBOL('Proc_2')
***ERROR SIGNATURE(*GEN) required with LVLCHK(*NO)
ENDPGMEXP

***** Export signature: C1C2C3C4C5C6C7C8C9D1D2D3D4D5D6D7.

* * * * * E N D O F B I N D E R L A N G U A G E L I S T I N G * * * * *

```

Figure 66. If LVLCHK(*NO) Is Specified, an Explicit Signature Is Not Valid

This is a serious error.

If LVLCHK(*NO) is specified, SIGNATURE(*GEN) is required.

The service program is not created.

Suggested Changes

1. Make one of the following changes:
 - Specify SIGNATURE(*GEN)
 - Specify LVLCHK(*YES)
2. Run the CRTSRVPGM command.

Signature Syntax Not Valid

[Figure 67 on page 182](#) shows a binder language listing that contains this error.

```

Binder Language Listing

      STRPGMEXP  SIGNATURE('"abcdefghijkl "')
***ERROR Signature syntax not valid
***ERROR Signature syntax not valid
***ERROR Syntax not valid.
***ERROR Syntax not valid.
      EXPORT    SYMBOL('Proc_2')
      ENDPGMEXP

      * * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *

```

This is a serious error.

The signature contains characters that are not valid.

The service program is not created.

1. Remove the characters that are not valid from the signature value.
2. Run the CRTSRVPGM command.

Symbol Name Required

Figure 68 on page 182 shows a binder language listing that contains this error.

Figure 68. No Symbol to Be Exported

This is a serious error.
No symbol name was found to export from the service program.
The service program is not created.

1. Make one of the following changes:
 - Remove the line in error from the binder language source file.
 - Add a symbol name to be exported from the service program.
2. Run the CRTSRVPGM command.

Symbol Not Allowed as Service Program Export

Figure 69 on page 183 shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
EXPORT SYMBOL(A)
***ERROR Symbol not allowed as service program export.
EXPORT SYMBOL(D)
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CD4.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *

```

Figure 69. Symbol Name Not Valid to Export from Service Program

This is a serious error.

The symbol to be exported from the service program was not exported from one of the modules to be bound by copy. Typically the symbol specified to be exported from the service program is actually a symbol that needs to be imported by the service program.

The service program is not created.

Suggested Changes

1. Make one of the following changes:

- Remove the symbol in error from the binder language source file.
- On the MODULE parameter of the CRTSRVPGM command, specify the module that has the desired symbol to be exported.
- Add the symbol to one of the modules that will be bound by copy, and re-create the module object.

2. Run the CRTSRVPGM command.

In this example, the source line of EXPORT SYMBOL(A) is removed from the binder language source file.

Symbol Not Defined

Figure 70 on page 183 shows a binder language listing that contains this error.

```

Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
EXPORT SYMBOL(A)
EXPORT SYMBOL(Q)
***ERROR Symbol not defined.
ENDPGMEXP
***** Export signature: 0000000000000000000000000000CE8.

* * * * *   E N D   O F   B I N D E R   L A N G U A G E   L I S T I N G   * * * * *

```

Figure 70. Symbol Not Found in the Modules That Are to Be Bound by Copy

This is a serious error.

The symbol to be exported from the service program could not be found in the modules that are to be bound by copy.

The service program is not created.

Suggested Changes

1. Make one of the following changes:

- Remove the symbol that is not defined from the binder language source file.
- On the MODULE parameter of the CRTSRVPGM command, specify the module that has the desired symbol to be exported.

- Add the symbol to one of the modules that will be bound by copy, and re-create the module object.
2. Run the CRTSRVPGM command.

In this example, the source line of EXPORT SYMBOL(Q) is removed from the binder language source file.

Syntax Not Valid

This is a serious error.

The statements in the source member are not valid binder language statements.

The service program is not created.

Suggested Changes

1. Correct the source member so it contains valid binder language statements.
2. Run the CRTSRVPGM command.

Exceptions in Optimized Programs

In rare circumstances, an MCH3601 exception message may occur in programs compiled with optimization level 30 (*FULL) or 40. This topic explains one example in which this message occurs. The same program does not receive an MCH3601 exception message when compiled with optimization level 10 (*NONE) or 20 (*BASIC). Whether the message in this example occurs depends on how your ILE HLL compiler allocates storage for arrays. This example might never occur for your language.

When you ask for optimization level 30 (*FULL) or 40, ILE attempts to improve performance by calculating array index references outside of loops. When you refer to an array in a loop, you are often accessing every element in order. Performance can be improved by saving the last array element address from the previous loop iteration. To accomplish this performance improvement, ILE calculates the first array element address outside the loop and saves the value for use inside the loop.

Take the following example:

```
DCL ARR[1000] INTEGER;
DCL I INTEGER;

I = init_expression; /* Assume that init_expression evaluates
                      to -1 which is then assigned to I */

/* More statements */
WHILE ( I < limit_expression )
    I = I + 1;
    /* Some statements in the while loop */
    ARR[I] = some_expression;
    /* Other statements in the while loop */
END;
```

If a reference to ARR[init_expression] would have produced an incorrect array index, this example can cause an MCH3601 exception. This is because ILE attempted to calculate the first array element address before entering the WHILE loop.

If you receive MCH3601 exceptions at optimization level 30 (*FULL) or 40, look for the following situation:

1. You have a loop that increments a variable before it uses the variable as an array element index.
2. The initial value of the index variable on entrance to the loop is negative.
3. A reference to the array using the initial value of the variable is not valid.

When these conditions exist, it may be possible to do the following so that optimization level 30 (*FULL) or 40 can still be used:

1. Move the part of the program that increments the variable to the bottom of the loop.
2. Change the references to the variables as needed.

The previous example would be changed as follows:

```
I = init_expression + 1;
WHILE ( I < limit_expression + 1 )
    ARR[I] = some_expression;
    I = I + 1;
END;
```

If this change is not possible, reduce the optimization level from 30 (*FULL) or 40 to 20 (*BASIC) or 10 (*NONE).

CL Commands Used with ILE Objects

The following tables indicate which CL commands can be used with each ILE object.

