MAXAM II

MACRO ASSEMBLER
EDITOR
MONITOR

For the CP/M Plus Operating System

Amstrad PCW 8256/8512
CPC 6128

Issue 1, (c) Arnor Ltd., March 1987

Any correspondence relating to Arnor products is welcomed. Specific comments should quote the version number, which is displayed at the top of the assembly listing and in the editor and monitor status screens.

This manual was written using PROTEXT and printed from camera-ready copy produced by PROTEXT on a Juki 6100 printer.

The MAXAM II programs were written using MAXAM and subsequently MAXAM II.

Arnor Ltd., 611 Lincoln Road, Peterborough PE1 3HA, England.

CONTENTS

All that is gold does not glitter,
Not all who wander are lost;
The old that is strong does not wither,
Deep roots are not reached by the frost.

From the ashes a fire shall be woken,
A light from the shadows shall spring;
Renewed shall be blade that was broken,
The crownless again shall be king.

J.R.R.Tolkien

INTRODUCTION

The Maxam II Assembler, Editor and Monitor provide the most versatile assembly language programming environment available for CP/M Plus on the CPC 6128, PCW 8256 and PCW 8512.

The macro assembler is an enhanced version of the best selling Maxam assembler. It has been specially tailored to meet all CP/M Plus programming requirements and, with the addition of macros, is extremely fast and powerful.

The editor is a full implementation of the Program mode of Protext.

The monitor has been written to complement the assembler, including many new features not available in other monitors, such as bank switching, conditional breakpoints and symbolic debugging.

Production of a program from coding using the editor through assembly with the macro assembler to final de-bugging using the monitor is a straight-forward process. All the programs link with each other: the assembler reads the editor files, the monitor disassembles code produced by the assembler and allows the symbol table to be loaded for symbolic debugging, and the editor can read spooled files from both the assembler and monitor.

The syntax used by the assembler and monitor in assembly mode is identical, you can even overwrite or add to code within the monitor using the in-built assembler and include the same symbols you defined in the Assembler.

Throughout this manual you will find detailed explanations of each feature with many examples of how to use them. As well as the three main sections on the assembler, editor and monitor there are a number of appendices providing information on the CP/M BDOS calls including how to access discs, reading the keyboard and writing to the screen.

For the beginner there is also a section introducing CP/M plus which will help you to start writing programs almost immediately. If you are new to the Z80 Microprocessor the section on the assembler provides much of the information you need, and the appendix listing all the Z80 mnemonics is provide so you can easily find the instructions you require.

This manual does not attempt to teach machine code programming, and assumes basic familiarity with the Z80 assembly language. If you are a newcomer to machine code programming you will need to consult a book on the subject. Though it may seem bewildering at first, persevere - machine code programming is both rewarding and enjoyable. You have made the right decision to purchase MAXAM II - as well as providing a sophisticated program development environment it is the ideal system for learning machine code.

CREATING A START OF DAY DISC

Before using Maxam II you will need to create a start of day disc. This ensures that you do not damage your original master disc and allows you to configure the disc to your own requirements.

You will need two CP/M Plus formatted discs with system tracks. On the CPC 6128 you can do this using DISCKIT3. On a PCW computer use DISCKIT. Simply follow the on-screen instructions. One of the discs must be formatted on both sides.

Label the disc you formatted on one side only "MAXAM II Start of day". Label SIDE 1 of the other disc "MAXAM II LARGE" and the other side "MAXAM II SMALL".

Follow the instructions in the appropriate section below.

PCW 8256/8512

1. Load CP/M
2. Insert the supplied MAXAM II disc in drive A with side 1 to the left.
3. Type one of the following:

   APED <MAKE8512 (For PCW 8512)
   APED <MAKE8256 (For PCW 8256)

4. Follow the on-screen instructions

CPC6128 with two disc drives

1. Load CP/M
2. Insert the supplied MAXAM II disc in drive A with side 1 uppermost.
3. Type:

   APED <MAKE6128 (For CPC 6128 with two drives)
4. Follow the on-screen instructions
5. Type DCOPY
6. Insert your MAXAM II LARGE disc in drive A
7. Insert the supplied MAXAM II disc in drive B
8. Press S and wait until copying is complete
9. Turn over the disc in drive A
10. Turn over the disc in drive B
11. Press "Y" then press "S"
12. When copying is complete press "N" to return to the editor.

