Redpaper

Gary Mullen-Schultz
Susan Gantner
Jon Paris
Paul Tuohy

RPG: Exception and Error Handling

The previous IBM® Redbooks® publication on RPG (*Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402) briefly covered the topic of exception and error handling, mostly from the perspective of ILE exception handlers. Since that time a number of things have happened.

- First, interest in error handling has grown among RPG programmers, particularly with the increased use of subprocedures.
- Second, new error handling techniques have been introduced into the RPG language itself through the MONITOR opcode.

For these reasons we felt that it was appropriate to handle the topic in a more comprehensive manner.

This publication provides an overview of the traditional OPM style methods of implementing exception/error handling and how they change in the ILE environment. It also discusses how you may use "pure" ILE methods to completely replace some of the more traditional methods.

Introduction

Exception/error handling is the term given to handling unexpected exception/errors in programs. No matter how well you test your programs there is always the possibility that an unforeseen error may occur (for example, a divide by zero, an array index error, a duplicate record on a write to a file).

The most common indication that you have an unexpected exception/error is that a program fails with a CPF message. This can be disastrous if the program fails in a production environment and a user receives the CPF message; he will either have a protracted conversation with the help desk or, even worse, will simply press Enter and, by default, cancel the program.

Users should never, ever see the "two-line screen of death" caused by an unhandled CPF message. Exception/error handling provides all the tools you need to ensure that, should a program fail, it will terminate in an orderly fashion, display a friendly message to the user, notify the help desk that there is a problem, and provide enough documentation for a programmer to be able to determine what caused the problem.

There is also the case where you may want to trap exception/errors for a specific operation or group of operations and handle them in the program. This applies especially to file operations where it is quite reasonable for a program to handle a duplicate record condition on a WRITE operation or a constraint violation on a DELETE operation.

What is an exception/error?

Your program makes a request (an RPG operation) to the operating system. The request fails and the operating system responds by sending an escape message to the program message queue. When the escape message arrives, a check is made to see if the program is going to handle the error (MONMSG in CL) and, if not, default error handling kicks in. You should be aware that there is a fundamental difference between default error handling for OPM and for ILE. For OPM default error handling means that a CPF message is immediately sent to indicate that the program failed, but for ILE the error message is first "percolated" back up the call stack to see if the calling procedure is going to handle the error. This process continues until the boundary of the activation group is reached. You will see the difference between OPM style and ILE style exception/error handling as you progress through this paper.

If you do not want the RPG default error handler to handle exception/errors, RPG provides a number of alternative methods:

- Trap exception/errors for a complete program using a *PSSR subroutine.
- Trap exception/errors for individual files using an INFSR subroutine.
- Trap exception/errors for a single operation using an E extender.
- Trap exception/errors for a group of operations using a MONITOR group.

Throughout this paper you will see how all of these alternative methods are handled starting with handling unexpected exception/errors at a program level. Our examples will illustrate a simple method to consistently provide orderly shutdown of the program.

Trapping at the program level

Let's start with the traditional methods of handling exception/errors in RPG programs that run in OPM style (that is, that run in the default activation group). In "Subprocedures and exception/errors" on page 17 you will see how most of these traditional methods may be replaced in an ILE environment.

RPG differentiates between program exception/errors (divide by zero, array index error etc.) and file exception/errors (record lock, duplicate record. etc.).

The sample program ERROR01 (Example 1) demonstrates two conditions that can cause an exception/error.

Example 1 Sample program ERROR01

```
FProduct1  UF   E           K Disk    UsrOpn

D DummyCode       S                   Like(ProdCd)
```

```
     D a               S             10I 0 Inz(20)
     D b               S             10I 0
     D c               S             10I 0

      /Free
          c = a/b;

          Chain DummyCode Product1;
          Open Product1;

          *InLR = *On;
      /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR01) SRCFILE(REDBOOK/EXCEPTSRC)
```

The first exception/error is on the divide operation which, since *b* has a value of 0, will result in an `RNQ0102 Attempt to divide by zero` message being issued.

The second exception/error occurs on the CHAIN operation which, since an attempt is made to retrieve a record before opening the file, will result in an `RNQ1211 I/O operation was applied to closed file PRODUCT1` message being issued.

Of course, when you run this program you will only see the divide by zero error since, once the error occurs, you are only presented with options to cancel the program, continue at *GETIN (that is, the start of the mainline), or obtain a dump.

Program exception/errors

If a program encounters an otherwise unhandled program exception/error, it checks to see if there is a *PSSR subroutine coded in the program. If there is, it executes the subroutine; if not, the program “fails” with an error message.

ERROR02 (Example 2 is the same sample program as ERROR0,1 but with a *PSSR subroutine included.

Example 2 Sample program ERROR02

```
     FProduct1  UF   E           K Disk    UsrOpn

(1)  D PSSRDone        S               N
     D FatalProgram    PR                  ExtPgm('FATALPGM')

     D DummyCode       S                   Like(ProdCd)
     D a               S             10I 0 Inz(20)
     D b               S             10I 0
     D c               S             10I 0

      /Free
          c = a/b;

          Chain DummyCode Product1;
          Open Product1;
          *InLR = *On;

          BegSR *PSSR;
(2)          Dump(A);
(3)          If Not PSSRDone;
                PSSRDone = *On;
                FatalProgram();
             EndIf;
(4)       EndSR '*CANCL';
      /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR02) SRCFILE(REDBOOK/EXCEPTSRC)
```

Later, you will see how the *PSSR routine may be placed in a copy member but, for now, the main points to note about the inclusion of the *PSSR routine are:

- The PSSRDone indicator and the prototype for FatalProgram would usually be included in a standard copy member as with prototypes and standard field definitions. You will see an example of a FatalProgram in a moment.
- The (A) extender on the DUMP operation means that the dump will A(lways) take place, regardless of whether the program was compiled with the DEBUG option.
- The reason for the PSSRDone indicator is in case there is an exception/error in the *PSSR subroutine itself—which would cause the *PSSR subroutine to be executed, which would fail—until the number of dump listings causes a serious problem on the system.
- One of the issues with a *PSSR is that when it is invoked by an error, it does not provide an option to return to the statement in error. Therefore, it should be used as a means of performing an orderly exit from the program. In theory, the EndSR for a *PSSR subroutine can contain a return-point instruction, but most of them only apply if you are using the RPG cycle, and even then are of marginal to dubious utility. Here, the value of *CANCL indicates that the program should cancel when it reaches the end of the subroutine but, as you will see, this should never actually happen because the fatal program never returns from the call.

Now, the divide by zero error will result in the *PSSR subroutine being executed, as opposed to the RNQ0102 message being issued.

The fatal program

FATALDSPF is a display file (Example 3) used to display a meaningful message to the user if an interactive job fails.

Example 3 Sample program FATALDSPF

```
   A          R FATALERROR
   A                                 10 21'Oh Dear. It has all gone terribly -
   A                                      wrong!'
   A                                      COLOR(RED)
```

FATALPGM is a CL program that gets called when a program fails with an unexpected exception/error:

```
        DclF  RedBook/FatalDsp
        Dcl   &Job        *Char (10)
        Dcl   &User       *Char (10)
        Dcl   &Nbr        *Char (6)
        Dcl   &Type       *Char (1)
        Dcl   &Reply      *Char (1)

        DspJobLog Output(*Print)
        SndMsg ('It has all just gone horribly wrong!') ToMsgQ(QSysOpr)

        RtvJobA Job(&Job) User(&User) Nbr(&Nbr) Type(&Type)
ReShow:
        If (&Type = '1') SndRcvF RcdFmt(FATALERROR)
        Else SndUsrMsg  Msg('You must end Job ' *Cat &Nbr *TCat '/' +
                            *Cat &User *TCat '/' *CAT &Job *BCat +
                            'now. Error reports have been produced') +
                            ToMsgQ(*Sysopr) MsgRpy(&Reply)

        Goto ReShow
```

Create this program with the command:

```
CRTDSPF FILE(REDBOOK/FATALDSP) SRCFILE(REDBOOK/EXCEPTSRC)
CRTBNDCL PGM(REDBOOK/FATALPGM) SRCFILE(REDBOOK/EXCEPTSRC)
```

The program prints a job log, sends a message to the system operator and either displays a meaningful/helpful screen for the user (if running interactively) or sends an inquiry message to the system operator (if running in batch). The screen keeps being displayed or the message being sent until the job is cancelled.

The complexity of the program called from the *PSSR subroutine is up to you. It does not have to be a CL program; it could be another RPG program. You might decide to have the program perform some sort of logging. You might pass an information data structure and/or status codes as parameters. It all depends on the level of detail you want to go to.

You decide what you want to put on the screen format FATALERROR in the display file FATALDSP—just keep it friendly and non-threatening. It is often useful for your screen to include an entry area where you can solicit information from the user about what they were doing at the time. Were they doing anything different or unusual? Did anything else odd happen today? Thank them for helping and apologize to them for the error—being overly polite makes sense if you have made an error!

But with the changes we have suggested so far, at least if a program fails in a production environment the user is looking at a helpful screen and you have a program dump and a job log to help identify the problem.

You will see another example of the type of functions that can be performed by a program such as this in “Identify a percolated message” on page 24.

File exception/errors

You now change the sample program so that it no longer fails with a decimal data error by simply specifying a non-zero value for b. But now, when you run the program, it fails with an `RNQ1211 I/O operation was applied to closed file PRODUCT1` message. The automatic execution of the *PSSR routine does not apply to file errors in the same way that it applies to program errors.

Just as a program has a *PSSR subroutine to process unhandled program exception/errors, each file can have its own subroutine to handle file exception/errors. This subroutine is identified using the INFSR keyword on the F spec for the file. This method works for traditional style programs but will not work if a file is referenced in a subprocedure. You will see how to handle file exception errors in subprocedures in “Trapping percolated errors” on page 23.

Using a separate subroutine per file made a lot of sense when using the RPG cycle, but is of little use in a full procedural environment. Each subroutine simply needs to provide an orderly means of exiting the program and you already have that in your *PSSR routine. So one way of handling file errors is to simply identify the *PSSR routine on the INFSR keyword for each file in your program as at 1 below in the program ERROR03 (Example 4).

Example 4   Sample program ERROR03

```
     FProduct1  UF   E           K Disk    UsrOpn
(1)  F                                     InfSR(*PSSR)

     D PSSRDone        S               N
     D FatalProgram    PR                  ExtPgm('FATALPGM')

     D DummyCode       S                   Like(ProdCd)
     D a               S             10I 0 Inz(20)
     D b               S             10I 0 Inz(10)
     D c               S             10I 0

      /Free
          c = a/b;

          Chain DummyCode Product1;
          Open Product1;

          *InLR = *On;

          BegSR *PSSR;
             Dump(A);
             If Not PSSRDone;
                PSSRDone = *On;
                FatalProgram();
             EndIf;
          EndSR '*CANCL';
      /End-Free
```

Create this program with the command:

```
   CRTBNDRPG PGM(REDBOOK/ERROR03) SRCFILE(REDBOOK/EXCEPTSRC)
```

Now, with one exception, if there is an exception/error on a file operation, the *PSSR subroutine is executed.

The one exception is with the opening of files. Normally the files are automatically opened by the RPG cycle. But this process occurs before any user code is invoked, therefore it cannot execute *PSSR if there is a problem during file open (for example, level check error).

To trap file open errors, you need to perform an implicit open of the file by specifying conditional open for the file (USROPN keyword on F Spec) and use the OPEN operation to open the file (usually coded in the *INZSR subroutine).

Can it get any easier?

Since the *PSSR routine that we are suggesting will be the same in every program, why not code it in its own member and include it in any program using a /COPY directive? In further program examples the *PSSR subroutine has been placed in a member named PSSRSTD. Of course, this means that it is much simpler to enhance your error handling procedures since changing this one piece of code will enhance all of the programs that use it.

The coding requirement for basic exception/error handling is to define an INFSR(*PSSR) for every file and a copy directive to include the *PSSR subroutine.

In "Trapping percolated errors" on page 23 you will see how a *PSSR routine need only be coded in one program when programs are running in an ILE environment.

Trapping at the operation level

One program's error is another program's feature. Certain exception/errors may be expected in a program, in which case you may not want the *PSSR routine to be invoked. For example, you may want your program to be able to trap and handle exceptions for record locking, constraint violations, certain decimal data errors, etc.

Exception/errors may be trapped at the operation level using error extenders or MONITOR groups.

Error extender

Some operation codes offer the equivalent of a command level MONMSG in CL. This is implemented through the "error extender" (replacing the old error indicator in the low position) and allows the programmer to respond to errors instead of the operation failing. Instead, if it receives an error, the %ERROR BIF is set and processing continues with the next operation. It is up to the programmer to determine whether or not there was an exception/error and what to do about it.

Program ERROR04 (Example 5) demonstrates the use of the E extender.

Example 5 Sample program ERROR04

```
FProduct1  UF   E            K Disk    UsrOpn
F                                      InfSR(*PSSR)

D PSSRDone        S              N
D FatalProgram    PR                   ExtPgm('FATALPGM')

D DummyCode       S                    Like(ProdCd)
D a               S             10I 0 Inz(20)
D b               S             10I 0 Inz(10)
D c               S             10I 0

 /Free
```

```
            c = a/b;

(1)         Chain(E) DummyCode Product1;
(2)         If %Error();
               Dsply 'The file has not been opened';
            EndIf;

(3)         Chain DummyCode Product1;
            Open Product1;

            *InLR = *On;

      /Copy EXCEPTSRC,PSSRSTD
      /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR04) SRCFILE(REDBOOK/EXCEPTSRC)
```

The points to note are:

- The E extender on the CHAIN operation means that the program continues to execute with the next operation as opposed to executing the subroutine specified on the INFSR (i.e., *PSSR) or failing with an error message.
- You check for an error using the %ERROR BIF. In this example, the DSPLY operation is used to display a message if there was an exception/error on the CHAIN operation.
- Processing continues with the next CHAIN operation which, as before, results in the execution of the *PSSR subroutine.

