AXR

ASSEMBLER EXTENSION ROM

FROM ACORN USER

FOR THE BBC MICROCOMPUTER

CONTENTS

Assembler Extension ROM (AXR)

6502 Assembler for the BBC B/B+ microcomputers

By Steve Picton of IFEL (Interface Electronics & Computing), Plymouth

The Assembler Extension ROM (AXR) is one of a series of products tested and supported by Acorn User, a monthly magazine for users of Acorn computers including the BBC micro and Electron. Look out for details of other products in Acorn User, or write to the address given below.

Revised edition 1st July 1986

While the Assembler ROM has been carefully tested, the publisher does not accept any liabilities with respect to the program.

The AXR and manual were written by Steve Picton, and edited by Steve Mansfield and Bruce Smith. Thanks are due to Mike Ginns for his assistance and suggestions.

Any correspondence about the AXR should by addressed to: AXR ROM, Acorn User, 141-143 Drury Lane, LONDON WC2B 5TF.

INTRODUCTION

The AXR from Acorn User is a versatile ROM for use with the BBC microcomputer. It has been written to circumvent the shortcomings of the built-in assembler (particularly that of the rather restrictive BASIC 1), and also to cope with the extra instructions available in the CMOS version of the 6502. Additionally, assembly may be performed directly from disc or tape files. All the facilities of the BASIC 2 assembler are implemented, including 'offset' assembly.

An understanding of assembly language programming on the 6502 is assumed throughout this manual. The way in which the AXR is used is very similar to the standard assembler contained within the BASIC interpreter, and most of your existing assembly language programs can be run with only slight changes.

OPERATING REQUIREMENTS.

For the assembler to work properly, the BBC micro (B or B+) must be fitted with operating system version 1.2. Your machine must also be fitted with BASIC.

FITTING THE CHIP

Disconnect the computer from the mains supply, and remove the lid of the computer.

In the case of the standard model B, you will need to undo the bolts that hold the keyboard in position. Insert the AXR chip into any one of the available ROM sockets. Take care that none of the pins become bent sideways, and also ensure that it is inserted with correct polarity. The small notch in the end of the chip should be facing the back of the computer.

Finally, re-assemble the machine in the reverse order.

SOME TERMINOLOGY

Throughout this manual, the following nomenclature is used.

&	A hexadecimal number follows.
<RETURN>	Press the RETURN key.

TESTING THE ROM

When correctly installed, typing *HELP should cause a listing of the ROM titles. The AXR followed by a version number

should appear in the list.

The ROM has two possible states, enabled or disabled. Being able to disable the ROM helps to reduce the possibility of 'clashes' with other ROMs. At power-on, or after a CTRL-BREAK, the ROM defaults to the 'off' state. You can enable the ROM with

*ONAXR and disable it with

*OFFAXR

It is suggested that the ROM should always be placed in the disabled state unless it is specifically required.

Another effect of *ONAXR is that it resets the value of OPT to 3. The reason for this, and the significance of OPT, are explained later.

Notice that ALL the 'star' commands on the AXR may optionally be preceded with the letters 'AU' (Acorn User). Furthermore, the case of the letters is unimportant, and commands may be abbreviated by means of a period. Hence "*auoNAxr" and "*AUOFFA." are both valid.

MEMORY USAGE

During the assembly process, extensive use is made of page 12. This page consists of the addresses &C00-&CFF inclusive. Zero page locations &60-&78 are also needed by the Assembler ROM.

In practice, this memory usage does not cause any difficulties. It is only during assembly that they are required. The final machine code program may use any section of memory whatsoever, within constraints imposed by the operating system and current language. If you want to place your machine code in page &C, then you will need to assemble it somewhere else first, and then move it down to &C00 onwards when assembly is finished. The 'offset' assembly facility takes care of the problem of absolute addresses, and is described later.

Use *OFFAXR (see above) when assembly is complete to guard against accidental corruption of memory.

ASSEMBLY LANGUAGE FROM WITHIN BASIC

The ability of the BBC micro to assemble code included in a BASIC program is one of its many strong points. The AXR was therefore specifically written to allow assembly to be performed in this way. The underlying principle of the AXR is that it assembles instructions when BASIC finds they have been used in an 'illegal' way. You do not have to enter and exit the assembler using [and]. This means that you can very often use the AXR to assemble your code just by removing the [and] symbols from existing programs. The programs below both

generate identical code (at &5000 onwards).

```
10 REM Standard assembler
20 P%=&5000
30 [ OPT 3
40 LDA #ASC"A"
50 JMP &FFE3
60 ] : CALL &5000
```

```
10 REM AXR Assembler
15 *ONAXR
20 P%=&5000
30 OPT 3
40 LDA #ASC"A"
50 JMP &FFE3
60 CALL &5000
```

It is important to realise that you should NOT use the [and] symbols if assembly is supposed to be performed by the AXR.

NAMES OF VARIABLES & CASE OF LETTERS

You have complete freedom of choice over the case of letters for the instruction mnemonics, for the two index registers and the Accumulator. The following are therefore all valid.

```
dEC a:PHx:lDx &E,y:cmp(&50,x):ASL A:ROR a
```

The various rules governing the names of variables are covered in the BBC micro User Guide. Most importantly, they must not start with a BASIC keyword. BASIC keywords are always in capital letters, so 'next' is a valid variable, but 'NEXT' is not. One of the the problems with the standard assembler, whereby an instruction such as 'ASL Addr' is taken to be accumulator addressing, is not present in the AXR.

Throughout this manual, we will adopt the convention of using capitals for mnemonics, and lower case letters for labels and variables.

MEMORY SHARING

When assembly language programs are run as part of a BASIC program, we can consider the available memory to be split up into three sections.

1) The memory used to hold the source (assembler) code. This starts from PAGE (often &E00 or &1900) and extends upwards to TOP. The size of the program is thus given by TOP-PAGE.

2) When the program is run, any variables that are defined are usually stored immediately above the main program, i.e. from TOP upwards. The more variables there are (which is effectively what a label or symbol is), the more memory must be set aside to contain them.

3) If the machine code is being written into RAM, then we clearly need to be absolutely certain that it is not going to corrupt either our source code or the memory used for the variables. Ideally, we should be able to set up a block of memory somewhere, and be quite sure that BASIC will not try to use it for anything. We can then write the machine code into this block.

This last requirement is achieved by special use of the DIM statement. The syntax used is

```
DIM <variable> <number of bytes-1>
```

Hence the use of 'DIM code 99' allocates a block of 100 memory bytes, starting at the address given by the variable 'code'. It is safe to write anything at all into this block without affecting the BASIC interpreter at all.

The number of bytes that should be reserved in this way depends on your application. Any number up to several thousand might be needed. It is best to err on the large side, but avoid choosing huge numbers as a matter of routine. If the statement

```
DIM block% 10000
```

was given, then about 10000 bytes are 'lost', and the BASIC ROM will think that there is correspondingly less memory for variable storage. Errors such as 'Bad MODE' and 'No room' are now much more likely to occur.

THE LOCATION COUNTER, P%

In the last section we saw how to set up a block of memory so that we can write the machine code into it without fear of corrupting memory required by BASIC. We must now look at how to make the Assembler ROM place the machine code that it produces into our chosen section of memory.

As assembly proceeds, the computer maintains a 'location counter' in the integer variable P% (see User Guide page 66). Used in this way, P% always holds the memory address into which the next instruction will be assembled. We would normally expect to initially set P% to the first address of a block of memory produced by a DIM statement. We have not bothered to do this in many of the examples that follow, because the amount of code that will be produced is so small. Instead, P% is just set to an absolute memory address at the start of an area which is known to be free -- e.g. P%=&5000. Sometimes P% is set to something like &A00 or &900 when short routines are in use. This is done so that it remains immune from the effect of LOADing in a new program, changing graphics mode etc.

In order to see the variable P% working, type in the following sequence of commands:

```
P%=&6000 : OPT 1     <RETURN>
```

(If you get a 'Mistake', then the Assembler ROM is disabled - type *ONAXR to remedy the situation)

This will cause the 'program counter' P% to be set to hex 6000. This in turn means that our machine code will start being produced from &6000 onwards. Ignore the effect of OPT 1 for the moment. Enter the single line

```
PHA                  <RETURN>
```

This should produce on the screen:

```
6000 48        PHA
```

The way to interpret this information is as follows. P% was initially set to &6000, so this is where we expect to have our machine instructions placed. The opcode of our PHA mnemonic (&48) has been written into address &6000. The assembly language statement that gave rise to the object code (just 'PHA' in this case) is listed out too.

We have said that P% is continuously pointing to the next address for assembly. If this is so, then P% should now be &6001, since PHA is a single byte instruction. Typing PRINT ~P% will confirm that this is the case. (The '~' symbol makes the computer print out a value in hexadecimal.)

HINT: Typing '[' followed by <RETURN> is a quick way of seeing the value of P%.

If assembly language is relatively new to you, it may be worthwhile typing in a few more instructions. Remember to press RETURN after each one.

```
ASL &97
LDA # 5
ROL &1234

oscli=&FFF7  :  JSR  oscli
```

Let us now put together a simple assembler program to print a letter 'A' on the screen. We can do this by loading the accumulator with an ASCII 'A', and then calling the subroutine at &FFE3, which is the operating system's character output routine.

```
10 P%=&6000  :    REM Do not bother with 'DIM'
20 OPT 1     :    REM OPT is explained later
30 osasci=&FFE3 : REM The subroutine address
40 LDA # ASC("A")
50 JSR osasci
60 RTS          : REM This returns control into the BASIC ROM
70 CALL &6000   : REM Execute the code
```

RUN this program to achieve the desired effect, and notice the listing that is produced. Users who are already familiar with the assembler in the BASIC ROM will see how similar this program is to a 'standard' BBC BASIC assembler program. The main difference is that the assembler entry/exit symbols ([and]) are not present.