CPC 6128 with one disc drive

1. Load CP/M

2. Insert the supplied Maxam II disc in drive A with side 1 uppermost.

3. Type: DCOPY

4. Follow the on-screen prompts inserting SIDE 1 of the supplied Maxam II disc at the "original disc" prompt, and inserting your MAXAM II LARGE disc at the "Copy disc" prompt

5. When copying is complete type "Y" to copy another disc

6. Follow the on-screen prompts inserting SIDE 2 of the supplied Maxam II disc at the "original disc" prompt, and inserting your MAXAM II SMALL disc at the "Copy disc" prompt.

7. Type "N" to finish copying discs

8. Type: APED to enter the Arnor program editor.

9. From now on at the prompt "Disc for A" insert your start of day disc in the drive, and at the prompt "Disc for B" insert your CP/M plus system disc (purple disc 1) in the drive.

10. Type COPY B:*.EMS A and follow the prompts.

11. Type COPY B:SUBMIT.COM A and follow the prompts.

12. Press ESC.

13. Type APED then press RETURN

14. Press ESC.

15. Type SAVE A:PROFILE.SUB

16. Insert your start of day disc at the prompt "Disc for A" and SIDE 2 of the Maxam II Master disc at the prompt "Disc for B.

17. Type COPY B:A*.* A and follow the prompts.

18. Type COPY B:*.PTR A and follow the prompts.

DEVELOPING A PROGRAM WITH MAXAM II

This section assumes basic familiarity with the editor. If you are unsure how to type in any examples read the section on the editor.

The following program calls the console input function via BDOS, stores each character entered and quits when RETURN is pressed. There is a deliberate bug introduced into the program. If you haven't spotted it yet we'll find out what it is when we look at debugging the program.

```
                org &100        ; Start

conin   equ 1   ; BDOS console input function
bdos    equ 5   ; BDOS call entry point
return  equ &2a ; Carriage return

                ld hl,buffer    ; Start of input buffer

loop            ld c,conin      ; BDOS function number
                call bdos       ; Get a character
                cp return       ; Is it return?
                jr z,done       ; Yes, go to done
                ld (hl),a       ; No, store it in the buffer
                inc hl          ; Increment pointer into buffer
                jr loop         ; Get another character

done            ret

buffer          rmem &100       ; Reserve &0100 bytes of memory

                end             ; End of program
```

To assemble this program put your start of day disc into drive A and enter CP/M. On a CPC 6128 type:

```
|CPM
```

or on a PCW switch on or press:

SHIFT-ALT-EXIT.

This takes you automatically into the editor. Once in the editor type in the program exactly as it appears and save by typing:

SAVE MYPROG

while in command mode then enter:

ASM MYPROG

Provided all is well you will see the program assembling on the screen. On completion of assembly the program MYPROG.COM is saved to disc.

Now run the program by typing:

*MYPROG

You will see that you can type in characters but can't quit by pressing RETURN. Type:

*

you'll see why in a minute.

So, we've found a bug. Well, there's a simple way of fixing that. Type:

MM MYPROG

This takes you into the monitor which loads in your program automatically.

We are now ready to debug the program. To do this we'll set a conditional breakpoint. Type:

SC A=13

This means "If the contents of register A is 13 then stop".

To run the program type:

SS

This means single step slowly. You'll see a number of lines of information scrolling by at the bottom of the screen. They are showing you the contents of the registers and flags and the current instruction being executed. Now type in some characters at the keyboard. You should see the contents of HL is incremented each time you do so. This is correct because characters are being entered into the buffer pointed to by HL which is incremented each time. Now to test the program press RETURN.

The scrolling has stopped and the next instruction to be executed is shown at the bottom of the window on the left. It should read: CP &2A. We set the conditional breakpoint routine to stop if A=13 but now we're checking whether a=&2a which is a "*".

Here's the bug, we were checking for the wrong character. Now we've solved the problem quit from the monitor by typing:

Q followed by Y

Re-enter the editor by typing:

APED

Then reload the source code and replace the line:

```
return equ &2a
```

with:

```
return equ 13
```

Having done this, and assuming there are no further errors, you can assemble a working program.

Although this is deliberately a simple example, you will see how easy it is to develop and debug a program by going through these three simple stages.

Of course, there are many more facilities available within Maxam II which extend the ease of development far further. These are all explained step by step as you progress through the manual.

THE ASSEMBLER

THE ASSEMBLER

You can run the assembler either from the editor or from CP/M. From the editor type ASM <filename>. If the filename is omitted the current program being edited will be assembled. If there is no program in memory you will be prompted. The object filename defaults to the source filename with the extension .COM, or may be specified by a second parameter in the command tail.

From CP/M type MA <filename>. Again, if no filename is specified you will be prompted.

