AS/400e

IBM

ILE Concepts

Version 4

AS/400e

ILE Concepts

Version 4

> **Note**
>
> Before using this information and the product it supports, be sure to read the information in ″Appendix D. Notices″.

Contents

About ILE Concepts (SC41-5606)

This book describes concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the OS/400 licensed program. Topics covered include module creation, binding, the running and debugging of programs, and exception handling.

The concepts described in this book 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 book also describes OS/400 functions that directly pertain to all ILE languages. In particular, common information on binding, message handling, and debugging are explained.

This book does not describe migration from an existing AS/400 language to an ILE language. That information is contained in each ILE high-level language (HLL) programmer's guide.

Who should read this book

You should read this book if:

- You are a software vendor developing applications or software tools
- You are experienced in developing mixed-language applications on the AS/400 system
- You are not familiar with the AS/400 system but have application programming experience on other systems
- Your programs share common procedures, and when you update or enhance those procedures, you have to re-create the programs that use them

If you are an AS/400 application programmer who writes primarily in one language, you should read the first four chapters of this book for a general understanding of ILE and its benefits. The programmer's guide for that ILE language can then be sufficient for application development.

AS/400 Operations Navigator

AS/400 Operations Navigator is a powerful graphical interface for Windows clients. With AS/400 Operations Navigator, you can manage and administer your AS/400 systems from your Windows desktop.

You can use Operations Navigator to manage communications, printing, database, security, and other system operations. Operations Navigator includes Management Central for managing multiple AS/400 systems centrally.

Figure 1 on page viii shows an example of the Operations Navigator display:

AS/400 Operations Navigator

File Edit View Options Help

Last changed: 8/12/99 5:57 AM

Central System. System01

Management Central (System01): Task Activity   Started by: All

Management Central (System 01)
  Task Activity
  Scheduled Tasks
  Definitions
    Command
    Package
  Monitors
  AS/400 Endpoint Systems
  AS/400 System Groups
    Chicago branches
My AS/400 Connections
  Mahlik
    Basic Operations
    Job Management
    Configuration and Service
      Hardware Inventory
      Software Inventory
      Fixes Inventory
      Collection Services
    Network
    Security
    Users and Groups
    Database
    File Systems
    Backup
    Application Development

Task	Type	Status	Systems and Groups
Collect Inventory (4)	Inventory	Completed	System 01

For Help, press F1

Figure 1. AS/400 Operations Navigator Display

This new interface has been designed to make you more productive and is the only user interface to new, advanced features of OS/400. Therefore, IBM recommends that you use AS/400 Operations Navigator, which has online help to guide you. While this interface is being developed, you may still need to use an emulator such as PC5250 to do some of your tasks.

Installing Operations Navigator

To use AS/400 Operations Navigator, you must have Client Access installed on your Windows PC. For help in connecting your Windows PC to your AS/400 system, consult *Client Access Express for Windows - Setup*, SC41-5507-01.

AS/400 Operations Navigator is a separately installable component of Client Access that contains many subcomponents. If you are installing for the first time and you use the **Typical** installation option, the following options are installed by default:

- Operations Navigator base support
- Basic operations (messages, printer output, and printers)

To select the subcomponents that you want to install, select the **Custom** installation option. (After Operations Navigator has been installed, you can add subcomponents by using Client Access Selective Setup.)

After you install Client Access, double-click the **AS400 Operations Navigator** icon on your desktop to access Operations Navigator and create an AS/400 connection.

Prerequisite and related information

Use the AS/400 Information Center as your starting point for looking up AS/400 technical information. You can access the Information Center from the AS/400e Information Center CD-ROM (English version: *SK3T-2027*) or from one of these Web sites:

```
http://www.as400.ibm.com/infocenter
http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
```

The AS/400 Information Center contains advisors and important topics such as CL commands, system application programming interfaces (APIs), logical partitioning, clustering, Java, TCP/IP, Web serving, and secured networks. It also contains Internet links to Web sites such as the AS/400 Online Library, AS/400 redbooks, and the AS/400 Technical Studio.

For a list of related publications, see the "Bibliography" on page 183.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other AS/400 documentation, fill out the readers' comment form at the back of this book.

- If you prefer to send comments by mail, use the readers' comment form with the address that is printed on the back. If you are mailing a readers' comment form from a country other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments by FAX, use either of the following numbers:
  - United States, Canada, and Puerto Rico: 1-800-937-3430
  - Other countries: 1-507-253-5192
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:

    RCHCLERK@us.ibm.com

    IBMMAIL, to IBMMAIL(USIB56RZ)
  - Comments on the AS/400 Information Center:

    RCHINFOC@us.ibm.com

Be sure to include the following:
- The name of the book.
- The publication number of the book.
- The page number or topic to which your comment applies.

Chapter 1. Integrated Language Environment® Introduction

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

Wherever possible, information is presented from the perspective of an RPG or COBOL programmer and is described in terms of existing AS/400* features.

What Is ILE?

ILE is a new set of tools and associated system support designed to enhance program development on the AS/400® system.

The capabilities of this new model can be exploited only by programs produced by the new ILE family of compilers. That family includes ILE RPG/400*, ILE COBOL/400*, ILE C/400*, 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 run-time 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 calling programs. 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, 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 migrating of code from other platforms

  Programs written on other platforms, such as UNIX**, are often modular. Those modules can be migrated to the AS/400 system and incorporated into an ILE program.

Reusable Components

ILE allows you to select packages of routines that can be blended into your own programs. Routines written in any ILE language can be used by all AS/400 ILE compiler users. The fact that programmers can write in the language of their choice ensures you 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 Run-Time 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 *CL and APIs* section that is found in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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 "Chapter 9. Debugging Considerations" on page 117.

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 on the AS/400, a programmer decided 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 AS/400 depends on a combination of the following:

   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/400® programs (PRGA, PRGB, and PRGC) form a small application in which PRGA calls PRGB, which in turn calls PRGC (see Figure 2). Under the rules of ILE COBOL/400, 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.

RV3W027-1
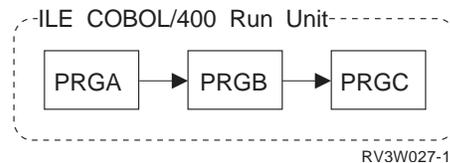
Figure 2. Three ILE COBOL/400 Programs in a Run Unit

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 3). An RPG program expects that its variables, files, and other resources remain intact until the program returns with the last-record (LR) indicator on.

RV3W028-1
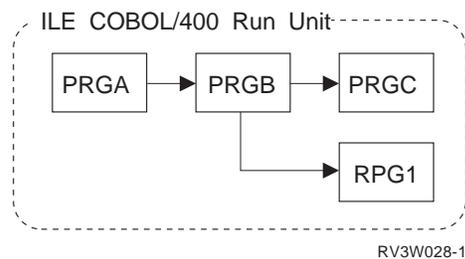
Figure 3. Three ILE COBOL/400 Programs and One ILE RPG/400® Program in a Run Unit

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 4 on page 6). An ILE COBOL/400 run unit itself is an activation group.
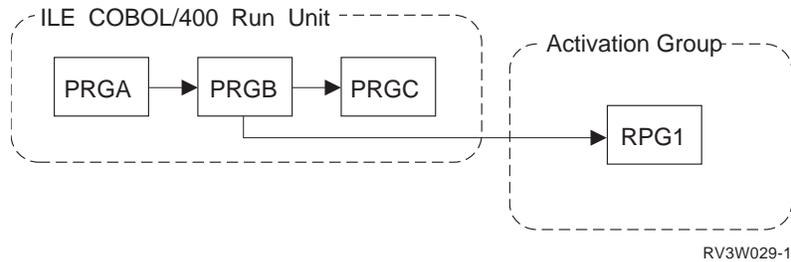
Figure 4. ILE RPG/400 Program in a Separate Activation Group

For information about the differences between an OPM run unit and an ILE run unit, see the *ILE COBOL for AS/400 Programmer's Guide*.

Better Code Optimization

The ILE translator does many more types of optimization than the original program model (OPM) translator does. Although each compiler does some optimization, the majority of the optimization on the AS/400 is done by the translator.

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.

Better Environment for C

C is a popular language for tool builders. Because of this, a better C language means that more and more of the latest application development tools are migrated to the AS/400. For you, this means a greater choice of:

    CASE tools

    Fourth-generation languages (4GLs)

    Additional programming languages

    Editors

    Debuggers

Foundation for the Future

The benefits and functions that ILE provides will be even more important in the future. Future ILE compilers will offer significant enhancements. As we move into object-oriented programming languages and visual programming tools, the need for ILE becomes even more apparent.

Increasingly, programming methods rely on a highly modularized approach. Applications are built by combining thousands of small reusable components to form the completed application. If these components cannot transfer control among themselves quickly, the resulting application cannot work.

What Is the History of ILE?

ILE is a stage in the evolution of OS/400* 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 Version 1 Release 2, the Extended Program Model (EPM) was introduced.

Original Program Model Description

Application developers on the AS/400 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 provided by the OS/400® to create and run a program 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 OS/400 encounters a call request. At run time, 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 5 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.

The broken line forming the OPM boundary indicates that OPM is an integral part of OS/400. 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, to perform string manipulations, or to send messages.
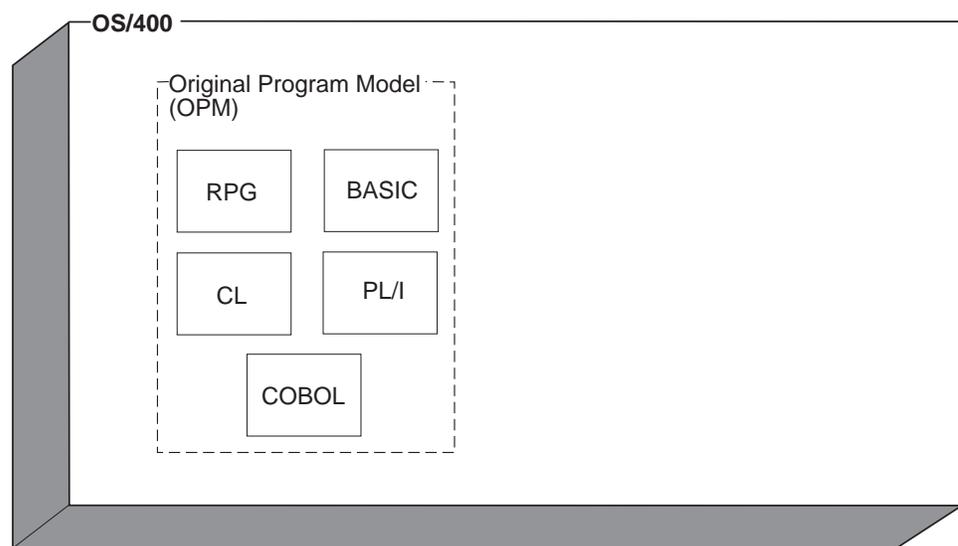
RV2W976-2

Figure 5. Relationship of OPM to OS/400

Principal Characteristics of OPM

The following list identifies the principal characteristics of OPM:

- Good for traditional RPG and COBOL programs

  OPM is ideal for supporting traditional RPG and COBOL programs, that is, relatively large, multifunction programs.

- 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 run time, 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 it's 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 that define procedures with local variables to run on an AS/400, OPM was enhanced. These enhancements are called the **Extended Program Model (EPM)**. As shown in Figure 6 on page 9, EPM was created to support languages like Pascal. Along with the base OPM support, EPM provides the ability to call procedures located in other programs. Because EPM uses the functions of OPM, some procedure calls in EPM turn into dynamic calls to the program containing the procedure. Conceptually, the entry point of the called EPM program provides the following functions:

> Initializes the program variables
>
> Calls the identified procedure

The system support to help resolve the procedure calls to the appropriate programs is provided by the Set Program Information (SETPGMINF) command.

Although EPM is a different programming model from OPM, it is closely tied to OPM. EPM is built as an additional layer above the AS/400 high-level machine interface. Most of the functions associated with OPM also apply to EPM.

In the following chapters of this manual, the term OPM means both OPM and EPM. Functions that apply only to EPM are specifically qualified with the term EPM.

RV3W103-0
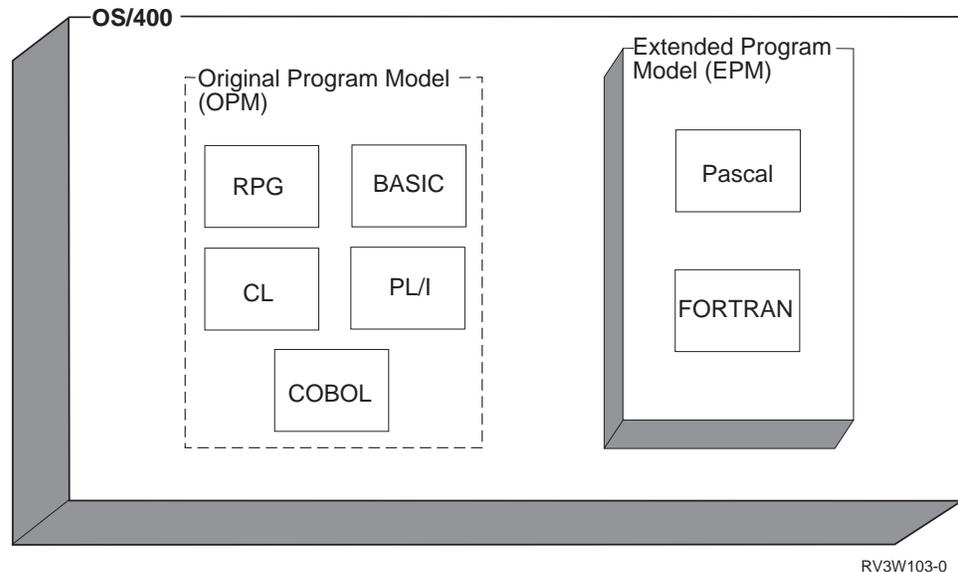
Figure 6. Relationship of OPM and EPM to OS/400

The shaded area around the EPM block indicates that, unlike OPM, EPM is not incorporated into OS/400. Rather, it is a layer on top of the operating system. It provides the additional support required for procedure-based languages.

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 12.

- Multiple entry points

  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 Windows** or OS/2*. Service programs are discussed in greater detail in "Service Program" on page 15.

- Frequent calls

  Procedure-based languages are by nature very call intensive. Although EPM provides some functions to minimize the overhead of calls, procedure calls between separately compiled units still have a relatively high overhead. ILE improves this type of call significantly.

Integrated Language Environment Description

As Figure 7 shows, ILE is tightly integrated into OS/400, 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.
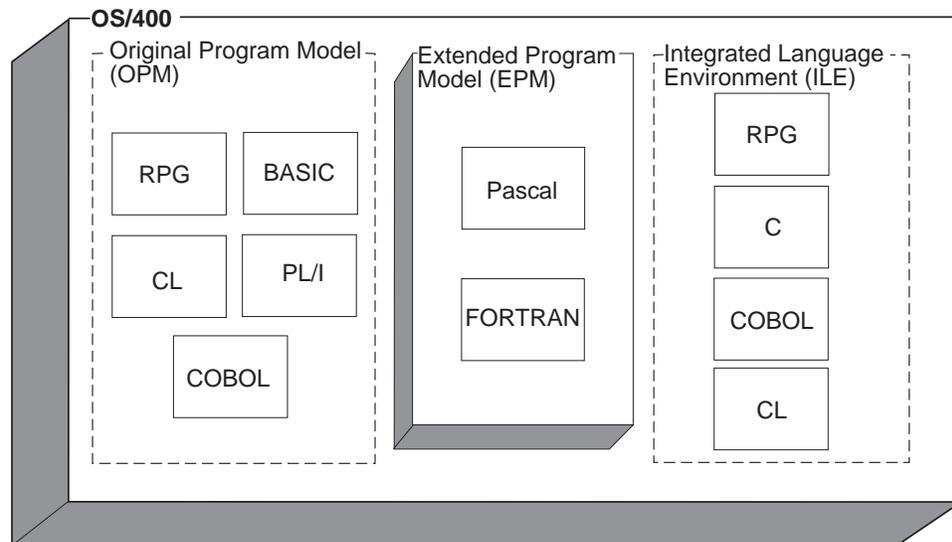
RV3W026-1

Figure 7. Relationship of OPM, EPM, and ILE to OS/400

Chapter 2. ILE Basic Concepts

Table 1 compares and contrasts the original program model (OPM) and the Integrated Language Environment (ILE) model. This chapter 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 8).

RV2W1003-2
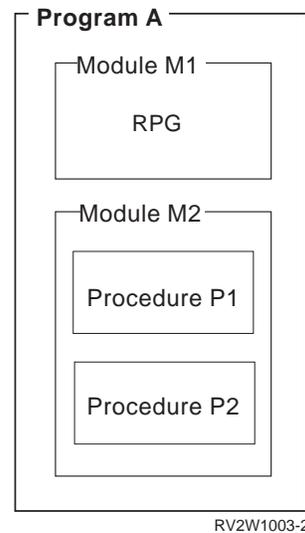
Figure 8. Structure of an ILE Program

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/400® 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/400, ILE COBOL/400, and ILE C/400 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 9 on page 13 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.
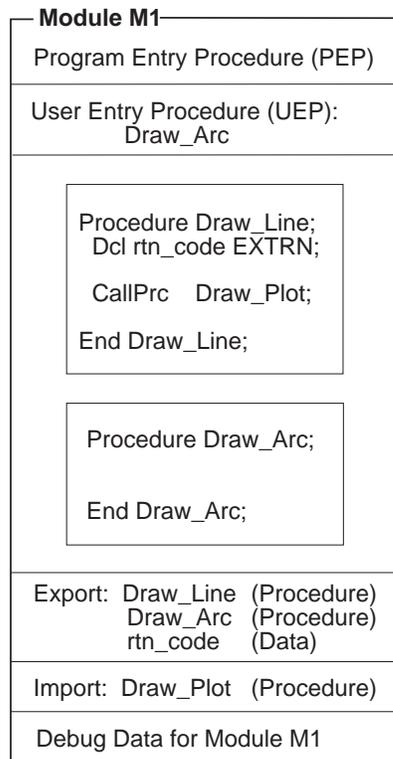
```
┌─ Module M1 ──────────────────────┐
│  Program Entry Procedure (PEP)    │
├───────────────────────────────────┤
│  User Entry Procedure (UEP):      │
│           Draw_Arc                │
│                                   │
│   ┌─────────────────────────────┐ │
│   │ Procedure Draw_Line;        │ │
│   │   Dcl rtn_code EXTRN;        │ │
│   │                             │ │
│   │   CallPrc   Draw_Plot;       │ │
│   │                             │ │
│   │ End Draw_Line;              │ │
│   └─────────────────────────────┘ │
│                                   │
│   ┌─────────────────────────────┐ │
│   │ Procedure Draw_Arc;         │ │
│   │                             │ │
│   │ End Draw_Arc;               │ │
│   └─────────────────────────────┘ │
│                                   │
├───────────────────────────────────┤
│ Export: Draw_Line  (Procedure)    │
│         Draw_Arc   (Procedure)    │
│         rtn_code   (Data)         │
├───────────────────────────────────┤
│ Import: Draw_Plot  (Procedure)    │
├───────────────────────────────────┤
│ Debug Data for Module M1          │
└───────────────────────────────────┘
                        RV3W104-0
```

Figure 9. 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 10 on page 15 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 new ILE debugger. Procedures from modules M2 and M4 cannot be debugged by using the new ILE debugger.