Sadly, the use of the E extender does not apply to all operation codes but it does apply to all file and "external related" operation codes (CALL, IN, OUT). Later, you will see how to use MONITOR for the other operation codes. Table 1 lists the operation codes eligible for an E extender or error indicator.

Table 1 Operation codes eligible for an E extender

ACQ	CLOSE	OCCUR	REL	TEST
ADDDUR	COMMIT	OPEN	RESET	UNLOCK
ALLOC	DEALLOC	NEXT	ROLBK	UPDATE
CALL	DELETE	OUT	REALLOC	WRITE
CALLB	DSPLY	POST	SCAN	XLATE
CALLP	EXFMT	READ	SETGT	XML-INTO
CHAIN	EXTRCT	RAEDE	SETLL	XML-SAX
CHECK	FEOD	READP	SUBDUR	
CHECKR	IN	READPE	SUBST	

Status codes

So how do you know which exception/error has occurred? The %STATUS BIF provides a five-digit status code that identifies the error. Program status codes are in the range 00100 to 00999 and File status codes are in the range 01000 to 01999. Status codes in the range 00000 to 00050 are considered to be normal (i.e., they are not set by an exception/error condition).

Status codes correspond to RPG runtime messages in the message file QRNXMSG (e.g., message RNQ0100 = status code 00100). You can view the messages using the command:

```
   DSPMSGD RANGE(*FIRST *LAST) MSGF(QRNXMSG) DETAIL(*FULL)
```

Table 2 lists some of the more commonly used status codes. Refer to the ILE RPG Reference manual (RPG IV Concept>File and Program Exception/Errors) for a full list of status codes.

Table 2   Commonly used status codes

Code	Description
00100	Value out of range for string operation
00102	Divide by zero
00112	Invalid Date, Time or Timestamp value.
00121	Array index not valid
00122	OCCUR outside of range
00202	Called program or procedure failed
00211	Error calling program or procedure
00222	Pointer or parameter error
00401	Data area specified on IN/OUT not found
00413	Error on IN/OUT operation
00414	User not authorized to use data area
00415	User not authorized to change data area
00907	Decimal data error (digit or sign not valid)
01021	Tried to write a record that already exists (file being used has unique keys and key is duplicate, or attempted to write duplicate relative record number to a subfile).
01022	Referential constraint error detected on file member.
01023	Error in trigger program before file operation performed.
01024	Error in trigger program after file operation performed.
01211	File not open.
01218	Record already locked.
01221	Update operation attempted without a prior read.
01222	Record cannot be allocated due to referential constraint error
01331	Wait time exceeded for READ from WORKSTN file.

It is useful to define a copy member that contains the definition of named constants that correspond to the status codes. This provides for “self-documenting” code. For example, the copy member ERRSTATUS contains the following definitions corresponding to the status codes shown in Example 6.

Example 6   Copy member ERRSTATUS

```
     D ERR_VALUE_OUT_OF_RANGE...
     D                 C                   00100
     D ERR_DIVIDE_BY_ZERO...
     D                 C                   00102
     D ERR_INVALID_DATE...
     D                 C                   00112
     D ERR_ARRAY_INDEX...
     D                 C                   00121
     D ERR_OCCUR_OUT_OF_RANGE...
     D                 C                   00122
     D ERR_CALL_FAILED...
     D                 C                   00202
     D ERR_CALLING...
     D                 C                   00211
     D ERR_POINTER_OR_PARAMETER...
     D                 C                   00222
     D ERR_DATA_AREA_NOT_FOUND...
     D                 C                   00401
     D ERR_IN_OR_OUT...
     D                 C                   00413
     D ERR_USE_DATA_AREA...
     D                 C                   00414
     D ERR_CHANGE_DATA_AREA...
     D                 C                   00415
     D ERR_DECIMAL_DATA...
     D                 C                   00907
     D ERR_DUPLICATE_WRITE...
     D                 C                   01021
     D ERR_REFERENTIAL_CONSTRAINT...
     D                 C                   01022
     D ERR_TRIGGER_BEFORE...
     D                 C                   01023
     D ERR_TRIGGER_AFTER...
     D                 C                   01024
     D ERR_NOT_OPEN...
     D                 C                   01211
     D ERR_RECORD_LOCKED...
     D                 C                   01218
     D ERR_UPDATE_WITHOUT_READ...
     D                 C                   01221
     D ERR_CANNOT_ALLOCATE_DUE_TO_RC...
     D                 C                   01222
     D ERR_WAIT_EXCEEDED...
     D                 C                   01331
```

The program ERROR04 simply assumed that the error being trapped was for the status 1211 (file not open). Program ERROR05 (Example 7) shows how to ensure that only a status of 1211 is being trapped.

Example 7 Sample program ERROR05

```
     FProduct1  UF   E           K Disk    UsrOpn
     F                                     InfSR(*PSSR)

     D PSSRDone        S               N
     D FatalProgram    PR                  ExtPgm('FATALPGM')

     D DummyCode       S                   Like(ProdCd)
     D a               S             10I 0 Inz(20)
     D b               S             10I 0 Inz(10)
     D c               S             10I 0

(1)   /Copy EXCEPTSRC,ERRSTATUS

      /Free
```

```
            c = a/b;

            Chain(E) DummyCode Product1;
            If %Error();
(2)            If %Status(Product1) = ERR_NOT_OPEN;
                  Dsply 'The file has not been opened';
(3)            Else;
                  ExSR *PSSR;
               EndIf;
            EndIf;

            Chain DummyCode Product1;
            Open Product1;

            *InLR = *On;

      /Copy EXCEPTSRC,PSSRSTD
      /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR05) SRCFILE(REDBOOK/EXCEPTSRC)
```

The main points to note are:

- A /COPY directive includes the copy member (ERRSTATUS) containing the named constants for the status codes.
- The IF condition for the %ERROR BIF now uses the %STATUS BIF to check if the status of the file indicates a *file not open* condition. You should always specify the file name with the %STATUS BIF to enable you to differentiate between the current statuses of different files. It is a good idea to do this even if you only have one file coded in the program—who knows how many files you will be using tomorrow!
- Otherwise, the program falls back on the trusty *PSSR routine.

Monitor groups

But what about those operation codes that do not allow an E extender? Is it possible to trap exception/errors for them on an individual basis? Yes it is!

V5R1 saw the introduction of a MONITOR group: it allows you to monitor a number of statements for potential errors, as opposed to checking them one at a time. In concept it is quite similar to a SELECT group. It has the following structure

```
Monitor;
    // Code to monitor
On-Error statuscode : statuscode :statuscode;
    // Handle Error
On-Error *FILE;
    // Handle Error
On-Error *PROGRAM;
    // Handle Error
On-Error *ALL;
    // Handle Error
EndMon;
```

A MONITOR group works as follows:

- The boundary is set by a MONITOR and an ENDMON operation.
- The MONITOR operation is followed by the code that you want to monitor for a possible exception/error. This could be a single line of code or an entire program or anything in between. It could also include other MONITOR blocks.
- The code is followed by a set of ON-ERROR operations that indicate the status codes to trap along with the appropriate code to handle the specified error. You may specify individual status codes or the reserved values of *FILE (any file exception/error) or *PROGRAM (any program exception/error) or *ALL (any file or program exception/error).
- If no error occurs in the MONITOR block (i.e., before the first ON-ERROR operation is reached) processing continues at the first instruction following the ENDMON operation code.
- If an error is detected in the block, control immediately passes to the ON-ERROR operations which are processed in sequence—working in the same way as WHEN operations in a SELECT group.
- If a match is found, the code for the ON-ERROR segment is processed and control passes to the instruction following the ENDMON operation. If no matching ON-ERROR condition is found, normal default handling applies (as if the MONITOR group did not exist).
- MONITOR groups may be nested. For some strange reason this seems to be a feature that has been underutilized by many RPG programmers.

Program ERROR06 (Example 8) shows the implementation of a MONITOR group.

Example 8 Sample program ERROR06

```
     FProduct1  UF   E           K Disk    UsrOpn
     F                                     InfSR(*PSSR)

     D PSSRDone        S               N
     D FatalProgram    PR                  ExtPgm('FATALPGM')

     D DummyCode       S                   Like(ProdCd)
     D a               S             10I 0 Inz(20)
     D b               S             10I 0 Inz(10)
     D c               S             10I 0

     D/Copy EXCEPTSRC,ERRSTATUS

      /Free
(1)       Monitor;
             c = a/b;

(2)          Chain DummyCode Product1;

             Chain DummyCode Product1;
             Open Product1;

(3)       On-Error ERR_NOT_OPEN;
             Dsply 'The file has not been opened';

(4)       On-Error ERR_DIVIDE_BY_ZERO:
                   ERR_DECIMAL_DATA;
```

```
              Dsply 'You got a number wrong!';

(5)        On-Error *FILE;
              Dsply 'You are doing something weird to a file';

(6)        On-Error *ALL;
              Dsply 'Who knows what happened?';
              ExSR *PSSR;
(1)        EndMon;

           *InLR = *On;

       /Copy EXCEPTSRC,PSSRSTD
       /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR06) SRCFILE(REDBOOK/EXCEPTSRC)
```

A number of changes have been made to the sample program:

- The code for which you want to trap exception errors is placed in a MONITOR group. If an exception/error is issued for any of the operations (between the MONITOR and the first ON-ERROR operation), control passes to the first ON-ERROR statement.
- The E extender has been removed from the first CHAIN operation since any error on the CHAIN will now be caught by the MONITOR.
- If the exception/error is a “File not open”, a message is displayed and processing continues at the ENDMON operation.
- If the exception/error is a “Divide by zero” or “Decimal data error,” a message is displayed and processing continues at the ENDMON operation.
- If there is any other file exception/error (as opposed to program exception/error), a message is displayed and processing continues at the *ENDMON* operation.
- If there is any other exception/error, a message is displayed and the **PSSR* subroutine is executed.

A MONITOR group also applies to code executed in called subroutines. The MONITOR group in ERROR07 (Example 9) works in the exact same way as the MONITOR group in ERROR06.

Example 9   Sample program ERROR07

```
     FProduct1  UF   E           K Disk    UsrOpn
     F                                     InfSR(*PSSR)

     D PSSRDone        S               N
     D FatalProgram    PR                  ExtPgm('FATALPGM')

     D DummyCode       S                   Like(ProdCd)
     D a               S             10I 0 Inz(20)
     D b               S             10I 0 Inz(10)
     D c               S             10I 0

     D/Copy EXCEPTSRC,ERRSTATUS
     /Free
        Monitor;
           c = a/b;

(1)        ExSR SubMonitor;

        On-Error ERR_NOT_OPEN;
           Dsply 'The file has not been opened';

        On-Error ERR_DIVIDE_BY_ZERO:
                 ERR_ARRAY_INDEX:
                 ERR_DECIMAL_DATA;
           Dsply 'You got a number wrong!';

        On-Error *FILE;
           Dsply 'You are doing something weird to a file';

        On-Error *ALL;
           Dsply 'Who knows what happened?';
           ExSR *PSSR;
        EndMon;

        *InLR = *On;

  (2)   BegSR SubMonitor;
           Chain DummyCode Product1;

           Chain DummyCode Product1;
           Open Product1;
        EndSR;

     /Copy EXCEPTSRC,PSSRSTD
     /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR07) SRCFILE(REDBOOK/EXCEPTSRC)
```

The file processing has been placed in the subroutine SubMonitor (2 above). Although the code of the subroutine is not physically in the MONITOR group, the fact that the EXSR is in the MONITOR group (1 above) means that all of the code in the subroutine is monitored. This would also apply to any subroutines that were executed from the called subroutine, as in Example 10.

Example 10 Sample program ERROR08

```
     FProduct1  UF   E           K Disk    UsrOpn
     F                                     InfSR(*PSSR)

     D PSSRDone        S               N
     D FatalProgram    PR                  ExtPgm('FATALPGM')

     D DummyCode       S                   Like(ProdCd)
     D a               S             10I 0 Inz(20)
     D b               S             10I 0 Inz(10)
```

```
D c               S             10I 0

D/Copy EXCEPTSRC,ERRSTATUS

 /Free
     Monitor;

        ExSR SubMonitor1;

     On-Error ERR_NOT_OPEN;
        Dsply 'The file has not been opened';

     On-Error ERR_DIVIDE_BY_ZERO:
              ERR_ARRAY_INDEX:
              ERR_DECIMAL_DATA;
        Dsply 'You got a number wrong!';

     On-Error *FILE;
        Dsply 'You are doing something weird to a file';

     On-Error *ALL;
        Dsply 'Who knows what happened?';
        ExSR *PSSR;
     EndMon;

     *InLR = *On;

     BegSR SubMonitor1;
        c = a/b;
        ExSR SubMonitor2;
     EndSR;

     BegSR SubMonitor2;
        Chain DummyCode Product1;
        Chain DummyCode Product1;
        Open Product1;
     EndSR;

 /Copy EXCEPTSRC,PSSRSTD
 /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR08) SRCFILE(REDBOOK/EXCEPTSRC)
```

Although a MONITOR group may trap exception/errors from a subroutine that executed from within the group, the same cannot be said for a subprocedure called from within the group. A call to a subprocedure is akin to a call to a program in that it results in a new entry in the call stack. If the code in a subprocedure fails, you can trap the error on the call to the subprocedure by checking for a status of 00202 “Called program or procedure failed.” You will see how exception/error handling works with subprocedures in “Subprocedures and exception/errors” on page 17.

