# 11 Errors and Error Recovery

The method that BASIC uses to generate an error is to execute a BRK instruction, which is followed by the error number and error message in the following format:

BRK instruction to generate the error
Single byte error number (ERR)
Error message (like ‘Mistake’)
A zero byte to terminate the message

The first section of this chapter describes the default BRK handler in BASIC, and what normally happens when an error is generated. The subsequent sections detail the errors which BASIC can generate, and any recovery from them (if possible), so that they can be intercepted in a similar way to the methods used in chapters 7 to 9.

## 11.1 The BASIC BRK handler

The Machine Operating System contains a BRK handler, which prints out the error message and restarts the current language. However, BASIC uses its own, so that it can allow errors to be trapped using the ‘ON ERROR’ statement.

BASIC keeps an ‘ON ERROR’ pointer in locations &16,&17 in page zero, which is normally set to point to the default error handler (in the ROM). This pointer tells the BASIC BRK handler the location of a set of BASIC statements which will deal with the error.

BASIC resets it to point to the default error handler every time it enters immediate mode (either when it initialises, or when it has finished executing a program), or whenever an ‘ON ERROR OFF’ statement is executed. When an ‘ON ERROR’ statement is executed, this pointer will be pointed at the start of the statements on the rest of the line, so that these will be executed when an error occurs.

The other advantage that BASIC gains by using its own error handler, is that the error messages can be tokenised. This means that keywords which appear in error messages (like the 'RENUMBER' in 'RENUMBER space') only take up 1 byte. The 'REPORT' statement, which is used to print out the error message, will convert these tokens into the correct keyword and print them out fully (this uses the 'ptoken' ROM routine).

**The action of the BASIC 1 BRK handler is:**

**1** Set up ERL. The base of PTRA will be at the start of the statement which caused the error, so a search is carried out through the program, keeping the line numbers, until the error line is found.

**2** Turn TRACE off

**3** Load the 'ON ERROR' pointer into PTRA, and start executing the statements making up the error handler by jumping to the 'Decode and execute command' entry. This executes the statements as if they had just been typed in as a command.

The default ERROR handler for BASIC1 reads:

```
  REPORT:IF ERL<>0 PRINT" at line ";ERL;
0 PRINT : END
```

The BASIC2 BRK handler has been changed slightly from the BASIC1 version; it will not allow commands to be part of the error handler. This means that you can't do 'ON ERROR LIST' with BASIC2; but it does also stop 'ON ERROR 10' (which may have been mistyped for 'ON ERROR GOTO 10') which corrupts the program, giving a 'Bad program' error.

**The action of the BASIC 2 BRK handler is:**

**1** Set up ERL.

**2** Turn TRACE off

**3** If the error number (ERR) is 0, the error is *fatal* (not to be trapped by an ON ERROR statement), so set the 'ON ERROR' pointer to point to the default error handler (i.e. perform 'ON ERROR OFF').

**4** Load the 'ON ERROR' pointer into PTRA, ready to execute it later.

**5** Clear the BASIC stacks, and restore the D AT A pointer. This is done in BASIC1 in the 'Decode and execute command' routine.

**6** Abandon the VDU queue (OSBYTE &DA). This is so that the first few characters of the error message to be printed will not be used as part of a multi-character VDU command (like VDU 19 or VDU 23).

**7** Acknowledge an ESCAPE condition. In BASIC 1, this is done by the 'Decode and execute command' routine.

**8** Set the OPT value to &FF (default)

**9** Execute the BASIC statements of the error handler at PTRA, as if they are part of a program.

The default ERROR handler for BASIC2 reads:

```
REPORT:IF ERL PRINT" at Line ";ERL:END ELSE PRINT:END
```

Note that the 'REPORT' statement is slightly different for each BASIC: in BASIC1 a VDU 6 command is sent before the error message is printed; in BASIC2 the error message is just printed. This means that if a program turns the screen off using a VDU 21 command, in BASIC1 any error messages will be printed on the screen, but in BASIC2 it will not.

## 11.2 Numbered errors

The errors detailed in this section have error numbers associated with them, and can be trapped by the BASIC ‘ON ERROR’ statement.

These can be recognised easily by a BRK handler, as &FD,&FE will point at the error number when the BRK handler is entered. Chapters 7 to 9 show how some of these errors can be intercepted.

## Error 1 − Out of range

This error is generated by the assembler when the address supplied to a branch instruction is too far away: it should be within −126 to +129 bytes of the branch instruction itself (i.e. within −128 to +127 of the instruction which would be executed if the branch did not take place).

This error (and the ‘No such variable’ error) will be suppressed if ‘OPT 0’ or ‘OPT 1’ is used in the assembler (i.e. bit 1 of OPT is zero). In this case, a displacement of 0 will be used for the branch, and assembly will be allowed to continue. However, due to the way in which the test for this bit is carried out, the ‘Out of range’ error will only be suppressed if the OPT setting used is either 0 or 1. In BASIC2, setting bit 2 of the OPT value enables remote assembly (see section 1.6.1); so if this facility is being used, this error will not be suppressed.

This error is recoverable, so that assembly can continue, although recovery should only be attempted if remote assembly is being used (in BASIC2).

**Error conditions:** (BASIC2 only)

Error number: 1 'Out of range'

Stack contents: RTI information 3 bytes

&28 current OPT value

A (current OPT value) DIV 2
X mnemonic number
Y undefined

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 1

**2** Bit 1 of the current OPT value (bit 0 of A) is 0

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the stack

**2** Set A to zero

**3** JMP to &86A5 (BASIC2 only)

This will use a zero dispacement for the branch, and assembly will continue.

# Error 2 − Byte

This error is generated by the assembler when a 2-byte value is used where only a single byte is allowed (the most significant 2 bytes of the 4-byte integer are ignored). The addressing modes which only allow a single byte are:

```
LDA #BB          / Immediate
LDA (BB),Y       / Post-indexed indirect
LDA (BB,X)       / Pre-indexed indirect
```

Recovery should not normally be attempted from this error, as potentially fatal mistakes in an assembler program may not be spotted; however it is possible to recover and just use the LSB. of the 2-byte word as the byte if required.

**Error conditions:**

Error number: 2 'Byte'

Stack contents: RTI information 3 bytes

IntA: value to be used in addressing mode

A MSB of the 16-bit value in IntA (non-zero)
X mnemonic number
Y undefined

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 2

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the stack

**2** JMP to &8669 (BASIC1) or &86A8 (BASIC2)

This will use only the LSB of the 2-byte value as the byte for the instruction, and assembly will continue.

# Error 3 − Index

This error is generated by the assembler if it finds an error in the syntax of any of the indexed addressing modes. The main causes of this are:

(a) The absence of an index in one of the indexed indirect modes. For example, 'LDA (&80)' will cause this error.

(b) A comma was found after the data, but no 'X' or 'Y' was found after the comma. For example, 'LDA &80,Z' will cause this error

(c) The wrong index register was used for this particular instruction. For example, 'LDY &80, Y' is not allowed.