ARITHMETIC EXPRESSIONS

Arithmetic expressions, both string and numeric, are permitted in the assembly language statements. The usual *, /, -, and + operators are allowed, as are BASIC keywords (MOD, DIV, OR etc). Operator precedence is described at some length on page 144 of the User Guide, and the user is encouraged to refer to this section if necessary.

In practice, complicated expressions are rarely needed, and simple add, subtract, and occasional multiply operations will normally suffice. The MOD and DIV operators are useful though, for setting up indirect pointers. E.g.

```
LDY #command DIV 256
LDX #command MOD 256  \Point to command
JSR &FFF7  \Call OSCLI
```

The arithmetic and logical instructions, all of which allow indirect addressing, carry a slight complication when brackets are used in the operand field. This is because if a '(' appears as the first non-space after the mnemonic, an indirect addressing mode is assumed. The difficulty arises if a bracket is being used in order to determine the way an expression is evaluated, rather than to imply a particular

addressing mode. The simple solution is to precede the expression with a dummy operation, such as '0+' or '1*'. E.g.

```
LDA 0+(&20+2)*3
CMP 1*(&15+3)*4
```

An error would occur without these dummy operators.

NUMBER BASES

The BBC computer will assume that all numbers are in base 10, unless prefixed by a '&' sign. This implies that a hex number follows. No other bases are normally recognised.

The Assembler ROM will allow the use of binary numbers. These are particularly handy for 'bit-masks' for use with the logical instructions. There are no restrictions on where they may occur in an arithmetic expression, as shown by the PRINT statement below. Binary numbers are prefixed by a '%', Examples.

```
PRINT %1110 + %110 * %101 * SQR(%1001)
AND # %10111111
ORA # %100
OPT %11
LDY # %1111
```

LABELS

When writing long assembly language programs, we will often need tables of numbers or text/error messages. We may also want to jump to other parts of the program. But as we make changes to the source code, so the absolute addresses change too. Therefore in one version of a program an instruction might assemble as JMP &5543 and, after a few changes, as JMP &5537. Clearly, JMP instructions should rarely have an address specified by an actual number, but instead by a label (i.e. use JMP label). Then if the program is changed, the value of the label probably changes, but the assembler always puts the correct address into the JMP instruction. Similar comments apply to data tables etc, accessed by e.g. LDA datatable,X.

We have already met the idea of using a variable name instead of an absolute address, e.g.

```
oscli=&FFF7 : JSR oscli
```

In the assembler, a label is formed by using a period '.', followed by a valid variable name. It is suggested that only lower case letters are used for labels.

The following program uses a simple loop to print five asterisks.

```
 10 P%=&6000
 20 OPT 1
 30 times=5
 40 osasci=&FFE3
 50 LDA # ASC("*") : REM The char. to print
 60 LDX # times
 70 .loop
 80 JSR osasci
 90 DEX
100 BNE loop
110 RTS
120 CALL &6000
```

When line 70 is encountered, the Assembler ROM creates the variable 'loop', and assigns to it the current value of P% (&6004 in this case). If we subsequently refer to the variable 'loop', the assembler will now know that it must use the value &6004.

Notice that when the label is referenced (line 100), the preceding period is not needed. The assembler calculates the correct offset to be used in the branch instruction. In this case the offset was &FA, but if any instructions were added (say NOP at line 95), the assembler will compensate, and correct it to &F9.

The only item that may follow a label on the same line is either another label, a colon, the keyword ELSE, or a comment (see below). The following examples illustrate this.

```
.loop .jumptable   .list
.error2   \The error message
```

The first example would create all three variables, and assign each one the current value of P%.

COMMENTS

Comments are denoted by a '\' symbol, located just below and to the right of BREAK. A comment may appear on its own, after a label, or after an instruction mnemonic. E.g.

```
\ This is just a comment
.fred  \'fred' is given the value of P%
TAX  \Save the counter in X
```

A comment should NOT contain a colon however. This is because a colon is treated as being a continuation symbol for multi-statement lines. Thus the text immediately after the ':' would be assumed to be part of the assembler program (or normal BASIC). In the following example, all the instructions would be assembled, at the address given by P% as usual.

```
t=9:LDA#t \time delay:TAX \save it:ASL A \double it
```

Many assemblers allow a semi-colon to be used for commenting. This has been deliberately avoided (for assembly from memory) because it is physically close, and very similar in appearance, to the ':' used for multi-statements. It would be easy to accidentally enter a semi-colon, thereby preventing the next instruction from being assembled.

PSEUDO-OPS

Pseudo-ops control the way in which the assembler operates. Some pseudo-ops do not generate any object code. Others do, like the data definition directives described later.

OPT

An important pseudo-op that we have mentioned briefly is OPT. It is an aide memoire for OPTion, and it controls, amongst other things, whether a listing of the assembled code is given.

The number specified after OPT must be thought of as being a single byte quantity. All of the eight bits have a certain meaning, unlike the assembler in the BASIC ROM, which only uses three bits (two in the case of BASIC 1). Bit 0 is the least significant bit, and bit 7 the most significant one.

BIT 0 - List control

This bit determines whether or not a listing is provided of the assembled code. When the bit is set, a listing is given. This is why we used OPT 1 previously. All the examples so far would have worked if OPT 0 had been used, but you would not have been able to see the value of P% changing, or the assembled machine instructions.

BIT 1 - Error control

We have not yet introduced the idea of a forward reference. The program below contains a forward reference.

```
110 BCS forward
200 ...
210 .forward   \A label
```

When the computer first encountered line 110, it would be impossible for it to assemble the instruction correctly. This is because it would not know the address of the label 'forward', and hence it couldn't calculate the correct offset for the branch.

As far as the computer is concerned, this situation constitutes a 'No such variable' error. The usual way round this problem is to perform what is called two-pass assembly - the computer literally goes through the source code twice. During pass 1, we make sure that the computer doesn't give 'No such variable' when it meets an unknown quantity. At the end of the first pass, the computer should have gathered together the values of all the labels and symbols used. Hence on the second pass, it will be able to substitute the right numbers at the right time. We would be wise to enable error

messages during pass 2, so that labels which were referenced, but had never been defined, were reported.

On the first pass, any unknown variable is assumed to have the value of P%. Since P% will invariably be greater than &FF, unknown labels/symbols are given a sixteen bit value. This means that zero page locations MUST be defined before they are used, otherwise absolute addresses are thrown into confusion.

Bit 1 of OPT controls the error reporting. When it is set, errors are given, and when it is clear errors are generally ignored. We therefore resolve our forward reference difficulty by the use of a FOR-NEXT loop, and give OPT a suitable value, depending on whether we are in pass 1 or 2.

During pass 1, there is not much point in having a listing given, and we don't want 'No such variable' errors reported. Hence we often use OPT 0.

In pass 2, we may want to have a listing (although it isn't necessary), and we certainly want errors to be detected. On pass 2 we will use OPT 3 here. The program below contains two forward references, but at the end of the second pass, it will have assembled correctly. Notice how the use of binary in line 10 helps to emphasise the bit settings.

```
 10 FOR pass=0 TO %11 STEP %11
 20 P%=&6000
 30 OPT pass
 40 SEC
 50 BCS skip
 60 BNE skip
 70 PHA
 80 .skip
 90 RTS
100 NEXT pass
110 CALL &6000
```

Notice that P% is reset to its starting value at the beginning of each pass by including it within the FOR-NEXT loop. This is most important.

It would be possible to add a line 85, say 'JMP start'. On pass 2 ONLY will you get an error message.

In an assembly language program, it turns out that there are many different types of error that might occur. For example, an index register used illegally, or an invalid addressing mode. When errors are supposed to be suppressed, the assembler will not ignore every type of error. Thus a statement like

```
CMP (pointer),X
```

will always be rejected with an 'Index' message, irrespective

of the OPT setting. The general rule is that if the situation could possibly be different on a second pass, then the error will be ignored. But the above example can never assemble correctly, and hence the error.

BIT 2 - Offset assembly

The concept of offset assembly is simple. We assemble our machine code in memory somewhere (&3000 say), but all the absolute addresses are based on the assumption that the code will be moved somewhere else before it is executed. One use for this is when we wish to assemble code for a sideways ROM, but we do not have a sideways RAM pack installed. In this case, we might assemble the code at &3000, and then 'blow' it into an EPROM. It would then be plugged into a sideways ROM socket, which fits into the memory map at &8000.

During offset assembly, the location counter P% is still used to keep track of absolute addresses. We need another counter, however, to maintain the actual memory address in which to write the machine code. The variable used by the Assembler ROM for this purpose is Q%. (BASIC 2 uses O% for offset assembly - note this difference.)

We need to be able to tell the Assembler ROM that Q% is being used for the code destination, not P%. This is done by setting bit 2 of OPT. Every time an instruction is assembled, the object code is now written into the address given by Q%. Both Q% and P% are then incremented by the correct amount.

The program below uses offset assembly. P% is primed with &6000, which is where the code is designed to execute. Q% is set to &5E00, and it is here that the object code is placed. Memory from &6000 onwards is not affected in any way when the program is run.

```
10 P%=&6000
20 Q%=&5E00
30 OPT %101  :  REM Bits 0 and 2 set
40 .back
50 JMP back
60 LDA back,Y
```

The listing obtained suggests at first that code is being written into memory at &6000. A memory editor, if you have one, or the disassembler described at the end of this manual will confirm that this is not so. In order to run the machine code, it would be necessary to shift the code beginning at &5E00 up to &6000. This is a result of the 'offset' assembly that has occurred.

