OPL

ERROR HANDLING

CONTENTS

SYNTAX ERRORS

Syntax errors are those which are reported when translating a procedure. (Other errors can occur while you're running a program.) The OPL translator will return you to the line where the first syntax error is detected.

All programming languages are very particular about the way commands and functions are used, especially in the way program statements are laid out. Below are a number of errors which are easy to make in OPL. The incorrect statements are in bold and the correct versions are on the right.

PUNCTUATION ERRORS

Omitting the colon between statements on a multi-statement line:

Incorrect	*Correct*
`a$="text"` **`PRINT a$`**	`a$="text" :PRINT a$`

Omitting the space before the colon between statements:

Incorrect	*Correct*
`a$=b$`**`:PRINT a$`**	`a$=b$ :PRINT a$`

Omitting the colon after a called procedure name:

Incorrect	*Correct*
`PROC proc1:`	`PROC proc1:`
`GLOBAL a,b,c`	`GLOBAL a,b,c`
`.`	`.`
`proc2`	`proc2:`
`ENDP`	`ENDP`

Using only 1 colon after a label in GOTO/ONERR/VECTOR (instead of 0 or 2):

Incorrect	*Correct*
`GOTO` **`below:`**	`GOTO below`
`.`	`.`
`below::`	`below::`

STRUCTURE ERRORS

The DO...UNTIL, WHILE...ENDWH and IF...ENDIF structures can produce a 'Structure fault' error if used incorrectly:

- Mixing up the three structures - e.g. by using DO...WHILE instead of DO...UNTIL.
- Using BREAK or CONTINUE in the wrong place.
- Using ELSE IF with a space, instead of ELSEIF.
- VECTOR...ENDV can also produce a 'Structure fault' error if used incorrectly.

Attempting to nest any combination of these structures more than eight levels deep will produce a 'Too complex' error.

ERRORS IN RUNNING PROCEDURES

OPL may display an error message and stop a running program if certain 'error' conditions occur. This may happen because:

- There is a mistake, or *bug*, in your program, which could not be detected during translation - for example, a calculation has involved a division by zero.
- A problem has occurred which prevents a command or function from working - for example, an APPEND command may fail because a disk is full.

Unless you include statements which can handle such errors when they occur, OPL will use its own error handling mechanism. The program will stop and an error message be displayed. The first line gives the names of the procedure in which the error occurred, and the module this procedure is in. The second line is the 'error message' - one of the messages listed at the end of this section. If appropriate, you will also see a list of variable names or procedure names causing the error.

If you were editing the module with the Program editor and you ran it from there, you would also be taken back to editing the OPL module, with the cursor at the line where the error occurred.

ERROR HANDLING FUNCTIONS AND COMMANDS

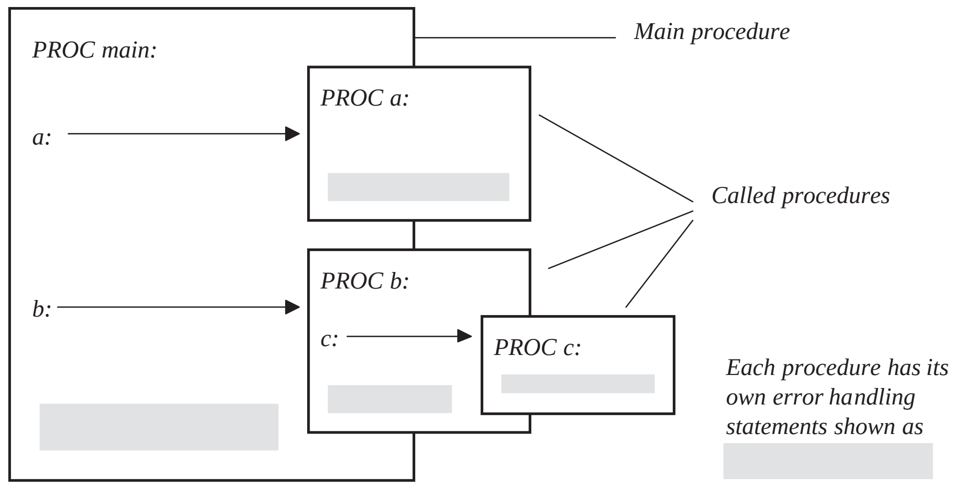
To prevent your program being stopped by OPL when an error occurs, include statements in your program which anticipate possible errors and take appropriate action. The following error handling facilities are available in OPL:

- TRAP temporarily suppresses OPL's error processing.
- ERR and ERR$ (and ERRX$ on the Series 5) find out what kind of error has occurred.
- ONERR establishes an error handler which can suppress OPL's error processing over whole modules.
- RAISE can be used to simulate error conditions.