**Error conditions:**

Error number: 3 'Index'

Stack contents: RTI information 3 bytes

IntA: value used in the istruction

A MSB of the 16-bit value in IntA (non-zero)
X mnemonic number
Y undefined

**This error is not recoverable.**

# Error 4 − Mistake

This error is generated by BASIC when an equals sign, '=', is not found after the first item of an assignment statement.

The usual cause of this is the mis-typing of a keyword at the start of a statement. When BASIC attempts to interpret the statement, it does not find a keyword, so it assumes that the item is a variable. When it doesn't find the '=' after it, it generates a 'Mistake' error. By trapping this error, it is possible to add in new statements or commands to the language (see chapter 7).

There are, in fact, 5 slightly different causes of a 'Mistake':

(a) A non-existent, but valid, variable name was found at the start of a statemen.t, but the first non-space character after it was not a '='.

(b) An existing variable was found at the start of a statement, but the first non-space character after it was not a '='. This looks the same as (a) above, but a slightly different action is taken by the BASIC interpreter.

(c) A 'LET' followed by a valid variable name was found, but no '=' was found after the variable.

(d) A pseudo-varible (like 'HIMEM') was found at the start of a statement, but no '=' was found after it.

(e) A 'FOR' was found, followed by a valid variable name, but no '=' was found after the variable.

Note that if an invalid symbol is found at the start of a statement, and not a valid variable name, then a 'Syntax error' (error 16) will be generated instead.

**Error conditions:**

| | | |
|---|---|---|
| Error number: | 4 | ‘Mistake’ |

| | | |
|---|---|---|
| Stack contents: | RTI information | 3 bytes |
| | Return address | 2 bytes |
| | (Return addr−(d) only | 2 bytes) |

PTRA: points to the character *after* the first non-space character of the line.

PTRB: points to the character *after* the character which was not an ‘=’.

| | |
|---|---|
| A | the character which was not an ‘=’ |
| X | undefined |
| Y | PTRB offset− 1 (i.e. points at char in A) |

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 4

**2** The name at the start of the statement can be recognised as a new command or statement keyword. To attempt this, a pointer could be constructed which points at the character one before PTRA, and recognition attempted from there. See section 7.4 for more on recognising keywords.

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the stack

**2** Pull the 2 bytes of return address from the stack

**3** If the first character of the statement was a pseudo-variable token (case (d)), then pull the other 2 bytes of return address from the stack. Normally a statement with a pseudo-variable at the start will not be recognised as a new command (unless one of the new keywords contains the token for it at the front), so this step does not need to be taken.

**4** The action of the new statement can now be performed. This should be a call to the ‘Check for end of statement’ routine at &9810 (BASIC1) or &9857 (BASIC2), to set up the pointers ready to continue with the next statement.

**5** Finally, after the action of the new statement has been completed, execution of the rest of the program can be continued with a JMP to &8B0C (BASIC1) or &8B9B (BASIC2). Alternatively, a restart of BASIC may be performed; this may be necessary if the program currently being run has been changed (by deleting a line, perhaps), as the syntax pointers may not point to the correct part of the program.

Note that pseudo-variables are not tokenised if followed by an alphanumeric character (see section 2.3. 1). This means that new commands may include these at the start of the new keyword (‘TIMER’,for example)

# Error 5 − Missing ,

This error is generated by BASIC if it fails to find a comma where one is required. Most of the functions which expect a comma separating their arguments will give this error if it is missing. For example, ‘A=POINT(X)’ will cause this error.

**Error conditions:**

| | | | |
|---|---|---|---|
| Error number: | 5 | Missing ‘,’ | |
| Stack contents: | RTI information<br>(undefined) | | 3 bytes |

| | |
|---|---|
| A | character which was not a comma |
| X | undefined |
| Y | undefined |

**This error is not recoverable**

# Error 6 − Type mismatch

This error is generated by BASIC if a string value was found where a number was expected, or a number was found where a string was expected. There are many ways that this error can be caused, including assigning a string to a number (and vice-versa) or giving the wrong type of argument to a function.

**Error conditions:**

| | | | |
|---|---|---|---|
| Error number: | 6 | Type mismatch | |
| Stack contents: | RTI information<br>(undefined) | | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 7 − No FN

This error is generated by BASIC when an equals sign is found at the start of a statement (signalling a return from a FN), but a FN is not currently being executed. The FN return routine only decides that a FN is in progress if the 6502 stack pointer is below &FC, and there is a FN token (&A4) as the first item on the stack, at &1FF. See section 5.3 for more on FNs and PROCs.

When inside a FN, the 6502 S register should be &F5 (the next available byte), and the contents of the stack should be:

| | |
|---|---|
| &1F6 | return addr to FN caller2 bytes |
| &1F8 | PTRB base MSB |
| &1F9 | PTRB base LSB |
| &1FA | PTRB offset |
| &1FB | number of parameters |
| &1FC | PTRA base MSB |
| &1FD | PTRA base LSB |
| &1FE | PTRA offset |
| &1FF Bottom: | &A4 (FN token) |

Note that the stack is ‘upside down’: the top of stack works downwards in page 1. Note also that the parameter values are stored on the BASIC STACK, rather than the 6502 stack.

Section 8.3 illustrates how this error can be used to throw away an overlayed FN when it exits, by substituting a different byte on the bottom of the 6502 stack when the FN is called.

**Error conditions:**

Error number: 7 ‘No FN’

| | | |
|---|---|---|
| Stack contents: | RTI information | 3 bytes |
| | undefined | |

PTRA: points to the character after the ‘=’

| | |
|---|---|
| A | undefined |
| X | copy of S (after TSX) |
| Y | undefined |

Recovery should only be attempted if:

**1** The error number at (&FD) is 7

**2** The condition of the stack due to which the error occurred can be determined.

To recover from the error:

**1** Pull the 3 bytes of RTI information from the stack.

**2** Evaluate the <numeric> or <string> following the ‘=’, and check that it is at the end of the statement.

**3** If we are in a FN (but it had been ‘hidden’ by changing the token at &1FF, for example) then executing an RTS will exit from the FN. The result of the FN should be in IntA, FPA, or StrA, with the result type stored in &27 (this is done automatically by the ‘Get <numeric> or <string>’ routine).

Note that the recovery performed in section 8.3 is more complex than this, as it also has to throw away the FN from the STACK.

# Error 8 − $ range

This error is generated by BASIC if an attempt is made to use the string indirection operator to assign or read from a string in page zero. For example, the statement 'PRINT $80' will cause this error.

It is possible to recover from this error to allow strings to be assigned in page zero, but it is not possible to read from a page zero string that has 'got through' the $ range check. If the BASIC 'Get value of variable' routine discovers that the address of an indirected string is only a single byte (i.e. in page zero), it will interpret it as 'CHR$' instead. Thus, if this error is being recovered, 'PRINT $&70' will behave the same as 'PRINT CHR$&70' (although '$&70=A$' will place A$ at location &70 onwards). This mechanism does not appear to have any possible use in BBC BASIC, as it should not allow the address of strings to be less than &100. However, the BASIC on the Acorn ATOM used '$' with a single-byte number instead of 'CHR$', so it could be left over from this.