Information data structures

If you need more information about the status of a program or any of the files in the program, you can make use of special data structures. The Program Status Data Structure (PSDS) provides information about the program's status and File Information Data Structures (INFDS) provide information about the status of files. The information in these data structures is maintained by the RPG runtime routines as the program is running.

The PSDS is identified by an SDS definition (7) and there can only be one per program. An INFDS (1) is associated with a specific file by using the INFDS keyword on the F Spec. The INFDS must be unique for a file—you cannot share a file information data structure between two files. These data structures contain an immense amount of information about the program and files and not just information relating to exception/errors. Again, refer to the ILE RPG Reference manual (RPG IV Concept>File and Program Exception/Errors; start at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp)

for a full list of the contents of the data structures.

Example 11 shows an example of a program status and a file information data structure.

Example 11 Sample of program status and file information data structure

```
     FScreens   CF   E             WorkStn
(1)  F                                     InfDS(WorkStnInfDS)
     F                                     InfSR(*PSSR)
     FDatabase  UF A E           K Disk
(1)  F                                     InfDS(DBFileInfDS)
     F                                     InfSR(*PSSR)

(2)  D WorkStnInfDS    DS                  NoOpt
     D                                     Qualified
(3)  D  Status           *Status
(4)  D  MsgId                         7    Overlay(WorkStnInfDS:40)
(5)  D  CursorRow                     3I 0 Overlay(WorkStnInfDS:370)
(5)  D  CursorCol                     3I 0 Overlay(WorkStnInfDS:371)
(5)  D  Min_RRN                       5I 0 Overlay(WorkStnInfDS:378)

(2)  D DBFileInfDS     DS                  NoOpt
     D                                     Qualified
     D  Status           *Status
(6)  D  LockedRecords                 5I 0 Overlay(DBFileInfDS:377)
(6)  D  RRN                           5I 0 Overlay(DBFileInfDS:397)

(7)  D ProgramStatus  SDS                  NoOpt
     D                                     Qualified
(8)  D  ProcedureName    *Proc
(4)  D  MsgId                         7    Overlay(ProgramStatus:46)
```

These are the main points to note:

- The program is an ILE program since it contains subprocedures; therefore, it must be created with DFTACTGRP(*NO).
- The information data structure should normally be defined using the NOOPT keyword. This is to ensure that, when a program is optimized, the latest values are applied in the data structure. The optimizer may keep some values in registers and only restore them to storage at predefined points during normal program execution. The use of NOOPT

ensures the values are always current. Although the use of NOOPT probably makes no difference for programs or modules that are compiled using default optimization, it is advisable to always specify it for file or program information data structures in case the program or module is ever recreated for a higher optimization level. For further information on optimization, look at the OPTIMIZE parameter on the CRTBNDRPG and CRTRPGMOD commands. Note that the INFDS is qualified to ensure that there are not any name conflicts with subfields in the data structure.

- Some of the definitions in the data structures are no longer required since they have been replaced by BIFs; the status code for a file or program can be determined using the %STATUS BIF.
- The MSGID fields identify the relevant CPF or MCH error message received by the program. Note the use of the OVERLAY keyword to avoid the need to use From/To positioning on the D-spec.
- Parts of the file information data structures will be different depending on the type of the file. For a display file, the CURSORROW field identifies the row on the screen at which the cursor was placed when the screen was input. The CURSORCOL field identifies the column on the screen at which the cursor was placed when the screen was input. The MIN_RRN field identifies the RRN of the subfile record at the top of the screen when it was input (which is a lot more dependable than using the SFLCSRRRN keyword in DDS).
- For a database file, the LOCKRECORDS field identifies the number of records currently locked. The RRN field identifies the relative record number of the current record.
- SDS identifies the data structure as being a program status data structure.
- PROCEDURENAME identifies the name of the program. Keywords may be used in place of the length and overlay positions for certain information, such as *PROC for the procedure name.

Program and file information data structures are other items that lend themselves to standard definitions in a copy member. Your first inclination may be to include the complete definition of the data structures, but you should resist the temptation. The standard definitions should only contain the minimum information you require; certain information in the data structures takes time to obtain so you should not define it unless you might use it. For more information on how to use qualified data structures to accomplish this, see:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/books_web/c0925086376.htm#HDRDQUALIF

ILE condition handlers

ILE condition handlers were described in detail in the previous IBM Redbooks publication, *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402 (4.2.7 Call stack and error handling), so we are not going to repeat the process here. However, "ILE CEE APIs" on page 38 does provide a simple example of an ILE condition handler being used, along with some of the other ILE APIs.

Subprocedures and exception/errors

Perhaps you have managed to get exception/error handling working in all of your programs and you start to experiment with subprocedures. Suddenly, exception/error handling doesn't seem to work in the same way.

You must remember that subprocedures are a feature of ILE. They are a form of “sub-program” and they result in their own entry in the call stack.

The other main consideration is that there is a major difference in the way exception/error handling works in the ILE runtime environment as opposed to OPM. In an ILE environment, exception/errors are “percolated” up the call stack.

Percolating messages mean that it is even easier to implement comprehensive exception/error handling in ILE than it is in OPM.

Percolation

An overview of percolation is provided in the previous IBM Redbooks publication (section 4.2.7.1 Call stack and error handling and section 4.2.7.2 Error handling in ILE) but here we give a brief explanation from the point of view of handling exception/errors in an application.

In an OPM environment, a program that is well down the call stack receives an exception/error. If the program does not have any exception/error handling, the RPG default handler will issue a function check message.

In an ILE environment, a subprocedure or a program that is well down the call stack receives an exception/error. If the subprocedure or program does not have any exception/error handling specified, the message is sent back up the call stack to the calling procedure. If the calling procedure does not have any exception/error handling defined, the message is again sent back up the call stack to the calling procedure and so on until the AG control boundary is reached.

It is important to note that messages are percolated to an AG control boundary that may or may not be the program that originally initiated the AG. Refer to the section on Control Boundaries Chapter 3 (ILE Advanced Concepts) of the in ILE Concepts manual.

If none of the subprocedures or programs in the call stack have exception/error handling defined, the message is re-signaled as an exception and control returns to the subprocedure or program that originally received the message and an attempt is made to call the RPG default handler. But subprocedures do not have a default RPG handler, which means that the call to the subprocedure causes an exception/error which, in turn, is percolated up the call stack. This process continues until a default handler can be called. This means that a function check may not be issued until an entry in the call stack is a mainline program (which will have a default error handler).

Let’s look at three examples of how exception/errors are percolated when none of the subprocedures or programs are using exception/error handlers (*PSSR, E extender or Monitor Group).

Example 1

ERROR10 (Example 12) is another version of the sample program used throughout this paper.

Example 12   Sample program ERROR10

```
(1)  H DftActGrp(*No) ActGrp('TEST1') Option(*SrcStmt:*NoDebugIO)

     FProduct1  UF   E           K Disk    UsrOpn

     D ProgramProc     PR
     D FileProc        PR
```

```
      /Free
(2)       ProgramProc();
          FileProc();
          *InLR = *On;
      /End-Free

     P ProgramProc     B
     D ProgramProc     PI

     D a               S             10I 0 Inz(20)
     D b               S             10I 0
     D c               S             10I 0
      /Free
(3)       c = a/b;
      /End-Free
     P                 E

     P FileProc        B
     D FileProc        PI

     D DummyCode       S                   Like(ProdCd)
      /Free
          Chain DummyCode Product1;
          Open Product1;
      /End-Free
     P                 E
```

Create this program with the command:

```
   CRTBNDRPG PGM(REDBOOK/ERROR10) SRCFILE(REDBOOK/EXCEPTSRC)
```

The important points to note are:

- The program is an ILE program since it contains subprocedures; therefore, it cannot be created with DFTACTGRP(*YES).
- All exception/error handling has been removed and the processing logic has been placed in two subprocedures, ProgramProc and FileProc.
- The divide by zero error has been reintroduced.

When you run this program you might expect it to fail with the `RNQ0102 Attempt to divide by zero` message at (3) above, but it doesn't! Instead, the program fails with a `RNQ0202 The call to PROGRAMPRO ended in error (C G D F)` message at (2) above. You see the following information in the joblog:

```
Attempt made to divide by zero for fixed point operation.
Function check. MCH1211 unmonitored by ERROR10 at statement 0000002100,
  instruction X'0000'.
The call to PROGRAMPRO ended in error (C G D F).
```

The "divide by zero" message was percolated up the call stack. Since there is no exception/error handler and the ProgramProc subprocedure does not have a default RPG exception/error handler, the call to ProgramProc ends in error. This again causes an exception/error which results in a function check since the call was issued from the mainline.

Example 2

ERROR11 (Example 13) is similar to ERROR10. The only difference is that the subprocedure ProgramProc is called from the subprocedure Nest2, which is called from the subprocedure Nest1.

Example 13 Sample program ERROR11

```
     H DftActGrp(*No) ActGrp('TEST2') Option(*SrcStmt:*NoDebugIO)

     FProduct1  UF   E           K Disk    UsrOpn

     D Nest1           PR
     D Nest2           PR
     D ProgramProc     PR
     D FileProc        PR

      /Free
(1)       Nest1();
          FileProc();
          *InLR = *On;
      /End-Free

     P Nest1           B
     D Nest1           PI
      /Free
          Nest2();
      /End-Free
     P                 E

     P Nest2           B
     D Nest2           PI
      /Free
          ProgramProc();
      /End-Free
     P                 E

     P ProgramProc     B
     D ProgramProc     PI

     D a               S             10I 0 Inz(20)
     D b               S             10I 0
     D c               S             10I 0
      /Free
          c = a/b;
      /End-Free
     P                 E

     P FileProc        B
     D FileProc        PI

     D DummyCode       S                   Like(ProdCd)
      /Free
          Chain DummyCode Product1;
          Open Product1;
      /End-Free
     P                  E
```

Create this program with the command:

```
   CRTBNDRPG PGM(REDBOOK/ERROR11) SRCFILE(REDBOOK/EXCEPTSRC)
```

This time the program fails with a `RNQ0202 The call to NEST1 ended in error (C G D F)` message at (1) above. You see the following information in the joblog:

```
Attempt made to divide by zero for fixed point operation.
Function check. MCH1211 unmonitored by ERROR11 at statement 0000003700,
  instruction X'0000'.
The call to NEST1 ended in error (C G D F).
```

The process here is exactly the same as it was in ERROR10. All that has been introduced is more levels in the call stack. Even though there are more levels involved, the end result is the same and the function check is issued for the mainline call to NEST1.

Example 3

The third example splits the sample program into two programs. ERROR15 (Example 14) starts an activation group and makes a dynamic call.

Example 14 Sample program ERROR15

```
(1)  H DftActGrp(*No) ActGrp('TEST3') Option(*SrcStmt:*NoDebugIO)

(2)  D Nest1          PR                  ExtPgm('ERROR16')

      /Free
(3)        Nest1();
          *InLR = *On;
      /End-Free
```

Create this program with the command:

```
   CRTBNDRPG PGM(REDBOOK/ERROR15) SRCFILE(REDBOOK/EXCEPTSRC)
```

The main points to note are:

- ► The program starts an activation group named TEST3.
- ► The Nest1 prototype identifies a dynamic call to the program ERROR16. It is important to note that this is a dynamic call to another program as opposed to a bound call to a subprocedure.
- ► The program issues a call to Nest1. If any operation further down the call stack should fail and is not handled, then the exception/error message will arrive at this program.

This is the ERROR16 program (Example 15), which is called from ERROR15.

Example 15 Sample program ERROR16

```
(1)  H DftActGrp(*No) ActGrp(*Caller) Option(*SrcStmt:*NoDebugIO)

     FProduct1  UF   E           K Disk    UsrOpn
     D Nest1           PR                    ExtPgm('ERROR16')
     D Nest2           PR
     D ProgramProc     PR
     D FileProc        PR

     D Nest1           PI

      /Free
(2)       Nest2();
          FileProc();
          *InLR = *On;
      /End-Free


     P Nest2           B
     D Nest2           PI
      /Free
          ProgramProc();
      /End-Free
     P                 E

     P ProgramProc     B
     D ProgramProc     PI

     D a               S             10I 0 Inz(20)
     D b               S             10I 0
     D c               S             10I 0
      /Free
          c = a/b;
      /End-Free
     P                 E

     P FileProc        B
     D FileProc        PI

     D DummyCode       S                    Like(ProdCd)
      /Free
          Chain DummyCode Product1;
          Open Product1;
      /End-Free
     P                 E
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR16) SRCFILE(REDBOOK/EXCEPTSRC)
```

The main points to note are:

- The program is created with an activation group of *CALLER, that is, this program runs in the same activation group (TEST3 in this example) as the procedure that called it.
- The program follows the same broad logic as before. The mainline issues procedure calls to Nest2 and FileProc. Nest2 calls ProgramProc.

When the first program is called, the second program fails with a `RNQ0202 The call to NEST2 ended in error (C G D F)` message at (2) above. You see the following information in the joblog:

```
Attempt made to divide by zero for fixed point operation.
Function check. MCH1211 unmonitored by ERROR16 at statement 0000003300,
  instruction X'0000'.
The call to NEST2 ended in error (C G D F).
```

The failure point is in the second program since there is a default RPG exception/error handler that handles the error in the mainline.

Trapping percolated errors

When you start to use subprocedures, the effects of percolation on exception/errors can be a little disconcerting because the apparent point of failure is higher up the call stack than the actual point where the exception/error occurred—especially when the subprocedure that actually failed is in another module and/or service program.

Don't be disconcerted. Handling exception/errors in an ILE environment is easier than in an OPM environment, with a couple of minor differences.

Remember that you want your default exception/error handler to provide an orderly means of aborting a process without terrifying a user with messages on the screen. Because of percolation, the only programs or procedures that require an exception/error handler are those that mark the control boundary of an AG.

