3 Program Control Mechanisms

Normally in a BASIC program, the statements are executed one after the other, working through the program. However, several statements are provided which allow this normal flow of control of the program to be changed, either by jumping to another part of the program, or by conditionally executing a series of statements.

BASIC keeps a text pointer, PTRA, which it uses to point to the statement currently being executed, in a similary way to the program counter (PC) in the 6502 (see section 2.2.5). Whenever any of these program control statements, like GOTO, change the flow of control of the program, this pointer is changed to point to the start of the new statement where execution of the program is to continue. When the interpreter continues, it will then start reading in from the statement pointed to by PTRA.

This section details the program control statements in BASIC, and describes the mechanisms that they use to operate.

5.1 GOTO

This is the simplest of the program control statements in BASIC. It just passes control from one part of the program to another.

The action of the BASIC GOTO statement is:

1 Get the line number or <numeric> following the GOSUB token, and set PTRA to point to the end of the statement.

2 Search the program to find a line with that line number; if it is not found, generate a ‘No such line’ error (error number 41).

3 If the line was found, then point the text pointer PTRA at the start of the first statement on that line. When the BASIC interpreter continues, it will execute statements from there onwards.

5.2 GOSUB...RETURN

The GOSUB statment is similar to the GOTO statement in that it passes control to another part of the program; but it also allows control to RETURN to the statement after the GOSUB statement when the subroutine has finished.

The GOSUB statement has to remember where to RETURN to after the end of the subroutine. A ‘GOSUB stack’ is used to hold the location of the statement following the GOSUB statement, so that the RETURN statement on the end of the subroutine can pass control back to that part of the program. The format of the GOSUB stack is:

&05CC+GSP	LSB of return address
&05E6+GSP	MSB of return address
&25	GOSUB stack pointer (GSP)

The action of the GOSUB statement is:

1 Get the line number or <numeric> following the GOTO token.

2 Search the program from the beginning to find a line with that line number; if it is not found, generate a ‘No such line’ error (error number 41).

3 If the GOSUB stack pointer is more than 25, there are already 26 return addresses (0 to 25) on the stack. In this case, generate a ‘Too many GOSUBs’ error (error number 37), to prevent the GOSUB stack from overflowing (it only has room for 26 entries).

4 If we get here, the GOSUB stack is not full, so push the base of PTRA, which now points to the end of the GOSUB statement, on to the the GOSUB stack. Increment the GOSUB stack pointer (GSP), ready for the next one.

5 Point the text pointer PTRA at the start of the first statement on the line found. When the BASIC interpreter continues, it will execute statements from there onwards.

When a RETURN statement is encountered, it has to retrieve the old value of PTRA, so that it can go back to the statement after the GOSUB which called it.

The action of the RETURN statement is:

1 If the GOSUB stack pointer is 0, the GOSUB stack is empty, and there is no address to return to. In this case, generate the ‘No GOSUB’ error (error number 38).

2 Pop the return address from the GOSUB stack, decrementing the GOSUB stack pointer to remove it. This return address is then put into PTRA. When the interpreter continues, it will execute statements from there onwards (i.e. starting with the statement after the GOSUB which called the subroutine).

5.3 PROCs and FNs

The ability to call PROCs and FNs is a very powerful feature of BBC BASIC, although as far as the interpreter is concerned it is just a more complex version of the GOSUB statement. With PROC and FN calls, not only does the return address have to be saved, so that control can be returned when the call is finished, but the values of parameters and local variables have to be saved so that they can be restored also.

Once a FN or PROChas been called, its name and location is added to a linked list on the BASIC HEAP, one list for FNs, and one for PROCs. This means that once a FN or PROC has been used, BASIC does not have to search through the whole of the program to find it again (like it does with the line numbers given to a GOTO or GOSUB statement). See section 3.1 for the format of these liked lists.