CL Commands Used with Modules

Table 14. CL Commands Used with Modules	
Command	Descriptive Name
CHGMOD	Change Module
CRTCBMOD	Create COBOL Module
CRTCLMOD	Create CL Module
CRTCMOD	Create C Module
CRTCPPMOD	Create C++ Module
CRTRPGMOD	Create RPG Module
DLTMOD	Delete Module
DSPMOD	Display Module
RTVBNDSRC	Retrieve Binder Source
WRKMOD	Work with Module

CL Commands Used with Program Objects

Table 15. CL Commands Used with Program Objects	
Command	Descriptive Name
CHGPGM	Change Program
CRTBNDC	Create Bound C Program
CRTBNDCBL	Create Bound COBOL Program
CRTBNDCL	Create Bound CL Program
CRTBNDCPP	Create Bound C++ Program
CRTBNDRPG	Create Bound RPG Program
CRTPGM	Create Program
DLTPGM	Delete Program
DSPPGM	Display Program
DSPPGMREF	Display Program References
UPDPGM	Update Program
WRKPGM	Work with Program

CL Commands Used with Service Programs

Table 16. CL Commands Used with Service Programs	
Command	Descriptive Name
CHGSRVPGM	Change Service Program
CRTSRVPGM	Create Service Program
DLTSRVPGM	Delete Service Program
DSPSRVPGM	Display Service Program
RTVBNDSRC	Retrieve Binder Source
UPDSRVPGM	Update Service Program
WRKSRVPGM	Work with Service Program

CL Commands Used with Binding Directories

Table 17. CL Commands Used with Binding Directories	
Command	Descriptive Name
ADDBNDDIRE	Add Binding Directory Entry
CRTBNDDIR	Create Binding Directory
DLTBNDDIR	Delete Binding Directory
DSPBNDDIR	Display Binding Directory
RMVBNDDIRE	Remove Binding Directory Entry
WRKBNDDIR	Work with Binding Directory
WRKBNDDIRE	Work with Binding Directory Entry

CL Commands Used with Structured Query Language

Table 18. CL Commands Used with Structured Query Language	
Command	Descriptive Name
CRTSQLCI	Create Structured Query Language ILE C Object
CRTSQLCBLI	Create Structured Query Language ILE COBOL Object
CRTSQLCPPI	Create SQL ILE C++ Object
CRTSQLRPGI	Create Structured Query Language ILE RPG Object

CL Commands Used with CICS

Table 19. CL Commands Used with CICS	
Command	Descriptive Name
CRTCICSC	Create CICS® ILE C Object
CRTCICSCBL	Create CICS COBOL Program

CL Commands Used with Source Debugger

Table 20. CL Commands Used with Source Debugger

Command	Descriptive Name
DSPMODSRC	Display Module Source
ENDDBG	End Debug
STRDBG	Start Debug

CL Commands Used to Edit the Binder Language Source File

Table 21. CL Commands Used to Edit the Binder Language Source

Command	Descriptive Name
EDTF	Edit File
STRPDM	Start Programming Development Manager
STRSEU	Start Source Entry Utility
Note: The following nonrunnable commands can be entered into the binder language source file:	
ENDPGMEXP	End Program Export
EXPORT	Export

Related information

For additional information about topics related to the ILE environment on the IBM i operating system, refer to the following publications:

- The [Backup and recovery](#) category provides information about planning a backup and recovery strategy, the different types of media available to save and restore system data, as well as a description of how to record changes made to database files using journaling and how that information can be used for system recovery. This manual describes how to plan for and set up user auxiliary storage pools (ASPs), mirrored protection, and checksums along with other availability recovery topics. It also describes how to install the system again from backup.
- The CL programming book provides a wide-ranging discussion of programming topics, including a general discussion of objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and immediate messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- [Communications Management](#)  provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.
- [ICF Programming](#)  provides information needed to write application programs that use communications and the intersystem communications function. This guide also contains information about data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- [Rational® Developer for i: ILE C/C++ Programmer's Guide](#)  describes how to create, compile, debug, and run ILE C and ILE C++ programs. The guide provides information about ILE and IBM i programming features; file systems, devices, and features; I/O operations; localization; program performance; and C++ runtime type information (RTTI).
- [Rational Developer for i: ILE C/C++ Language Reference](#)  describes language conformance to the *Programming languages - C* standards and the *Programming languages - C++* standards.
- [Rational Developer for i: ILE C/C++ Compiler Reference](#)  contains reference information for the ILE C/C++ compiler, including preprocessor statements, macros, pragmas, command line use for IBM i and Qshell environments, and I/O considerations.
- [ILE C/C++ Runtime Library Functions](#)  provides reference information about ILE C/C++ , runtime library functions, include files, and runtime considerations.
- [Rational Developer for i: ILE COBOL Programmer's Guide](#)  describes how to write, compile, bind, run, debug, and maintain ILE COBOL programs on the IBM i operating system. It provides programming information about how to call other ILE COBOL and non-ILE COBOL programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.
- [Rational Developer for i: ILE COBOL Language Reference](#)  describes the ILE COBOL programming language. It provides information on the structure of the ILE COBOL programming language and on the structure of an ILE COBOL source program. It also describes all Identification Division paragraphs, Environment Division clauses, Data Division paragraphs, Procedure Division statements, and Compiler-Directing statements.

- [Rational Developer for i: ILE RPG Programmer's Guide](#)  is a guide for using the RPG IV programming language, which is an implementation of ILE RPG in the Integrated Language Environment (ILE) on the IBM i operating system. It includes information about creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use IBM i files and devices in RPG programs. Topics include information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- [Rational Developer for i: ILE RPG Language Reference](#)  provides information needed to write programs for the IBM i operating system using the RPG IV programming language. This manual describes, position by position and keyword by keyword, the valid entries for all RPG specifications, and provides a detailed description of all the operation codes and built-in functions. This manual also contains information on the RPG logic cycle, arrays and tables, editing functions, and indicators.
- [Intrasystem Communications Programming](#)  provides information about interactive communications between two application programs on the same system. This guide describes the communications operations that can be coded into a program that uses intrasystem communications support to communicate with another program. It also provides information about developing intrasystem communications application programs that use the intersystem communications function.
- [Security reference](#)  tells how system security support can be used to protect the system and the data from being used by people who do not have the proper authorization, protect the data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.
- The [Work management](#) topic, under the Systems management category of the IBM i Information Center, provides information about how to create and change a work management environment. Other topics include a description of tuning the system, collecting performance data including information about record formats and contents of the data being collected, working with system values to control or change the overall operation of the system, and a description of how to gather data to determine who is using the system and what resources are being used.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
2800 37th Street NW
Rochester, MN 55901-4441
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Programming interface information

This ILE C/C++ Compiler Reference publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "[Copyright and trademark information](http://www.ibm.com/legal/copytrade.shtml)" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Index

Special Characters

[_C_TS_calloc\(\)](#) 56
[_C_TS_free\(\)](#) 56
[_C_TS_malloc\(\)](#) 56
[_C_TS_malloc64\(\)](#) 56
[_C_TS_realloc\(\)](#) 56
[_CEE4ALC](#) allocation strategy type [107](#)