In Example 14 on page 21, simply adding the standard *PSSR subroutine to the mainline of the first program means that the *PSSR subroutine will handle any unhandled exception/error received by any program or subprocedure further down the call stack.
ERROR17(Example 16) is the same as ERROR15 but with the standard *PSSR subroutine added.

Example 16 Sample program ERROR17

```
H DftActGrp(*No) ActGrp('TEST4') Option(*SrcStmt:*NoDebugIO)

D Nest1           PR                  ExtPgm('ERROR16')

D PSSRDone        S              N
D FatalProgram    PR                  ExtPgm('FATALPGM')

 /Free
      Nest1();
     *InLR = *On;
 /Copy EXCEPTSRC,PSSRSTD
 /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR17) SRCFILE(REDBOOK/EXCEPTSRC)
```

The divide by zero error generates the following messages in the joblog:

```
Attempt made to divide by zero for fixed point operation.
RPG status 00202 caused procedure ERROR17 in program REDBOOK/ERROR17 to
  stop.

The file error generates the following messages in the joblog:

I/O operation was applied to closed file PRODUCT1.
RPG status 00202 caused procedure ERROR17 in program REDBOOK/ERROR17 to
  stop.
```

The advantages of using this method are:

- The programs or procedures that mark the control boundary for an activation group are the only programs or procedures that require traditional exception/error handling and the inclusion of a *PSSR subroutine.
- You do not need to specify the *INFSR keyword on F specs. Actually, you cannot compile a module that specifies an INFSR for a file that is referenced in a subprocedure; the compiler issues the message:

  ```
  RNF5416 The subprocedure calculation specification refers to a file
  that has the INFSR keyword specified
  ```

The disadvantages of using this method are:

- Since all programs and subprocedures now share a common *PSSR, you no longer get a dump listing of the subprocedure or program that failed. But remember that programs and procedures further down the call stack may still have their own exception/error handling. So, in the case where you feel a dump listing might be required, code the appropriate routines where they are needed.
- You must be extremely careful that an exception/error that you expect to percolate back to the starting program is not inadvertently trapped by a program or subprocedure further down the call stack. This is especially true with calls to subprocedures in a monitor group or CALL operations with an E extender. An example of inadvertently trapping exception/errors is shown in “The problem with Throw and Catch” on page 56.
- You need to identify which program or subprocedure originally received the exception/error. For example, if a subprocedure 20 levels down in the call stack fails with a *divide by zero,* how does the first entry in the call stack identify the original message, line number, program (or service program) and procedure that received the message? Let’s see how to do this.

Identify a percolated message

The message details for any message in the joblog show details of the program, module, procedure, and line number that received the message. For example, the message details for the RNX1211 message show the following:

```
To program . . . . . . . . . . :   ERROR16
  To library . . . . . . . . . :     REDBOOK
  To module  . . . . . . . . . :     ERROR16
  To procedure . . . . . . . . :     FILEPROC
  To statement . . . . . . . . :     4200
```

But when it comes to identifying exception/errors that caused a program or subprocedure to fail, you should not depend on messages in the joblog. The amount of detail shown in a joblog is dependent on the logging level for the job; see the help for the LOG parameter on the Change Job (CHGJOB) command for details. For example, no exception/errors are shown in the joblog for a job with a logging level of LOG(1 00 *SECLVL).

A joblog contains a filtered view of the messages that were sent to program or procedure message queues in the job. This means that you can write a subprocedure that will retrieve the percolated message from the program/procedure message queue of the program that starts the activation group.

We are going to take the opportunity to write a slightly more complex exception/error reporting program to work in conjunction with the CL FATALPGM written earlier. The RPGLE program FATALPGMA uses the Receive Program Message (QMHRCVPM) API to retrieve the percolated message and some of the Dynamic Screen Management APIs to capture an image of the current screen displayed. A report containing details of the error message and an image of the screen is produced.

The member STDMSGINFO contains the following standard prototype and data structure definitions shown in Example 17:

Example 17 Member STDMSGINFO

```
        // Standard API Error data structure used with most APIs
(1)  D APIError        DS                  Qualified
     D  BytesProvided              10I 0 inz(%size(APIError))
     D  BytesAvail                 10I 0 inz(0)
     D  MsgId                       7A
     D                              1A
     D  MsgData                   240A

       //-------------------------------------------------------------
       // Message APIs
       //-------------------------------------------------------------

       // Receieve Message from Program Message Queue
(2)  D ReceiveMsg      PR                  ExtPgm('QMHRCVPM')
(3)  D  MsgInfo                  3000    Options(*VarSize)
     D  MsgInfoLen                 10I 0 Const
(3)  D  FormatName                  8    Const
     D  CallStack                  10    Const
     D  CallStackCtr               10I 0 Const
     D  MsgType                    10    Const
     D  MsgKey                      4    Const
     D  WaitTime                   10I 0 Const
     D  MsgAction                  10    Const
     D  ErrorForAPI                      Like(APIError)

       //-------------------------------------------------------------
(4)    // Dynamic Screen Manager APIs
       //-------------------------------------------------------------

       // Create Input Buffer
     D CreateInputBuffer...
     D                 PR            10I 0 ExtProc( 'QsnCrtInpBuf' )
     D  BufferSize                 10I 0 Const
```

```
      D  Increment                    10I 0 Const  Options(*Omit)
      D  MaximumSize                  10I 0 Const  Options(*Omit)
      D  BufferHandle                 10I 0 Options(*Omit)
      D  Error                              Like(APIError) Options(*OMIT)

        // Read Screen
      D ReadScreen      PR            10I 0 ExtProc( 'QsnReadScr' )
      D  BytesRead                    10I 0 Options( *Omit )
      D  BufferHandle                 10I 0 Const  Options( *Omit )
      D  CmdBufferhandle...
      D                               10I 0 Const  Options( *Omit )
      D  EnvironmentHandle...
      D                               10I 0 Options( *Omit )
      D  Error                              Like(APIError) Options(*OMIT)

        // Retrieve pointer to data in input buffer
      D RetrieveDataPtr...
      D                 PR              *   ExtProc( 'QsnRtvDta' )
      D  BufferHandle                 10I 0 Const
      D  DataPointer                    *   Options( *Omit )
      D  Error                              Like(APIError) Options(*OMIT)


        //-----------------------------------------------------------------
        // Base Formats
        //-----------------------------------------------------------------

 (5)  D DummyPtr        S               *
         // DS returned by QMHRCVPM for format RCVM0300
 (6)  D RCVM0300        DS                  Qualified Based(DummyPtr)
      D  ByteReturned                 10I 0
      D  ByteAvail                    10I 0
      D  MsgSeverity                  10I 0
 (7)  D  MsgId                         7A
      D  MsgType                       2A
      D  MsgKey                        4A
      D  MsgFileName                  10A
      D  MsgLibSpec                   10A
      D  MsgLibUsed                   10A
      D  AlertOption                   9A
      D  CCSIDCnvIndText...
      D                               10I 0
      D  CCSIDCnvIndData...
      D                               10I 0
      D  CCSIDMsg                     10I 0
      D  CCSIDReplace                 10I 0
 (8)  D  LenReplace1                  10I 0
      D  LenReplace2                  10I 0
 (8)  D  LenMsgReturn                 10I 0
      D  LenMsgAvail                  10I 0
 (8)  D  LenHelpReturn                10I 0
      D  LenHelpAvail                 10I 0
      D  LenSenderReturn...
      D                               10I 0
      D  LenSenderAvail...
```

```
     D                              10I 0
(9)  D  MsgData                   5000A

       // Sender structure returned in RCVM0300
(10) D RCVM0300SndRcvInfo...
     D                DS                 Qualified Based(DummyPtr)
     D  SendingJob                  10A
     D  SendIngJobProfile...
     D                              10A
     D  SendingJobNo                 6A
     D  DateSent                     7A
     D  TimeSent                     6A
     D  SendingType                  1A
     D  ReceivingType                1A
     D  SendingPgm                  12A
     D  SendingModule               10A
     D  SendingProcedure...
     D                             256A
     D                               1A
     D  NoStateNosSending...
     D                              10I 0
     D  StateNosSending...
     D                              30A
(11) D  ReceivingPgm                10A
(11) D  ReceivingModule...
     D                              10A
(11) D  ReceivingProcedure...
     D                             256A
     D                              10A
     D  NoStateNosReceiving...
     D                              10I 0
(11) D  StateNosReceiving...
     D                              30A
     D                               2A
     D  LongSendingPgmNameOffset...
     D                              10I 0
     D  LongSendingPgmNameLength...
     D                              10I 0
     D  LongSendingProcNameOffset...
     D                              10I 0
     D  LongSendingProcNameLength...
     D                              10I 0
     D  LongReceivingProcNameOffset...
     D                              10I 0
     D  LongReceivingProcNameLength...
     D                              10I 0
     D  MicroSeconds                 6A
     D  SendingUsrPrf               10A
     D  Names                     4000A
```

The main items of interest in the copy member are:

- The standard API error structure is described in detail in Chapter 10 of *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402.

- ReceiveMsg is the prototype for the Receive Program Message API (QMHRCVPM).
- Message information is returned in the MsgInfo parameter in the format specified in the FormatName parameter. The required format of MsgInfo is defined by the RCVM0300 data structure (see 6 to 9 below).
- Prototypes are defined for the Dynamic Screen Management (DSM) APIs used to retrieve an image of the current screen: Create Input Buffer (QsnCrtInpBuf), Read Screen (QsnReadScr) and Retrieve Data Pointer (QsnRtvDta).
- Definitions of the structures required by the QMHRCVPM API are based on a pointer that is never set. This means that, although included in a program, the data structures never occupy any memory. Programs that require these formats can define corresponding data structures using the LIKEDS keyword.
- The RCVM0300 data structure defines the format of the data returned by the QMHRCVPM API for the RCVM0300 format name.
- The MsgId field is the message ID of the received message.
- The sum of LenReplace1, LenMsgReturn and LenHelpReturn provides the offset to the position of the sender information in the MsgData parameter.
- The MsgData field contains details of message data (the number of characters defined by LenReplace1) followed by the first level message text (the number of characters defined by LenMsgReturn) followed by the second-level message text (the number of characters defined by LenHelpReturn) followed by the sender/receiver data (the number of characters defined by LenSenderReturn). The sender/receiver data is further defined as a structure (next item).
- The RCVM0300SenderReceiverInfo data structure defines the structure of the sender/receiver information returned in a portion of the MsgData field in the RCVM0300 data structure.
- These fields identify the program (ReceivingPgm), module (ReceivingModule), procedure (ReceivingProcedure), and statement number (StateNosReceiving) that originally received the exception/error message. Interestingly enough, the sender/receiver information does not identify the library for the program.

The FatalError (program FATALPGMA) produces a report detailing the exception/error details. This is the DDS for the ERRORLST print file, shown in Example 18.

Example 18 DDS for the ERRORLST print file

```
A* ERRORLST - Exception/Error Report
A          R HEAD
A                                      SKIPB(03)
A                                     4'Page: '
A                                    +0PAGNBR EDTCDE(Z)
A                                    33'Exception/Error Report'
A                                      SPACEA(2)
A                                     4'Date: '
A                                    +0DATE
A                                      EDTCDE(Y)
A                                    21'Time: '
A                                    +0TIME
A                                    37'Job: '
A            JOBNO          6    0   +0
A            USER          10    0   +1
A            NAME          10    0   +1
A                                      SPACEA(2)
```

```
A                                    4'An unexpected exception/error was -
A                                     detected in a program'
A                                     SPACEA(1)
A                                    4'The message details are:'

A          R DETAIL
A                                     SPACEB(2)
A                                    4'Program:   '
A            PROGRAM       10   0   +0SPACEA(1)
A                                    4'Module     '
A            MODULE        10   0   +0SPACEA(1)
A                                    4'Procedure: '
A            PROCEDURE     60   0   +0SPACEA(2)
A                                    4'Statement: '
A            STATEMENT     10   0   +0SPACEA(1)
A            MSGTEXT       80   0    1SPACEA(2)
A            HLPTEXT01     80   0    1SPACEA(1)
A            HLPTEXT02     80   0    1SPACEA(1)
A            HLPTEXT03     80   0    1SPACEA(1)
A            HLPTEXT04     80   0    1SPACEA(1)
A            HLPTEXT05     80   0    1SPACEA(1)
A            HLPTEXT06     80   0    1SPACEA(1)
A            HLPTEXT07     80   0    1SPACEA(1)
A            HLPTEXT08     80   0    1SPACEA(1)
A            HLPTEXT09     80   0    1SPACEA(1)
A            HLPTEXT10     80   0    1SPACEA(1)

A          R SCREEN80                 SPACEB(2)
A                                    1'----------'
A                                  +10'----------'
A                                  +10'----------'
A                                  +10'----------'
A                                     SPACEA(1)
A            ROW8001       80        1SPACEA(1)
A            ROW8002       80        1SPACEA(1)
A            ROW8003       80        1SPACEA(1)
A            ROW8004       80        1SPACEA(1)
A            ROW8005       80        1SPACEA(1)
A            ROW8006       80        1SPACEA(1)
A            ROW8007       80        1SPACEA(1)
A            ROW8008       80        1SPACEA(1)
A            ROW8009       80        1SPACEA(1)
A            ROW8010       80        1SPACEA(1)
A            ROW8011       80        1SPACEA(1)
A            ROW8012       80        1SPACEA(1)
A            ROW8013       80        1SPACEA(1)
A            ROW8014       80        1SPACEA(1)
A            ROW8015       80        1SPACEA(1)
A            ROW8016       80        1SPACEA(1)
A            ROW8017       80        1SPACEA(1)
A            ROW8018       80        1SPACEA(1)
A            ROW8019       80        1SPACEA(1)
A            ROW8020       80        1SPACEA(1)
A            ROW8021       80        1SPACEA(1)
A            ROW8022       80        1SPACEA(1)
```

```
     A            ROW8023       80         1SPACEA(1)
     A            ROW8024       80         1SPACEA(1)
     A                                     1'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'

     A          R SCREEN132                 SPACEB(2)
     A                                     1'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                      SPACEA(1)
     A            ROW13201     132         1SPACEA(1)
     A            ROW13202     132         1SPACEA(1)
     A            ROW13203     132         1SPACEA(1)
     A            ROW13204     132         1SPACEA(1)
     A            ROW13205     132         1SPACEA(1)
     A            ROW13206     132         1SPACEA(1)
     A            ROW13207     132         1SPACEA(1)
     A            ROW13208     132         1SPACEA(1)
     A            ROW13209     132         1SPACEA(1)
     A            ROW13210     132         1SPACEA(1)
     A            ROW13211     132         1SPACEA(1)
     A            ROW13212     132         1SPACEA(1)
     A            ROW13213     132         1SPACEA(1)
     A            ROW13214     132         1SPACEA(1)
     A            ROW13215     132         1SPACEA(1)
     A            ROW13216     132         1SPACEA(1)
     A            ROW13217     132         1SPACEA(1)
     A            ROW13218     132         1SPACEA(1)
     A            ROW13219     132         1SPACEA(1)
     A            ROW13220     132         1SPACEA(1)
     A            ROW13221     132         1SPACEA(1)
     A            ROW13222     132         1SPACEA(1)
     A            ROW13223     132         1SPACEA(1)
     A            ROW13224     132         1SPACEA(1)
     A            ROW13225     132         1SPACEA(1)
     A            ROW13226     132         1SPACEA(1)
     A            ROW13227     132         1SPACEA(1)
     A                                     1'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                   +10'----------'
     A                                      SPACEA(1)

     A          R FOOTER
     A                                       SPACEB(3)
     A                                      4'*** Error details complete ***'
```