These facilities put you in control and must be used carefully.

STRATEGY

You should design the error handling of a program in the same way as the program itself. OPL works best when programs are built up from procedures, and you should design your error handling on the same basis. Each procedure should normally contain its own local error handling:



The error handling statements can then be appropriate to the procedure. For example, a procedure which performs a calculation would have one type of error handling, but another procedure which offers a set of choices would have another.

TRAP

❺ TRAP can be used with any of these commands: APPEND, BACK, CANCEL, CLOSE, COPY, CREATE, DELETE, ERASE, EDIT, FIRST, gCLOSE, gCOPY, gFONT, gPATT, gSAVEBIT, gUNLOADFONT, gUSE, INPUT, INSERT, LAST, LCLOSE, LOADM, LOPEN, MKDIR, MODIFY, NEXT, OPEN, OPENR, POSITION, PUT, RAISE (see below), RENAME, RMDIR, UNLOADM, UPDATE and USE.

❸ TRAP can be used with any of these commands: APPEND, BACK, CACHE, CLOSE, COMPRESS, COPY, CREATE, DELETE, ERASE, EDIT, FIRST, gCLOSE, gCOPY, gFONT, gPATT, gSAVEBIT, gUNLOADFONT, gUSE, INPUT, LAST, LCLOSE, LOADM, LOPEN, MKDIR, NEXT, OPEN, OPENR, POSITION, RENAME, RMDIR, UNLOADM, UPDATE and USE.

The TRAP command immediately precedes any of these commands, separated from it by a space - for example:

```
TRAP INPUT a%
```

If an error occurs in the execution of the command, the program does not stop, and the next line of the program executes as if there had been no error. Normally you would use ERR on the line after the TRAP to find out what the error was.

EXAMPLE

When INPUT is used without TRAP and a text string is entered when a number is required, the display just scrolls up and a `?` is shown, prompting for another entry. With TRAP in front of INPUT, you can handle bad entries yourself:

```
PROC trapinp:
  LOCAL profit%
  DO
  PRINT
  PRINT “Enter profit”,
  TRAP INPUT profit%
  UNTIL ERR=0
  PRINT “Valid number”
  GET
ENDP
```

This example uses the ERR function, described next.

ERR, ERR$ AND ERRX$

When an error occurs in a program, check what number the error was, with the ERR function:

```
e%=ERR
```

If ERR returns zero, there was no error. The value returned by ERR is the number of the last error which occurred it changes when a new error occurs. TRAP sets ERR to zero if no error occurred. Check the number it returns against the error messages listed at the end of this section.

The ERR$ function gives you the message for error number `e%`:

```
e$=ERR$(e%)
```

You can also use ERR and ERR$ together:

```
e$=ERR$(ERR)
```

This returns the error message for the most recent error.

❺ The ERRX$ function gives you the extended message for the current error:

```
e$=ERRX$
```

For example, ‘Error in *MODULE\PROCEDURE*,*EXTERN1*,*EXTERN2*,...’. This is the message which would have been presented as an alert if the error had not been trapped. The use of this function gives the list of missing externals and procedure names when an error has been trapped.

EXAMPLE

The lines below anticipate that error number -101 (‘File already open’) may occur. If it does, an appropriate message is displayed.

```
TRAP OPEN “main”,A,a$
e%=ERR
IF e%              REM Checks for an error
  IF e%=-101
    PRINT “File is already open!”
  ELSE
```

```
    PRINT ERR$(e%)
  ENDIF
ENDIF
```

The inner IF...ENDIF structure displays either the message in quotes if the error was number -101, or the standard error message for any other error.

TRAP INPUT/EDIT AND THE ESC KEY

If in response to a `TRAP INPUT` or `TRAP EDIT` statement, the Esc key is pressed while no text is on the input/ edit line, the ‘Escape key pressed’ error (number -114) will be raised. (This error will only be raised if the INPUT or EDIT has been trapped. Otherwise, the Esc key still leaves you editing.)