A

Abnormal End (CEE4ABN) bindable API [115](#)
access ordering
 shared storage [157](#)
ACTGRP 86
ACTGRP (activation group) parameter
 *CALLER value 94
 activation group creation [32](#)
 program activation [30](#), [32](#)
actions
 storage synchronizing [159](#)
activation
 description [25](#)
 dynamic program call [101](#)
 program [29](#)
 program activation [35](#)
 service program [35](#), [98](#)
activation group
 ACTGRP (activation group) parameter
 *CALLER value 94
 activation group creation [30](#)
 program activation [30](#), [32](#)
 benefits of resource scoping [9](#)
 bindable APIs (application programming interfaces) [127](#)
 call stack example [30](#)
 commitment control
 example [9](#)
 scoping [124](#)
 control boundary
 activation group deletion [34](#)
 example [37](#)
 creation [31](#)
 data management scoping [46](#), [124](#)
 default [32](#)
 deletion [33](#)
 management [91](#)
 mixing COBOL with other languages [10](#)
 multiple applications running in same job [91](#)
 original program model (OPM) [32](#)
 reclaim resources [92–94](#)
 resource isolation [30](#)
 resources [30](#)
 reuse [33](#)
 scoping [46](#), [124](#)
 service program [94](#)
 shared open data path (ODP) example [9](#)
 system-named [32](#), [34](#)

activation group (*continued*)
 user-named
 deletion [34](#)
 description [32](#), [91](#)
activation group selection for teraspace storage model [51](#)
adaptive code generation (ACG) [149](#)
advanced concepts [29](#)
ALWLIBUPD parameter
 on CRTPGM command [86](#)
 on CRTSRVPGM command [86](#)
ALWUPD parameter
 on CRTPGM command [86](#)
 on CRTSRVPGM command [86](#)
API (application programming interface)
 Abnormal End (CEE4ABN) [115](#)
 activation group [127](#)
 CEE4ABN (Abnormal End) [115](#)
 CEEDOD (Retrieve Operational Descriptor Information) [103](#)
 CEEGSI (Get String Information) [103](#)
 CEEHDLR (Register User-Written Condition Handler) [43](#), [111](#)
 CEEHDLU (Unregister User-Written Condition Handler) [43](#)
 CEEMGET (Get Message) [118](#)
 CEEMOUT (Dispatch Message) [118](#)
 CEEMRCR (Move Resume Cursor) [113](#)
 CEEMSG (Get, Format and Dispatch Message) [118](#)
 CEENCOD (Construct Condition Token) [115](#)
 CEESGL (Signal Condition)
 condition token [115](#), [118](#)
 description [41](#)
 CEETSTA (Test for Omitted Argument) [101](#)
 Change Exception Message (QMCHGEM) [113](#)
 condition management [127](#), [129](#)
 Construct Condition Token (CEENCOD) [115](#)
 control flow [127](#)
 date [128](#)
 debugger [129](#)
 Dispatch Message (CEEMOUT) [118](#)
 dynamic screen manager (DSM) [130](#)
 error handling [129](#)
 exception management [127](#), [129](#)
 Get Message (CEEMGET) [118](#)
 Get String Information (CEEGSI) [103](#)
 Get, Format and Dispatch Message (CEEMSG) [118](#)
 HLL independence [127](#)
 list of [127](#), [130](#)
 math [128](#)
 message handling [129](#)
 Move Resume Cursor (CEEMRCR) [113](#)
 naming conventions [127](#)
 original program model (OPM) and ILE [103](#)
 procedure call [129](#)
 program call [129](#)
 Promote Message (QMHPRMM) [113](#)
 QCAPCMD [94](#)

API (application programming interface) *(continued)*

- QMCHGEM (Change Exception Message) [113](#)
- QMHPRMM (Promote Message) [113](#)
- QMHSNDPM (Send Program Message) [41](#), [111](#)
- Register User-Written Condition Handler (CEEHDLR) [43](#), [111](#)
- Retrieve Operational Descriptor Information (CEEDOD) [103](#)
- Send Program Message (QMHSNDPM) [41](#), [111](#)
- services [8](#)
- Signal Condition (CEESGL)
 - condition token [115](#), [118](#)
 - description [41](#)
- source debugger [129](#)
- storage management [129](#)
- supplementing HLL-specific runtime library [127](#)
- Test for Omitted Argument (CEETSTA) [101](#)
- time [128](#)
- Unregister User-Written Condition Handler (CEEHDLU) [43](#)

application

- multiple
 - running in same job [91](#)

application programming interface (API)

- Abnormal End (CEE4ABN) [115](#)
- activation group [127](#)
- CEE4ABN (Abnormal End) [115](#)
- CEEDOD (Retrieve Operational Descriptor Information) [103](#)
- CEEGSI (Get String Information) [103](#)
- CEEHDLR (Register User-Written Condition Handler) [43](#), [111](#)
- CEEHDLU (Unregister User-Written Condition Handler) [43](#)
- CEEMGET (Get Message) [118](#)
- CEEMOUT (Dispatch Message) [118](#)
- CEEMRCR (Move Resume Cursor) [113](#)
- CEEMSG (Get, Format and Dispatch Message) [118](#)
- CEENCOD (Construct Condition Token) [115](#)
- CEESGL (Signal Condition)
 - condition token [115](#), [118](#)
 - description [41](#)
- CEETSTA (Test for Omitted Argument) [101](#)
- Change Exception Message (QMCHGEM) [113](#)
- condition management [127](#), [129](#)
- Construct Condition Token (CEENCOD) [115](#)
- control flow [127](#)
- date [128](#)
- debugger [129](#)
- Dispatch Message (CEEMOUT) [118](#)
- dynamic screen manager (DSM) [130](#)
- error handling [129](#)
- exception management [127](#), [129](#)
- Get Message (CEEMGET) [118](#)
- Get String Information (CEEGSI) [103](#)
- Get, Format and Dispatch Message (CEEMSG) [118](#)
- HLL independence [127](#)
- list of [127](#), [130](#)
- math [128](#)
- message handling [129](#)
- Move Resume Cursor (CEEMRCR) [113](#)
- naming conventions [127](#)
- original program model (OPM) and ILE [103](#)
- procedure call [129](#)

application programming interface (API) *(continued)*

- program call [129](#)
- Promote Message (QMHPRMM) [113](#)
- QCAPCMD [94](#)
- QMCHGEM (Change Exception Message) [113](#)
- QMHPRMM (Promote Message) [113](#)
- QMHSNDPM (Send Program Message) [41](#), [111](#)
- Register User-Written Condition Handler (CEEHDLR) [43](#), [111](#)
- Retrieve Operational Descriptor Information (CEEDOD) [103](#)
- Send Program Message (QMHSNDPM) [41](#), [111](#)
- services [8](#)
- Signal Condition (CEESGL)
 - condition token [115](#), [118](#)
 - description [41](#)
- source debugger [129](#)
- storage management [129](#)
- supplementing HLL-specific runtime library [127](#)
- Test for Omitted Argument (CEETSTA) [101](#)
- time [128](#)
- Unregister User-Written Condition Handler (CEEHDLU) [43](#)

argument

- passing
 - in mixed-language applications [102](#)

argument passing

- between languages [102](#)
- by reference [99](#)
- by value directly [99](#)
- by value indirectly [99](#)
- omitted arguments [101](#)
- to procedures [99](#)
- to programs [101](#)

automatic storage [105](#)

B

basic listing [163](#)

benefit of ILE

- binding [7](#)
- code optimization [11](#)
- coexistence with existing applications [8](#)
- common runtime services [8](#)
- language interaction control [10](#)
- modularity [7](#)
- resource control [8](#)
- reusable components [8](#)
- source debugger [8](#)

bind

- by copy [22](#), [64](#)
- by reference [23](#), [64](#)

bindable API

- services [8](#)

bindable API (application programming interface)