Create this print file with the command:

```
CRTPRTF FILE(REDBOOK/ERRORLST) SRCFILE(REDBOOK/EXCEPTSRC)
```

The exception/error report simply lists the message details along with details of the program/procedure that received the message and, if the program is running interactively, a copy of the screen.

These are the H, F, and D specs of the FATALPGMA program; see Example 19.

Example 19   Sample program FATALPGMA

```
(1)  H DftActGrp(*No) ActGrp(*Caller) Option(*SrcStmt:*NoDebugIO)

(2)  FERRORLST  O    E             Printer ExtFile('REDBOOK/ERRORLST')
     F                                     USROPN
     F                                     OFlInd(OverFlow)

(3)   /Copy EXCEPTSRC,STDMSGINFO

     D FatalError      PR                  ExtPgm('REDBOOK/FATALPGMA')
(4)  D FatalProgram    PR                  ExtPgm('REDBOOK/FATALPGM')

     D FatalError      PI

(5)  D HlpText         DS
     D  HlpText01
     D  HlpText02
     D  HlpText03
     D  HlpText04
     D  HlpText05
     D  HlpText06
     D  HlpText07
     D  HlpText08
     D  HlpText09
     D  HlpText10

     D ScreenIn        DS
     D  Row8001                            OverLay(ScreenIn:1)
     D  Row8002                            OverLay(ScreenIn:*Next)
     D  Row8003                            OverLay(ScreenIn:*Next)
     D  Row8004                            OverLay(ScreenIn:*Next)
     D  Row8005                            OverLay(ScreenIn:*Next)
     D  Row8006                            OverLay(ScreenIn:*Next)
     D  Row8007                            OverLay(ScreenIn:*Next)
     D  Row8008                            OverLay(ScreenIn:*Next)
     D  Row8009                            OverLay(ScreenIn:*Next)
     D  Row8010                            OverLay(ScreenIn:*Next)
     D  Row8011                            OverLay(ScreenIn:*Next)
     D  Row8012                            OverLay(ScreenIn:*Next)
     D  Row8013                            OverLay(ScreenIn:*Next)
     D  Row8014                            OverLay(ScreenIn:*Next)
     D  Row8015                            OverLay(ScreenIn:*Next)
     D  Row8016                            OverLay(ScreenIn:*Next)
     D  Row8017                            OverLay(ScreenIn:*Next)
     D  Row8018                            OverLay(ScreenIn:*Next)
     D  Row8019                            OverLay(ScreenIn:*Next)
     D  Row8020                            OverLay(ScreenIn:*Next)
     D  Row8021                            OverLay(ScreenIn:*Next)
     D  Row8022                            OverLay(ScreenIn:*Next)
     D  Row8023                            OverLay(ScreenIn:*Next)
     D  Row8024                            OverLay(ScreenIn:*Next)

     D  Row13201                           OverLay(ScreenIn:1)
     D  Row13202                           OverLay(ScreenIn:*Next)
     D  Row13203                           OverLay(ScreenIn:*Next)
     D  Row13204                           OverLay(ScreenIn:*Next)
     D  Row13205                           OverLay(ScreenIn:*Next)
     D  Row13206                           OverLay(ScreenIn:*Next)
     D  Row13207                           OverLay(ScreenIn:*Next)
     D  Row13208                           OverLay(ScreenIn:*Next)
     D  Row13209                           OverLay(ScreenIn:*Next)
     D  Row13210                           OverLay(ScreenIn:*Next)
     D  Row13211                           OverLay(ScreenIn:*Next)
     D  Row13212                           OverLay(ScreenIn:*Next)
     D  Row13213                           OverLay(ScreenIn:*Next)
     D  Row13214                           OverLay(ScreenIn:*Next)
     D  Row13215                           OverLay(ScreenIn:*Next)
     D  Row13216                           OverLay(ScreenIn:*Next)
     D  Row13217                           OverLay(ScreenIn:*Next)
     D  Row13218                           OverLay(ScreenIn:*Next)
     D  Row13219                           OverLay(ScreenIn:*Next)
     D  Row13220                           OverLay(ScreenIn:*Next)
     D  Row13221                           OverLay(ScreenIn:*Next)
     D  Row13222                           OverLay(ScreenIn:*Next)
     D  Row13223                           OverLay(ScreenIn:*Next)
     D  Row13224                           OverLay(ScreenIn:*Next)
     D  Row13225                           OverLay(ScreenIn:*Next)
     D  Row13226                           OverLay(ScreenIn:*Next)
     D  Row13227                           OverLay(ScreenIn:*Next)

(6)  D ProgramStatus  SDS
     D  NAME                               Overlay(ProgramStatus:244)
     D  USER                               Overlay(ProgramStatus:254)
     D  JOBNO                              Overlay(ProgramStatus:264)

(7)  D MsgBack         DS                  LikeDs(RCVM0300) Inz

     D InfoPtr         S               *
(8)  D MsgInfo         DS                  LikeDs(RCVM0300SndRcvInfo)
     D                                     Based(InfoPtr)

     D i               S             10I 0
     D SetMsgKey       S              4    Inz(*ALLx'00')
     D BufferHandle    S             10I 0
     D BytesReturned   S             10I 0
     D DataPtr         S               *
     D CatchScreen     DS                  LikeDS(ScreenIn)
     D                                      Based(DataPtr)
```

The main points of interest are:

- The H spec indicates that the program should run in the activation group of the caller.
- The F spec for the ERRORLST print file uses the EXTFILE keyword to ensure that the print file can be found. In this case it is better not to depend on the library list to locate the print file.
- The STDMSGINFO member containing prototypes and standard definitions is included.
- The prototype for the original FATALPGM is included. FATALPGM is called once the error report has been produced.
- Data structures are used to map the print fields (from ERRORLST) to fields retrieved from APIs. The HlpText data structure is used to redefine the retrieved second level message text for printing. The ScreenIn data structure is used to redefine the retrieved screen image (either 24x80 or 27x132). Note that lengths do not need to be defined for the subfields since they are externally defined on the ERRORLST print file.
- A program status data structure is used to identify the job.
- MsgBack is the data structure that will receive the exception/error message. The LIKEDS keyword is used to define it like the standard RCVM0300 data structure in STDMGINFO.
- MsgInfo is the data structure that will contain the sender/receiver information from the message data field in the MsgBack data structure. The LIKEDS keyword is used to define it like the standard RCVM0300SndRcvInfo data structure in STDMGINFO.

The program (Example 20) starts by retrieving and reporting the error.

Example 20 Sample program to retrieve and report error

```
      /Free

(1)        Open ErrorLst;
           Write Head;

(2)        ReceiveMsg( MsgBack
                     : %size(MsgBack)
                     : 'RCVM0300'
                     : '*'
                     : 2
                     : '*PRV'
                     : SetMsgKey
                     : 0
                     : '*SAME'
                     : APIError);

(3)        If MsgBack.ByteAvail > 0;

(4)            MsgText = %SubSt(MsgBack.MsgData:
                                MsgBack.LenReplace1 + 1:
                                MsgBack.LenMsgReturn);
               HlpText = %SubSt(MsgBack.MsgData:
                                MsgBack.LenReplace1 +
                                MsgBack.LenMsgReturn + 1:
                                MsgBack.LenHelpReturn);
(5)            InfoPtr = %Addr(MsgBack.MsgData)
```

```
                   + MsgBack.LenReplace1
                   + MsgBack.LenMsgReturn
                   + MsgBack.LenHelpReturn;
              Program = MsgInfo.ReceivingPgm;
              Module = MsgInfo.ReceivingModule;
              Procedure = MsgInfo.ReceivingProcedure;
              Statement = MsgInfo.StateNosReceiving;

(6)           Write Detail;
              If OverFlow;
                 Write Head;
                 OverFlow = *Off;
              EndIf;

          EndIf;
```

The main points are:

- The FatalError subprocedure starts by opening the ERRORLST print file and printing the headings. This identifies the failing job.
- The QMHRCVPM API is called to retrieve the last message in the program/procedure message queue of the calling program/procedure. The call stack counter is 2 since the calling procedure/program will be two entries back in the call stack; a dynamic call is made to this program so it will have a PEP in the call stack. The message action is set to *SAME to ensure that the message remains in the job log.
- The ByteAvail field in the Msgback data structure identifies whether or not a message was retrieved from the message queue.
- The first level and second level message texts are retrieved from the MsgData field in the MsgBack data structure using the returned lengths to determine the starting positions and lengths to retrieve.
- The returned lengths are again used to set the basing pointer for the MsgInfo data structure so it is mapped on to the relevant part of the MsgData field in the MsgBack data structure.
- The message details are printed.

The program (Example 21) continues by capturing the current screen image (if it is an interactive job).

Example 21 Sample program to capture current interactive screen image

```
(1)       BufferHandle =  CreateInputBuffer( 27 * 132
                                          : *Omit
                                          : *Omit
                                          : *Omit
                                          : APIError );

(2)       If APIError.BytesAvail = 0;
(3)           BytesReturned = ReadScreen( *Omit
                                       : BufferHandle
                                       : *Omit
                                       : *Omit
                                       : *Omit );

(4)           DataPtr = RetrieveDataPtr( BufferHandle
                                        : *Omit
                                        : *Omit );
(5)           ScreenIn = %SubSt(CatchScreen:1:BytesReturned);
(6)           For i = 1 to BytesReturned;
                 If (%SubSt(ScreenIn:i:1) > x'19') And
                    (%SubSt(ScreenIn:i:1) < x'40');
                     %SubSt(ScreenIn:i:1) = *Blank;
                 EndIf;
              EndFor;
(7)           If BytesReturned = 1920;
                 Write Screen80;
              Else;
                 Write Screen132;
              EndIf;
           EndIf;
```

The main points are:

- The Create Input Buffer API is called to get a buffer handle for the screen.
- Only call the remainder of the DSM APIs if a buffer handle was successfully created.
- Read the screen and determine the number of bytes returned; this will be 1920 for a 24x80 screen or 3564 for a 27x132 screen. This API captures an image of all characters on the screen.
- Get a pointer to the buffer for the screen just read. The CatchScreen data structure is based on this pointer.
- Copy the screen buffer (CatchScreen) to the ScreenIn data structure so the print fields are populated.
- We have replaced any attribute bytes in the captured screen image with blanks. If you wish you can replace any attribute byte with the identifying character you deem appropriate, for example to mark the beginning of entry fields.
- Print either the 24x80 or 27x132 image of the screen, dependent on the number of characters retrieved.

The program finishes by printing the footer, closing the print file, and calling the FatalProgram to notify the error and provide a meaningful message to the user.

```
Write Footer;
           Close ErrorLst;

           FatalProgram();
           *InLR = *On;
      /End-Free
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/FATALPGMA) SRCFILE(REDBOOK/EXCEPTSRC)
```

Making use of the FatalError program means that the *PSSR subroutine must change. Instead of calling the FatalPgm program, the *PSSR subroutine will call the FatalError program. ERROR18 (Example 22) is an altered version of ERROR17 with a new *PSSR coded.

Example 22 Sample program ERROR18

```
     H DftActGrp(*No) ActGrp('TEST5') Option(*SrcStmt:*NoDebugIO)
     D Nest1           PR                  ExtPgm('ERROR16')

(1)  D FatalError      PR                  ExtPgm('REDBOOK/FATALPGMA')

      /Free
           Nest1();
          *InLR = *On;

(2)       BegSR *PSSR;
             Monitor;
                 FatalError();
             On-Error;
                 Dsply 'Agggghhhhhh!';
             EndMon;
          EndSR '*CANCL';
      /End-Free
```

Create this program with the command:

```
   CRTBNDRPG PGM(REDBOOK/ERROR18) SRCFILE(REDBOOK/EXCEPTSRC)
```

The main points to note are:

- The FatalError prototype identifies a dynamic call to the FATALPGMA program. The program name is qualified to remove any dependency on a library list.
- The *PSSR subroutine is a much simpler version of the previous one in that all it does is call FatalError. All code in the *PSSR subroutine is placed in a MONITOR group to ensure there are not unrequested repeated calls to *PSSR.