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

```
┌─ *PGM (PGMEXAMP) ──────────────────────────┐
│ Program Entry Procedure (Use PEP in module M3)       │
│ User Entry Procedure:      (Use P3 in module M3)     │
│                                                       │
│  ┌─ Module M1 ──────┐    ┌─ Module M2 ──────┐        │
│  │  Procedure P1;    │    │ PEP               │        │
│  │                   │    │ UEP: P2           │        │
│  │   DCL D EXTRN;     │    ├───────────────────┤        │
│  │                   │    │ Procedure P2;      │        │
│  │   CallPrc print;   │    │  CallPrc  P1;      │        │
│  │                   │    │                    │        │
│  │  End P1;          │    │  CallPrc  P4;      │        │
│  ├───────────────────┤    │                    │        │
│  │  Debug Data       │    │ End P2;            │        │
│  └───────────────────┘    └────────────────────┘        │
│                                                       │
│  ┌─ Module M3 ──────┐    ┌─ Module M4 ──────┐        │
│  │ PEP               │    │ Procedure P4;     │        │
│  │ UEP:  P3          │    │                    │        │
│  │                   │    │   DCL X REAL;      │        │
│  ├───────────────────┤    │                    │        │
│  │ Procedure P3;      │    │   D=SIN(X);        │        │
│  │   CallPrc P2;      │    │                    │        │
│  │ End P3;            │    │                    │        │
│  ├───────────────────┤    │ End P4;            │        │
│  │  Debug Data       │    └────────────────────┘        │
│  └───────────────────┘                                │
│                                                       │
│ Internally resolved imports: P1, P2, P4, D            │
│                                                       │
│ Used PEP:   Defined in module M3                      │
│        UEP:   Procedure P3 in module M3               │
│                                                       │
│ Externally resolved imports:                          │
│           print in *LIBL/PRINTS                       │
│           SIN in MATHLIB/MATHFUNC                     │
│                                                       │
└───────────────────────────────────────────────────────┘
```
RV2W980-5

Figure 10. 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".
- 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 may need; hence the name service program. An example of a set of service programs provided by OS/400 are the run-time procedures for a language. These run-time procedures often include such items as mathematical procedures and common input/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 64.

Figure 11 on page 17 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 10 on page 15. 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 called dynamically.
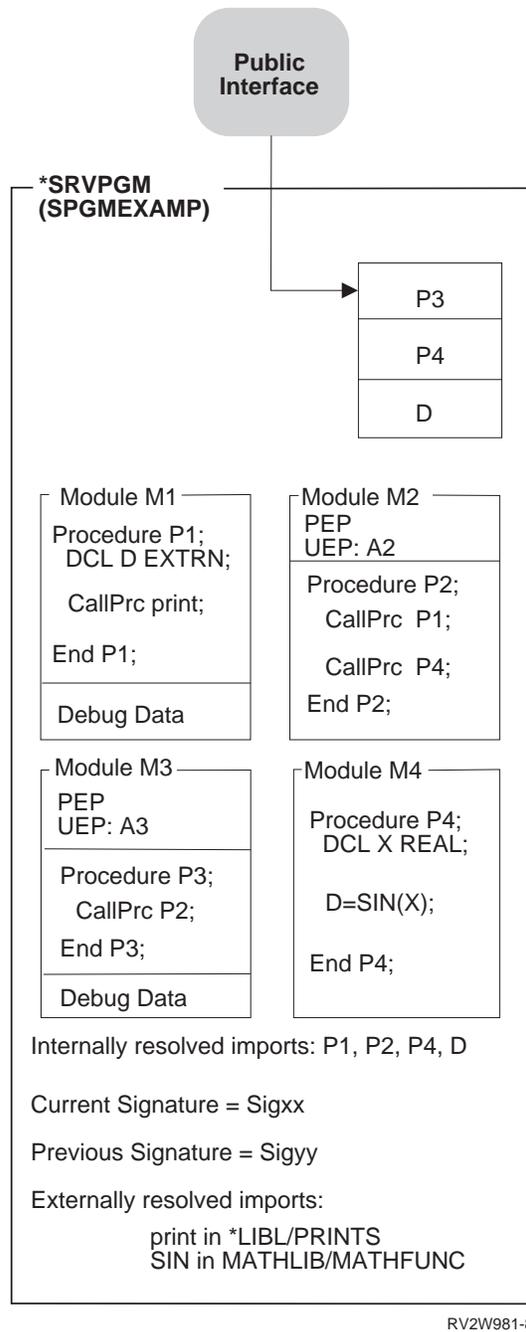
Figure 11. Conceptual View of an ILE Service Program

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 runnable procedures and data items.
- 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 63.

Binding Directory

A **binding directory** contains 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 packaging the modules or service programs that you may need when creating your own ILE program or service program. For example, one binding directory may contain 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 contains, 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 "Appendix C. CL Commands Used with ILE Objects" on page 175.

Figure 12 on page 19 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 12. 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 <u>and</u> if the named object provides an export to satisfy the unresolved import request.

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 11 on page 17, 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 11 on page 17 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 run time, 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 13 and Figure 14 on page 21 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.
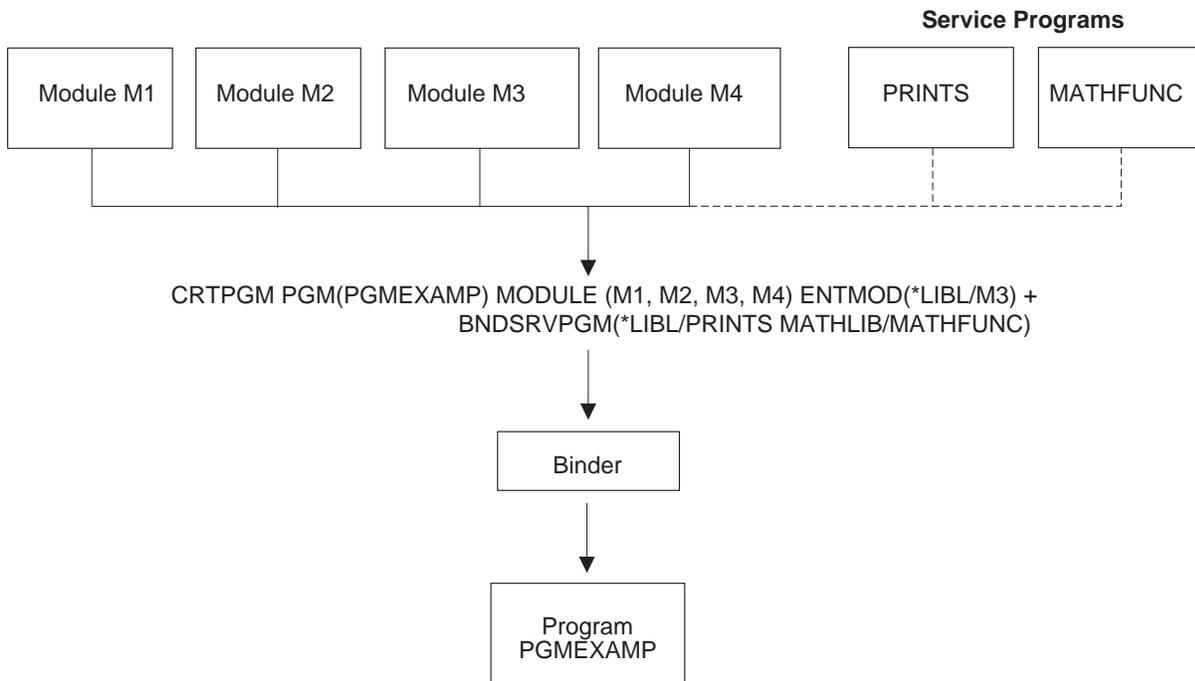
Figure 13. Creation of an ILE Program. The broken line indicates that the service programs are bound by reference instead of being bound by copy.

CRTSRVPGM SRVPGM(SPGMEXAMP) MODULE (M1, M2, M3, M4) EXPORT(*SRCFILE) +
SRCFILE(*LIBL/QSRVSRC) SRCMBR(*SRVPGM) BNDSRVPGM(*LIBL/PRINTS MATHLIB/MATHFUNC)

RV3W030-1
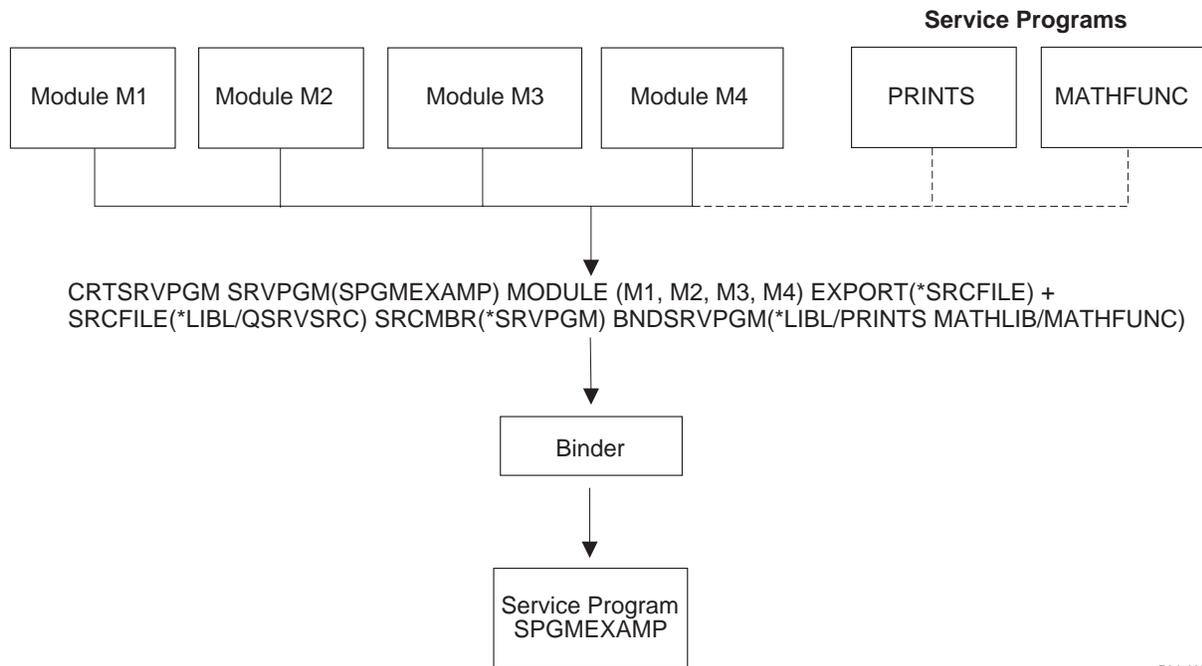
Figure 14. Creation of a Service Program. The broken line indicates that the service programs are bound by reference instead of being bound by copy.

For additional information on creating an ILE program or service program, see "Chapter 4. Program Creation Concepts" on page 51.

Calls to Programs and Procedures

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.

The binder can handle a procedure name that is up to 256 characters long. 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. Dynamic program calls include the following:

- An OPM program can call another OPM program or an ILE program, but it cannot call a service program.

- An ILE program can call an OPM program or another ILE program, but it cannot call a service program.
- A service program can call an OPM program or an ILE program, but it cannot call another service 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 15 on page 23 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.

```
┌ Program A ──────────────┐                  ┌ Service Program B ──────────┐
│  ┌ Module ──────────┐   │                  │  ┌ Module ──────────┐       │
│  │  Proc: P1        │   │                  │  │  Proc: P6        │       │
│  │   CallPrc P2     │───┐                  │  │   CallPrc P7     │───┐   │
│  │  End P1          │   │ Static            │  │  End P6          │   │ Static
│  └──────────────────┘   │ Procedure         │  └──────────────────┘   │ Procedure
│  ┌ Module ──────────┐   │ Call              │  ┌ Module ──────────┐   │ Call
│  │  Proc: P2        │◄──┘                  │  │  Proc: P7        │◄──┘   │
│  │  End P2          │   │                  │  │  End P7          │       │
│  └──────────────────┘   │                  │  └──────────────────┘       │
└─────────────────────────┘                  └─────────────────────────────┘

┌ Program C ──────────────┐                  ┌ Service Program D ──────────┐
│  ┌ Module ──────────┐   │                  │  ┌ Module ──────────┐       │
│  │  Proc: P3        │   │ Static Procedure │  │  Proc: P4        │       │
│  │   CallPrc P4     │───┼──── Call ───────►│  │                  │       │
│  │  End P3          │   │                  │  │  End P4          │       │
│  └──────────────────┘   │                  │  └──────────────────┘       │
└─────────────────────────┘                  └─────────────────────────────┘

┌ Service Program E ──────┐                  ┌ Service Program F ──────────┐
│  ┌ Module ──────────┐   │                  │  ┌ Module ──────────┐       │
│  │  Proc: P5        │   │ Static Procedure │  │  Proc: P4        │       │
│  │   CallPrc P4     │───┼──── Call ───────►│  │                  │       │
│  │  End P5          │   │                  │  │  End P4          │       │
│  └──────────────────┘   │                  │  └──────────────────┘       │
└─────────────────────────┘                  └─────────────────────────────┘
                                                         RV2W993-2
```

Figure 15. Static Procedure 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. Because service programs are not called, they are activated during the call to a program that directly or indirectly requires their services.

Activation performs the following functions:
- Uniquely allocates the static data needed by the program or service program
- Changes the symbolic links to service programs providing the exports into links to physical addresses

No matter how many jobs are running a program or service program, only one copy of that object's instructions resides in storage. However, each program activation has its own static storage. So even when one program object is used concurrently by many jobs, the static variables are separate for each activation. A program can also be activated in more than one activation group, even within the same job, but activation is local to a particular activation group.

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 28.

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 run time.

For more information on activation groups, refer to "Activation Group" on page 29.

Error Handling

Figure 16 on page 25 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 16 on page 25 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.

```
Original Program Model          Integrated Language
(OPM)                           Environment (ILE)

  ┌──────┐  ┌──────┐              ┌─────┐ ┌──────┐ ┌──────┐ ┌────────┐
  │  CL  │  │ RPG  │              │  C  │ │  CL  │ │ RPG  │ │ COBOL  │
  └──────┘  └──────┘              └─────┘ └──────┘ └──────┘ └────────┘
```

| HLL - Specific Handlers | | HLL - Specific Handlers | Direct Monitors | ILE Conditions |

| Unhandled Exception Default Actions | | Unhandled Exception Default Actions |

| Exception Message Architecture |
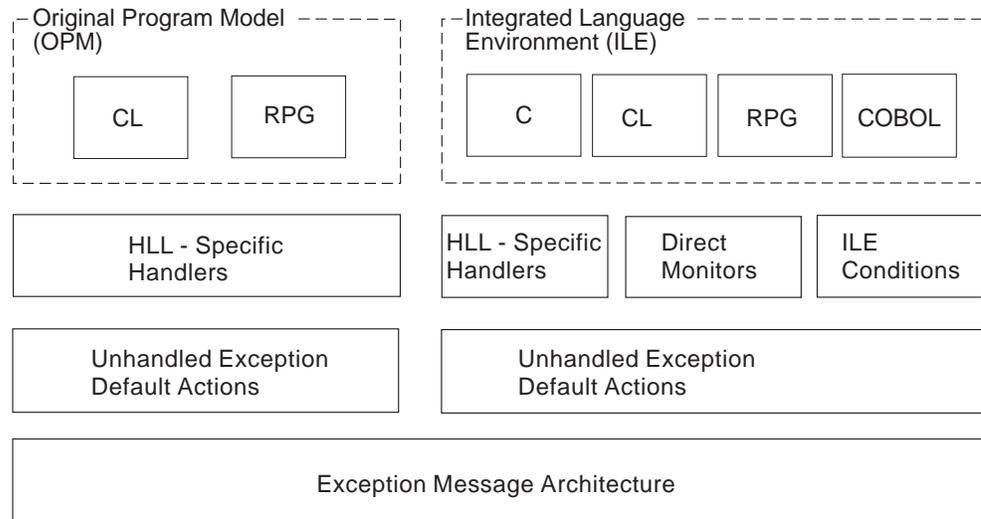
RV3W101-0

Figure 16. 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

On the AS/400, **optimization** means maximizing the run-time performance of the object. All ILE languages have access to the optimization techniques provided by the ILE optimizing translator. Generally, the higher the optimizing request, the longer it takes to create the object. At run time, 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, you probably want a high level of optimization when you use a module in production. Test your code at the optimization level at which you expect to use it. Verify that everything works as expected, then make the code available to your users.

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 "Chapter 9. Debugging Considerations" on page 117 for debug considerations. Refer to "Appendix B. Exceptions in Optimized Programs" on page 173 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 AS/400 system but can be used to debug only OPM programs.

When you debug an optimized module, some confusion may result. When you use the ILE debugger to view or change a variable being used by a running program or procedure, the following happens. The debugger retrieves or updates the data in the storage location for this variable. At level 20 (*BASIC), 30 (*FULL), or 40 optimization, the current value of a data variable may be in a hardware register, where the debugger cannot access it. (Whether a data variable is in a hardware register depends on several factors. Those factors include how the variable is used, its size, and where in the code you stopped to examine or change the data variable.) Thus, the value displayed for a variable may not be the current value. For this reason, you should use an optimization level of 10 (*NONE) during development. Then, for best performance, you should change the optimization level to 30 (*FULL) or 40 during production.

For more information on the ILE debugger, see "Chapter 9. Debugging Considerations" on page 117.

Chapter 3. ILE Advanced Concepts

This chapter describes advanced concepts for the ILE model. Before reading this chapter, you should be familiar with the concepts described in "Chapter 2. ILE Basic Concepts" on page 11.

Program Activation

Activation is the process used to prepare a program to run. Both ILE programs and ILE 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 34.

Figure 17 on page 28 shows two ILE program objects stored in permanent disk storage. As with all OS/400 objects, these program objects may be shared by multiple concurrent users running in different OS/400 jobs. Only one copy of the program's code exists. When one of these ILE programs is called, however, some variables declared within the program must be allocated and initialized for each program activation.

As shown in Figure 17, each program activation supports at least one unique copy of these variables. Multiple copies of variables with the same name can exist within one program activation. This occurs if your HLL allows you to declare static variables that are scoped to individual procedures.
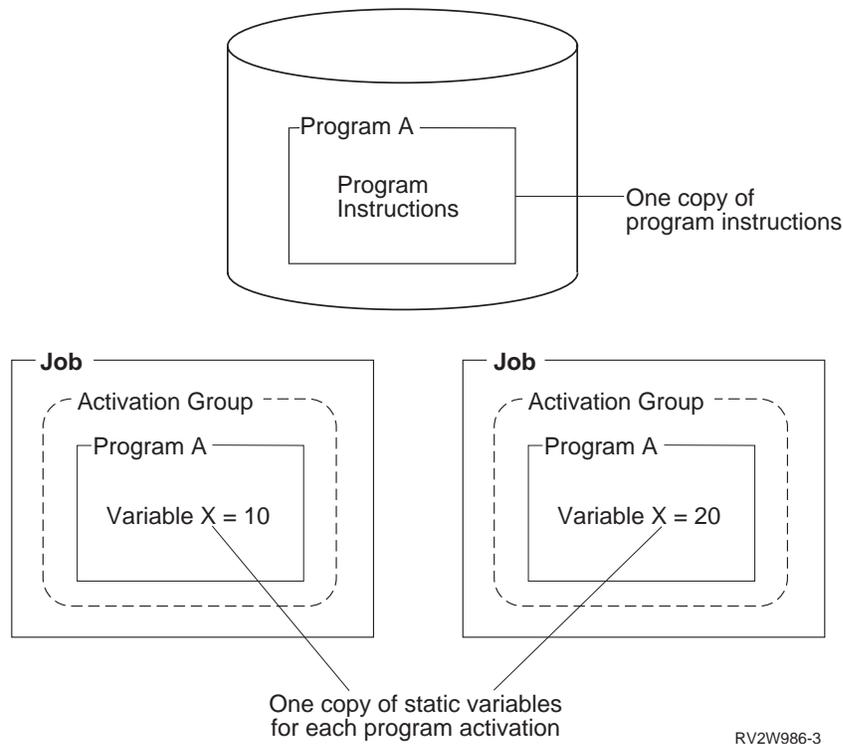
Figure 17. One Copy of Static Variables for Each Program Activation

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 29 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 AS/400 libraries are considered <u>different</u> program objects when applying this rule.