**Error conditions:**

| | | | |
|---|---|---|---|
| Error number: | 8 | '$ range' | |
| Stack contents: | RTI information | | 3 bytes |
| | return address | | 2 bytes |

IntA: address of the defined-address string

| | |
|---|---|
| A | 0 |
| X | undefined |
| Y | undefined |

Recovery should only be attempted if:

**1** The error number at (&FD) is 7

To recover from the error:

**1** Pull the 3 bytes of RTI information from the stack.

**2** Set the type of of the variable to be a defined string, by storing &80 in location &2C (the 'type' byte of the variable descriptor block).

**3** Clear the Z flag (this may have been done already), and set the C flag: this indicates that a valid string variable has been found (see 'Find variable' in section 10.5).

**4** Execute an RTS instruction, to return to the code which called the 'Find variable routine'.

## Error 9 − Missing "

This error is generated by BASIC if the end of the line is found before the closing quote mark of a string. Anything which uses quoted strings (i.e. READ, INPUT, and the 'Get <stringfactor>' routine) can cause this error.

**Error conditions:**

| | | | |
|---|---|---|---|
| Error number: | 9 | Missing " | |
| Stack contents: | RTI information<br>undefined | | 3 bytes |

| | |
|---|---|
| A | &0D |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 10 − Bad DIM

This error is generated by BASIC if an error is encountered in a 'DIM' statement. The possible causes of this are:

(a) An attempt is made to re-dimension an array which already exists

(b) One of the dimensions of the array is either negative, or greater than &3FFF

(c) The total number of bytes required by the array is greater than &FFFF

(d) The size given to a 'reserve bytes' DIM is either less than −1, or greater than &FFFE

(e) An invalid variable name is found as the DIM subject

See also error 11 − 'DIM space'.

**Error conditions:**

Error number: 10 'Bad DIM'

Stack contents: RTI information 3 bytes

A undefined
X undefined
Y undefined

**This error is not recoverable**

# Error 11 − DIM space

This error is generated by BASIC if there is not enough memory for the space required by a ‘DIM’ statement. This can be caused by:

(a) The new value of the HEAP pointer calculated for an array would be above the BASIC STACK, or would have ‘wrapped round’ the memory map

(a) The new value of the HEAP pointer calculated for a ‘reserve bytes’ DIM would be above the BASIC STACK; no test for wrap-round is made (so ‘DIM A% &FFFE’ will move the HEAP pointer down by 1 byte).

If the DIM statement runs out of memory while it is allocating space for the name of the array on the HEAP, then a ‘No room’ error will be produced instead.

This error can only be recovered if more space can be allocated somehow (by forcing a MODE change and shifting the STACK, perhaps). The two possible causes of this error, (a) and (b), must be recovered differently.

**Error conditions:**

Error number: 11 ‘DIM space’

Stack contents: RTI information 3 bytes

&37,&38 If (a): copy of old HEAP pointer in &2,&3 If (b): undefined (probably lower than (a))

HEAP: If (a): points at ‘offset’ byte of array header
If (b): old value

A undefined
X MSB of new HEAP pointer
Y LSB of new HEAP pointer
C set

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 11 (&B)

**2** The new HEAP pointer (in A,Y) is above the BASIC STACK pointer. If it is not, the HEAP pointer has wrapped round over the top of the memory, and recovery should be aborted.

**3** The BASIC STACK can be shifted up out of the way, so that there is enough room for the new HEAP.

**4** The STACK has not already been corrupted by the array header information. In case (a), the 'offset' byte pointed to by the old HEAP pointer gives the number of bytes already written on to the HEAP; if these would be above STACK, then the STACK has been corrupted. In case (b) there is no header information.

To recover from the error:

**1** Pull the 3 bytes of RTI information from the 6502 stack.

**2** Shift the BASIC STACK so that the STACK pointer is above the required new HEAP pointer (moving the HEAP would be more tricky, due to all the pointers which point into it).

**3** Test if the pointer in locations &37 and &38 is equal to the pointer in locations &2 and &3: if it is, then the error is due to (a); otherwise it is due to (b).

**4** If the error is due to (a), execute a JMP to &91A0 (BASIC1) or &91EB (BASIC2); if it was due to (b), execute a JMP to &90B5 (BASIC1) or &9108 (BASIC2).

The new HEAP value will be set, and the DIM statement will continue (the DIM'd area will also be cleared if it is for an array).

# Error 12 − Not LOCAL

This error is generated by the BASIC 'LOCAL' statement if a FN or PROC is not currently being executed.

BASIC decides that a FN or PROC is not in progress, if the 6502 stack pointer is &FC or above. See section 5.3 for more on PROCs and FNs.

**Error conditions**

Error number: 12 'Not LOCAL'

Stack contents: RTI information 3 bytes

A undefined
X copy of S (by 'TSX')
Y undefined

**This error is not recoverable.**

# Error 13 − No PROC

This error is generated by BASIC when an 'ENDPROC' statement is found, but a PROC is not currently being executed. The ENDPROC handler only decides that a PROC is in progress if the 6502 stack pointer is below &FC, and there is a PROC token (&F2) as the first item on the stack, at &1FF. See section 5.3 for more on FNs and PROCs.

When inside a PROC, the 6502 S register should be &F5 (the next available byte), and the contents of the stack should be: 'Not LOCAL'

| | | |
|---|---|---|
| &1F6 | return addr to PROC caller | 2 bytes |
| &1F8 | PTRB base MSB | |
| &1F9 | PTRB base LSB | |
| &1FA | PTRB offset | |
| &1FB | number of parameters | |
| &1FC | PTRA base MSB | |
| &1FD | PTRA base LSB | |
| &1FE | PTRA offset | |
| &1FF Bottom: | &F2 (PROC token) | |

Note that the stack is 'upside down': the 'top of stack' works downwards in page 1. Note also that the old parameter values are stored on the BASIC STACK, rather than the 6502 stack.

Section 8.3 illustrates interception of this error to remove an overlayed PROC from the STACK when it exits, by changing the token on the bottom of the stack when it is called.

**Error conditions:**

| | | |
|---|---|---|
| Error number: | 13 | 'No PROC' undefined |
| Stack contents: | RTI information | 3 bytes |
| | undefined | |

PTRA: points to the character after the 'ENDPROC'

| | |
|---|---|
| A | undefined |
| X | copy of S (after TSX) |
| Y | undefined |

Recovery should only be attempted if:

**1** The error number at (&FD) is 13

**2** The condition of the stack which caused the error can be determined.

To recover from the error:

**1** Pull the 3 bytes of RTI information from the stack.

**2** Call the routine to 'Check end of statement at PTRA', at &9810 in BASIC1 or &9857 in BASIC2.

**3** If we are in a PROC (but it had been 'hidden' by changing the token at &1FF, for example), executing an RTS will exit from the PROC. This could be done by JMPing to the 'Check end of statement' routine instead.

