Dodging Bullets in Your DP Shop

Victoria Shoemaker
Taurus Software, Inc.
770 Welch Road Suite 3A
Palo Alto, CA 94304

Introduction

At last count, there were exactly 3,572,614 HP3000 programs that mishandled error conditions. Do any of your coworker's programs write records to already full data sets without giving anyone a clue? How many of you have received phone calls at obscure hours by some poor user wondering exactly what is going on in this $STDLIST? What percentage of operators do you think know how to read an octal stack dump and then fix the problem? The cost of recovering from a program that kept on running when it should have stopped because something was wrong can be staggering.

Error handling within a computer programs and JCL means recognizing error conditions and taking appropriate action. There are generally three courses of action that can be taken when an error is detected:

A. Recover from the error,
B. Print an error message and abort the program,
C. Ignore the error entirely.

All too often programs take Action C. This can lead to countless headaches and nightmares for users and programmers alike, and is rarely the best way to handle an error condition.

Action B, print an error message and abort is an acceptable method of handling errors; however, it is often used as a copout by lazy programmers. Unfortunately for most of us, this is the method used by MPE when it encounters an error: Print a system failure message on the console, and die.

Action A, recover from the error, is often the best method of error handling, but it is also the most difficult and costliest one to implement. It would be unreasonable to always attempt to recover from error conditions. If your program can't open the data base, it's tough to recover, so print an error message and

abort. Batch programs should abort more often than online programs. Often an online program should print an error message to the user and let the user decide what to do next. You wouldn't want your editor to abort if you tried to Text in a file that you misspelled.

The key to good error handling is to detect the error as soon as it occurs. Whenever you wait, assuming any errors will get detected down the road, you run the risk not being able to figure out the cause of the problem or an even worse fate of never discovering there was a problem until it's too late.

As an example: What if a program that writes to a database, doesn't check to see if the DBPUT worked? (As I've seen before) If it's an online program, the user may simply keep entering data for hours without knowing that all of her bits are going into the great bit bucket in the sky. If it's a batch program, then maybe hundreds of honest hard-working employees mysteriously won't get paychecks on Friday.

Error handling within programs

The first step to handling an error condition is detecting it. The second step is for the program to decide what to do with it: recover, abort, or ignore.

Detecting errors

For example: If your program is reading down a chain in an Image detail data set when the DBGET fails, what should the program do? Your program should probably be able to recover if the error is an end-of-chain error, but should probably abort with an error message if it's any other Image error. Your program should check for an error condition after every system procedure call. It cannot be stressed enough, how important it is to check for an error condition after EVERY system procedure call. Remember, the sooner the error is detected, the better off you and everyone else will be.

Processing Errors

Whenever a program makes a system procedure or intrinsic call, the following steps should be taken after checking for an error to ensure proper error handling:

- If no error occurred, then continue processing
- If an error occurred:

    1. Retrieve error number
    2. Determine if error is recoverable.

        If recoverable, recover.

        If error is not recoverable:

        1. Retrieve and print error message based on error number.
        2. Abort the program, if appropriate.

When to Abort

When should the program abort and when should the program simply print an error message and then continue processing?

1. All severe errors should abort the program.
2. Errors within a batch program should abort.

3. Errors within an online program should print a message and continue if possible.

4. Programs that may run batch or online SHOULD CHECK whether they are being in batch, or online using either the WHO or FRELATE intrinsic and abort or continue accordingly. This is where many programs have problems.

File system errors should be detected by checking the condition code after every file system intrinsic call. The condition code is part of the hardware status word and is set by every file system intrinsic. The condition code can have one of three different values:

CCE - Condition Code Equal. This means that the intrinsic call was successful.

CCG - Condition Code Greater than. This means that a warning condition occurred, and that the intrinsic call may or may not have worked, depending upon which intrinsic was called. Check the intrinsics manual.

CCL - Condition Code Less than. This means that an error condition occurred and that the intrinsic failed.

Checking Condition Code in SPL

In SPL, check the condition code by simply using a relational operator with no expression. The condition code must be checked immediately after the intrinsic call. In addition, be careful not to assign the return value of an intrinsic call into an indexed array because this will destroy the condition code returned by the intrinsic call.