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 UPDPGM and UPDSRVPGM 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 within the activation group. Figure 18 on page 29 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 91.
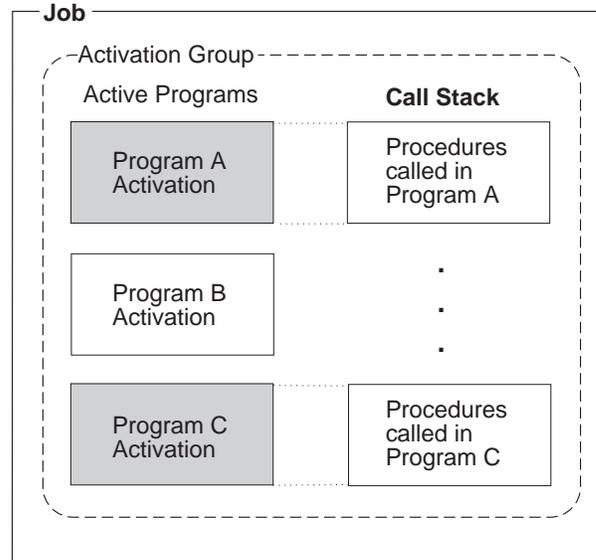
Figure 18. Program May Be Active But Not on the Call Stack

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 and automatic program variables

Dynamic storage

Temporary data management resources

Certain types of exception handlers and ending procedures

The static and automatic program variables and dynamic storage are assigned separate address spaces for each activation group. This provides some 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 19. 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.
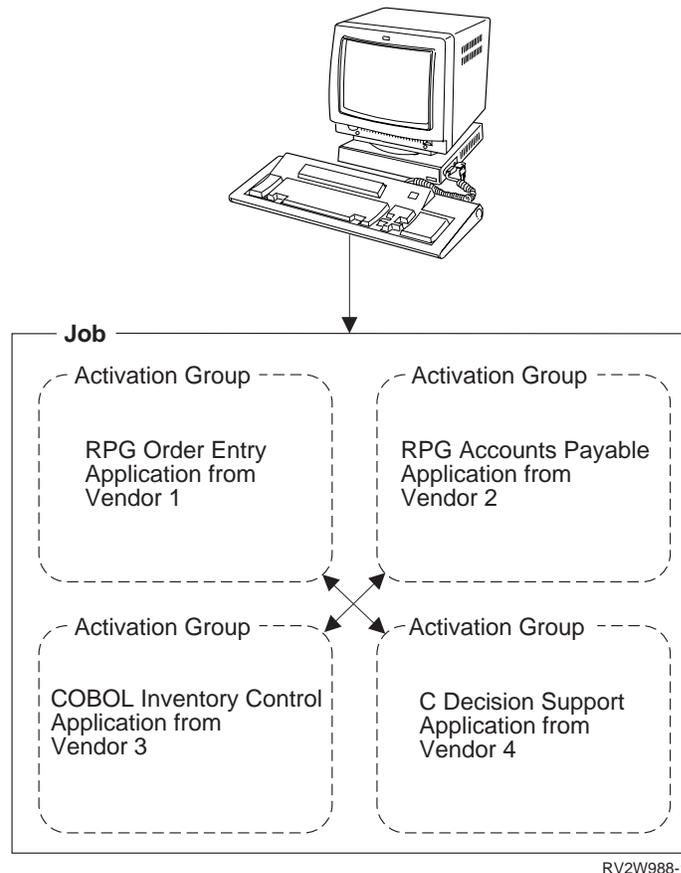
Figure 19. Activation Groups Isolate Each Vendor's Application

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 automatic program variables and dynamic storage that has not been deallocated is returned to the system.

Activation Group Creation

You can control the run-time creation of an ILE 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 ILE 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. ILE 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 ILE 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.

All activation groups within a job have a name. Once an activation group exists within a job, it is used by ILE 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. You can, however, use the ACTGRP parameter on the UPDPGM and UPDSRVPGM to change the name of the activation group.

Default Activation Groups

When an OS/400 job is started, the system creates two activation groups to be used by OPM programs. One activation group is reserved for OS/400 system code. The other activation group is used for all other OPM programs. You cannot delete the OPM default activation groups. They are deleted by the system when your job ends.

ILE programs and ILE service programs can be activated in the OPM default activation groups if two conditions are satisfied:
- The ILE programs or ILE service programs were created with the activation group *CALLER option.
- The call to the ILE programs or ILE service programs originates in the OPM default activation groups.

Because the default activation groups cannot be deleted, your ILE HLL end verbs cannot provide complete end processing. Open files cannot be closed by the system until the job ends. The static and heap storages used by your ILE programs cannot be returned to the system until the job ends.

Figure 20 on page 32 shows a typical OS/400 job with an ILE activation group and the OPM default activation groups. The two OPM default activation groups are combined because the special value *DFTACTGRP is used to represent both groups. The boxes within each activation group represent program activations.
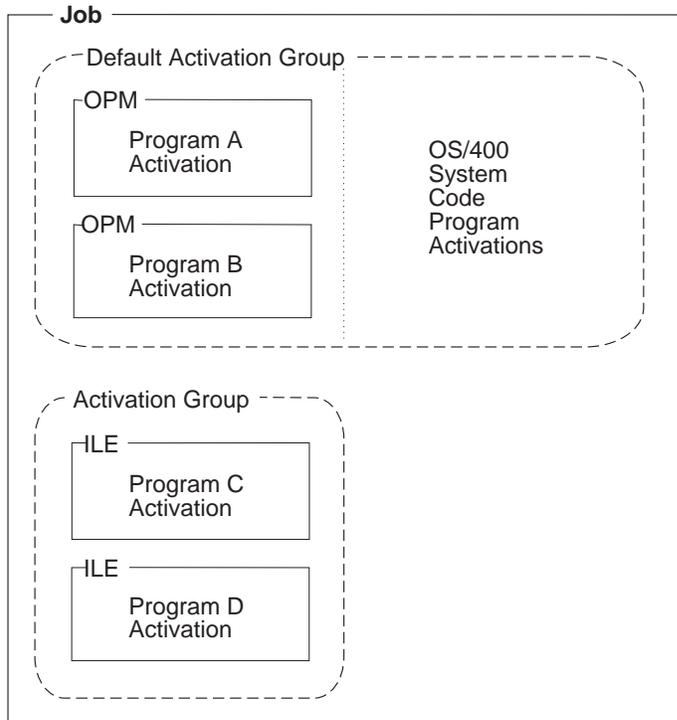
Figure 20. Default Activation Groups and ILE Activation Group

ILE 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 the activation group without ending or deleting the 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 leave an activation group and return to a call stack entry (see "Call Stack" on page 91) that is running in another activation group in the following ways:

- HLL end verbs

  For example, STOP RUN in COBOL or exit() in C.
- 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 in your activation group.

You can delete an activation group from your application by using HLL end verbs. 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 in 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 36 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. The same operations used within a system-named activation group deletes the activation group. System-named activation groups are always 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 of an activation group, refer to the ILE HLL programmer's guides.

Figure 21 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 activation group. For the user-named activation group (created with the ACTGRP(name) option), a normal return from P1 does not delete the activation group.
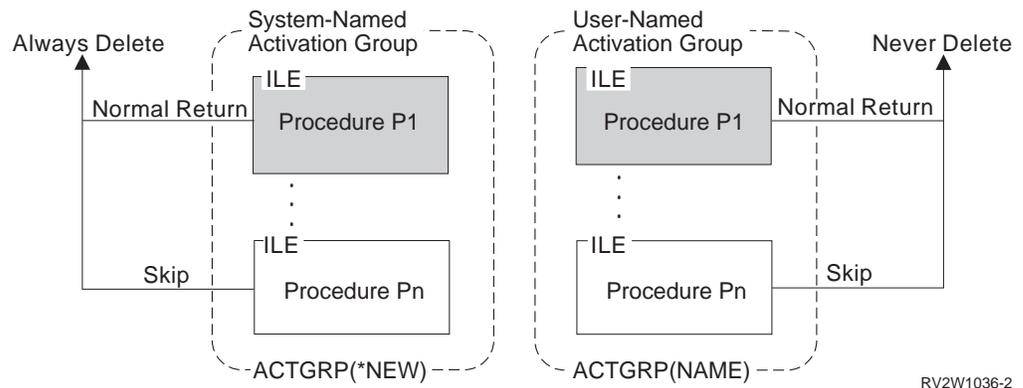
Figure 21. Leaving User-Named and System-Named Activation Groups

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 22 on page 34 shows an OS/400 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 for the ILE procedures 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.

```
Job
  ┌─Activation Group in Use─────────────┐
  │  Active Programs        Call Stack  │
  │  ┌──────────────┐   ┌─────────────┐ │
  │  │              │   │ Procedures  │ │
  │  │  Program A   │...│ called in   │ │
  │  │              │   │ Program A   │ │
  │  └──────────────┘   └─────────────┘ │
  │                                     │
  │  ┌──────────────┐   ┌─────────────┐ │
  │  │              │   │ Procedures  │ │
  │  │  Program B   │...│ called in   │ │
  │  │              │   │ Program B   │ │
  │  └──────────────┘   └─────────────┘ │
  └─────────────────────────────────────┘

  ┌─Activation Group Not in Use─┐
  │      Active Programs        │
  │   ┌──────────────────┐      │
  │   │ Program C        │      │
  │   │ Activation       │      │
  │   └──────────────────┘      │
  │                             │
  │   ┌──────────────────┐      │
  │   │ Program D        │      │
  │   │ Activation       │      │
  │   └──────────────────┘      │
  └─────────────────────────────┘
```
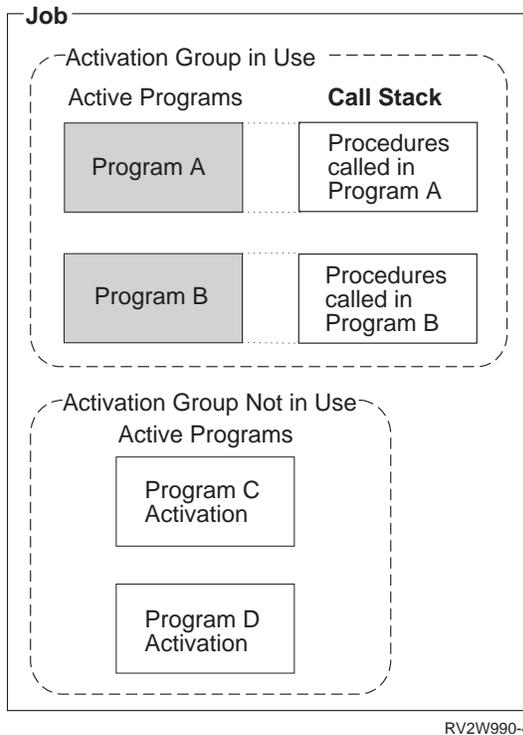
RV2W990-4

Figure 22. 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 46 for details on the data management processing that occurs when an activation group is deleted.

Service Program Activation

This topic discusses the unique steps the system uses to activate a service program. The common steps used for programs and service programs are described in "Program Activation" on page 27. The following activation activities are unique for service programs:

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

An ILE program activated for the first time within an activation group, is checked for binding to any ILE service programs. If service programs have been bound to the program being activated, they are also activated as part of the same dynamic call processing. This process is repeated until all necessary service programs are activated.

Figure 23 shows ILE program A bound to ILE service programs B, C, and D. ILE service programs B and C are also bound to ILE service program E. The activation group attribute for each program and service program is shown.
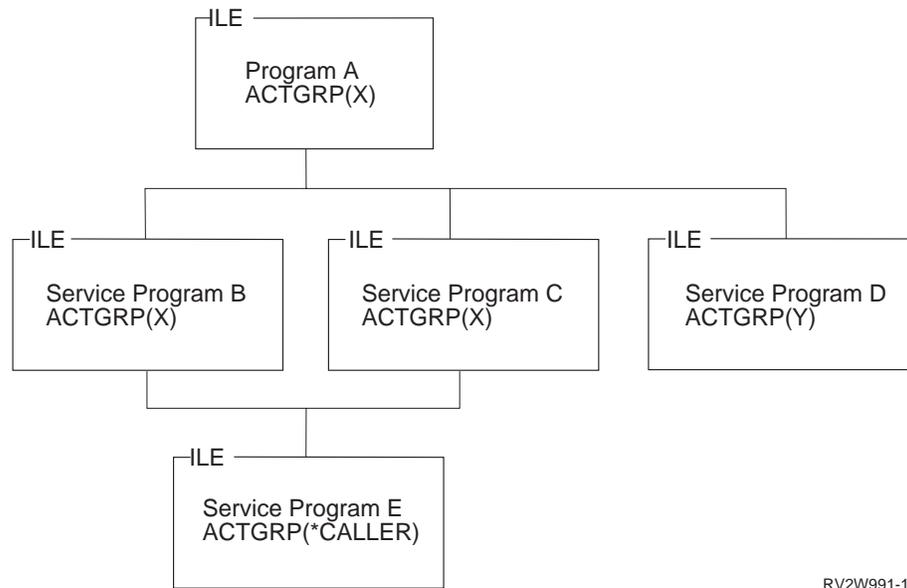
```
                    ┌─ILE ──────────────┐
                    │  Program A         │
                    │  ACTGRP(X)         │
                    └────────────────────┘
        ┌───────────────────┼──────────────────────┐
  ┌─ILE ──────────┐  ┌─ILE ──────────┐    ┌─ILE ──────────────┐
  │ Service Program B│  │ Service Program C│    │ Service Program D  │
  │ ACTGRP(X)      │  │ ACTGRP(X)      │    │ ACTGRP(Y)          │
  └────────────────┘  └────────────────┘    └────────────────────┘
        └───────────────────┤
                    ┌─ILE ──────────────┐
                    │ Service Program E  │
                    │ ACTGRP(*CALLER)    │
                    └────────────────────┘
                                              RV2W991-1
```

Figure 23. Service Program Activation

When ILE program A is activated, the following takes 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 23, 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 run time.

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 ILE service program objects. In Figure 23, the current authority of the caller of program A is used to check authority to program A and all of the service programs. The authority of program A is also used to check

authority to all of the service programs. Note that the authority of service program B, C, or D is not used to check authority to service program E.

Control Boundaries

ILE takes the following action when an unhandled function check occurs, or an HLL end verb is used. 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 ILE Activation Groups" and "Control Boundaries for the OPM Default Activation Group" on page 37 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 in a different nondefault activation group.
- Any ILE call stack entry for which the immediately preceding call stack entry is an OPM program.

Control Boundaries for ILE Activation Groups

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

Figure 24 on page 37 shows two ILE activation groups and the control boundaries established by the various calls. Procedures P2, P3, and P6 are potential control boundaries. For example, when you are running in procedure P7, procedure P6 is the control boundary. When you are running in procedures P4 or P5, procedure P3 becomes the control boundary.
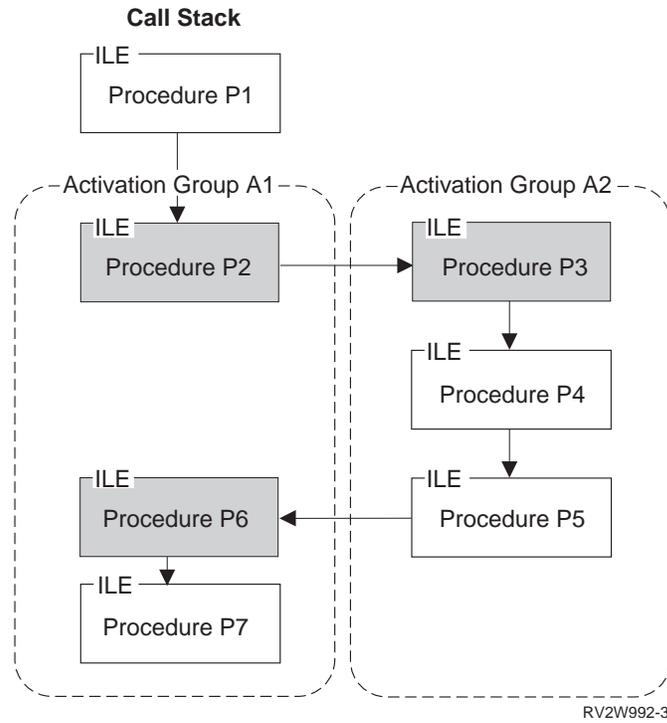
Call Stack

Figure 24. Control Boundaries. The shaded procedures are control boundaries.

Control Boundaries for the OPM Default Activation Group

This example shows how control boundaries are defined when an ILE program is running in the OPM default activation group.

Figure 25 on page 38 shows three ILE procedures (P1, P2, and P3) running in the OPM default activation group. This example could have been created by using the CRTPGM command or CRTSRVPGM command with the ACTGRP(*CALLER) parameter value. Procedures P1 and P3 are potential control boundaries because the preceding call stack entries are OPM programs A and B.
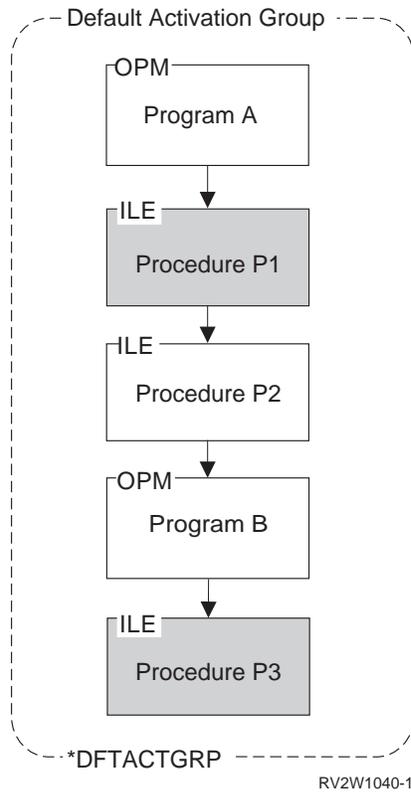
Figure 25. Control Boundaries in the Default Activation Group. The shaded procedures are control boundaries.

Control Boundary Use

When you use an ILE HLL end verb, 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 in an ILE activation group, any HLL end verb or unhandled function check causes the activation group to be deleted. When the nearest control boundary is not the oldest call stack entry in an ILE 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 within the same activation group.

Figure 24 on page 37 shows procedure P2 and procedure P3 as the oldest call stack entries in their activation groups. Using an HLL end verb 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 26. Specific reference information and additional concepts are found in "Chapter 8. Exception and Condition Management" on page 107. Figure 26 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.
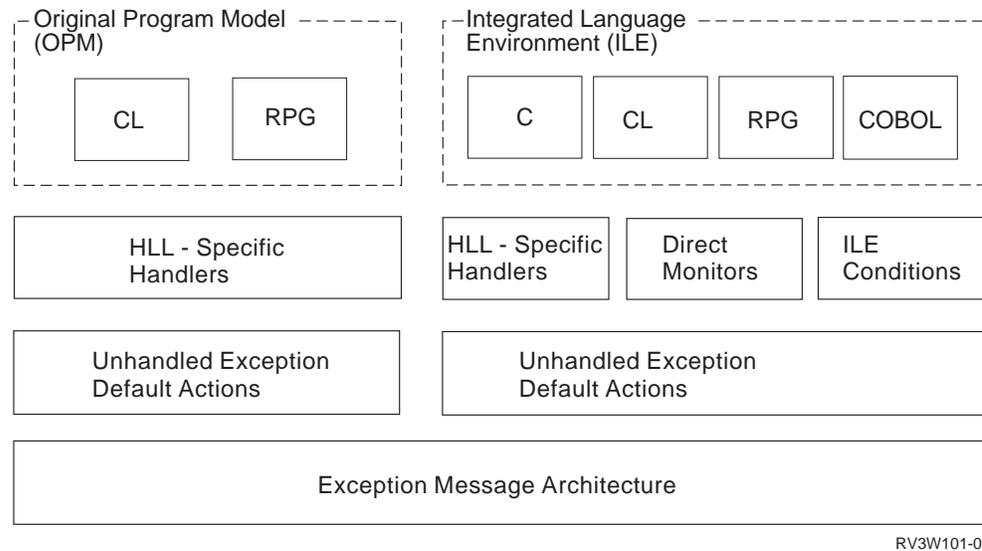

RV3W101-0

Figure 26. ILE and OPM Error Handling

Job Message Queues

A message queue exists for every call stack entry within each OS/400 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 OS/400 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.