# Error 14 − Array

This error is generated by the BASIC ‘Find variable’ routine. It will be caused either if an array name is referenced which has not already been dimensioned; or if the array referenced has fewer dimensions than the one in the original DIM statement (if it has more than the one in the DIM statement, a ‘Missing )’ error will be generated).

Error conditions

Error number: 14 ‘Array’

Stack contents: RTI information 3 bytes
undefined

A undefined
X undefined
Y undefined

This error is not recoverable.

# Error 15 − Subscript

This error is generated by the BASIC ‘Find variable’ routine, if the subscript which is used with an array is out of range. This can be caused if the subscript is negative, or if it is larger than the subscript which the array was DIM’d with.

**Error conditions**

Error number: 15 ‘Subscript’

Stack contents: RTI information 3 bytes
undefined

A undefined
X undefined
Y undefined

**This error is not recoverable.**

# Error 16 − Syntax error

This error is generated by the BASIC ‘Check for end of statement’ routine if the end of a statement was not found. It can also be caused if the first character of the statement is not a statement token, a variable name, or a special symbol (like ‘*’, ‘=’, or ‘[’); as BASIC will assume that it is dealing with an empty statement. For example, ‘COUNT’ at the start of a statement will generate a ‘Syntax error’. It will also be caused if an invalid variable name was found after a ‘LET’.

In BASIC1, this error can also be caused if the ‘#’ is missing after a statement or function which expects a file handle. BASIC2 has the new error ‘Missing #’ (error 45) for this condition.

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 16 | ‘Syntax error’ | |
| Stack contents: | RTI information<br>undefined | | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 17 − Escape

This error is generated by the BASIC ‘Check for end of statement’ routine (or the last part of it, which tests the ESCAPE flag in &FF) if an ESCAPE condition is active (i.e. the ESCAPE key has been pressed).

If this error is to be recovered from (ignored), then the ESCAPE condition should be acknowledged with a call to OSBYTE &7E before continuing (or it could be just cleared by OSBYTE &7C). If this is not done, then the escape condition will still be active on return to the BASIC interpreter; and it will generate this error again at its earliest opportunity.

A better way of ‘recovering’ from this error is to disable the ESCAPE key, to prevent the error from being generated in the first place.

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 17 | ‘Escape’ | |
| Stack contents: | RTI information | | 3 bytes |
| | return address | | 2 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 17

To recover from the error:

**1** Pull the 3 bytes of RTI information from the stack.

**2** Call OSBYTE &7E (or OSBYTE &7C) to acknowledge the ESCAPE condition.

**3** Execute an RTS

# Error 18 − Division by zero

This error is generated by the BASIC division routines if the divisor of the the attempted division is zero.

**Error conditions**

Error number: 18 'Division by zero'

Stack contents: RTI information
undefine

A undefined
X undefined
Y undefined

**This error is not recoverable.**

# Error 19 − 'String too long'

This error is generated by BASIC if an attempt is made to form a string longer than 255 characters. This can either be caused by concatenating 2 long strings together, or by the STRING$ function creating a string which is longer than 255 bytes. Note that only the LSB of the number sent to the STRING$ command is used; so STRING$(260, "*") will produce a string of 4 asterisks, but STRING$(130, "**") will produce an error.

**Error conditions**

Error number: 19 'String too long'

Stack contents: RTI information 3 bytes
undefined

A undefined
X undefined
Y undefined

**This error is not recoverable.**

# Error 20 − Too big

This error is generated by BASIC if an overflow occurs. This can be due to:

(a) A floating point number has overflowed after the end of a calculation. This is discovered by the 'Round and check for overflow' routine, before the floating point number is written out to memory (or to one of the temporary stores).

(b) An attempt was made to 'fix' (i.e. convert to integer) a number which would not fit into a 32-bit 2's complement integer.

Note that this error is not generated when two 32-bit integers are added or subtracted: if an overflow happens here, it will go undetected (try 'PRINT 2000000000+2000000000').

**Error conditions**

| | | |
|---|---|---|
| Error number: | 20 | 'Too big' |
| Stack contents: | RTI information<br>undefined | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 21 − -ve root

This error is generated by BASIC if the 'SOR' routine is given a negative argument. ASN and ACS can also generate this error (if the ABS value of their argument is greater than 1), because they are derived from ATN using the SQR routine:

```
ASN(X) = ATN(X/SQR(1-X*X))
ACS(X) = PI/2 - ASN(X)
```

**Error conditions**

| | | |
|---|---|---|
| Error number: | 21 | '-ve root' |
| Stack contents: | RTI information<br>undefined 21 22 '-ve root' | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 22 − Log range

This error is generated by BASIC if the 'LN' routine is given a negative or zero argument. LOG can also generate this error, as it is derived from LN:

```
LOG(X) = LN(X)/LN(10)
```

(BASIC stores 1/LN(10) as a constant, and uses a multiply to convert the LN to a LOG.)

**Error conditions**

| | | |
|---|---|---|
| Error number: | 22 | 'Log range' |
| Stack contents: | RTI information<br>undefined | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 23 − Accuracy lost

This error is generated by the BASIC SIN, COS, or TAN routines if the binary exponent of the floating point argument is &98 or greater. If it is, then at least 24 of the 32 bits in the mantissa make up the integer part of the number, leaving only 8 bits (or less) for the fractional part. This gives a resolution of worse than 1/256 (0.004) in the result from a SIN or COS (and all of this from the least significant byte).

The angle given to these trigonometric routines is reduced to the range 0 to PI/2 by subtracting a multiple of PI/2 from it. This does not introduce a significant amount of extra inaccuracy, as BASIC stores a more accurate (41 bits) − PI/2 as 2 separate numbers: a 'coarse' − PI/2, and an accurate adjustment to it.

**Error conditions**

Error number: 23 'Accuracy lost'

| Stack contents: | RTI information | 3 bytes |
|---|---|---|
| | return addr | 2 bytes |

FPA: number to find quadrant and offset from

| | |
|---|---|
| A | binary exponent of FPA |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 24 − Exp range

This error is generated by BASIC if an attempt is made to take the EXP of a number greater than or equal to 89.5. However, using EXP with an argument between 88 and 89.5 will produce a 'Too big' error. This error can also be generated by the exponentiation operator, as it is derived from the EXP and LN functions :

```
A^B = EXP(B*LN(A))
```

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 24 | 'Exp range' | |
| Stack contents: | RTI information | | 3 bytes |
| | undefined | | |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable**

# Error 25 − Bad MODE

This error is generated by the BASIC 'MODE' statement if there is not enough room for the new MODE above the HEAP or the TOP of the BASIC program, or if the BASIC ST ACK is not empty; i.e. if an attempt is made to change MODE inside a FN or a PROC. HIMEM and the STACK pointer are reset by a MODE change, and if this happened inside a FN or PROC, BASIC would probably crash on exit (like it does if you set 'HIMEM' inside a FN or PROC).