Any unhandled exception/error detected in any program or procedure in the call stack results in the *PSSR routine above being executed and the subsequent call to FatalError.

Figure 1 on page 37 shows an example of the resulting error exception report produced when the file error is detected in the FileProc procedure in ERROR16.

```
 Page:    1                    Exception/Error Report


   Date: 29/11/06    Time: 11:34:41  Job: 056418 TUOHYP     COMCONPTA

   An unexpected exception/error was detected in a program
   The message details are:


   Program:    ERROR16
   Module      ERROR16
   Procedure:  FILEPROC
   Statement:  4200

I/O operation was applied to closed file PRODUCT1.


Cause . . . . . :   RPG procedure FILEPROC in program REDBOOK/ERROR16 attempted
Operation CHAIN on file PRODUCT1 while the file was closed. Recovery  . . . :
Contact the person responsible for program maintenance to determine the cause
of the problem.



---------          ----------          ----------          ----------
MAIN                           OS/400 Main Menu
                                                            System:   xxxxxxxx
Select one of the following:

     1. User tasks
     2. Office tasks
     3. General system tasks
     4. Files, libraries, and folders
     5. Programming
     6. Communications
     7. Define or change the system
     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. iSeries Access tasks

    90. Sign off

Selection or command
===> call error18

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu

---------          ----------          ----------          ----------

           *** Error details complete ***
```

Figure 1   Error report from sample program ERROR16

You have seen how you can extend the functionality of the program that is called when an unexpected exception/error occurs. This program could be extended even further to incorporate the automatic generation of a log number, recording the incident to a database file, communicating with the operator, or whatever else you deem appropriate.

ILE CEE APIs

Some of the ILE CEE APIs (Information Center>Programming>APIs>APIs by Category>ILE CEE) can be very useful when used in conjunction with exception/error handling; specifically the Condition Management APIs and the Activation Group and Control Flow APIs.

Condition Management APIs

The use of the relevant ILE Condition Management APIs was covered in the previous IBM Redbooks publication (*Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402, section 4.2.7 Call stack and error handling). Refer to it for details. But for the sake of completeness we are including a very simple ILE condition handler to demonstrate how it works in conjunction with exception/error handling.

When a condition handler is registered it applies not just to the program or procedure in which it is registered but to all programs or procedures further down the call stack. Condition handlers may be “nested”, that is, another condition handler may be registered in another program or procedure in the call stack.

One of the most useful features of an ILE condition handler is that it can determine if it will handle the exception/error and whether or not processing should continue, or whether the exception/error should be percolated up the call stack.

The STDMSGINFO copy member (Example 23) contains the prototypes for the Register a Condition Handler (CEEHDLR) and Un-Register a Condition Handler (CEEHDLU) APIs and the base definition of a condition token.

Example 23 Prototypes for CEEHDLR and CEEHDLU

```
  //-----------------------------------------------------------------
  // ILE CEE APIs
  //-----------------------------------------------------------------

  // Register a Condition Handler
D RegisterHandler...
D                 PR                  ExtProc('CEEHDLR')
D  pConHandler                    *   ProcPtr Const
D  CommArea                       *   Const
D  Feedback                     12A   Options(*Omit)

  // Un-Register a Condition Handler
D UnRegisterHandler...
D                 PR                  ExtProc('CEEHDLU')
D  pConHandler                    *   ProcPtr Const
D  Feedback                     12A   Options(*Omit)

  // Condtion Token passed to/from ILE Handler
D Base_ConditionToken...
D                 DS                  BASED(DummyPtr)
```

```
     D                                       Qualified
     D  MsgSev                          5I 0
     D  MsgNo                           2A
     D                                  1A
     D  MsgPrefix                       3A
     D  MsgKey                          4A
```

ERROR21 (Example 24) is an amended version of ERROR18 that includes the registering and un-registering of an ILE condition handler along with the coding of the condition handler itself.

Example 24   Sample program ERROR21

```
 (1)  H DftActGrp(*No) ActGrp('TEST6') Option(*SrcStmt:*NoDebugIO)

      /Copy EXCEPTSRC,STDMSGINFO

     D Nest1           PR                  ExtPgm('ERROR16')

     D FatalError      PR                  ExtPgm('REDBOOK/FATALPGMA')

(2)  D ILEHandler      PR
     D  TokenIn                            LikeDS(Base_ConditionToken)
     D                                     Const
     D  pCommArea                     *    Const
     D  Action                      10I 0
     D  TokenOut                           LikeDS(Base_ConditionToken)

(3)  D pILEHandler     S              *    ProcPtr
     D                                     Inz(%PAddr('ILEHANDLER'))

      /Free
(4)       RegisterHandler(pILEHandler : *Null : *OMIT);

          Nest1();

(5)       UnRegisterHandler(pILEHandler : *OMIT);
          *InLR = *On;

          BegSR *PSSR;
             Monitor;
                 FatalError();
             On-Error;
                 Dsply 'Agggghhhhhh!';
             EndMon;
          EndSR '*CANCL';
      /End-Free


     P ILEHandler      B                   Export
     D ILEHandler      PI
     D  TokenIn                            LikeDS(Base_ConditionToken)
     D                                     Const
     D  pCommArea                     *    Const
     D  Action                      10I 0
     D  TokenOut                           LikeDS(Base_ConditionToken)
     D RESUME          C                   10
     D PERCOLATE       C                   20

      /Free
           Select;
             When TokenIn.MsgPrefix = 'MCH' And
                   TokenIn.MsgNo = X'1211';
(6)            Action    = RESUME;
             Other;
(7)            Action = PERCOLATE;
           EndSl;
           Return;
      /End-Free
     P                 E
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/ERROR21) SRCFILE(REDBOOK/EXCEPTSRC)
```

The main points to note are:

- The inclusion of the bindable ILE CEE APIs does not require any special binding directory.
- ILEHandler is the prototype for the condition handler registered in the program.
- pILEHandler is the procedure pointer to the ILE condition handler that will be registered in the program.
- The ILEHandler subprocedure is registered as an ILE condition handler. It now applies to all further entries on the call stack unless superseded by another handler (E extender, MONITOR group, or another ILE condition handler). In this example a communications area is not used, so the second parameter is null.
- The ILE condition handler is unregistered at the end of the program.
- The ILE condition handler indicates that all *divide by zero* errors should be ignored by resuming. It is not the usual course of action we would recommend since the resulting value is unpredictable, but it is applicable in the simple example you see here.
- All other errors are percolated.

The end result of registering the ILEHandler condition handler is that the divide by zero exception/error in the ProgramProc subprocedure in ERROR16 will be ignored, whereas any other exception/error will result in the “normal” default exception/error process.

Activation Group and Control Flow APIs

The ILE CEE Activation Group and Control Flow APIs provide features that may be useful in controlling the orderly termination of activation groups or the opportunity to run clean-up or termination routines when a program or procedure is canceled (for example, close open files). Some of the more commonly used Activation Group and Control Flow APIs are:-

- CEERTX - Register Call Stack Entry Termination User Exit Procedure
- CEEUTX - Un-Register Call Stack Entry Termination User Exit Procedure
- CEETREC - Normal End (of activation group)
- CEE4AGE - Register Activation Group Exit Procedure

You will see an example that uses each of these APIs, but first here is a brief description of how they work.

CEETREC

CEETREC may be used to end an activation group “normally.” It may be called from any program or procedure within the activation group and the activation group is immediately ended much in the same way as the Reclaim Activation Group (RCLACTGRP) command can end an activation group. The main difference between using CEETREC and RCLACTGRP is that RCLACTGRP cannot reclaim the activation group in which the command is run.

In RPG, CEETREC provides the same functionality as the exit() function in C and the STOP RUN operation in COBOL.

This is the prototype for CEETREC:

```
D EndAG           PR                  ExtProc('CEETREC')
D  Language_RC                  10I 0 Const Options(*Omit)
D  User_RC                      10I 0 Const Options(*Omit)
```

The parameters to CEETREC are usually omitted. Both parameters are optional “return codes” that are input to CEETREC.

Although control is not returned to any of the entries in the call stack, any clean-up procedures registered for call stack entries are called before the activation group ends.

CEERTX and CEEUTX

CEERTX allows you to register a procedure that is called when the call stack entry for which it is registered is ended by anything other than a return to the caller. This might be caused by a call ending abnormally or a request to end the activation group using CEETREC, or RCLACTGRP, or a request to end the job using ENDJOB, or a user canceling the program or procedure using option 2 from the system request menu.

Multiple exit procedures may be registered for a call stack entry. They are called in reverse order of registration (LIFO).

These are the prototypes for CEERTX and CEEUTX:

```
D RegisterCancelStack...
D                 PR                  ExtProc('CEERTX')
D  pCancelProc                    *   ProcPtr Const
D  CommArea                       *   Const Options(*Omit)
D  Feedback                     12A   Options(*Omit)

D UnRegisterCancelStack...
D                 PR                  ExtProc('CEEUTX')
D  pCancelProc                    *   ProcPtr Const
D  Feedback                     12A   Options(*Omit)
```

The parameters are:

- The first parameter for both APIs is a procedure pointer to the procedure being registered or unregistered.
- The second parameter to CEERTX is a pointer to a communications area. Since you have no means of defining your own parameters with this API, the communications area provides a means of passing application-specific data to the procedure.

- Both APIs have an omissible feedback area.

The procedure being registered requires a prototype with the following parameters:-

```
D CancelStack     PR
D  CommArea                      *   Const
```

The single parameter is the pointer to the communications area specified when the procedure was registered.

CEE4AGE

CEE4AGE allows you to register a procedure that is called when an activation group ends. The procedure can perform any type of tidying up that you may deem necessary before the activation group is reclaimed, for example write information to a log file, tidy up work files or user spaces.

Multiple exit point procedures may be registered. They are called in reverse order of registration (LIFO).

This is the prototype for CEE4RAGE:

```
D RegisterCancelAG...
D                 PR                 ExtProc('CEE4RAGE')
D  pCancelProc                   *   ProcPtr Const
D  Feedback                    12A   Options(*Omit)
```

The first parameter is a procedure pointer to the procedure being registered. The second parameter is an omissible feedback area.

The procedure being registered will be passed four parameters by the system and therefore requires a prototype with the following parameters:

```
D AGCancel        PR
D  AGMark                      10I 0 Const
D  Reason                      10I 0 Const
D  ResultCode                  10I 0
D  User_RC                     10I 0
```

The parameters are:

- The first parameter is a marker that uniquely identifies the AG within the job.
- You are primarily interested in the second parameter which identifies why and how the AG is ending; you will see how this is interpreted in the upcoming example.
- The third parameter is a result code that is passed between AG Exit procedures when multiple procedures are registered (a value of 0 is passed to the first exit procedure). Table 3 shows the possible values for the result code.

Table 3 Result code values

Code	Meaning
0	Do not change the action.
10	Do not perform any pending error requests. This is used if a previous exit procedure specified a result code of 20 and a subsequent procedure recovers from the error. The message CEE9901, indicating an application error, is not sent.

20	Send message CEE9901 to the caller of the control boundary after the remaining exit procedures are called.
21	Send message CEE9901 to the caller of the control boundary. The remaining exit procedures registered by the CEE4RAGE API are not called. This is used if an unrecoverable error occurs in the exit procedure requesting this action.

- The fourth parameter is a user result code that is passed between AG Exit procedures; set it as you see fit.

Example

Let's have a look at the Activation Group and Control Flow APIs in action. The member CANCEL01 contains global definitions, a mainline and a subprocedure (Example 25).

Example 25 Global program definitions in member CANCEL01

```
(1)  H DftActGrp(*No) ActGrp('CANCEL') Option(*SrcStmt:*NoDebugIO)

(2)   /Copy EXCEPTSRC,STDMSGINFO
     D SubProc01       PR                  ExtPgm('CANCEL02')

(3)  D AGCancel        PR
     D  AGMark                       10I 0 Const
     D  Reason                       10I 0 Const
     D  ResultCode                   10I 0
     D  User_RC                      10I 0

     D x               S             10I 0

      /Free
(4)       x = x + 1;
          Dsply x;

(5)       If x > 2;
             SubProc01();
          EndIf;
          Return;

          BegSR *InzSR;
(6)          RegisterCancelAG(%pAddr('AGCANCEL'):*Omit);
          EndSr;
      /End-Free
```

The main points to note are:

- The program runs in a named activation group called CANCEL. This means that the AG remains in the job when this program is exited.
- The prototypes for the CEE APIs described above have been placed in the copy member STDMSGINFO.
- AGCancel is the prototype for the procedure that will be registered as an exit procedure for the AG.
- The program simply adds 1 to x and displays the value.
- When x is greater than 2 (this happens when the program has been called three times in succession) a dynamic call is made to the program CANCEL02.

- The subprocedure AGCancel is registered as an exit procedure for the AG. This is performed in the *INZSR subroutine to ensure that the procedure is only registered once; if it were in the mainline, the procedure would be registered on every call and would result in the procedure being called multiple times when the AG ends.

The member CANCEL01 also contains the definition of the AGCancel subprocedure (Example 26). The subprocedure does not have to be placed in the same module that registers it; it could just as easily be placed in a module in a service program.