IBM has online information on how to send and receive exception messages by using the message handling APIs. Refer to the *CL and APIs* section of the *Programming* category for the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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.

IBM has online information on these message types and other OS/400 message types. Refer to the *CL and APIs* section of the *Programming* category for the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

An exception message is sent in the following ways:

- Generated by the system

  OS/400 (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/400, the raise() function generates a C signal. Neither ILE RPG/400 nor ILE COBOL/400 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.

Additional capabilities defined for ILE will allow you to 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 on direct monitor handlers and ILE condition handlers. IBM provides online information that explains how to change an exception message. Refer to the Change Exception Message (QMHCHGEM) API in the *CL and APIs* section of the *Programming* category for the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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 "Chapter 8. Exception and Condition Management" on page 107.

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 26 on page 39 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.

Figure 27 on page 43 shows unhandled exception messages within ILE. In this example, procedure P1 is a control boundary. Procedure P1 is also the oldest call stack entry in the activation group. Procedure P4 incurred an exception message that was unhandled. Percolation of an unhandled exception continues until either a control boundary is reached or the exception message is handled. An unhandled exception is converted to a function check when it is percolated to the control boundary. If the exception is an escape, the function check is generated. 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. If it is a status exception, the exception is handled, and the sender of the status is allowed to continue. The resume point (shown in procedure P3) is used to define the call stack entry at which exception processing of the function check should continue. For ILE, the next processing step is to send the special function check exception message to this call stack entry. This is procedure P3 in this example.

The function check exception message can now be handled or percolated to the control boundary. If it is handled, normal processing continues and exception processing ends. If the function check message is percolated to the control boundary, ILE considers the application to have ended with an unexpected error. A generic failure exception message is defined by ILE for all languages. This message is CEE9901 and is sent by ILE to the caller of the control boundary.

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.
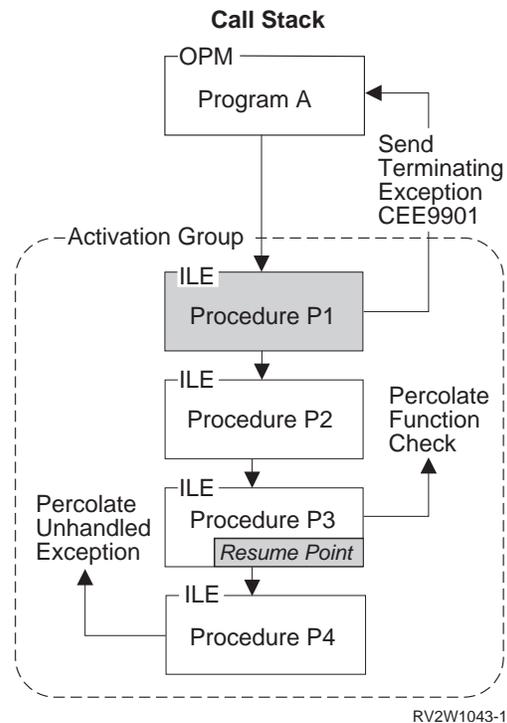
Call Stack

Figure 27. 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 26 on page 39, 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/400, this capability is enabled through a #pragma directive. ILE COBOL/400 does not directly declare an exception monitor around limited HLL source statements in the same sense that ILE C/400 does. An ILE COBOL/400 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 same mechanism. 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 via CEEHDLR). Control then proceeds to the USE declaratives in COBOL.

ILE condition handlers allow you to **register** an exception handler at run time. 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 run time 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/400 signal function can be used to handle exception messages. HLL-specific error handling in RPG includes the ability to code *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 28 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 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 28 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 may 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. If we categorize

COBOL statement-scoped conditional imperatives as HLL-specific handlers, those imperatives take priority over the ILE condition handler.

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.
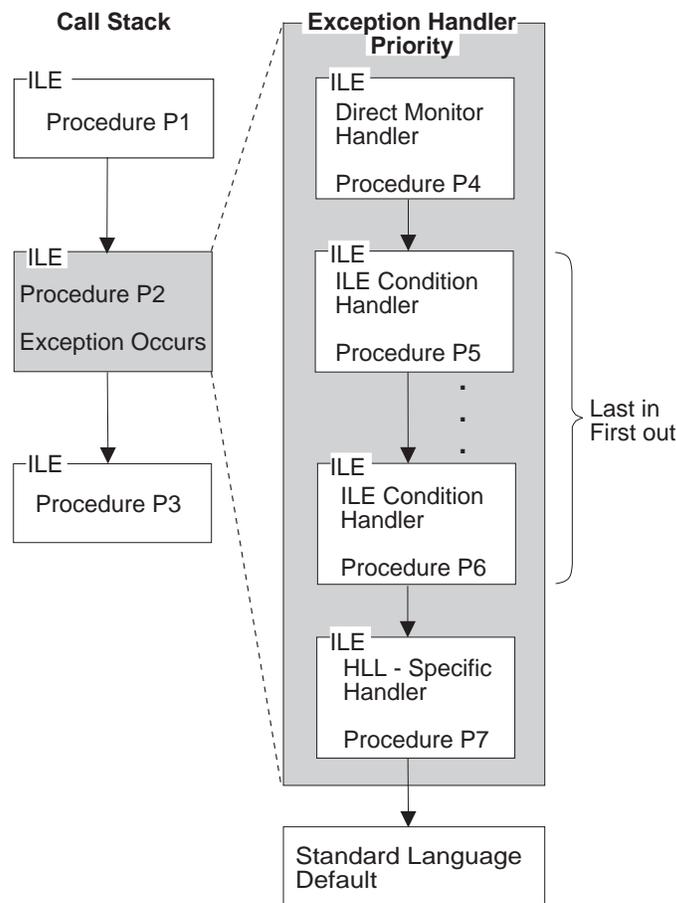
Figure 28. Exception Handler Priority

ILE Conditions

To allow greater cross-system consistency, ILE has defined a feature that allows you to work with error conditions. An ILE **condition** is a system-independent representation of an error condition within an HLL. For the OS/400 operating system, 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 SAA participating systems. This data
structure contains information that allows you to associate the condition with the
underlying exception message.

ILE condition handlers and the percolation model described previously conform to
an SAA architecture. 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 "Chapter 8. Exception and Condition
Management" on page 107.

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 "Chapter 10. 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 29 on page 47 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.
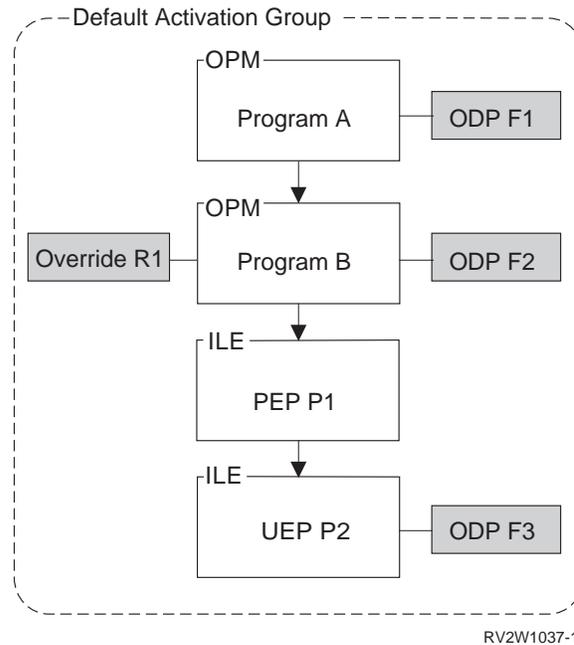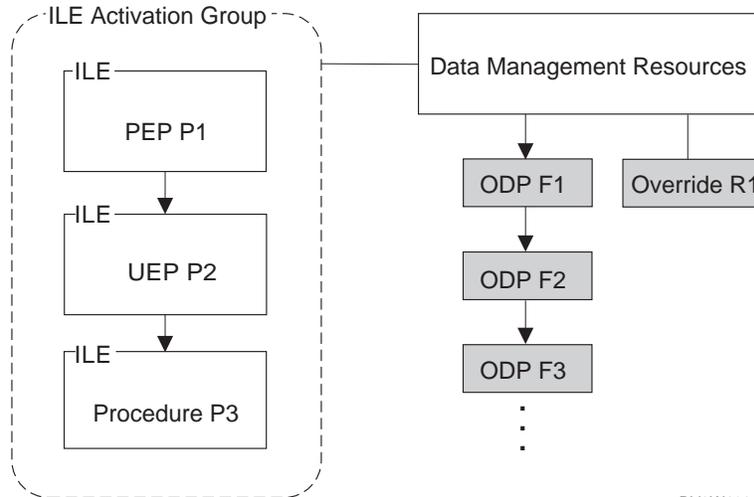

Figure 29. Call-Level Scoping. ODPs and overrides may be scoped to the call level.

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 of the ILE program or ILE service program 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 by programs running in the activation group. Figure 30 on page 48 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 the 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 30. Activation Group Level Scoping. ODPs and overrides may be scoped to an activation group.

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 31 on page 49 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.
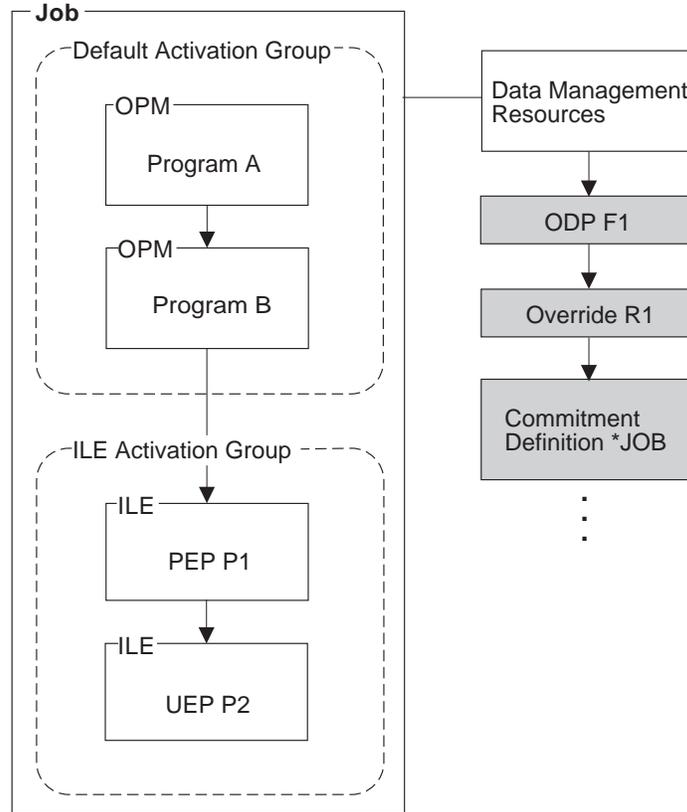
Figure 31. Job Level Scoping. ODPs, overrides, and commitment definitions may be scoped to the job level.

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 "Chapter 10. Data Management Scoping" on page 123.

Chapter 4. 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 chapter explains concepts associated with the binder and with the process of creating ILE programs or service programs. Before reading this chapter, you should be familiar with the binding concepts described in "Chapter 2. ILE Basic Concepts" on page 11.

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 2 shows the commands and their parameters with the default values supplied.

Table 2. 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/QSRVSRC) SRCMBR(*SRVPGM)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Run time	ACTGRP(*NEW)	ACTGRP(*CALLER)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER)	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER)

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 60). 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 the user has the necessary authorities. See the *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 QPRCRTPG 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. If the command being attempted when the authorization list cannot be found is Create Pascal Program (CRTPASPGM) or Create Basic Program (CRTBASPGM), the result is a function check.

Table 3. Possible Values for QUSEADPAUT

Values	Description
authorizationlist name	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).

Table 3. Possible Values for QUSEADPAUT (continued)

Values	Description
*NONE	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. See the *Security - Reference* to find out what authorities are required.

For more information about the QUSEADPAUT system value, see the *Security - Reference* .

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 run-time 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 run time, the following occurs:
- If the program object or service program was created or updated for a Version 2 Release 3 system, error message MCH3203 is issued. That message says, "Function error in machine instruction."
- If the program object or service program was created or updated for a Version 3 Release 1 system, 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 run time.

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 exceed the limit, you can use one the following methods:

1. Use binding directories to bind 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 started 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(payroll/*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 libraries 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.

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.
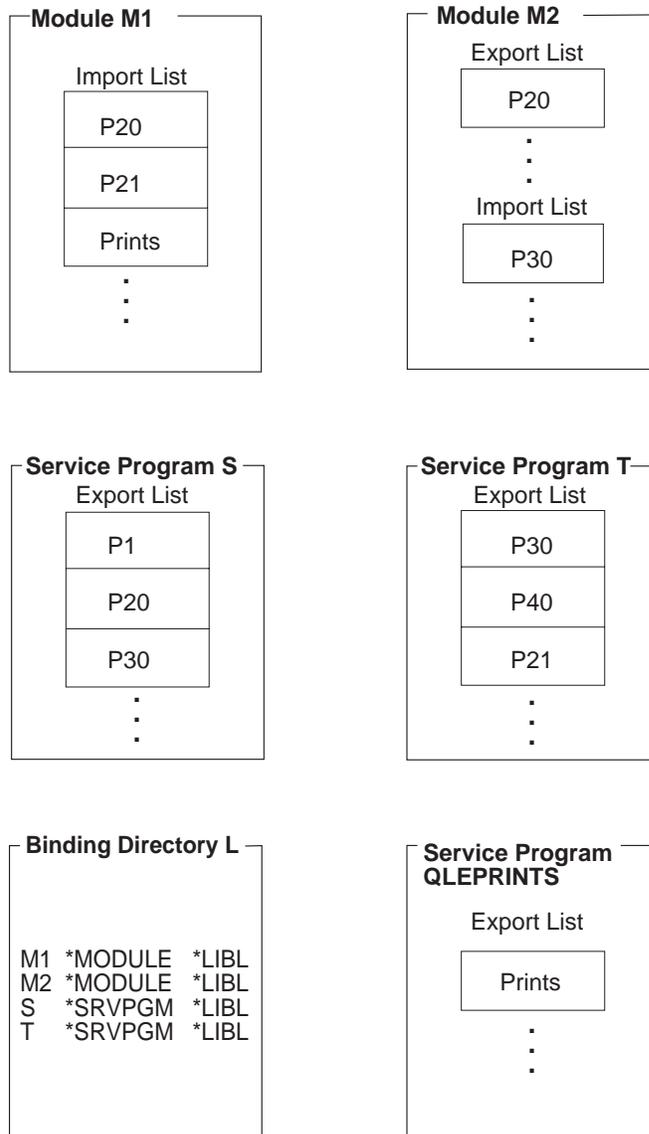
The following examples show how symbol resolution works. The modules, service programs, and binding directories in Figure 32 on page 56 are used for the CRTPGM requests in Figure 33 on page 57 and Figure 34 on page 59. Assume that all the identified exports and imports are procedures.

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))

```
┌─ Module M1 ──────────┐          ┌─ Module M2 ──────────┐
│                      │          │     Export List      │
│     Import List      │          │   ┌──────────────┐   │
│   ┌──────────────┐   │          │   │     P20      │   │
│   │     P20      │   │          │   └──────────────┘   │
│   ├──────────────┤   │          │          .           │
│   │     P21      │   │          │          .           │
│   ├──────────────┤   │          │     Import List      │
│   │    Prints    │   │          │   ┌──────────────┐   │
│   └──────────────┘   │          │   │     P30      │   │
│          .           │          │   └──────────────┘   │
│          .           │          │          .           │
│                      │          │          .           │
└──────────────────────┘          └──────────────────────┘


┌─ Service Program S ──┐          ┌─ Service Program T ──┐
│     Export List      │          │     Export List      │
│   ┌──────────────┐   │          │   ┌──────────────┐   │
│   │     P1       │   │          │   │     P30      │   │
│   ├──────────────┤   │          │   ├──────────────┤   │
│   │     P20      │   │          │   │     P40      │   │
│   ├──────────────┤   │          │   ├──────────────┤   │
│   │     P30      │   │          │   │     P21      │   │
│   └──────────────┘   │          │   └──────────────┘   │
│          .           │          │          .           │
│          .           │          │          .           │
└──────────────────────┘          └──────────────────────┘


┌─ Binding Directory L ┐          ┌─ Service Program ────┐
│                      │          │   QLEPRINTS          │
│                      │          │     Export List      │
│                      │          │   ┌──────────────┐   │
│ M1 *MODULE  *LIBL    │          │   │    Prints    │   │
│ M2 *MODULE  *LIBL    │          │   └──────────────┘   │
│ S   *SRVPGM  *LIBL   │          │          .           │
│ T   *SRVPGM  *LIBL   │          │          .           │
│                      │          │                      │
└──────────────────────┘          └──────────────────────┘
                                              RV2W1054-3
```

Figure 32. Modules, Service Programs, and Binding Directory

Program Creation Example 1

Assume that the following command is used to create program A in Figure 33 on page 57:

```
CRTPGM  PGM(TEST/A)
        MODULE(*LIBL/M1)
        BNDSRVPGM(*LIBL/S)
        BNDDIR(*LIBL/L)
        OPTION(*DUPPROC)
```
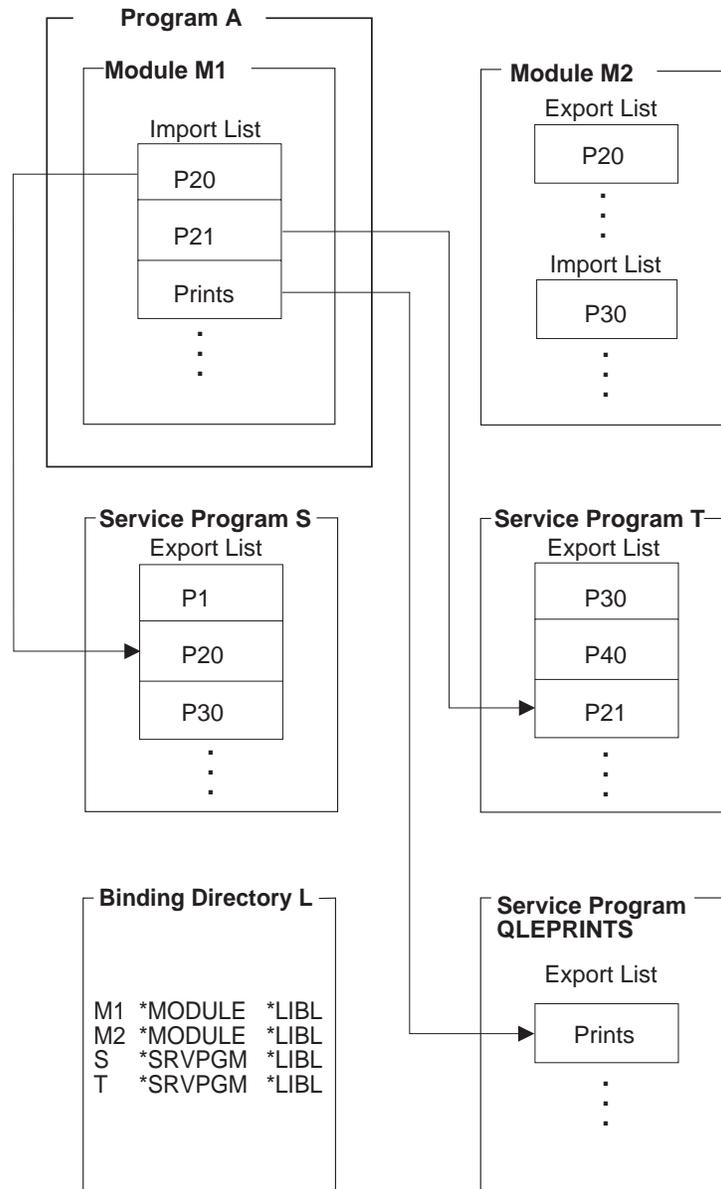
Figure 33. Symbol Resolution and Program Creation: Example 1