It is possible to recover from this error and perform the MODE change if the BASIC ST ACK can be preserved. This can be achieved by either shifting it to where the new HIMEM is, or (more simply) by leaving HIMEM where it was, and only allowing MODE changes which leave the bottom of screen memory higher than this. See section 9.1 for a 'Bad MODE' trap program.

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 25 | 'Bad MODE' | |
| Stack contents: | RTI information | | 3 bytes |
| | &16 MODE change character | | 1 byte |

PTRA: points at the statement delimiting character &2A

&2A Prospective MODE number (LSB of IntA)

A undefined
X undefined
Y undefined

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 25

**2** The bottom of the new MODE (found using OSBYTE &85) would not be below the top of the HEAP

**3** The bottom of the new MODE would not be below TOP

**4** The contents of the BASIC STACK can be preserved

**To recover from the error:**

**1** Check that the bottom of the new MODE would not be below the current HIMEM, and abort the MODE change if it would be.

**2** Pull the 3 bytes of RTI information from the stack. Pull the MODE change character from the 6502 stack, and print it (using OSWRCH)

**3** Get the new mode number from &2A, and send that to OSWRCH

**4** Continue with the execution of the BASIC statements by making a JMP to the 'Continue execution' routine at &8B0C (BASIC1) or &8B9B (BASIC2).

This will allow a MODE change inside a FN or PROC, although HIMEM must be brought down below the bottom of the lowest MODE first. It will always allow a MODE change to a smaller mode. It should also be possible to allow mode changes to a larger mode without previously allocating the space, but that would involve shifting the BASIC ST ACK bodily, and repointing the STACK pointer.

# Error 26 − No such variable

This error is generated by the BASIC 'Get <factor> or <string-factor>' routine if it tries to read the value of a variable which does 't exist. If the assembler is being used with an OPT value which has bit 1 cleared (i.e. OPT 0, 1, 4, 5), this error will be suppressed, and the current value of P% will be returned by the 'Get <factor>' routine instead. This error is suppressed if OPT 4 or 5 is used (unlike error 1 'Out of range').

By trapping this error it is possible to to add new functions to BASIC. Note, however, that the first character to be found after the name of the function must not be a '(', or BASIC will think that it is an array, and generate the 'Array' error instead (this is much more difficult to recover from). Bracketed expressions can be included after a new function, but the first '(' must be separated from the function name by a space.

**Error conditions**

| | | |
|---|---|---|
| Error number: | 26 | 'No such variable' |

| | | |
|---|---|---|
| Stack contents: | RTI information | 3 bytes |
| | return address | 2 bytes |

| | |
|---|---|
| PTRB: | points to the character after the end of the name |
| &2C: | type of the variable (if C=0) |
| (&37) | points 1 before the start of the name |
| &39 | length of the name (if C=0) |
| A | undefined |
| X | undefined |
| Y | undefined |
| C | 0=non-existent variable; 1=invalid name |

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 26

**2** The C flag is 0, signalling that a valid (but non-existent) variable name was found (unless you are trying to recognise a special symbol).

**3** The name can be matched with the name of a new function. The length of the function name should be the same as that in &39 (if it is not, PTRB will have to be adjusted to point after the function name). Note that the first character of the name can be read by the sequence:

```
LDY #1
LDA (&37),Y
```

**To recover from the error:**

**1** Ensure that the non-existent variable is actually a new function; if it is not, recovery should be aborted.

**2** Pull the 3 bytes of RTI information from the stack.

**3** Evaluate the function, and place the value in IntA, StrA, or FPA (depending on the type).

**4** Load A with a byte which signals the type of the value of the function. This should be the last action performed before returning, as it sets the Z and N flags which will be tested by the code which is returned to. The type bytes are:

| | |
|---|---|
| String: | &00 |
| Integer: | &40 |
| Real: | &FF |

**5** Execute an RTS

This will return the value of the new function to the code which called the 'Get <factor> or <string-factor>' routine.

# Error 27 − Missing )

This error is generated by BASIC if a closing bracket is expected, but none is found. This can either be caused by leaving off the ‘)’, or by sending too many arguments to a function, or too many dimensions to an array.

**Error conditions**

| | | |
|---|---|---|
| Error number: | 27 | ‘Missing )’ |
| Stack contents: | RTI information<br>undefined | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 28 − Bad HEX

This error is generated by BASIC if the first character after an '&' was not a hexadecimal digit (i.e. 0 to 9, or A to F).

It is possible to recover from this error (if, for example, you want an '&' by itself to mean 0)

**Error conditions**

Error number: 28 Bad HEX

Stack contents: RTI information
return address

IntA: 0

A 0
X Y
Y PTRB offset

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 28

To recover from the error:

**1** Pull the 3 bytes of RTI information from the stack.

**2** Load A with &40, to signal that the type of the result is an integer.

**3** Execute an RTS

This will return 0 to the code which called the 'Get <factor> or <string-factor>' routine, if no HEX character followed the '&'.

# Error 29 − No such FN/PROC

This error is generated by BASIC if an attempt is made to access a FN or PROC which is not defined inside the program. First, the FN/PROC handler tries to find it in the list on the HEAP; if it isn't found, it looks through the program for the definition; if it still does 't find it, this error is generated.

If this error is trapped, it is possible to overlay procedures and functions from disc, for example, and continue execution. Any routine which attempts to recover from this error should be very careful with the state of the 6502 stack, as the FN/PROC routine is in the middle of saving the information it needs to enable it to return properly at the end of the PROC or FN. See chapter 8 for more on overlaying FNs and PROCs.

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 29 | No such FN/PROC | |
| Stack contents: | RTI information | | 3 bytes |
| | PTRA offset | | 1 byte |
| | FN/PROC token (&A4/&F2) | | 1 byte |

(&37) points 1 before the calling PROC/FN token

A copy of &B (PTRA base LSB)
X undefined
Y 1

Recovery should only be attempted if:

**1** The error number at (&FD) is 29

**2** The FN or PROC can be overlayed (from disc, for example).

**3** The FN or PROC is of the correct type (the token is held in location &1FF)

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the stack.

**2** Save PTRA base on the stack, by pushing the contents of &B followed by the contents of &C.

**3** Load the FN or PROC to be overlayed, allocating space for it as necessary.

**4** Restart the FN/PROC handler, to execute the FN or PROC.

There are two major alternative ways to re-start the FN/PROC handler:

(a) Set PTRA base (in &B,&C) to point to the first byte of the program section just overlayed (this will be the &0D usually at PAGE). Then JMP to &B149 (BASIC1) or &B11A (BASIC2). This will cause the 'Look for FN/PROC in program' routine to search for the FN/PROC again, but this time starting from PTRA base, instead of PAGE. When the FN/PROC is found, it will be added to the list, and the main FN/PROC handler will be re-joined.

(b) Set PTRA base to point to the byte following the name of the defined PROC or FN in the overlayed section (this will be a '(' if any arguments are being used). Then JMP to &B223 (BASIC1) or &B1F4 (BASIC2). This directly rejoins the FN/PROC handler, without adding the name of the overlayed FN/PROC to the list.