Example 26 Program subprocedure AGCancel

```
     P AGCancel        B
     D AGCancel        PI
     D  AGMark                       10I 0 Const
     D  Reason                       10I 0 Const
     D  ResultCode                   10I 0
     D  User_RC                      10I 0

(1)  D                 DS
     D  Reason0                      10I 0
     D  Reason1                       3I 0 Overlay(Reason0)
     D  Reason2                       3I 0 Overlay(Reason0:*Next)
     D  Reason3                       3I 0 Overlay(Reason0:*Next)
     D  Reason4                       3I 0 Overlay(Reason0:*Next)

(2)  D END_NORMAL      C                   128
     D END_RCLACTGRP   C                   32
     D END_ENDJOB      C                   16
     D END_CEETREC     C                   8
     D END_FUNCGECK    C                   4
      /Free
          Reason0 = Reason;
(3)       If %BitAnd(Reason3:END_NORMAL) = x'00';
             Dsply 'AG ended normally';
          Else;
             Dsply 'AG ended abnormally';
          EndIf;
          Select;
             When %BitAnd(Reason3:END_RCLACTGRP) <> x'00';
                Dsply 'AG ended by RCLACTGRP';
             When %BitAnd(Reason3:END_ENDJOB ) <> x'00';
                Dsply 'AG ended by Job Ending';
             When %BitAnd(Reason3:END_CEETREC) <> x'00';
                Dsply 'AG ended by exit request (CEETREC)';
             When %BitAnd(Reason3:END_FUNCGECK) <> x'00';
                Dsply 'AG ended unhandled function check';
             Other;
                Dsply 'AG ended';
          EndSl;
      /End-Free
     P                 E
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/CANCEL01) SRCFILE(REDBOOK/EXCEPTSRC)
```

The main points to note are:

- Even though the reason code is passed as an integer, you need to decipher bit settings in the four bytes in order to determine why and how the AG is ending. One way of achieving this is to remap the 4-byte integer to four separate bytes and use the %BITAND BIF to determine whether or not the relevant bits are set. Table 4 is taken from the description of the CEE4RAGE API and provides the Common Reason Codes for Ending Activation Groups and Call Stack Entries, byte 3 (bits 16 to 23) being the most relevant.

Table 4 Common Reason Codes for Ending Activation Groups and Call Stack Entries

Bit	Meaning
Bits 0	Reserved
Bits 1	Call stack entry is canceled because an exception message was sent.
Bits 2-15	Reserved
Bit 16	0 - normal end 1 - abnormal end
Bit 17	Activation Group is ending.
Bit 18	Initiated by the Reclaim Activation Group (RCLACTGRP) command.
Bit 19	Initiated as a result of the job ending.
Bit 20	Initiated by an exit verb, for example exit() in C, or the CEETREC API.
Bit 21	Initiated by an unhandled function check.
Bit 22	Call stack entry canceled because of an out-of-scope jump, for example *longjmp()* in C.
Bits 23-31	Reserved

- Named constants are used to identify the bit settings to be tested, for example a value of 128 indicates that bit 0 of a byte is on, or a value of 32 indicates that bit 2 of a byte is on.
- The %BITAND BIF is used to determine whether or not the relevant bits are set in the third byte of the reason code and to condition the relevant message being displayed.

You are now ready for the first test. Call the program CANCEL01 and the value of x is displayed:

```
DSPLY           1
```

Call CANCEL01 a second time and the value of x is again displayed:

```
DSPLY           2
```

Now issue the command RCLACTGRP CANCEL (do not call CANCEL01 for a third time) and the following is displayed:

```
DSPLY  AG ended normally
DSPLY  AG ended by RCLACTGRP
```

Reclaiming the activation group causes the AGCancel subprocedure to be called.

Now let's add some additional complexity. The member CANCEL02 contains global definitions, a mainline, and a number of subprocedures (Example 27).

Example 27 Global program definitions in member CANCEL02

```
(01) H DftActGrp(*No) ActGrp(*Caller) Option(*SrcStmt:*NoDebugIO)

      /Copy EXCEPTSRC,STDMSGINFO
```

```
     D SubProc02       PR
     D SubProc03       PR
     D SubProc04       PR
     D SubProc05       PR
     D  Action                       10I 0 Const

(2)  D AGCancelNext    PR
     D  AGMark                       10I 0 Const
     D  Reason                       10I 0 Const
     D  ResultCode                   10I 0
     D  User_RC                      10I 0

(3)  D CancelStack     PR
     D  CommArea                       *   Const

      /Free
(4)       SubProc02();
          Return;

          BegSR *InzSR;
(5)          RegisterCancelAG(%pAddr('AGCANCELNEXT'):*Omit);
          EndSr;
      /End-Free
```

The main points to note are:

- The program runs in the activation group of the calling program or procedure. In this instance this program will run in the CANCEL AG.
- AGCancelNext is the prototype for the procedure that will be registered as an exit procedure for the AG. This will be a second exit procedure being registered for the AG.
- CancelStack is the prototype for the procedure that will be registered as the exit procedure for call stack entries. This program uses the same exit procedure for multiple call stack entries (since all it does is display a message), but you can have as many different exit procedures as you need as long as the prototype definition is correct.
- The program simply calls the subprocedure SubProc02.
- The subprocedure AGCancelNext is registered as an exit procedure for the AG. This is the second exit procedure that is registered for the AG (the first being in CANCEL01).

This is the AGCancelNext subprocedure (Example 28). It is a simplified version of the exit procedure in CANCEL01. All it does is display a message.

Example 28   Program subprocedure AGCancelNext

```
     P AGCancelNext    B
     D AGCancelNext    PI
     D  AGMark                       10I 0 Const
     D  Reason                       10I 0 Const
     D  ResultCode                   10I 0
     D  User_RC                      10I 0
      /Free
          Dsply 'AGCancelNext in CANCEL02';
      /End-Free
     P                 E
```

This is the CancelStack subprocedure (Example 29). It simply displays the message based on the pointer passed as the communication area.

Example 29 Program subprocedure CancelStack

```
     P CancelStack     B
     D CancelStack     PI
     D  CommArea                       *   Const

     D Msg             S             20    Based(CommArea)
      /Free
          Dsply Msg;
      /End-Free
     P                 E
```

This is the SubProc02 subprocedure which is called from the mainline of CANCEL02 (Example 30). It registers CancelStack as an exit procedure with a pointer to the relevant message as the second parameter. It then issues a call to SubProc03.

Example 30 Program subprocedure SubProc02

```
     P SubProc02       B
     D SubProc02       PI

     D Msg             S             20    Inz('Cancel SubProc02')
      /Free
          RegisterCancelStack(%pAddr(CancelStack)
                             :%Addr(Msg)
                             :*Omit);
          // Your code here
          SubProc03();
      /End-Free
     P                 E
```

This is the SubProc03 subprocedure which is called from the SubProc02 subprocedure (Example 31).

Example 31 Program subprocedure SubProc03

```
     P SubProc03       B
     D SubProc03       PI

     D Msg             S             20    Inz('Cancel SubProc03')
      /Free
(1)       RegisterCancelStack(%pAddr(CancelStack)
                             :%Addr(Msg)
                             :*Omit);
          // There might be runnable code here
(2)       UnRegisterCancelStack(%pAddr(CancelStack)
                             :*Omit);
          // Your code here
          Monitor;
(3)          SubProc04();
          On-Error;
          EndMon;
(4)       SubProc05(0);
(5)       SubProc05(1);
```

```
      /End-Free
     P                 E
```

The main points to note are:

- It registers CancelStack as an exit procedure with a pointer to the relevant message as the second parameter.
- It un-registers the exit procedure after some code has been run. The exit procedure would have been called if any of the operations between registering and un-registering had caused the subprocedure to cancel.
- It issues a call to SubProc04. The call is in a MONITOR group since the call is going to fail and you do not want the error to be percolated back up the call stack.
- It issues a call to SubProc05 with a parameter value of 0.
- It issues a call to SubProc05 with a parameter value of 1.

This is the SubProc04 subprocedure which is called from the SubProc03 subprocedure (Example 32). It registers CancelStack as an exit procedure with a pointer to the relevant message as the second parameter. It then performs an invalid divide by zero (does this look familiar?) which causes the call to the subprocedure to end in error; this, in turn, causes the exit procedure for the call stack entry to be called.

Example 32 Program subprocedure SubProc04

```
     P SubProc04       B
     D SubProc04       PI

     D Msg             S             20    Inz('Cancel SubProc04')
     D a               S             10I 0 Inz(20)
     D b               S             10I 0
     D c               S             10I 0
      /Free
          RegisterCancelStack(%pAddr(CancelStack)
                             :%Addr(Msg)
                             :*Omit);
          // Your code here
          c = a/b;
      /End-Free
     P                 E
```

This is the SubProc05 subprocedure which is called twice from the SubProc03 subprocedure (Example 33). It registers CancelStack twice as an exit procedure with a pointer to the relevant message as the second parameter. The subprocedure ends the activation group if the value of the passed parameter is 1.

Example 33 Program subprocedure SubProc05

```
     P SubProc05       B
     D SubProc05       PI
     D  Action                       10I 0 Const

     D Msg1            S             20    Inz('Cancel SubProc05 1')
     D Msg2            S             20    Inz('Cancel SubProc05 2')
     D a               S             10I 0 Inz(20)
     D b               S             10I 0
     D c               S             10I 0
```

```
  /Free
      RegisterCancelStack(%pAddr(CancelStack)
                         :%Addr(Msg1)
                         :*Omit);
      RegisterCancelStack(%pAddr(CancelStack)
                         :%Addr(Msg2)
                         :*Omit);
      // Your code here
      If Action = 1;
         EndAG(*Omit:*Omit);
      EndIf;
  /End-Free
 P                 E
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/CANCEL02) SRCFILE(REDBOOK/EXCEPTSRC)
```

Call the program CANCEL01 three times and the following is displayed:

```
DSPLY          1
DSPLY          2
DSPLY          3
DSPLY  Cancel SubProc04
DSPLY  Cancel SubProc05 2
DSPLY  Cancel SubProc05 1
DSPLY  Cancel SubProc02
DSPLY  AGCancel in CANCEL02
DSPLY  AG ended normally
DSPLY  AG ended by exit request (CEETREC)
```

The third call causes CANCEL02 to be called.

The call to SubProc04 is caused by the divide by zero error.

The final call to SubProc05 results in the AG being ended and all exit procedures being called; starting with the call stack procedures (working back up the call stack) and ending with the AG exit procedures.

Further information

For further information on using the ILE CEE APIs in relation to exception/error handling, refer to section 4.2.7, “Call stack and error handling” of *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402, and Chapter 9, “Exception and Condition Management” in the ILE Concepts manual. In the ILE Programmers Guide (Information Center>Programming>Languages>RPG>ILE RPG>Programmer Guide) look at Calling Programs and Procedures/Using Bindable APIs and Debugging and Exception Handling/Handling Exceptions.

Priority of handlers

When an exception/error occurs in a program or a procedure, this is the priority of the error handlers in RPG IV:

- E operation extender (or error indicator)
- MONITOR group
- ILE condition handler
- File exception/error subroutine or program exception/error subroutine (*PSSR)
- RPG default error handler (which does not apply to subprocedures)

Remember that exception/errors are percolated back up the call stack in an ILE environment, so you must ensure that you are not inadvertently trapping an unexpected exception/error.

Using percolation - try, throw, and catch

There are a myriad ways in which programs process application validation and error notification. There are even more methods available when you introduce subprocedures.

The MONITOR group offers the ability to try running some operations and catching any exception/errors that may occur. With a couple of very simple subprocedures you can use MONITOR groups as a means of catching and identifying application errors as well as exception/errors.

As a means of comparison, let's look at a "normal" means of notifying errors.

A traditional approach

TRYCATCH01 shows an example of a traditional means of performing validation between a buying price and a selling price (Example 34).

Example 34 Sample program TRYCATCH01

```
H DftActGrp(*No) ActGrp(*New) Option(*SrcStmt:*NoDebugIO)

D FatalError      PR                  ExtPgm('REDBOOK/FATALPGMA')

D ValidCosts      PR             7
D  BuyPrice                     11  2 Const
D  SellPrice                    11  2 Const

D Buy             S             11  2
D Sell            S             11  2
D Msg             S              7
 /Free
     Dsply 'Enter Buying Price:' ' ' Buy;
     Dsply 'Enter Selling Price:' ' ' Sell;

     Msg = ValidCosts(Buy:Sell);
     If Msg <> *Blanks;
        Dsply Msg;
     EndIf;
```

```
          *InLR = *On;

          BegSR *PSSR;
             Monitor;
                 FatalError();
             On-Error;
                 Dsply 'Agggghhhhhh!';
             EndMon;
          EndSR '*CANCL';
      /End-Free

     P ValidCosts      B
     D ValidCosts      PI             7
     D  BuyPrice                     11  2 Const
     D  SellPrice                    11  2 Const

     D ErrMsgId        S              7

      /Free
          Select;
             When BuyPrice >= SellPrice;
                ErrMsgId = 'ERR0001';
             When (SellPrice/BuyPrice) > 3 ;
                ErrMsgId = 'ERR0002';
          EndSl;
          Return ErrMsgId;
      /End-Free
     P                 E
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/TRYCATCH01) SRCFILE(REDBOOK/EXCEPTSRC)
```

The program inputs and validates a buying price and a selling price. The ValidCosts subprocedure performs two validations between the two values passed as parameters; it returns a 7-character message ID indicating the error. The calling procedure checks whether the returned value is blanks and, if not, processes any error. While this is a workable method of notifying errors, it is dependent on the calling procedure checking the return value from the subprocedure.

Call the program a few times providing valid and invalid values for the buying and selling prices. Note how the standard exception/error handling is invoked if you provide a value of zero for the buying price: it causes a divide by zero error in the ValidCosts subprocedure.

Throw

Instead of simply returning a message ID, you can instigate your own exception/error by having a program or procedure send an escape message. The easiest way to this is to write a subprocedure to do it.

This requires a couple of additions to the STDMSGINFO copy member (Example 35).