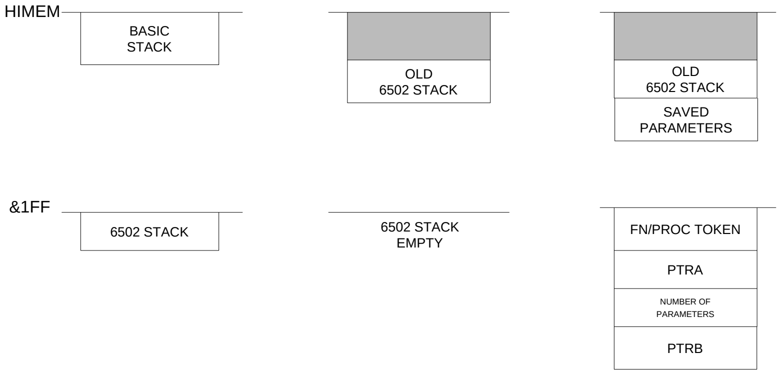
After the FN or POC has been found, any parameters which need to be passed are handled. In the description below *formal parameter* refers to the parameter used in the FN or PROC definition; and *actual parameter* refers to the parameter which is passed to it.

Although PROC is a statement and FN is a function (and hence returns a value), the mechanism which is used when they are called is very similar. To deal with both of them, there is a standard FN/PROC handler which is called by both the FN function and the PROC statement.

The PROC statement has to copy PTRA into PTRB before calling this handler, and then use PTRB (rather than PTRA) to check that it is at the end of the statement when the call has returned. The FN/PROC handler must not alter PTRA, because this is not used in the expression evaluator (and hence the FN function must not change it). The FN function does not need to do any of this (as PTRB will be set up correctly for it), and the FN/PROC handler returns directly to the code which called the FN when it has finished.

The action of the FNIPROC handler is:

1 Save the contents of the 6502 stack on the BASIC stack (with a byte to give the old 6502 stack pointer), and reset the 6502 stack pointer to &1FF. The 6502 stack works downwards in page 1, and the stack pointer points to the next available byte, so it is now empty (fig 5.1 (b)). The 6502 stack is not very big − only 256 bytes − and saving it in this manner allows deep recursion of FNs and PROCs without overflowing the small 6502 stack.



(a) (b) (c)

Figure 5.1 − FN/PROC stack use.

2 Save the FN or PROC token as the first item on the 6502 stack, at &1FF. The FN token is &A4, and the PROC token is &F2. This allows the ENDPROC or FN return statement (‘= ’) to check that it is inside the correct type of call before it exits.

3 Save PTRA on the 6502 stack.

4 Scan the name of the FN/PROC call. If there is not one immediately following the FN or PROC token, generate a ‘Bad call’ error (error number 30).

5 Search for the name of the FN or PROC in the list of already used calls. If it is found, don’t bother to look through the program for it.

6 If the FN or PROC was not in the list, look through the program from the beginning until a DEF FN or a DEF PROC is found with the correct type and name. This search uses PTRA to look through the program (which is why it was saved at stage 3). If it is found, add it to the list; otherwise, restore the base of PTRA from the 6502 stack (this will tell the error handler on which line the error occurred), and generate a ‘No such FN/PROC’ error.

7 Set PTRA to point to the location found by the search (or found in the list). This will point to the first character following the name after the DEF FN or DEF PROC. If there are any parameters, this character will be an opening bracket, ‘(’.

8 If there are any parameters in the definition, check that they match with those in the call. If they do, push the value and the *variable descriptor block* of each *formal* parameter on the BASIC STACK (i.e. the one in the definition), and assign the new value to it given by the value of the actual parameter in the call. Saving the value and *variable descriptor block* allows the formal parameters to be restored to their original values after the call has returned. If the parameters do not match, restore the base of PTRA from the 6502 stack (for the error handler), and generate an ‘Arguments’ error (error number 31).

9 Push the number of parameters on the 6502 stack, so that the correct number can be restored when returning from the call. If there were no parameters, this will be 0.

10 Save PTRB on the 6502 stack. This points to the next part of the line to be interpreted, and will need to be restored after the call has returned. The stacks are now in the state shown in fig 5.1(c).

11 Start off the call by executing a JSR to the statement interpreter, which will start executing statements from PTRA. This leaves this return address on the 6502 stack ready for a FN return statement or an ENDPROC statement (all other statements JMP back to the statement interpreter when they have finished; only the ENDPROC and FN return statements finish by executing an RTS).

12 When we get here, the FN or PROC has finished. If it was a FN, then the result type will be in &27, and the value will be in IntA, StrA, or FPA as appropriate.

