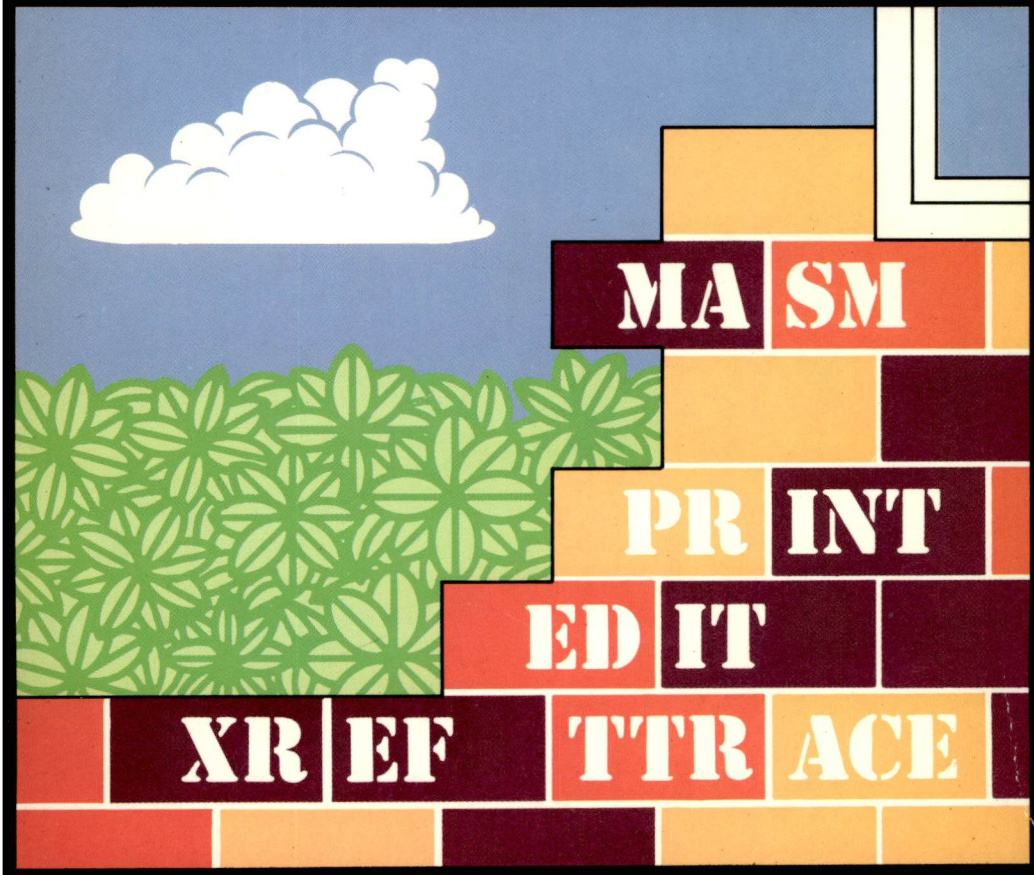
6502 Development Package

on the BBC Microcomputer with 6502 Second Processor

BARRY MORRELL



6502 Development Package

on the BBC Microcomputer with 6502 Second Processor

BARRY MORRELL

ACORNSOFT

Acknowledgements

The 6502 Development Package was developed by Jon Thackray. Thanks are also due to Pete Cockrell and David Christensen.

Barry Morrell

First published in 1984 by Acornsoft Limited

Note: Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

FIRST EDITION

Acornsoft Limited, Betjeman House, 104 Hills Road,
Cambridge CB2 1LQ, England. Telephone (0223) 316039

CONTENTS

Preface

This user guide is intended for owners of the 6502 Development Package. It describes how to use the following utilities: EDIT, MASM, PRINT, XREF, SRCXREF and TTRACE (as well as PR and BTRACE - versions of PRINT and TTRACE for the I/O processor). Together, these utilities and the 6502 Second Processor provide a powerful means of producing machine code programs for use with any 6502-based machine, particularly a BBC Microcomputer Model A or B, or an Acorn Electron.

Before reading this book you should be familiar with the concepts covered in the *BBC Microcomputer System User Guide*. You should also be reasonably familiar with the standard set of 6502 assembler mnemonics.

If you are not familiar with 6502 assembler, the following books may help you:

Creative Assembler for the BBC Microcomputer Model B and Acorn Electron by Jonathan Griffiths, a Penguin publication

Assembly Language Programming for the BBC Microcomputer by Ian Birnbaum, a Macmillan publication

The approach adopted within this book is to give you 'hands-on experience' of MASM and its associated utilities as quickly as possible: this is vital if you are to appreciate the range of facilities available. By the time you have finished reading the book you will be familiar with most of the facilities of MASM. After this, you will find the *6502 Development Package Reference Card* of use.

1 Getting under way

The 6502 Development Package is designed for use on the BBC Microcomputer Model B; it will not work upon any other type of microcomputer. You should also have the following equipment:

- A 6502 Second Processor
- Disc drive(s)

As well as this book, the Development Package consists of:

- A copy of the *6502 Development Package Reference Card*
- A floppy disc labelled *6502 Development Package*
- A function key card

If any of these is missing you should contact your Acorn dealer.

Before you go any further, you should start up your system and take a security copy of the disc. The procedure, which uses *ENABLE and *BACKUP, is described in your *Disc Filing System User Guide*. You are reminded that copyright exists in this software and that you may only make a single security copy for your own use.

Next, insert your copy of the disc in a drive and look at the files on it (*CAT command). They should be as follows:

MASM	A macro assembler
XREF	A MASM-dependent cross-referencer
SRCXREF	A free-standing cross-referencer
IOMASM	MASM code in the I/O processor
TTRACE	A trace package for the second processor
BTRACE	A trace package for the BBC Model B
PRINT	A print utility for the second processor
PR	A print utility for the BBC Model B
EDIT	A text editor

If any of these files are missing you should contact your Acorn dealer.

Finally, you should slip the function key card provided under the clear plastic slip above the red function keys on your keyboard, lining it up so that GOTO LINE comes immediately above the red function key 'f0'. This legend will be your guide when you are inputting and amending your programs using EDIT.

2 Developing programs with the Second Processor

With the 6502 Second Processor, you can develop sophisticated programs in MASM assembler. These can then be run in any 6502-based machine, but particularly either your BBC Microcomputer Model B or the 6502 Second Processor.

You can, of course, develop programs in other languages using the Second Processor.

2.1 What is MASM?

MASM is a macro assembler and forms the main part of this package. For the present we will forget about the macro and concentrate on the assembler part of the name.

Most computer languages are high level languages; BASIC, FORTRAN, Pascal and COBOL are examples. They are generally aimed at particular areas of use; for example, FORTRAN is designed to aid in the solution of mathematical and scientific problems. Because of this, they are often highly-specialised and may be unsuitable for some functions. Additionally programs in languages like BASIC have their instructions interpreted every time the program is run, so they can be slow.

Machine code, on the other hand, is extremely efficient in operation and does not need processing by an interpreter every time it is run. However, it is not efficient from the point of view of time spent programming in it.

Assembly language offers a compromise between these two approaches. Instructions in assembly language have an almost one-to-one correspondence with machine code. They also have mnemonics which are easier to recognise than machine code (though not as easy as high level language instructions). In addition, programs written in assembly language do not have to be translated from mnemonics into machine code every time they are run. Instead, the translation is done once, by an assembler such as MASM. From this stage onwards, the machine code object program can be loaded and run as many times as required.

Let's look at the way assemblers such as MASM work before going any further. Below, on the left, is a sample of MASM assembly code (the source code); at the right of it is the equivalent machine code and some typical memory addresses:

```
      LDXIM     1
loop  INX
      CPXIM     100

      BNE       loop
```

&2000	A2
	1
	E8
	E0
	64
	D0
&2006	FB

To produce the code on the right from that on the left, MASM scans through the program from start to finish and builds up a symbol table. MASM makes a note, in its symbol table, of the position of each label (such as 'loop') so that, when it encounters the label in subsequent instructions, it can insert an actual address in the instruction (or at least a representation of the actual address).

This works fine for pieces of code like the one shown above. However, what happens in the case of the code shown below, where the address 'server' is further down the program than the instruction that uses it?

```
            ....
      LDXIM     100

      JSR       server
            ....
            ....
            ....
server      ....
            ....
```

&2000	A2
	64
	20
	00
	30
	.
&3000	.
	.

In this case, the assembler has no address for 'server' when it encounters the instruction. 'server' is called a forward reference.

MASM, like most assemblers, resolves this problem by always making two passes through the program source code. During the first pass, a symbol table is built up; during the second pass, the instructions are converted into object code.

The actual process of assembly is more complicated than the way we have described it, but this description should suffice for the present. If you want to know more about how assemblers work, *Assemblers and Loaders*, by D W Barron might be of interest.

3 Developing a simple MASM program

There are four main stages to writing a program and getting it working using the 6502 Development Package:

- Designing the program structure
- Coding the program source (using the EDIT utility)
- Assembling the program (using MASM) and running it
- Debugging the program (using PRINT to obtain the necessary listings and a TRACE utility).

It is assumed that the user is familiar with designing the structure of a program, since the method is similar for all programming languages. Each of the subsequent stages will be explained in this chapter by getting you to develop a simple program yourself, using the 6502 Development Package and your computer.

3.1 Creating program source using EDIT

The EDIT program helps you to create and amend program source files. It is described fully in chapter 11, 'Creating source files using EDIT', but for now only a simple introduction is given so that you can create a small source file.

To run the editor, insert the program disc that comes with the package into the current drive, type:

```
*EDIT
```

and press RETURN. This will cause the editor to be entered with no text, so that the screen is almost blank. The only text is an inverse video asterisk ('*') at the top left hand corner and a couple of words on the bottom line. The asterisk is the 'end of file marker' and it tells you where the end of the text is. The bottom line is called the 'status line', since it gives various pieces of information about the editor.

To add text when you are in the editor you simply have to type it. The end of text marker will be moved along to make room. If you make a mistake simply press the DELETE key as usual. At the end of a line press the RETURN key and the flashing cursor will move onto the next line.

The arrow keys are used to move around the screen (though naturally you can't move below the bottom line or above the top one). Armed with this information, you should be able to type in the short example program listed below:

```
zerop  *     &70
       ORG   &1900
test   LDAIM 0
       STAZ  zerop
       LDAIM fin / &100 + 1
       STAZ  zerop+1
       LDYIM 0
loop   LDAIY zerop
       PHA
       LDAIM &AA
       STAIY zerop
       CMPIY zerop
       BNE   error
       LDAIM &55
       STAIY zerop
       CMPIY zerop
       BNE   error
       PLA
       STAIY zerop
       INY
       BNE   loop
       INCZ  zerop+1
       LDAZ  zerop+1
       CMPIM &80
       BNE   loop
       BRK
       =     1
       =     "MEMORY OK"
       =     0
error  BRK
       =     2
       =     "MEMORY FAULT"
       =     0
fin
       END
```

Note that a carriage return is necessary at the end of the source code (after the END directive) otherwise a '`Line too long`' error will be generated.

When you are happy that the text on the screen is as shown above you should save the file on disc. To do this, press the function key marked 'f3'. This will produce a prompt to which you should reply with the name of the file in which the text must be saved. Let's use the name 'TEST' for this example. Press RETURN after the filename; the text will be saved. Note that the disc supplied with the pack has a write-protect label on it to prevent accidental erasure of any files. A separate disc should be used for your own programs.

Note that it does not matter what case the labels and opcodes are in, but the code is easier to read with them as shown. The exact number of spaces between items is also not critical, so long as there is at least one.

3.2 Assembling the program and running it

Now that you have created a program source file you will want to assemble it. You can do this by issuing an operating system command to call the assembler. When in EDIT, press the function key 'f1'. This will prompt you with an asterisk, implying that you should type an OS command. To call the assembler, type:

```
MASM
```

then press RETURN. It is possible to issue 'star' commands from all of the programs in the package. For all of them apart from EDIT, you simply type the command prefixed by an asterisk, just as you would in BASIC. You can use operating system commands such as *CAT, *COPY and *DELETE in command mode.

You will enter the command mode of the MASM utility and the following prompt will be displayed:

```
Action:
```

To assemble your program type:

```
ASM TEST
```

then press the RETURN key. MASM will print the following prompt:

```
Macro library:
```

You can ignore this for the present; its significance will be described later. Merely press the RETURN key.

MASM will now do two passes through your program source, as described in chapter 2, 'Developing programs with the Second Processor'; it will tell you when it finishes each pass. When the assembly is complete, it will print the following message on your display then return to command mode

```
Assembly finished, no errors
Action:
```

If you don't get this message, go back to EDIT to repair your source. The information you need is described in chapter 11, 'Creating source files using EDIT'.

Now, assuming there were no errors, your program will have been assembled into a file with the same name as your source file, but it will be in the directory X. In the present case, the file will have the file specification X.TEST and you can run the program by typing

```
*X.TEST
```

then pressing RETURN. Do this now; the program will display the following message:

```
MEMORY OK
```

3.3 Producing a listing of your program

If your program did not work, you would want a listing of it to help you debug it. You would also want a listing even if it did work, to use as part of your program documentation.

You can produce the listing using the PRINT utility. First of all, ensure that your printer is connected according to the instructions in the BBC Microcomputer System User Guide. Now press CTRL B to enable it and then type the following:

```
*PRINT
```

You will get the following prompt:

```
File name:
```

and you should type TEST then press the RETURN key. Next, you will be given the prompt:

```
Parameters:
```

and you should reply by typing:

```
W80 L66 N
```

then pressing the RETURN key. Your program will now be printed. To disable the printer you should press CTRL C.

Note that some ROMs will respond to a *PRINT command, and if you encounter this problem it will be necessary to use */PRINT or *RUN PRINT to load the utility from disc.

You can also produce an assembly listing of your program using MASM, but this is only possible if the first pass was successful.

3.4 Program development

Of course, program development is not as simple as in the example given. In reality you will make mistakes and the program will not work first time, unless it is a very simple one! When this happens you will need debugging tools beyond a mere PRINT utility.

The 6502 Development Package gives you some of these tools, but we will not describe them yet. Later, in chapter 14, 'Debugging your programs', we will introduce some bugs into the program that you developed in this chapter. We will then use the debugging tools to remove them. For the present, we will forget about them and look a bit deeper into the 6502 itself and other parts of the Development Package.

4 The building bricks of MASM

The first section of this chapter describes the mnemonics of MASM instructions and their related address modes. The second section describes the format of MASM source code. Finally, the MASM operators, which handle arithmetic and logical operations (for example), are covered.

4.1 MASM mnemonics and address modes

Table 4.1 shows the 6502 instructions and their equivalent MASM mnemonics. These are similar, but the MASM mnemonics have additional characters that indicate their address mode.

Mnemonic	ADC	AND	ASL	BCC*	BCS*	BEQ*	BIT
Immed.	ADCIM	ANDIM					†*BITIM*
Abs.	ADC	AND	ASL				BIT
ZeroPage	ADCZ	ANDZ	ASLZ				BITZ
Abs,X	ADCAX	ANDAX	ASLAX				†*BITAX*
Abs,Y	ADCAY	ANDAY					
Z,X	ADCZX	ANDZX	ASLZX				†*BITZX*
Z,Y							
(Z,X)	ADCIX	ANDIX					
(Z),Y	ADCIY	ADCIY					
Accum.			ASLA				
(Ind.)	†*ADCI*	†*ANDI*					

Mnemonic	BMI*	BNE*	BPL*	†*BRA**	BRK*	BVC*	BVS*

Standard 6502 Mnemonics

Mnemonic	CLC*	CLD*	CLI*	†*CLR**	CLV	CMP	CPX
Immed.						CMPIM	CPXIM
Abs.				†*CLR*		CMP	CPX
ZeroPage				†*CLRZ*		CMPZ	CPXZ
Abs,X				†*CLRAX*		CMPAX	
Abs,Y						CMPAY	
Z,X				†*CLRZX*		CMPZX	
Z,Y							
(Z,X)						CMPIX	
(Z),Y						CMPIY	
Accum.							
(Ind.)						†*CMPI*	

Mnemonic	CPY	DEC	DEX*	DEY*	EOR	INC	INX*
Immed.	CPYIM				EORIM		
Abs.	CPY	DEC			EOR	INC	
ZeroPage	CPYZ	DECZ			EORZ	INCZ	
Abs,X		DECAX			EORAX	INCAX	
Abs,Y					EORAY		
Z,X		DECZX			EORZX	INCZX	
Z,Y							
(Z,X)					EORIX		
(Z),Y					EORIY		
Accum.		†*DECA*				†*INCA*	
(Ind.)					†*EORI*		

Mnemonic	INY*	JMP	JSR	LDA	LDX	LDY	LSR
Immed.				LDAIM	LDXIM	LDYIM	
Abs.		JMP	JSR	LDA	LDX	LDY	LSR
ZeroPage				LDAZ	LDXZ	LDYZ	LSRZ
Abs,X				LDAAX		LDYAX	LSRAX
Abs,Y				LDAAY	LDXAY		
Z,X				LDAZX		LDYZX	LSRZX
Z,Y					LDXZY		
(Z,X)		†*JMIX*		LDAIX			
(Z),Y				LDAIY			
Accum.							LSRA
(Ind.)		JMI		†*LDAI*			

*These instructions have the standard mnemonics in implied or relative addressing mode.

†*Mnemonics in italics are available only with CMOS processors.*

Mnemonic	NOP*	ORA	PHA*	PHP*	†*PHX**	†*PHY**	PLA*
Immed.		ORAIM					
Abs.		ORA					
ZeroPage		ORAZ					
Abs,X		ORAAX					
Abs,Y		ORAAY					
Z,X		ORAZX					
A,Y							
(Z,X)		ORAIX					
(Z),Y		ORAIY					
Accum.							
(Ind.)		†*ORAI*					

Mnemonic	PLP*	†*PLX**	†*PLY**	ROL	ROR	RTI*	RTS*
Immed.							
Abs.				ROL	ROR		
ZeroPage				ROLZ	RORZ		
Abs,X				ROLAX	RORAX		
Abs,Y							
Z,X				ROLZX	RORZX		
Z,Y							
(Z,X)							
(Z),Y							
Accum.				ROLA	RORA		
(Ind.)							

Mnemonic	SBC	SEC*	SED*	SEI*	STA	STX	STY
Immed.	SBCIM						
Abs.	SBC				STA	STX	STY
ZeroPage	SBCZ				STAX	STXZ	STYZ
Abs,X	SBCAX				STAAX		
Abs,Y	SBCAY				STAAY		
Z,X	SBCZX				STAZX		STYZX
Z,Y						STXZY	
(Z,X)	SBCIX				STAIX		
(Z),Y	SBCIY				STAIY		
Accum.							
(Ind.)	†*SBCI*				†*STAI*		

Mnemonic	†*STZ*	TAX*	TAY*	†*TRB*	†*TSB*	TSX*	TXA*
Immed.							
Abs.	†*STZ*			†*TRB*	†*TSB*		
ZeroPage	†*STZZ*			†*TRBZ*	†*TSBZ*		
Abs,X	†*STZAX*						
Abs,Y							
Z,X	†*STZZX*						
Z,Y							
(Z,X)							
(Z),Y							
Accum.							
(Ind.)							

Mnemonic	TXS*	TYA*

Standard 6502 Mnemonics

*These instructions have the standard mnemonics in implied or relative addressing mode.

†*Mnemonics in italics are available only with CMOS processors.*

Table 4.1 MASM mnemonics

The rest of this section briefly describes the address modes that you can use with the BBC Microcomputer. In this description, examples of the appropriate MASM mnemonics are given and the equivalent 6502 instruction is given alongside it in brackets, for example: RORA (ROR).

4.1.1 Implied addressing

In this address mode, the address is implicitly defined by the operation code of the instruction, for example, INX, INY, CLC and SEC. All implied address instructions consist of one byte, and are the same as in the BBC BASIC assembler.

4.1.2 Accumulator addressing

Instructions in this address mode consist of one byte and involve operations upon the accumulator. Examples of these instructions are ROLA (ROL) and RORA (ROR). The mnemonics are the same in MASM and in the BBC BASIC assembler.

4.1.3 Immediate addressing

In this type of addressing, the information to be accessed is held in the second byte of the instruction. Examples of instructions that are used in this mode are SBCIM (SBC) and ADCIM (ADC).

4.1.4 Absolute addressing

In this address mode the second and third bytes of the instruction point to the argument address: the second byte points to the low order byte of the address and the third byte points to the high order byte. You can access the entire 64K bytes of addressable memory in this mode.

Note that absolute addressing is automatically truncated to zero page addressing if the argument is in zero page.