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 57 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 run-time routine provided by the compiler in this example. When compiling a module, the compiler specifies as the default the binding directory containing its own run-time service programs and the ILE run-time service programs. That is how the binder knows that it should look for any remaining unresolved references in the run-time service programs provided by the compiler. If, after the binder looks in the run-time 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 34 on page 59 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)
```
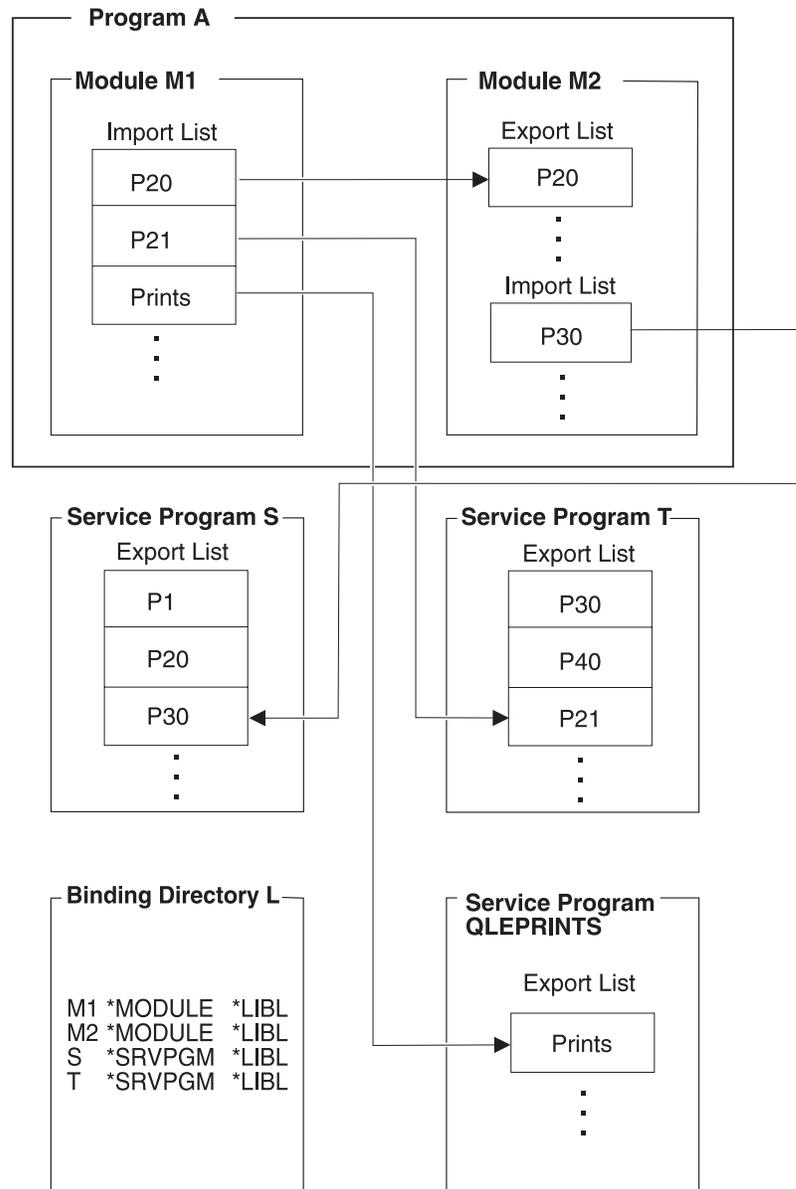
Figure 34. Symbol Resolution and Program Creation: Example 2

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 33 on page 57
with Figure 34 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 run-time 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 63.

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 2 on page 51).

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 object 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), and source member (SRCMBR) 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 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 64). 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 based on exports from a module or from a set of modules. The file created by the RTVBNDSRC command contains all symbols eligible to be exported from the modules, specified in the binder language syntax. You can edit this file to include only the symbols you want to export, then specify this file on the SRCFILE parameter of the CRTSRVPGM command.

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. This value is the easiest one to use because you do not have to generate the binder language source. However, a service program with EXPORT(*ALL) specified can be difficult to update or correct once the exports are used by other programs. If the service program is changed, the order or number of exports could change. Therefore, the signature of that service program could 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/400 can define exports as global or static. Only external variables declared in ILE C/400 as global are available with EXPORT(*ALL). In ILE RPG/400, the following are available with EXPORT(*ALL):

- The RPG program name (not to be confused with *PGM object)
- Variables defined with the keyword EXPORT

In ILE COBOL/400, 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 63.
- 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 the SRCFILE and SRCMBR parameters 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/QSRVSRC)
          SRCMBR(*SRVPGM)
```

For this command to create the service program, a member named UTILITY must be in the source file QSRVSRC. 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.

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 61.) 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 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 a weak data in the binder source language causes the bind to fail.

Table 4 summarizes the types of imports and exports that are supported by some of the ILE languages:

Table 4. 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 nomonocase 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:

- *Licensed Information Document: ILE RPG for AS/400, GI10-4931*
- *Licensed Information Document: ILE COBOL for AS/400, GI10-4932*
- *Licensed Information Document: ILE C for AS/400, GI10-4933*

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 35 on page 65 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 35. Example of Binder Language in a Source File

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

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.

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. Therefore, a signature provides an easy and convenient way to validate the public interface to a service program. A signature does not validate the interface to a particular procedure within a 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.

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

If an incompatible change is made to a service program, exiting programs that remain bound to it may 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 **CRTPGM** or **CRTSRVPGM** after the incompatible change is made.

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. If more than one of the STRPGMEXP commands is marked *CURRENT, the first one is assumed to be the current one. That command is represented by the current signature.

Level Check Parameter on the STRPGMEXP Command

The level check (LVLCHK) parameter on the STRPGMEXP command specifies whether the binder should automatically check the public interface to a service program. Specifying LVLCHK(*YES), or letting the value default to LVLCHK(*YES), causes the binder to examine the signature parameter. The signature parameter determines whether the binder uses an explicit signature value or generates a nonzero signature value. If the binder generates a signature value, 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 being recompiled.

Specifying LVLCHK(*NO) disables the automatic signature checking. You may decide to use this feature if the following conditions exist:
- You know that certain changes to the interface of a service program do not constitute incompatibilities.
- You want to avoid updating the binder language source file or recompiling clients.

Use the LVLCHK(*NO) value with caution because it means that you are responsible for manually verifying that the public interface is compatible with previous levels. Specify LVLCHK(*NO) only if you can control which procedures of the service program are called and which variables are used by its clients. If you cannot control the public interface, run-time or activation errors may occur. See "Binder Language Errors" on page 156 for an explanation of the common errors that could occur from using the binder language.

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 value for a service program by using the Display Service Program (DSPSRVPGM) command and specifying DETAIL(*SIGNATURE).

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 35 on page 65. 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 "Appendix A. Output
Listing from CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM Command" on
page 149.

In the binder language examples, each module contains more than one procedure.
This structure is more typical of ILE C/400 than of ILE RPG/400, but 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 36 on page 70, 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.

```
                    Service Program
                    MYLIB/FINANCIAL

    ┌──────────┐              ┌──────────────┐
    │  Public  │              │    Term      │
    │ Interface│──────────────▶├──────────────┤
    │          │              │    Rate      │
    └──────────┘              ├──────────────┤
                              │   Amount     │
                              ├──────────────┤
                              │   Payment    │
                              └──────────────┘

                         ┌─ Module MONEY ──────┐
                         │                      │
                         │  Procedure Amount    │
                         │                      │
                         │  Procedure Payment   │
                         │                      │
                         └──────────────────────┘

                         ┌─ Module RATES ──────┐
                         │                      │
                         │   Procedure Term     │
                         │                      │
                         │   Procedure Rate     │
                         │                      │
                         └──────────────────────┘

                         ┌─ Module CALCS ──────┐
                         │                      │
                         │  Procedure CALC1     │
                         │                      │
                         │  Procedure CALC2     │
                         │                      │
                         └──────────────────────┘

              Current Signature = Sig 123
```
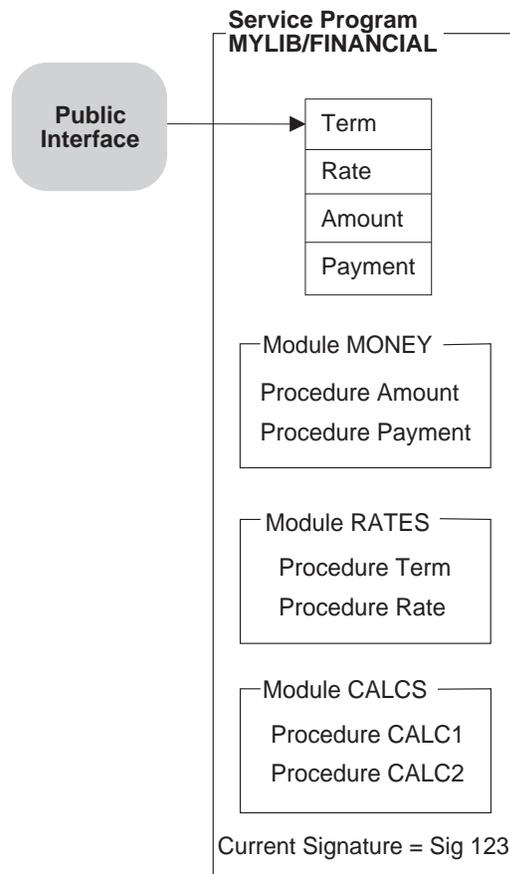
RV2W1051-3

Figure 36. 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 37 on page 71.
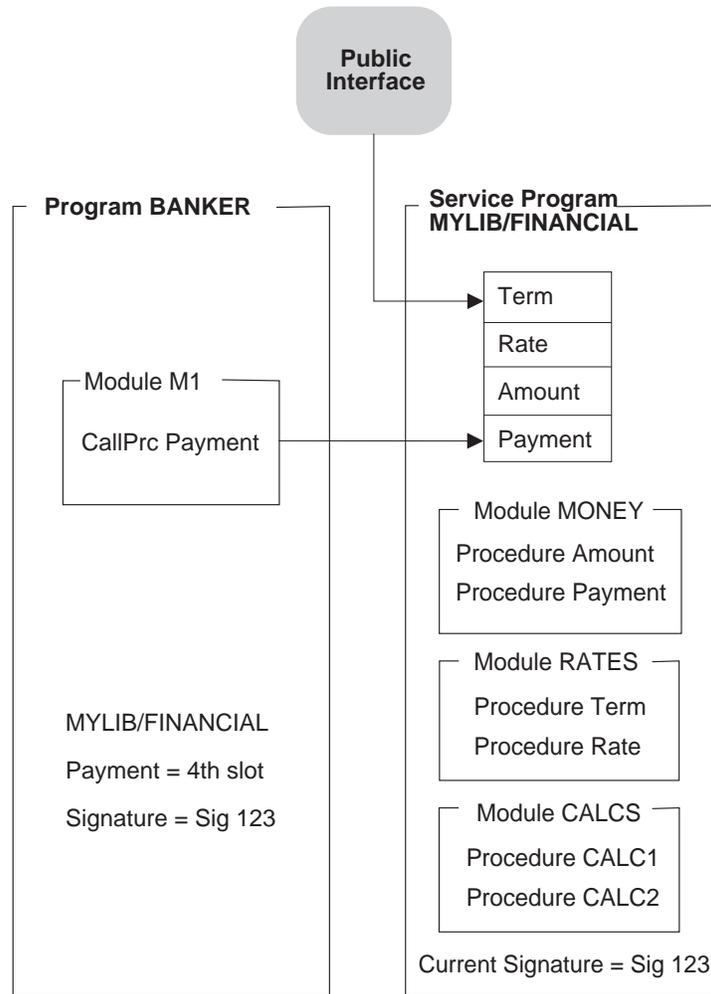
Figure 37. Using the Service Program FINANCIAL

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 run time.

As shown in Figure 37, 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 38, the updated binder
language specified on page 72 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)
```


RV2W1052-4
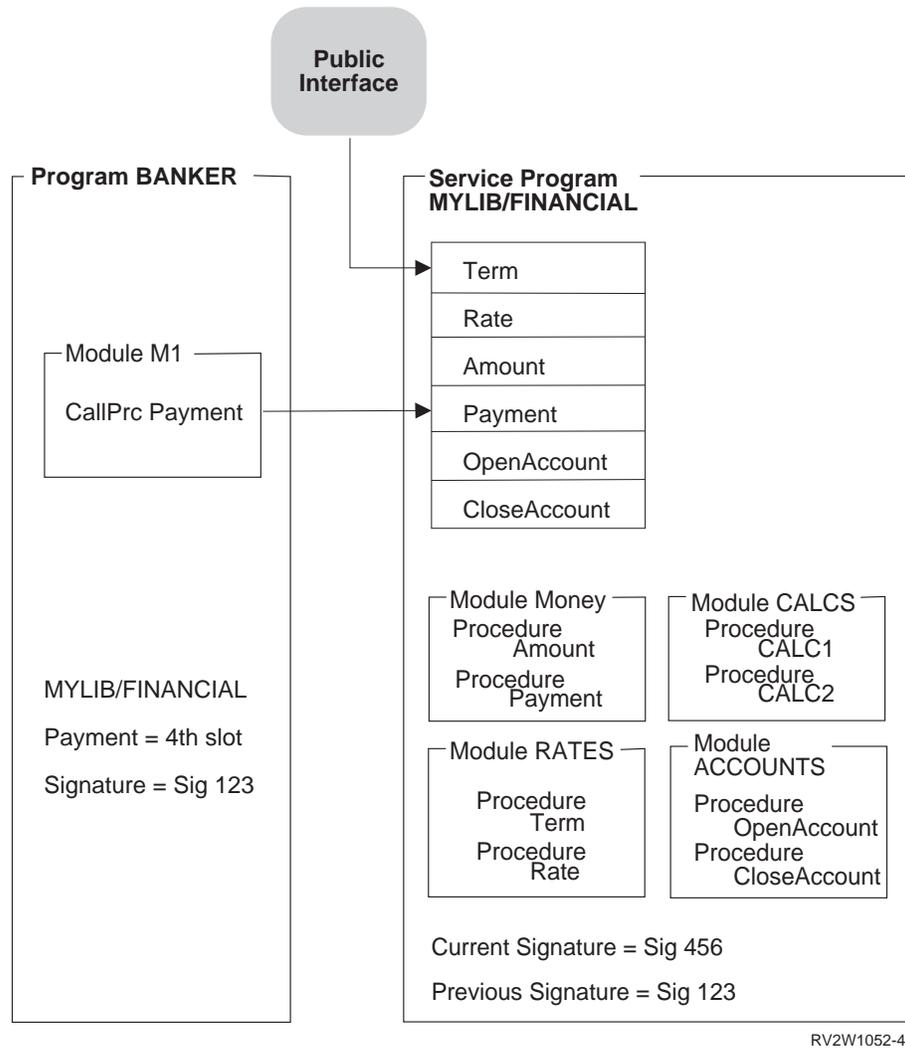
Figure 38. Updating a Service Program by Using the Binder Language

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')
ENDPGMEXP
```

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:
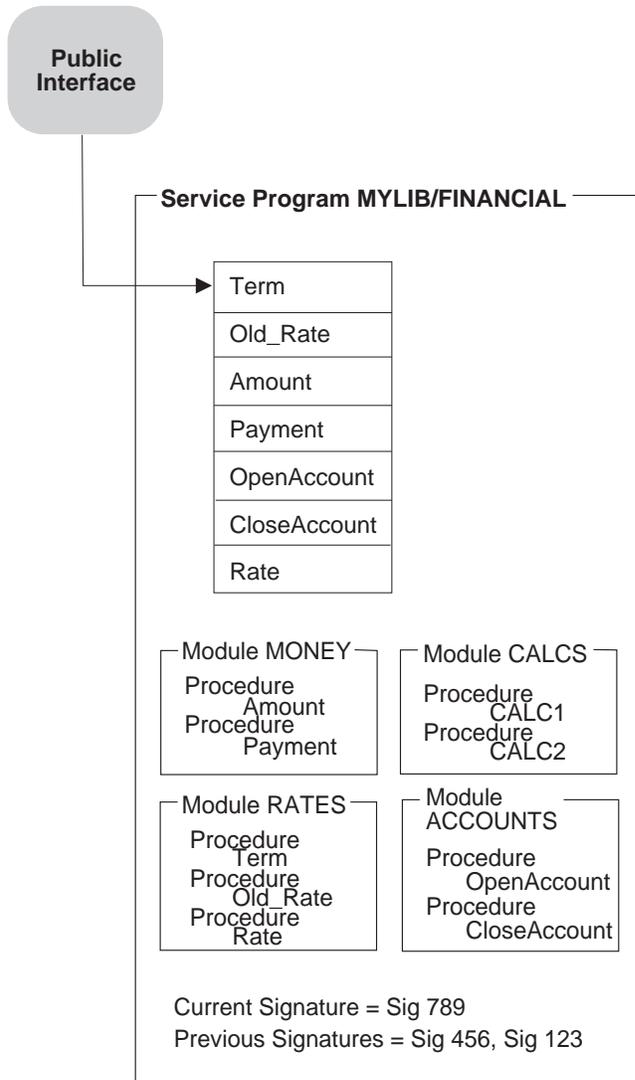
Figure 39. Updating a Service Program by Using the Binder Language

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) command changes the attributes of a program without requiring recompiling. Some of the changable attributes follow:

- The optimization attribute.
- The user profile attribute.
- Use adopted authority attribute.
- The performance collection attribute.
- The profiling data attribute.
- The program text.

The user can also force re-creation of a program even if the specified attributes are the same as the current attributes. Do this by specifying the force program re-creation (FRCCRT) parameter.

Re-creating a program with CHGPGM or CHGSRVPGM while one or more jobs is using it causes an "Object Destroyed" exception to occur. With the new value, *NOCRT, you can prevent this from inadvertently happening. With this new value you can do a CHGCMDDFT to default FRCCRT to *NOCRT. If any parameters are changed on CHGPGM or CHGSRVPGM that may cause a re-create, specify *NOCRT. A DEP statement in the command definition of CHGPGM and CHGSRVPGM will detect the condition, and no programs changes will occur. At this point you can change the FRCCRT parameter value to either *YES or *NO to get the same results. If you remove any observability, or change the text

description, you will not need to re-create the program object. If that is the case, FRCCRT(*NOCRT) will work as FRCCRT(*NO). For compatability, IBM does not ship *NOCRT as the default.

Other jobs running the program may fail by changing any of the parameters that are listed below:

- The Optimize program prompt (OPTIMIZE parameter).
- The Use adopted authority prompt (USEADPAUT parameter).
- The Enable performance collection prompt (ENBPFRCOL parameter).
- The Profiling data prompt (PRFDTA parameter).
- The User profile prompt (USRPRF parameter).

