Chapter 6

Program Organization

The procedure specification is used to name and delineate RPG subprocedures. See chapter 2 for more information on all RPG IV specifications. Zero or more subprocedures can be included in a source file. The term *source file*, as used in this text, refers to a single container of RPG source. It relates directly to an AS/400 source file member or to a plain ASCII text file on another computer operating system. When a source file is compiled, an object of type *MODULE is created. A machine-readable RPG program is created by linking (referred to as *program binding*) one or more modules.

Machine-readable RPG IV programs consist of one or more *mainline procedures* and zero or more subprocedures. The mainline procedure is the entry point of an RPG program. It is the area traditionally referred to as the mainline calculations, but also includes the file, input, definition, and output specifications. The mainline procedure is the same as the program name. Hence, when an RPG program is called, the mainline procedure is being evoked.

Subprocedures are isolated by beginning and ending procedure specifications. These subprocedures can be called through the CALLP or CALLB operations. In addition, when a procedure is prototyped, it is also considered a function, and can be evoked similarly to RPG built-in functions.

SOURCE FILE LOCATION

Contemporary RPG IV compilers accept source code from the highly structure database native to IBM's OS/400 operating systems (on which RPG IV originated) or from the flat-file systems such as those available under Linux, UNIX and Microsoft Windows operating systems. A version of the flat-file systems, referred to as the Integrated File System or "IFS", is also built into OS/400.

The OS/400 RPG IV compiler accepts and compiles source code from either file system.

Under the native OS/400 database, source code is stored in source file members. Under this database structure, database files contain members, members are the final element in the structure and members contain the actual data. Whereas under a flat-file system, the source file contains the actual source code.

To resolve this difference when using the flat-file system to store and compile source, most application development environments encourage the programmer to create a directory structure that simulates the native OS/400 database file structure. For example:

Under OS/400, the database file named QRPGLESRC has been created in the library named ORDENTRY. Within QRPGLESRC there are five source members that contain RPG IV source code. Those member names are as follows:

1. ORDERS
2. CUSTMAINT
3. CUSTSCH
4. PRTORD
5. SHIPORD

To identify one of these source members to the compiler, the following command parameter syntax would be specified:

```
CRTBNDRPG  PGM(CUSTMAINT)  SRCFILE(ORDENTRY/CUSTMAINT) SRCMBR(CUSTMAINT)
```

Using the IFS to store source code is relatively uncommon in the OS/400 world, although flat-file systems are the standard practice for storing source code on all other operation systems. The CRTBNDRPG and CRTRPGMOD commands accept source from the IFS through the use of the SRCSTMF parameter.

The SRCSTMF parameter identifies the source file and the location of the source file in the IFS. When the SRCSTMF parameter is used the SRCFILE and SRCMBR parameters are not allowed. An error will occur if SRCFIEL and SRCMBR are specified with the SRCSTMF parameter. By default, the SRCSTMF parameter defaults to SRCSTMF(*NONE).

If an IFS directory structure named */mysrc/qrpglesrc* has been created to store source code, and the source files named ORDERS, CUSTMAINT, CUSTSCH, PRTORD, SHIPORD are stored in the directory, the file names should be suffixed with the traditional SEU source type. In the case of RPG IV, this would be RPGLE. So these files would be referred to as follows:

```
/mysrc/qrpglesrc/orders.rpgle
/mysrc/qrpglesrc/custmaint.rpgle
/mysrc/qrpglesrc/custsch.rpgle
/mysrc/qrpglesrc/prtord.rpgle
/mysrc/qrpglesrc/shipord.rpgle
```

To compile any of these source members using the CRTRPGMOD command, the following command could be used:

```
CRTRPGMOD MODULE(SHIPORD) SRCSTMF('/mysrc/qrpglesrc/shipord.rpgle')
```

SOURCE FILE MEMBER CONTENTS

A source file can contain just the mainline procedure, the mainline procedure and one or more subprocedures, or only subprocedures. When only subprocedures exist within a source file, a header specification within that source file must include the NOMAIN keyword. Figure 6.1 illustrates the structure of a source file that contains a mainline procedure and two subprocedures.

MODULE DEFINITION

RPG IV programs consist of one mainline procedure and zero or more subprocedures. A source file can contain just the mainline procedure, the mainline procedure and one or more subprocedures, or only subprocedures. When only subprocedures exist within a source file, the header specification within that source file should include the NOMAIN keyword. This keyword instructs the compiler to avoid automatically inserting the RPG runtime support for the *program logic cycle*. The program logic cycle typically is not required and it adds unnecessary overhead to the compiled object.