STA, STX and STY are examples of mnemonics for absolute addressing; they are the same in MASM and the BBC BASIC assembler.

4.1.5 Zero page addressing

This address mode gives a shorter execution time than absolute addressing and uses only two bytes. It accesses the first page of memory. Zero page addressing instructions include STAZ (STA) and STXZ (STX). Note, however, that MASM automatically truncates absolute addressing to zero page addressing if the address given is in zero page.

4.1.6 Absolute indexed addressing

This is used with the X and Y index registers. The two forms are also called 'Absolute, X', 'Absolute, Y' and are shown as 'Abs, X' and 'Abs, Y' in Table 4.1. The instructions used have three bytes and the target address is formed by adding the contents of either X or Y and the address in the second and third bytes.

You can use this type of addressing to access tables by putting the base address as the second and third byte of the instruction then using the X or Y register as a displacement pointer.

Examples of absolute indexed addressing are shown below on the left in MASM code. Their equivalents in BBC BASIC assembler are shown on the right.

```
        LDAAX   tabst                   LDA   tabst,X
        STAAY   tabst                   STA   tabst,Y
```

Note that code such as LDAAX &70 is truncated by MASM to LDAZX &70. This means that it is not possible to assemble code such as LDAAX &70 without resorting to the = directive (see section 5.2). This problem will not normally be encountered, except in programs such as:

```
loop    LDAAX &100
        STAAX &FF
        DEX
        BNE loop
```

which will not do what is expected if assembled in MASM.

4.1.7 Zero page indexed addressing

This is similar to absolute indexed addressing but the target addresses are in page zero. The instructions themselves are two bytes long and the last byte is the base address of the area to be accessed. The contents of this byte are added to the contents of the X or Y register to give the target address.

The two forms of this address mode are also called 'Zero Page, X' and 'Zero Page, Y', and are shown as 'Z,X' and 'Z,Y' in Table 4.1. Examples of MASM code using this mode are given below on the left. Their equivalent in BBC BASIC assembler is shown alongside them.

```
tabst *         &75                 tabst=&75
          ......                        ......
          ......                        ......
        LDAZX   tabst               LDA tabst,X
        STAZX   tabst               STA tabst,X
        STXZY   tabst               STX tabst,Y
```

4.1.8 Relative addressing

This can only be used with branch instructions and these are two bytes long. The second byte is the displacement of the target address from the instruction after the branch instruction: negative values are backwards jumps, positive values are forward jumps. The maximum jump is 128 bytes backwards or 127 bytes forwards.

Relative addressing instructions have the same mnemonics in both MASM and BBC BASIC assembler.

4.1.9 Zero page indexed indirect addressing

This is also called 'Indirect, X addressing' and is shown as '(Z,X)' in Table 4.1. Here, the target address is held in a location in page zero and accessed 'indirectly' through this location.

The instructions used have two bytes and their second byte is added to the contents of the X index register to give an address in page zero. This address contains the low order byte of the target address and the next location in page zero contains the high order byte of the target address.

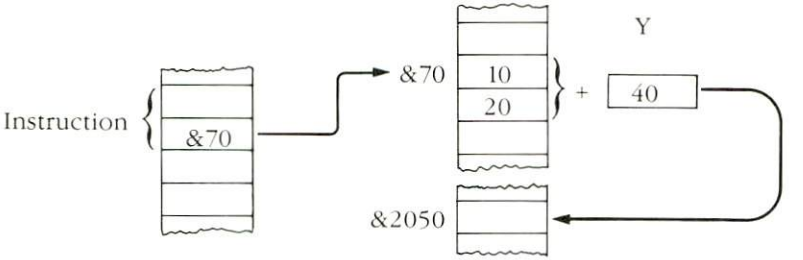
The following are examples of the instructions used in this mode: ADCIX (ADC), CMPIX (CMP) and LDAIX (LDA).

4.1.10 Absolute indexed indirect addressing

This is a special mode that is only available on CMOS 6502s and for just one instruction: the jump instruction. It is similar to the address mode described in the last section, zero page indexed indirect, but the operand is two bytes long (absolute) rather than one (zero page). Thus, the action of the jump instruction using this mode is to add the value of X to the operand and jump to the location stored in the sum of these two. The mnemonic for the instruction is 'JMIX'.

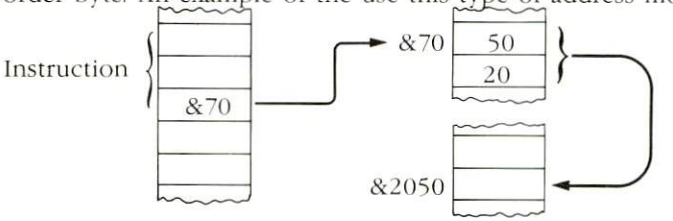
4.1.11 Indirect indexed addressing

This address mode is also called 'Indirect, Y' and is shown under the heading '(Z),Y' in Table 4.1. Its instructions are two bytes long and include ADCIY (ADC) an example of which is shown in the diagram below. The second byte of the instruction points to a location in page zero and the contents of this location are added to the contents of the Y register to give the low order byte of the target address. To get the high order byte of this address the carry from the addition is added to the contents of the next location in page zero.



4.1.12 Zero page indirect addressing

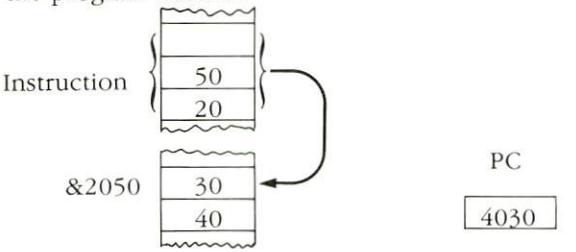
In this address mode, which only exists in CMOS CPUs, the instructions are two bytes long. The second byte points to a location in page zero that contains the low order byte of the target address. The next byte in page zero contains the high order byte. An example of the use this type of address mode is:



4.1.13 Absolute indirect addressing

The instruction in this address mode JMI (JMP) is three bytes long. The second byte consists of the low order byte of an address and the third byte the high order byte of the address.

The contents of this new address contain the low order byte of the target address and the next byte contains the high order byte. The target address is loaded into the program counter.



A 'Bad JMI' error is caused by an instruction of the form JMI &XXFF when MASM is in CPU0 mode which will crash when executed on a standard NMOS.

4.2 The format of MASM source code

MASM source code takes the following general form:

```
<label>    <opcode>    <operand>    <;comment>
```

(unless the line contains a directive as described in chapter 5).

Where things appear between < and > in this book they should not be taken literally; they indicate a 'class' of items. The line above, for example, means that a line of MASM source code consists of four parts: a label, an opcode, an operand and a comment. None of these is compulsory (except that if there is an operand then there must be an opcode); for example, opcodes such as INX do not need an operand. Also, comments are never needed by MASM, but they should be included to help anyone who reads your program. Below are some examples of MASM source lines:

```
bell    LDAIM  7       ;Sound bell
        JSR    oswrch

    ;  Just a comment on this line!
```

The first example has all four of the possible components present. 'bell' is a label; in this case, the label could be the destination for a

```
        JSR    bell
```

instruction, and so stands for an address within the program. Next comes the opcode; in this case, a 6502 instruction that means 'load the accumulator immediate'. You may be more familiar with the

```
        LDA    #7
```

form of this instruction. MASM's way of specifying address modes has already been outlined.

The operand on the first line is '7', which happens to be the ASCII bell control character. The comment (which must be preceded by ';' to warn MASM) is the last item on the line. MASM completely ignores this part; it is present purely to help anyone who is trying to understand the program.

The second example above has neither a label nor a comment (labels are needed only if a reference is made to the line elsewhere in the program). The opcode in this example is 'JSR' (jump to subroutine) and the operand is 'oswrch'. 'oswrch' is another example of a label; this time, the label refers not to a location within the program, but to an address elsewhere (in the Machine Operating System ROM, to be precise). It would have been equally valid to use '&FFEE' here, as this is the numeric address of the oswrch routine. However, labels (or symbols) make a program more readable and should be used whenever possible. The mechanism for defining symbols is described later, in chapter 5, 'The MASM directives'.

The third line in the example above is empty; blank lines are accepted by MASM and are used to separate routines in the listing. Again, this is purely to improve the appearance of the listing rather than something that MASM demands.

The fourth line in the example shows how a comment can occupy an entire line of the source file.

Labels can consist of letters and numbers, but must start with a letter. They can be up to six characters long.

Note that if a line has no label, any opcode present must start after the first column (that is, there must be a space before the opcode appears).

4.2.1 Using expressions in operands

MASM allows you to use a general expression in operands, where a number is required, just as in high-level languages like BASIC. These expressions consist of the normal features of an operand (symbols, numeric constants and string constants) combined with operators (described in section 4.3). Some examples of these expressions are given below.

```
LDAIM   &F0:AND:&1F
ADCIM   &F0:OR:&0F
LDAIM   bytes/size
```

4.2.2 Symbols

Symbols are strings of alphanumeric characters starting with a letter and having up to five further characters. Note that the letters may be upper or lower case.

We have already come across one type of symbol: the address label. Symbols are identifiers which stand for constant values throughout the program. Thus, the label 'loop' in the memory check program of section 3.1 always has the value of the address at which it appears in the program. Also, wherever 'loop' is used in an operand, that address will be substituted.

Symbols can also be defined explicitly using the '*' directive (described in chapter 5, 'The MASM directives'). This is different from the '*' operator used in expressions, and it stands for 'define symbol'. Its effect is similar to the assignment operator '='. The example below defines two symbols:

```
oswrch  *     &FFEE
limit   *     &100
```

The first symbol is used as an address (in this case, the address of the operating system's character output routine) and it might be used in a line such as:

```
loop    JSR   oswrch
```

The second symbol defines some value which might appear many times in a program and deserves a name for the sake of readability, as in:

```
        LDA   start+limit
```

Notice that when a symbol is being defined with '*', the identifier must appear in the label field of the line, that is, it must start at the first column of the line. Because of this rule, symbols are often referred to as 'labels' whether they stand for an address in the program or not.

There are two special symbols which will be defined in the next chapter, but are mentioned here for completeness: these are the program counter and the variable counter. They define the current points in the program and the variable space.

4.2.3 Numeric constants

Numbers that appear in MASM source code can be in decimal or hexadecimal. A typical decimal number is '32768', and '&100' is a hexadecimal number (hex numbers are preceded by the character '&'). Numeric constants are limited to two bytes precision (&0000 to &FFFF in hex).

4.2.4 String constants

These are enclosed in double quotation marks ("), for example the string:

```
        =     "Memory OK"
```

in the memory check program of section 3.1. If the quotation mark itself is required, you need to use two sets of quotation marks; for example, to obtain the string consisting of just the character '"', you should use """". String constants can be used in place of a string of single bytes or as a number. An example of a string of single bytes is given above. The general form of this use is:

```
<label> = <byte> (,<byte>)
```

The part in brackets can occur any number of times, as in the following:

```
mesg    =       65,66,67
```

This would insert the codes for the characters 'A', 'B' and 'C' into your program. A more readable alternative would be:

```
mesg    =       "ABC"
```

Another use for string constants is shown below. Here, two symbols are defined by giving them 'string' values.

```
prompt  *       ">"
endsym  *       "@"
    ..........
    ..........
        LDAIM prompt
        JSR   oswrch
    ..........
    ..........
        JSR   osrdch
        CMPIM endsym
        BEQ   fin
    ..........
    ..........
```

What actually happens is that the symbols have the ASCII values of the characters assigned to them. Using symbols in this way would enable the prompt or end symbol to be changed merely by altering the two initial '*' lines.

4.3 MASM operators

MASM provides an extensive set of operators for use in expressions. The power of these operators rivals some high-level languages and they are described in detail below.

4.3.1 Arithmetic operators

These operators include:

+	Add
-	Subtract
*	Multiply
/	Divide
:MOD:	Remainder after division

Examples of the use of arithmetic operators are given below:

```
LDA    start+limit

LDAIM fin / &100+1
```

You may recall the last example; it was used in the memory check program to calculate the start address of the page above the program.

An example of the use of :MOD: is:

```
LDXIM offset :MOD: 256
```

which would load the low byte of 'offset' into the X register.

4.3.2 Logical operators

These include the following:

:OR:	Inclusive OR function
:EOR:	Exclusive OR function
:AND:	Logical AND function

Some examples will help to explain their effect:

	Result
`LDAIM &F0 :AND: &1F`	`LDAIM &10`
`LDXIM &12 :EOR: &0F`	`LDXIM &1D`
`ADCIM &F0 :OR:  &0F`	`ADCIM &FF`

Of course, actual examples would include items other than constants, otherwise you might as well do the operations in your head.

4.3.3 Rotation and shift operators

These operators take a bit pattern and either shift or rotate it a number of positions to the left or the right. You can imagine them as two basic operations:

```
:SHxy:      for shift
:ROxy:      for rotate
```

The suffix x can be either L or R for left or right, respectively. y is 1 for operation on one byte and is omitted for operation on two bytes. Some examples might clarify this:

		Result
`&10`	`:SHR1: 2`	`&04`
`&AA`	`:ROL1: 3`	`&55`
`&1234`	`:SHL:  2`	`&48D0`

4.3.4 String operators

These operators join (concatenate) two strings or strip out part of one string. They are as follows:

:CC: concatenates two strings as follows:

```
"ABC" :CC: "123"   yields the string   "ABC123"
```

:LEFT: takes a string on the left and a numeric expression on the right to give the left-most substring. For example,

```
"ABC123" :LEFT: 3   yields   "ABC"
```

:RIGHT: takes a string on the left and a numeric expression on the right to give the right-most substring. For example,

```
"ABCD123" :RIGHT: 4   yields   "D123"
```

Again, it would be more natural to use expressions rather than constant values for operands.

4.3.5 Arithmetic unary operators

These act on a single operand. They include:

+	No effect
-	Negate the operand
:LSB:	Take the least-significant byte of the operand
:MSB:	Take the most-significant byte of the operand
!	Set the most-significant bit of the operand's lesser byte
/	Swap bytes of operand

+ has no net effect on an expression, whereas - negates it. All the unary operators have the highest precedence, so brackets may be needed:

```
- (add1 + size)   yields   -add1 - size
- add1 + size     yields   -add1 + size
```

:LSB: and :MSB: yield, respectively, the least significant and most significant bytes of their operand. For example,

```
:LSB: &1234   yields   &34
:MSB: &1234   yields   &12
```

Notice that :LSB: and :MSB: are equivalent to the following expressions:

```
:LSB: x   =   x :MOD: &100
          =   x :AND: &FF

:MSB: x   =   x / &100
```

In practice, of course, you would use expressions for :LSB: and :MSB: to operate upon, rather than constants. For example:

```
LDXIM :LSB: cmdlin
LDYIM :MSB: cmdlin
JSR   oscli
```

! takes the least significant byte of the operand and sets its most significant bit. The most significant byte is not affected by the operation. For example,

```
!   &01   yields  &81
! &0101   yields  &0181
```

Again, you would use expressions in the operand, rather than constants, otherwise you might as well do the calculations yourself.

/ swaps the order of the bytes attached to it. For example,

```
/ &1234   yields  &3412
/   &32   yields  &3200
```

4.3.6 The :NOT: unary operator

The :NOT: operator takes a two byte operand and inverts the state of each bit. For example,

```
:NOT: &AA     yields  &FF55
:NOT: &FFEE   yields  &11
```

4.3.7 The unary string operators

The last three operators we have to look at are :LEN:, :STR: and :CHR:.

:LEN: gives the number of characters in its string operand. For example,

```
:LEN: "ABC123"            yields  6
:LEN: ("12" :CC: "32")    yields  4
```

:STR: takes a two byte numeric operand and converts it into a string containing the hexadecimal equivalent of the operand. For example,

```
:STR: 10      yields  "000A"
:STR: &F      yields  "000F"
:STR: &1234   yields  "1234"
```

When it is used on logical operators, :STR: returns the two strings "T" and "F", corresponding to TRUE and FALSE. For example,

```
:STR: (12 > 13)   yields  "F"
:STR: (12 < 13)   yields  "T"
```

:CHR: takes an arithmetic value and turns it into a string without changing its value. For example,

```
:CHR: &41   yields  "A"
```

4.4 Operating system commands

You can include operating system commands in your source files. These must be put in the label field, for example:

```
     .............
     .............

;  Switch to serial printer for listing
*FX5,2
     .............
     .............
```

It should be noted that it is not recommended that *SPOOL be used to save assembly listings to disc.

5 The MASM directives

At some stage you might want to tell the assembler to reserve some variable space, end an assembly or perform some other action. You would do this through the use of 'directives' in your source file and these are described below.

5.1 Declaring symbols (*)

Symbols are declared using the * directive. This has the following format:

```
<label> *       <expression>
```

The label should obey the rules for symbols, and the expression should yield a numeric result in the range &0000 to &FFFF (in decimal, this can be taken to be 0 to 65535 for unsigned numbers, or -32768 to 32767 for signed numbers). Some examples of this directive are given below:

```
oswrch  *       &FFEE
osbyte  *       &FFF4
table1  *       table2+&10
table2  *       &70
del     *       &7F
eoln    *       "!"
```

Notice the different uses to which the symbols, or labels, will be put: the first two are subroutine addresses which should be familiar to users of the BBC Microcomputer.

Next, there are two addresses of tables; these could be fixed data areas that the program should know about. The definition of table1 uses table2 in its operand expression. There is nothing wrong with this: it is known as a forward reference, and we have come across these before in chapter 2, 'Developing programs with the Second Processor'. Forward references will be 'resolved' during the first pass of the assembler so that, at the start of the second pass, all label values should be known. A consequence of this particular forward reference is that table1 will be assumed to be a two-byte (non-page zero) address, so an instruction like:

```
        LDA     table1
```

will use absolute addressing rather than zero page addressing. However,

```
        LDA     table2
```

will use zero page addressing, since table 2 is known to be less than &100 during the assembler's first pass. The only real significance of all this is that the program will be slightly longer in its assembled form than if there were no forward references at all.

Note, however, that 'nested' forward references cannot be resolved by MASM, for example:

```
label2 *       label1
label1 *       code1

code1  ;label position determined code assembly
```

which will give an 'Undefined symbol' error on pass2 through the first line.

The last two examples of the * directive define character constants. They give names to two frequently-occurring values in the program, and can be used in circumstances such as:

```
        CMPIM eoln          ;End of line reached?
        BEQ   endln
        CMPIM del           ;Delete character?
```

5.2 Defining a byte of data (=)

You will sometimes need to include some data bytes inside a program. To do this, you would use the = directive. This has the format:

```
<label> =      <expression>
```

where the label is optional.

One place where you might use this directive is when the 6502 BRK command is being used to issue an error message (as is the convention with the BBC Microcomputer). The BRK instruction is followed by an error string and this string is terminated by a zero byte:

```
error1  BRK                 ;An error
        =       254         ;Error number
        =       "Too many parameters"   ;Error message
        =       0           ;Error terminator

error2  BRK                 ;And so on
```

As discussed above, the string will be converted into a sequence of bytes. In fact, all of the data bytes above could be replaced be the one line:

```
        =       254,"Too many parameters",0
```

Another use for the '=' directive is in setting up a table for use by the program. The one below is a table of powers of two, located at 'powers':

```
powers  =       1,2,4,8,16,32,64,128
        ...........
        ...........
        ...........
        LDX     index
        ORAAX   powers
```

The fragment of code shows how the table might be used. Register X is loaded from location 'index', which should contain a value between zero and seven, and this is used to index the table of powers during the OR-ing operation.

5.3 Defining a byte pair (&)

Two bytes can be inserted into the program at once using the & directive. This could be used, for example, when setting up a table of addresses in memory, such as:

```
jmptab  &       reset1  ;Set up jump table
        &       reset2
        &       plot
        &       draw
        &       erase
        ...........
        ...........
```

The value of the label reset1 will be placed at address jmptab in standard low-byte, high-byte order. The next two bytes will contain the value of reset2, and so on. Each time an '&' is encountered, the program counter will be incremented by two.

The program which manipulates the jump table shown above might contain the following instructions:

```
exec    LDA     action      ;Get the action code.
        ASLA                ;Times two for indexing
        TAX                 ;with the X register.
        LDAAX   jmptab      ;Get jump address low byte.
        STA     jmpvec      ;Save in jump vector.
        LDAAX   jmptab+1    ;Same with high byte.
        STA     jmpvec+1
        JMI     jmpvec      ;Do the indirect jump.
```

Incidentally, there is a more efficient way of handling jump tables. You should break the above table into two separate halves, one containing the low bytes and the other the high bytes; each address should be one less than the location to be jumped to. To use the table, you load the high byte and push it onto the stack, load the low byte and push it onto the stack then do the branch using an RTS instruction. This will save you two instructions.