Additionally, forcing a program re-creation by specifying FRCCRT(*YES) may cause other jobs running the program that is being changed to fail.

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, Figure 40 on page 79

Public Interface

Service Program MYLIB/FINANCIAL

Term
Old_Rate
Amount
Payment
OpenAccount
CloseAccount
Rate

Module MONEY
Procedure
    Amount
Procedure
    Payment

Module MONEY
Procedure
    Amount
Procedure
    Payment

Module CALCS
Procedure
    CALC1
Procedure
    CALC2

Module RATES
Procedure
    Term
Procedure
    Old_Rate
Procedure
    Rate

Module ACCOUNTS
Procedure
    OpenAccount
Procedure
    CloseAccount

Current Signature = Sig 789
Previous Signatures = Sig 456, Sig 123

RV3W105-0

Figure 40. Replacing a Module in a Service Program

Be careful not to update a program or service program while the program remains activated in another job. Otherwise, updates to that job remain inactive until the reclaimed by activation or signoff.

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 can not 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 program. Each time you run the UPDPGM or UPDSRVPGM commands, the updated bound service program from the specified library is used. You can also change the library name when activating the program. 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 you transfer an ILE application to AS/400. This parameter also allows you to change the activation group name when you activate a program or service program. The change is ony allowed for 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. The service program export is different if EXPORT(*ALL) is specified.

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 export with the duplicate name selected for binding may be different. If the module being replaced was specified on the CRTPGM or CRTSRVPGM command before the object that contains the selected export, the selected export is selected. (If the data item is weak, it still may not be selected.)

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.

  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 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.

  Using a few large programs rather than many small programs has the additional advantage of using less storage.

- 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 very few right 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.

As an example, approximately 10 service programs are provided for the language-specific and common run-time routines provided by the OS/400. Over 70 modules went into creating these 10 service programs. This ratio seems to be a good balance for performance, understandability, and maintainability.

Chapter 5. Activation Group Management

This chapter 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/400) 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 implement 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 41 on page 86 shows a technique that you may use to share common service functions while keeping the performance advantages of a user-named activation group.
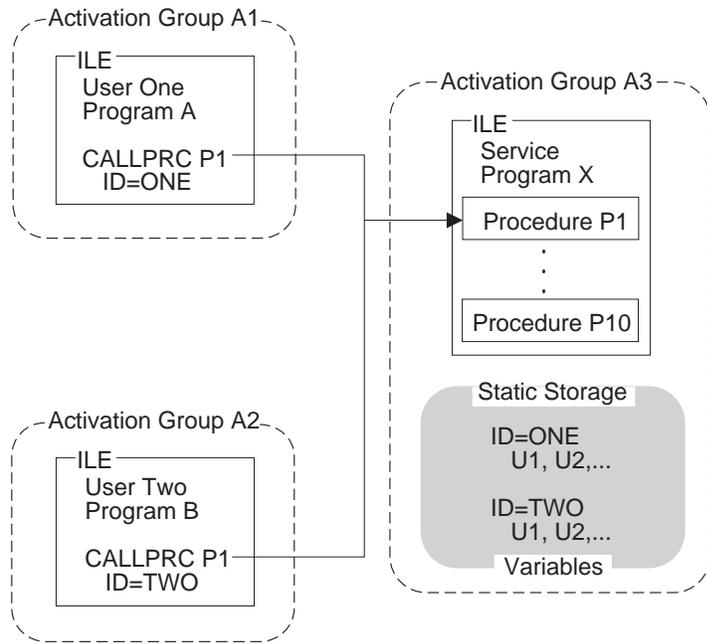
Figure 41. Multiple Applications Running in the Same Job

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. An initialization routine to return a unique handle for each user is implemented by you.

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 42 on page 87 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.

Default Activation Group

Call Stack

OPM	OPM	
Program A Activation	Program A	ODP F1
Number 101	Number 102	Number 102

OPM	OPM	
Program B Activation	Program B RCLRSC LVL(*)	ODP F2
Number 103	Number 104	Number 104

OPM	OPM	
Program C Activation	Program C	ODP F3
Number 105	Number 106	Number 106

ILE	ILE	
Program D Activation	Program D	Shared ODP F1
Number 107	Number 108	Number 108

*DFTACTGRP

Activation Group A1

ILE	ILE
Program A Activation	PEP P1
Number 199	Number 200

ILE
UEP P2
Number 201

RV3W100-0

Figure 42. 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 group.

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 CRTBNDxxx command 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 CRTBNDxxx command with DFTACTGRP(*YES) specified, the RCLRSC command reinitializes any activations that have been created in the 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 the default activation group. The RCLRSC command does not reinitialize static storage of service programs and does not affect nondefault activation groups.

To use the RCLRSC command directly from ILE, you can use either the QCAPCMD API or an ILE CL procedure. The QCAPCMD API allows you to directly call system commands without the use of a CL program. In Figure 42 on page 87, directly calling system commands is important because you may want to use the call-level number of a particular ILE procedure. Certain languages, such as ILE C/400, also provide a system function that allows direct running of OS/400 commands.

Reclaim Activation Group Command

The Reclaim Activation Group (RCLACTGRP) command can be used to delete a nondefault 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

  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.

Chapter 6. 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 chapter discusses different examples of dynamic program calls and static procedure calls, which were introduced in "Calls to Programs and Procedures" on page 21. A third type of call, the procedure pointer call, is introduced.

In addition, this chapter discusses original program model (OPM) support for OPM and ILE application programming interfaces (APIs).

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 job. 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 in the previous stack entry.

Call Stack Example

Figure 43 on page 92 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 12. 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.
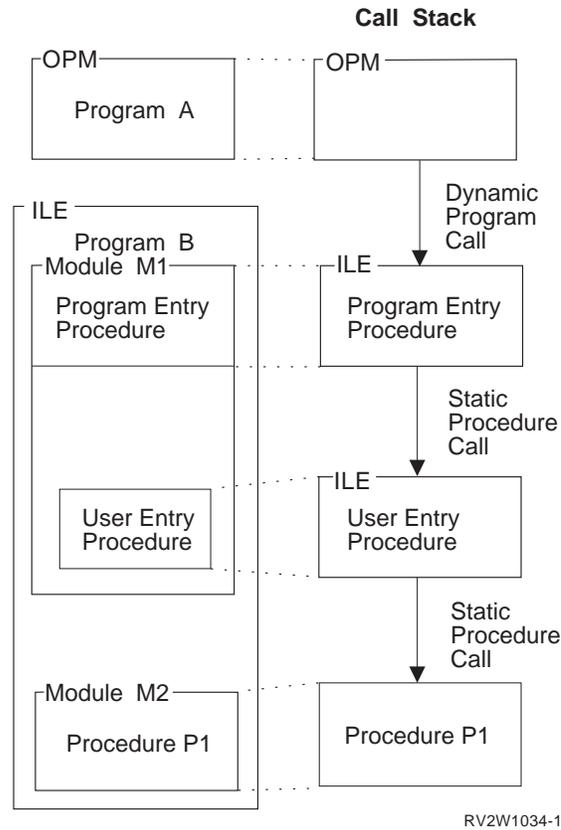
Call Stack

Figure 43. Dynamic Program Calls and Static Procedure Calls on the Call Stack

Figure 43 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 that is currently processing. The current call stack entry 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 removes itself 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 P1 is removed from the stack. Then the UEP returns control to the PEP, and the UEP is removed from the stack. Finally, the PEP returns control to Program A, and the PEP is removed from the stack. Only Program A 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 run time: 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, all the service programs that

are bound to this program 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, 2, and 3, the static procedure call does not cross an activation group boundary. The call path length, which affects performance, is identical. This call path is much shorter than the path for a dynamic program call to an ILE or OPM program. In example 4, 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 must be bound to the calling procedure during binding. The call always accesses the same procedure. This contrasts with a call to a procedure through a pointer, where the target of the 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. Procedure pointer calls can additionally access procedures in any ILE program that has been activated.

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 44 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

```
┌──────────────────────┐
│                      │
│  a  copy  of  argument │
│                      │
└──────────────────────┘
```

By value, indirectly

```
┌──────────────┐         ┌──────────────────────┐
│   pointer    │────────▶│  a  copy  of  argument │
└──────────────┘         └──────────────────────┘
```
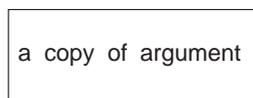
By reference
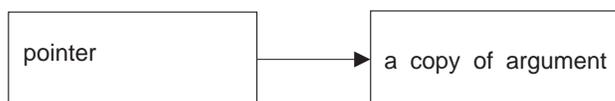
```
┌──────────────┐         ┌──────────────────────┐
│   pointer    │────────▶│  the  actual  argument │
└──────────────┘         └──────────────────────┘
                                     RV2W1027-1
```

Figure 44. 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/400 passes and accepts arguments by value, directly, while for ILE COBOL/400 and ILE RPG/400, 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.

A maximum of 400® 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/400 passes and accepts arguments by value directly, widening integers and floating-point values. Arguments can also be passed by value indirectly by specifying the #pragma argument directive for a called function.
- ILE COBOL/400 passes arguments by reference or by value indirectly. ILE COBOL/400 accepts parameters only indirectly.
- ILE RPG/400 passes and accepts arguments by reference.
- ILE CL passes and accepts arguments by reference.

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 45 on page 95. As described in the ILE HLL programmer's guides, ILE languages that support function results use a common mechanism for returning function results.
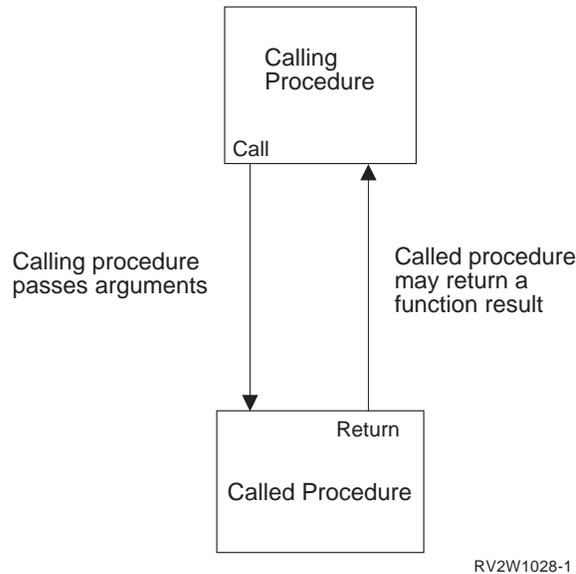
Figure 45. Program Call Argument Terminology

Omitted Arguments

All ILE languages can simulate omitted arguments, which allows the use of the feedback code mechanism for ILE condition handlers and other run-time procedures. For example, if an ILE C/400 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 *CL and APIs* section of the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm** 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.

EPM programs can make program calls and procedure calls. EPM programs can also be called by other programs and procedures.

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. ILE programs that have not been activated yet must be called by a dynamic program call.

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, an EPM 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) usually pass arguments by reference, meaning that the called program receives the address of the arguments. EPM programs can receive arguments passed by reference, by value directly, or by value indirectly.

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. Information on how to use the different passing methods is contained in the ILE HLL programmer's guides, and, for passing methods in EPM, in the *Pascal User's Guide*, SC09-1844.

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/400 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/400 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/400 provides a #pragma argument to pass value arguments to other ILE procedures by value indirectly.

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 should 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 on bindable APIs in the *CL and APIs* section of the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm** 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, it will fail during run time.

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: Descriptors can be an impediment to interlanguage communication when they are generated regardless of need. Descriptors increase the length of the call path, which can diminish performance.

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 (CEESGI) 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 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 46.
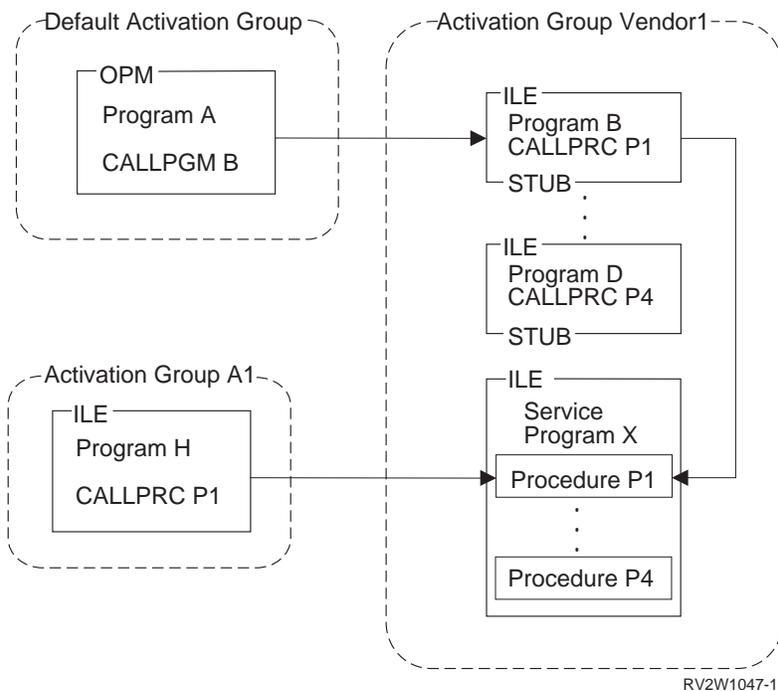
Figure 46. Supporting OPM and ILE APIs

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.

You should write the implementation of your API in an ILE procedure (procedure P1 in this example). This procedure may be called either directly through a procedure call or indirectly through a dynamic program call. You should not implement any functions such as sending exception messages that depend on a specific call stack structure. 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 end verbs delete the activation group whether the call originated from an OPM program or an ILE program.

Chapter 7. 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 run-time 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 run time. 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 run time.

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 chapter concentrates on dynamic storage and the ways in which it can be controlled.

Dynamic Storage

There are two types of dynamic storage: Heap and teraspace.

Heap

The operating system allows the use of multiple heaps that are dynamically created and discarded. 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 programs and procedures that use the heap.

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.

- You can dynamically extend the size of a heap 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 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" and the discussion of heap attributes.

You may 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. IBM provides online information that explains the storage management-bindable APIs. Refer to the *CL and APIs* section that is in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

Single-Heap Support

Languages that do not have intrinsic multiple-heap storage support, such as ILE C/400, use the default 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.

ILE C/400 Heap Support

ILE C/400 provides optional heap support to what is already provided by the system.

If you choose to use this optional support, 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.

If you do not choose this optional support, you can use both the storage management bindable APIs and the malloc(), calloc(), realloc(), and free() functions.

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.

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 (CEECRHP) 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 CEECRHP 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. IBM provides online information that explains all of the _CEE4ALC allocation strategy attributes. Refer to the *CL and APIs* section that is found in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

Teraspace

A teraspace is temporary, dynamic storage that programs and procedures can use at run time. It provides a 1-terabyte address space to a process. A teraspace is not a space object. This means it is not a system object and that you cannot refer to it by using a system pointer. However, teraspace storage is addressable with space pointers. Teraspace storage allocations are controlled with a set of APIs.

Teraspace Characteristics

Teraspace adds capabilities beyond heap. Teraspace has the following characteristics:

- The machine provides at most one teraspace per process, when a running program requests teraspace storage. Teraspace can exist no longer than the time between process initiation and process completion.
- Unless you explicitly request shared access, you can only address a teraspace from one process. The system does not support other attempts to address it from other processes. Such an action has undefined behavior.
- Teraspace has much larger capacity than heap. You can dynamically increase the size of a teraspace to satisfy allocation requests. The maximum size of a teraspace is 1 terabyte (1024 gigabytes).
- The maximum size of any single allocation from a teraspace is limited to 2147483408 bytes if the ILE runtime interface malloc(), calloc(), or realloc() is used. If you use the Portable Operating System Interface for Computer Environments (POSIX) Shared Memory API's, then the maximum size of a single allocation is 4,294,967,295 bytes (4 GB - 1).

- A teraspace may consist of many individually allocated and deallocated separate areas. Addressability "holes" may exist between these areas.

Bindable APIs for Teraspace

IBM provides bindable APIs for allocating and discarding teraspace.[1]
- _C_TS_malloc() allocates storage within a teraspace.
- _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.

Enabling the POSIX shared memory interfaces also provides Teraspace support. See the Interprocess Communication APIs section and the shmget() interface in the UNIX-type API's manual for more information.

Compiler Options

ILE C and C++ compilers provide teraspace options to:
- Request that generated code is enabled to work with teraspace storage. This includes teraspace storage that is allocated by the generated code and parameters that are passed from other teraspace-enabled programs and service programs.
- Use teraspace versions of storage interfaces without source code changes. For example, malloc() is mapped to _C_TS_malloc().

See *ILE C for AS/400 Programmer's Guide*, *LPS: VisualAge for C++ for AS/400*, or *VisualAge C++ for OS/400 Read Me First!* for details on these compiler options.

Additional Comments

You can see the teraspace attribute on the DSPMOD, DSPPGM, and DSPSRVPGM commands and their APIs. Use the DSPMOD, DSPPGM, and DSPSRVPGM commands to display the teraspace attributes of a module, program, and service program, respectively. You can see the teraspace attribute from DETAIL(*MODULE) when you use option 5 to display each module's details.

Storage Management Bindable APIs

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 (CEEFRST) bindable API frees one previous allocation of heap storage.
    The Get Heap Storage (CEEGTST) bindable API allocates storage within a heap.
    The Reallocate Storage (CEECZST) bindable API changes the size of previously allocated storage.

---

1. Teraspace compiler options are available from ILE C and C++ compilers to automatically map malloc(), free(), calloc() and realloc() to their teraspace versions.

- Extended heap operations. These operations can be used only on user-created heaps.

  The Create Heap (CEECRHP) bindable API creates a new heap.

  The Discard Heap (CEEDSHP) bindable API discards an existing heap.

  The Mark Heap (CEEMKHP) bindable API returns a token that can be used to identify heap storage to be freed by the CEERLHP bindable API.

  The Release Heap (CEERLHP) 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 CEECRHP bindable API.

IBM provides specific online information information about the storage management bindable APIs. See the *CL and APIs* section of the *Programming* category under the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

Chapter 8. Exception and Condition Management

This chapter provides additional details on exception handling and condition handling. Before you read this chapter, read the advanced concepts described in "Error Handling" on page 39.

The exception message architecture of the OS/400 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 chapter. The term **exception handler** is used in this chapter to mean either an OS/400 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 47 on page 108 shows an example of the handle cursor and resume cursor.