Note that if (a) is being used, the same error may be generated again if the name is still not found; if (b) is being used, the name will not be tested (and does not even need to be in the file itself, as long as PTRA can still be set up to point to the character which would be after it).

# Error 30 − Bad call

This error is generated by BASIC if no valid name is found after a PROC or FN token. Note that there can be no spaces between the FN or PROC token, and the name.

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 30 | 'Bad call' | |
| Stack contents: | RTI information | | 3 bytes |
| | PTRA base MSB | | 1 byte |
| | PTRA base LSB | | 1 byte |
| | PTRA offset | | 1 byte |
| | FN/PROC token (&A4/&F2) | | 1 byte |

(&37) points 1 before the PROC/FN token

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | 2 |

**This error is not recoverable.**

# Error 31 − Arguments

This error is generated by BASIC if the number of parameters passed to a FN or PROC is not the same as in the definition of the FN or PROC. It can also be caused if the types of the parameters do not match (i.e. a string being passed where a number is expected).

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 31 | 'Arguments' | |
| Stack contents: | RTI information | | 3 bytes |
| | PTRA offset | | 1 byte |
| | FN/PROC token (&A4/&F2) | | 1 byte |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 32 − No FOR

This error is generated by the BASIC ‘NEXT’ statement if there is nothing on the FOR stack. See section 5.6 for more on FOR...NEXT loops.

**Error conditions**

Error number: 32 ‘No FOR’

Stack contents: RTI information 3 bytes

A undefined
X 0
Y undefined

**This error is not recoverable.**

# Error 33 − Can’t match FOR

This error is generated by the BASIC ‘NEXT’ statement if the loop variable was specified (as in ‘NEXT I’), but it could not find a FOR loop using that variable on the FOR stack. This error will not be generated if the variable specified in the ‘NEXT’ statement does not exist: a ‘Syntax error’ (error 16) will be generated instead.

**Error conditions**

Error number: 33 ‘Can’t match FOR’

Stack contents: RTI information 3 bytes

FOR stack: empty

A 0
X 0
Y undefined

**This error is not recoverable.**

# Error 34 − FOR variable

This error is generated by the BASIC ‘FOR’ statement if there is no valid numeric variable after the FOR (i.e. either it is invalid, or it is a string variable). This variable can be an indirected variable (like ‘!X’), although single byte variables should not be used, as NEXT does not deal with them properly.

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 34 | ‘FOR variable’ | |
| Stack contents: | RTI information | | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

# Error 35 − Too many FORs

This error is generated by the BASIC ‘FOR’ statement if there are already 10 ‘FOR’ loops on the FOR stack (see section 5.6).

It is possible to recover from this error, to extend the FOR stack into the REPEAT stack area, for example. This should not normally be attempted, as any REPEAT statement will corrupt an extended FOR stack.

**Error conditions**

Error number: 35 'Too many FORs'

Stack contents RTI information 3 bytes

FOR stack: full
&26 &96 (or greater if already recovered)

Initial value already assigned to loop variable

A undefined
X undefined
Y copy of FOR stack pointer in &26

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 35

**2** No REPEATs will be used in the program (or GOSUBs if the GOSUB stack area will be used as well).

**3** The FOR stack pointer (in &26 and Y) is less than &BE (this gives room for 3 more entries). If the GOSUB stack area is to be used as well, the FOR stack pointer should be less than &F2 (this gives a total of 17 entries in the FOR stack).

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the 6502 stack

**2** JMP to &B7F5 (BASIC1) or &B7DA (BASIC2)

This will continue with the FOR statement, as though the FOR stack had not overflowed. The Y register should not be altered by the recovery routine, as it is used on return to the FOR handler.

# Error 36 − no TO

This error is generated by the BASIC ‘FOR’ statement if the first non-space character after the initial value that the loop variable is to be set to, is not a ‘TO’ token. The initial value must be a <numeric>.

Recovery from this error is not easily possible, although it could be trapped to allow ‘FOR lists’; i.e. a line of the form:

```
FOR I=1,3 TO 5,10
```

which would step through the loop with I taking the values 1,3,4,5, and 10. If this was to be implemented, a new ‘NEXT’ statement would have to be used for this type of ‘FOR’ (possibly trapped from the ‘Mistake’ error), as the normal NEXT would not handle it.

**Error conditions**

Error number: 36 ‘No TO’

Stack contents: RTI information 3 bytes

Initial value already assigned to loop variable

PTRB: points to the character after that in A

&26 FOR stack pointer

(&37) ddress of the loop variable
&39 type of the loop variable

A character after the initial value (not ‘TO’)
X undefined
Y copy of FOR stack pointer in &26

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 36

**2** An alternative form of the 'FOR' statement can be used. Another NEXT should be used for this structure ('ENDFOR'?), to handle the next value to be assigned to the loop variable.

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the 6502 stack

**2** Handle the new FOR structure, either using the FOR stack, or by creating a different stack. The address and type of the loop variable (i.e. its variable descriptor block) is already on the FOR stack.

**3** If a FOR list is being used, the ENDFOR will have to look at the next item on the list; thus the current value of PTRB should be saved for it.

**4** If the whole of the FOR list is to be parsed before the loop is entered, the 'Check for end of statement' routine at &9810 (BASIC1) or &9857 (BASIC2) should be called after the FOR list has been checked. Then the statements in the loop can be started with a JMP to the 'Continue execution' routine at &8B0C (BASIC1) or &8B9B (BASIC2).

**5** If the FOR list is not to be parsed until the ENDFOR tries to use it, execution can be continued with a JMP to the 'Skip rest of line, and continue' routine at &8AED (BASIC1) or &8B7D (BASIC2). This will continue execution on the next program line (alternatively, the new FOR routine could just search for a ':', and continue from there).

# Error 37 − Too many GOSUBs

This error is generated by the BASIC 'GOSUB' statement if there are already 26 GOSUBs on the GOSUB stack. See section 5.2 for more on GOSUBs.

Due to way that the GOSUB stack is stored (as 2 stacks, one after the other), it is not easily possible to recover this error and extend the stack in a similar manner to the FOR stack.

**Error conditions**

Error number: 37 'Too many GOSUBs'

Stack contents: RTI information 3 bytes

&25: &1A (i.e. GOSUB stack pointer = 26)

A undefined
X undefined
Y &1A (copy of location &25)

**This error is not recoverable.**

# Error 38 − No GOSUB

This error is generated by the BASIC 'RETURN' statement if the GOSUB stack is empty.