Source files that contain only subprocedures are permitted to contain data definitions outside the scope of any of the subprocedures.

Variables declared within the mainline procedure are called *global variables*. Variables declared outside of any subprocedure (where the mainline procedure would normally be placed) also are called global variables. Variables declared within subprocedures are called *local variables*. See the subheading Scope for more information on global and local variables.

Figure 6.2 contains the annotated source for an RPG program that contains a mainline procedure and one subprocedure.

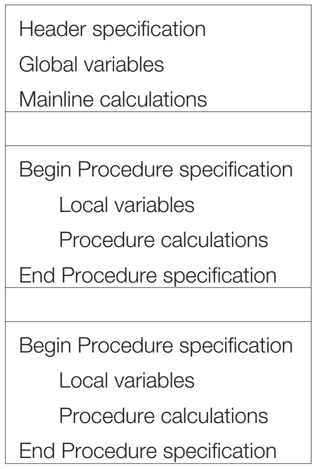

Figure 6.1: Source file structure with mainline procedure.

```
.....H.Functions+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     H DatFmt(*USA)

.....FFileName++IFEASFRlen+LKeylnKFDevice+.Functions+++++++++++++++++++++++++++
     FCustmast  UF   E            K DISK     PREFIX(CM_)
     D/COPY QRPGLESRC,STDINC
.....DName+++++++++++EUDS.......To/Len+TDc.Functions++++++++
     D Cust_Total      S             11P 2
     D** Prototype of the TOUPPER( char-var ) function
     D ToUpper         PR          1Ø24A
     D  tu_input                   1Ø24A    CONST

.....CSRnØ1.............OpCode(ex)Extended-factor2+++++++++
.....CSRnØ1Factor1+++++++OpCode(ex)Factor2+++++++Result++++++++Len++DcHiLoEq....
     C                   Read      CustRec                                58
     C                   Dow       NOT *IN58
     C                   If        ToUpper(CM_Cstnam) = ‘IBM CORP.’
     C                   DELETE    CustRec
     C                   Endif
     C                   Else
     C                   Add       Sales         Cust_Total
     C                   Read      CustRec                                58
     C                   EndDo
     C                   MOVE      *ON           *INLR
     C                   return


     PProcedure++++++..BE.................Functions++
     P ToUpper         B
     DName+++++++++++EUDS.......Length+TDc.Functions++
     D ToUpper         PI          1Ø24A
     D  InputStg                   1Ø24A    CONST

     D RtnValue        S                    LIKE(InputStg)
     D lower           C                    Const(‘abcdefghijklmnopqrstuvwxyz’)
     D upper           C                    Const(‘ABCDEFGHIJKLMNOPQRSTUVWXYZ’)

     CSRnØ1Factor1+++++++OpCode(ex)Factor2+++++++Result++++++++Len++DcHiLoEq
     C     lower:UPPER   xLate     InputStg      RtnValue
     C                   Return    RtnValue
     P ToUpper         E
```

These definition specifications are the prototype for the TOUPPER procedure. Note the TU_INPUT field is simply a place holder. It is not a variable nam,e that can be

Body of program “mainline” procedure.

The P specification is used (albeit redundantly) to delineate the beginning and end of a procedure.

The definition specifications define the procedure interface. These are the INPUT and OUTPUT parameters.

Figure 6.2: Source member with embedded procedures.

Modules are defined at the source file level. RPG programs can call other programs dynamically or statically. They also can call subprocedures; however, all subprocedures require the static call interface.

As shown in Figure 6.2, variables can be defined within the mainline RPG code or in a subprocedure. In this example, the subprocedure TOUPPER is embedded within the same source file as the mainline program. The TOUPPER procedure is called to convert a character string to capital letters.

When source files are compiled, the output from the compiler is a module. If a program is made up of only one source file, then only one module is generated. If a program is made up of multiple source files, multiple modules are generated. Those modules must be bound together to create a single program object that can be called and run. This program object is referred to as a *machine readable object* or as *executable*. Figure 6.3 illustrates this source-to-module-to-program transition.