Call Stack

```
┌─ILE────────────────────┐
│                        │
│   Procedure P1         │
│   CALLPRC P2           │
│      ┌──────────────┐  │
│      │ Resume Point │  │
│      └──────────────┘  │
└────────────│───────────┘
             ▼
┌─ILE────────────────────┐          Exception
│                        │          Handler
│   Procedure P2         │          List
│   CALLPRC P3           │       ┌──────────┐
│      ┌──────────────┐  │       │          │         Handle
│      │ Resume Point │──┼───────│░░░░░░░░░░│◄───────  Cursor
│      └──────────────┘  │       │          │
└────────────│───────────┘       │          │
             ▼                   └──────────┘
┌─ILE────────────────────┐
│                        │
│  Procedure P3          │
│  Exception Occurred    │
│      ┌──────────────┐  │                   Resume
│      │ Resume Point │◄─┼───────────────── Cursor
│      └──────────────┘  │
└────────────────────────┘
          .
          .
          .
┌────────────────────────┐
│                        │
│  Exception Handler     │
│  Procedure P10         │
│                        │
└────────────────────────┘
```
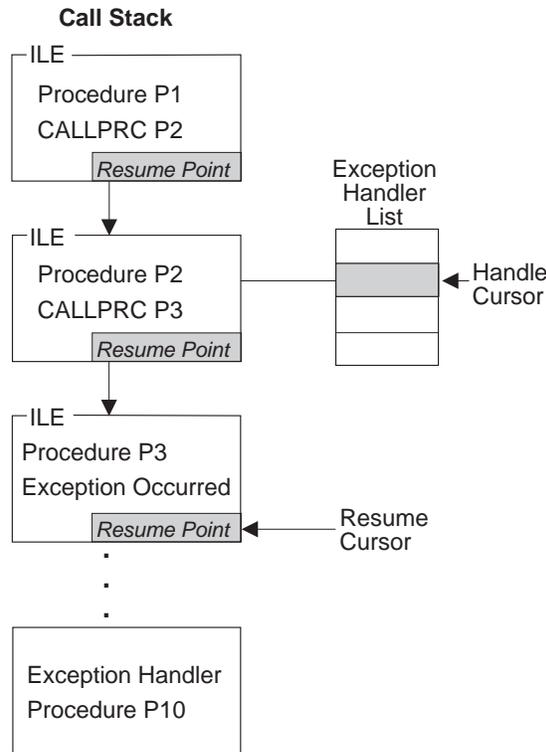
                                          RV2W1044-0

Figure 47. 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 47, 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/400 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. IBM provides online information that describes the actual return code values for the Register a User-Written Condition Handler (CEEHDLR) bindable API. Refer to the *CL and APIs* section that is found in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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. IBM provides online information that describes the actual return code values for the Register a User-Written Condition Handler (CEEHDLR) bindable API. Refer to the *CL and APIs* section that is found in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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 *CL and APIs* section that is found in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

For an ILE condition handler, you communicate the promote action by setting a return code value and returning to the system. IBM provides online information that describes the actual return code values for the Register a User-Written Condition Handler (CEEHDLR) bindable API. Refer to the *CL and APIs* section that is found in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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 5 contains default responses that the system takes when an exception is unhandled at a control boundary.

Table 5. 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.

Table 5. Default Responses to Unhandled Exceptions (continued)

Message Type	Severity of Condition	Condition Raised by the Signal a Condition (CEESGL) Bindable API	Exception Originated from Any Other Source
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 OS/400 exception messages represented in a manner independent of the system. An ILE condition token is used to represent an ILE condition. **Condition handling** refers to the ILE functions that allow you to handle errors separately from language-specific error handling. Other SAA 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 48 on page 113 displays a map of the condition token. The starting bit position is shown for each field.
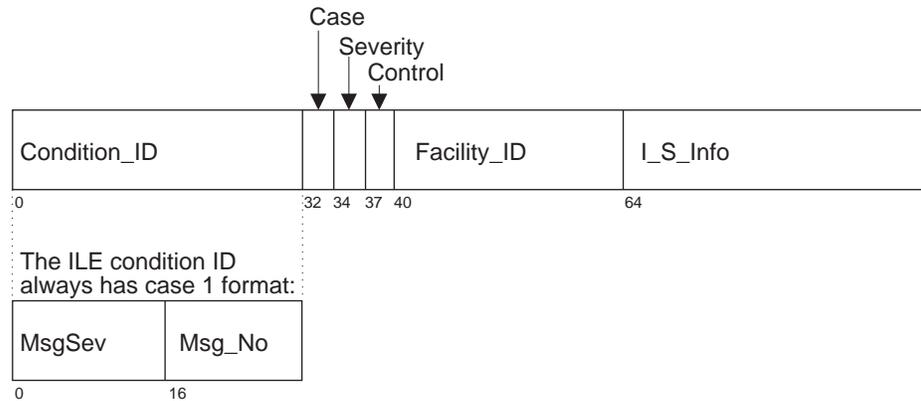
Figure 48. ILE Condition Token Layout

Every condition token contains the components indicated in Figure 48:

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 5 on page 110 for a list of ILE condition severities. See Table 7 on page 114 and Table 8 on page 115 for the corresponding OS/400 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 run time. Table 6 on page 114 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 5 on page 110 for a list of ILE condition severities. See Table 7 on page 114 and Table 8 on page 115 for the corresponding OS/400 message severities.

Msg_No
>A 2-byte binary number that identifies the message associated with the condition. The combination of Facility_ID and Msg_No uniquely identifies a condition.

Table 6 contains the facility IDs used in ILE condition tokens and in the prefix of OS/400 messages.

Table 6. Facility IDs Used in Messages and ILE Condition Tokens

Facility ID	Facility
CEE	ILE common library
CPF	OS/400 XPF message
MCH	OS/400 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 OS/400 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 run-time 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 6 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 7 and Table 8 on page 115 show how ILE condition severity maps to OS/400 message severity.

*Table 7. Mapping AS/400 *ESCAPE Message Severities to ILE Condition Severities*

From AS/400 Message Severity	To ILE Condition Severity	To AS/400 Message Severity
0-29	2	20
30-39	3	30
40-99	4	40

*Table 8. Mapping AS/400 *STATUS and *NOTIFY Message Severities to ILE Condition Severities*

From AS/400 Message Severity	To ILE Condition Severity	To AS/400 Message Severity
0	0	0
1-99	1	10

OS/400 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.

Chapter 9. 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 chapter 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 complied with OPM source debug data.
  - By setting the OPMSRC parameter of the STRDBG command to indicate *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. For example, you can refer to the *ILE RPG for AS/400 Programmer's Guide* for information on debugging OPM RPG programs in the ILE debug environment.

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

Whether a module is observable and whether it is fully optimized affect the ability to debug it.

Module **observability** refers to data that can be stored with a module that allows it to be changed without being compiled again. **Optimization** is a process where the system looks for processing shortcuts that reduce the amount of system resources necessary to produce the same output.

Observability

Module observability consists of two types of data:

Debug Data
> Represented by the *DBGDTA value. This data is necessary to allow a module to be debugged.

Creation Data
> Represented by the *CRTDTA value. This data is necessary to translate the code to machine instructions. The module must have this data for you to change the module optimization level.

Once a module is compiled, you can only remove this data. Using the Change Module (CHGMOD) command, you can remove either type of data from the module, or remove both types. Removing all observability reduces the module to its minimum size (with compression). Once this data is removed, you cannot change the module in any way unless you compile the module again and replace the data. To compile it again, you must have authority to the source code.

Optimization Levels

Generally, if a module has creation data, you can change the level at which the source code is optimized to run on the system. Processing shortcuts are translated into machine code, allowing the procedures in the module to run more efficiently. The higher the optimization level, the more efficiently the procedures in the module run.

However, with more optimization you cannot change variables and may 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/400 includes, ILE RPG/400 /COPY files, and ILE COBOL/400 COPY files).

***ALL** All debug views are produced. As for the list view, the amount of debug data may be very large.

ILE RPG/400 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 appendix on testing in the *CL Programming* book.

OPM and ILE Debugger Support

The OPM and ILE debugger support enable source level debugging of the OPM programs through the ILE Debugger APIs. For information on ILE Debugger APIs, see the *CL and APIs* section under the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**. The OPM and ILE debugger support provide 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 AS/400 RPG, COBOL, or CL compiler. You must set the OPTION parameter *SRCDBG or *LSTDBG for the compilation. AS/400 does not support System/36™ compilers nor EPM compilers.

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.

National Language Support 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/400, ILE COBOL/400, or ILE CL modules, identifier names in debug commands are converted to uppercase by the source debugger and therefore may be redisplayed differently.

Chapter 10. Data Management Scoping

This chapter contains information on the data management resources that may be used by an ILE program or service program. Before reading this chapter, you should understand the data management scoping concepts described in "Data Management Scoping Rules" on page 46.

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 the 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 the 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 *Backup and Recovery*.

Local SQL cursors

You can create SQL programs for ILE compiler products. The SQL cursors used by an ILE program may be scoped to either the module or activation group. You may specify the SQL cursor scoping through the end SQL (ENDSQL) parameter on the Create SQL Program commands.

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 *CL and APIs* section under the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

Open data links (open file management)

The Enable Link (QOLELINK) API enables a data link. If you use this API from within an ILE activation group, the data link is scoped to that activation group. If you use this API from within the default activation group, the data link is scoped to the call level. For more information about open data links, see the *CL and APIs* section under the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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. IBM has online information about Common Programming Interface (CPI) Communications conversations. See the *CL and APIs* section under the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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 *CL and APIs* section under the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

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 *Backup and Recovery*

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 the 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 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. 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 the OPM 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 <u>never</u> 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.

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

the notify object is updated.

Chapter 11. 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 on the ILE bindable APIs, see the *CL and APIs* section of the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

ILE Bindable APIs Available

Most bindable APIs are available to any HLL that ILE supports. Naming conventions of the bindable APIs are as follows:

- Bindable APIs with names beginning with CEE are based on the SAA Language Environment* specifications. These APIs are intended to be consistent across the IBM SAA systems. For more information about the SAA Language Environment, see the *SAA CPI Language Environment Reference*.
- Bindable APIs with names beginning with CEE4 or CEES4 are specific to the AS/400 system.

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 (CEEFMDT)
Return Default Date String for Country (CEEFMDA)
Return Default Time String for Country (CEEFMTM)
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 (CEERAN0)

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.

IBM provides online information about the DSM bindable APIs. Refer to the *CL and APIs* section that is found in the *Programming* category of the **Information Center** at the following Web site - **http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm**.

Chapter 12. Advanced Optimization Techniques

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. Additionally, different AS/400 models have different instruction cache capabilities. A model with limited caching capabilities may not see as much improvement as a model with extensive caching capabilities. Therefore, you may want to consider your customer set to determine the model to profile.

Program profiling is only available for ILE programs and service programs that meet the following conditions:

- The programs were created specifically for V4R2M0 or later releases.
- The program's target release is the same as the current system release level.
- The programs that are compiled by using an optimization level of *FULL (30) or above.

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. It reorders code within the bound module so that the more frequently used condition does not branch. This improves the instruction cache utilization by increasing the likelihood that the next instruction is in the instruction cache, reducing the need to fetch it from main memory.

Procedure order profiling records the number of times each procedure calls another procedure within the program. It reorders the procedures within the program so that the most frequently called procedures are packaged together. This reduces the paging by increasing the likelihood that the called procedure is brought into main memory at the same time the calling procedure was brought in.

Even though you can choose to apply only block order profiling to your program, it is recommended that you apply both types. This provides the best 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.

Program profiling is sensitive to the level of the optimizing translator that is used. IBM recommends performing the steps on the same machine. (Otherwise the collected data may not be usable.)

- Do not apply optimizing translator PTFs between the time programs are enabled for profiling data collection and the next time it is applied.
- Do not apply system release levels between the time programs are enabled for profiling data collection and the time it is applied.

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 collect profiling data after creation through either the Change Program (CHGPGM) or Change Service Program (CHGSRVPGM) commands an observable program must do the following:

- Specify *COL on the profiling data (PRFDTA) parameter that will affect all modules bound in the program that:
  - Must have a target release that matches the current system release which is V4R2M0 or later.
  - Have an optimization level of 30 or above

Note: You can only enable profiling data for programs and service programs that are created for the same release level as the system they are on. The modules in the programs and service programs must also have their created for release at the same level as the system.

Enabling a module or program to collect profiling data requires that the object be retranslated. 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 changes 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 138 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 137 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. IBM ships both commands 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.

Notes:
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 137 for details.

2. Profiling data counts are not written to DASD each time they are incremented as 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 which the program is running in.

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. Additionally it instructs the machine to use the code within procedures (basic block profiling data) in the program for optimal performance.
3. It removes the machine instructions from the program that were previously added when the program was enabled to collect profiling data. This means that the program cannot collect profile data.
4. The collected profiling data is stored 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 134. **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, 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 to apply:
- Block order profiling data (*APYBLKORD)
- Procedure profiling data (*APYPRCORD)
- Or both types of profiling data (*APYALL) or (*APYPRCORD)

IBM recommends using *APYALL.

Applying profiling data to the program creates and saves two new forms of observability with the program. You can remove these new 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 retranslated.
- Applying procedure order profiling data to the program implicitly adds *PRCORD 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 retranslated 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 retranslated will also cause it to no longer be block order profiled.

Note: Removing *CRTDTA observability will also cause *BLKORD observability to be removed implicitly. This is because *BLKORD observability is only needed when the program is retranslated. Since the program cannot be retranslated if *CRTDTA observability is removed, *BLKORD is no longer needed and is also removed. However *PRCORD observability is not removed.

In addition it is not recommend to retranslate the program with *BLKORD observability with a different version of the optimizing translator from the one used to enable the program and apply the profiling data. Optimizing translator PTFs and new releases of the operating system may invalidate some of the basic block profiling data.

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 retranslated. 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 must do one of the following steps to accomplish this.
- Specify *NOCOL on the profiling data (PRFDTA) parameter of the Change Program (CHGPGM)
- Specify *NOCOL on the profiling data (PRFDTA) parameter of the Change Service Program (CHGSRVPGM) commands.

This will change the program back to the state before it collected profiling data. You can change the PRFDTA value of the modules to *NOCOL with the CHGMOD command or by recompiling the modules, and rebind 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 retranslated.

  Note: The optimization level of a program that has profiling data applied cannot be changed. 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 retranslated. 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 retranslated. 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 translated again. This change request will cause program A to still be block order profiled.

- Program A has procedure order and block 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 retranslates the program). This will cause 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.

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 it, a program is:
- First profiled specifying *APYALL or *APYPRCORD (which is the same as *APYALL).
- Then the *PRCORD observability needs to be removed and the program retranslated.

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 modules. 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.

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 passed compiler options affect the code that is generated for a Module, an ILE Program object, or a compiled Java® program. This section will only discuss Licensed Internal Code options in relation to ILE. For information related to Java, see the online help information for the LICOPT parameter of the AS/400

CRTJVAPGM command. Another information source is the web page in the online document http://publib.boulder.ibm.com/pubs/html/as400/v4r4/ic2924/info/java/rzaha/devkit.htm. Only sophisticated users should use Licensed Internal Code options. For example, you can use some of the options for fine tuning code optimization, and other uses aid in debugging a program.

Currently Defined Options

The Licensed Internal Code options that are currently defined for ILE are:

[no]foldfloat
Specifies that the system may evaluate constant floating-point expressions at compile time.

[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.

Note that for each 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 on the CHGMOD (Change Module), CHGPGM (Change Program), and CHGSRVPGM (Change Service Program) commands on the AS/400 in order to apply them to an existing object. The parameter name on the command is LICOPT. 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')
```

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, ILE 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.

OS/400 may define new Licensed Internal Code options in future releases (or within a given release via 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, ILE Program, or Service Program that has new options applied, to a release that does not support the options. However, the release must be V4R5M0 or later. The system ignores and no longer applies unsupported Licensed Internal Code options for re-translated objects if the LICOPT parameter of a command does not specify the options. This type of re-translation can occur when the system re-translates the object by using LICOPT(*SAME) on the CHGMOD, CHGPGM, or CHGSRVPGM commands. This type of re-translation also occurs when the system automatically translates the object. This does not prevent re-translation. On the other hand, any attempts to specify the same 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 AS/400 commands DSPMOD, DSPPGM, and DSPSRVPGM 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 re-translation of the object did not support them. For more information, please see the "Release Compatibility" on page 141 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 retranslated 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.

Chapter 13. Shared Storage Synchronization

Shared storage provides an efficient means for communication between two or more concurrently running threads. This chapter 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 which take advantage of shared storage, developers 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. Storage access ordering problems are also known as storage synchronization or memory consistency problems. Two or more cooperating threads relying on a specific ordering of updates to shared storage without serializing the storage accesses cause these problems. There could be a storage access ordering problem if the following condition exists: One thread stores values to two shared variables, and another thread has an implicit dependency on observing those updates in a certain order.

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 chapter briefly touch on race conditions.

You can avoid shared storage access ordering problems by ensuring that the system performs storage synchronization actions by using the threads reading from and writing to shared storage. In practice you can solve both types of problems simultaneously using a variety of thread synchronization mechanisms available on the AS/400.

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.

Only the following conditions require storage synchronization: Two or more threads are attempting 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                        Thread B
  --------------                  -------------
     Y = 1;                          print(X);
     X = 1;                          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 threads see that other threads' storage updates are vital to the correctness of the program. Let us consider a typical scenario which 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                     All Other Threads
    -------------------                     -------------------------
        DATA = 10                       loop until FLAG has value 1
        FLAG = 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.

Storage Synchronizing Actions

When an ordering of shared storage accesses is required, you must meet the following condition: All threads which 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 appearing prior to the synchronizing action in the thread's logical flow complete before those accesses appearing 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 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 constitute synchronization of storage 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
Message Queues	Enqueue, Dequeue	V3R2M0
Compare-and-Swap	Successful store to target	V3R1M0

Additionally, the following action, which is not synchronization of a thread, constitutes synchronization of a storage action:

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, it is necessary that all threads dependent on the access ordering 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 1 Solution

A preferred method for solving the problem in the example above is to avoid the dependency between the data and flag values altogether. You can do this by using a more robust thread synchronization scheme. Although you could employ many of the thread synchronization techniques, one that lends itself well to this problem is a semaphore. (Support for semaphores has been available since AS/400 Version 3, Release 2.)

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                          Wait for semaphore count to reach 0
        Decrement semaphore                use DATA
```

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
    -------------------------------------       -------------------------------------
    /* Do some work on the shared data */       /* Do some work on the shared data */
    for (int i=0; i<10; ++i) {                  for (int i=0; i<10; ++i) {
      /* Wait until the locked flag is clear */   /* Wait until the locked flag is clear */
      while (locked == 1) {                        while (locked == 1) {
        sleep(1);                                     sleep(1);
      }                                             }

      locked = 1;  /* Set the lock */             locked = 1;     /* Set the lock */

      /* Update the shared data */                /* Update the shared data */
      data1 += 5;                                 data1 += 4;
      data2 += 10;                                data2 += 6;

      locked = 0;  /* Clear the lock */           locked = 0;     /* Clear the lock */
    }                                           }
```

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 employing the compare-and-swap mechanism.

```
           THREAD A                                    THREAD B
-------------------------------------      -------------------------------------
for (i=0; i<10; ++i) {                     for (i=0; i<10; ++i) {
  /* Get an exclusive lock on the shared     /* Get an exclusive lock on the shared
     data.  We go into a wait state until       data.  We go into a wait state until
     the lock is granted. */                    the lock is granted. */
  locksl( LOCK_LOC, _LENR_LOCK );            locksl( LOCK_LOC, _LENR_LOCK );

  /* Update the shared data */               /* Update the shared data */
  data1 += 5;                                data1 += 4;
  data2 += 10;                               data2 += 6;

  /* Unlock the shared data */               /* Unlock the shared data */
  unlocksl( LOCK_LOC, _LENR_LOCK );          unlocksl( LOCK_LOC, _LENR_LOCK );
}                                          }
```

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 Compare-and-Swap

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 location 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 Compare-and-Swap. This is a lower-level mechanism that can to provide a very simple and fast locking protocol if there is little contention for the lock. In this solution, the system will use Compare-and-Swap to attempt to set the lock. If that attempt fails (because the system found the lock already set), the thread will wait for a while and then try again until acquiring the lock. After updating the shared data, the system will use Compare-and-Swap again to release the lock. In this example, assume that, in addition to the shared data items, the threads also share the address of a location "LOCK". Furthermore, assume that the lock was initialized to zero (either through static initialization or some prior synchronized initialization).