If a single character is placed into two bytes using the & directive, the second byte will be zero.

Thus,

```
        &       "A"
        &       "B"
```

will result in the four bytes 65,0,66 and 0 being inserted into the object code.

5.4 Reserving variable space (^, @ and #)

The variable storage area is separate from the object program area. It has its own origin, which is defined using the ^ directive, and its own location counter, @. This directive has the format:

```
        ^       <expression>
```

where <expression> is fully defined during the first pass of the assembler. If there is no ^ directive in a program, @ will be set to &0000. There can be as many ^ directives as you wish in a program, so data areas can be separated into logical places in the memory.

To reserve space in the variable storage area, you use the # directive. Its format is:

```
<label> #       <expression>
```

Whenever this directive is encountered, the variable counter @ is incremented by the appropriate amount and <label> is given the old value of @. For example, after the sequence:

```
        ^       &2000
mesg1   #       &100
mesg2   #       24
```

the variable counter '@' would contain &2118 because 280 (&118) bytes were reserved by the '#' directives, mesg1 would be &2000 and mesg2 would be &2100.

An example of the use of the '#' directive might be when space is needed for the input buffer in an interactive program. For example:

```
        ^       &2100       ;Initialise variable counter
osword  *       &FFF1
buflen  *       40          ;buffer length
buffer  #       buflen      ;reserve 40 bytes
parblk  #       5           ;5 bytes for parameter block

        ..............
        ..............
        ..............

input   LDAIM :LSB:buffer   ;Set up parameter
                            ;block buffer
        STA   parblk        ;address
        LDAIM :MSB:buffer
        STA   parblk+1
        LDAIM buflen        ;Max. line length
        STA   parblk+2
        LDAIM " "           ;Min. ASCII value
        STA   parblk+3
        LDAIM &FF           ;Max. ASCII value
        STA   parblk+4
        LDXIM :LSB:parblk   ;XY point to parblk
        LDYIM :MSB:parblk
        LDAIM 0             ;Input is osword 0
        JSR   osword
```

Notice that the expression after the # may not contain forward references, otherwise a phasing error will result. For example, the following code will generate an error:

```
        #       A
A       *       1
```

An interesting alternative, which may be used if the parameter block will never change throughout a program run, is to use '=' and '&' to set it up:

```
parblk  &       buffer
        =       buflen
        =       " ",&FF
```

If this method is used, only the statements from the 'LDXIM' onwards are necessary to perform the input.

You can, if you wish, preset store to zero using the % directive. This has the format:

```
%       <expression>
```

This fills store with zeroes from the address in the program counter.

The areas of memory that are available for use in both processors, when defining memory, are described in the table in the next section.

5.5 Defining the start of code (ORG and .)

The directive ORG is used to define the address at which assembled code will originate. This has the format:

```
        ORG     <expression>
```

The expression must be fully defined at this point, so any symbols that it contains should be introduced before the ORG directive. Examples of the directive are given below:

```
        ORG     &1900       ;For disc
start   *       &1200       ;Change as appropriate
        ...........
        ...........
        ...........
        ORG     start
```

Associated with the ORG directive is the program counter (`.`). This marks the address of the current point in the program (during assembly) and is initialised by the ORG directive. Thus, after the statement

```
        ORG     &1900
```

using '`.`' in an expression will yield the number &1900. Whenever an instruction is assembled, '`.`' is incremented so that it holds the address of the first byte in the following instruction. For example, if the line

```
        JSR     init
```

were to appear next, the program counter '`.`' would become &1903 since a JSR instruction takes up three bytes.

There can be only one ORG per source file, so if the object code is to lie in several discrete 'lumps', several files will need to be assembled separately with different ORG lines. If an ORG is included, it must appear before any instructions which increment the program counter (all 6502 instructions, & etc). Omitting ORG sets '`.`' to &0000 initially.

You can put machine code in the following places in your BBC Microcomputer system:

Processor	Space available
BBC Model B	Between OS High Water Mark and HIMEM Between locations &900 and &AFF inclusive, if you are not using the RS423 channel or sequential cassette I/O Between locations &C00 and &CFF inclusive, if you do not want to use programmed characters
6502 Second Processor	Between locations &400 and &F7FF inclusive

If the machine code is to be called from, say, BASIC, it should not corrupt the language workspace unless control will not be passed back.

5.6 Linking source files (LNK and <)

The problem of having only one ORG directive per file can be overcome by using the LNK directive. Its format is as follows:

```
        LNK    <file name>
```

When this directive is encountered, MASM gives up assembling the current file and restarts from the beginning of the new one. As the latter can contain its own ORG directive, your object code can be split into as many separate parts as you want. All of the source file after a LNK directive is ignored, but the symbols in one file are accessible from another, so the following will work:

```
     ;This is in file1
symb1    *      symb2*2
         ORG    &1900
         ..........
         ..........
         ..........
         LNK    file2
     ;Anything after this in file1 will be ignored
```

```
      ;This is in file2
symb2   *       &1200
        ..........
        ..........
        ..........
        END
```

What happens here is that 'symb1' is defined by referring to a symbol in file2. This forward reference will be resolved when file2 is linked in using the LNK directive, so symb1 would have the value &2400. Notice that there is no 'ORG' statement in file2; in this case, its instructions will be assembled immediately after the last one in file1, as if the files had actually been concatenated. When the 'END' directive is encountered in file2, the assembly process will either start pass2 (if pass1 has been completed without error) or it will be terminated; the assembler will not return to the place just after the 'LNK' in file1 and continue from there.

The file specified after a LNK is a string, of at most ten characters.

You can specify the source drive using the < directive. This has the format:

```
        <       <expression>
```

where <expression> is the source file drive code. Note that this directive applies only when using the disc filing system; it is ignored by other systems.

5.7 Ending an assembly (END)

When MASM encounters the directive END in the operand field of a line, it either starts the second pass (if it has completed the first pass without error) or it terminates the assembly process. This directive is similar to the LNK directive in that any source lines following it will be ignored by the assembler.

During the assembly of a file or sequence of files, there must be one END directive somewhere, otherwise a 'No END' error will occur.

An END directive is not needed in a file which links with another one.

5.8 Defining page titles (TTL)

One of the MASM commands described in chapter 12, 'Using MASM to assemble your programs', enables you to produce an assembler listing of your program (assuming no errors were encountered). This listing is in a standard format, and each page has a header or title. If you wish, you can alter the latter by using the TTL directive. For example, after the line:

```
        TTL SORT PACKAGE INPUT MODULE
```

has been assembled, all subsequent page headings will contain the string

There can be as many TTL directives in a program as you wish. However, if they are too close together, not all of them will be used unless you explicitly start a new page every time TTL is used (see the following section).

5.9 Setting print options (OPT)

You can control the assemble-time output using the OPT directive. This has the following format:

```
OPT    <expression>
```

where <expression> is treated as a four bit number. Bits 0 to 3 of this number have the following meaning:

Bit	Meaning when set
0	Turn printing ON
1	Turn printing OFF
2	Start a new page in the listing
3	Reset the line count in the listing

The first two options allow selective listing, provided the PRINT ON command has been issued (see chapter 12, 'Using MASM to assemble your programs').

5.10 Using the special instruction set (CPU)

MASM will recognise the extra instructions which are provided by the CMOS version of the 6502 processor; these include PHX and CLR. If you want to use these in your programs you must first use the CPU directive. This has the format:

```
CPU    <0 or 1>
```

CPU 0 (the default) will restrict MASM's knowledge to the standard 6502 instructions; CPU 1 will allow the extra CMOS instructions to be assembled without errors being generated. Any CPU directive in your program must appear before the first code-generating op-code is met. Thus, it is not possible to (or even very likely that you would want to) change the instruction set half way through the source code.

5.11 Changing object file drives (>)

You can assemble object files to a different drive using the > directive. This has the format:

```
>       <expression>
```

where <expression> is the object file drive code.

6 Program example

Now that you understand the basic components of the MASM assembly language it is a good time to consolidate them in the form of a complete program example. Clearly, it is not possible to include all of the features easily into one program, but the following example shows many of them in use.

The program takes a file name that you input and produces a histogram which shows the distribution of character codes in the file. The screen output is produced in MODE 4.

```
;       A File Character Distribution Program
;
;       For BBC Microcomputer Model A or B
;
    TTL File Characteristics
;       Define OS Addresses etc.
oswrch  *       &FFEE
osasci  *       &FFE3
osword  *       &FFF1
osfind  *       &FFCE
osbget  *       &FFD7
zerop   *       &70             ;User zero page locations
buflen  *       10              ;Input buffer size
tablen  *       &100            ;Table of counts
code    *       &1B00           ;Execution address
minasc  *       &20             ;First printing char
maxasc  *       &7F             ;Last legal ASCII code
nopcd   *       &EA             ;NOP instruction's code
mode    *       22              ;VDU MODE byte
clscd   *       12              ;CLS
plotcd  *       25              ;VDU PLOT byte
movecd  *       4               ;Absolute move
drawcd  *       1               ;Relative draw
        ^       code-tablen-buflen
buffer  #       buflen          ;Space for input buffer
table   #       tablen          ;One byte per ASCII code

        ORG     code
        JMP     count           ;Start vector
```

```
parblk  &       buffer          ;Set up osword buffer
        =       buflen
        =       minasc
        =       maxasc


count   JSR     init            ;Zero the table
        JSR     input           ;Get the file name
        JSR     open            ;Try to find it
count1  JSR     osbget          ;Read a byte
        BCS     eof             ;We've reached the end
        TAX                     ;INC the entry for char.
        INCAX   table
        BNE     count1          ;Count is > &FF

eof     JSR     close           ;Close the file
        JSR     plot            ;Plot the results
        RTS                     ;Go home

;---------------------------------------------------------

init    LDAIM   0               ;Put zeros in the table
        TAX
initlp  STAAX   table
        DEX
        BNE     initlp
        RTS

;---------------------------------------------------------

input   JSR     print             ;Print prompt
        =       clscd,"File name? "
        NOP
        LDAIM   0                 ;Osword 0 is INPUT
        LDXIM   :LSB: parblk      ;XY points to parblk
        LDYIM   :MSB: parblk
        JSR     osword
        RTS

;---------------------------------------------------------

print   PLA                     ;This prints the string
        STA     zerop           ;following the JSR print
        PLA                     ;Store pointer to string
        STA     zerop+1
        LDYIM   0
```

```
print1  INC   zerop        ;Next address
        BNE   printf
        INC   zerop+1
printf  LDAIY zerop        ;Get a byte
        CMPIM nopcd        ;Last byte?
        BEQ   prnret       ;Yes
        JSR   osasci
        JMP   print1       ;Always
prnret  JMI   zerop

;-------------------------------------------------------------

open    LDAIM &40                  ;Osfind &40 is OPENIN
        LDXIM :LSB: buffer         ;XY points to file name
        LDYIM :MSB: buffer
        JSR   osfind
        TAY                        ;Get file handle in Y
        BNE   opnret               ;File found

        BRK                        ;Otherwise cause error
        =     &86                  ;DFS Error number
        =     "File not found"     ;Standard message
        =     0                    ;End of error

opnret  RTS

;-------------------------------------------------------------

close   LDAIM 0            ;Osfind 0 = CLOSE#Y
        JSR   osfind
        RTS

;-------------------------------------------------------------

plot    LDAIM mode         ;Do MODE 4
        JSR   oswrch
        LDAIM 4
        JSR   oswrch

        JSR   print
        =     "Distribution of ASCII Codes",13
        =     "----------------------------",13
        NOP

        LDXIM 0            ;Do each entry in table
```

```
plotlp  LDAIM plotcd        ;PLOT 4,X*4,0
        JSR   oswrch
        LDAIM movecd
        JSR   oswrch
        TXA                 ;X
        JSR   low           ;VDU A*4
        TXA
        JSR   high          ;VDU (A*4)/&100
        LDAIM 0             ;Y=0
        JSR   oswrch
        JSR   oswrch

        LDAIM plotcd        ;PLOT 1,0,(table?X)*4
        JSR   oswrch
        LDAIM drawcd
        JSR   oswrch
        LDAIM 0             ;DX=0
        JSR   oswrch
        JSR   oswrch
        LDAAX table         ;DY=(table?X)*4
        JSR   low
        LDAAX table
        JSR   high

        INX                 ;Next byte
        BNE   plotlp
        RTS

;-------------------------------------------------------------

low     ASLA                ;Send low byte of A*4 to
        ASLA                ;oswrch
        JSR   oswrch
        RTS

;-------------------------------------------------------------

high    ROLA                ;Send hi byte of A*4 to
        ROLA                ;oswrch
        ROLA
        ANDIM 3
        JSR   oswrch
        RTS

;-------------------------------------------------------------

        END
```

7 Using macros in your programs

Quite often, the same pattern of instructions occurs several times in a program. In such a situation, it would be useful if you could type just one line, then sit back and have it expanded for you. For example, look at the following error handling code:

```
err1    BRK
        =       &83
        =       "Too few arguments"
        =       0
```

This type of code might occur at several points in a large program. Wouldn't it be handy if you could type something like:

```
err1    ERROR &83,"Too few arguments"
```

every time such a piece of code was needed, and have the assembler expand it for you?

This type of facility is called a macro, and it gives MASM its name: MASM is a macro assembler.

Before you use a line such as:

```
err1    ERROR &83,"Too few arguments"
```

in your program, the macro must be defined. Thus, the macro ERROR could be defined as follows:

```
        MACRO
$label  ERROR $errnum,$errstr
$label  BRK
        =       $errnum
        =       $errstr
        =       0
        MEND
```

This is called the 'formal definition' of the macro, and the parameters $label, $errnum and $errstr are called its 'formal parameters'. When you use the macro, you will supply parameters such as

```
err1
&83
"Too few arguments"
```

and their values will be substituted in place of the formal parameters.

There is a tendency to think of a macro definition as being the same as a subroutine to which the program can JSR. This is wrong! When a subroutine is called, a single instruction

```
        JSR    label
```

is assembled. However, when a macro is called, all the instructions in the macro's body are assembled 'in line'. Thus, the liberal use of long macros can result in an object file much larger than might be expected: this demands some caution on your part.

You can probably appreciate now how the use of even a simple macro such as ERROR can save much typing (and in the process reduce the number of typing mistakes you might make).

Now that we have described the basic principles, we will look at macros in more detail. Look at the line

```
err1    ERROR &83,"Too few arguments"
```

The fields are similar to a normal line: first comes a label (optional, as usual), then the opcode (the name of the macro in this case), then the operand. In fact, macros can take several operands, or parameters; the one above has two.

Now look at the formal definition of this macro:

```
        MACRO
$label  ERROR $errnum,$errstr
$label  BRK
        =     $errnum
        =     $errstr
        =     0
        MEND
```

The first line is the directive MACRO. This warns MASM that a macro definition follows. The second line is the 'header line': it tells MASM the name of the macro being defined (in this case 'ERROR'), and the names of the parameters it takes. These parameter names are standard identifiers, but have the prefix $. They correspond positionally to the parameters which are used when the macro is called.

Notice that '$label' appears twice in succession. This is normal in a macro definition: the first occurrence gives a name to the label used when the macro is called, and immediately below it the name is actually used (substituted) in the assembler code.

7.1 Default values in macros

There are 'extras' which are useful to know when using macros. First, it is possible to initialise the formal parameters to a default value which will be used if the actual parameter is ever omitted. For example, look at the following macro:

```
        MACRO
$label  INPUT $prompt="""?"""
$label  JSR   print
        =     $prompt
        NOP
        .........
        .........
        MEND
```

This assembles the instructions needed to perform the equivalent of a BASIC

```
INPUT "Prompt",A$
```

statement. The prompt could be a parameter to the macro, but if you wanted, a default '?' might be used instead.

In the definition shown above, the parameter $prompt is given a default value of "?" by the ="""?""" part. Note the lack of spaces; this is important. Note also the escape quote sequence """" that is used to obtain the string "?". The $prompt parameter will be used, subsequently, in setting up a call to the standard print routine which was used in chapter 6, 'Program example'.

Now look at the following code which uses 'INPUT':

```
getstr  INPUT "How many eggs? "
        .........
        .........
        INPUT ¦
```

The first call to INPUT provides an actual parameter, "How many eggs? ", and this will be used in the macro's body. The second call, however, has the parameter ¦ (vertical bar), and this will be taken to mean 'use the default, if there is one'.

7.2 Missing parameters

If the macro has more than one parameter, those that are missing must be indicated by putting commas in the calling line. For example, you might define the macro 'MAC1' as follows:

```
        MACRO
$label  MAC1   $P1,$P2,$P3
        .........
        .........
        MEND
```

and then call it in the following ways:

```
        MAC1 1,2,3
        MAC1 1,,
        MAC1 ,1,
```

In the first call, all three parameters are present. In the second, only $P1 is set up (to 1), and the third call only uses $P2. Any parameters which are missing will be set to null, which effectively means that they disappear when the body of the macro is assembled. Note that quotation marks should not be put round string parameters when they are used in macro calling statements, otherwise a 'Type mismatch' error will be generated.

7.3 Parameter names

Since $ marks the start of a parameter name, it is vital that MASM can distinguish the actual character '$'. Where confusion may arise, the 'escape sequence' $$ can be used to denote a single $. Look at the following macro, for example:

```
        MACRO
$label  PRICE $cost
        =     "The price is $$$cost."
        MEND
```

Here, the first two '$' symbols are replaced by a single '$' currency symbol. The last '$' marks the start of the parameter name '$cost', which will be substituted as usual.

This example illustrates another property of macro parameters: they are substituted wherever they occur in the macro's body. The example below shows a macro which produces instructions that will swap either the X or Y register with a specified memory location:

```
        MACRO
$label  SWAP  $reg,$mem
$label  PHA                 ;Save A
        LDA   $mem          ;Get $mem
        ST$reg $mem         ;Save $reg in $mem
        TA$reg              ;Put $mem in $reg
        PLA                 ;Restore A
        MEND
```

Some code which uses this macro is shown below:

```
start   SWAP  X,&1000
        SWAP  Y,&1001
```

Wherever $reg appears in the body of this macro, it is changed to X or Y. This applies within string constants (otherwise the $ example above wouldn't work properly); it also applies within comments, if some substitution has occurred earlier in the line.

If you need to use a macro parameter name immediately before some other identifier, confusion could arise. Consider this macro definition, for example:

```
        MACRO
$label  GETPUT $mode
$label  $modeA              ;Load or Store A
        ..........
        ..........
        MEND
```

The idea here is that a call to GETPUT will have either 'LD' or 'ST' as a parameter and, after substitution, the third line of the macro definition would have either 'LDA' or 'STA' as its operand. However, '$modeA' is a perfectly valid parameter name, and MASM always tries to find the longest name possible. Thus, instead of recognising the parameter '$mode' followed by the text 'A', MASM will interpret the third line as a single object, namely, the parameter '$modeA'. As no such parameter exists, an error will occur.

To avoid this difficulty, you can insert a point '.' after the parameter name, to explicitly terminate it. In the last example, the third line should now be:

```
$label   $mode.A              ;Load or Store A
```

This time, MASM will see the end of the parameter and do a proper substitution. The '.' will then disappear.

7.4 Nesting macro calls

Another property of macros is that calls to them can be nested: one macro's definition can contain a reference to another macro. Suppose, for example, that there is a macro called PLOT defined somewhere in a program. This macro could be used by other macros as follows:

```
         MACRO
$label   MOVE   $x,$y
$label   PLOT   4,$x,$y
         MEND
         .........
         .........
         MACRO
$label   DRAW   $x,$y
$label   PLOT   5,$x,$y
         MEND
         .........
         .........
         MOVE   0,0
         .........
```

The line 'MOVE 0,0' will lead indirectly to the call:

```
         PLOT   4,0,0
```

and this will result in the generation of actual instructions. In some circumstances, it might even call yet another macro.

It is perfectly correct to have macro definitions without parameters, for example:

```
        MACRO
$label  SAVER   ;Save the registers
$label  PHP
        PHA
        TYA
        PHA
        TXA
        PHA
        MEND


        MACRO
$label  RESTOR  ;Restore the registers
        PLA
        TAX
        PLA
        TAY
        PHA
        PLP
        MEND

entry   SAVER
        .........
        .........
end     RESTOR
```

7.5 Macro libraries

You can keep a number of macros in a file and have them included in your program source by MASM itself. This macro library file can only contain macro definitions, comments, a LNK directive or an END directive.