BIT 3 - Write to file

In all the examples so far, the object code has been

written directly into memory. Bit 3 of OPT provides us with the means to send the output to a file. This may be useful when memory is short, since available memory then only has to be shared between the source code and the various labels etc. Setting bit 3 forces the computer to send its assembled output to a file.

Before anything can be sent to the data file (which can be on tape or disc), we first of all have to open that file for output. In order to simplify this process, another pseudo-op has been provided. In fact, there are several such pseudo-ops which relate to an output file used in this way.

1) OFL <filename>

This creates an Output FiLe under the name given by the parameter. The filename can be any string expression, so that

```
OFL "A"+"B"
```

attempts to open an output file called 'AB'. Generally, OFL would occur near the start of the assembly language program.

2) CFL

CFL will Close the FiLe that was opened by the use of OFL. No parameters are required, but a comment may appear after CFL.

3) FXA <Xaddress>, FLA <Laddress>

FXA means File eXecution Address, and FLA means File Load Address. These pseudo-ops will not work with the tape filing system.

When machine code is written to a file, we may want to be able to *RUN that file when assembly is finished. *RUN will only work if the load and execution addresses of the file are properly set. FXA and FLA make this very easy to do. FXA evaluates the expression after it, and sets the execution address of the o/p file to this value. FLA works in a similar way, but writes the file load address only. Load and execution addresses are not necessarily the same.

We will give an example of all these pseudo-ops when we have described the use of bit 4.

BIT 4 - Inhibit write to RAM

If, as mentioned above, memory conservation is important and we want to write the assembled machine code straight to a file, it may be desirable to prevent the computer from writing to RAM as well. This is achieved by setting bit 4 of OPT. Listings can still be obtained, and will appear as normal. The memory locations shown by P% will not be altered in any way however.

This next example illustrates the use of bits 3 and 4, and the various pseudo-ops just described. It creates a machine code program that will simply print the alphabet when it is *RUN. We have now used the 'recommended' way of reserving memory, i.e. the DIM statement.

```
 10 DIM code% 50 : REM Enough
 20 OFL "ALPHA" \The name of the o/p file
 30 FOR pass=1 TO 2 : REM 2 passes needed
 40 IF pass=1 THEN OPT &10 ELSE OPT &1B
 50 P%=code%
 60 .load_address : PHA
 70 .execution : LDX #25
 80 .printloop : CLC
 90 TXA : ADC #ASC"A"
100 JSR &FFE3
110 DEX : BPL printloop : RTS
120 NEXT pass
130 FXA execution
140 FLA load_address
150 CFL \Close the file
160 *RUN ALPHA
```

(Use *INFO ALPHA if you have a disc drive to see that the load and execution addresses are correctly set.)

The file 'ALPHA' is opened for output in line 20. We don't include it in the FOR-NEXT loop, because there is no need to open it twice.

In line 40, we select OPT &10 on the first pass. This sets bit 4 only and clears all the rest. The reasoning is as follows.

As usual, the listing and error reporting are inhibited on the first pass, so bits 0 and 1 are clear. We are not using offset assembly, so bit 2 is reset also.

On pass 1, the assembled code will not be correct. We therefore avoid writing it to file. Hence bit 3 is clear too.

Setting bit 4 prevents the machine code being written into memory. It wouldn't really matter if it was - we would just get two identical copies of the object code, one on file, the other in RAM.

Things are rather different on the second pass. A listing is given, and errors are reported by the use of bits 0 and 1. Bit 2 remains clear, since offset assembly is not relevant. Bit 3 however is now set, so that the computer will write the (now correct) code to the file 'ALPHA'. The bit settings are %00011011, or &1B.

Lines 50-110 form the simple assembly language program to print the alphabet backwards.

Lines 130 and 140 set the file's execution and load addresses respectively. Again, there is no point in having these pseudo-ops within the FOR-NEXT loop. PHA in line 60 was just a 'dummy' to ensure that the load/execution addresses were not quite the same.

Four bytes are set aside on the disc for both load and execution addresses. If we want to set the two most significant bytes of the load/execution addresses to &FF, we can use BASIC's OR operator, e.g.

```
FXA execution OR &FFFF0000
FLA load_address OR &FFFF0000
```

BIT 5 - Branch to Jump (B-J)

The 6502's branch instructions all use an addressing mode known as relative addressing. Such branches are limited in their range to about 128 bytes either forward or backwards from the current memory address given by P%. You cannot BCS from &6000 down to &E00!

The simple loops used in previous examples never came anywhere near being 'Out of range'. But as we add more instructions in the loop, so the branching distance increases. Eventually, the branching distance is too great for the 6502 to handle. The branch is now 'Out of range'. The assembler ignores this error if bit 1 of OPT is clear

Consider the following program.

```
100 .label
500 .....
900 BCS label
```

If the BCS is out of range, a simple solution would be to reverse the sense of the test, and use a JMP instruction. i.e. replace line 900 with

```
900 BCC hopover : JMP label : .hopover
```

The effect is the same. But there is a penalty, in that this sequence uses five bytes of object code, not two.

Frequently though, the size of the object program is not a prime consideration. We can avoid being plagued by 'Out of range' errors by setting bit 5 of the OPT pseudo-op. Whenever the Assembler ROM detects a branch distance above the maximum, it will now substitute for you the 'reverse branch + JMP' illustrated above. By 'reverse' branch we mean that BCC becomes BCS, BEQ becomes BNE etc. The exception is BRA, which has no inverse, and so it is turned into an ordinary JMP. The two programs below illustrate the B-J option. The first gives 'Out of range', the second does not.

```
10 P%=&6000          10 P%=&6000
20 OPT %11           20 OPT %100011
30 .back             30 .back
40 P%=&7000          40 P%=&7000
50 BCS back          50 BCS back
```

The concept of B-J does have a single complication. This occurs in the case of a forward reference. It turns out that all forward references need to be treated as Branch+JMP sequences if bit 5 is set.

Now this may be rather wasteful on space, particularly if it is obvious from simple observation that an ordinary branch would do. Therefore a special symbol '*' may be placed before the destination label/address. This tells the assembler to temporarily ignore the B-J option. Examine the way in which this program assembles.

```
10 FOR pass=1 TO 2
20 IF pass=1 THEN OPT &20 ELSE OPT &23
30 P%=&6000
40 BCS next
50 PHA : .next
60 NEXT pass
```

Although the branching distance is small, the assembler still changes it to a 'reverse branch+JMP' because it is a forward reference. Change line 40 to 'BCS *next', and then re-assemble it.

Of course, the '*' cannot be used to achieve the impossible. An 'Out of range' error may now occur.

You can also use '*' for backward references. Generally, it is not necessary, because the assembler will deal with this situation 'intelligently'.

BIT 6 - Enhanced error reporting

Newcomers to assembly language are often bewildered by the number of different instructions, and the addressing modes that each one does and does not permit. The idea behind the 'enhanced error reporting' is that the computer prints out the statement at fault, and indicates what it expected to find, and where.

For example, try the following.

```
P%=&6000 : OPT &40 \Set bit 6 : LDA (&70),X
```

Here the X register has been used 'illegally', and so we get an 'Index' message. The computer has also printed a little arrow under the 'X', so you know exactly where the problem is. The statements below will also give errors of one sort or another.

```
LDA #5000
CPX &50,X
JMP (&100
BIT &30,Y
SMB 8,&70
BBR 5
```

Bit 7 - Flag duplicate labels

A label to the assembler is any valid variable name. Labels are used in preference to absolute addresses, because the assembler always takes care of changing their values whenever the program is altered. It can cause all sorts of problems however, if the same label is accidentally used twice in the same program.

Setting bit 7 of OPT instructs the assembler to give an error message ('Duplicate label') if it encounters a label which has already been defined. It goes without saying that bit 7 should NOT be set on a second pass through the source code, since the whole idea of the first pass was to create all the necessary labels. Therefore the labels are bound to exist on the second pass!

This program gives you an error at line 50, because 'looper' has already been created in line 20.

```
10 P%=&6000 : OPT  %10000011 \Bit 7 set
20 .looper
30 LDX #0
40 LDY #%1111
50 .looper
60 RTS
```

If OPT 3 had been used, then this duplication would have gone undetected.

You may occasionally wish to redefine a label, even though 'Duplicate label' trapping is active. This is possible by special use of a '#'. In the program above, we could have used

```
50 .#looper
```

and the variable 'looper' would then have been assigned a new value, &6004 in this case, rather than causing an error.

OTHER PSEUDO-OPS

ORG Specify ORiGin

Up till now, we have been using statements such as P%=&6000 to set the location at which the code is supposed to be assembled. An alternative method is to use the ORG directive. The following are equivalent.

```
P%=&6000                ORG &6000
```

Both will work, but ORG is a fairly standard pseudo-op in assemblers, whereas P% is unique to the BBC micro.

DST Specify DeSTination

When offset assembly is being performed (see previous section), we need to tell the computer where to place the object code. We have done this in the past by using e.g. Q%=&5E00. DST may be used to achieve the same thing.

```
E.g. Q%=&4559           DST &4559
```

Both will set Q% to &4599.

DF DeFine

The DF directives are invaluable for introducing text, error messages, data tables etc, into the object code. There are five variations on DF, four numeric ones and one string form.

DF operates rather like EQU (EQUate) in BASIC 2. BASIC 1 lacks EQU in any shape or form.

1) DFB <expr> (,expr)

DeFine Byte. The numbers after DFB are evaluated as integers, and the least significant byte used for the result. Hence DFB 7 would generate a single byte of code, the number 7 in this case. It is identical to EQUB 7 used in BASIC 2.