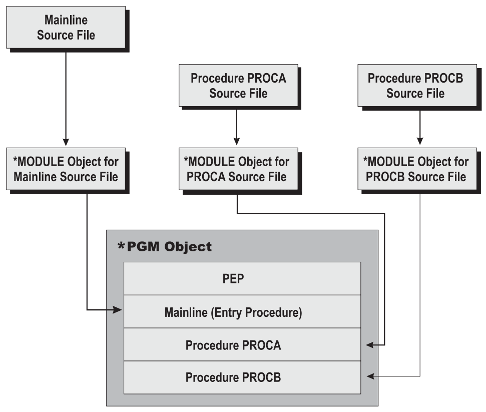
Mainline Source File
Procedure PROCA Source File
Procedure PROCB Source File
*MODULE Object for Mainline Source File
*MODULE Object for PROCA Source File
*MODULE Object for PROCB Source File
*PGM Object
PEP
Mainline (Entry Procedure)
Procedure PROCA
Procedure PROCB

*Figure 6.3: Source file transition to *PGM object.*

RPG allows separately compiled and independent programs to be bound together to form a single program object. The source file for each program is compiled to generate a module. Each module can be bound to create independent program objects. The individual modules, however, also can be bound together to create one larger program object.

There are two distinctions between separately compiled, independent programs and single programs that are composed of several bound modules:

- The single bound program is larger and is, by definition, a single-program object. In contrast, the separately compiled and separately bound programs are independent objects.
- The method of evoking one program from the other for a single-bound program is through the CALLB operation. For independent programs, the CALL operation is used.

There are four types of RPG IV source program organizations that can generate an object module. These types of source programs, and the commands used to compile them, are listed in Table 6.1 and explained in the text that follows.

Table 6.1: Source Program Structures

Description	Compiler Command	Service Program[1]	Activation Group
Traditional, single-source file (all in one) RPG program.	CRTBNDRPG	No	*DFTACTGRP
Single-source file that utilizes one or more embedded subprocedures.	CRTBNDRPG	Yes	QILE (i.e., named activation group)
Source file that uses zero or more embedded subprocedures, and one or more external subprocedures.	CRTRPGMOD	Yes	*NEW
Source files containing only subprocedures.	CRTRPGMOD	N/A	N/A

[1]Indicates whether the generated *MODULE can be bound into a service program. Also requires the NOMAIN keyword on the Header specification.

Traditional, Single-Source File RPG Program

The traditional, single-source file RPG type of program organization is typically used in legacy applications or in new applications that have a limited scope. These kinds of programs can be statically bound together by using the CRTRPGMOD and the CRTPGM commands instead of the single-source-to-program CRTBNDRPG command. If one large bound program is created, the CALL operations must be changed to the CALLB operation.

Single-Source File Using Embedded Subprocedures

The single-source file with embedded subprocedures is another kind of all-in-one program. The program is contained within a single source file and contains one or more subprocedures to perform various tasks. This type of program can be compiled using the single-source-to-program CRTBNDRPG command. However, the defaults for the command need to be changed. Specifically, on the AS/400, programs that contain subprocedures cannot be run in the default activation group. They must be run in another activation group. When compiling this type of source file, typically, the QILE activation group is specified.

Multiple Source Files with Embedded or External Subprocedures

For new applications, the most common type of program has multiple source files with embedded or external subprocedures. Typically, there is a main *application source file*. It contains call interfaces (i.e., CALL, CALLB, CALLP, or prototypes) to the other programs and subprocedures. The CRTRPGMOD compiler command is used to generate modules for each separate source file. Procedures used by the application should probably be exported with the EXPORT keyword. This allows them to be evoked from other modules within the application.

A typical scenario for this type of application is one where the application source file contains the mainline procedure and controls the program's flow (logic). The other source files contain supporting subprocedures. Normally, the header specification is embedded in the secondary source files with the NOMAIN keyword specified. This allows the compiler to avoid generating redundant RPG program cycle code directly into each module. Only the mainline procedure requires the RPG program cycle code.

SCOPE

As illustrated in Figure 6.1, a source file containing the mainline RPG program code (mainline procedure) includes all global data. This includes fields, arrays, data structures, constants, files, input fields, and output fields.

The term *scope* is used to define the limit of visibility of a program item such as a field. For example, all fields have a scope based on the program structure. A field has a scope, which is global or local, that limits visibility of the field as follows:

Scope	Visibility
Local	Subprocedure
Global	Source file/module

If a subprocedure is specified, it has access to all global variables within the same source file. In addition, it has its own local variables. Local variables are available only to the subprocedure in which they are declared. Figure 6.4 illustrates the scope of variable definitions within RPG IV modules and subprocedures. Remember, for purposes of this text, the term source file means AS/400 source file member.