When MASM is run with the ASM command, it gives you the prompt:

```
Macro library:
```

and you can then specify the name of the file containing your macros.

If you want to include macros from a number of files you can link them together using the LNK directive described in section 5.6. MASM will search the files until a macro file with END is encountered.

8 Conditional assembly

The [and] directives enable a section of a source file to be assembled only if a certain condition is true. Their use applies particularly to macro definitions, although they can, in fact, be used anywhere in the source listing.

The [directive is known as 'IF' and the ']' directive as 'END IF'. Look at the following code:

```
[<logical expression>
<conditional instructions to be assembled>
]
<rest of instructions>
```

If the <logical expression> yields a true result then the conditional instructions will be assembled. Once the] is encountered, assembly continues as normal.

An example might make things clearer. The macro definition below provides a selective version of SAVER that was described in the last chapter:

```
        MACRO
$label  SAVE   $p1,$p2,$p3
$label
        [      "$p1"<>""   ;if $p1 contains register
        ST$p1  TEMP$p1     ;store in corr. temp. store
        ]
        [      "$p2"<>""   ;similarly for $p2
        ST$p2  TEMP$p2
        ]
        [      "$p3"<>""   ;finally $p3
        ST$p3  TEMP$p3
        ]
        MEND
        ............
        ............
        SAVE   A,,
        SAVE   X,A,Y
        SAVE   Y,A,,
        ............
        ............
        END
```

The action of the macro is simple. There are three conditional tests, each testing one of the parameters. If any parameter contains the name of the A, X or Y registers then the contents of that register are stored in the memory location with label TEMP followed by the name of that register. For example, the contents of the Accumulator are stored in TEMPA.

Note that the parameters may be in any order. For example, if the call to SAVE is:

```
SAVE   A,X,
```

then $P1 will have the value A, $P2 the value X and $P3 will have no value. Thus the test:

```
[      "$P1"<>""
```

will become:

```
[      "A"<>""
```

when the parameters are substituted. This is obviously a true condition, so the next line will be assembled as:

```
STA    TEMPA      ;store in corr. temp. store
```

The next test will cause:

```
STX    TEMPX
```

to be assembled, and the third test will become:

```
[      ""<>""
```

which is clearly false, so the next line will not be assembled.

When your source program is assembled, the code ignored by your conditional assembly directives will be suppressed. If you want to list this code on assembly you can do so using the MASM TERSE command (see section 12.1.10).

There is an enhancement to the IF directive in the shape of ¦, or ELSE. With this new directive it is possible to make MASM take one of two paths that are dependent upon the result of the logical expression. The enhanced form of the IF directive becomes:

```
        [       <logical expression>

;Do these if <logical expression> is true
        .............
        .............
        .............


;Otherwise, do these
        .............
        .............
        .............
        ]

;Carry on assembling
```

This construct is related to the 'IF ... THEN ... ELSE ...' structure of some high-level languages. It is, of course, perfectly acceptable to have labels in the conditional parts of a source file. They will be assigned a value if they lie in the part which is actually assembled, otherwise they will be ignored.

The parts that are being conditionally assembled may themselves contain `[`, `|` and `]` directives. In other words, conditionals may be nested. A skeletal example is given below:

```
        [       expr1
        .............
        .............
        [       expr2
;Assembled if expr1 :LAND: expr2
        .............
        .............

;Assembled if expr1 :LAND: :LNOT: expr2
        .............
        .............
        ]
;Assembled if expr1
        .............
        .............

;Assembled if :LNOT: expr1
        .............
        .............
        ]
```

8.1 Logical expressions

What constitutes a <logical expression> will now be described. You have already come across a number of them, for example:

```
"SP" = "P"
```

The '=' symbol above is a relational operator and logical expressions are formed by using these. MASM provides a whole range of relational operators:

Operator	Meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<> or /=	not equal to

All of these take two operands which can be either string or arithmetic expressions. For arithmetic expressions, the meanings of the relational operators are as might be expected:

```
12+4   > 10   yields   TRUE
42    <> 42   yields   FALSE
```

and so on. Of course, actual occurrences of these operators would involve non-constant operands.

Using strings with relational expressions is a little different and great care should be taken since the results of relational tests are not those that would be obtained in, for example, BASIC. The two cases of <> and = are fairly straightforward:

```
"A" :CC: "BC" = "ABC"   yields   TRUE
    "A"       = "B"     yields   FALSE
  "XYZ"      <> "XY"    yields   TRUE
```

So two strings are only counted as equal if they are exactly the same in all characters. For >= and <= the string interpretations read:

s1 >= s2 if s2 is a leading substring of s1
s1 <= s2 if s1 is a leading substring of s2

where s1 and s2 are string expressions. s1 is a leading substring of s2 if the first (:LEN: s1) characters of s2 are the same as s1. In BASIC, this could be expressed as follows:

```
IF INSTR(s1$,s2$)=1 THEN REM s2$ is a leading substring of s1$
```

Here are some examples of the use of these operators with strings:

```
"A"    <= "A"    yields  TRUE
""     <= "A"    yields  TRUE
"A"    <= "B"    yields  FALSE
"ABC" >= "AB"    yields  TRUE
```

The > and < operators are the same, except that the case when the operands are equal yields 'FALSE'. This is shown in the following BASIC example:

```
IF INSTR(s1$,s2$)=1 AND s1$<>s2$ THEN REM s2$ < s1$
```

Here are some examples of these operators:

```
"A" < "A"    yields  FALSE
"A" < "AB"   yields  TRUE
""  < "A"    yields  TRUE
"B" > "A"    yields  FALSE
```

Logical expressions formed by using the relational operators can be combined with the Boolean operators :LOR:, :LEOR:, :LAND: and :LNOT:. These are the same as the binary logical operators :OR:, etc., but with the prefix L, which indicates that they are used with logical values rather than numbers. The actions of these operators are as expected, that is,

```
L1 :LAND: L2   is TRUE iff L1 AND L2 are TRUE
L1 :LOR:  L2   is TRUE if L1 OR L2 is TRUE
L1 :LEOR: L2   is TRUE if either, but not both,
                  of L1 or L2 is TRUE
   :LNOT: L1   is TRUE iff L1 is FALSE
```

(iff being 'if and only if').

The order of precedence of these operators is such that brackets are rarely used in conditional expressions; they are lower than the relational operators, which in turn have lower precendence than the arithmetic operators, so a complex expression such as:

```
a+b <= add1 :LAND: add1 <> start
```

has the bracketed meaning of

```
((a+b) <= add1) :LAND: (add1 <> start)
```

The :LNOT: operator has very high precedence, in common with the other unary operators, such as -(unary minus) and :NOT:

8.2 Global and local variables

The only symbols discussed so far have had a fixed value which is defined at some time during the first pass of MASM through the source file. The exception to this rule is the case of macro parameters, which take on the values assigned to them at a given call of the macro. It is possible, however, for you to define variable symbols whose values can be updated throughout the assembly process. These come in two varieties: local and global variables. Local variables are accessible only within the macro definition to which they belong, whereas global variables have the whole of the source file for their scope.

Before variables can be used they must be declared. This is done using one of the following directives:

Directive	Meaning
GBLA	Define a Global Arithmetic Variable
GBLL	Define a Global Logical Variable
GBLS	Define a Global String Variable
LCLA	Define a Local Arithmetic Variable
LCLL	Define a Local Logical Variable
LCLS	Define a Local String Variable

If these directives are classed as <directive>, then the general form of a variable declaration is as follows:

```
<directive>   <variable name>
```

where <variable name> obeys the same rules as macro parameters. The first three directives shown above are used in the main part of the source file and the last three are used within macro definitions. Some examples are shown below:

```
GBLA   $usage     ;A global arithmetic variable
GBLL   $flag      ;A global logical variable

MACRO
FRED
LCLS   $str       ;A local string variable
.............
.............
MEND
```

When a variable is defined, it is set to the default value for that type. More precisely, arithmetic variables are set to zero, logical ones to FALSE and strings to null. To give a variable a value, one of the SET directives must be used. There is one of these for each type of variable and their format is as follows:

```
<variable name> <SET directive> <expression>
```

The SET directives are SETA, SETL and SETS for arithmetic, logical and string variables respectively. Examples of them are given below:

```
$count   SETA   $count+1
$modelb  SETL   memsze = &8000
$errmsg  SETS   "MACRO Parameters wrong"
```

Once it has been given a value, a variable can be used in the appropriate type of expression, just like a normal symbol. The following example shows how the last three variables could be used:

```
space    #      $count+1
         [      $modelb
;These will be assembled if $modelb is TRUE
         .............
         .............

;Otherwise, these will be assembled
         .............
         .............
         ]
mesg     =      "$errmsg"
```

Notice that string variables behave similarly to macro parameters (see Appendix A for an explanation of the exact mechanism used for substitution). The substitution is a literal one into the source text, so that the third example above needs the quotes("") around "$errmsg". The alternative would be to insert the quotes when the string is defined, and this method is shown below:

```
$errmsg  SETS   """MACRO Parameters wrong"""
mesg     =      $errmsg
```

This use of variables means that you could say:

```
$label   SETS   "lab1"
$label   LDAIM  &FF
```

and this would be assembled as follows:

```
lab1     LDAIM  &FF
```

Since macros may be called from a number of places in the source program, it is important that each call creates unique labels. Consider the macro definition given below, together with some calls to it:

```
         MACRO
$label   INC2   $addr
$label   INC    $addr
         BNE    lab1
         INC    $addr+1
lab1     MEND
         ............
         ............
         INC2   &1200
         ............
         ............
         INC2   count1
```

Here, the first call of INC2 will create a label 'lab1' which will correctly mark the next point in the program. However, the second call to INC2 will also try to create this symbol, so MASM will produce a 'Symbol already defined' error. One way around this problem is to use a global variable which can be incremented every time the macro is used, thus giving a series of unique labels. In fact, two variables have to be used as you can't do arithmetic on string variables. In the listing below, '$count' is a global arithmetic variable which is incremented by the macro INC2. This macro contains a local string variable '$lab' whose value is obtained from '$count'.

```
        GBLA   $count
$count  SETA0          ;Not necessary

        MACRO
$label  INC2   $addr
        LCLS   $lab
$label  INC    $addr
$lab    SETS   "inc":CC: (:STR: $count :RIGHT: 3)
$label  INC    $addr
        BNE    $lab
        INC    $addr+1
$lab                   ;Local label
$countSETA$count+1 ;Increment counter
        MEND
        ............
        ............

update  INC2   count
        ............
        ............
        INC2   &70
```

This time, the label '$lab' is derived from a complex-looking expression. It is made up from the letters 'inc', followed by the rightmost three characters from the conversion of '$count' into a hex string. Thus, the first expansion of INC2 would produce the label 'inc000', the next one 'inc001', and so on. The eleventh one would be 'inc00A', not 'inc010'.

8.3 Routines and local labels

In section 8.2, a global variable was used to generate a unique label for use within the body of the macro definition. It is possible, however, to specify explicitly the scope of certain labels. These labels are numbers in the range zero to 99, and they belong to a particular routine as defined by the ROUT directive. For example, look at the following code:

```
mult    ROUT                ;8*8 bit multiply
        LDA    op1          ;Get first operand
        BEQ    #F01mult     ;Skip if times 0
        ............
00      ROLA                ;Times two
        BCC    #B00mult
        ............
01      RTS                 ;And return
div     ROUT
```

Here, the routine 'mult' is defined as the instructions which lie between the two ROUT directives. Any two-digit labels in this range will be treated as being local to 'mult'. A reference to a local label is of the form:

```
#<options><label num><routine name>
```

and two examples are given in the code above, namely '#F01mult' and '#B00mult'.

The <routine name> part is optional, and is simply the name of the routine to which the specified label belongs. <label num> is a two-digit number, and <options> is a sequence of zero to two characters which tell MASM where to look for the label. It is made up of two parts and the first character is defined as follows:

Character	Meaning
Nothing	Look backwards or forwards for the label
B	Look Backwards only for the label
F	Look Forwards only for the label

The last character is defined as follows:

Character	Meaning
Nothing	Look at this macro level and above (nearer to the source level)
A	Look at any macro level
T	Look at this macro level only

Usually, the default cases will suffice when specifying a label, so you may find yourself using things like:

```
JMP    #12name
```

quite frequently. Note that local labels and routines can be used anywhere, not just within macro definitions.

Note, also, that it is possible to doubly define local labels without MASM generating an error message. For example:

```
ZAP     ROUT
01      LDAIM 0
01      LDAIM 0
```

Unless this is done deliberately and with great care a number of errors will be caused.

9 Repetitive assembly

9.1 The WHILE...WEND loop

Given the ability to vary the value of symbols throughout an assembly, it becomes possible to implement some kind of 'looping' facility; in other words, it is possible for you to assemble a group of instructions repeatedly until some condition becomes false. MASM achieves this with the two directives WHILE and WEND. The format of the loop is:

```
        WHILE <logical expression>

        ;Assemble these instructions
        ............
        ............
        ............
        WEND          ;Marks the end of the loop

        ;Carry on assembling as usual.
        ............
        ............
```

WHILE...WEND loops can appear anywhere in a source program, not just within a macro definition. It is usually convenient, however, to put such loops within a macro, rather than embed them in the main program. Below is an example of a WHILE...WEND loop. It emulates part of the MOS clear screen routine in that it generates instructions to store the contents of the accumulator in 80 pages of memory. This memory is indexed by the X register.

```
ramtop  *       &8000
onek    *       1024

        MACRO
$label  CLSMAC
$label  LCLA    $addr
$addr   SETA    ramtop-20*onek
        WHILE   $addr <> ramtop
        STAAX   $addr
$addr   SETA    $addr + &100
        WEND
        MEND
```

When called, CLSMAC will generate 80 STAAX instructions and this method is obviously much less error-prone than typing in all of the lines separately. Notice that as the test for the WHILE condition is made at the top of the loop, there is a possibility that no instructions at all will be generated; in other words, there is a possibility that the body of the loop will never be reached. A contrived example is given below:

```
        WHILE 1 > 2
```

In this example, one is never going to be greater than two, so all of the loop's instructions will simply be skipped.

Another application of the WHILE...WEND loop is to generate a number of instructions which varies according to the value of some macro parameter. For example, consider a multiple precision rotation to the right:

```
        MACRO
$label  RORN  $addr,$bytes=2
        CLC
$label  LCLA  $count
$count  SETA  0
        WHILE $count <> $bytes
        ROR   $addr+$count
$count  SETA  $count+1
        WEND
        MEND
        ............
        ............
        RORN  &1200,4
        ............
        ............
        RORN  zerop,
```

The first example call to RORN rotates the four bytes from &1200 to &1203 right by one bit; the second one rotates the two bytes 'zerop' and 'zerop+1' by one bit. Notice the use of the default parameter in the second example.

9.2 The MEXIT directive

A further directive which may appear in a macro definition is MEXIT. This causes an exit to be made from the current macro as if the MEND instruction had been encountered. It can be used to terminate WHILE loops or IFs if some abnormality has occurred. An example of its use is given below:

```
        MACRO
$label  PRINT $string
$label  [     "$string"=""   ;Null string?
        MEXIT                ;Yes, end
        ]
        JSR   prints         ;Otherwise do
        =     "$string"      ;macro
        NOP
        MEND                 ;And terminate

        PRINT Hello there    ;No quotes needed
        ............
        ............
        PRINT                ;This will do MEXIT
```

Note that MEXIT is the only way to leave a macro definition from within an IF ([) or WHILE. You must not use MEND within either of these constructs.

10 Trapping errors in source code

It is good practice when developing programs to check macro parameters. For example, the coding in section 9.2 where PRINT was called with a null string could be treated as an error, if required. To aid error trapping, the ASSERT directive is provided and this can be used either inside or outside macros. It has the format:

```
ASSERT <logical expression>
```

If the <logical expression> yields a TRUE result, nothing happens. A FALSE result, however, will cause MASM to stop what it is doing and return control to the command level. Thus, the macro PRINT might have the line:

```
ASSERT :LEN: "$string" > 0
```

When this line is met during the second invocation of PRINT in section 9.2, the logical expression would be FALSE (since :LEN: "" is zero), so assembly would terminate with an 'Assert failed at line XXXX' error. MASM's check on the logical expression is made on the second pass only, so any symbol can be used in the expression, no matter where it is defined in the program.

A similar device is the ! directive. This has the following format:

```
!       <arithmetic expression>,<string>
```

During the each pass of the assembly, the arithmetic expression item is evaluated. If it is zero, nothing happens. If, however, the item yields a non-zero result on the first pass, the string will be printed out along with the message 'stopped at line XXXX', and assembly will stop. During the second pass, the string will be printed, but assembly will not stop. Because the expression is evaluated on both passes, it must not contain any forward references.

11 Creating source files using EDIT

This chapter describes EDIT, the program editor which was introduced briefly in chapter 3, 'Developing a simple MASM program'. As noted earlier, the function key card should be placed above the function keys when using EDIT, since they are used a great deal.

11.1 Entering EDIT

EDIT is called by issuing the operating system command *EDIT. So, in order to use the editor, you must be in a situation where issuing a command line is possible. This could be, for example, in the command mode of MASM or one of the cross-referencers, or from within BASIC. Note that some ROM utilities respond to *EDIT and if this happens it will be necessary to type */EDIT to load the editor from disc. It is also possible to load a file to be EDITed at the same time as calling EDIT by entering *EDIT <filename> or */EDIT <filename>.

When entered, EDIT selects screen mode 0 as this gives the maximum number of characters that the BBC Microcomputer can display (80 columns by 32 lines). There is no 'memory overhead' associated with the screen mode when using EDIT as it only runs on the Second Processor. The capacity for text is over 47000 characters, regardless of the mode.

11.2 Adding text

When the EDIT command is given, the screen clears apart from the very top and bottom lines. On the top line is an inverse video * which marks the end of text. The fact that it is at the start of the page implies that there is no text being edited currently. Just below the marker is the flashing cursor. This always marks where characters will be put into the text.

The bottom line of the screen contains the 'status line'. This gives you information about various modes of operation which are described later.

To add text, simply type it in. The end of text marker will be moved along to make room for the new text. To illustrate some of the features of the editor, the entry of the simple MASM program listed below will be described in detail:

```
oswrch  *       &FFE3
        ORG     &1B00

cset    LDXIM " "
loop    TXA
        JSR     oswrch
        INX
        CPXIM "~"
        BNE loop
        RTS
        END
```

First type the top line exactly as shown above (the exact number of spaces is not essential, but at least one must be used in each case). If you make a mistake, press the DELETE key to erase the incorrect characters. Notice that pressing DELETE removes the character before the flashing cursor. An alternative is to press COPY to delete the character at the cursor position. When you reach the end of the line, press RETURN. This will move the cursor to the start of the next line.

The second line of text is indented. To achieve this, simply press the Space Bar the correct number of times. It is also possible to use the right-arrow cursor key, though pressing the Space Bar is probably more convenient.

The third line of the program is blank. Producing blank lines simply involves pressing RETURN if the cursor is at the start of the line. In this example, it means pressing RETURN twice after the '0' of line two instead of just once.

Line four can be typed as normal, as can line five. The next few line are indented and, to save typing here, the TAB key can be used.

The TAB key can be used in two ways: by default it uses tab stops which are eight characters apart. However, we need the other mode which is accessed by pressing SHIFT TAB. TAB then acts as described below. To return to the default mode SHIFT TAB should be pressed again.

The first line ('JSR oswrch') should be typed as normal, pressing the Space Bar to achieve the indentation. When you press RETURN to get on to the next line, instead of typing spaces, press TAB instead. This moves the cursor so that it is underneath the 'J' of the line above. In this mode, TAB moves the cursor so that it is underneath the first non-space character of the line above. It is useful, therefore, when many lines need to be indented by the same amount.

You can now type in the remaining lines, using TAB in the same way. At this stage, assuming no errors have been made, the file could be assembled, as described in chapter 12, 'Using MASM to assemble your programs'.

11.3 Using the cursor keys

Now that we have some text to manipulate, you can start using the various facilities of EDIT. First of all, you need to be able to move around the screen, changing various parts of the text. To do this, the four cursor (arrow) keys are used. If you press one of these keys, the cursor moves one space in the direction indicated, that is one character to the left or right for the left- and right-arrow keys respectively, and one line up or down for the up- and down-arrow keys (scrolling the screen when the cursor gets near the top or bottom).

As an example, change the label 'cset' on line four above to 'charset'. To do this, use the arrow keys to position the cursor on the 's' of 'cset'. Then type the extra characters 'h', 'a' and 'r'. The text after the cursor will be shifted to the right to make room for the new text. This happens when the editor is in insert mode. There is another mode called overtype mode which is described in section 11.8.