An improvement over the standard BASIC assembler (BASIC 2) is that more than one parameter can be given on the same line. A comma must be used to separate them. In BASIC 2 we might use

```
EQUB 13 : EQUB 3 : EQUB 9 : EQUB 37 : EQUB 25*2
```

but the AXR reduces this to the much more convenient form

```
DFB 13 , 3 , 9 , 37 , 25*2
```

2) DFW <expr> (,expr)

DeFine Word. It works in much the same way as DFB, but a two-byte value is generated. Bytes are produced in the usual lo-hi format, which is standard 6502 convention. Thus

```
P%=&6000 : OPT 3 : DFW &1234
```

places &34 in &6000, and &12 in &6001. As with DFB, multiple expressions can be used, separating with commas as normal. P% is incremented by two for each parameter.

3) DFD <expr> (,expr)

DeFine Double word. Here, each expression is evaluated, and the full four bytes of the result are made use of. The lo-byte is used first, and the hi-byte last. Four is added to P% for each parameter.

4) DFR <expr> (,expr)

DeFine Real, the last numeric DF. Occasionally, it may be necessary to include in the object code a series of floating point numbers. These might be a table of conversion factors for example. DFR can be used to do this, and it creates five bytes in the same form that the BASIC interpreter uses for its real numbers. This format is the exponent, followed by the four-byte mantissa (which includes the sign bit). There is usually no obvious relationship between the bytes generated, and original number, as shown by, for example, DFR 4.567.

NOTES

a) A single DFB, DFW, DFD or DFR cannot produce more than 80 bytes at a time. If it is necessary to generate large tables of numbers for any reason, then you will have to use several DF statements in order to keep the 'Too many bytes' error at bay.
b) It is possible to put a comment (using "\") after each parameter, explaining what each one does. Such comments must not contain any commas, because the AXR assumes that this marks the beginning of the next number. E.g.

```
.table : DFB 12 \CLS , 7 \beep , 10 \cursor down
```

DFS <string expr>

DeFine String. Identical to the EQUS used on BASIC 2, this provides a straightforward way of introducing text into machine code programs. The string given in the operand field is written into memory at the address given by P% (or Q% if applicable), and finally P% is incremented by the length of the string. DFS

does not automatically put a carriage return at the end of the string. This is easily done if required by putting '+CHR$13' on the end, or even using DFB 13 as the next statement.

This program demonstrates how we print a simple error (via BRK) if the number in location &70 is greater than 123.

```
10 DIM code 30
20 FOR pass=0 TO 3 STEP 3
30 ORG code : OPT pass
40 memloc=&70 : max=123
50 LDA memloc
60 CMP #max+1 : BCC ok
70 BRK : DFB 20 : DFS "Too big": DFB 0
80 .ok : RTS
90 NEXT pass : CALL code
```

Another use for DFS is to 'reserve' a section of memory. If we needed to set up a block in the object code consisting of 200 zeros, then an easy way to do this would be

```
DFS STRING$(200,CHR$(0))
```

SWR SideWays RAM

If you are developing software for sideways ROMs, the use of a sideways RAM module saves a lot of time. Ideally, we should be able to write the assembled object code straight to the sideways RAM, at &8000-&BFFF.

This is usually possible only if you have an expansion board fitted which forces any write operation in the region &8000-&BFFF through to a particular socket. The AXR Assembler will allow machine code to be written into any chosen socket number, assuming the socket contains RAM of course. This is the case even if there is no expansion board.

We tell the assembler which socket is in use by means of the pseudo-op SWR. This is followed by a number which should be in the range 0-15. Arithmetic expressions are allowed, but hardly necessary. There is no need to use SWR if you are always assembling in the main RAM area (&E00-&7C00). But if P% is set to &8000 before assembly begins, then SWR will usually be needed.

SWR makes use of the least significant byte of the integer variable R%. Thus SWR 7 will assign R% a value of 7.

Listings produced when code is written to sideways RAM will be correct. Both BASIC 1 and BASIC 2 produce strange results, because the data is written to S/W RAM, but is read back for printing from the BASIC ROM itself. Hence the listed output bears no relation to the code that was actually assembled.

N.B. The writing of code directly to sideways RAM does not work with certain systems -- for example, Solidisk's which has its write access register at &FE60.

CONDITIONAL ASSEMBLY

Conditional assembly can be performed very simply. It is even easier than with the BASIC 1/2 assemblers, because there is no assembler entry/exit needed ([and]). E.g.

```
IF G%=12345 THEN SEC ELSE ASL A : TAX
```

This will almost certainly assemble the ASL A:TAX instructions, since G% is unlikely to have the value 12345.

Conditional assembly is not needed very often, but it can be useful for including extra instructions for debugging purposes, or configuring special versions of a program.

MACROS

It sometimes happens in assembly language programming that we find ourselves writing out the same (or similar) sequence of instructions many times over. In cases like these, it is possible to save a certain amount of typing by using a macro. The idea is that we write our instruction sequence once only, and then give it a name so that it can be identified. Whenever we want this special instruction sequence to be assembled, we just call the macro.

The Assembler ROM uses procedures to implement macros in a very versatile way.

Suppose that we have a particular task in mind which involves frequently swapping over the X and Y registers, leaving A and P unaltered. A suitable set of assembly language statements could be

```
PHP:PHA:TYA:PHA:TXA:TAY:PLA:TAX:PLA:PLP
```

Let us further imagine that after each register swapping exercise, we have to do tasks 1, 2, 3, 4 and 5. So our program outline looks like this.

```
a) Do task 1
b) Swap registers
c) Do task 2
d) Swap registers
e) ....

i) Do task 5, and end
```

The program below achieves this. Each 'task' is just a JSR straight to an RTS instruction. The procedure 'swapreg' is defined in lines 160-180. Whenever we use PROCswapreg in the main program this procedure is executed, and the full sequence of instructions making up this procedure is assembled.

```
 10 DIM code 100
 20 FOR pass=0 TO 3 STEP 3
 30 OPT pass : ORG code
 40 JSR task1
 50 PROCswapreg
 60 JSR task2
 70 PROCswapreg
 80 JSR task3
 90 PROCswapreg
100 JSR task4
110 PROCswapreg
120 JSR task5 : RTS
130 .task1 : .task2 : .task3 : .task4 : .task5 : RTS
140 NEXT pass : END
150
160 DEFPROCswapreg
170 PHP :PHA :TYA :PHA :TXA :TAY :PLA :TAX :PLA :PLP
180 ENDPROC
```

The real power of macros arises from the fact that we can supply them with parameters. Thus the macro can produce different object code every time it is called, simply by supplying it with different numbers.

To illustrate this, assume that we wish to issue several OSBYTE calls in order to set the system up in a particular way. On the BBC micro, OSBYTE calls are performed by loading the accumulator with the OSBYTE number, loading X and Y with suitable parameters, and then doing a JSR &FFF4. For example we might use

```
LDA#5 : LDX#2 : LDY#0 : JSR &FFF4
```

The following program will generate the machine code which, when executed, is equivalent to

```
*FX 5,2,0
*FX 12,8,0
*FX 138,0,65
*FX 8,4,0
```

```
 10 DIM code 100
 20 ORG code : OPT 3
 30 PROCosbyte(5,2,0)
 40 PROCosbyte(12,8,0)
 50 PROCosbyte(138,0,65)
 60 PROCosbyte(8,4,0)
 70 END
 80
 90 DEF PROCosbyte(a%,x%,y%)
100 LDA #a%
110 LDX #x%
120 LDY #y%
130 JSR &FFF4
140 ENDPROC
```

Notice how 'readable' and compact our assembler code becomes, much more like a high level language. It only contains a single line of assembler mnemonics, but that line is assembled four times, with different parameters each time. The same principle can be extended to OSWORD, e.g.

```
PROCreadclock(address%)
```

which could be made to read the system clock into the address given by 'address%'.

HINTS

The following points are worth noting when assembling code using the AXR.

1) DO use *ONAXR near the start of the program to enable the AXR.
2) DO remember to use the OPT directive. *ONAXR gives a default value of 3 to OPT
3) DO remember to set P% (and Q% if it is in use) to the appropriate values. Such assignments should be within the FOR-NEXT loop if two-pass assembly is being performed.

You may also wish to make a point of disabling the AXR with *OFFAXR when assembly is complete, to prevent accidental corruption of page &C.

4) DON'T use the [and] symbols to enter/exit the assembler.

ASSEMBLY FROM DISC

All the programs in previous sections were held in the machine's RAM for assembly. There is much to be said for this approach - speed of assembly and convenience are two factors.

There are disadvantages with this method however. The overriding one is that for long programs the source code takes up such a lot of memory, and limits the size of the symbol table. This problem is solved completely by storing the source code on disc (or tape) files. A direct benefit is that it is now possible to insert plenty of blank lines to make the code easier to read and extensive comments that describe the program logic, with no worries about running out of space.

The source code may be created with any wordprocessor. WORDWISE, WORDWISE PLUS and VIEW are all ideal.

THE SOURCE PROGRAM

Each line of the source file(s) consists of a sequence of ASCII characters. The end of the line is marked by a carriage return, ASCII 13, or the end of the file itself. A line must not be longer than 235 characters. We no longer have all the facilities of a BASIC program immediately available to us, but this is hardly a problem with assembler source code.

Lines in the source file should not start with a number, and no multi-statement lines are permitted. The exception to this is when a procedure is being defined for macros. This is detailed later on.

Each line will have the following general form.