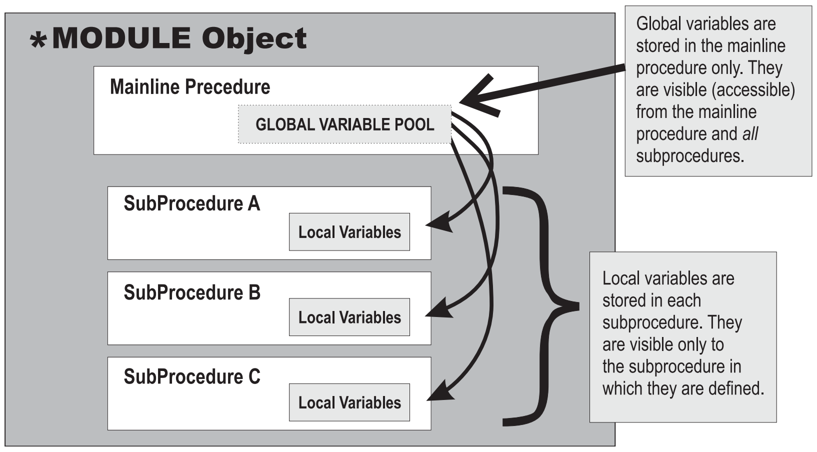
*MODULE Object
Mainline Precedure
GLOBAL VARIABLE POOL
SubProcedure A
Local Variables
SubProcedure B
Local Variables
SubProcedure C
Local Variables
Global variables are stored in the mainline procedure only. They are visible (accessible) from the mainline procedure and *all* subprocedures.
Local variables are stored in each subprocedure. They are visible only to the subprocedure in which they are defined.

Figure 6.4: Scope of global and local variables.

Local variables of the same name as a global variable are supported in RPG. In this situation, within the subprocedure, the local variable has precedence over the global variable. The properties of the local and global variables of the same name need not be similar. For example, a global field named ITEM could be defined as a 10-position character field within the mainline procedure. Within a subprocedure, the ITEM field (such as a five-digit packed decimal field) can be defined differently. Typically, this kind of situation is avoided through program design.

All global variables within a source file are visible only to the mainline procedure and the subprocedures that are embedded in that source file. For example, an application is made up of two source files named MYAPP and TOOLS. The source file named MYAPP contains a global variable named CUSTNAME. Subprocedures included in the MYAPP source file have access to CUSTNAME.

The mainline procedure (if one exists) and all subprocedures specified in the TOOLS source file do not have access to the CUSTNAME field.

STATIC AND AUTOMATIC STORAGE

Static storage is part of the program's memory that is retained for the duration of the program's or procedure's runtime. *Automatic storage* is part of the program's memory that is automatically allocated and released each time the procedure in which it is declared is called and ended respectively. Within a subprocedure, local variables are, by default, declared in automatic storage.

The STATIC keyword can be used on the definition specification within a subprocedure to force a local variable to have the STATIC storage attribute. When a local variable is set to STATIC, its content remains unchanged each time the subprocedure is evoked. There is only one copy of the variable for the duration of the entire runtime of the program. This includes recursively called subprocedures.

For example, if subprocedure P1 calls subprocedure P2, and P2 calls P1 (which is perfectly valid in RPG IV), variables declared as automatic storage (which is the default property) have new instances created (i.e., a second copy of the variable is automatically generated). Variables declared as static storage are assigned a storage location only the first time the subprocedure is evoked. Subsequent invocations use the original storage locations. This process doesn't affect the data within that location. Hence, the data stored within a static storage variable is available to each invocation.

IMPORT AND EXPORT

Any global variable (declared with the definition specification) can be exported for use by other modules with the same application program. The EXPORT keyword is used to export a global variable. This property makes the variable available to separately compiled source modules. Figure 6.5 illustrates how to code EXPORT and IMPORT keywords in separate source files.

Figure 6.5 illustrates two independent RPG source files. The first source file, STATUSONE (on the left of Figure 6.5), contains a variable named STATUSCODE. This variable has the EXPORT keyword specified. This field property means that the field is defined in STATUSCODE and resides within it, but is available for importing by any other module.