Program format

The program to be assembled consists of a sequence of statements. Each statement has the format:

<label field> <instruction field> <comment field>

One line may contain several statements, separated by colons. A statement is made up of the three parts shown above, each of which may be empty. The comment field, if present, must start with a semicolon - the effect of a semicolon is that the assembler will ignore all characters until the next colon or end of line. To allow colons in comments use two semicolons together. All characters to the end of the line are then ignored.

Instructions and names used in the source code may be typed in upper or lower case. Thus 'START', 'start' and 'Start' all refer to the same label.

The label may be preceded by a full stop. This tells the assembler that what follows is a label. If there is no full stop the assembler will attempt to recognise a Z80 instruction or assembler directive, and if it fails will take the first item as a label. Labels must begin with a letter and can be any length.

Thus to use a Z80 instruction as a label it must be prefixed with a full stop, e.g. '.halt'.

Assembler error messages

Whenever an error occurs during an assembly the offending line is listed, a beep is sounded, and a self explanatory message is displayed.

There are 3 degrees of severity of error that can be produced by the assembler. The most serious is 'fatal error' which causes the assembler to give up immediately. The different fatal errors are listed in the reference section.

Fatal errors are reported on the first pass. All other errors are reported on the second pass, and do not cause the assembler to give up. Instead the numbers of each type are counted and the total numbers are printed when the assembly is finished.

The least serious is 'warning'. A warning message is given for something which can be assembled but it is likely that the programmer meant something different. This occurs in the following cases:

1. An expression evaluates to more than 8 bits, when an 8 bit value is required, e.g. LD A,300. Note: a warning is not given if the high order 8 bits are all 1, so e.g. LD C,-3 is allowed.
2. Spurious text is found after the statement has been correctly assembled. This may be the result of missing out a colon or semicolon.

All other errors are labelled 'error'. If any occur the program will have to be re-assembled.

The ORG directive

In the absence of a directive telling the assembler where to put the code it will automatically assemble it at &0100. The ORG directive tells the assembler what address to use.

Notes: (i) Any number of ORG directives may be used.
(ii) The expression may not contain undefined symbols.

NOCODE and CODE

Syntax: NOCODE
Syntax: CODE

Occasionally it is useful to assemble a program without writing any code to file - perhaps just to check that it assembles correctly, or to assemble a small routine which is to be input in hexadecimal (maybe on another computer). The directive NOCODE achieves this. The directive CODE cancels the effect of NOCODE.

W8080

If you are writing code you wish to transport to an 8080 based micro you can still write the code using Z80 mnemonics (many of which have direct 8080 equivalents) but using the W8080 directive you will be warned of any instructions which are not 8080 compatible during assembly. This also makes it easier to adapt Z80 code to 8080.

Note that 8080 code may be directly converted to 8088/8086 code, and so this mode should be used if programs will need to be converted to run on an IBM PC or Amstrad 1512.

The END directive

Syntax: END

The END directive simply tells the assembler to stop. It may be omitted, but has two uses:

1. To avoid assembling the whole program - temporarily put in an END directive.

2. END causes the storage location to be output in the listing. A useful ploy is to put 'LIST:END' as the last line of source code so you can see where the end of the program is.

Expressions

Arithmetic expressions may be used throughout the assembler - wherever a number is required. This includes operands of Z80 instructions and assembler directives. The expression evaluator is documented in the appendix on "The Expression Evaluator".

All expressions are evaluated to 16 bit unsigned integers. Overflow is ignored, and the least significant 16 bits of the result is used.

Symbols

The assembler keeps a table of symbols, each with an assigned 16 bit value. A symbol is similar to a BASIC variable. The assembler makes two passes; on the first pass it sets up the symbol table and on the second pass it creates the object code using the symbol table to calculate jump addresses etc. On the first pass, when a symbol that has not yet been defined is referred to it is put into the symbol table. The value is filled in when the symbol is defined. These forward references must all be resolved on the first pass; error messages will indicate any symbols that remained undefined. No symbol may be assigned different values on the two passes - if this occurs the assembler may generate many errors.

Macro names and local macro labels are also included in the symbol table.

There are some assembler directives which do not allow any forward references because the expression value must be known on pass 1. These include ORG - the code origin must be well-defined for it would otherwise be impossible for the assembler to generate the correct symbol table. The full list of these directives is given in the reference section.

An identifier is the name of a symbol. Valid identifiers must satisfy the following rules:

1. The first character must be a letter.

2. The other characters may be any of: letter, digit, question mark (?), full stop (.), underline (_).

There is no length restriction, nor are there any reserved words.

4 ways to define a symbol

1. As a label. This is an identifier at the start of a statement, possibly preceded by a full stop. The symbol is assigned the value of the current code location.