Deleting small sections of text is similarly easy. Move the cursor to the character after the item to be deleted, then press DELETE the appropriate number of times. Again, the text will move, this time to the left. Notice that if you move the cursor to the beginning of a line and press DELETE, the current line will join up with the one above it. This happens because you have deleted the (usually invisible) carriage-return character which separates the two lines.

The cursor keys can be used in combination with SHIFT or CTRL to make more drastic movements around the screen. If SHIFT is pressed with the up or down keys, the text will scroll up or down by a 'page'. The length of a page depends upon the screen mode in use. This facility is useful for moving rapidly over a large region of text. SHIFT pressed with the left and right keys moves the cursor backwards and forwards by one word (a word in this context is a sequence of alphanumeric (but not _) characters separated by groups of any other characters).

The CTRL key can be used with cursor up or cursor down to reach the top or bottom of the text, respectively. Pressing CTRL with cursor left or right moves the cursor to the start and end of the current line respectively.

11.4 The cursor edit mode

The use of cursor editing with the COPY key in the editor is slightly different to its use in, say MASM command mode. Normally, to copy some text when typing in a MASM command, the arrow keys are used to move the copy cursor to the required part of the screen, then COPY is pressed to do the copying. Copy mode ends when RETURN is pressed.

In the editor, the only difference is that instead of one of the cursor keys initiating copy mode, SHIFT COPY must be pressed. The cursor keys can then be used to move to the text to be copied, and COPY can be pressed the appropriate number of times. Copy mode is terminated when ESCAPE (rather than RETURN) is pressed.

This technique is very versatile as it means that the cursor keys may be used to move around the screen when inserting or deleting text, and also to move the copy cursor around as usual. The obvious application of copying is to duplicate identical or nearly identical lines.

When in cursor edit mode, the status line reflects this by containing '`Cursor Editing`' in inverse video.

11.5 The function keys

It can be seen that quite a lot of editing may be achieved simply by using the keys discussed so far. However, EDIT provides functions which makes the manipulation of program text even easier. These functions are accessed by using the red function keys at the top of the keyboard. The function key card gives some clues as to how they are used and the following text describes the functions in detail.

As some of the keys provide more than one function, it is sometimes necessary to press SHIFT or CTRL at the same time as the key itself. For example, pressing function key 0 searches for a given line number, whereas SHIFT function key 0 switches the display of carriage-returns on and off. In the text below, 'function key 'n' will be abbreviated to 'fn', where 'n' is 0 to 9.

If you want to type a string of characters several times, you can do so using the function keys. Pressing the function keys together with CTRL and SHIFT enables the soft key strings to be accessed. Thus, if you type the command '*KEY0 LDAIM' from command mode, pressing CTRL SHIFT f0 in the editor will produce this string. Note, however, that this is not the default situation and the OS command *FX228,1 must be issued to enable strings to be generated in this way.

11.6 Changing display mode

As mentioned above, the editor uses MODE 0 when it is first called. The command SHIFT f5 is used to change mode. When it is issued the prompt:

```
New mode:
```

is given, to which you should reply with a digit in the range 0-7 followed by RETURN. There is a slight difference in the way that characters are displayed in MODE 7; for example, markers are shown as inverse video digits in the soft modes (0-6) but as plain white 'blobs' in MODE 7. As mentioned above, using low memory-cost modes gives no advantage when using EDIT.

An alternative to specifying 0-7 for the mode is to type the letter 'D' instead. This causes EDIT to use its 'descriptive mode', which is MODE 0, but with a great deal of help information displayed at the top of the screen. For example, a layout of the function keys is shown together with their actions, and whenever a function key is pressed, detailed information about what it does is shown in a window.

Descriptive mode is very useful when you are becoming familiar with the editor and can also serve as a reminder later on, when you have forgotten a particular detail. There are, however, a couple of disadvantages with this mode. One of these is the lack of space for text on the screen, and the other is that the screen scrolls more slowly than usual. Nevertheless, descriptive mode should make the process of learning a new editor easier than it might otherwise be.

A condensed version of mode 'D' is mode 'K', which only prints the key layout without the other help information.

11.7 Saving, loading and inserting text

It is obviously useful to be able to save and load files from the editor. There are three commands for manipulating files from within EDIT (pressing a function key will be treated as issuing a command in this chapter). The commands are:

f2 - Load the text from a file
f3 - Save text to a file
SHIFT f2 - Insert text from a file.

In addition, there are

f9 - Restore old text
SHIFT f9 - Delete the text being edited

Pressing f3 produces the prompt:

```
Type filename to save:
```

In response you should give the name of the file into which the text should be saved. If an error occurs, it is reported and you will be prompted to press the ESCAPE key to continue.

If you press RETURN instead of typing a filename, the editor will look for a name at the start of the text. The filename should be preceded by the '>' character and terminated by a space or carriage-return. The '>' should occur within the first 128 characters of the text. In MASM programs, the name should be in a comment so that the assembler doesn't try to interpret it as part of the program:

```
; >test
        ORG    &1B00
```

This will make the editor use the filename 'test' whenever RETURN is pressed in response to a filename prompt.

Another alternative is to press COPY then RETURN. This will use the 'current' filename, ie the name last used in a load, save or insert command. The current filename is updated every time one of these commands is executed.

If you wish, you can save only part of the file you are editing. The method you should use is to put a marker at the start of the portion of text to be saved, move the cursor to the end of the section, and then use f3 as described above. The setting and deletion of markers is described in section 11.10.

Text is loaded using the command f2. The prompt this time is:

```
Type filename to load:
```

This command wipes out any text already in the machine. Errors are reported in the manner described above. Additionally, if the file is too large to read into the computer, the error 'File too long' is given (unless the Cassette Filing System is in use). As with save, RETURN and COPY RETURN may be used instead of a proper filename.

SHIFT f2 inserts a file into a particular place in the text. Its prompt is:

```
Type filename to insert:
```

to which you should reply with a filename, RETURN or COPY RETURN. The file will be read into the text at the current cursor position. Text before the cursor will be unaffected; characters at and after the cursor will be shifted up in order to make room for the file, and the cursor will be placed at the start of the text just read in. This command is useful for inserting frequently-used procedures into programs.

Note that when text has been inserted using SHIFT f2, the current filename will become the name of the file that is inserted, not that already in the machine. Hence, COPY RETURN should be used with care in these circumstances.

Pressing SHIFT f9 causes a prompt to the effect that all the current text will be deleted if any key is pressed. If no key is pressed within a short time (about ten seconds), or if ESCAPE is pressed, then the previous editing mode is resumed. SHIFT f9 is fairly drastic, so the command should obviously be used with some caution. If you press SHIFT f9 (and another key) or BREAK by mistake, you can restore the text using f9. Using SHIFT f9 twice in succession deletes the text irretrievably.

11.8 Insert and overtype modes

So far, only insert mode has been discussed. When the editor is first called, text typed at the keyboard is inserted into the file by shifting everything after the cursor to the right. In order to replace a word, it must be deleted and then retyped. It is sometimes more convenient to be able to replace characters simply by overwriting them.

In EDIT, you can switch between insert and overtype mode by pressing SHIFT f1. This key acts as a 'toggle' so that if you are in overtype mode, pressing it will put you in insert mode, and if you are in insert mode, pressing SHIFT f1 will put you in overtype mode.

The bottom (status) line of the display shows the current typing mode as '`Insert`' or '`Over`'. Try pressing SHIFT f1 several times to see the effect on the status line.

To see how overtype mode differs from insert mode, press SHIFT f1 until the status line has '`Over`' in it and move the cursor to the start of a line already containing some text. Now start to type. The line will be overwritten by what you typed rather than be moved in order to make room for it.

The way in which carriage-return characters are treated differs between insert and overtype modes. In insert mode, as we have seen, you can delete a carriage-return by moving the cursor to the start of the previous line and pressing DELETE. You may have also discovered that you can split a line into two by moving the cursor to where you want the split to appear and pressing RETURN.

In overtype mode, neither of the above actions is possible. Pressing RETURN merely moves the cursor to the start of the next line. Pressing DELETE at the start of the line will move the cursor up to the end of the next line, but will not delete the carriage-return itself. Thus, you cannot split or join lines in overtype mode. The COPY key and the function keys work in overtype mode exactly as in insert mode.

11.9 Special characters in the text

You can type control characters in the same way as any other character. To distinguish them, they are shown in inverse video in MODEs 0 to 6 and as white 'blobs' in MODE 7. ESCAPE is ignored when typing in text and is used to abort commands such as 'find and replace'.

A special inverse character is RETURN, which is the same as CTRL M. Since a RETURN character appears on the end of every line, showing it as an inverse 'M' would be very distracting. Normally, then, EDIT does not show carriage-returns. However, pressing SHIFT f0 will make them visible. Pressing it a second time will render the inverse 'M's invisible again, so it acts as a toggle (like SHIFT f1 for insert/overtype mode).

By making the RETURNs visible, you can see if there are any unnecessary trailing blanks on a line (these can occur when the cursor is past the end of the normal text and the Space Bar is pressed inadvertently).

The character DELETE (whose code is 127) is shown as a small white rectangle in all modes. Characters with codes greater than 127 are displayed as normal (so user-defined characters and MODE 7 colour codes have their usual effect). However, characters with codes between 127 and 255 may only be entered from the keyboard using *FX228 and SHIFT CTRL function keys (see the *BBC Microcomputer System User Guide* for details). Note that the codes between 128 and 192 are used by the function keys and so should not be entered except as described in section 11.13 'Using command macros'.

11.10 Dealing with blocks of text

There are five operations which can be performed on a block of text. These are:

SHIFT f8 - Delete a block of text
f7 - Copy a block of text
SHIFT f7 - Move a block of text
f3 - Save text to a file (described in section 11.7)
f5 - Global find & replace (described in section 11.12.3)

In addition, two commands are needed to set and reset markers, which are used in conjunction with the above commands. These are:

f6 - Set marker
SHIFT f6 - Clear marker(s)

11.10.1 Deleting a block

To delete a block of text, two 'delimiters' are needed, one at the start and one at the end of block. One of the delimiters is the cursor and the other is a marker. Consider the text:

```
The quick
brown fox jumps
over the
lazy dog.
```

Suppose you want to delete the middle two lines. This is accomplished thus: move the cursor to the 'b' in 'brown'. Press f6. This sets marker 1 (an inverse '1'), which will act as the first delimiter, at the cursor position. Then move the cursor to the 'l' in 'lazy' and press SHIFT f8. This will delete the required two lines.

Notice the exact characters which are deleted: from the first delimiter inclusive to the last delimiter exclusive, so the first delimiter should be placed at the first character in the block and the second delimiter should be just after the last character.

Note also that the marker can equally well be the second delimiter. The two lines could have been deleted by setting the marker at the 'l' in 'lazy' and moving the cursor up to the 'b' in 'brown' before pressing SHIFT f8. The result would be the same.

11.10.2 Copying a block

As described earlier, you can copy text using the COPY key. This can be very tedious if there is a lot to copy, so a way is provided of duplicating a whole block of text. Again, there are two delimiters needed to mark the area and a way of marking the destination of the copied block.

Suppose you want to copy the text:

```
LDAIM ">"
JSR   oswrch
```

so that it occurs twice in succession. This is done as follows: move the cursor to the 'L' of 'LDAIM' and press f6. This sets marker one. Move the cursor to below the 'J' of 'JSR' and press f6. This sets marker two. Move the cursor to the required destination to (this must not be between the two markers or a 'Bad marking' error will occur, but it can be on the second marker). In the present example the cursor should be moved to marker two. Now press f7. This produces a copy of the required lines.

Notice again that the block copied lies between delimiter one inclusive and delimiter two exclusive and that, in this case, the markers are still present.

11.10.3 Moving a block

Moving is equivalent to copying a block then deleting the original. The block to be moved is delimited by two markers and it is moved to the position of the cursor when SHIFT f7 is pressed. Again it is illegal for the cursor to be within the region delimited by the markers when the move command is given.

11.10.4 Deleting the marker(s)

The delete block and move block commands automatically delete any markers present. However, it is sometimes desirable to delete the markers without having to execute a 'block' command. Pressing SHIFT f6 will delete the active marker(s).

Apart from when cursor editing is active, the status line indicates how many markers there are in the text. You can have at most two.

11.11 The scroll margins

You may have noticed that when the cursor is moved near to the top or bottom of the screen, the text scrolls and the cursor stays on the same line. This usually happens when the cursor tries to move above the fourth line from the top or below the fourth line from the bottom. These two lines mark the so-called 'scroll margins' and may have their positions altered.

The commands to set and reset the scroll margins are:

CTRL f6 - Set the top scroll margin
CTRL f7 - Set the bottom scroll margin
SHIFT f3 - Reset the scroll margins

To set the top scroll margin, CTRL f6 is used. First position the cursor on the line at which you want scrolling to occur when moving up the text, then press CTRL f6. Similarly, to set the bottom scroll margin, move the cursor to the line below which you do not want it to move and press CTRL f7.

Pressing SHIFT f3 will set the margins to the top and bottom of the screen.

Note that, when the very top (or bottom) of the text is reached, the cursor can be moved into the top (or bottom) margin so that the first (or last) few lines may be edited.

If CTRL f6 and CTRL f7 are pressed in succession without moving the cursor, the margins will be set to the same line. The result is that any vertical movement of the cursor will cause the screen to scroll.

The scroll lines are used by various search commands. For example, the f4 search command described later causes its target to be displayed on the bottom scroll line and f0 displays the new line on the top scroll line. Also, when CTRL down is used to move to the end of text, the last line is displayed on the bottom scroll line.

11.12 Finding and replacing text

One of the most useful features of EDIT will now be described. When editing a large file, it is often desirable to find the occurrence of a particular word or phrase, perhaps with a view to changing it to something else. Scanning through by eye is tedious and prone to error. Another requirement is to be able to jump to a given line in the text (this is necessary as MASM gives the line number at which an error occurred). Being able to quickly find this erroneous line speeds up debugging.

EDIT has the ability to find a given line, find and selectively replace one string with another and count all occurrences of a string, optionally replacing it with something else. The relevant commands are:

f0 - Goto a line number
f4 - Find and selectively replace a string
f5 - Globally count and replace a string

11.12.1 Finding a given line

Pressing f0 produces the prompt:

```
At line xx, new line:
```

to which you should type the number of the line to be found (the top line is number one). If the line number specified is greater than the number of lines in the document, a 'Line not found' error is generated. Otherwise, the screen is updated so that the line in question becomes the current line.

11.12.2 Finding and selectively replacing a string

A more general way of searching the file is searching for a particular string. EDIT lets you search for simple strings such as 'begin', but more powerfully for such things as 'all identifiers beginning with A'. The command f4 finds strings and, if necessary, changes them to something else. In this context the string being sought is called the 'pattern' and the string with which it will be replaced is called the 'replacement'.

In response to f4 EDIT will produce the prompt:

```
Find and replace:
```

You should type one of two things: a pattern, followed by RETURN, or a pattern, then a '/' as a separator, then a replacement and finally RETURN. Examples are:

`begin`<RETURN>	(Find occurrences of '`begin`')
`for/FOR`<RETURN>	(Replace occurrences of '`for`)
`=/:=`<RETURN>	(Replace occurrences of '`=`')
`then /`<RETURN>	(Delete occurrences of '`then `')

In the last example, the replacement is a null string, which leads to the pattern being deleted.

The search for the pattern begins at the cursor position, so it's a good idea to move the cursor to the top of the file (CTRL up-arrow) if you want to find all occurrences. When the pattern is located, the editor updates the screen so that the pattern is on the bottom scroll line and prompts with:

```
R(eplace), C(ontinue) or ESCAPE
```

If you press 'R' and specified a replacement then the change is made and the next occurrence sought. If you press 'R' but didn't specify a replacement you will be prompted with 'Replace string:' so that you can give one, then the change will be made and the next occurence sought. If you press 'C', this occurrence is skipped and the next one sought. If you press ESCAPE, the search ends. After the last occurrence has been found the editor returns to normal mode, and displays 'Not found' on the end of the status line to indicate the end of the search.

11.12.3 Globally counting and replacing a string

The command f5 acts in a similar way to the last one, but assumes that if you specify a replacement string, all occurrences of the pattern should be replaced. If no replacement is given, then the number of times the pattern occurs in the file is counted. The prompt for the pattern and (optional) replacement is:

```
Global replace:
```

Typical replies are:

`FRED/JOHN`<RETURN>	(Replace all occurrences of '`FRED`')
`HELLO/`<RETURN>	(Delete all occurrences of '`HELLO`')
`for`<RETURN>	(Count all occurrences of '`for`')

Notice that the only difference between counting occurrences of the pattern and deleting them is whether or not a '/' appears at the end of the line. The f5 command does the search and replace automatically without any prompts. As this is a 'global' search and replace, the search starts from the top of the file, independent of the cursor position (but see the next paragraph). After the search (and replace), the number of times the pattern was found is given on the status line in the form:

```
1234 found
```

It is possible to make the global search and replace slightly less so by setting a marker before issuing the command. If this is done, only text between the marker and the cursor will be affected. It can be useful when, for example, only the occurrences of an identifier in a particular routine need to be altered.

11.12.4 Patterns

The patterns used by the search commands may be regarded as expressions. In fact, the formal name for patterns is 'regular expression'. They may be thought of as having constant parts (literal text) and variable parts (wildcards, ranges, choices, repeats and inversions). This section describes the different parts in detail.

In the examples given so far, the patterns and their replacements have been simple strings. However, by using special symbols in a pattern, it is possible to specify the variable parts mentioned in the list above. The special characters available in patterns are:

`.`	match any character
`@`	match any alphanumeric (0-9, A-Z, a-z, or _)
`#`	match any digit (0-9)
`[xyz]`	match any of 'x', 'y' and 'z'
`a-z`	match any character between 'a' and 'z' (inclusive)
`$`	match the carriage-return character
`¦c`	match CTRL c (c should be a CTRL-key character)
`~c`	match anything but c (which can be a wildcard or a set)
`\c`	match c (with no special meaning attached to c)
`*c`	match zero or more of c (shortest match)
`^c`	match one or more of c (longest match)

A `.` in a pattern will match any single character in the range ASCII 0 to ASCII 255. All of the wildcards may be duplicated, so '`..`' will match any two characters, and so on. `@` is slightly more restrictive and will match those characters which are allowed in identifiers. `#` will match any of the ten characters in the range '0' to '9', that is, the digits.

If neither of `@` or `#` provides a suitable range of characters to match, it is possible to define your own range using `-` . Thus 'A-F' will match any valid hexadecimal letter. Another way is to put several choices inside square brackets. Only one of the characters in the brackets will be matched. For example '`[¦I$ ]` will match either TAB (CTRL I), carriage-return or space. These characters are sometimes known collectively as white-space.

You can, if you wish, combine ranges and choices. For example, to match any hexadecimal character, the choice '`[0-9A-F]`' would be used. Since existing wildcards may be put in a range, this could also be expressed as '`[#A-F]`'. Another example is a pattern to match characters which may occur at the start of a MASM identifier: '`[a-zA-Z]`'. This can be read as 'match any character in the range 'a' to 'z' or 'A' to 'Z''.

Note that when letters are being matched, they are case equated. That is, upper and lower case letters are not counted as different and the pattern 'egg' will match the expected 'egg' as well as the unexpected 'Egg' and 'eGg'. This is useful when editing MASM files as the assembler does not differentiate between upper and lower case letters in identifiers. When letters are used in conjuction with the special symbols ' - ' (for range) and square brackets (for choice), then they are treated 'literally', so the pattern 'a-z' will only match lower case letters between 'a' and 'z'.

$** is a convenient way of putting the carriage-return character in a search string (the RETURN key can't be used for obvious reasons). The vertical bar ¦ has the same meaning as within *KEY and filename strings, that is, 'make the next character a control character'. Thus, '¦@' means ASCII 0, '¦A' means ASCII 1 and so on. '¦M' is the same as '**$', '¦[' is the ESCAPE character, ¦? is DELETE and '¦I' is TAB.

The action of ~ is to match anything but the sub-pattern that follows. Thus '~A' will match anything but 'A' (or 'a'), '~**#**' will match any non-digit and '~A-Z' will match anything that isn't an upper case letter.

The backslash character \ is needed to remove any special meaning from the symbol which follows it. Thus, '**$**' stands for '**$**', not carriage-return, '\¦' prevents the character after the bar from being interpreted as a control character, '**.**' means '**.**' not 'any character', and '\a' means lower case 'a' only, not 'A' as well. \/ is necessary to get the slash itself rather than the delimiter.