- Abnormal End (CEE4ABN) [115](#)
- activation group [127](#)
- CEE4ABN (Abnormal End) [115](#)
- CEEDOD (Retrieve Operational Descriptor Information) [103](#)
- CEEGSI (Get String Information) [103](#)
- CEEHDLR (Register User-Written Condition Handler) [43](#), [111](#)

- bindable API (application programming interface) (*continued*)
 - CEEHDLU (Unregister User-Written Condition Handler) [43](#)
 - CEEMGET (Get Message) [118](#)
 - CEEMOUT (Dispatch Message) [118](#)
 - CEEMRCR (Move Resume Cursor) [113](#)
 - CEEMSG (Get, Format and Dispatch Message) [118](#)
 - CEENCOD (Construct Condition Token) [115](#)
 - CEESGL (Signal Condition)
 - condition token [115](#), [118](#)
 - description [41](#)
 - CEETSTA (Test for Omitted Argument) [101](#)
 - condition management [127](#), [129](#)
 - Construct Condition Token (CEENCOD) [115](#)
 - control flow [127](#)
 - date [128](#)
 - debugger [129](#)
 - Dispatch Message (CEEMOUT) [118](#)
 - dynamic screen manager (DSM) [130](#)
 - error handling [129](#)
 - exception management [127](#), [129](#)
 - Get Message (CEEMGET) [118](#)
 - Get String Information (CEEGSI) [103](#)
 - Get, Format and Dispatch Message (CEEMSG) [118](#)
 - HLL independence [127](#)
 - list of [127](#), [130](#)
 - math [128](#)
 - message handling [129](#)
 - Move Resume Cursor (CEEMRCR) [113](#)
 - naming conventions [127](#)
 - original program model (OPM) and ILE [103](#)
 - procedure call [129](#)
 - program call [129](#)
 - Register User-Written Condition Handler (CEEHDLR) [43](#), [111](#)
 - Retrieve Operational Descriptor Information (CEEDOD) [103](#)
 - Signal Condition (CEESGL)
 - condition token [115](#), [118](#)
 - description [41](#)
 - source debugger [129](#)
 - storage management [129](#)
 - supplementing HLL-specific runtime library [127](#)
 - Test for Omitted Argument (CEETSTA) [101](#)
 - time [128](#)
 - Unregister User-Written Condition Handler (CEEHDLU) [43](#)
- binder [22](#)
- binder information listing
 - service program example [169](#)
- binder language
 - definition [73](#)
 - ENDPGMEXP (End Program Export) command [74](#), [75](#)
 - error [171](#)
 - examples [77](#), [84](#)
 - EXPORT (Export Symbol) [74](#), [76](#)
 - STRPGMEXP (Start Program Export)
 - LVLCHK parameter [75](#)
 - PGMLVL parameter [75](#)
 - SIGNATURE parameter [75](#)
 - STRPGMEXP (Start Program Export) command [75](#)
- binder listing
 - basic [163](#)
 - extended [165](#)

- binder listing (*continued*)
 - full [166](#)
 - service program example [169](#)
- binding
 - benefit of ILE [7](#)
 - large number of modules [64](#)
 - original program model (OPM) [12](#)
- binding directory
 - CL (control language) commands [188](#)
 - definition [21](#)
- binding statistics
 - service program example [171](#)
- BNDDIR parameter on UPDPGM command [86](#)
- BNDDIR parameter on UPDSRVPGM command [86](#)
- BNSRVPGM parameter on UPDPGM command [86](#)
- BNSRVPGM parameter on UPDSRVPGM command [86](#)
- by reference, passing arguments [99](#)
- by value directly, passing arguments [99](#)
- by value indirectly, passing arguments [99](#)

C

- C signal [41](#)
- call
 - procedure [24](#), [97](#)
 - procedure pointer [97](#)
 - program [24](#), [97](#)
- call message queue [40](#)
- call stack
 - activation group example [30](#)
 - definition [97](#)
 - example
 - dynamic program calls [97](#)
 - static procedure calls [97](#)
- call-level scoping [46](#)
- callable service [127](#)
- Case component of condition token [116](#)
- CEE4ABN (Abnormal End) bindable API [115](#)
- CEE4DAS (Define Heap Allocation Strategy) bindable API [107](#)
- CEE9901 (generic failure) exception message [42](#)
- CEE9901 function check [41](#)
- CEECRHP (Create Heap) bindable API [107](#)
- CEECRHP bindable API [107](#)
- CEECSZT (Reallocate Storage) bindable API [107](#)
- CEEDOD (Retrieve Operational Descriptor Information) bindable API [103](#)
- CEEDSHP (Discard Heap) bindable API [106](#), [107](#)
- CEEFRST (Free Storage) bindable API [107](#)
- CEEGSI (Get String Information) bindable API [103](#)
- CEEGTST (Get Heap Storage) bindable API [107](#)
- CEEHDLR (Register User-Written Condition Handler) bindable API [43](#), [111](#)
- CEEHDLU (Unregister User-Written Condition Handler) bindable API [43](#)
- CEEMGET (Get Message) bindable API [118](#)
- CEEMKHP (Mark Heap) bindable API [106](#), [107](#)
- CEEMOUT (Dispatch Message) bindable API [118](#)
- CEEMRCR (Move Resume Cursor) bindable API [113](#)
- CEEMSG (Get, Format and Dispatch Message) bindable API [118](#)
- CEENCOD (Construct Condition Token) bindable API [115](#)
- CEERLHP (Release Heap) bindable API [106](#), [107](#)
- CEESGL (Signal Condition) bindable API