You can use this feature to enable someone to press the Esc key to escape from editing or inputting a value. For example:

```
PROC trapInp:
  LOCAL a%,b%,c%
  PRINT “Enter values.”
  PRINT “Press Esc to exit”
  PRINT “a% =”, :TRAP INPUT a% :PRINT
  IF ERR=-114 :GOTO end :ENDIF
  PRINT “b% =”, :TRAP INPUT b% :PRINT
  IF ERR=-114 :GOTO end :ENDIF
    PRINT “a%*b% =”,a%*b%
    PAUSE -40
    RETURN
end::
  PRINT :PRINT “OK, finishing...”
  PAUSE -40
  RETURN
ENDP
```

ONERR...ONERR OFF

ONERR sets up an error handler. This means that, whenever an error occurs in the procedure containing ONERR, the program will jump to a specified label instead of stopping in the normal way. This error handler is active until an ONERR OFF statement.

You specify the label after the word ONERR.

The label itself can then occur anywhere in the same procedure - even above the ONERR statement. After the label should come the statements handling whatever error may have caused the program to jump there. For example, you could just have the statement `PRINT ERR$(ERR)` to display the message for whatever error occurred.

All statements after the ONERR command, including those in procedures called by the procedure containing the ONERR, are protected by the ONERR, until the ONERR OFF instruction is given.

EXAMPLE

```
PROC div0:
  ONERR errHand
  PRINT 1/0
  REM cause divide by zero error -8
  RETURN    REM don’t get to this line
errHand::
  ONERR OFF
  PRINT “Error:”;err,err$(err)
  IF ERR=-8
    REM divide by zero error = -8
    PRINT “Division by zero is illegal”
  ENDIF
  GET
ENDP
```

Statements protected by ONERR

If an error occurs in the lines between `ONERR errHand` and `ONERR OFF`, the program jumps to the label `errHand::` where a message is displayed.

Always cancel ONERR with ONERR OFF immediately after the label.

WHEN TO USE ONERR OFF

You could protect the whole of your program with a single ONERR. However, it’s often easier to manage a set of procedures which each have their own ONERR...ONERR OFF handlers, each covering their own procedure. Secondly, an endless loop may occur if all errors feed back to the same single label.

For example, the diagram below shows how an error handler is left active by mistake. Two completely different errors cause a jump to the same label, and cause an inappropriate explanatory message to be displayed. In this example an endless loop is created because `next:` is called repeatedly:

```
PROC first:
  ONERR label
  a=log(-1)
  ...
  label::
  PRINT “Log error”
  next:
ENDP

PROC next:
  PRINT 2/0
  ONERR OFF
ENDP
```

MULTIPLE ONERRS

You can have more than one ONERR in a procedure, but only the most recent ONERR is active. Any errors cause a jump to the label for the most recent ONERR.

ONERR OFF disables *all* ONERRs **in the current procedure**. If there are ONERRs in **other** procedures above this procedure (*calling procedures*) these ONERRs are not disabled.

TRAP AND ONERR

TRAP has priority over ONERR. In other words, an error from a command used with TRAP will not cause a jump to the error handler specified with ONERR.

RAISE

The RAISE command generates an error, in the same way that OPL raises errors whenever it meets certain conditions which it recognises as unacceptable (for example, when invalid arguments are passed to a function). Once an error has been raised, either by OPL itself or by the RAISE command, the error-handling mechanism currently in use takes effect - the program will stop and report a message, or if you've used ONERR the program will jump to the ONERR label.

There are two reasons for using RAISE:

- You may want to mimic OPL's error conditions in your own procedures. For example, if you create a new procedure which performs a calculation and returns a value, you may want to RAISE an 'Overflow' or 'Divide by zero' error if unsuitable numbers are passed as parameters.
  In this case, you would RAISE one of the standard error numbers. You could handle this yourself with ONERR, or let OPL handle it in the normal way.

- OPL raises only a limited range of errors for general use, and you may want to raise new kinds of error codes specific to your program or particular circumstances.
  In this case, you would RAISE a new error number. With ONERR on, RAISE would go to the ONERR label, where you would have code to interpret your new error numbers. You could then display appropriate messages.
  You can use any positive number (from 0 to 127) as a new error code. Do not use any of the numbers in the list that follows.

You may also find RAISE useful for testing your error handling.

EXAMPLE

```
PROC main:
  REM calling procedure
  PRINT myfunc:(0.0)                   REM will raise error -2
ENDP


PROC myfunc:(x)
  LOCAL s
  REM returns 1/sqr(x)
  s=SQR(x)
  IF s=0
    RAISE -2
    REM 'Invalid arguments'
    REM avoids 'divide by zero'
  ENDIF
  RETURN (1/s)
ENDP
```

This uses RAISE to raise the 'Invalid arguments' error not the 'Divide by zero' error, since the former is the more appropriate message.