Some examples:

```
LEN := FREAD(FNUM, BUF, BUFLEN);
IF = THEN
   PROCESS'RECORD
ELSE
   IF > THEN  << END OF FILE REACHED >>
      END'OF'FILE
   ELSE  << SOME OTHER ERROR >>
      FILE'SYSTEM'ERROR;

FNUM := FOPEN(FILENAME, FOPTS, AOPTS);
IF < THEN
   FILE'OPEN'ERROR;
<< WE DO NOT NEED TO CHECK CCG, BECAUSE
   FOPEN DOESN'T RETURN IT >>
```

Do **NOT** do the following:

```
FILENUMBER(N) := FOPEN(FNAME, 3);
```

The array index N, will destroy the condition code.

```
BUFLENGTH := FREAD(FILENUM, BUFFER, LEN);
NUMREADS := NUMREADS + 1;
IF <> THEN   << READ FAILED >>
   HANDLE'READERROR;
```

This will not work, the statement after the FREAD will destroy the condition code.

Checking Condition Code in PASCAL

In Pascal the condition code may be checked by using the CCODE function. The CCODE function works as if it were a local variable to the current procedure that is set each time an intrinsic is called. CCODE may be checked any time before the next intrinsic call within the same procedure. Unlike SPL, the condition code does not need to be checked immediately after the intrinsic call, and you may assign the result of and intrinsic call into an array. The CCODE function returns the following values: 0, for CCG, 1, for CCL, and 2 for CCE. It often helps to use a "const" statement at the beginning of your program defining these three values. Example:

```
LEN := FREAD(FNUM, BUF, BUFLEN);
IF CCODE = 2 THEN  (* READ WAS OK *)
   PROCESS_RECORD
ELSE IF CCODE = 0 THEN
   DO_END_OF_FILE
ELSE  (* CCODE MUST BE 1 *)
   FILE_SYSTEM_ERROR;
```

Checking Condition Code in COBOL

You can only check condition codes in COBOL II. You must define the name of your condition code variable within the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. The COBOL condition code is similar to the SPL condition code in that you must check it immediately after the intrinsic call and may not assign the result of the intrinsic call to an indexed variable. The condition code variable may only be compared with zero. Condition code equal to zero means CCE, less than zero CCL, and greater than zero CCG. Example:

```
ENVIRONMENT DIVISION.
SPECIAL-NAMES.
   CONDITION-CODE IS CONDCODE.
.
```

```
PROCEDURE DIVISION.
.
CALL INTRINSIC "FREAD"        USING FNUM, BUFFER,
BUFFERLEN
                              GIVING BYTESREAD.
IF CONDCODE > 0 THEN
   PERFORM END-OF-FILE
ELSE
   IF CONDCODE < 0 THEN
      PERFORM HANDLE-FILE-ERROR
   ELSE
      PERFORM PROCESS-RECORD.
```

Printing Error Messages

Whenever your program detects a file system error that it does not know how to recover from, it should print an error message and stop. There are several intrinsics that facilitate this.

FCHECK Intrinsic

The FCHECK intrinsic is used to request the file system error that has most recently occurred. The first two parameters to FCHECK are:

- the file number of the file on which the error occurred
- the error number to be returned to the program.

If an error occurred during an FOPEN intrinsic, a value of zero should be passed to FCHECK as the file number. Be careful when using FCHECK because there is an unfortunate ambiguity with file system error numbers. A file system error of zero can mean one of three things occurred:

- The end of file was reached.
- There was **no** file system error.
- In rare circumstances a file system error occurred, but the system did not set the internal error number for FCHECK to retrieve.

Other intrinsics

The FERRMSG intrinsic is used to translate the file system error number into an error message. This message is usually more helpful than simply printing the file system error number.

PRINTFILEINFO prints a file tombstone and is another intrinsic that is often called when a file system error is detected. EDITOR, FCOPY and other programs call PRINTFILEINFO when they discover a problem.