13 Restore PTRB from the 6502 stack. This points to the place in the line where interpreting should continue.

14 Pull the number of parameters from the 6502 stack. If there were any, restore the old value of each one by pulling its *variable descriptor block* and value from the BASIC STACK.

15 Restore PTRA from the 6502 stack. The only thing left now on the stack, is the FN or PROC token, which was used to tell the ENDPROC or FN return statement which type of call it was in.

16 Recover the old 6502 stack from the BASIC stack. The stacks are now back to the state that they were when the FN/PROC handler was called (fig 5.1(a)).

17 Retrieve the type of the result from &27 into A, in case this is a FN. If it is a PROC, this stage is not needed, but does no harm.

18 Execute an RTS to return to the code which called the FN/PROC caller. In the case of a FN, this returns to the expression evaluator, with the type of the result of the FN in A, and the result itself in IntA, FPA, or StrA. In the case of a PROC, this returns to the PROC statement handler, which sets PTRA to point to the next statement (using PTRB to find out where the FN/PROC handler had got up to), and jumps back to the statement interpreter to continue execution after the PROC.

By trapping the ‘No such FN/PROC’ error generated if the DEF FN or DEF PROC is not found in stage 6 above, procedures and functions can be overlayed from disc (or tape, but it’s not so useful). There is more on overlaying FNs and PROCs in chapter 8.

The LOCAL statement inside a FN or PROC has to save the old value of variables in a similar way to parameters passed to the call. Each variable in the LOCAL statement has its value pushed on the BASIC STACK, followed by its *variable descriptor block*; and the ‘Number of parameters’ byte on the 6502 stack is incremented. The current value of the variable is then set to zero. Saving it in this manner means that its old value will be restored as if it was just another parameter, when the call returns.

The ENDPROC statement and the ‘=’ (FN return) statement check the state of the stack before they return (just returning could have disastrous results if they didn’t). If they find that there are not at least 4 items on the 6502 stack (there won’t be any if it isn’t in a PROC or a FN), they generate a ‘No FN’ or ‘No PROC’ error. Also, if the token at &1FF (the bottom of the stack) does not match (i.e. a PROC token for ENDPROC, or a FN token for the FN return statement), this error is also generated. Otherwise, if everything is OK, then they execute an RTS (after evaluating the <numeric> in the case of the FN return statement) to return to the FN/PROC handler at stage 12 above.

When executing statements inside a FN or PROC, the 6502 S register contains &F5 (i.e. the next available byte on the stack is at &1F5), and the state of the stack is as follows:

&1F6	RTS addr for FN/PROC handler	2 bytes
&1E8	PTRB base MSB	1 byte
&1F9	PTRB base LSB	1 byte
&1FA	PTRB offset	1 byte
&1FB	number of parameters	1 byte
&1FC	PTRA base MSB	1 byte
&1FD	PTRA base LSB	1 byte
&1FE	PTRA offset	1 byte
&1FF Bottom:	FN/PROC token (&A4/&F2)	1 byte

Note that when the FN/PROC handler gets back at stage 12, the RTS address has been removed from the top.

5.4 IF...THEN...ELSE

This construction allows the statements after the THEN or the ELSE parts to be executed conditionally, depending on the value of the <testable-condition> found after the IF part.

The action of the IF satement is:

1 Evaluate the <testable-condition> following the IF token (i.e. the <numeric> after the IF token: they are just the same).

2 If the <testable-condition> evaluated to be 0 (i.e. false), then scan through the line until an ELSE token or the end of the line is found. If no ELSE was found on the line, then continue execution on the next line. Otherwise, set PTRA to point to the character after the ELSE token, and continue at stage 4.

3 If the <testable-condition> evaluated to be anything other than 0 (i.e. true), check for a THEN token. If there isn’t one, JMP to the statement interpreter to continue executing the rest of the line after the <numeric> (you don’t have to use a THEN). If there is a THEN token, set PTRA to point to the character after it, and continue at stage 4.

4 Check for a (tokenised) line number following the THEN or ELSE; if there is one, execute a GOTO to that line number. Otherwise, JMP to the statement interpreter to continue executing the rest of the line.