‹label› ‹mnemonic› ‹operand› ‹comment›

Users will realise that not all of these fields are mandatory. After all, some mnemonics, like TYA, would not expect to have an operand after them. Spaces in the source code are not normally significant. Many assemblers that read code from files are fussy about spaces, which means that a word like 'TAY' can mean two different things, depending on where it occurs in the line. If it was in the very first position, it becomes a label. When preceded by just one space however, it becomes the familiar 6502 mnemonic. The AXR avoids this confusion by allowing most of the above four fields to appear anywhere in the line. An 'operand' would need something before it of course. Also remember that at least one space would be required to separate a mnemonic from a label on the same line, otherwise the mnemonic will become part of the label. For example.

```
.move TAY
.moveTAY
```

The first is the label 'move' and the instruction 'TAY'. The second is just a label 'moveTAY'.

THE LABEL

Labels are defined just as they are when assembling from RAM, i.e. by a period followed by a valid variable name.

THE MNEMONIC

The second field can be one of two things. First, and most obviously, any 6502 mnemonic may appear here. The full extended 6502 instruction set is supported. Second, a pseudo-op is valid. Examples are ORG, DFB, DFR and so on.

Since the first field (the label) is frequently optional, and since spaces are not important, it may be that the 'mnemonic' becomes the first field not the second! The examples below illustrate this.

```
.mainloop   TAX
TAX
```

Both would allow the TAX instruction to be assembled, but the first also creates the label 'mainloop'.

THE OPERAND

Operands are used in exactly the same way as they are during assembly from memory, e.g.

```
PHA
LDA #39
LDA (&70),Y
```

THE COMMENT

Comments are always preceded by a '\' or a semi-colon for assembly from files. Semi-colons were specifically excluded for commenting purposes in BASIC programs, because of the similarity with the continuation symbol, ':'. Apart from allowing semi-colons, comments may be used as described previously, e.g.

```
\This is just a comment
\So is this
.loop  LDX # 4  \Set counter
PHA             \Preserve the digit counter
```

EQU - EQUate

The AXR uses this directive to assign a particular value to a label. It acts rather like BASIC's LET statement. In BASIC, we might use

```
osasci = &FFE3
```

but when assembling directly from a file, the equivalent syntax becomes

```
.osasci  EQU  &FFE3
```

AN EXAMPLE

The *ASSM command is used for assembly of the source code, but before this can be of any use, we must create the source file. We already have enough information to put together a simple program in the correct format, so a worked example will now be given. The user is now assumed to have either WORDWISE or VIEW (say), with which to produce the main text. The example below should now be entered, although the comments can be left out of course.

```
\Program to print out A-Z

ORG &6000              \Effectively P%=&6000
.first EQU ASC("A")     \LET first=ASC("A")
.last EQU ASC("Z")      \LET last=ASC("Z")
.oswrch EQU &FFEE       \LET oswrch=&FFEE
LDX # first
.loop
TXA
JSR oswrch
INX
CPX # last+1
BNE loop
JMP &FFE7               \New line

\End of program
```

Save this to file under the name 'TEST1' in the appropriate way (menu option 1 on WORDWISE).
*ASSM will only work from BASIC. Therefore re-enter BASIC by using *BASIC.

The AXR is normally a two-pass assembler when dealing with disc or tape files. It is clearly necessary to be able to control things such as error reporting, listings, offset assembly etc. Since the OPT pseudo-op encountered previously is not permitted in the source code on file, the OPT information

is provided as part of the *ASSM command itself.
The *ASSM syntax is

*ASSM <src> <OPT 1> (<OPT 2>) (<dest>)

<src> is the name of the source file, just TEST1 in this case. This parameter MUST be given.

<OPT 1> also is obligatory because at least one pass of the source file(s) is performed by the assembler. <OPT 1> should never exceed &1FF.

The various bits of the OPT parameters have the following meaning, bit 0 being the least significant one.

Bit no.	Meaning
0	Listing control, 0=no listing
1	Error reporting, 0=don't report
2	Offset assembly, 0=not active
3	Send output to file, 0=not to file
4	Inhibit writing to RAM, 1=inhibit
5	Convert out of range branch to jump, 0=don't convert
6	Not used when assembling from file
7	Flag duplicate labels, 0=don't flag
8	no second pass, 1=No second pass required

Notice that there are now 9 bits in total, not 8, although bit 6 is not used here. The extra one, bit 8, is only used on the rare occasions that you wish to perform a single pass through the source code.

During the first pass, the assembler will make use of the parameter <OPT 1>. So we have to decide which bits should be set, and which ones reset. Going back to our example, let us assume that we wish to assemble it straight into memory, rather than write the object code to another file. An examination of the above table should indicate that OPT 0 would be suitable for the first OPT number.

We must now decide on (<OPT 2>). The extra pair of round brackets indicate that this parameter may not be needed. However, the only occasions that we can leave it out are when we have told the assembler (with bit 8 of <OPT 1>) that a second pass is not required. Put another way, if <OPT 1> is less than &100, then the second OPT number must be given.

The second OPT number is used by the assembler during pass 2. For this example we can build it up as follows. Bits 0 and 1 can now be set, and this will give a listing of the assembled code, and report any errors such as 'No such variable'. Offset assembly is not needed, so bit 2 remains clear. We proceed in this way with the remaining bits, setting or clearing them as needed. All the remaining bits would be 0 in this case. Notice that bit 8 of (<OPT 2>) is never required; its sole function is in the <OPT 1> parameter, to indicate that (<OPT 2>) is not

needed.
In binary, our second OPT number is

```
0 0 0 0 0 0 1 1
```

i.e. OPT 3.

The full command to assemble the source file TEST1 is now

```
*ASSM  TEST1  0  3
```

(or *ASSM TEST1 0 %11)

When assembly is finished, it is possible to use

```
CALL &6000
```

to execute the machine code.

The final parameter (<dest>) is used when we wish to send the object code straight to a file. Recall that when bit 3 of OPT is set, the output from the AXR is written to file.
How would we modify the OPT values to send the output to a file called 'OBJECT' and not to RAM, and dispense with the listing? Refer back to the table of bit settings to see that the following bit patterns would be suitable.

```
Bit no.       8 76543210

First  OPT.   0 00010000   &10
Second OPT.     00011010   &1A
```

We do NOT usually send the code to the file on the first pass, since the first pass will not assemble correctly due to forward references. That is why bit 3 is zero on pass 1.
The source file TEST1 is now assembled by

```
*ASSM  TEST1  &10  &1A  OBJECT
```

The binary patterns may be used instead of hex if preferred. Remember to include the '%' prefix.

SOME FURTHER PSEUDO-OPS

The following pseudo-ops are only allowed in disc/tape files, and not in assembler programs resident in memory. The exceptions are VDU, END, IF, ELSE and PROC which of course are standard BASIC keywords.

It is often convenient to split up the source code into several files. During assembly, an instruction is then required to tell the assembler that the current input file should be closed, and another one opened.

This is done via the pseudo-op CHN, analogous to CHAIN used in BASIC. All lines in a file after a CHN statement are ignored for obvious reasons, so CHN will normally be the very last line in a file (if it appears at all.)

CHN is followed by the next filename from which the source code is to be read, e.g.

```
CHN "PROG2"
```

Example.

```
\Save this one as 'TEST1'
ORG &6000
LDA # 2
CHN "B"    \Read from the next file

\Save this one as 'B'
LDX # 7
CHN "C"    \Read from the last file

\Save this one as 'C'
LDY # %1110101
RTS
\End of program
```

The code could then be assembled exactly as before, i.e. with

```
*ASSM  TEST1  0  3
```

LST ‹expr› Control listing

This pseudo-op is ignored during pass 1. On the second pass however, it can be used to turn the listing either on or off. LST should be followed by any numeric expression. That expression is then evaluated on pass 2. If the result is zero, the listing is turned off, and if it is non-zero, it is forced on. This completely overrides the listing requirements specified by the second OPT parameter. It is useful when the source code is lengthy, but you do not want it listed out in its entirety during assembly. Examples.

```
LST 0 \Turn listing off
LST 1 \Turn listing on
```

VDU ‹expr› (,expr) Send number to VDU driver

VDU allows any sequence of bytes to be sent via OSWRCH at &FFEE. It works in much the same way as the BASIC VDU statement, the only difference being that the semi-colon form is not supported. This is because the AXR thinks that a ';' denotes a comment.

Use VDU with care. VDU 22,0 would have the effect of forcing the computer into graphics mode 0, possibly destroying the symbol table. The main purpose of allowing VDU to be used is so that the printer can be turned on and off (VDU 2 and VDU 3) during assembly. Example.

```
VDU 3      \Printer off
```

END Terminate current pass

The END directive forces the assembler to abandon the current pass. If pass 1 was in progress, then the first source file is opened, and the second pass commences. If pass 2 was already under way, then the assembler halts. Thus 'END' is useful for inserting into programs to prevent assembly from proceeding further. For example:

```
END \Stop
```

CONDITIONAL ASSEMBLY

The three pseudo-ops IF, ELSE and FI relate to conditional assembly. A single level of nesting only is allowed. The general form is

```
IF ‹condition› (comment)
block 1
ELSE (comment)
block 2
FI
```

When the computer comes to the 'IF' directive, it evaluates the condition after it. If the result is non-zero ('TRUE'), then everything in block 1 would be assembled as normal. If the expression is FALSE however (zero), then block 1 would be ignored.

The ELSE directive reverses the current assembly status. If assembly is in progress as a result of an IF expression being TRUE, then ELSE causes assembly of statements to cease.

If, on the other hand, a previous 'condition' was FALSE, then assembly will resume.

A 'FI' marks the end of a conditional block.