2. By the EQU (equate) directive.
<identifier> EQU <expression>

The symbol is defined and assigned the value of the expression, which must be well-defined (i.e. contain no forward references). If the symbol is already defined an error message will be given (unless the old value and the new are the same). In other words, EQU may not be used to redefine a symbol.

3. By the LET directive.
   LET <identifier> = <expression>

This has the same effect as EQU except that LET allows redefinition of symbols. Note: for compatibility with other assemblers this may be written <identifier> DEFL <expression>.

4. As a macro (see "MACROS").

Putting data into the object code

The 3 directives explained in this section cause data to be assembled at the current code location. In all cases both the code location and storage location are incremented.

BYTE <list of expressions and strings>
TEXT <list of expressions and strings>

BYTE and TEXT are different names for the same thing. They take a list of parameters, each of which can be an arithmetic expression or a text string. Each expression is evaluated and the result put in the object code. Each string is sent directly to the object code, character by character. Strings may be enclosed in either single or double quotes; if the closing quote is omitted the string is assumed to be the rest of the line.

Note: a single character string is considered a numeric constant, so expressions such as "A"+&80 are allowed.

```
Examples:       BYTE 1,3,count*3+1,"q" or 128
                TEXT "A string ending with cr-lf",13,10
```

WORD <list of expressions>

Each expression is evaluated and the 2 byte result put in the object code, low byte first.

```
Example:        WORD &C000,address
```

STR <list of strings>

STR is similar to BYTE and TEXT with the exception that it will only take a list of strings and the last character in each string has the top bit set. This is useful when printing a string character by character, as you then only need test if each character has its top bit set to know when you've reached the end of a string.

Example: STR "Hello world"

RMEM <expression>

RMEM causes the assembler to reserve the specified number of bytes of memory. The object code location is incremented by the value of the expression. The reserved space is filled with zeros. The expression may not contain forward references.

Examples:

```
.buffer256      RMEM 256
.word           RMEM 2
```

Occasionally the reserved space needs to be filled with a value other than zero. This can be done by giving a second expression parameter. The space is filled with the least significant byte of the expression's value.

Example:

```
.data           RMEM &200,&FF
```

Compatibility with other assemblers:

The following alternative directive names are allowed:

```
DEFB, DB, DEFM  ... same as BYTE, TEXT.
DEFW, DW        ... same as WORD.
DEFS, DS        ... same as RMEM.
```

Conditional assembly

Conditional assembly is used when two or more versions of a program are needed (eg. Amsdos and CP/M Plus versions on a CPC 6128). This feature enables any number of different versions to be assembled from the same source code.

This is done by defining blocks of source code that are to be assembled only if some condition holds. The formats of IF blocks are:

```
1.      IF  <expression>
        <code to be assembled if expression is true>
        ENDIF

2.      IF  <expression>
        <code to be assembled if expression is true>
        ELSE
        <code to be assembled if expression is false>
        ENDIF
```

The expression may be any arithmetic expression. In this context the value of the expression is considered to be a signed 16 bit number, with 'true' represented by any non-zero number and 'false' by zero.

```
Examples:       IF FLAG AND 32
                IF $-start>&2000
                IF FLAG1 OR FLAG2
```

A recommended use is to define a variable which holds the value -1 for true and 0 for false.

Example: suppose a program comes in two versions, for Amsdos and CP/M Plus, and there are a few differences between the two. Define a variable at the start of the source code:

```
LET amsdos=-1    ; to assemble the amsdos version
LET amsdos=0     ; to assemble the CP/M version
```

Then enclose each section where the code differs in an IF block, as follows:

```
IF amsdos
<code for amsdos version>
ELSE
<code for CP/M version>
ENDIF
```

Complex expressions may be used in IF directives, and will work as expected if true is any positive or negative value and false is zero. So if variables which only ever hold the values 0 or -1 are used the usual results hold (-1 OR 0 is true, -1 AND 0 is false, NOT -1 is false, and so on).

```
Example:        IF flag1=1 and flag2=2
```

Warning: although 1 and 2 both represent true, the expression flag1 and flag2 evaluates to 0 (i.e. false). Use instead:

```
                IF [flag1>0] and [flag2>0]
```

IFNOT

For convenience IFNOT may be used instead of IF. It simply reverses the logic of the IF directive. It is retained for compatibility with the MAXAM I assembler and is the same as IF NOT <expression>.

```
IFNOT <expression>
<code to be assembled if expression is false>
ELSE
<code to be assembled if expression is true>
ENDIF
```

Nesting IF blocks

IF blocks may be nested up to a depth of 10. It is, however, unusual to need nesting deeper than 2 levels.

```
Example:        IF amsdos
                <amsdos code>
                ELSE: IF large_cpm
                <CP/M code>
                ELSE
                <CP/M small version code>
                ENDIF
                ENDIF
```

IF1, IF2

These special forms of the IF directive return the value 'true' on pass 1 and 2 of the assembly, respectively. They may be of some use for printing different messages on each pass, but Z80 instructions and directives should not be placed within an IF1 or IF2 block.