- CEESGL (Signal Condition) bindable API (*continued*)
 - condition token [115](#), [118](#)
 - description [41](#)
- CEETREC API [33](#), [37](#), [39](#)
- CEETSTA (Test for Omitted Argument) bindable API [101](#)
- Change Exception Message (QMHCHGEM) API [113](#)
- Change Module (CHGMOD) command [120](#)
- characteristics of teraspace [49](#)
- Check lock value [161](#)
- CHGMOD (Change Module) command [120](#)
- CICS
 - CL (control language) commands [188](#)
- CL (control language) command
 - CHGMOD (Change Module) [120](#)
 - RCLACTGRP (Reclaim Activation Group) [94](#)
 - RCLRSC (Reclaim Resources)
 - for ILE programs [94](#)
 - for OPM programs [93](#)
- Clear lock value [161](#)
- code optimization
 - errors [185](#)
 - levels [120](#)
 - performance
 - compared to original program model (OPM) [11](#)
 - levels [27](#)
 - module observability [120](#)
- coexistence with existing applications [8](#)
- command, CL
 - CALL (dynamic program call) [101](#)
 - CRTPGM (Create Program) [61](#)
 - CRTSRVPGM (Create Service Program) [61](#)
 - ENDCMTCTL (End Commitment Control) [124](#)
 - OPNDBF (Open Data Base File) [123](#)
 - OPNQRYF (Open Query File) [123](#)
 - RCLACTGRP (Reclaim Activation Group) [34](#)
 - RCLRSC (Reclaim Resources) [92](#)
 - STRCMTCTL (Start Commitment Control) [123](#), [124](#)
 - STRDBG (Start Debug) [119](#)
 - Update Program (UPDPGM) [85](#)
 - Update Service Program (UPDSRVPGM) [85](#)
- command, CL (control language)
 - CHGMOD (Change Module) [120](#)
 - RCLACTGRP (Reclaim Activation Group) [94](#)
 - RCLRSC (Reclaim Resources)
 - for ILE programs [94](#)
 - for OPM programs [93](#)
- commitment control
 - activation group [124](#)
 - commit operation [124](#)
 - commitment definition [124](#)
 - ending [125](#)
 - example [9](#)
 - rollback operation [124](#)
 - scope [124](#)
 - transaction [124](#)
- commitment definition [123](#), [124](#)
- Common Programming Interface (CPI) Communication, data management [124](#)
- component
 - reusable
 - benefit of ILE [8](#)
- condition
 - definition [45](#)
 - management (*continued*)
 - management (*continued*)
 - bindable APIs (application programming interfaces) [127](#), [129](#)
 - relationship to message [117](#)
- Condition ID component of condition token [116](#)
- condition token
 - Case component [116](#)
 - Condition ID component [116](#)
 - Control component [116](#)
 - definition [45](#), [115](#)
 - Facility ID component [116](#)
 - feedback code on call to bindable API [117](#)
 - Message Number component [116](#)
 - Message Severity component [116](#)
 - MsgNo component [116](#)
 - MsgSev component [116](#)
 - relationship to message [117](#)
 - Severity component [116](#)
 - testing [117](#)
- Construct Condition Token (CEENCOD) bindable API [115](#)
- control boundary
 - activation group
 - example [37](#)
 - default activation group example [38](#)
 - definition [37](#)
 - function check at [114](#)
 - unhandled exception at [114](#)
 - use [39](#)
- Control component of condition token [116](#)
- control file syntax for IPA [138](#)
- control flow
 - bindable APIs (application programming interfaces) [127](#)
- CPF9999 (function check) exception message [42](#)
- CPF9999 function check [41](#)
- Create Heap (CEECRHP) bindable API [107](#)
- Create Program (CRTPGM) command
 - ACTGRP (activation group) parameter
 - activation group creation [32](#)
 - program activation [30](#), [32](#)
 - ALWLIBUPD (Allow Library Update) [86](#)
 - ALWUPD (Allow Update) parameter [85](#), [86](#)
 - BNDDIR parameter [64](#)
 - compared to CRTSRVPGM (Create Service Program) command [61](#)
 - DETAIL parameter
 - *BASIC value [163](#)
 - *EXTENDED value [165](#)
 - *FULL value [166](#)
 - ENTMOD (entry module) parameter [70](#)
 - MODULE parameter [64](#)
 - output listing [163](#)
 - program creation [17](#)
 - service program activation [36](#)
- Create Service Program (CRTSRVPGM) command
 - ACTGRP (activation group) parameter
 - *CALLER value [94](#)
 - program activation [30](#), [32](#)
 - ALWLIBUPD (Allow Library Update) parameter [86](#)
 - ALWUPD (Allow Update) parameter [86](#)
 - BNDDIR parameter [64](#)
 - compared to CRTPGM (Create Program) command [61](#)
 - DETAIL parameter
 - *BASIC value [163](#)

- Create Service Program (CRTSRVPGM) command (*continued*)
 - DETAIL parameter (*continued*)
 - *EXTENDED value [165](#)
 - *FULL value [166](#)
 - EXPORT parameter [71](#)
 - MODULE parameter [64](#)
 - output listing [163](#)
 - service program activation [36](#)
 - SRCFILE (source file) parameter [71](#)
 - SRCMBR (source member) parameter [71](#)
- creation of
 - debug data [120](#)
 - module [88](#)
 - program [61](#), [88](#)
 - program activation [29](#)
 - service program [88](#)
- cross-reference listing
 - service program example [170](#)
- CRTPGM
 - BNDSRVPGM parameter [64](#)
- CRTPGM (Create Program) command
 - compared to CRTSRVPGM (Create Service Program) command [61](#)
 - DETAIL parameter
 - *BASIC value [163](#)
 - *EXTENDED value [165](#)
 - *FULL value [166](#)
 - ENTMOD (entry module) parameter [70](#)
 - output listing [163](#)
 - program creation [17](#)
- CRTSRVPGM
 - BNDSRVPGM parameter [64](#)
- CRTSRVPGM (Create Service Program) command
 - ACTGRP (activation group) parameter
 - *CALLER value [94](#)
 - compared to CRTPGM (Create Program) command [61](#)
 - DETAIL parameter
 - *BASIC value [163](#)
 - *EXTENDED value [165](#)
 - *FULL value [166](#)
 - EXPORT parameter [71](#)
 - output listing [163](#)
 - SRCFILE (source file) parameter [71](#)
 - SRCMBR (source member) parameter [71](#)
- cursor
 - handle [111](#)
 - resume [111](#)

D

- data compatibility [102](#)
- data links [123](#)
- data management scoping
 - activation group level [46](#), [124](#)
 - call level [46](#), [92](#)
 - commitment definition [123](#)
 - Common Programming Interface (CPI) Communication [124](#)
 - hierarchical file system [124](#)
 - job-level [47](#), [125](#)
 - local SQL (Structured Query Language) cursor [123](#)
 - open data link [123](#)
 - open file management [123](#)
 - open file operation [123](#)

- data management scoping (*continued*)
 - override [123](#)
 - remote SQL (Structured Query Language) connection [123](#)
 - resource [123](#)
 - rules [45](#)
 - SQL (Structured Query Language) cursors [123](#)
 - user interface manager (UIM) [123](#)
- data sharing
 - original program model (OPM) [12](#)
- date
 - bindable APIs (application programming interfaces) [128](#)
- debug data
 - creation [120](#)
 - definition [16](#)
 - removal [120](#)
- debug environment
 - ILE [119](#)
 - OPM [119](#)
- debug mode
 - addition of programs [119](#)
 - definition [119](#)
- debug support
 - ILE [121](#)
 - OPM [121](#)
- debugger
 - bindable APIs (application programming interfaces) [129](#)
 - CL (control language) commands [189](#)
 - considerations [119](#)
 - description [28](#)
- debugging
 - across jobs [120](#)
 - bindable APIs (application programming interfaces) [129](#)
 - CCSID [290](#) [121](#)
 - CCSID 65535 and device CHRID [290](#) [121](#)
 - CL (control language) commands [189](#)
 - error handling [121](#)
 - globalization
 - restriction [121](#)
 - ILE program [18](#)
 - module view [120](#)
 - observability [120](#)
 - optimization [120](#)
 - unmonitored exception [121](#)
- default activation group
 - control boundary example [38](#)
 - original program model (OPM) and ILE programs [32](#)
 - teraspace [32](#)
- default exception handling
 - compared to original program model (OPM) [42](#)
- default heap [105](#)
- Define Heap Allocation Strategy (CEE4DAS) bindable API [107](#)
- deletion
 - activation group [33](#)
- direct monitor
 - exception handler type [43](#), [111](#)
- Discard Heap (CEEDSHP) bindable API [106](#), [107](#)
- Dispatch Message (CEEMOUT) bindable API [118](#)
- DSM (dynamic screen manager)
 - bindable APIs (application programming interfaces) [130](#)
- duplicate symbol [65](#)
- dynamic binding
 - original program model (OPM) [12](#)