Two error messages associated with conditional assembly are 'Too many IF's, and 'No IF'. The first would occur if 'IF' was found twice, without a 'FI' in between. The latter would appear if 'ELSE' (say) occurred without an 'IF'. Example.

```
.test EQU 1
IF test=0
PHA

ELSE
ASL A
TAX
STX test+&70

FI
```

Here, only the ASL, TAX and STX instructions would be assembled, because 'test' does not equal zero.

MACROS

Macro calls can be made by the use of a PROC, in exactly the same way as when assembling from RAM. The question now is how to define the procedure, so that it is accessible to the source file.

There are two ways of doing this. One is to ensure that all the procedures that will be used are loaded into memory (from BASIC) before issuing the *ASSM command. This works well enough, but it would be helpful if we could write the macros (i.e. procedures) at the same time as the main code.

Any line in the source file(s) may start with a line number in the range 1-32767. If it does, then that line is inserted into main memory, just as if it had been typed in from BASIC. Procedures are therefore defined with the usual DEFPROC, but with a line number in front. Instead of being treated as a line of assembler, it is then entered into memory.

To get the idea behind this line insertion, generate the following text on the wordprocessor. Notice that multi-statement lines are now allowed, and remember that the first and last lines are only comments.

```
\Macro 'osbyte'
10 DEFPROCosbyte(a%,x%,y%)
20 LDA #a% : LDX #x% : LDY #y%
30 JSR &FFF4:ENDPROC
\End macro
```

Save this under the name TEST2, and re-enter BASIC

(*BASIC). Now type

```
*ASSM  TEST2   &100
```

Type the LIST command when assembly is complete, and you will see that the three lines representing our procedure definition are now present. Incidentally, the use of &100 as the <OPT 1> parameter meant that only one pass was performed by the assembler since bit 8 is set.

There is one very important point to bear in mind when lines are inserted into a program. The entire variable catalogue is cleared, except for the resident integers.

In practice, this means that procedure definitions should always be defined right at the start of the first program. Labels and symbols must appear afterwards. Furthermore, on pass 2, the assembler will ignore lines of the source code which begin with a line number. This avoids the undesirable effect of wiping out all variables created on pass 1!

This source code uses procedure calls to implement the OSBYTE macro. Load the TEST2 text into the wordprocessor again, and add these extra lines underneath the existing ones.

```
ORG &6000
PROCosbyte(5,2,0)
PROCosbyte(8,4,0)
PROCosbyte(138,0,ASC"A")
RTS
```

Save the entire code as TEST2, and then re-enter BASIC. Assemble the TEST2 file with the command

```
*ASSM  TEST2  0  3
```

(This assembles it into RAM, at &6000. A listing is given on the second pass.)

MESSAGES AND USER INPUT

The use of procedures extends beyond macro implementation. We may wish to print a message indicating that another disc should be inserted in the drive, or perhaps to input the number of iterations of a loop that are needed. Since a PROC in the source file hands control over to BASIC temporarily, this is very easy to do. The program below illustrates this.

```
\These lines go into memory
10 DEFPROCgetnum : CLS
20 INPUT "Input number (0-255)",num%
30 ENDPROC
\End of BASIC code
```

```
PROCgetnum
ORG &6000
LDA #num%
```

Save as TEST3, and assemble with

```
*ASSM  TEST3 &103
```

(i.e. no second pass, report errors and give a listing).

The screen now clears (CLS in line 10), and the prompt to input a number appears. Enter 7 (say) followed by RETURN. When the 'LDA #num%' statement assembles, you will see that the number 7 has been used as the operand for the LDA instruction.

The use of a procedure is the only way in which you can define a string variable, should you want to do so. EQU will only work with numeric variables.

A USEFUL MEMORY LOCATION -- &70

The AXR assembler uses memory location &70 as a 'pass counter'. During pass one, it contains zero, and on the second pass it contains 1. Do NOT attempt to alter this location directly during assembly. It is permissible however, to test the value of its contents so that we can act upon a section of the source code on only one of the two passes, not both. This prevents you from having to answer exactly the same question twice. The program below uses two-pass assembly, but only asks the user to type in a message once.

```
10 DEFPROCask
20 INPUT "Enter message",mess$ : ENDPROC
\
.passcount EQU &70
.osasci EQU &FFE3
ORG &6000
IF ?passcount=0          \First pass?
PROCask                  \Yes, ask question
FI                       \End conditional part
LDX#0
.loop LDA message,X
JSR osasci
INX
CPX #LEN(mess$)          \All printed?
BCC loop
JMP &FFE7                \New line, exit
\
.message DFS mess$       \The user input
\Finish
```

Save as TEST4, and assemble with

```
*ASSM  TEST4  0  3
```

Use CALL &6000 to see the original message reprinted on the screen.

OPERATING SYSTEM CALLS

Any line beginning with an asterisk in the source file is passed over to the operating system for processing. One use for this would be to enable the source file itself to configure the printer (baud rate etc). E.g.

```
*FX 5,2
```

Do not put a comment on the same line as the star command.

ERROR REPORTING

The AXR has been designed to make tracking down errors in the program as painless as possible.

Whenever an error is found in the source file, a message appears indicating which line of the file caused the error, the name of the file currently being read, the offending statement, and the relevant message, e.g.

```
Error at line 37 of file DATA1

LDA (&50),X

Index
```

Simply go back to the edit mode of your wordprocessor, and change the offending line.

HINT: WORDWISE PLUS owners can use the CURSOR DOWN command to advantage. In the example just given, CURSOR DOWN 37-1 would locate the exact line immediately. This simple trick only works if there are no lines that are greater than 39 characters, between the start of the text and the line in question.

MACRO LIBRARY

Macros are essentially procedures, held in the computer's memory in BBC BASIC form. If there are many macros in use, this obviously represents a certain amount of wasted space, since only one macro is called at a time. (Macro 'nesting' is in fact possible by having one procedure call another, but this capability is unlikely to be needed.)

To make the best use of available memory, the *LOAD command can come to our aid. *LOAD is a standard operating system command which enables any type of file to be loaded into

memory at a specified address. *LOAD, like any other star command, can be recognised in a source file by the Assembler ROM. The general idea is that all the procedures are written in advance (from BASIC), and then each one SAVEd to disc (the tape filing system is not really suitable for this technique, due to its speed limitations.) The source file then *LOADs each procedure (macro) into memory as and when it is needed. This is really a form of 'disc overlay'.

Herein lies the first complication. Some procedures will be longer than others, and it is essential that the computer is made to start storing its variables in memory so that the longest procedure will not corrupt them. This is done by raising the default value of LOMEM. It is at LOMEM that BASIC starts storing its variables, and it is usually set to TOP by default.

Step 1 is to write all the procedures that will be needed. The example program to illustrate this technique will use OSBYTE and OSWORD calls, and also a 'home cursor' command. Enter BASIC, and type in the three procedures below. SAVE each one to disc under the name given in line 10.

```
10 REM Macro "M.osb"
20 DEFPROCosbyte(a%,x%,y%)
30 LDA #a% : LDX #x% : LDY #y%
40 JSR &FFF4:ENDPROC

10 REM Macro "M.rclock"
20 DEFPROCreadclock(addr%)
30 LDA #1 \Read system clock
40 LDX #addr% MOD 256
50 LDY #addr% DIV 256
60 JSR &FFF1 \Call OSWORD : ENDPROC

10 REM Macro "M.curhome"
20 DEFPROChomecursor
30 LDA #30 : JSR &FFEE : ENDPROC
```

The longest of these is clearly the second one, and the *INFO command will confirm this. If PAGE is normally &1900, then the end of this procedure in memory would be about 140 bytes above this. Allowing a generous safety margin, LOMEM must be set to &1A00 to ensure that all variables are created and stored well clear of the end of this procedure. Our source program can now be entered.

```
\This 'macro' will set LOMEM when called
10 DEFPROClomem : LOMEM=&1A00 : ENDPROC
\End macro

ORG &6000
\Now set LOMEM to &1A00 on pass 1 only
\See above for details of location &70
IF ?&70=0
```

```
PROClomem
\Done
FI

.oswrch EQU &FFEE
LDA#7
JSR oswrch

\Do some OSBYTE calls
*LOAD  M.osb  1900
PROCosbyte(12,3,0)
PROCosbyte(11,25,0)

\Home the text cursor
*LOAD  M.curhome  1900
PROChomecursor

\Read the system clock into memory
*LOAD  M.rclock  1900
PROCreadclock(time)

RTS

.time
DFB 0,0,0,0,0  \Read time into these bytes
```

Save this text as TEST5, and then re-enter BASIC. Assemble it into memory with the command

```
*ASSM  TEST5  0  3
```

The point to notice is that we never had more than one procedure held in memory at a time. This means that you can build up a library of useful macros, and simply *LOAD them in as necessary. Ideally, try to plan the number of *LOADing operations to a minimum, since disc accesses carry a time overhead.

ERROR MESSAGES

This section summarises the error messages that may be issued by the assembler.

Bad delimiter

After a statement has been assembled, only certain characters may appear on the line. e.g. colon, backslash etc.

Bad line

A pseudo-op has been used illegally, or a line of code in the source file makes no sense.

Can't evaluate

This message is very unlikely to occur, and is only possible when using the AND or EOR mnemonics. It results from the way the tokenising routine in the BASIC ROM works. You can prevent this message occurring by always ensuring that there is at least one space between the mnemonic and the number. E.g. use

```
EOR 12345      rather than     EOR12345
```

Duplicate label

Multiple label definitions can be trapped by setting bit 7 of the OPT pseudo-op.

Not in BASIC

Various commands, e.g. *ONAXR, only work from BASIC.