The undocumented GENMSGU intrinsic can be used to print out a file system error message. It has two single word integer parameters passed by value. The first parameter is the message set number from CATALOG.PUB.SYS, and the second parameter is the error message number. Use message set 8 for file system errors. (Set number 2 can be used for MPE errors returned by the COMMAND intrinsic.)

Image error handling is a bit easier than the file system. After every Image call, check the first word of the status array. If the first word is zero, then what you tried to do worked; otherwise it didn't. If the program understands the error number, such as 15 for end-of-chain, the program should be able to take appropriate recovery. If the program doesn't understand the error number, it should go into error mode. There are basically two ways to handle Image errors:

- Call DBERROR to get the Image error message, print the message and then resume with the program.
- Call DBEXPLAIN to print all pertinent Image error information, then abort the program.

VPLUS error handling is similar to Image error handling. After every VPLUS procedure call check the first word of the COM area. If the first word is zero then everything is OK, otherwise you've got problems.

Usually, the program should call VERRMSG to get the error message, then should zero the status word, then call VPUTWINDOW to put the error message in the VPLUS window. If the program decides to abort, the program should either call VCLOSETERM before printing any error messages or it should do the following:

- Call the FCONTROL intrinsic using the 49th word of the VPLUS COMAREA as the file number and 12 as the control code to turn the terminal echo back on for the user.

- Escape sequences should also be printed to the terminal to turn format mode off (<esc>X), turn block mode off (<esc>&k0B), turn memory lock off (<esc>m), and home down (<esc>F).

There are many other intrinsics, virtually all of which return a status code via the condition code, or return an error number or both. Unfortunately, MPE does not provide any mechanism for converting an error number into error message. The program must either convert the error number itself, or simply print the error number out as part of the error message.

When a program encounters an error that it cannot recover from, then it should abort by going through a special abort procedure. The abort procedure should do the following:

1. Print the file system/IMAGE/VPLUS error message that caused the abort.

2. Print an error message specific to the program and location within the program that detected the error. Ideally, no two error messages detected from different points within a program should print the same message.

3. Terminate the program by calling the QUIT intrinsic, do not use STOP RUN in COBOL programs or the TERMINATE intrinsic to abort a program.

As an alternative to calling QUIT, the program may set the high order bit of the system JCW by calling the SETJCW with a negative value, then call the TERMINATE intrinsic. Doing this causes the program to end in an error state. The QUIT intrinsic does this automatically for the user. Note that the JCW may also be set with the PUTJCW intrinsic, or the COMMAND intrinsic with a SETJCW command.

Error handling within job streams is often done carelessly. The :CONTINUE command should only be used when necessary. Wanton placement of :CONTINUE commands within a job can be hazardous to your health.

For example, many programmers make the mistake of putting :CONTINUE commands before :PURGE commands. This is almost always incorrect. If the :PURGE command attempts to purge a file that does not exist, then a WARNING is issued, not an error. No :CONTINUE command is necessary for the job to continue. In most circumstances in which the :PURGE command fails, the job *should* stop running because there is a problem, such as the file being accessed by another program.

Using JCW

The system defined job control words (JCWs) CIERROR and JCW can be used very successfully within job streams. These JCWs are managed by both the system and the user.

The JCW called CIERROR is set by the MPE whenever an error or warning occurs with an MPE command. It is set to the command interpreter error number if there is a problem, otherwise its value is not changed. Unfortunately, there is no way to tell by looking at the number whether it is an error or a warning. There are many uses of CIERROR. This example will check if a file exists:

```
:SETJCW CIERROR = 0
:CONTINUE
:LISTF MYFILE;$NULL
:IF CIERROR = 0 THEN
:  TELLOP MYFILE is alive and well.
:ELSE
: TELLOP HELP! MYFILE is not there!
:ENDIF
```

The JCW called JCW is used to help the job determine what happened with a program run. When a program is run, MPE sets JCW to zero.

If the program is successful, then JCW may be set to a value from zero to 16383 to indicate its success.

If a warning occurs during the program, then the program may set JCW to a value from 16384 to 32767.

If an error occurs during the program, then the program should set the value of JCW to a value from 32768 to 65535.

If a program terminates with the value of JCW from 32768 to 65535, then MPE will consider that the program has terminated abnormally and generate command interpreter error 989. A job would have had to have a :CONTINUE command before the run of the program for the job to continue.