Sometimes it is useful to be able to find where a sequence of characters occurs, especially when whatever is matched is to be replaced by something else. A typical example would be to delete trailing spaces at the end of lines. Here, we want to look for zero or more space characters followed by a carriage-return and replace them with just the carriage return. One way of doing this would be to repeatedly use:

```
Global replace: $/$<RETURN>
```

Eventually, all the trailing spaces will be deleted and f5 will come back with '0 found'. However, the global replace command will have to be issued several times: one for each space on the end of the line with the most trailing spaces. What we need here is a mechanism for letting us match any number of spaces in one go. The asterisk '*' performs this task. In a pattern, the sequence '*c' means match zero or more of the character 'c'. For example the trailing spaces on each line can be deleted in one go by:

```
Global replace:   * $/$<RETURN>
```

This can be read as 'replace zero or more spaces followed by a carriage-return with a carriage return'. Preceding a character with an asterisk is called 'forming a closure' over that character, but is usually read, as indicated above, as 'zero or more of'.

It is possible to form closures over any type of pattern, not just simple characters. For example, it is possible to match zero or more digits using the pattern '*#' or zero or more upper-case letters with '*A-Z'.

Notice that it is not very useful to end a pattern with a closure pattern because it always matches the shortest string it can and this includes the null string. An example will make this clearer. Suppose we wanted to replace all strings of the form 'fred' followed immediately by zero or more digits with the string 'fredId'. Our first try might be:

```
Global replace: fred*#/fredId<RETURN>
```

We might expect this to replace 'fred123' with 'fredId'. In fact it will really become 'fredId123'. The reason is that the pattern 'fred*#' will always match just 'fred', as the '*#' part will match zero digits if it can. What we really need here is a way of matching all of the digits that come after 'fred', instead of none of them. To do this, another form of closure is provided: '^'. This acts similarly to '*', but differs in two ways: firstly it matches one or more of the following character, and secondly matches the longest string possible rather than the shortest.

```
Global replace: fred^#/fredId<RETURN>
```

You may realise that the pattern '^c' matches exactly the same strings as 'c*c~c', where 'c' is any sub-pattern that may occur after a '*'. However, the '^' form is more convenient, especially when replacing rather than just searching for strings.

It can be seen that the closure (also called 'multiple match') facility is very powerful, but must be used with care. In order to use '*' to match one or more of a character (as opposed to zero or more) the format:

```
Find and replace: c*c<RETURN>
```

should be used. Below are some more example patterns.

If you want to find all identifiers beginning with A or a, you can use the following format:

```
Find and replace: A*@~@<RETURN>
```

Here, the 'A' matches the first character; the '*@' matches the rest of the identifier up to the non-alphanumeric character matched by '~@'.

If you want to find all integer constants, you can use the following format:

```
Find and replace: ^#<RETURN>
```

An integer is, of course, just one or more digits.

If you want to find all non-null strings in a BASIC program, you can use the following format:

```
Find and replace: "^."<RETURN>
```

Here, the '"' matches the opening quote; the '^.' matches one or more characters and the '"' matches the end quote. Note, however, that it will not find '""""'.

To find all the '\'s in a file, you can use the following format:

```
Find and replace: \\<RETURN>
```

Two backslashes are needed as '\' itself is a special character. It therefore needs a preceding '\' to 'quote' it.

If you want to find all MASM '*' directives, you can use the following format:

```
Find and replace: $^@* \*<RETURN>
```

The '*' directive must be preceded by a label, and a label must start at the first character on the line. Thus, '$^@' matches the identifier at the start of the line; '* ' matches the zero or more spaces separating it from the directive, and '*' matches the directive itself, the backslash taking away the asterisk's special meaning.

To find all blank lines, you can use the following:

```
Find and replace: $* $<RETURN>
```

A blank line is simply where a carriage-return is followed by another one with only zero or more spaces intervening. The first '$' matches a carriage-return; the '* ' matches zero or more spaces; the second '$' matches the carriage-return of the blank line. Note that this pattern will only find the first of one or more blank lines.

If you want to find all the integer variables in a BASIC program you can do so by using the following:

```
Find and replace: ^@\%<RETURN>
```

The '^@' matches one or more alphanumerics; the '\%' matches the percent sign at the end. The percent sign is 'quoted' with backslash because, although it has no special meaning in patterns, it does in replacement strings. In the editor, all special characters that may be used in pattern matching have to be quoted if they are to be used as themselves, whether their special meaning is relevant to the context or not.

Actually, the pattern as given will match something like '123%', which isn't a proper identifier. The problem is that '@' matches digits. To be strictly accurate, we should use:

```
Find and replace: [a-zA-Z_£]*@\%<RETURN>
```

so that the first character has to be a letter, an underscore or a pound sign, all of which may start a BASIC identifier.

If you want to match any control character, you can use the following:

```
Find and replace: ¦@-¦_<RETURN>
```

'¦@' is CTRL @, the lowest valued control character (ASCII 0) and '¦_' is CTRL _, the highest control character (ASCII 31).

To match any word in a text file, you can use the following format:

```
Find and replace: ^[a-zA-Z]<RETURN>
```

This pattern stems from the simple definition of a word: the longest sequence of one or more upper or lower case letters.

If you want to match a hexadecimal constant, you can use the following:

```
Find and replace: \&^[#A-F]<RETURN>
```

This is simply '&' followed by one or more hex characters. Again, '&' is quoted as it has a special meaning in replacements.

Two useful global search without replace commands are:

```
Global replace: .<RETURN>
Global replace: $<RETURN>
```

These display the number of characters and lines in the file, respectively, on the status line.

11.12.5 Replacements

Although it is obviously very useful to be able to find patterns of the type described above, it is even more useful to be able to replace them with something else. A replacement can be either a literal string, such as 'fred', or it may contain various special characters. These are as follows:

$	Stands for a carriage return
¦ c	Means CTRL c
\ c	Means c (with no special meaning)
&	Means whatever was matched by the pattern
% n	Means field number n

In the list above, 'c' stands for a character and 'n' stands for a digit between 0 and 9. The first three items have already been encountered in patterns.

The ampersand '&' is a character which only has a special meaning in the replacement string (though, as we saw earlier, it still needs to be quoted in patterns). It means 'whatever the pattern matched'. Clearly, if the pattern was just a literal such as 'until', then that is what '&' will stand for. However, when wildcards and repeats are used in the pattern, it is not possible to know exactly what was matched. Suppose you want to duplicate all digits so that, for example, '1' becomes '11' and '123' becomes '112233'. This could be achieved with the following global replace:

```
Global replace: #/&&<RETURN>
```

Whatever is matched by the pattern will become the ampersand in the replacement. Notice that although global replacing was used in the example above, the same replacement string could have been used in a selective replace just as legally.

The final special replacement character acts as a more restricted version of '&'. The percent sign '%' is followed by a digit n between 0 and 9. This combination stands for the nth field of the pattern. A 'field' is defined as a wildcard character, a multiple match (that is, a symbol preceded by '*' or '^'), an inverted match (that is, a character preceded by '~'), a range ('a-z') or a choice ('[135791]'). The fields are numbered from zero on the left. Some examples might make this clearer:

```
Find and replace: A*@+.<RETURN>
```

Here, the characters matched by the '*@' are field 0, and the character matched by the '.' is field 1.

```
Find and replace: ##~@* <RETURN>
```

Here, there are four fields: '#', '#', '~@' and '* ' respectively.

```
Find and replace: ~*#^#<RETURN>
```

This matches any number of non-digits followed by one or more digits. The first field is '~*#' and the second field is '^#'.

Below are some examples of using the fields in replacement strings:

If you want to reverse the order of alternate characters, you can do so as follows:

```
Global replace: ../%1%0<RETURN>
```

If you want to delete the % sign from integer variables in BASIC', you can do so as follows:

```
Global replace: ^@\%/%0<RETURN>
```

(Note that the quoted '%' sign in the pattern is for the character at the end of BASIC integer variables.)

If you want to put a '$' after all variables beginning with 'S_' and delete the 'S_', you can do so as follows:

```
Global replace: S_^@/%0\$<RETURN>
```

(Again, note the quote sign '\' before the '$' so that it is not treated as a special character by EDIT.)

Notice that when replacing (rather than simply finding) patterns, you have to be very precise about what marks the end of a string. For example, to insert '_1' at the end of all identifiers, it is insufficient to use:

```
Global replace: @*@/&_1<RETURN>
```

This will, in fact, cause '_1' to be placed after the first letter of identifiers as the '*@' will match the bare minimum, that is, nothing at all. It is necessary to explicitly find the last character in the identifier and use the following:

```
Global replace: @*@@/%0%1_1%2<RETURN>
```

This time the '_1' is placed between the '*@', that is, the tail of the identifier, and the '~@' that is, the non-identifier character which marks the end.

More simply, using '^', we could say:

```
Global replace: ^@/&_1<RETURN>
```

though this uses a less strict rule to determine what constitutes an identifier.

11.13 Using command 'macros'

As mentioned earlier, you can generate strings from function keys if CTRL and SHIFT are used together with the key. Since EDIT commands are really just characters above 127, if these are put into function key definitions you can issue several commands at a single keystroke.

Most usefully, this facility may be used to execute several global replaces in quick succession. The code of the global replace command is 133. To obtain this in a function key string '!!!E' is used. Thus, to make the string produced by f0 replace all tabs with spaces, the following sequence of commands must be used:

First, press f1 to enter an operating system command. Then type the following (note that you don't need an * before the command:

```
FX228,1
KEY0!!!E!I/ !M<RETURN>
<RETURN>
```

The first line sets up the CTRL SHIFT function key status so that this combination generates strings. The next line programs key zero so that it contains the characters necessary to perform the global replace: '!!!E' issues the command (as if you'd typed f5); the '!I/ ' is the pattern and replacement part, and '!M' is the RETURN at the end of the command. The third line returns you to edit mode.

When CTRL SHIFT f0 is pressed, the string defined above will be produced. Because you are holding CTRL and SHIFT down at the same time, the carriage-return at the end won't be printed. The command will not be executed until you let go of the keys.

Obviously, you can build up quite useful command strings using the function keys. Here is a list of the strings required to generate various commands:

Command	Alone	+ SHIFT	+ CTRL
f0	!!!@	!!!P	!!<SPACE>
f1	!!!A	!!!Q	!!!
f2	!!!B	!!!R	!!"
f3	!!!C	!!!S	!!#
f4	!!!D	!!!T	!!$
f5	!!!E	!!!U	!!%
f6	!!!F	!!!V	!!&
f7	!!!G	!!!W	!!'
f8	!!!H	!!!X	!!(
f9	!!!I	!!!Y	!!)
TAB	!!!J	!!!Z	!!*
COPY	!!!K	!!![	!!+
left	!!!L	!!!\	!!,
right	!!!M	!!!]	!!-
down	!!!N	!!!^	!!.
up	!!!O	!!!_	!!/
DELETE	!?	!?	!?

Notice that you can generate cursor moves since, within the editor, the cursor keys are treated as extra function keys. As another example, to set-up a key to insert a file called 'decl' the following would be used:

```
<f1>
KEY0!!!Rdecl!M<RETURN>
<RETURN>
```

Note that it is possible to write a command file using EDIT which may then be '*EXECed'. The method is to put, say, 'f5' in the file as the letter 'f' and the figure '5' wherever a 'global replace' will be required, and then to perform a global replace, thus:

```
Global replace: f5/<f5><RETURN>
```

where <f5> indicates pressing the f5 key, which will appear as a space on the display.

12 Using MASM to assemble your programs

Chapter 3 showed you how to use MASM to assemble a simple program. This chapter describes the facilities of MASM in more detail.

To start using MASM, you should type the line:

```
*MASM
```

You can do this from the normal system prompt or from the command mode of all the 6502 Development Package utilities. After MASM has been loaded, it will enter MODE 7 and print its prompt 'Action: '. Any of the commands described below can be entered at this point. If you press ESCAPE, MASM will stop what it is doing as soon as possible and return to command level. Pressing BREAK will have the same effect, but MASM will not be able to take precautions against corrupting your files, so only press BREAK in dire emergencies, and NEVER press it when a disc drive light is on.

If you wish, you can shorten MASM commands to the shortest distinct substring, for example, instead of the following:

```
ASM FRED
```

you could type:

```
A FRED
```

Beware, however, of commands that have common beginnings, for example, SAVE, STOP, SYMBOL. Here, the command S will choose the first command beginning with S that HELP prints.

12.1 MASM commands

This section describes each of the MASM commands in detail.

MASM commands can be typed in upper or lower-case; they mean the same thing either way. At any time when MASM is waiting for a command, you can give an operating system command line by prefixing it with '*'. For example, the command line:

```
*CAT
```

would produce a catalogue of the currently selected drive.

You can enter the first parameter required by any command on the same line as the command itself. If it is not given, you will be prompted by MASM. Thus to turn on an assembly listing, you could use either one of the following lines:

```
Action : PRINT ON
```

or

```
Action : PRINT
Option : ON
```

File names which you use in commands should obey the rules laid down in the User Guide for the filing system currently in use. In addition they must be valid MASM identifiers (whose maximum length is six characters) since if a longer name is used, MASM only uses the first six characters and will probably, therefore, give a 'not found' error.

12.1.1 The ASM command (Assemble program)

This is the most important command available in the assembler. Thse dialogue is as follows:

```
Action : ASM <file name>
```

or:

```
Action : ASM

Source file : <file name>
```

and you will then be prompted as follows:

```
Macro library :
```

You can now type the name of a macro library file or, if you have none, you can press RETURN. If you have typed ASM without a file name attached, you will now be prompted for the file name. MASM will then look for the source file on the disc and, if it finds it, try to assemble it. If you specify the name of a non-existent file, this will result in an error message. MASM will always return to the command level after such an error.

The file specified in the ASM command should be of the format described in previous chapters of this book; any discrepencies will be pointed out. MASM will give an error message which specifies the type of error and the line number in the file (and macro if appropriate) where the error was detected. The offending line will also be printed.

After detecting an error in the first pass, MASM will, in most cases, continue trying to assemble the file so that all errors will be detected (see the STOP command below, though). Be warned, however, that one error may generate several more as a 'side effect', so correcting this one may improve matters considerably. Common errors are:

- Forgetting the END directive
- Forgetting to define a label
- Misspelling a label

The last two of these are the sort which can generate several error messages from a single mistake in your source file.

MASM will print which pass it is on at the beginning of the pass. If the first pass was completed without error, the second pass will be started. During the second pass, the object file (or files if other source files are 'LNKed' in) is produced; this contains machine code instructions which correspond to the program being assembled. The object file will have the same name as the source file, but it will be held in the directory 'X'. Thus, if you type:

```
ASM test
```

MASM will place the object code in a file called 'X.test'. The load and execution address of this file will be as was specified by the ORG statement in the source file (or &0000 if no such instruction was given). Addresses in the program will be set with &0000 as the high-order bytes; this means that your object file will load into the 6502 Second Processor unless you force it to do otherwise when you SAVE it (see the SAVE command later).

Your object file can be run using the OS command:

```
*X.<filename>
```

Since all of your source code must be loaded into the Tube at once, there is a limit on the size of this code: it is currently about 17K. However, the LNK directive can be used to increase the total size of your source program.

12.1.2 The PRINT command (turn listing on or off)

It is helpful, when debugging a program, to have the assembler produce a listing of the code it has assembled, along with the addresses and values it has generated; Figure 12.1 shows such a listing. If you use the command

```
PRINT ON
```

all subsequent successful assemblies will produce a listing. You can disable this facility by using the command:

```
PRINT OFF
```

This is the default state upon entering MASM.

Listing can also be controlled from within the source file using the TTL and OPT directives; these were explained in chapter 5, 'The MASM directives'.

To send the listing to your printer as well as to the screen, you need to enable the device using the appropriate control codes. Typing CTRL B at any time during a command input will turn on the printer (if it has been selected properly with *FX5), and CTRL C will turn the printer off. Use of the printer is explained fully in the *BBC Microcomputer System User Guide*. For listings on the screen, you will find it useful to engage page mode (CTRL N). This can be disengaged by typing CTRL O.

```
Acorn macro assembler     Page 1

Pass 1
Pass 2
0001 0000 0070      zerop   *       &70
0002 0000                   ORG     &1900
0003 1900 A9 00     test    LDAIM   0
0004 1902 85 70             STAZ    zerop
0005 1904 A9 1A             LDAIM   fin / &100 + 1
0006 1906 85 71             STAZ    zerop+1
0007 1908 A0 00             LDYIM   0
0008 190A B1 70     loop    LDAIY   zerop
0009 190C 48                PHA
0010 190D A9 AA             LDAIM   &AA
0011 190F 91 70             STAIY   zerop
0012 1911 D1 70             CMPIY   zerop
0013 1913 D0 22             BNE     error
0014 1915 A9 55             LDAIM   &55
0015 1917 91 70             STAIY   zerop
0016 1919 D1 70             CMPIY   zerop
0017 191B D0 1A             BNE     error
0018 191D 68                PLA
0019 191E 91 70             STAIY   zerop
0020 1920 C8                INY
0021 1921 D0 E7             BNE     loop
0022 1923 E6 71             INCZ    zerop+1
0023 1925 A5 71             LDAZ    zerop+1
0024 1927 C9 80             CMPIM   &80
0025 1929 D0 DF             BNE     loop
0026 192B 00                BRK
0027 192C 01                =       1
0028 192D 4D 45 4D          =       "MEMORY OK"
0029 1936 00                =       0
0030 1937 00        error   BRK
0031 1938 02                =       2
0032 1939 4D 45 4D          =       "MEMORY FAULT"
0033 1945 00                =       0
0034 1946           fin
0035 1946                   END
Assembly finished, no errors
```

Figure 12.1 Sample assembler listing

12.1.3 The WIDTH command (set printer width)

You can specify the width of your output using this command; this can be between zero and 127 characters. MASM will try to format the output from the assembler passes (and the SYMBOL command) as neatly as possible within this range. For viewing on the screen, WIDTH 39 gives the best results, since each line of assembled code fits on to one screen line. For printers, WIDTH 79 or WIDTH 80 is probably the best choice. Note that if output is attempted of a line containing a greater number of characters than that specified by WIDTH the line is truncated to fit that width, and not wrapped round to the next line.

12.1.4 The LENGTH command (set the printer length)

This command lets you specify the height of pages on your printer. After each page, MASM gives a form feed (this turns into a clear screen on your display). The parameter can be in the range zero to 127, but most printers have page lengths of between 60 and 70 lines (for screen viewing, 127 is probably the best choice).

If you wish, you can force a form feed using the appropriate OPT directive in your source file.

12.1.5 The SYMBOL command (print a symbol table)

After an assembly, successful or otherwise, you can get a listing of all the symbols MASM came across in the file. This is very useful when debugging a program, and you have three alternatives:

```
SYMBOL A
SYMBOL N
SYMBOL S
```

The first two give listings in alphabetic or numeric order. The last one prompts you for a symbol name and then prints its value.

Each symbol is listed, together with the value associated with it (in hex) if this has been set. Symbols declared but unused are marked with a *. Symbols which are undefined are given the value XXXX. The format of a table produced by SYMBOL is affected by the current WIDTH and LENGTH settings.

12.1.6 The STOP command (stop on errors)

If the command:

```
STOP ON
```

is issued before assembling a file, any error will stop the assembly process immediately. MASM will then wait for you to press ESCAPE, after which control will be returned to command level.

If you use the command:

```
STOP OFF
```

instead, the whole file will be processed for the first pass regardless of the number of errors produced. STOP OFF is the default value of the command when you load MASM.

12.1.7 The SAVE command (save the object file)

The SAVE and GET commands allow you to save files to disc and load them. They are similar to the *SAVE and *LOAD OS commands, but give you more freedom in the specification of load and execution addresses. If you type:

```
SAVE <file name>
```

the following prompts will be given:

Prompt	Meaning
Start address	This is where the file to be saved starts in the Second Processor
End address	This is where the file to be saved ends in the Second Processor
Load address	This is where you want the file to be reloaded in memory
Proc.	This is the processor in which you want the file to be reloaded
	I for I/O (BBC), T for Tube H host P parasite
	Note that I and H are identical in their effect, as are T and P
Exec. address	This is what the execution address should to be set to
Proc.	This is the processor in which you want the file to be executed

The two 'Proc.' prompts are needed to set the high- order bytes of the load and execution addresses for the file. For example, suppose you want to save a &400 byte section of the Tube processor's memory, starting from &5600. The file is to be saved with the name 'BODGE' so that it loads at &1600 in the BBC Microcomputer's memory and starts to execute at &1645 when it is run. The following dialogue would achieve this:

```
Action : SAVE BODGE
Start address : &5600
End address : &5A00
Load address : &1600
Proc. (T/P/H/I) I
Exec. address : &1645
Proc. (T/P/H/I) I
```

You could check that the load and execution addresses, and the length, have been set as required by using the disc filing system command *INFO:

```
*INFO BODGE
```

Notice that the addresses you use can be any valid MASM expressions, rather than just simple numbers. For example, look at the following:

```
Load address : start
Exec. address : start+tablen
```

Here, 'start' and 'tablen' are symbols in the current symbol table.