Out of range

Relevant only to branch instructions (including BBR and BBS), this indicates that the branching distance is above the permitted maximum. Bit 5 of OPT can be used to prevent this error as described previously.

Byte/Word

Some operands cannot be greater than 255, or sometimes 65535. The following would give an error of this type.

```
LDA #4445          DFW  &776612
```

Index

An index register has been used illegally.

Missing ,

An insufficient number of parameters has been supplied, e.g.

```
BBR 5
```

Missing)

Fairly self-explanatory, e.g.

```
EOR (&43,X
```

Too many bytes

Relevant only to the DF (DeFine) pseudo-ops. A single numeric DF statement cannot generate more than 80 bytes. DFS may create a string of up to 255 characters in length.

Can't open

The computer cannot open the specified file for some reason.

Type mismatch

Expressions expecting a numeric quantity do not like strings, and vice versa!

Bad binary

Binary numbers are recognised by the Assembler ROM. There must be at least one '0' or '1' after the '%' sign, with no spaces in between.

Bad name

Filenames are restricted to 13 characters in the *ASSM command.

Bad OPTs

The OPTs specified in the *ASSM command are checked for 'suspect' numbers. In particular, the branch to jump bit (5) and the offset assembly bit (2) must be the same on both passes. If they are not, this error will occur.

Too big

The first OPT parameter in *ASSM should not exceed &1FF, and the second &FF. In fact, since it would not normally be useful to set the 'duplicate label' bit (7) on a second pass,

&7F is the sensible maximum of (<OPT 2>). The assembler actually ANDs the second OPT number with &7F anyway, to ensure that the most significant bit is clear.

ESCAPE

The ESCAPE key can be used to abandon the assembly of files on tape or disc.

Line too long

No line in a source file may exceed 235 characters.

Too many IFs

No more than one level of conditional assembly is permitted.

No IF

ELSE and FI in a source file must both have an 'IF' statement before they may be used.

Bad bit

Some of the new 65C02 instructions reference a particular memory bit. This must be specified by a number in the range 0-7 with no arithmetic expressions.

```
SMB 4,&13           is valid, but
BBR n,&13,label     is not.
```

OTHER STAR COMMANDS

This section describes the other star commands on the ROM. Like *ASSM, they may be typed in either upper or lower case letters, abbreviated with a period if desired, and optionally prefixed by 'AU' (or 'au') to avoid command clashes.

Notice also that the AXR will accept arithmetic expressions for any of its numeric parameters. Such expressions must not include any BASIC keywords however, because keywords are not tokenised in star commands.

*BITS

*BITS (or *AUBITS) prints out a table illustrating the meaning of the various bits of the OPT directive. It is intended to be a brief guide as to how to use OPT, rather than a detailed description. Remember that bit 6 is not used during assembly from file, and bit 8 is not used when assembling from memory.

*LVAR

The Assembler ROM contains a variable dumping utility. This can be very handy for checking that labels and symbols have the values intended. In response to

```
*LVAR
```

the computer prints out the values of all currently defined variables in both decimal and hex. String variables are also listed, with 'unprintable' characters replaced by a period.

This command may not work correctly if memory is very scarce, as result of say

```
DIM A 100000
```

*DSM

Syntax: *DSM (<address>) (<ROM>)

The DSM facility has been provided to complement the assembler section of the ROM. The command may be used to disassemble any section of the computer's memory, including sideways ROMs. The disassembler recognises all the extra instructions available on the enhanced 6502 processors.

There are two optional numeric arguments with DSM. Both

may be entered in either decimal or hex (preceded by a '&'). The first is the starting address in memory from where disassembly is to commence. If no address is specified, then the default value is zero (&0000). There is no finishing address required, and the program will proceed until ESCAPE is pressed. Of course, certain keys will determine the path that the disassembler will follow, and these are described later.

The second parameter should be a quantity between 0 and 15 decimal. It represents a socket number inside the computer, and is used to decide which of the sideways ROMs is to be disassembled. This parameter is only relevant if the memory addresses being disassembled are in the range &8000-&BFFF, and its default value is 15. Disassembly of the Assembler ROM itself cannot be performed.

Unrecognised opcodes are denoted by '---' . The ASCII characters corresponding to the various numbers are shown on the extreme right hand side of the screen, so it is relatively simple to spot text or messages.

RTS (ReTurn from Subroutine) instructions are highlighted by the appearance of a '&' symbol in the middle of the screen. This facility can be helpful when looking for the end of subroutines, but you should remember that subroutines may terminate with a JMP instruction, e.g. JMP &FFF4

A number of powerful subroutines exist within the MOS, available to both BASIC and machine code programmers alike. Standard operating system calls are identified and labelled by name. The recognised calls are as follows:

Routine	Address	Routine	Address	Routine	Address
OSBYTE	FFF4	OSNEWL	FFE7	OSWORD	FFF1
OSCLI	FFF7	OSARGS	FFDA	OSBGET	FFD7
OSBPUT	FFD4	OSWRCH	FFEE	OSASCI	FFE3
OSRDRM	FFB9	OSGBPB	FFD1	OSFIND	FFCE
OSINIT	FFC2	OSREAD	FFC5	OSRDCH	FFE0
OSEVEN	FFBF	OSFILE	FFDD		

Similarly, the standard vectors in page two are also labelled. E.g. JMP (&20E) is identified as WRCHV.

Various keys alter the way in which the disassembler operates. In addition to the 'single-key' types, most of the 'control' characters perform their usual function, viz:

Ctrl-B: Simultaneously pressing the control key and B will send all subsequent output to a printer as well as the screen. Depending on how your printer is configured, it may be necessary to issue certain FX calls before this will work properly (e.g. to select the correct baud rate). The User Guide provides more information on doing this.

Ctrl-C: Turn printer off.

Ctrl-N: Set 'paged' mode on. Under these conditions, output stops after a certain number of lines have been printed. Pressing the SHIFT key will allow a further screenful of lines to be displayed.

Ctrl-O: Set paged mode off. This reverses the effect of Ctrl-N.

The other keys related to the disassembler are:

A - Pressing 'A' will prompt for a new disassembly address and sideways ROM. Since it is most likely that an address will be entered in hex, the computer provides the necessary '&' symbol automatically. This can be removed if necessary with the DELETE key. The ROM number should be in the usual range 0-15. One or more spaces must of course separate the two quantities. Pressing <ESCAPE> at this point will terminate the disassembler.

B - Back one byte. The instruction at the current address less one is disassembled.

C - Continuous disassembly. Memory will be disassembled continuously, much too rapidly to read. This is the obvious mode to use when dumping any appreciable quantity of code to a printer. Pressing CTRL and SHIFT together will halt the scrolling process.

J - Jumps on. Pressing 'J' will cause any subsequent unconditional branch to be followed. This covers subroutine calls and jump instructions, both direct and indirect. Conditional branches have no effect. If an RTS is encountered in this mode and a subroutine is currently being examined, then disassembly will return to the instruction after the appropriate 'JSR'.

O - Jumps Off. This turns off the jump-following facility.

R - Force Return from subroutine. If disassembly has proceeded into a subroutine, pressing 'R' will immediately cause a return to the point after the calling 'JSR' instruction. If no subroutine is being followed, the next instruction in sequence will simply be disassembled. The limit to the number of JSR instructions that can be 'remembered' (or nested) is 127.

S - Step-wise disassembly. This reverses the C (continuous) mode. Under these circumstances one instruction is disassembled every time a key is pressed. This facilitates stepping through the path that a program is assumed to follow.

<SPACE> The space bar can be used to follow the course defined by a conditional branch instruction. If the instruction just disassembled was any kind of conditional branch, then pressing <SPACE> will cause that branch to occur. This is true irrespective of whether the 'jumps' mode is on or off.

If the jumps-mode is off, then pressing the space bar will also cause subroutines and jump instructions to be followed. Additionally, if an RTS was the last instruction, it attempts to return from the subroutine by retrieving the necessary address. If there is no JSR pending, the next instruction in sequence is disassembled.

<SPACE> will temporarily suppress the effect of 'jumps-on'. If unconditional jumps are being followed (by the use of 'J' - see above), and a 'JSR' or 'JMP' is decoded, then pressing the space bar will stop the jump from taking place. This prevents operating system calls such as the ones mentioned previously from being disassembled. There is nothing wrong with examining the subroutines in the MOS, but it is very time consuming since most of them ramble about in a rather opaque way, and so it is not always very revealing.

? - This key is only relevant to the new JMP (Abs,X) instruction. The effective jump address that the processor will ultimately use depends not only on the absolute address given in the instruction, but also on the contents of the X register at the time. Pressing '?' after an indexed indirect jump will cause the computer to prompt

X=

Type in the value required and press RETURN. Only the least significant byte of the number entered will be made use of. The computer will calculate the same address that the processor would have done, and disassembly recommences from there. This provides a simple way of checking that jump tables have been formed correctly.

<ESCAPE> - Exit the disassembler.

COMMAND KEY SUMMARY

A - New Address/ROM
B - Back one byte
C - Continuous
J - Jumps on
O - Jumps off
R - Return from subroutine
S - Step-wise disassembly
? - Input X for indexed indirect jump
<SPACE> - Invert jump action

ESCAPE - Finish

*STRIP

Syntax: *STRIP <fsp> (<:)>

Many users like to use a wordprocessor for editing BASIC programs. The way this is done is normally to use *SPOOL (from BASIC) followed by a filename. This opens a spool file for output. The program in memory is then listed with the LIST command. When a spool file is active, all output to the screen also goes to the named file. When listing is complete, the use of *SPOOL on its own will close the spool file. The net effect of all this is that the given file can now be edited with a wordprocessor. You cannot successfully load BASIC programs straight into WORDWISE (say), because of the BASIC tokens.