Example 35 Additions to the STDMSGINFO copy member

```
     D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
```

```
D  ErrorMsgId                    7    Const
D  MsgFile                      20    Const
D  MsgData                    3000    Const
D  MsgDtaLen                    10I 0 Const
D  MsgType                      10    Const
D  CallStack                    19    Const
D  CallStackCtr                 10I 0 Const
D  MsgKey                        4
D  ErrorForAPI                        Like(APIError)

D Throw           PR                  ExtProc('Throw')
D  MsgId                         7    Const
D  MsgDataIn                  3000    Const Varying
D                                     Options(*Omit:*NoPass)
D  MsgFileIn                    20    Const
D                                     Options(*NoPass)
```

SndPgmMsg is a prototype for the Send Program Message (QMHSNDPM) API. QMHSNDPM is called to send an escape message back up the call stack.

Throw is a prototype for a subprocedure that calls the QMHSNDPM API to send a requested message back up the call stack.

The Throw subprocedure is coded in the source member MSGPROCS (Example 36).

Example 36   Program subprocedure Throw

```
(1)  H NoMain Option(*SrcStmt:*NoDebugIO)

      /Copy EXCEPTSRC,STDMSGINFO

     P Throw           B                   Export
     D Throw           PI
     D  MsgId                         7    Const
     D  MsgDataIn                  3000    Const Varying
     D                                     Options(*Omit:*NoPass)
     D  MsgFileIn                    20    Const
     D                                     Options(*NoPass)

(2)  D MsgFile         S             20    Inz('ERRORS    REDBOOK   ')
     D MsgData         S           3000
     D MsgKey          S              4


      /Free

(3)       If %Parms() > 2;
              MsgFile = MsgFileIn;
          EndIf;

          If %Parms() > 1;
              If %Addr(MsgDataIn) <> *Null;
                  MsgData = MsgDataIn;
              EndIf;
          EndIf;
```

```
(4)       SndPgmMsg( MsgId
                   : MsgFile
                   : MsgData
                   : %Len(%Trim(MsgData))
                   : '*ESCAPE'
                   : '*'
                   : 1
                   : MsgKey
                   : APIError);
          Return;

      /End-Free
     P                 E
```

The main points to note are:

- The NOMAIN keyword is specified on the H spec since MSGPROCS only contains subprocedures.
- The default message file is ERRORS in the library REDBOOK (this file may be overridden by the third parameter passed). This message file contains the messages ERR0001 and ERR0002.
- The subprocedure determines if optional parameters were passed and overrides work fields accordingly.
- The SndPgmMsg procedure is called to send the requested error message. The message is sent as an escape message and is sent to the previous entry in the call stack.

Catch

RPG's standard exception/error handling will now trap any message you send (using the Throw procedure) as a 202 status error (called program or procedure failed). You need another subprocedure to catch the error so you can determine what it is.

Again, this requires a couple of additions to the STDMSGINFO copy member (Example 37).

Example 37 Additions to STDMSGINFO copy member

```
D Catch          PR                 ExtProc('CATCH')
     D  CaughtMessage                    LikeDS(Base_CaughtMessage)

     D Base_CaughtMessage...
     D                DS                 BASED(DummyPtr)
     D                                   Qualified
     D  MsgId                       7A
     D  MsgFile                    20A
     D   MsgFileName               10A   OverLay(MsgFile)
     D   MsgLibName                10A   OverLay(MsgFile:*Next)
     D  MsgText                   132A
     D  Msgdata                  3000A
```

Catch is a prototype for a subprocedure that retrieves a thrown message. Message information is returned in the data structure passed as a parameter.

Base_CaughtMessage defines the data structure passed as a parameter to Catch. The data structure contains subfields defining the message ID, message file, first level message text, and message data for the message retrieved by the Catch subprocedure.

The Catch subprocedure is also coded in the source member MSGPROCS (Example 38).

Example 38   Program subprocedure Catch

```
     P Catch           B                   Export
     D Catch           PI
     D  CaughtMessage                      LikeDS(Base_CaughtMessage)

     D MsgBack         DS                  LikeDs(RCVM0300) Inz
     D SetMsgKey       S              4    Inz(*ALLx'00')
      /Free
           Clear CaughtMessage;
           ReceiveMsg( MsgBack
                     : %size(MsgBack)
                     : 'RCVM0300'
                     : '*'
                     : 1
                     : '*PRV'
                     : SetMsgKey
                     : 0
                     : '*REMOVE'
                     : APIError);

           If MsgBack.ByteAvail > 0;
              CaughtMessage.MsgId = MsgBack.MsgId;
              CaughtMessage.MsgFileName = MsgBack.MsgFileName;
              CaughtMessage.MsgLibName = MsgBack.MsgLibUsed;
              CaughtMessage.MsgText =
                         %SubSt(MsgBack.MsgData:
                                MsgBack.LenReplace1 + 1:
                                MsgBack.LenMsgReturn);
              If MsgBack.LenReplace1 > 0;
                 CaughtMessage.MsgData =
                            %SubSt(MsgBack.MsgData:
                                   1:
                                   MsgBack.LenReplace1);

              EndIf;
           EndIf;

      /End-Free
     P                 E
```

Create the MSGPROCS service program with these commands:

```
CRTRPGMOD MOSULE(REDBOOK/MSGPROCS) SRCFILE(REDBOOK/EXCEPTSRC)
CRTSRVPGM SRVPGM(REDBOOK/MSGPROCS) EXPORT(*ALL)
```

The binding directory ERROR contains an entry for the MSGPROCS service program.

Catch uses the ReceiveMsg procedure (QMHRCVPM) to retrieve the last message in the program message queue of the calling program or procedure and places the retrieved information in the parameter data structure.

Using throw and catch

TRYCATCH02 shows the implementation of the Throw and Catch subprocedures (Example 39).

Example 39 Sample program TRYCATCH02

```
     H DftActGrp(*No) ActGrp(*New) Option(*SrcStmt:*NoDebugIO)
(1)  H BndDir('ERROR')

(2)   /Copy EXCEPTSRC,STDMSGINFO

     D FatalError      PR                  ExtPgm('REDBOOK/FATALPGMA')

(3)  D ValidCosts      PR
     D  BuyPrice                     11  2 Const
     D  SellPrice                    11  2 Const

(4)  D CaughtMessage   DS                  LikeDS(Base_CaughtMessage)
     D Buy             S             11  2
     D Sell            S             11  2
      /Free
          Dsply 'Enter Buying Price:' ' ' Buy;
          Dsply 'Enter Selling Price:' ' ' Sell;

(5)       Monitor;
             ValidCosts(Buy:Sell);
          On-Error;
(6)          Catch(CaughtMessage);
             Dsply CaughtMessage.MsgId;
          EndMon;
          *InLR = *On;

          BegSR *PSSR;
             Monitor;
                 FatalError();
             On-Error;
                 Dsply 'Agggghhhhhh!';
             EndMon;
          EndSR '*CANCL';
      /End-Free

     P ValidCosts      B
     D ValidCosts      PI
     D  BuyPrice                     11  2 Const
     D  SellPrice                    11  2 Const

      /Free
          Select;
             When BuyPrice >= SellPrice;
(7)             Throw('ERR0001');
```

```
           When (SellPrice/BuyPrice) > 3 ;
              Throw('ERR0002');
        EndSl;
        Return;
    /End-Free
   P                 E
```

Create this program with the command:

```
CRTBNDRPG PGM(REDBOOK/TRYCATCH02) SRCFILE(REDBOOK/EXCEPTSRC)
```

The main points to note are:

- The ERROR binding directory is specified on the H spec in order to bind to the Throw and Catch subprocedures in the MSGPROCS service program.
- The STDMSGINFO copy member is included for the required prototypes.
- ValidCosts no longer returns a message ID.
- CaughtMessage is used as the parameter for the Caught subprocedure.
- The call to ValidCosts is placed in a MONITOR group. Any exception/error will cause control to pass to the ON-ERROR.
- The Catch procedure is called to determine what the error was.
- When there is an error, the Throw procedure is called to send the relevant error message and signal an exception/error to the calling program or procedure.

The problem with Throw and Catch

There are two issues you must be aware of when using Throw and Catch.

The process of calling the Throw subprocedure immediately ends the subprocedure that issues the call. If the ValidateCosts subprocedure were coded as follows:

```
If BuyPrice >= SellPrice;
  Throw('ERR0001');
EndIf;
If (SellPrice/BuyPrice) > 3 ;
  Throw('ERR0002');
EndIf;
```

The second validation would not be performed if the first validation caused ERR0001 to be thrown. The fact that the Throw procedure sends an escape message means that the call to the Throw procedure ends in error. Since this error is not trapped, the procedure ends immediately.

When you call TRYCATCH02 and provide a value of 0, the divide by zero error is now trapped by the MONITOR and control passes to the ON-ERROR operation as opposed to the exception/error being percolated up the call stack.

You could try to handle these exception/errors after the catch but it is probably easier to handle them in the Catch subprocedure.

The Catch subprocedure can check the message ID retrieved and if it is not an application error (starts with ERR), the message is re-thrown. The following code is added to the Catch subprocedure after the contents of the parameter data structure have been set.

```
If %SubSt(CaughtMessage.MsgId:1:3) <> 'ERR';
  Throw( CaughtMessage.MsgId
       : CaughtMessage.MsgData
       : CaughtMessage.MsgFile);
EndIf;
```

The problem with re-throwing the error is that the Catch subprocedure is now identified as the subprocedure that received the exception/error.

Another alternative would be to have the Catch subprocedure call the FatalError procedure. This method would require two changes: the message action parameter on the call to ReceiveMsg in the Catch subprocedure should be *SAME as opposed to *REMOVE and the Call Stack Counter parameter on the call to ReceiveMsg in the FATALPGMA program should be 3 as opposed to 2 (you want it to retrieve the message from the program message queue of the program or procedure that called Catch).

Conclusion

The implementation of exception/error handling in your programs is straightforward.

In an environment where you are not using ILE features such as subprocedures (that is, the programs are running in the default activation group), you can use a combination of the following:

- Write a standard *PSSR subroutine and place it in a copy member.
- Write a “Fatal Error” program that is called from the *PSSR subroutine.
- Include the *PSSR subroutine in programs using a /COPY directive.
- Include an INFSR(*PSSR) keyword for every file.
- Implicitly open files.
- Use E extenders and Monitor groups to handle specific exception errors.

In an environment where you are running programs in ILE activation groups (that is, the programs are not running in the default activation group) you can also write and register your own condition handlers and exit procedures.

Just think of all the frustration and irritation that can be saved by making these few changes!

The team that wrote this IBM Redpaper

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Gary Mullen-Schultz is a certified Consulting IT Specialist at the ITSO, Rochester Center. He leads the team responsible for producing RoadRunner and Blue Gene/L™ documentation, and focuses on i5/OS® application development topics such as PHP, Java™, and WebSphere®. He is a Sun™ Certified Java Programmer, Developer, and Architect, and has three issued patents.

Susan Gantner has experience spanning over 24 years in the field of application development. She began as a programmer developing applications for corporations in

Atlanta, Georgia, working with a variety of hardware and software platforms. She joined IBM in 1985 and quickly developed a close association with the Rochester laboratory during the development of the AS/400® system.
Susan worked in Rochester, Minnesota for five years in the AS/400 Technical Support Center. She later moved to the IBM Toronto Software Laboratory to provide technical support for programming languages and AD tools on the AS/400. She left IBM in 1999 to devote more time to teaching and consulting. Her primary emphasis is on enabling customers to take advantage of the latest programming and database technologies on OS/400®.
Susan is one of the founding members of System i™ Developer. She is a regular speaker at COMMON conferences and other technical conferences around the world and holds a number of Speaker Excellence medals from COMMON. Susan is a founding member of System i Developer, whose Web site is:

(http://www.systemideveloper.com).

Jon Paris started his career with IBM midrange systems when he fell in love with the System/38™ while working as a consultant. This love affair ultimately led him to joining IBM. In 1987, Jon was hired by the IBM Toronto Laboratory to work on the S/36 and S/38 COBOL compilers. Subsequently Jon became involved with the AS/400 and in particular COBOL/400®.
In early 1989 Jon was transferred to the Languages Architecture and Planning Group, with particular responsibility for the COBOL and RPG languages. There he played a major role in the definition of the new RPG IV language and in promoting its use with IBM Business Partners and Users. He was also heavily involved in producing educational and other support materials and services related to other AS/400 programming languages and development tools, such as CODE/400 and VisualAge® for RPG.
Jon left IBM in 1998 to focus on developing and delivering education focused on enhancing AS/400 and iSeries® application development skills.
Jon is one of the founding members of System i Developer. He is a frequent speaker at User Groups meetings and conferences around the world, and holds a number of speaker excellence awards from COMMON. Jon is a founding member of System i Developer.

Paul Tuohy has worked in the development of IBM midrange applications since the '70s. He has been IT manager for Kodak Ireland Ltd. and Technical Director of Precision Software Ltd. and is currently CEO of ComCon, a midrange consultancy company based in Dublin, Ireland. He has been teaching and lecturing since the mid-'80s.
Paul is the author of *Re-engineering RPG Legacy Applications*, *The Programmers Guide to iSeries Navigator* and the self-teach course “iSeries Navigator for Programmers.” He writes regular articles for many publications and is one of the quoted industry experts in the IBM Redbooks publication *Who knew you could do that with RPG IV?*
As well as speaking at RPG & DB2® Summit®, Paul is also an award-winning speaker who speaks regularly at US COMMON and other conferences throughout the world. Paul is a founding member of System i Developer.

Thanks to the following people for their contributions to this project:

Jenifer Servais
International Technical Support Organization, Rochester Center

Notices

This document REDP-4321-00 was created or updated on February 26, 2008.

Send us your comments in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
- Send your comments in an email to:
  redbooks@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Redpaper™

Trademarks