- dynamic program call
 - activation [101](#)
 - CALL CL (control language) command [101](#)
 - call stack [97](#)
 - definition [24](#)
 - examples [24](#)
 - original program model (OPM) [11](#), [101](#)
 - program activation [29](#)
 - service program activation [35](#)
- dynamic screen manager (DSM)
 - bindable APIs (application programming interfaces) [130](#)
- dynamic storage [105](#)

E

- Enabling program
 - collecting profiling data [132](#)
- enabling programs for teraspace [49](#)
- End Commitment Control (ENDCMTCTL) command [124](#)
- End Program Export (ENDPGMEXP) command [75](#)
- End Program Export (ENDPGMEXP), binder language [74](#)
- ENDCMTCTL (End Commitment Control) command [124](#)
- ENDPGMEXP (End Program Export), binder language [74](#)
- ENTMOD (entry module) parameter [70](#)
- entry point
 - compared to ILE program entry procedure (PEP) [16](#)
 - Extended Program Model (EPM) [12](#)
 - original program model (OPM) [11](#)
- EPM (Extended Program Model) [12](#)
- error
 - binder language [171](#)
 - during optimization [185](#)
- error handling
 - architecture [26](#), [39](#)
 - bindable APIs (application programming interfaces) [127](#), [129](#)
 - debug mode [121](#)
 - default action [42](#), [114](#)
 - language specific [41](#)
 - nested exception [114](#)
 - priority example [44](#)
 - recovery [41](#)
 - resume point [41](#)
- error message
 - MCH4439 [64](#)
- escape (*ESCAPE) exception message type [40](#)
- exception handler
 - priority example [44](#)
 - types [43](#)
- exception handling
 - architecture [26](#), [39](#)
 - bindable APIs (application programming interfaces) [127](#), [129](#)
 - debug mode [121](#)
 - default action [42](#), [114](#)
 - language specific [41](#)
 - nested exception [114](#)
 - priority example [44](#)
 - recovery [41](#)
 - resume point [41](#)
- exception management [111](#)
- exception message
 - C signal [41](#)
 - CEE9901 (generic failure) [42](#)

- exception message (*continued*)
 - CPF9999 (function check) [42](#)
 - debug mode [121](#)
 - function check (CPF9999) [42](#)
 - generic failure (CEE9901) [42](#)
 - handling [41](#)
 - ILE C raise() function [41](#)
 - percolation [42](#)
 - relationship of ILE conditions to [117](#)
 - sending [40](#)
 - types [40](#)
 - unmonitored [121](#)
- exception message architecture
 - error handling [39](#)
- export
 - definition [16](#)
 - order [65](#)
 - strong [70](#), [72](#), [170](#)
 - weak [70](#), [72](#), [170](#)
- EXPORT (Export Symbol) [76](#)
- EXPORT (Export Symbol), binder language [74](#)
- EXPORT parameter
 - service program signature [71](#)
 - used with SRCFILE (source file) and SRCMBR (source member) parameters [71](#)
- export symbol
 - wildcard character [76](#)
- Export Symbol (EXPORT), binder language [74](#)
- extended listing [165](#)
- Extended Program Model (EPM) [12](#)
- external message queue [40](#)

F

- Facility ID component of condition token [116](#)
- feedback code option
 - call to bindable API [117](#)
- file system, data management [124](#)
- Free Storage (CEEFRST) bindable API [107](#)
- full listing [166](#)
- function check
 - (CPF9999) exception message [42](#)
 - control boundary [114](#)
 - exception message type [41](#)

G

- generic failure (CEE9901) exception message [42](#)
- Get Heap Storage (CEEGTST) bindable API [107](#)
- Get Message (CEEMGET) bindable API [118](#)
- Get String Information (CEEGSI) bindable API [103](#)
- Get, Format and Dispatch Message (CEEMSG) bindable API [118](#)
- globalization restriction for debugging [121](#)

H

- handle cursor
 - definition [111](#)
- heap
 - allocation strategy [106](#)
 - characteristics [105](#)
 - default [105](#)

- heap (*continued*)
 - definition [105](#)
 - user-created [106](#)
- heap allocation strategy [106](#)
- Heap support [107](#)
- history of ILE [11](#)
- HLL specific
 - error handling [41](#)
 - exception handler [44](#), [111](#)
 - exception handling [41](#)

I

- ILE
 - basic concepts [15](#)
 - compared to
 - Extended Program Model (EPM) [12](#)
 - original program model (OPM) [12](#), [15](#)
 - definition [7](#)
 - history [11](#)
 - introduction [7](#)
 - program structure [15](#)
- ILE condition handler
 - exception handler type [43](#), [111](#)
- import
 - definition [16](#)
 - procedure [18](#)
 - resolved and unresolved [64](#)
 - strong [72](#)
 - weak [72](#)
- interlanguage data compatibility [102](#)
- interprocedural analysis
 - IPA control file syntax [138](#)
 - partitions created by [141](#)
 - restrictions and limitations [140](#)
 - usage notes [140](#)

J

- job
 - multiple applications running in same [91](#)
- job message queue [40](#)
- job-level scoping [47](#)

L

- language
 - procedure-based
 - characteristics [13](#)
- language interaction
 - consistent error handling [42](#)
 - control [10](#)
 - data compatibility [102](#)
- language specific
 - error handling [41](#)
 - exception handler [44](#), [111](#)
 - exception handling [41](#)
- level check parameter on STRPGMEXP command [75](#)
- level number [92](#)
- Licensed Internal Code options (LICOPTs)
 - currently defined options [143](#)
 - displaying [148](#)
 - release compatibility [148](#)

- Licensed Internal Code options (LICOPTs) (*continued*)
 - restrictions [147](#)
 - specifying [147](#)
 - syntax [147](#)
- LICOPTs (Licensed Internal Code options) [143](#)
- listing, binder
 - basic [163](#)
 - extended [165](#)
 - full [166](#)
 - service program example [169](#)