❺ TRAP RAISE ERR%

`TRAP RAISE err%` can be used to clear the TRAP flag and sets ERR value to `err%`. For example, using `err%=0` will clear ERR.

ERROR MESSAGES

These are the numbers of the errors which OPL can raise, and the message associated with them:

Number	*Message*
-1	General failure
-2	Invalid arguments
-3	O/S error
-4	Service not supported
-5	Underflow (number too small)
-6	Overflow (number too large)
-7	Out of range
-8	Divide by zero
-9	In use (e.g. serial port being used by another program)
-10	No system memory
-11	Segment table full
-12	Semaphore table full
-13	Process table full/Too many processes
-14	Resource already open
-15	Resource not open
-16	Invalid image/device file
-17	No receiver
-18	Device table full
-19	File system not found (e.g. if you unplug cable to PC)
-20	Failed to start
-21	Font not loaded

-22	Too wide (dialogs)
-23	Too many items (dialogs)
-24	Batteries too low for digital audio
-25	Batteries too low to write to Flash

FILE AND DEVICE ERRORS

-32	File already exists
-33	File does not exist
-34	Write failed
-35	Read failed
-36	End of file (when you try to read past end of file)
-37	Disk full
-38	Invalid name
-39	Access denied (e.g. to a protected file on PC)
-40	File or device in use
-41	Device does not exist
-42	Directory does not exist
-43	Record too large
-44	Read only file
-45	Invalid I/O request
-46	I/O operation pending
-47	Invalid volume (corrupt disk)
-48	I/O cancelled
-50	Disconnected
-51	Connected
-52	Too many retries
-53	Line failure
-54	Inactivity timeout
-55	Incorrect parity
-56	Serial frame (usually because Baud setting is wrong)
-57	Serial overrun (usually because Handshaking is wrong)
-58	Cannot connect to remote modem
-59	Remote modem busy
-60	No answer from remote modem

-61	Number is black listed (you may try a number only a certain number of times; wait a while and try again)
-62	Not ready
-63	Unknown media (corrupt SSD)
-64	Root directory full (on any device, the root directory has a maximum amount of memory allocated to it)
-65	Write protected
-66	File is corrupt (Media is corrupt on Series 3c)
-67	User abandoned
-68	Erase pack failure
-69	Wrong file type

TRANSLATOR ERRORS

-70	Missing “
-71	String too long
-72	Unexpected name
-73	Name too long
-74	Logical device must be A-Z (A-D on Series 5)
-75	Bad field name
-76	Bad number
-77	Syntax error
-78	Illegal character
-79	Function argument error
-80	Type mismatch
-81	Missing label
-82	Duplicate name
-83	Declaration error
-84	Bad array size
-85	Structure fault
-86	Missing endp
-87	Syntax Error
-88	Mismatched (or)
-89	Bad field list
-90	Too complex
-91	Missing ,

-92	Variables too large
-93	Bad assignment
-94	Bad array index
-95	Inconsistent procedure arguments

OPL SPECIFIC ERRORS

-96	Illegal Opcode (corrupt module translate again)
-97	Wrong number of arguments (to a function or parameters to a procedure)
-98	Undefined externals (a variable has been encountered which hasn’t been declared)
-99	Procedure not found
-100	Field not found
-101	File already open
-102	File not open
-103	Record too big (data file contains record too big for OPL)
-104	Module already loaded (when trying to LOADM)
-105	Maximum modules loaded (when trying to LOADM)
-106	Module does not exist (when trying to LOADM)
-107	Incompatible translator version (OPL file needs retranslation)
-108	Module not loaded (when trying to UNLOADM)
-109	Bad file type (data file header wrong or corrupt)
-110	Type violation (passing wrong type to parameter)
-111	Subscript or dimension error (out of range in array)
-112	String too long
-113	Device already open (when trying to LOPEN)
-114	Escape key pressed
-115	Incompatible runtime version
-116	ODB file(s) not closed
-117	Maximum drawables open (maximum 8 windows and/or bitmaps allowed)
-118	Drawable not open
-119	Invalid Window (window operation attempted on a bitmap)
-120	Screen access denied (when run from Calculator)

➎ SERIES 5 SPECIFIC ERRORS

-121	OPX not found
-122	Incompatible OPX version
-123	OPX procedure not found
-124	STOP used in callback from OPX
-125	Incompatible update mode
-126	In database transaction or started changing fields

➎ Constants for all error values are supplied in Const.oph. See the ‘Calling Procedures’ section of the ‘Basics.pdf’ document for details of how to use this file and Appendix E in the ‘Appends.pdf’ document for a listing of it.

INDEX