Note that once the IF statement has decided that the THEN section is to be executed, the IF statement does not prevent it from ‘falling into’ the ELSE clause; this is done by the general statement interpreter itself. If it discovers that there is an ELSE token on the end of the statement it has just executed, it will just skip the rest of the line instead (as if it was a REM statement). This means that lines like:

```
PRINT "HELLO" ELSE MISTAKE
```

will not give an error, but the ELSE clause will never be executed.

5.5. REPEAT...UNTIL

This is the simplest of BASIC’s two loop structures, the other being the FOR. ..NEXT loop. Using this loop, control is repeatedly passed back to the statements following the REPEAT until the UNTIL clause is satisfied.

This loop structure uses a stack in page 5 to save the location of the start of the statement after the REPFAT, so that the UNTIL statement knows where to pass control back to if it is not satisfied. The format of the REPEAT stack is

&5A4+RSP	LSB of repeat address
&5B8+RSP	MSB of repeat address
&24	REPEAT stack pointer (RSP)

The action of the REPEAT statement is:

1 Check that the REPEAT stack pointer (RSP) is less than 20 (&14). If it isn’t, the REPEAT stack is full, so generate a ‘Too many REPEATs’ error (error number 44).

2 PTRA points to the character after the REPEAT token, so push that address on the REPEAT stack, incrementing the REPEAT stack pointer.

3 JMP to the statement interpreter to continue execution with the statements after the REPEAT token.

The action of the UNTIL statement is:

1 Evaluate the <testable-condition> following the UNTIL token, checking that it is at the end of the statement (if it isn't at the end of the statement, a 'Syntax error' is generated).

2 Check that the REPEAT stack is not empty (i.e. the REPEAT stack pointer is not 0). If it is, generate a 'No REPEAT' error (error number 43).

3 If the <testable-expression> evaluated in stage 1 was zero, get the address of the statement following the REPEAT from the REPEAT stack, leaving it on there for the next time this UNTIL statement is encountered. Set PTRA to this address, and JMP to the statement interpreter to continue execution at the statement after the REPEAT.

4 If the <testable-expression> was not zero, remove the top entry from the REPEAT stack by decrementing the REPEAT stack pointer, and JMP to the statement intepreter to continue execution with the statements following the UNTIL statement.

5.6 FOR...NEXT

This loop structure allows a series of statements to be performed a set number of times, with a different value of the control variable each time. This is a more complex loop than the REPEAT ... UNTIL loop, as far as the interpreter is concerned, because it takes more time to set up, and there is more to do every time it goes round the loop.

This loop has to save the address and type of the control variable, the STEP size, the TO limit, and the address of the statement after the FOR statement. For this, it has a stack in page 5 in the following format:

&500−50E	First 15-byte FOR entry
&50F−51F	etc.
&587−595	Tenth 15-byte FOR entry
&26	FOR stack pointer (FSP) (multiple of 15)

The FOR stack pointer is an offset from &500 to the next available 15-byte FOR slot. The format of each 15-byte entry is:

&00	Address of control variable	2 bytes
&02	Type of control variable	1 byte
&03	STEP size	5 bytes
&08	TO limit	5 bytes
&0D	Address after FOR statement	2 bytes

If the control variable is integer, it only uses 4 of the 5 bytes allocated for the STEP size and TO limit.

The action of the FOR statement is:

1 Get the variable following the FOR token; this is going to be the ‘control variable’. If it is invalid, or a string variable, generate a ‘FOR variable’ error (error number 34).

2 Check for an equals sign (‘=’) following the variable; if there isn’t one, generate a ‘Mistake’ error (error number 4).

3 Evaluate the <numeric> after the equals sign, and set the value of the control variable to this.

4 If the FOR stack pointer is &96 (150) or more, there are already 10 FOR loops in operation and the FOR stack is full. If this is the case, generate a ‘Too many FORs’ error (error number 35).

5 Save the address and type of the variable (i.e. its *variable descriptor block*) on the FOR stack.

6 If the next character on the line is a TO token, evaluate the <numeric> after it (making sure it is the same type − real or integer − as the control variable), and save that on the FOR stack. If it isn't a TO token, generate a 'No TO' error (error number 36).

7 If the next character is a STEP token, get the <numeric> following that to use as the step size (making sure it is of the correct type again). If it isn't a STEP token, use 1 as the STEP size instead.

8 Check that we are now at the end of the statement, and set PTRA to point to the next statement.

9 Save PTRA on the FOR stack, to tell NEXT where to return to, and move the FOR stack pointer up by 15 bytes to cover this new FOR entry.