```
           THREAD A                                    THREAD B
-------------------------------------      -------------------------------------
/* Do some work on the shared data */      /* Do some work on the shared data */
for (i=0; i<10; ++i) {                     for (i=0; i<10; ++i) {

  /* Set the expected value for the lock */  /* Set the expected value for the lock */
  unlck_val = 0;                             unlck_val = 0;

  /* Attempt to set the lock using           /* Attempt to set the lock using
     Compare-and-Swap */                        Compare-and-Swap */
  while (_CMPSWP(&LOCK, &unlck_val, 1) == 0) {  while (_CMPSWP(&LOCK, &unlck_val, 1) == 0) {
    sleep(1);                                  sleep(1);
    unlck_val = 0;                             unlck_val = 0;
  }                                          }

  /* Update the shared data */               /* Update the shared data */
  data1 += 5;                                data1 += 4;
  data2 += 10;                               data2 += 6;
```

```
    /* Unlock the shared data... must use CMPSWP        /* Unlock the shared data... must use CMPSWP
       again to ensure other jobs/threads see              again to ensure other jobs/threads see
       update to shared data prior to clearing            update to shared data prior to clearing
       of the lock. */                                     of the lock. */
    _CMPSWP(&LOCK, &locked_val, 0);                    _CMPSWP(&LOCK, &locked_val, 0);
  }                                                  }
```

Here, the threads use Compare-and-Swap to perform a race-free test and update of
the lock variable. This solves the race condition experienced in the original
problem fragments. It also addresses the storage access ordering problem. As noted
earlier, Compare-and-Swap is a synchronizing action. Using Compare-and-Swap to
set the lock prior to reading the shared data, ensures that the threads will read the
most recently updated data. Using Compare-and-Swap to clear the lock after
updating the shared data ensures that the updates are available for subsequent
reads by any thread.

Appendix A. Output Listing from CRTPGM, CRTSRVPGM, UPDPGM, or UPDSRVPGM Command

This appendix 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 68.

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 49, Figure 50, and Figure 51 on page 151 show this information.

```
      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
    Authority   . . . . . . . . . . . . . . . . . . . . :      *LIBCRTAUT
      Text . . . . . . . . . . . . . . . . . . . . . . :

      Module     Library       Module     Library       Module     Library       Module     Library

      MONEY      MYLIB         CALCS      MYLIB
      RATES      MYLIB         ACCTS      MYLIB

      Service                  Service                  Service                  Service
      Program    Library       Program    Library       Program    Library       Program    Library

      *NONE

      Binding                  Binding                  Binding                  Binding
      Directory  Library       Directory  Library       Directory  Library       Directory  Library

      *NONE
```

Figure 49. Values Specified on CRTSRVPGM Command

Brief Summary Table

```
    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   * * * * *
```

Figure 50. Brief Summary Table

```
                              Binding Statistics

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  * * * * *
```

Figure 51. Binding Statistics

Extended Listing

If you specify DETAIL(*EXTENDED) on the CRTPGM, CRTSRVPGM, UPDPGM, 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). Figure 52,
Figure 53 on page 152, and Figure 54 on page 153 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

           * * * * *  E N D  O F  E X T E N D E D  S U M M A R Y  T A B L E  * * * * *
```

Figure 52. Extended Summary Listing

Binder Information Listing

```
Module . . . . . . . . . . . . :    MONEY
  Library . . . . . . . . . . :      MYLIB
  Bound  . . . . . . . . . . :       *YES

     Number    Symbol    Ref        Identifier              Type    Scope    Export    Key

     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

     Number    Symbol    Ref        Identifier              Type    Scope    Export    Key

     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

     Number    Symbol    Ref        Identifier              Type    Scope    Export    Key

     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

     Number    Symbol    Ref        Identifier              Type    Scope    Export    Key

     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
```

Figure 53. Binder Information Listing (Part 1 of 2)

```
Service program  . . . . . . . :    QC2SYS
  Library  . . . . . . . . . . :      *LIBL
  Bound  . . . . . . . . . . :        *NO

     Number    Symbol    Ref      Identifier              Type    Scope    Export    Key

     0000001F  Def                system                  Proc             Strong

Service program  . . . . . . . :    QLEAWI
  Library  . . . . . . . . . . :      *LIBL
  Bound  . . . . . . . . . . :        *YES

     Number    Symbol    Ref      Identifier              Type    Scope    Export    Key

     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
```

Figure 53. Binder Information Listing (Part 2 of 2)

```
                                    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: 00000000ADCEFEE088738A98DBA6E723.
STRPGMEXP  PGMLVL(*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
ENDPGMEXP
******** Export signature: 000000000000000000ADC89D09E0C6E7.


              * * * * *   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. Binder Language Listing

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. Figure 55 on page 154 shows a
partial example of the additional data provided.

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
CEEFMTM	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	00000183			*SRVPGM	*LIBL	QLEAWI
CEEMKHP	000001A2			*SRVPGM	*LIBL	QLEAWI
CEEMOUT	00000184			*SRVPGM	*LIBL	QLEAWI
CEEMRCR	00000182			*SRVPGM	*LIBL	QLEAWI
CEEMSG	00000185			*SRVPGM	*LIBL	QLEAWI
CEENCOD	00000186			*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
CEEUTCO	000001B7			*SRVPGM	*LIBL	QLEAWI
CEE4ABN	00000192			*SRVPGM	*LIBL	QLEAWI
CEE4CpyDvfb	0000019A			*SRVPGM	*LIBL	QLEAWI
CEE4CpyIofb	00000199			*SRVPGM	*LIBL	QLEAWI
CEE4CpyOfb	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 HdlrRouterEh	0000018F			*SRVPGM	*LIBL	QLEAWI
Q LE RtxRouterCh	0000018E			*SRVPGM	*LIBL	QLEAWI
Rate	0000000B			*MODULE	MYLIB	RATES
Term	0000000A			*MODULE	MYLIB	RATES

Figure 55. Cross-Reference Listing

Listing for Example Service Program

Figure 51 on page 151, Figure 53 on page 152, and Figure 55 on page 154 show some of the listing data generated when DETAIL(*FULL) was specified to create the FINANCIAL service program in Figure 38 on page 73. 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 (Figure 53 on page 152) 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 Figure 53 on page 152 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 Figure 55 on page 154 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 51 on page 151 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 56 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 56. 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 57 on page 158 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 57. 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 58 shows a binder language listing that contains this error.

```
                        Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(B)
ENDPGMEXP
********* Export signature: 00000000000000000000000000000CD2.
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 58. 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 59 shows a binder language listing that contains this error.

```
                    Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(B)
ENDPGMEXP
******** Export signature: 000000000000000000000000000000CD2.
STRPGMEXP  PGMLVL(*PRV)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(B)
ENDPGMEXP
******** Export signature: 000000000000000000000000000000CD2.
******** 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 59. 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 60 on page 160 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: 00000000000000000000000000CDED3.


          * * * * *  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. 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 61 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: 0000000000000000000000000000000C1.


          * * * * *  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 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 62 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: 000000000000000000000000000CDE3.
STRPGMEXP
  EXPORT SYMBOL(A)
******** Multiple 'current' export blocks not allowed, 'previous' assumed.
  EXPORT SYMBOL(B)
ENDPGMEXP
******** Export signature: 00000000000000000000000000000CD2.


       * * * * *   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. 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 63 on page 162 shows a binder language listing that contains this error.

```
                         Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
ENDPGMEXP
******** Export signature: 0000000000000000000000000000000000.
***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 63. 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 64 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 64. 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 65 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 65. 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 66 on page 164 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: 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 66. 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 67 shows a binder language listing that contains this error.

```
                              Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT SYMBOL(A)
   EXPORT SYMBOL(B)
ENDPGMEXP
******** Export signature: 000000000000000000000000000000CD2.
   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 67. 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 68 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: 000000000000000000000000000000FFC2.


    * * * * *  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 68. 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 69 shows a binder language listing that contains this error.

```
                        Binder Language Listing

STRPGMEXP  PGMLVL(*PRV)
   EXPORT SYMBOL(A)
ENDPGMEXP
******** Export signature: 000000000000000000000000000000C1.
***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 69. 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 70 on page 167 shows a binder language listing that contains this error.

```
STRPGMEXP PGMLVL(*CURRENT)
EXPORT ("Z"<<<)
***ERROR No matches of wildcard specification
EXPORT ("B"<<<)
ENDPGMEXP
******** Export signature: 0000000000000000000000000000FFC2.


   * * * * *   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. 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 71 shows a binder language listing that contains this error.

```
STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(B)
ENDPGMEXP
******** Export signature: 00000000000000000000000000000CD2.
STRPGMEXP  PGMLVL(*PRV)
ENDPGMEXP
******** Export signature: 00000000000000000000000000000000.
***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 71. 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 72 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: E05A838485868788899192939495969  7.

     * * * * *   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 72. 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 73 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 73. 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 74 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   * * * * *
```

Figure 74. What Is Specified for the Signature Value Is Not Valid

This is a serious error.

The signature contains invalid characters.

The service program is not created.

Suggested Changes

1. Remove invalid characters from the signature value.
2. Run the CRTSRVPGM command.

In this case, remove the " characters from the signature field.

Symbol Name Required

Figure 75 shows a binder language listing that contains this error.

```
                        Binder Language Listing

STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL(A)
  EXPORT SYMBOL(')
***ERROR Symbol name required.
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 75. 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.

Suggested Changes
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.

In this example, the source line EXPORT SYMBOL("") is removed from the binder language source file.

Symbol Not Allowed as Service Program Export

Figure 76 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: 000000000000000000000000000000CD4.


        * * * * *   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 76. 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 77 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: 000000000000000000000000000000CE8.


        * * * * *  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 77. 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.

Appendix B. Exceptions in Optimized Programs

In rare circumstances, an MCH3601 exception message may occur in programs compiled with optimization level 30 (*FULL) or 40. This appendix 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 caused by ILE attempting to calculate the first array element address prior to 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).

Appendix C. CL Commands Used with ILE Objects

The following lists indicate which CL commands can be used with each ILE object.

CL Commands Used with Modules

CHGMOD
  Change Module

CRTCMOD
  Create C Module

CRTCBLMOD
  Create COBOL Module

CRTCLMOD
  Create CL 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

CHGPGM
  Change Program

CRTBNDC
  Create Bound C Program

CRTBNDCBL
  Create Bound COBOL Program

CRTBNDCL
  Create Bound CL 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

CHGSRVPGM
>> Change Service Program

CRTSRVPGM
>> Create Service Program

DLTSRVPGM
>> Delete Service Program

DSPSRVPGM
>> Display Service Program

UPDSRVPGM
>> Update Service Program

WRKSRVPGM
>> Work with Service Program

CL Commands Used with Binding Directories

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 Command Used with Structured Query Language

CRTSQLCI
>> Create Structured Query Language ILE C/400 Object

CRTSQLCBLI
>> Create Structured Query Language ILE COBOL/400 Object

CRTSQLRPGI
>> Create Structured Query Language ILE RPG/400 Object

CL Commands Used with Source Debugger

DSPMODSRC
Display Module Source

ENDDBG
End Debug

STRDBG
Start Debug

CL Commands Used to Edit the Binder Language Source File

STRPDM
Start Programming Development Manager

STRSEU
Start Source Entry Utility

The following nonrunnable commands can be entered into the binder language
source file:

ENDPGMEXP
End Program Export

EXPORT
Export

STRPGMEXP
Start Program Export

Appendix D. 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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, 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.

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.

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
3605 Highway 52 N
Rochester, MN 55901-7829
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 information 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.

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 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 illustrates 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. 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 IBM's application programming interfaces.

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_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

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

Application System/400
AS/400
C/400
COBOL/400
DB2
DB2/400
IBM
ILE
Integrated Language Environment
Language Environment
Operating System/400
OS/2
OS/400
RPG/400
RPG IV
SAA
System 36
Systems Application Architecture
400

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. 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 in the United States and other countries licensed exclusively through The Open Group.

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

For additional information about topics related to the ILE environment on the AS/400 system, refer to the following IBM AS/400 publications:

- *Backup and Recovery*, SC41-5304-04 , 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.

- *CL Programming*, SC41-5721-03 , provides a wide-ranging discussion of AS/400 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 impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.

- *Communications Management*, SC41-5406-02 , 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*, SC41-5442-00 , provides information needed to write application programs that use AS/400 communications and the OS/400 intersystem communications function (OS/400-ICF). This guide also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.

- *ILE C for AS/400 Programmer's Guide*, SC09-2712-01 , provides information on how to develop applications using the ILE C/400 language. It includes information about creating, running, and debugging programs. It

also includes programming considerations for interlanguage program and procedure calls, locales, exception handling, database files, externally described files, and device files. Some performance tips are also described. An appendix includes information on migrating source code from EPM C/400 or System C/400 to ILE C/400.

- *ILE C for AS/400 Language Reference*, SC09-2711-01 , provides information about how to write programs that adhere to the Systems Application Architecture C Level 2 definition and use ILE C/400 specific functions such as record I/O. It also provides information on ILE C/400 machine interface library functions.

- , provides quick reference information about ILE C/400 command syntax, elements of C, SAA C library functions, ILE C/400 library extensions to SAA C, and ILE C/400 machine interface library extensions.

- *ILE COBOL for AS/400 Programmer's Guide*, SC09-2540-01 , describes how to write, compile, bind, run, debug, and maintain ILE COBOL/400 programs on the AS/400 system. It provides programming information on how to call other ILE COBOL/400 and non-ILE COBOL/400 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.

- *ILE COBOL for AS/400 Reference*, SC09-2539-01 , describes the ILE COBOL/400 programming language. It provides information on the structure of the ILE COBOL/400 programming language and on the structure of an ILE COBOL/400 source program. It also describes all Identification Division paragraphs, Environment Division clauses, Data Division paragraphs, Procedure Division statements, and Compiler-Directing statements.

- *ILE RPG for AS/400 Programmer's Guide*, SC09-2507-02 , is a guide for using the RPG IV programming language, which is an implementation of ILE RPG/400 in the Integrated Language Environment (ILE) on the AS/400 system. It includes information on 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 AS/400 files and devices in RPG programs. Appendixes 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.

- *ILE RPG for AS/400 Reference*, SC09-2508-02 , provides information needed to write programs for the AS/400 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*, SC41-5447-00 , provides information about interactive communications between two application programs on the same AS/400 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 on developing intrasystem communications application programs that use the OS/400 intersystem communications function (OS/400-ICF).

- *Security - Reference*, SC41-5302-04 , 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.

- *Work Management*, SC41-5306-03 , 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 on 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.

Index

Special Characters

A

C

C environment 6
C signal
   ILE C/400 40
call
   procedure 21, 91
   procedure pointer 91
   program 21, 91
call-level scoping 46
call message queue 39
call stack
   activation group example 28
   definition 91
   example
      dynamic program calls 91
      static procedure calls 91
callable service 129
Case component of condition token 113
CEE4ABN (Abnormal End) bindable
  API 111
CEE4DAS (Define Heap Allocation
  Strategy) bindable API 106
CEE9901 (generic failure) exception
  message 42
CEE9901 function check 40
CEECRHP (Create Heap) bindable
  API 103, 106
CEECRHP bindable API 104
CEECZST (Reallocate Storage) bindable
  API 105
CEEDOD (Retrieve Operational
  Descriptor Information) bindable
  API 98
CEEDSHP (Discard Heap) bindable
  API 102, 106
CEEFRST (Free Storage) bindable
  API 105
CEEGTST (Get Heap Storage) bindable
  API 105
CEEHDLR (Register User-Written
  Condition Handler) bindable API 44,
  107
CEEHDLU (Unregister User-Written
  Condition Handler) bindable API 44
CEEMGET (Get Message) bindable
  API 115
CEEMKHP (Mark Heap) bindable
  API 102, 106
CEEMOUT (Dispatch Message) bindable
  API 115
CEEMRCR (Move Resume Cursor)
  bindable API 109
CEEMSG (Get, Format and Dispatch
  Message) bindable API 115
CEENCOD (Construct Condition Token)
  bindable API 112
CEERLHP (Release Heap) bindable
  API 103, 106
CEESGI (Get String Information) bindable
  API 98
CEESGL (Signal Condition) bindable API
  condition token 112, 115
  description 40
CEETSTA (Test for Omitted Argument)
  bindable API 95
Change Exception Message
  (QMHCHGEM) API 109

Change Module (CHGMOD)
  command 118, 119
CHGMOD (Change Module)
  command 118, 119
CL (control language) command
  CHGMOD (Change Module) 119
  RCLACTGRP (Reclaim Activation
    Group) 88
  RCLRSC (Reclaim Resources)
    for ILE programs 88
    for OPM programs 88
code optimization
  errors 173
  levels 118
  performance
    compared to original program
      model (OPM) 6
    levels 25
    module observability 118
coexistence with existing applications 3
command, CL
  CALL (dynamic program call) 95
  CHGMOD (Change Module) 118
  CRTPGM (Create Program) 51
  CRTSRVPGM (Create Service
    Program) 51
  ENDCMTCTL (End Commitment
    Control) 125
  OPNDBF (Open Data Base File) 123
  OPNQRYF (Open Query File) 123
  RCLACTGRP (Reclaim Activation
    Group) 33
  RCLRSC (Reclaim Resources) 86
  STRCMTCTL (Start Commitment
    Control) 123, 125
  STRDBG (Start Debug) 117
  Update Program (UPDPGM) 78
  Update Service Program
    (UPDSRVPGM) 78
command, CL (control language)
  CHGMOD (Change Module) 119
  RCLACTGRP (Reclaim Activation
    Group) 88
  RCLRSC (Reclaim Resources)
    for ILE programs 88
    for OPM programs 88
commitment control
  activation group 125
  commit operation 124
  commitment definition 125
  ending 126
  example 4
  rollback operation 124
  scope 124, 125
  transaction 124
commitment definition 123, 125
Common Programming Interface (CPI)
  Communication, data
  management 124
Compare-and-Swap 147
component
  reusable
    benefit of ILE 2
condition
  definition 45
  management 107

condition *(continued)*
    bindable APIs (application
      programming interfaces) 129,
      131
    relationship to OS/400 message 114
Condition ID component of condition
  token 113
condition token 113
  Case component 113
  Condition ID component 113
  Control component 113
  definition 45, 112
  Facility ID component 113
  feedback code on call to bindable
    API 115
  Message Number component 113
  Message Severity component 113
  Msg_No component 113
  MsgSev component 113
  relationship to OS/400 message 114
  Severity component 113
  testing 114
Construct Condition Token (CEENCOD)
  bindable API 112
control boundary
  activation group
    example 36
  default activation group example 37
  definition 36
  function check at 110
  unhandled exception at 110
  use 38
Control component of condition
  token 113
control flow
  bindable APIs (application
    programming interfaces) 129
CPF9999 (function check) exception
  message 41
CPF9999 function check 40
Create Heap (CEECRHP) bindable
  API 103, 106
Create Program (CRTPGM) command
  ACTGRP (activation group) parameter
    activation group creation 31
    program activation 28, 31
  ALWLIBUPD (Allow Library
    Update) 79
  ALWUPD (Allow Update)
    parameter 78, 79
  BNDDIR parameter 53
  compared to CRTSRVPGM (Create
    Service Program) command 51
  DETAIL parameter
    *BASIC value 149
    *EXTENDED value 151
    *FULL value 153
  ENTMOD (entry module)
    parameter 60
  MODULE parameter 53
  output listing 149
  program creation 13
  service program activation 35
Create Service Program (CRTSRVPGM)
  command
  ACTGRP (activation group) parameter
    *CALLER value 88

Readers' Comments — We'd Like to Hear from You

AS/400e
ILE Concepts

Publication No. SC41-5606-04

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you?　☐ Yes　☐ No

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

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You

SC41-5606-04

Fold and Tape   **Please do not staple**   Fold and Tape
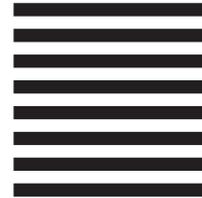
NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL   PERMIT NO. 40   ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
ATTN DEPT 542 IDCLERK
3605 HWY 52 N
ROCHESTER MN 55901-7829

Fold and Tape   **Please do not staple**   Fold and Tape

Readers' Comments — We'd Like to Hear from You

SC41-5606-04

IBM®

Spine information:

IBM

AS/400e

AS/400 ILE Concepts V4R5

Version 4