M

- Mark Heap (CEEMKHP) bindable API [106](#), [107](#)
- math
 - bindable APIs (application programming interfaces) [128](#)
- maximum width
 - file for SRCFILE (source file) parameter [72](#)
- MCH4439 error message [64](#)
- message
 - bindable API feedback code [117](#)
 - exception types [40](#)
 - queue [40](#)
 - relationship of ILE conditions to [117](#)
- message handling
 - bindable APIs (application programming interfaces) [129](#)
- Message Number (MsgNo) component of condition token [116](#)
- message queue
 - job [40](#)
- Message Severity (MsgSev) component of condition token [116](#)
- modularity
 - benefit of ILE [7](#)
- module object
 - CL (control language) commands [187](#)
 - creation tips [88](#)
 - description [16](#)
- MODULE parameter on UPDPGM command [86](#)
- MODULE parameter on UPDSRVPGM command [86](#)
- module replaced by module
 - fewer exports [87](#)
 - fewer imports [87](#)
 - more exports [88](#)
 - more imports [87](#)
- module replacement [85](#)
- module view
 - debugging [120](#)
- Move Resume Cursor (CEEMRCR) bindable API [113](#)
- multiple applications running in same job [91](#)

N

- nested exception [114](#)
- notify (*NOTIFY) exception message type [40](#)

O

- observability [120](#)
- ODP (open data path)
 - scoping [45](#)
- omitted argument [101](#)
- Open Data Base File (OPNDBF) command [123](#)

- open data path (ODP)
 - scoping [45](#)
- open file operations [123](#)
- Open Query File (OPNQRYF) command [123](#)
- operational descriptor [102](#), [103](#)
- OPM (original program model)
 - activation group [32](#)
 - binding [12](#)
 - characteristics [12](#)
 - compared to ILE [15](#), [17](#)
 - data sharing [12](#)
 - default exception handling [42](#)
 - description [11](#)
 - dynamic binding [12](#)
 - dynamic program call [101](#)
 - entry point [11](#)
 - exception handler types [43](#)
 - program entry point [11](#)
- OPNDBF (Open Data Base File) command [123](#)
- OPNQRYF (Open Query File) command [123](#)
- optimization
 - benefit of ILE [11](#)
 - code
 - levels [27](#)
 - module observability [120](#)
 - errors [185](#)
 - interprocedural analysis [136](#)
 - levels [120](#)
- optimization technique
 - profiling program [131](#)
- optimizing translator [11](#), [27](#)
- optimizing your programs with IPA [138](#)
- ordering concerns
 - storage access [161](#)
- original program model (OPM)
 - activation group [32](#)
 - binding [12](#)
 - characteristics [12](#)
 - compared to ILE [15](#), [17](#)
 - data sharing [12](#)
 - default exception handling [42](#)
 - description [11](#)
 - dynamic binding [12](#)
 - dynamic program call [11](#), [101](#)
 - entry point [11](#)
 - exception handler types [43](#)
 - program entry point [11](#)
- output listing
 - Create Program (CRTPGM) command [163](#)
 - Create Service Program (CRTSRVPGM) command [163](#)
 - Update Program (UPDPGM) command [163](#)
 - Update Service Program (UPDSRVPGM) command [163](#)
- override, data management [123](#)

P

- parameters on UPDPGM and UPDSRVPGM commands [86](#)
- partitions created by IPA [141](#)
- passing arguments
 - between languages [102](#)
 - by reference [99](#)
 - by value directly [99](#)
 - by value indirectly [99](#)
 - in mixed-language applications [102](#)

- passing arguments (*continued*)
 - omitted arguments [101](#)
 - to procedures [99](#)
 - to programs [101](#)
- PEP (program entry procedure)
 - call stack example [97](#)
 - definition [16](#)
 - specifying with CRTPGM (Create Program) command [70](#)
- percolation
 - exception message [42](#)
- performance
 - optimization
 - benefit of ILE [11](#)
 - errors [185](#)
 - levels [27](#), [120](#)
 - module observability [120](#)
- pitfalls
 - shared storage [157](#)
- pointer
 - comparing 8- and 16-byte [53](#)
 - conversions in teraspace-enabled programs [54](#)
 - lengths [53](#)
 - support in APIs [56](#)
 - support in C and C++ compilers [54](#)
- priority
 - exception handler example [44](#)
- procedure
 - definition [12](#), [16](#)
 - passing arguments to [99](#)
- procedure call
 - bindable APIs (application programming interfaces) [129](#)
 - compared to program call [24](#), [97](#)
 - static
 - call stack [97](#)
 - definition [24](#)
 - examples [25](#)
- procedure pointer call [97](#), [99](#)
- procedure-based language
 - characteristics [13](#)
- profiling program [132](#)
- profiling types [131](#)
- program
 - access [70](#)
 - activation [29](#)
 - CL (control language) commands [187](#)
 - comparison of ILE and original program model (OPM) [17](#)
 - creation
 - examples [66](#), [68](#)
 - process [61](#)
 - tips [88](#)
 - passing arguments to [101](#)
- program activation
 - activation [29](#)
 - creation [29](#)
 - dynamic program call [29](#)
- program call
 - bindable APIs (application programming interfaces) [129](#)
 - call stack [97](#)
 - compared to procedure call [97](#)
 - definition [24](#)
 - examples [24](#)
- program entry point
 - compared to ILE program entry procedure (PEP) [16](#)
 - Extended Program Model (EPM) [12](#)

- program entry point (*continued*)
 - original program model (OPM) [11](#)
- program entry procedure (PEP)
 - call stack example [97](#)
 - definition [16](#)
 - specifying with CRTPGM (Create Program) command [70](#)
- program isolation in activation groups [30](#)
- program level parameter on STRPGMEXP command [75](#)
- program structure [15](#)
- program update
 - module replaced by module
 - fewer exports [87](#)
 - fewer imports [87](#)
 - more exports [88](#)
 - more imports [87](#)
- Promote Message (QMHPRMM) API [113](#)

Q

- QCAPCMD API [94](#)
- QMHCHGEM (Change Exception Message) API [113](#)
- QMHPRMM (Promote Message) API [113](#)
- QMHSNDPM (Send Program Message) API [41](#), [111](#)
- QUSEADPAUT (use adopted authority) system value
 - description [62](#)
 - risk of changing [62](#)

R

- race conditions [161](#)
- RCLACTGRP (Reclaim Activation Group) command [34](#), [94](#)
- RCLRSC (Reclaim Resources) command
 - for ILE programs [94](#)
 - for OPM programs [93](#)
- Reallocate Storage (CEEZST) bindable API [107](#)
- Reclaim Activation Group (RCLACTGRP) command [34](#), [94](#)
- Reclaim Resources (RCLRSC) command
 - for ILE programs [94](#)
 - for OPM programs [93](#)
- recovery
 - exception handling [41](#)
- register exception handler [43](#)
- Register User-Written Condition Handler (CEEHDLR)
 - bindable API [43](#), [111](#)
- Release Heap (CEERLHP) bindable API [106](#), [107](#)
- removal of debug data [120](#)
- resolved import [64](#)
- resolving symbol
 - description [63](#)
 - examples [66](#), [68](#)
- resource control [8](#)
- resource isolation in activation groups [30](#)
- resource, data management [123](#)
- restriction
 - debugging
 - globalization [121](#)
- resume cursor
 - definition [111](#)
 - exception recovery [41](#)
- resume point
 - exception handling [41](#)
- Retrieve Binder Source (RTVBNDSRC) command [71](#)