12.1.8 The GET command (load a file or files)

This command is complementary to SAVE. After entering:

```
GET <file name>
```

the following prompt will be displayed:

```
New, Own or Previous address (N/O/P)
```

The three possible responses to this question have the following meanings:

Response	Meaning
N	You want to specify the address at which file will load (in the Tube)
O	The file is to be loaded into the address given in its directory entry, but in the Tube, even if the load address is in the I/O processor
P	The file is to be loaded after the last file loaded

In addition, if the file name ends in a two-digit number (for example, MOS02), you will be asked for an 'offset'. This specifies the last file you want to load in a sequence. For example:

```
Action: GET MOS02
Offset: 05
New, Own or Previous address (N/O/P) N
Address: &1200
```

This will result in the files MOS02, MOS03, MOS04 and MOS05 being loaded successively. The first file (MOS02) will be loaded at &1200 and the other three will be appended to it automatically, whatever the address specified for the first file.

12.1.9 The XREF command (make a cross-reference file)

The XREF command is associated with the cross-reference utility. This is a debugging tool and it is described in detail in chapter 14, 'Debugging your programs'. Basically, the cross-reference utility finds all of the occurences of certain symbols in your source program, thus saving you looking through the entire assembly listing. To be able to do this, it needs a file which contains all the relevant information. MASM can be instructed to produce this file by using the XREF command. This must be issued before assembling the file, so a typical command sequence would be:

```
Action: XREF
Xref output file: xrout
Action: ASM source
```

These instructions would assemble the file as normal and also a produce a file called 'xrout'. The latter must be cited when the cross reference utility is used.

12.1.10 The MLEVEL command (suppress macro level information)

To help keep track of local labels, MASM stores information about the points at which it enters and leaves macros in a local label table. If you have a lot of macros in your program, this table can become full even when you have no local labels. You can stop it filling by using the MLEVEL command.

The command has the default value of ON and, if you give it the alternative value OFF, macro level information will be omitted from the local label table. You give it the appropriate value by typing one of the following:

```
MLEVEL ON
MLEVEL OFF
```

Note that after MLEVEL OFF is issued the macro level specifiers in local label usages are ignored.

12.1.11 The TERSE command (print conditional assembly source code)

The TERSE command allows you to print the source code that is ignored by conditional assembly. To do this, type:

```
TERSE OFF
```

The default value is ON; this suppresses the source code.

13 Producing program listings

The 6502 Development Package contains a general-purpose printing program: the PRINT utility. This allows you to print source files on the screen or printer, with variable-size pages, line numbers and assembler formatting, amongst other things.

To load the utility you type:

```
*PRINT
```

You will then be prompted as follows:

```
File name:
```

At this point, you can type one of the OS commands (for example, *CAT) or the name of the file to be printed. If you select the latter option, the following prompt will then be displayed:

```
Parameters:
```

Here, you can again type one of the OS commands, or, instead, you can type a list of PRINT parameters separated by spaces. There are nine parameters which you can use; they are set by typing the initial letter of the parameter followed by the value to be assigned to that parameter (where relevant).

The initial letters of the parameters are as follows:

Parameter	Meaning
W	This defines the page width and is set to a default value of 92
L	This defines the page length and is set to a default value of 60
T	This defines the length of the page heading (title) and is set to a default value of five
H	This defines a page heading. It can be any string of up to 46 characters, enclosed in quotes (""). Its default value is the filename
N	This indicates that you want the line number printed on each line
A	This indicates that you want the output formatting into fields as understood by the assembler
P	This indicates that you want the printer to stop after printing each page. Printing will be resumed when you press the SHIFT key
E	This indicates that you want to output text to the Econet printer
R	This runs the print program

You can set these parameters in any order and you will be prompted until the 'R' parameter is set. Examples of the dialogue are given below:

```
*PRINT
File name: TEST1
Parameters: L70
Parameters: W79
Parameters: R

File name: TEST2
Parameters: L70 W79 N R
```

If you wish, you can avoid the 'Parameter' prompt by setting the parameters in the 'Filename' prompt. In this case, you do not need to use the 'R' parameter: it will be set automatically. For example:

```
File name: TEST3 W79 L70 N
```

There is a version of PRINT called PR which is identical except that it is located in the I/O processor rather than the 6502 Second Processor, and so may be used to print source files when a second processor is not available.

14 Debugging your programs

This chapter is divided into four sections: the first one is a brief introduction to the principles of debugging and how the MASM utilities can help with this. The other three sections describe the debugging utilities XREF, SRCXREF, and TTRACE and BTRACE. If you are an experienced programmer and understand debugging you can probably omit reading the first section.

14.1 Introduction

Even if you are the world's best programmer your programs will still have bugs in them at some time. When that time comes you will need all of the tools at your disposal.

The most useful tool of the lot is the human brain! Every program you write should be dry run before you put it on to your computer. This involves thinking through every step in the program and it can save many a wasted hour on the computer.

Later on, most of your bugs will be removed in this way. However, in the beginning you may not fully understand the nuances of assembly code, and you might expect different results from an instruction than those which are possible. At this time, some further debugging tools will help a lot.

Look at the following program, for example:

```
zerop   *       &70
        ORG     &1900
test    LDAIM   0
        STAZ    zerop
        LDAIM   fin / &100 + 1
        STAZ    zerop+1
        LDYIM   0
loop    LDAIY   zerop
        PHA
        LDAIM   &AA
        STAIY   zerop
        CMPAY   zerop
        BNE     error
        LDAIM   &55
        STAIY   zerop
```

```
        CMPAY zerop
        BNE   error
        PLA
        STAIY zerop
        INY
        BNE   loop
        INCZ  zerop+1
        LDAZ  zerop+1
        CMPIM &80
        BNE   loop
        BRK
        =     0
        =     "MEMORY OK"
        =     0
error   BRK
        =     0
        =     "MEMORY FAULT"
        =     0
fin
        END
```

This is similar to the program you developed in chapter 3, 'Developing a simple MASM program', but a couple of 'bugs' have been introduced into it. If you ran the program, you would see the following message displayed:

```
MEMORY FAULT
```

This might imply that there is something wrong with your 6502 Second Processor, but as it is the first time you've run the program that should make you suspicious.

The place in the program at which the message is generated is easy to find: it is at the label 'error'. It is also easy to find which parts of the program cause a branch to 'error', but if your program was a large one there may be many such branches. When this is the case, you can save yourself a lot of reading by using the cross-reference utility (XREF). This enables you to find the occurence of various symbols in an assembly language program. It will tell you the values (if any) of the symbols and where they are defined.

Before you can use XREF you must produce a cross-reference file which it can use; this is done using the XREF command in MASM (see chapter 12, 'Using MASM to assemble your programs'). The following dialogue would produce a cross-reference file called 'xrout' for you and also assemble your program (which is assumed to be in the source file 'TEST'):

```
Action : XREF
Xref output file : xrout
PRINT ON
```

(type CTRL B to enable the printer before pressing RETURN)

```
Action : ASM TEST

Macro library :<RETURN>
```

Turn the printer off with CTRL C.

Next, you can activate XREF by typing:

```
*XREF
```

and it will reply with the prompt:

```
Action :
```

XREF holds a list of symbols which you want it to search for and you need to ADD 'error' to this list. You do it by typing 'ADD error', then, when the prompt:

```
Symbol :
```

appears, you should press ESCAPE. You can now get XREF to scan through the cross-reference file by typing 'XREF' in reply to the 'Action :' prompt. It will then prompt you as follows:

```
Xref File :
```

and you should type 'XROUT'. XREF will scan the file and print the message:

```
ERROR defined  line 0030 in file TEST
ERROR used     line 0013 in file TEST
ERROR used     line 0017 in file TEST
```

It will then return to the action prompt.

There is another, more limited version of the cross-reference utility which operates directly upon your source program file without needing MASM. This is called the source file cross-referencing utility (SRCXREF) and it is described in section 14.3.

Once you have found all the branches to 'error' in your program, you will need to find out which one caused the 'fatal' result. You can do this using the TRACE utility. First of all, load your program by typing:

```
*LOAD X.TEST
```

Now load the trace utility by typing:

```
*TTRACE
```

Note that this is the utility for tracing code in the 6502 Second Processor; if the code is located in the I/O processor then the BTRACE utility should be used. This is identical to TTRACE except that it may only be used in mode 7. Throughout this description TRACE is used to refer to whichever version is being used.

The TRACE utility will reply with the prompt '+'. TRACE will execute your program and give you reporting information as it does this. You can 'set' different types of reporting level; for example, it will report the state of the Y register or the execution address after each instruction. For the first run through, we will ask it to report the execution address; this should enable us to find out where the branch to 'error' occurred. You should type:

```
RT ADDR
```

to do this. Now, you can get TRACE to run your program by typing:

```
EN &1900
CO
```

'EN' sets the entry point of the program and 'CO' continues execution from that point. TRACE will print out the following list of execution addresses as it runs your program:

```
1900
1902
1904
1906
1908
190A
190C
190D
190F
1911
1914
1939

MEMORY FAULT
```

If you look at your assembly listing, you will see that '1939', the last execution address, is the address of 'error'. The address before it, '1914', must be the address of the instruction which caused the fatal branch to 'error'.

Now that we have isolated the code which is causing the trouble, it would be useful if we could stop the program executing, before it branches to 'error', and look at what is happening more closely. We can do this by inserting what is called a 'breakpoint' in the program at location 1914 (the branch to 'error'). The program will run until it reaches this point then stop, allowing you to inspect the registers and memory locations, or even to change their contents. You set the breakpoint by typing:

```
BS &1914
```

Now set the reporting level to report everything, and run the program by typing:

```
RT ALL
EN &1900
CO
```

You will get a display in the following format, showing you the results of every instruction executed and the status at the breakpoint (it is best to move to an 80 column mode when using TTRACE if possible):

```
1900 A9 00     LDAIM &00   A=01 X=54 Y=07 P=..1B.... S=E2
1902 85 70     STAZ  &70   A=00 X=54 Y=07 P=..1B..Z. S=E2
1904 A9 1A     LDAIM &1A   A=00 X=54 Y=07 P=..1B..Z. S=E2
1906 85 71     STAZ  &71   A=1A X=54 Y=07 P=..1B.... S=E2
1908 A0 00     LDYIM &00   A=1A X=54 Y=07 P=..1B.... S=E2
190A B1 70     LDAIY &70   A=1A X=54 Y=00 P=..1B..Z. S=E2
190C 48        PHA         A=AA X=54 Y=00 P=N.1B.... S=E2
190D A9 AA     LDAIM &AA   A=AA X=54 Y=00 P=N.1B.... S=E1
190F 91 70     STAIY &70   A=AA X=54 Y=00 P=N.1B.... S=E1
1911 D9 70 00  CMPAY &0070 A=AA X=54 Y=00 P=N.1B.... S=E1
Stopped at break point
1914 D0 23     BNE   &23   A=AA X=54 Y=00 P=N.1B...C S=E1
```

Note that the contents of some of the registers may not be exactly as shown above, since they are dependent on the machine configuration.

The CMPAY instruction compares the contents of the accumulator (AA on the last line of the display) with the contents of (&70 + Y) and branches to the error routine if they are not equal. You can check the contents of address &0070 by typing the following:

```
PT &70 &71
```

and you will get the result:

```
0070 00 1A FF 00 00 00 00 00
```

You would probably realise that the instruction should have been a CMPIY instruction and you could confirm it by treating the contents of addresses &0070 and &0071 as a 16-bit address and looking at their contents, in turn.

These memory locations point to the address &1A00, and you can inspect its contents by typing:

```
PT &1A00 &1A01
```

You will get the result:

```
1A00 AA 00 00 00 00 00 00 00
```

The contents of &1A00 are the same as those of the accumulator and it is clearly this address which the compare instruction should have referenced. You could prove this by 'patching' the CMPAY instruction with a CMPIY and rerunning the program. To do this, you type:

```
PS &1911 (RETURN)
D1+
+
EA (RETURN)
```

TRACE will print out the contents of the location before you change it. You type '+' to move on to the next location; if you wanted to change the previous location, you could display it by typing '-'. In the text above, 'D1' is the CMPIY opcode; the following address (&70) does not need to be changed. However, since CMPIY needs one byte less than CMPAY, you have to use a 'no-operation' instruction (opcode 'EA') as well.

You need to change the CMPAY at address &191A, as well, before you can run your program successfully. You do this by typing:

```
PS &191A (RETURN)
D1 +
+
EA (RETURN)
```

Now get rid of the breakpoint by typing:

```
BC &1914
```

(this is the 'breakpoint clear' instruction). You can get rid of the information reporting facility and run your program by typing:

```
RT NONE
EN &1900
CO
```

Your program will take much longer to execute than it would normally; this is because TRACE slows it down by a factor of about 100.

The preceding dialogue has been very simplistic; in reality, you would have spotted the errors much sooner. However, it serves to illustrate the most common debugging facilities. These are described in much more detail in the following sections.

14.2 Using the cross-referencer (XREF)

To run the cross referencer, simply type *XREF when the Development Package disc is in the currently selected drive. This will take you into the command mode. XREF can only be used if there is a 6502 Second Processor fitted to the BBC Microcomputer.

The prompt for XREF is:

```
Action :
```

You can get a list of XREF's commands by typing 'HELP' in reply to the action prompt. You will get the following display:

```
Commands available:

ADD
CLEAR
HELP
INIT
LIST
RESULT
SUMMARY
XREF
```

A command can be activated by typing the appropriate command name in upper or lower case. The command can be abandoned at any time by pressing ESCAPE; this will return you to the menu followed by the prompt. You can enter OS commands after the 'Action:' prompt by prefixing them with a *, as in:

```
Action:*CAT
```

The purpose of XREF is to help you find the occurence of various symbols in an assembly language program. It will tell you the symbols' values (if any) and where they are defined and used throughout the program. In order to do this, XREF needs a cross-reference file which it can use; this is done using the XREF command in MASM (see chapter 12, 'Using MASM to assemble your programs', for details).

Below is a description of each of XREF's commands. The first three relate to the entry of symbols which are to be referenced, the last three are to do with the actual cross referencing.

14.2.1 The ADD command (add symbols)

The first thing XREF needs is a list of symbols to look up for you. This is entered using the command ADD. Once the command has been issued, XREF will repeatedly prompt you as follows:

```
Symbol:
```

You should reply with a valid symbol name which you expect to find in your program. XREF will check the symbols in the file using the same rules as MASM, so replying '123', for example, will produce the following message:

```
Bad symbol
```

Once all the symbols have been added, you should reply with a single point '.' to get back into command mode.

You can also enter the symbols from a file, using the *EXEC command. For example, if you envisage using a set of, say, 10 symbols which are common to many of your programs, the following file could be created with the *BUILD command:

```
Command : *BUILD comms
0001 A
0002 symb1
0003 symb2
..........
..........
0011 symb10
0012 .
0013 Escape
```

The first line is the A command (for ADD), then come the ten symbols to be entered, then the '.' to terminate command entry. Once created, the file can be used as follows:

```
Command : *EXEC comms
```

This will cause the file 'comms' to be treated as the input source until its end is reached. See the *Disc Filing System User Guide* for more details of *BUILD and *EXEC.

The maximum number of labels which XREF can handle at once is 1024. This should be sufficient for most purposes.

14.2.2 The CLEAR command (clear symbol table)

CLEAR will remove a single named symbol from the symbol table. You can type CLEAR followed by the symbol name in reply to the action prompt. Or you can issue the command CLEAR, followed by RETURN, and XREF will prompt you as follows:

```
Symbol:
```

You should reply with a valid symbol name which you want deleting from the symbol table. XREF will return you to command mode when it has successfully deleted your symbol.

14.2.3 The INIT command (initialise symbol table)

This command will restart the program by clearing out the symbol table.

14.2.4 The LIST command (list symbol table)

You can check the current contents of the symbol table by entering the command LIST. The symbol names will be printed in a simple list across the page, for example,

```
ERROR FIN LOOP TEST ZEROP
```

14.2.5 The XREF command (cross-reference a file)

Once the symbols are correct, you need to tell XREF which file you want cross-referenced. You can type XREF followed by the filename. Typing XREF will produce the prompt:

```
Xref file :
```

to which you should reply with the name of a file that was produced by MASM when its XREF command was used. XREF will read this file and look up the names you entered into the symbol table. It will then print a summary and return to the prompt stage. If the name given does not refer to a valid cross-reference file then a 'Read error' will be generated.

14.2.6 The RESULT command (print results)

This should be the command you issue after cross-referencing a file with command XREF. The program will print a list of the locations (line numbers) and the file at which the various symbols in the symbol table appear. If there are many symbols in the table, and/or the file which produced the cross-reference file was very large, there may be many entries printed out. In this case, it is wise to use a printer to get a hard-copy of the reults, or alternatively send them to a *SPOOL file where you can examine them at your leisure. Remember that all the OS * commands can be issued after the 'Command :' prompt, and the printer can be enabled and disabled using CTRL B and CTRL C respectively.

After the table of symbols entries has been printed, a summary will appear. This summary mentions two types of symbols: it gives a warning for any symbols in the table which were not found at all in the cross-reference file; and it gives a comment for any symbols which were defined in the source but not used in the program.

14.2.7 The SUMMARY command (set the summary flag)

The printing of a summary can be disabled using this command. If you type SUMMARY, the following prompt will appear:

```
Summary ? (Set/Unset) :
```

You should reply 'S' if you want a summary to be printed or 'U' if you don't. Control then returns to the command level.

14.3 Using the free-standing cross-referencer (SRCXREF)

The free-standing cross-reference utility is a more-limited version of XREF which does not need a cross-reference file produced by MASM. To run it, simply type *SRCXREF when the 6502 Development Package disc is in the currently selected drive. This will take you into the command mode. SRCXREF can only be used if there is a 6502 Second Processor fitted to the BBC Microcomputer.

The prompt for SRCXREF is:

```
Action:
```

You can get a list of SRCXREF's commands by typing:

```
HELP
```

and it should be as shown below:

```
Commands available:

ADD
CLEAR
HELP
INIT
LIST
RESULT
XREF
```

A command can be activated by typing the appropriate command name (for example 'ADD') in upper or lower case. It can be abandoned at any time by pressing ESCAPE; this will return you to command mode. You can enter OS commands after the 'Action:' prompt by prefixing them with a *, as in:

```
Action: *CAT
```

Below is a description of each of SRCXREF's commands.

14.3.1 The ADD command (add symbols)

The first thing SRCXREF needs is a list of symbols to look up for you. This is entered using the ADD command. Once the command has been issued, SRCXREF will repeatedly prompt you as follows:

```
Symbol:
```

You should reply with a valid symbol name which you expect to find in your program. SRCXREF will check the symbols in the file using the same rules as MASM, so replying '123', for example, will produce the following message:

```
Bad symbol
```

Once all the symbols have been added, you should reply with a single point '.' to get back into command mode.

14.3.2 The CLEAR command (remove symbol)

CLEAR is the inverse operation to ADD. It will remove a single named symbol from the symbol table. Once the command has been issued, SRCXREF will prompt you as follows:

```
Symbol:
```

You should reply with a valid symbol name which you want deleting from the symbol table. When you have done this, SRCXREF will return to command mode.

14.3.3 The INIT command (initialise symbol table)

This command will restart the program by clearing out the symbol table entirely.

14.3.4 The LIST command (list items in symbol table)

You can check the current contents of the symbol table by entering the LIST command. The symbols will be printed in a simple list across the page, together with a count, for example:

```
ERROR LOOP ZEROP
3 symbols in table
```

14.3.5 The XREF command (cross-reference a file)

Once the symbols are correct, you need to tell SRCXREF which source file you want cross-referenced. Typing 'XREF' will produce the prompt:

```
Xref file :
```

to which you should reply with the name of a file at which the cross-referencer is to start. The referencing will continue along a series of files which have been 'joined' using the LNK directive and will finish when END is encountered or a LNK file is not found. Note that MASM directives to change drives are ignored, so you must do this using OS commands.

SRCXREF will read the file(s) and look up the names you entered in the symbol table. It will then print a summary and return to command mode.