Aborting a Job

When a job encounters a nonrecoverable error, it should abort. This is most easily done by doing nothing because MPE will handle it for you if you don't use :CONTINUEs and your programs abort properly. By using this method, an operator can always tell if a job succeeded or failed by taking a quick glance at the bottom of the $STDLIST.

A perhaps better way of aborting a job is to use :CONTINUE commands before each run of a program, then use JCW checking to determine if the program succeeded. If ever a program within the job fails, use the :TELLOP command to notify the operator that the job has failed. Example below:

```
!JOB AR1002J,BATCH.AR
!SETJCW ERROR, 0
!CONTINUE
!RUN AR10021P.PROG.AR
!IF JCW >= FATAL THEN
!   SETJCW ERROR = 1
!ENDIF
!IF ERROR = 0 THEN
!   CONTINUE
!   RUN AR10022P.PROG.AR
!ENDIF
!IF ERROR = 0 AND JCW >= FATAL THEN
!   SETJCW ERROR = 2
!ENDIF
!IF ERROR = 0 THEN
!   CONTINUE
!   RUN AR10023P.PROG.AR
!ENDIF
!IF ERROR = 0 AND JCW >= FATAL THEN
!   SETJCW ERROR = 3
!ENDIF
!
!IF ERROR <> 0 THEN
!   TELLOP **************************************
!   TELLOP **      JOB                          **
!   TELLOP **      AR1002J                      **
!   TELLOP **      FAILED!                      **
```

```
!   TELLOP **************************************
!   SHOWJCW
!   ABORT
!ELSE
!   TELLOP Job AR1002J completed successfully.
!ENDIF
!EOJ
```

This job stream has several noteable features:

1. It is written is such a way that can be easily read and modified. The :IF statements never get more than one level deep.

2. When the job fails an easily identifiable message is printed on the console for the operator. Presumably, the program that failed has printed an error message on the $STDLIST that will enable the operator or programmer to pinpoint the problem.

3. By using the :ABORT command to terminate the job stream, MPE will stop processing the job and print a message at the bottom of the $STDLIST that the operator can easily recognize as a failed job. Note that the :ABORT command is not intended for this purpose but serves nicely.

4. When the job succeeds, a simple message is printed to the console and the job terminates with an :EOJ command, a signal to the operator reading the $STDLIST that the job was successful.

Let's not forgot abort user-defined commands. Basically, error handling within UDCs is the same as it is within job streams. The :CONTINUE command performs the same function in UDCs as it does within jobs, it allows the rest of the UDC to complete if one of the commands encounters an error. An example of this would a copy UDC as follows:

```
COPY FROMFILE,TOFILE
FILE INPUT = !FROMFILE
FILE OUTPUT = !TOFILE
CONTINUE
RUN MYCOPY
RESET INPUT
RESET OUTPUT
****
```

This UDC would make certain that the INPUT and OUTPUT file equations were reset regardless of the success of the program MYCOPY.

Programatic error handling:

1. Check the condition code or status word after EVERY intrinsic or system procedure call.

2. Retrieve and display system error message whenever appropriate.

3. When a program running within a job cannot recover from an error, it should always abort, even if it is an online program.

4. When a program running online detects an error, print a message and continue if possible, otherwise abort.

5. When aborting a program, always print a unique error message in addition to the system error message, and always set the job control word JCW to a fatal value.

Error handling within jobs:

1. Do not abuse the :CONTINUE command. If there is a problem with the job, it should abort.

2. Use the :CONTINUE command before each command that could fail; then use JCW checking to ensure it succeeded.

3. If a job fails, print an easily recognizable message on the console.

4. Make certain that an operator can easily determine if a job failed by glancing at the $STDLIST.

5. Successful jobs should always end with an :EOJ command.

Conclusion

For a data processing shop to run as smoothly as possible, the programs and job streams need to be written so that errors get detected as soon as possible after they happen. Once an error is detected, appropriate handling of the error is imperative, whether it be a simple error message or a program abort.

By following the guidelines in described in this paper, you will be well on the road to DP pie in the sky.