**Error conditions**

Error number: 38 'No GOSUB'

Stack contents: RTI information 3 bytes

&25: 0

A undefined
X undefined
Y 0 (copy of GOSUB stack pointer in &25)

**This error is not recoverable.**

# Error 39 − ON syntax

This error is generated by the BASIC ‘ON’ statement if the first non-space character following the <factor> after the ‘ON’ is not a ‘GOTO’ or a ‘GOSUB’ token. This may be caused if the <factor> is mis-formed, as in:

```
ON A#3 GOTO ...
```

**Error conditions**

| | | |
|---|---|---|
| Error number: | 39 | ‘ON syntax’ |
| PTRA: | points to the character after that in X | |
| A | undefined | |
| X | non-space character after the <factor> | |
| Y | undefined | |

**This error is not recoverable**

# Error 40 − ON range

This error is generated by the BASIC ‘ON’ statement if the controlling <factor> is either less than 1, or greater than the number of entries in the ‘GOTO’ or ‘GOSUB’ list.

This error can be avoided by using an ‘ELSE’ clause after the GOTO or GOSUB list (such as ‘ON I GOTO 20,30 ELSE END’), but in BASIC1 the ‘GOTO’ or ‘GOSUB’ token is left on the 6502 stack if the ELSE clause is executed. If this ELSE clause is executed inside a FN or PROC, the return from this FN or PROC will fail, as the return address will no longer be on the top of the stack. In BASIC2, this has been rectified, and the ELSE clause works correctly.

**Error conditions**

| | | |
|---|---|---|
| Error number: | 40 | ‘ON range’ |
| Stack contents: | RTI information | 3 |
| | (token − BASIC1 only 1 byte | 1 |

PTRA: points to the last part of the statement handled

| | |
|---|---|
| A | &0D |
| X | undefined |
| Y | offset from PTRA base to point end o line |

**This error is not recoverable.**

# Error 41 − No such line

This error is generated by the BASIC ‘Evaluate and find line number’ routine if the line number it is given does not exist. This routine is used by GOTO, GOSUB, and RESTORE, so all of these can generate this error if given a non-existent line number.

This error could be recovered from if, for example, some sort of program overlaying mechanism is being used.

**Error conditions**

| | | |
|---|---|---|
| Error number: | 41 | ‘No such line’ |
| Stack contents: | RTI information | 3 bytes |
| | return address | 2 bytes |

&2A,&2B: line number which was not found

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |
| C | 1 |

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 41

**2** The line can be looked for in an alternative area (for example, in an overlayed program section)

**To recover from the error:**

**1** Pull the 3 bytes of  from the stack

**2** Find the line in the alternative program section, and set the pointer at &3D,&3E to point 1 before the first byte of text of the line (i.e. to point to the length byte of the line). Care should be taken not to generate this error again, unless some flag is used to signal that this overlay has already been tried. If the line number is not found in the new section, and the error is generated again, this recovery routine will be called repeatedly, and the machine will 'hang up'.

**3** When the line has been found, clear the carry flag (to signal that the line has been found), and execute an RTS.

This will return to the code which called the 'Evaluate and find line number' routine, which will then continue.

# Error 42 − Out of DATA

This error is generated by the BASIC 'Find next DATA item' routine of the 'READ' statement if all of the DATA items in the program have been read.

This error could be recovered, either if some sort of overlaying mechanism is being used, or perhaps by forcing a 'RESTORE' on an 'Out of DATA' error.

**Error conditins**

| | | | |
|---|---|---|---|
| Error number: | 42 | 'Out of DATA' | |
| Stack contents: | RTI information | | 3 bytes |
| | return address | | 2 bytes |

&1C,&1D: point after the last DATA item read

A undefined
X undefined
Y undefined

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 42

**2** Either a RESTORE will be forced, or the DATA will be found in an alternative area

**3** The DATA pointer in &1C,&1D does not still point at PAGE. If it does, there is no DATA in the program at all, and so forcing a RESTORE would have no effect.

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the stack

**2** Set PTRB to point to the area where the DATA will be read from. This will be PAGE to force a RESTORE to the start of the program, or it will point to the new area if an overlay has been loaded.

**3** Execute a JMP to &BB7 A (BASIC1) or &BB60 (BASIC2). This re-starts the 'Find next DATA item' routine looking from PTRB. If PTRB points at a comma or a 'DATA' token when the routine is re-started, then that routine will return to the READ statement handler, with PTRB pointing at the following DATA item.

Care should be taken that this recovery routine is not called again due to a failure to find any DATA in the new area. The DATA pointer could be used as a flag for this, by setting it to PAGE inside this recovery routine. If no DATA is found on return to the READ handler, then this error will be generated again, but with the DATA pointer still set to PAGE.

# Error 43 − No REPEAT

This error is generated by the BASIC ‘UNTIL’ statement if the REPEAT stack is empty.

**Error conditions**

Error number: 43 ‘No REPEAT’

Stack contents: RTI information

PTRA: points to the end of the UNTIL statement

&24: 0 (REPEAT stack empty)

A undefined
X 0 (copy of REPEAT stack pointer in &24)
Y undefined

**This error is not recoverable.**

# Error 44 − Too many REPEATS

This error is generated by the BASIC ‘REPEAT’ statement if the REPEAT stack already contains 20 entries.

The REPEAT stack cannot be extended like the FOR stack, as it saves the MSB and LSB of the pointer in 2 stacks, 1 after the other. See section 5.5 for more on REPEAT loops.

**Error conditions**

Error number: 44 ‘Too many REPEATs’

Stack contents: RTI information 3 bytes

&24 &14 (REPEAT stack full with 20 entries)

A undefined
X &14 (copy of REPEAT stack pointer in &24)
Y undefined

**This error is not recoverable.**

# Error 45 − Missing #

This error is generated by the BASIC file handling routines if the file handle given to a BPUT, BGET, PTR, or EXT is not preceeded by a '#'. This error is only generated by BASIC2; BASIC1 will generate a 'Syntax error' (error 16) instead.

**Error conditions** (BASIC2 only)

Error number: 45 'Missing #'

Stack contents: RTI information

| | |
|---|---|
| A | character not a '#' |
| X | undefined |
| Y | undefined |

**This error is not recoverable.**

## 11.3 Fatal errors

These errors cannot be trapped by the 'ON ERROR' statement. Some of them are just messages, with a JMP to immediate mode after the message has been printed; others have error number 0, which cannot be trapped (in BASIC 2).

Some of the errors in this section can still be intercepted by a BRK handler, although those that can be intercepted, will all have error number 0. This means that the error message string following the error number byte must be tested if the error is to be identified correctly.

## Bad program

This message is printed if the current program in memory has been corrupted when a check is made. After the message has been printed, a JMP is made to restart BASIC in immediate mode: this cannot be trapped.

If the program is OK, the 'Bad program' check routine resets TOP to the top of the program, and returns to the calling routine. The check is made when:

(a) A new program has been loaded (either by 'LOAD' or 'CHAIN').

(b) An 'OLD' statement has been executed

(c) A 'LIST' statement is about to be executed

(d) A 'RENUMBER' command is about to be executed

(e) An 'END' statement is executed. As an END statement is executed at the end of the default BASIC ERROR handler, this check will also be made whenever an error occurs.

See section 9.2 for a 'Bad program' salvage routine.

# Failed at xxx

This message is printed by the 'RENUMBER' command if it finds any references to non-existent line numbers. This error cannot be trapped, but it will not abort the RENUMBERing of the program; it will just produce a list of the lines on which it found unresolved line number references.