10 Finally, JMP to the statement interpreter to continue execution with the statements after the FOR statement.

The action of the NEXT statement is:

1 Look for a varable name after the NEXT token. If there is one, get its *variable descriptor block* and look down the FOR stack, throwing away the top entry, until the same variable is found. If the FOR stack was empty, generate a 'No FOR' error (error number 32); if the FOR stack wasn't empty, but a FOR loop could not be found with the same control variable, then generate a 'Can't match FOR' error (error number 33).

2 If there was no variable after the NEXT, check that the FOR stack is not empty (generate a 'No FOR' error if it is empty).

3 Get the type and address of the control variable, so that real and integer loop variables can be handled separately. Note, however, that NEXT does not differentiate between single-byte and 4-byte integers (although FOR does), so a single byte variable like '?A%' may give unpredictable results if used as a control variable.

4 Add the STEP size to the control variable.

5 If the new value of the control variable is inside the TO limit (less than or equal if STEP is positive; greater than or equal if STEP is negative) set PTRA to the address of the statement after the FOR statement (from the FOR stack), and JMP to the statement interpreter to continue execution with those statements.

6 If the new value of the control variable is outside the TO limit, move the FOR stack pointer down by 15 bytes to remove the top entry.

7 Set PTRA to point to the next character of the NEXT statement. If it is a comma (‘,’), go back to stage 1 as if it was a new NEXT statement (i.e. we have a multiple NEXT statement). Otherwise, JMP to the statement interpreter to continue execution with the statements following the NEXT statement.

5.7 ON...GOTO/GOSUB

This program control statement allows control to be passed to different parts of the program, depending on the value after the ON.

The action of the ON statement is:

1 If the first chracter after the ON token is an ERROR token, then go to the ON ERROR handler (section 5.8).

2 Evaluate the <numeric> following the ON token.

3 If the next character is not a GOTO or a GOSUB token, generate an ‘ON syntax’ error (error number 39).

4 Save the GOTO or GOSUB token on the 6502 stack.

5 If the value of the <numeric> was less than zero or greater than 255, give up trying to match it; otherwise, count along the list of line numbers to try find the entry corresponding to the ON control value. If the entry was found, pop the GOTO or GOSUB token from the 6502 stack, and jump into the GOTO or GOSUB routine (depending on the token) to pass control to that line number.

6 If no match was made, remove the token from the 6502 stack, and look to see if there is an ELSE token on the line. If there is, handle it as if it was an ELSE in an IF statement (i.e. if there is a line number after the ELSE token, GOTO it, otherwise continue execution with the statements after the ELSE token).

7 If there is no ELSE token on the line, generate an ‘ON range’ error (error number 40).

In BASIC1, the token is not popped from the 6502 stack at stage 6; so if an ELSE clause is found and executed, the 6502 stack state has been messed up. If the ON statement was inside a FN or PROC (which keeps its return address on the 6502 stack), this will cause BASIC to crash on the FN or PROC return. The ON statement works correctly without the ELSE clause; and this bug has been cured in BASIC2 anyway.

5.8 ON ERROR

This statement does not directly change control of the program execution like the other program control mechanisms, but it does still involve using the pointers in a similar way. It changes the BASIC statements that the error handler executes when an error is generated.

BASIC keeps an ON ERROR pointer in page zero at &16,&17. This points to the start of a section of BASIC which will be executed when an error occurrs.

In BASIC1 the default error handler (stored as 2 lines in the ROM starting at &B443) is:

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

In BASIC2 the default error handler (only 1 line at &B433) is:

```
REPORT:IF ERL PRINT" at line ”;ERL:END ELSE PRINT:END
```

The action of the ON ERROR statement is:

1: If the first character after the ERROR token is an OFF token, set the ON ERROR pointer to point to the default error handler, and JMP to the statement interpreter to continue with the statements after the ON ERROR OFF statement.

2: If the character was not an OFF token, then set PTRA to point to the first character after the ON ERROR, and set the ON ERROR pointer to point to this. This means that, should an error occurr, these statements will be executed as the error handler.

3: Finally, skip the rest of the line as if it was a REM statement (we don’t want to execute the error handler yet), and continue execution of the program on the next line.