- Retrieve Operational Descriptor Information (CEEDOD)
 - bindable API [103](#)
- reuse
 - activation group [33](#)
 - components [8](#)
- rollback operation
 - commitment control [124](#)
- RPLLIB parameter on UPDPGM command [86](#)
- RPLLIB parameter on UPDSRVPGM command [86](#)
- runtime services [8](#)

S

- scope
 - commitment control [124](#)
- scoping, data management
 - activation group level [46](#), [124](#)
 - call level [46](#), [92](#)
 - commitment definition [123](#)
 - Common Programming Interface (CPI) Communication [124](#)
 - hierarchical file system [124](#)
 - job-level [47](#), [125](#)
 - local SQL (Structured Query Language) cursor [123](#)
 - open data link [123](#)
 - open file management [123](#)
 - open file operation [123](#)
 - override [123](#)
 - remote SQL (Structured Query Language) connection [123](#)
 - resource [123](#)
 - rules [45](#)
 - SQL (Structured Query Language) cursors [123](#)
 - user interface manager (UIM) [123](#)
- Send Program Message (QMHSNDPM) API [41](#), [111](#)
- sending
 - exception message [40](#)
- service program
 - activation [35](#), [98](#)
 - binder listing example [169](#)
 - CL (control language) commands [188](#)
 - creation tips [88](#)
 - definition [19](#)
 - description [13](#)
 - signature [71](#), [74](#)
 - static procedure call [98](#)
- Severity component of condition token [116](#)
- shared open data path (ODP) example [9](#)
- shared storage
 - pitfalls [157](#)
- shared storage access ordering [157](#)
- shared storage synchronization [157](#)
- Signal Condition (CEESGL) bindable API
 - condition token [115](#), [118](#)
 - description [41](#)
- signature
 - EXPORT parameter [71](#)
- signature parameter on STRPGMEXP command [75](#)
- single-heap support [106](#)
- single-level storage model [50](#)
- source debugger
 - bindable APIs (application programming interfaces) [129](#)
 - CL (control language) commands [189](#)
 - considerations [119](#)

- source debugger (*continued*)
 - description [28](#)
- specifying Licensed Internal Code options [147](#)
- SQL (Structured Query Language)
 - CL (control language) commands [188](#)
 - connections, data management [123](#)
- SRCFILE (source file) parameter
 - file
 - maximum width [72](#)
- SRCMBR (source member) parameter [71](#)
- SRVPGMLIB on UPDSRVPGM command [86](#)
- stack, call [97](#)
- Start Commitment Control (STRCMTCTL) command [123](#), [124](#)
- Start Debug (STRDBG) command [119](#)
- Start Program Export (STRPGMEXP) command [75](#)
- Start Program Export (STRPGMEXP), binder language [74](#)
- static procedure call
 - call stack [97](#)
 - definition [24](#)
 - examples [25](#), [98](#)
 - service program [98](#)
 - service program activation [36](#)
- static storage [105](#)
- static variable [29](#), [91](#)
- status (*STATUS) exception message type [40](#)
- storage
 - shared [157](#)
- storage access
 - ordering concerns [161](#)
- storage access ordering concerns [161](#)
- storage management
 - automatic storage [105](#)
 - bindable APIs (application programming interfaces) [129](#)
 - dynamic storage [105](#)
 - heap [105](#)
 - static storage [92](#), [105](#)
- storage model
 - single-level storage [50](#)
 - teraspaces [50](#)
- storage synchronization, shared [157](#)
- storage synchronizing
 - actions [159](#)
- storage synchronizing actions [159](#)
- STRCMTCTL (Start Commitment Control) command [123](#), [124](#)
- STRDBG (Start Debug) command [119](#)
- strong export [70](#), [72](#), [170](#)
- STRPGMEXP (Start Program Export), binder language [74](#)
- structure of ILE program [15](#)
- Structured Query Language (SQL)
 - CL (control language) commands [188](#)
 - connections, data management [123](#)
- support for original program model (OPM) and ILE APIs [103](#)
- symbol name
 - wildcard character [76](#)
- symbol resolution
 - definition [63](#)
 - duplicate [65](#)
 - examples [66](#), [68](#)
- syntax rules for Licensed Internal Code options [147](#)
- system value
 - QUSEADPAUT (use adopted authority)
 - description [62](#)
 - risk of changing [62](#)
 - use adopted authority (QUSEADPAUT)

- system value (*continued*)
 - use adopted authority (QUSEADPAUT) (*continued*)
 - description [62](#)
 - risk of changing [62](#)
- system-named activation group [32](#), [34](#)

T

- teraspaces
 - allowed storage model for program types [50](#)
 - characteristics [49](#)
 - choosing storage model [50](#)
 - converting programs or service programs to use [52](#)
 - enabling in your programs [49](#)
 - interaction of single-level storage and teraspaces storage models [51](#)
 - pointer conversions [54](#)
 - pointer support in interfaces [56](#)
 - selecting compatible activation group [51](#)
 - specifying as storage model [50](#)
 - usage notes [55](#)
 - using 8-byte pointers [53](#)
- teraspaces storage model [50](#)
- Test for Omitted Argument (CEETSTA) bindable API [101](#)
- testing condition token [117](#)
- thread local storage (TLS) [108](#)
- time
 - bindable APIs (application programming interfaces) [128](#)
- tip
 - module, program and service program creation [88](#)
- transaction
 - commitment control [124](#)
- translator
 - code optimization [11](#), [27](#)

U

- UEP (user entry procedure)
 - call stack example [97](#)
 - definition [16](#)
- unhandled exception
 - default action [42](#)
- unmonitored exception [121](#)
- Unregister User-Written Condition Handler (CEEHDLU)
 - bindable API [43](#)
- unresolved import [64](#)
- Update Program (UPDPGM) command [85](#)
- Update Service Program (UPDSRVPGM) command [85](#)
- UPDPGM command
 - BNDDIR parameter [86](#)
 - BNDSRVPGM parameter [86](#)
 - MODULE parameter [86](#)
 - RPLLIB parameter [86](#)
- UPDSRVPGM command
 - BNDDIR parameter [86](#)
 - BNDSRVPGM parameter [86](#)
 - MODULE parameter [86](#)
 - RPLLIB parameter [86](#)
- use adopted authority (QUSEADPAUT) system value
 - description [62](#)
 - risk of changing [62](#)
- user entry procedure (UEP)
 - call stack example [97](#)

- user entry procedure (UEP) (*continued*)
 - definition [16](#)
- user interface manager (UIM), data management [123](#)
- user-named activation group
 - deletion [34](#)
 - description [32](#), [91](#)

V

- variable
 - static [29](#), [91](#)

W

- watch support [121](#)
- weak export [70](#), [72](#), [170](#)
- wildcard character for export symbol [76](#)



Product Number: 5770-SS1

SC41-5606-12