# Line space

This error is generated by the 'Insert line in program' routine if there is not enough room to insert the line into the program (i.e. the length of the line is longer than the gap between TOP and HIMEM).

This error, although 'fatal' to BASIC, could be recovered from if more memory could be allocated (by forcing a MODE change, perhaps).

**Error conditions**

| | | | |
|---|---|---|---|
| Error number: | 0 | 'Line space' | |
| Stack contents: | RTI information | | 3 bytes |
| | return address | | 2 bytes |

| | |
|---|---|
| IntA: | line number of line to be inserted |
| &700− | line to be inserted (keyboard buffer) |
| &3B,&3C | points to the first character to be inserted |
| A | undefined |
| X | undefined |
| Y | undefined |

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 0, followed by the string ‘Line space’, terminated by a zero byte.

**2** HIMEM can be moved up from its present position, perhaps by a MODE change. If it can’t be moved, then recovery should be aborted.

To recover from the error:

**1** Pull the 3 bytes of RTI information from the stack

**2** Change MODE to shift HIMEM to a higher value

**3** Execute a JMP to &BC96 (BASIC1 or BASIC2 − the addresses coincide).

This will re-enter the routine to insert the line in the program. Note that if this recovery is attempted *without* moving HIMEM up, then this error will just be generated again, and the machine will ‘hang up’.

# No room

This error is generated by BASIC if an attempt is made to extend the HEAP above the STACK, or extend the STACK below the HEAP. In BASIC1, this is a message which is printed before a JMP to immediate mode (so it gives no line number); but in BASIC2 it is an error with error number 0.

In BASIC2 it is possible to trap this error, and recover from it under certain circumstances (providing some more memory can be found from somewhere); but in BASIC1 it does not go through the BRK handler, and so cannot be trapped.

The 'No room' error can be caused in one of 3 ways:

(a) An attempt was made to allocate space for a new *variable information block* on top of the HEAP. If this is the case, then the error is not recoverable, because the 'Allocate new information block' routine clears the space for the block before checking for a clash with the STACK: thus the contents of the STACK will be corrupted.

(b) An attempt was made to allocate space for a dynamic string on the HEAP. This error is recoverable, as a clash with the STACK is tested for before the string is written into the new area.

(c) An attempt was made to allocate space on the BASIC STACK. This error is also recoverable, because a clash with the HEAP is tested for before the item to be pushed is written into the allocated area.

These 3 different causes of a 'No room' must be handled differently, as they require different return conditions, and in the case of (a), recovery should not be attempted at all.

**Error conditions** (BASIC2 only)

Error number: 0 'No room'

Stack contents: RTI information 3 bytes

If (a):

| | |
|---|---|
| A | 0 |
| X | 0 |
| Y | 1 |
| C | 1 |

If (b):

| | |
|---|---|
| A | undefined |
| X | MSB of attempted new HEAP |
| Y | LSB of attempted new HEAP |
| C | 1 |

If (c):

| | |
|---|---|
| A | LSB of attempted new STACK (copy of location &4) |
| X | undefined |
| Y | MSB of attempted new STACK (copy of location &5) |
| C | 0 |

**Recovery should only be attempted if:**

**1** The error number at (&FD) is 0, followed by the string 'No room', terminated by a zero byte.

**2** The error was not caused by case (a). If the carry flag was clear when the BRK occurred (this should be tested from the RTI information on the 6502 stack) then it was due to case (c), and recovery is possible. Otherwise, if the X register is non-zero it was due to case (b), and recovery is also possible. If the carry flag was set, and the X register is zero, it was due to case (a), and recovery should be aborted.

**To recover from the error:**

**1** Pull the 3 bytes of RTI information from the stack (the top byte was the 6502 status word when the BRK occurred, and the carry can be checked from there)

**2** Allocate some more memory. This could either be done by forcing a mode change, or perhaps by throwing away any overlayed program sections which have been placed between HIMEM and the bottom of the screem. Both of these will involve shifting the STACK bodily, and pointing the STACK pointer (in &4,&5) at the bottom of the new ST ACK.

**3** Check that the HEAP/STACK clash does not still exist: it may be that not enough memory could be cleared. If (c) is being dealt with, then the STACK and HEAP will be in the pointers already; but in case (b), the old HEAP pointer is in &2,&3 and the new one is in X (MSB) and Y (LSB).

**4** If (c) is being dealt with, then simply executing an RTS will return to the code that called the 'Check for STACK/ HEAP clash' routine.

**5** If (b) is being dealt with, then the 'Assign string' routine can be continued with a JMP to &8C6F (BASIC2 only). The new HEAP pointer must be in the X and Y registers as on entry (alternatively, if the new HEAP pointer is already set up by the recovery routine, a JMP can be made to &8C73 instead).

Trapping this routine, together with trapping the 'No such FN/ PROC' error (error 29), would give a very neat method of procedure and function overlaying. When a FN or PROC is not found in the program, the STACK can be shifted down and an overlay loaded from disc between HIMEM and the bottom of the screen; and when the computer runs out of memory and issues a 'No room' error, the overlay can be removed, and the STACK shifted up again.

# RENUMBER space

When the RENUMBER statement is used, it creates a list of the old line numbers above TOP so that it can match up the GOTO and GOSUB references after the lines have been renumbered. This error is generated if there is not enough room between the TOP of the program and HIMEM to fit this list.

**Error conditions**

Error number 0 'Renumber space'

Stack contents: RTI information

A undefined
X undefined
Y undefined

**This error is not recoverable**

# Silly

This error is generated by the AUTO or RENUMBER commands if the interval in their call is either 0 or greater than 255.

It is possible to recover from this error (if you really want to have all the lines in your program with the same line number).

**Error conditions**

Error number 0 'Silly'

| Stack contents: | RTI information | 3 bytes |
|---|---|---|
| | return address | 2 bytes |

IntA: AUTO/RENUMBER interval

A 0 if the interval = 0, non-zero if interval > 255
X undefined
Y undefined

**This error should only be recovered if:**

**1** The error number at (&FD) is 0, followed by the string 'Silly', terminated by a zero byte.

To recover from the error:

**1** Pull the 3 bytes of RTI information from the 6502 stack

**2** Execute a JMP to &8F28 (BASIC1) or &8F8B (BASIC2)

This will continue with the AUTO or RENUMBER command, ignoring any silly restrictions on the size of the interval.

## STOP at line xxx

This error is generated by the BASIC 'STOP' statement. In BASIC1, this is just a message which is printed before a JMP to immediate mode; but in BASIC2 it is an error with error number 0. The BASIC2 error message does not used the 'STOP' token (probably because it was converted from the BASIC1 message).

**Error conditions** (BASIC2 only)

| | | |
|---|---|---|
| Error number | 0 | 'STOP' |
| Stack contents: | RTI information | 3 bytes |

| | |
|---|---|
| A | undefined |
| X | undefined |
| Y | undefined |

**This error is not recoverable**