This procedure is satisfactory to some extent, but the line numbers are often a pest. *STRIP will send your BASIC program to a file in ASCII format (detokenised), and you can then load it into the wordprocessor. Line numbers will have been removed however. In addition, the statements

```
NEW
AU.          (short for AUTO)
```

will be placed right at the start of the file. This means that you can just EXEC the file back into memory from BASIC if you want to, and have the line numbers put back again automatically.

*STRIP itself should be used from BASIC. You may optionally place a ':' symbol after the filename <fsp>, and this has the effect of splitting up multi-statement lines. Remember that multi-statement lines are not permitted when assembling directly from a file, so this facility turns out to be quite useful.

Examples

```
*STRIP  data1
*STRIP  MYFILE  :
```

The second example takes the program in memory, and spools it out under the name 'MYFILE', but without any line numbers. It also splits up multi-statements. If you now type

```
*EXEC  MYFILE      <RETURN>
```

you will obtain your original program, but with no multi-statement lines present.

THE EXTRA INSTRUCTIONS

This section describes instructions that are not available on the standard 6502 fitted to the model B computer. The AXR will assemble all the instructions listed here, which means that code can be produced for execution on the second processor, Master series, or any other 6502 based systems. There can be a speed bonus and memory saving by the use of these instructions, but there is a penalty in terms of lack of transportability of code from one machine to another.

Mnemonic	Addressing mode	
PHY	Implied	Push register Y
PLY	Implied	Pull register Y
PHX	Implied	Push register X
PLX	Implied	Pull register X

It is now no longer necessary for stack operations involving X and Y to go through the accumulator.

DEC A	Accumulator	Subtract 1 from the A register
INC A	Accumulator	Add 1 to the A register
ORA (zp)	Indirect	OR accumulator
AND (zp)	Indirect	AND accumulator
EOR (zp)	Indirect	EOR accumulator
ADC (zp)	Indirect	Add to accumulator
STA (zp)	Indirect	Store accumulator
LDA (zp)	Indirect	Load accumulator
CMP (zp)	Indirect	Compare accumulator
SBC (zp)	Indirect	Subtract from accumulator

zp is a zero-page location. These instructions operate like indirect indexed addressing with the Y register set to zero.

SMB n,zp	Zero page	Set bit n of location zp
RMB n,zp	Zero page	Reset bit n of location zp

zp is a zero-page location. n is a 3 bit number (0-7) and cannot be a variable.
E.g. negflag=&83 : SMB 7,negflag

STZ abs	Absolute
STZ abs,X	Absolute indexed
STZ zp	Zero page
STZ zp,X	Zero page indexed

Store a zero in the specified memory location.

```
BIT zp,X      Zero page indexed
BIT abs,X     Absolute indexed
BIT #imm      Immediate (V flag not affected)
BRA label     Relative
```

The BIT test with new addressing modes, and also an unconditional branch.

```
JMP (abs,X)  Indexed indirect jump

TSB abs      Absolute
TSB zp       Zero page
TRB abs      Absolute
TRB zp       Zero page
```

Test and Set/Reset memory bits

```
BBR n,zp,label              Zero page
BBS n,zp,label              Zero page
```

Branch on bits Reset/Set
E.g. BBS 3,&70,label

There is a program on the AXR ROM itself which illustrates these mnemonics being used. This program and others are described in the next section.

PROGRAMS ON THE AXR

The AXR contains a number of demonstration programs. They are present in ROM Filing System (RFS) format.

You will need to issue a *ROM command in order to activate the RFS. Once this has been done, the ROM may be catalogued by means of *CAT (or *.).

All the programs except "TABLE" are in BBC BASIC form, and each one may be loaded into memory by means of the familiar LOAD command.

In order to keep the size of the programs to a minimum, extensive use is made of multi-statement lines. Remember that *STRIP can be used to convert the programs to a more legible form. For example, to split up the program "DEM3" on the ROM, you could use the following steps. It is assumed here that a disc drive is present, but the principle is applicable to tapes.

```
*ROM                          <RETURN>
LOAD  "DEM3"                  <RETURN>
*DISC                         <RETURN>
*STRIP  ANYFILE  :            <RETURN>
*EXEC  ANYFILE                <RETURN>
```

There will now be a 'split up' version of DEM3 in memory, which can be resaved to disc if needed.

The following sections give a brief description of the purpose of each program.

The programs were written by Mike Ginns.

DEM1

Purpose: To illustrate duplicate label trapping

There is not much that needs to be said here. Recall that setting bit 7 of OPT causes a 'Duplicate label' error if an attempt is made to define a label that already exists. This program just goes through the assembly language twice, with bit 7 set on the second pass.

DEM2

This program provides a difficult test for the ROM, by checking the operation of recursive macro calls. The resulting code itself is useless!

DEM3

This program contains all the instructions and addressing modes supported by the 65C02 microprocessor. When the program is run, the following occurs.

a) The code is written directly to RAM, then *SAVEd to disc.
b) The code is re-assembled directly to another file.
c) The files are compared to make sure that they are the same.

This test serves two main purposes. First, it ensures that all 65C02 instructions do assemble. Second, it confirms that assembly to RAM and directly to disc produces identical object code.

DEM4

DEM4 tests the ROM's ability to assemble code with an address offset. It generates code to run in sideways RAM, but places it in memory from location &3000 onwards. This block of memory is *SAVEd to disc, then reloaded into sideways RAM. If the code has assembled correctly, then a sideways 'ROM' will be produced which simply reports its existence on BREAK.

N.B. This program obviously needs sideways RAM! Furthermore, it is necessary to be able to *LOAD data directly into the S/W RAM at &8000 for this simple routine to work. Alternatively, the software that accompanied your sideways RAM should explain how to move a ROM image from disc to the sideways RAM itself.

SIDE

This is a general purpose sideways ROM generator. The program is basically a series of conditional assembly clauses which allow the user to very easily define the type of ROM that he or she requires. After configuring a copy of the program for a particular ROM type, all that is required is to insert the assembly language routines which make up the ROM at the marked places. Running the program will then create a disc file containing an image of the user's ROM ready for loading into sideways RAM or blowing into an EPROM.

The variable 'save$' should be set to the name of the file under which the ROM image is to be saved to disc.

Next there are some variables which define the ROM's header code information.

language - TRUE or FALSE
Has the ROM got a language entry point?

service - TRUE or FALSE
Has the ROM got a service entry point - it normally should.

version - 0-255
The binary version number of the ROM.

title$
A string giving the ROM's title.

version$
An optional version string containing the ROM's version message.

copyright$
The name of the person who holds copyright on the ROM. Notice that the '(C)' is included automatically.

The next section (lines 60-70) contains twenty flag variables which should be set to either TRUE or FALSE. These indicate whether the corresponding service is to be provided by the ROM. For instance, you would set ' _command=TRUE ' if you wanted to write a 'star' command type of ROM.

The last section (line 370) contains a series of labels. There is one label for every service type that the ROM could provide, If a ROM provides one of these services then the user code to implement it should be placed after the appropriate label. (You will almost certainly need to use *STRIP to split this program up, as described previously). Each routine should end with an RTS instruction.

Notice that user routines are responsible for preserving the necessary registers, and for setting the accumulator to zero on exit if applicable.

At the end of the service entry points is the '.language' label. This marks the entry to the ROM if it is specified as being a language.

TABLE/TBL_DEM

This is an ASCII file and should be EXEC'ed onto the end of the user's program.

An extra line containing the label definition '.label' should first of all be added at the end of your program. Then, the 'TABLE' file is EXEC'ed into memory, thereby tacking the procedure onto the end of the current program. 'label' actually marks where the user's data tables are to be stored.

to create a table in the program simply make a call to the table procedure. To do this a number of parameters have to be passed. These are;

1) pass
The first parameter should be the value of the current assembly pass. The table procedure requires two-pass assembly and creates the data tables when the value of pass is non-zero.

2) size
This parameter defines the size of the table in bytes that is to be created.

3) pointer
This is the zero-page address which is to be used as a pointer to the data table. The procedure assembles code to load these two locations with the first address in the data table. The pointer can then be used to index into the table as required. If an indirect pointer is not needed then setting this parameter to -1 suppresses the code output.

4) content$
This is a string parameter. It defines what the contents of the data table are to be. This is done by giving a numeric expression in the string which, when evaluated, gives the data for a particular entry in the table. The variable 'location' may be used in the expression to define data which is dependent on the data's offset position in the table. This will almost always be the case. The following examples should make this clear;

"location*3" - A three times table will be created
i.e. 0,3,6,9...

"3^location" - A sequence of cubic powers will be created
i.e. 1,3,9,27...

The expression need not be based on the entry offset, e.g.

"RND(255)" - A table of random numbers.

The data specified is written into memory using the pling operator (!). This means that data larger than 256 is encoded in the normal lo-byte, hi-byte format. It should also be remembered that the expression given is limited to data which can be written to memory with a '!'. For example, giving the 'content$' parameter as

"SIN(location)"

would not work as the real value of SINE is never greater than 1, and a table of 0's would result. It is often possible to get round the problem by e.g.

"1000*SIN(location)"

5) increment
As the expression given in parameter 4 may define data which is larger than one byte, it is necessary to tell the procedure how many bytes each table entry is.. For example, a table of 6502 addresses needs two bytes per entry and so 'increment' is set to 2.

TBL_DEM

The file 'TBL_DEM' is a demonstration of the TABLE procedure at work. It creates three tables; one contains the alphabet, and another two contain the (X,Y) co-ordinates of a sine graph. These illustrate how the procedure may be used in practice.

NOTES

NOTES

NOTES