14.3.6 The RESULT command (print results)

This should be the command you issue after using the XREF command. The program will print the symbol name and a definition of its usage, for example:

```
ERROR
ERROR   used at line 15 in EXAMPL
LOOP
LOOP    used at line 23 in EXAMPL
ZEROP
ZEROP   used at line 6 in EXAMPL

3 symbols in table
```

If there are many symbols in the table, there may be many entries printed out. In this case, it is wise to use a printer to get a hard-copy of the results, or, alternatively, to send them to a *SPOOL file where you can examine them at your leisure. The printer can be enabled and disabled using CTRL B and CTRL C respectively.

14.4 Using the TRACE utilities (TTRACE and BTRACE)

The TRACE utility comes in two forms and the appropriate form should be substituted wherever the term TRACE is used throughout this book. BTRACE will run on the BBC Microcomputer and works on either side of the Tube, however for use in the 6502 Second Processor the locations used by TTRACE (&E600-&F7FF) are often more convenient.

Except in special circumstances, every instruction executed by the program is traced. In most cases this is done by placing them in a 'scratch pad' and allowing them to run; in other cases (for example, JSR and BRK) it is done by simulation.

Information which you request is reported before the execution of each instruction; the amount of this information is determined by the current reporting level (defined in some commands). Errors are reported at the global reporting level (defined by the 'RT' command) and this may differ from the current reporting level.

You can, if you wish, define an interrupt key. This can be used to break into tracing or lengthy print operations, for example.

BTRACE occupies the memory from &6A00 to &7C00. It also uses &6800 to &69FF, as variables area, and locations &7E and &7F in page zero.

TTRACE occupies the memory from &E600 to &F7FF. It also uses &E400 to &E5FF, as variables area, and locations &E0 and &E1 in page zero.

All numbers which are input to TRACE must be in hexadecimal and must be preceded by '&'. If a one-byte value is expected by TRACE and you give it a two-byte value, the low byte will be used.

14.4.1 Use of breakpoints and reporting

This group of commands includes the following:

BS Set breakpoint
BC Clear breakpoint
DB Display breakpoints
RT Set global reporting level
RH Set reporting high memory point
RL Set reporting low memory point
ST Stack trust
TC Clear trust address
TS Set trust address
DT Display trusts
IK Set interrupt key
BH Set break high memory point
EN Set entry point for trace
CO Continue tracing
SS Snapshot

Each of the commands is described below.

Set breakpoint (BS)

This command has the following format:

```
BS <address> <optional count>
```

and it will set a breakpoint at <address>. If you do not specify the count, a count of zero will be assumed.

When <address> is reached, TRACE will break in if the count is zero; otherwise it will decrement the count and continue.

The following example will set a breakpoint at address &1914 with a count of zero:

```
BS &1914
```

Clear breakpoint (BC)

This has the format:

```
BC <address>
```

and it removes the breakpoint at <address>.

Display breakpoints (DB)

This gives a list of the currently-set breakpoints, with their counts, in descending order of address, for example:

```
191D 0000
1914 0020
```

To use this command, you merely type 'DB'.

Set global reporting level (RT)

This command has the format:

```
RT <reporting options>
```

and it defines the 'level' of tracing information output. The following reporting options are available:

Option	Meaning
NONE	Report nothing
AB	Report all but the following options (defined)
ALL	Report everything described below
ADDR	Report the execution address
HEX	Report the opcode in hexadecimal form
OP	Report the opcode in mnemonic form
A	Report contents of A register
X	Report contents of X register
Y	Report contents of Y register
P	Report contents of flag register
S	Report contents of stack pointer

If you do not use the RT command, a default value of 'ALL' will be used. The output from this looks like the following:

```
          ............
          ............
1900 A9 00 LDAIM &00   A=01 X=54 Y=07 P=..1B.... S=F8
1902 85 70 STAZ  &70   A=00 X=54 Y=07 P=..1B..Z. S=F8
          ............
          ............
```

Set reporting high memory point (RH)

This command allows you to define a page address in memory above which reporting will not take place. It has the format:

```
RH <byte value>
```

Reporting will be suppressed if the program counter high byte equals or exceeds <byte value>. The default value is &80, corresponding to an address of 32768 decimal.

Set reporting low memory point (RL)

This command allows you to define a page address in memory below which reporting will not take place. It has the format:

```
RL <byte value>
```

Reporting will be suppressed if the program counter high byte is less than <byte value>. The default value is &00.

Stack trust (ST)

This command allows you to suppress reporting in subroutines if the stack 'level' goes below a given value. Its format is:

```
ST <byte value>
```

It is used in conjunction with the TS command (described below): TS allows you to to suppress reporting in a given subroutine; ST allows you to suppress it in subroutines below a given 'level'.

Reporting will resume when the stack level returns to the given level or above.

Set trust (TS)

This command adds a trust address to the 'trust table'. It allows you to suppress the trace output in a particular subroutine. The format of the command is as follows:

```
TS <address>
```

If, on tracing a JSR instruction, <address> is found to be in the trust table, reporting will be temporarily turned off until the stack level rises to the current level or above.

Clear trust address (TC)

This removes a trust address from the trust table. Its format is as follows:

```
TC <address>
```

Display trust addresses (DT)

You can display the current trust addresses by typing 'DT'. They will be given in descending order of address and will look something like the following:

```
1952
2100
3002
4043
```

Set interrupt key (IK)

If you wish, you can define an interrupt key and this could be used to break into over-long tracing operations, for example. The format of the command is as follows:

```
IK <byte value>
```

<byte value> corresponds to the value of the key as defined in the section on INKEY in the *BBC Microcomputer System User Guide* and must be given in the hexadecimal form. For example,

```
IK &B8
```

would define the '@' key as an interrupt key.

If you are interpreting or printing store, the interrupt key will be checked before the start of each line of output. If you are tracing, it will be checked at most once every 256 instructions. When the key is found to be depressed, a return will be made to command level (the '+' prompt).

Set break high-memory point (BH)

If the program counter is above the break high-memory point, the interrupt key will not be checked. You can specify this address using the BH command. Its format is as follows:

```
BH <byte value>
```

where <byte value> is the high byte of the address.

The following example would set the break high-memory point to &F000:

```
BH &F0
```

Set entry point (EN)

This command is used to define the starting point for tracing. Its format is as follows:

```
EN <address>
```

EN does not cause tracing to start; this operation is performed by the CO command (see below).

Continue tracing (CO)

The 'CO' command continues tracing from the current address; it is also used to start tracing, after you have defined the start point using the 'EN' command. The format of the command is as follows:

```
CO <optional reporting level>
```

If a current address has not been defined, the command will fail.

The reporting level, if present, will be made the current reporting level; otherwise, the global reporting level will be used.

Snapshot (SS)

This instruction displays the full status at the start of the current instruction. It does execute the instruction.

14.4.2 Looking at memory

This group of commands includes the following:

IT Interpret (disassemble) store
PT Print store

Interpret store (IT)

This command allows you to interpret (disassemble) store from the given address; it displays the addresses and opcodes in the following way:

```
190F 91 70 STAIY &70
1911 D1 70 CMPIY &70
1913 EA    NOP
1914 D0 23 BNE   &23
```

Standard MASM mnemonics are used.

The format of the command is as follows:

```
IT <address> <optional address>
```

Store will be disassembled from <address> and the process will stop at <optional address>. If the latter is not present, disassembly will stop at &FFFF. To produce the above example, the following command would have been used:

```
IT &190F &1914
```

Print store (PT)

This command allows you to print store from the given address; it displays the contents in the following way:

```
0070 00 1A 00 00 00 00 36 00
0078 00 00 36 00 00 00 36 00
0080 02 FF 52 DF FF FF FF FF
```

The format of the command is as follows:

```
PT <address> <optional address>
```

Store will be printed from <address> and the process will stop at <optional address>. If the latter is not present, printing will stop at &FFFF. To produce the above example, the following command would have been used:

```
PT &0070 &0087
```

14.4.3 Patching memory and registers

This group of commands includes the following:

PS Patch store locations
PR Patch register

Each of these commands is described below.

Patch store (PS)

This command allows you to modify data and instructions in the computer's memory. Its format is as follows:

```
PS <address>
```

After you type the command, the contents of <address> will be displayed and you can then modify them. The command is interactive, in that you can modify a number of locations in succession with the minimum amount of effort. If you type '+' after an address, or in place of an address, the contents of the next address will be displayed and you can modify them also. Similary, if you type '-', the contents of the previous location will be displayed and you can modify these. When you have finished modifying store, you can return to command level by pressing either RETURN or ESCAPE.

As an example, the following dialogue will change locations &1911 and &1913 to &D1 and &EA, respectively:

```
PS &1911
1911 D9 D1+
1912 70 +
1913 00 EA <RETURN>
```

Here, the underlined items would be typed in by you.

Patch register (PR)

This command allows you to modify ('patch') the contents of the A, X, Y, P or S register; only one of these can be modified with a single command. Its format is as follows:

```
PR <regname> <byte value>
```

For example, the following commands will alter the X and A registers to &70 and &FF:

```
PR X &70
PR A &FF
```

14.4.4 Memory protection

This group of commands includes the following:

SP Store protect
DP Display store protections
SA Store allow (unprotect)

Each of these commands is described below.

Store protect (SP)

If a memory location in your program is being corrupted and you do not know how this is happening, you can find out by using this command. It has the following format:

```
SP <address> <byte value>
```

Suppose, for example, that the location &1900 should contain the value '&55', but it is being changed somewhere in your program. You would type the following:

```
SP &1900 &55
```

Each time a store-modifying instruction is obeyed, TRACE would check if it is altering address &1900. If it is, a message similar to the following will be displayed and you can inspect the code that is causing the problem:

Protection failure at 1900 was 88 should be 55

Display protections (DP)

This command gives you a list of protected locations, in descending order of address. All you need do is type in 'DP' and you will get a display which looks something like the following:

```
191D DO
1918 91
190C 48
```

Store allow (SA)

This command removes a given location from the table of protected addresses. Its format is as follows:

```
SA <address>
```

For example, the following command would remove &1900 from the protection table:

```
SA &1900
```

14.4.5 Realtime tracing

Single-stepping through time-critical subroutines to find errors will cause complications. The commands described below are intended to help you in this situation. They include the following:

RS Set realtime point
RC Clear realtime point
DR Display realtime points

Set realtime point (RS)

This command has the following format:

```
RS <address>
```

<address> will be added to the table of realtime addresses. Whenever a JSR instruction is executed, its address will be checked against those in the table. If it is found, then instead of simulating the JSR instruction, the call will be made from within TRACE and the subroutine will run at real time speed.

Clear realtime point (RC)

This command removes an address from the realtime table. It has the following format:

```
RC <address>
```

Display realtime points (DR)

This command gives you a list of realtime points, with their expected values,in descending order of addresses. All you need do is type 'DR' and you will get a display something like the following:

```
1950
2102
3000
4002
```

14.4.6 Miscellaneous

Set sideways ROM number (SR)

This command allows you to trace code in sideways ROMs. It has the following format:

```
SR <hex value>
```

If <hex value> is > = 0, tracing will take place only in the sideways ROM specified. If it is < 0 tracing will take place in all sideways ROMs.

Since this command applies to the sideways ROMs in the BBC Microcomputer, it does not apply when tracing in the Tube.

Restrict tracing of operation codes (TO)

This command restricts the class of operation codes which will be reported. It has the following format:

```
TO <integer>
```

the contents of <integer> have the following significance:

<bit>	Class of operation codes reported
0	Control codes
1	Loops
2	Stacks and tests
3	Arithmetic and the rest

Thus,

TO &F traces all instructions
TO &3 traces loops and control codes only
TO &8 traces arithmetic instructions

APPENDIX A The macro substitution method

The following algorithm is used to substitute variables and parameters when a macro is called:

1 Substitute macro parameters throughout the macro

2 For each line do the following:

   (i) If it is not a directive, substitute for string variables in the line

   (ii) Else, if the directive is not LCL or GBL, substitute for strings after the directive

   (iii) Else (for LCL or GBL), do nothing

3 Next line, until MEND or MEXIT

APPENDIX B MASM error messages

MASM error messages can be divided into two groups: those which always stop an assembly (called 'fatal' errors) and those which stop an assembly only if STOP ON is set (called 'non-fatal' errors). Both groups are listed below:

B.1 MASM fatal errors

Assembly stopped

This is printed at the end of the first pass of an assembly in which one or more errors were detected.

ASSERT failed

This is caused when the condition part of an ASSERT directive did not yield a TRUE result. It warns you that something you assumed to be true about the assembly was not.

Bad command

This is given when MASM does not recognise a command typed in response to the 'Action:' prompt. The HELP command will cause a list of valid commands to be displayed.

Bad expression

When one of the expressions given in response to the SAVE command's prompts cannot be understood by MASM (eg contains an identifier it doesn't know about), this error is given.

Bad FS

This means 'Bad filing system' as is given when an attempt is made to use disc MASM on the Net/ADFS.

Bad macro definition

This is caused by an error occuring between the MACRO and MEND parts of a macro definition, eg a badly formed MACRO line.

Bad macro library

If the response to the ASM command's 'Macro library:' prompt does not yield a string that can be used as a filename, this error is given.

Bad nesting

This is caused by the end marker of one 'structure' being encountered when another type of structure is still open. It is roughly analogous to ending a FOR loop with an UNTIL in BASIC. The structures of MASM are WHILE..WEND, MACRO..MEND, and `[..|..]`.

Bad offset

This is caused when the response to the GET command's 'Offset:' prompt does not yield a suitable number.

Bad option

This is caused by specifying an illegal option in one of the MASM commands. For example, only 'A' and 'N' are valid SYMBOL options.

Bad processor

This is caused by giving an illegal processor type letter in the SAVE command. Valid processors are 'T' or 'P' (for the Tube), or 'I' or 'H' (for the I/O processor).

Bad qualifier

This error is given when an attempt is made to assemble a source file in directory X. This is where MASM puts the object files, so the source file would be overwritten.

Bad value

This occurs in the WIDTH and LENGTH commands when an illegal number is given.

Can't open

This means that MASM can't open the XREF output file, for example, because a locked file of the same name already exists.

Code overwriting source

This occurs when the amount of code generated by a program is much larger than the source code generating it. It should never occur if MASM is used properly.

Doubly defined MACRO

This error is given when the same macro name occurs in more than one MACRO directive. Macros should only be defined at a single point, and can't be redefined.

END/LNK in macro

The only thing that may terminate a macro definition is MEND. If either of the above directives occurs in a macro definition, this error will be given.

Escape

This is printed whenever the user presses ESCAPE.

Expression stack overflow

This means that an expression in an operand is too complex for MASM to evaluate. It should only occur in expressions where brackets are nested extremely deeply.

File too big

Source files must be less than 17K. This error means you should split a large source file into two or more smaller ones.

Heap overflow

This means MASM has run out of space in which to store string variables. It can only be cured by changing the source program to use shorter strings.

Local label table overflow

This is caused by calling macros with local labels very frequently. See the command MLEVEL for a way of curing the problem.

Mac def in expansion

This means MASM encountered two MACRO directives without an intervening MEND. Although macro calls may be nested, macro definitions may not.

Macro nesting too deep

Macros may only call each other (or themselves) to a level of eight deep. Deeper nesting causes this error.

Macro parameter table full

This occurs when the text to be substituted by macro parameters totals more than 250 characters.

Macro space exhausted

This means MASM has run out of space to save macro definitions. It is very unlikely that this will happen, but if it does you will have to use fewer or shorter macros.

No macro being defined

This is caused when a MEND directive is encountered without a corresponding MACRO.

No symbols

This is printed when a SYMBOL command is issued before an ASM or after an ASM in which no symbols were defined.

Stack fault

Will only occur if there is a bug in MASM: report it to Acornsoft.

Stack overflow

This occurs when structures (WHILE..WEND, `[..|..]`) are nested too deeply. It will not occur with sensible use of the structures.

Stack underflow

See 'Stack fault'.

Stopped

This is printed when a ! directive is used and causes assembly to stop. See chapter 10.

Symbol table overflow

This occurs when more than 1536 symbols have been encountered in the source. The BBC MOS, BASIC and Acornsoft COMAL all assemble without this error occurring, so you shouldn't get it.

Too many macros

See 'Macro space exhausted'.

B.2 MASM non-fatal errors

Bad directive use

This occurs when a symbol used in a directive's expression has not been defined yet.

Bad JMI

This is caused by an instruction of the form JMI &XXFF when MASM is in CPU 0 mode, which will crash when executed on a standard NMOS 6502 (but not the CMOS type).

Bad label

A label must be between one and seven characters long and may only contains letter or digits. It must start with a letter. Anything in the label field of a line not obeying these rules will cause this error to be generated.

Bad local label number

These must be two-digit numbers (see section 8.3).

Bad drive number

This is printed when the number after a > or < directive is not a valid drive.

Bad opcode

This is caused by an unrecognised instruction/directive in the opcode field of a line of assembly source.

Bad operand

This is reported when an expression contains an object which should be an operand, eg a label or variable, but isn't recognised as such.

Bad operator

This is reported when an expression contains an object which should be an operator, eg + or :CC:, but isn't recognised as such.

Bad OPT

The expression after the OPT directive should lie in the range 0-15.

Bad routine name

The name of a routine (section 8.3) should conform to the same rules as any other label.

Bad string length request

This occurs when an attempt is made to create a string variable whose length is greater than 127 characters.

Bad zero page value

This is given when an address operand which should be less than 256 isn't, eg LDAZX 4321 will give this error.

Badly defined manifest symbol

This occurs when a symbol is forward-referenced too indirectly for the assembler to resolve it by the end of the first pass. An example of code that might cause this is:

```
LDA A
A * B
B * 1
```

In this example, the symbol A is defined by a forward reference to B. Thus B will be known by the end of the first pass, but A won't be known until the end of the second pass. The LDA instruction cannot therefore know the value (and hence number of bytes of) its operand, and cannot be assembled.

Division by zero

This occurs during expression evaluation when the right hand operand of a divide operator is zero.

Double defined variable

A variable defined using the GBL directive may only be defined once. Thereafter it may be set to different values using SET.

End of line missing

This occurs when the end of the source file is encountered halfway through a source line. All lines in MASM source should be terminated by a carriage return.

Expansion line too long

When string variables and macro parameters are substituted, they may make the source line longer than it was. If the line grows to more than 250 characters after substitution, this error will be given.

Label already defined

Once a label has been set to a certain value, it retains it for the rest of the first pass and all of the second pass. An attempt to redefine a label that has already been set will yield this error.

Line too long

A source line must be less than 255 characters long, and terminated by a carriage return character.

Missing [

If the assembler comes across an 'else' (!) or 'endif' and has no corresponding 'if' ([), this error is given.

Missing WHILE

This error occurs when a WEND directive is encountered and the assembler has not had a WHILE to match it with.

No current macro

This is given when an attempt is made to access local variables outside of the macro level. Only global variables may be used outside of macros.

Offset to <label> out of range

This occurs in branch instructions when the destination label is more than 129 bytes after or 126 bytes before the first byte of the branch.

Syntax error

This means that MASM is unable to make any sense of a line of source code.

Too late for LCL directive

The LCL directives must be the initial lines of a macro definition. Inserting other directives or code-generating lines between the MACRO and LCL will cause an error to be given.

Too late for ORG

The ORG directive must be given before any code has been generated by the source in the current file.

To late to change CPU

The CPU directive must be given before any opcodes have been encountered in the assembly.

Type mismatch

This is given when strings and numbers are mixed illegally in expressions, eg 123 :CC: "MASM" (both operands of :CC: should be strings).

Unknown symbol

This error is caused by an attempt to access a symbol which has not been defined in an expression.

Unknown variable symbol

This is caused in a similar way to the last error, but refers to symbols that are preceded by the variable sign '**$**'.

Index

6502 Development Package

on the BBC Microcomputer with 6502 Second Processor

About this book

This manual describes how to use the utilities contained in the 6502 Development Package. These provide a powerful means of producing machine code programs using a BBC Microcomputer and a 6502 Second Processor.

It is assumed that the user is familiar with the concepts covered in the *BBC Microcomputer System User Guide* and to some extent with 6502 assembler mnemonics.

The *6502 Development Package Reference Card* accompanying this manual provides a summary of the facilities available in a convenient form.

About the author

Barry Morrell has worked in the computing industry since 1970. Since 1976 he has been a technical author, managing a team of authors for a national computer manufacturer before becoming a freelance writer.

Acornsoft Limited, Betjeman House, 104 Hills Road,
Cambridge CB2 1LQ, England. Telephone (0223) 316039

ISBN 0 907876 93 5

SBD22