The other source file, STATUSCHK (shown on the right in Figure 6.5), is called by STATUSONE. The CALLP operation on line 5 performs the CALL operation to the STATUSCHECK file. Within the STATUSCHK module, the field STATUSCODE is also defined (line 2). In STATUSCHK, however, the STATUSCODE field contains the keyword IMPORT. This field property indicates that storage for the field is not allocated within the STATUSCHK module, but rather is located (i.e., *resolved*) when the related modules are bound together.

Entry Module: StatusOne

Called to start program; calls STATUSCHK

```
.....H++++++++++++++++++++++++++++++++++++++++++++++
0001 H
      * Source Module: STATUSONE

.....DName+++++++++++EUDS.......Length+TDc.Functions+
0002 D StatusCode      S              5I 0 EXPORT

0003 D StatusCheck     PR

.....CSRn01Factor1+++++++OpCode(ex)Factor2+++++++++++
0004 C                   Eval      StatusCode = 1
0005 C                   CallP     StatusCheck
0006 C                   Eval      *INLR = *ON
0007 C                   return
```

Bound-In Module: StatusChk

Called by STATUSONE

```
.....H+++++++++++++++++++++++++++++++++++++++++++++++
0001 H
      * Source Module: STATUSCHK

.....DName+++++++++++EUDS.......Length+TDc.Functions+
0002 D StatusCode      S              5I 0 IMPORT

0003 D NoStatus        PR
0004 D Billing         PR
0005 D LateCharge      PR


.....CSRn01Factor1+++++++OpCode(ex)Factor2+++++++++++
0006 D StatusCheck     PR
0007 P StatusCheck     P                   EXPORT
0008 C                   Select
0009 C                   When      StatusCode = 0
0010 C                   CallP     NoStatus
0011 C                   When      StatusCode = 1
0012 C                   CallP     Billing
0013 C                   When      StatusCode = 2
0014 C                   CallP     LateCharge
0015 C                   Other
0016 C                   Exsr      Errors
0017 C                   EndSL
0018 C                   Return
0019 CSR   Errors        BegSr
0020 C                   Eval      StatusCode = -1
0021 CSR                 EndSr
0022 P StatusCheck     E
```

Figure 6.5: Independent source files.

Only one copy of STATUSCODE exists, and its storage is allocated within STATUSONE. Any changes to this field in any module are immediately, and instantly, reflected in the other modules. They all reference the same storage location.

Figure 6.6 shows how modules within a single program share exported variables. The module in which the variable is defined is the module where the variable resides. Other modules that want to access the exported variable declare a variable with the same properties and specify the IMPORT keyword. When the modules are combined by the program binder, references to the exported variable (that is, all imported variables) are resolved. The imported variables are, essentially, pointers to the exported variable. Keep in mind the following points when using IMPORT and EXPORT:

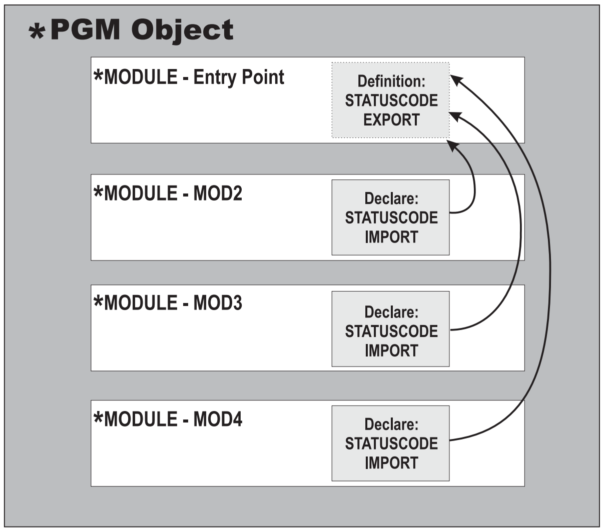
*PGM Object
*MODULE - Entry Point
Definition: STATUSCODE EXPORT
*MODULE - MOD2
Declare: STATUSCODE IMPORT
*MODULE - MOD3
Declare: STATUSCODE IMPORT
*MODULE - MOD4
Declare: STATUSCODE IMPORT

Figure 6.6: Modules sharing IMPORT/EXPORT data.

- Only global variables can be imported or exported. Local variables (fields declared within a subprocedure) cannot be exported.
- The module that contains the variable definition and storage is the one that contains the EXPORT keyword.
- To access an exported variable from within a subprocedure in a separate module, declare a global variable in the second module, specify the IMPORT keyword, and refer to that global variable within the subprocedure.
- Subprocedure names must be exported (using the EXPORT keyword) in order to access them from other modules.