REPEAT/UNTIL

A Repeat/Until loop is a powerful form of conditional assembly allowing you to assemble sections of code (possibly adapting parts using IF etc.) repeatedly until the condition following the UNTIL is true.

Example:

```
LET row=1

REPEAT

IF row=1 or row=8
BYTE 1,1,1,1,1,1,1,1

ELSE
BYTE 1,0,0,0,0,0,0,1

ENDIF

LET row=row+1
UNTIL row=9
```

This example creates an 8 by 8 data table with a border of ones around the outside. Of course you could develop the above routine even further by allowing for a data table of n by n size, and so on.

Note: REPEAT must be on a line of its own and REPEAT/UNTIL loops cannot be nested.

Reading source code from a file

Syntax: READ <filename>

When the assembler finds a READ directive it will open the specified file, assemble the contents of the file, and then return to the line in memory following the READ directive.

The file should be a text file (produced by APED or another editor). The filename can be enclosed in quotes. If quotes are omitted the assembler will assume the filename to be the string following the READ directive to the end of the line.

Nested reads are supported. So a file read in can also read from another file.

Note: Nothing can follow a READ statement on the same line.

STOP

The STOP command causes reading from the file containing the STOP to terminate. In the case of nested reads the assembly returns to the last read file. This command may be useful when reading a number of subroutines from a file leaving out others.

A useful hint

Although READ displays each filename it only does so on the second pass. If the program is split between several text files it is helpful for the assembler to print the name of each file it reads on the first pass allowing you to keep track of the assembly. This is easily accomplished by making the first line of each file something like:

```
1 PRINT "<name of file>      <date>"
```

This is also useful when editing; without a name at the top of a file it is easy to forget which file you are editing.

The number 1 causes the assembler to reset its line counter to 1. This means that error messages will give the correct physical line number within the file where the error occurred. The editor has a command to move to a specified line, so using these features together speeds up debugging.

Writing object code to a file

Syntax: WRITE <filename>

By default the assembler writes a file to disc using your filename but with a COM extension. The WRITE directive tells the assembler to create a file using the filename you specify, and store all subsequent object code in the file (unless disabled with NOCODE). If no filename extension is given the the extension .COM is used.

The extension .L has a special meaning (see the section on linking).

Assembler commands

Commands control the listing and output produced by the assembler. They do not appear on the assembly listing themselves unless a label is attached to the command or there is an error in the command.

LIST, NOLIST

LIST turns on the assembler listing. This is the initial state.
NOLIST turns off the assembler listing.

PRINT <string>

The string is displayed on the screen, even if the listing is turned off. The string may, optionally, be enclosed in quotes. This string can include variables. To print the value of a variable in hexadecimal precede it with "&" or to print in decimal precede it with "$".

example:

```
PRINT "The code ends at &endprog and is $len bytes long"
```

PAUSE <string>

The assembler will wait until a key is pressed. PAUSE only operates if listing is enabled. It allows part of a long listing to be examined. A string to be printed can follow the pause. This string can include variables as with PRINT.

BEEP

BEEP, as its name suggests, causes the assembler to beep. It is useful in conjunction with PRINT when prompting to change discs or issuing a warning.

INKEY

The command INKEY instructs the assembler to wait for a key press and set a variable to the Ascii value returned.

Example:

```
PRINT "Do you want the (L)arge or (S)mall version? "
INKEY version

IF version="L" or version="l"
<large version code>
ELSE
<small version code>
ENDIF
```

DUMP

If a DUMP command appears anywhere in the program when listing is enabled a complete alphabetical list of all defined symbols with their values in hexadecimal will be produced when the assembly has finished.

Listing to the printer, file or screen

OUTPUT F,S,P

The output command can be followed by any combination of F, S or P (for file, screen or printer). Output following this command will be directed to the selected output devices.

If you select the F option the file written to will have the filename of the source file by default. If the WRITE command is used the output file will have that filename. In either case it will have the extension .PRN.

Seven parts of a listing

There are 7 parts to a listing as can be seen in the following example. They are:

1. Line number. The assembler looks for a line number at the start of a line (in decimal). It it finds one it uses it, otherwise it counts the lines. These line numbers refer to physical lines - colon separators do not change the line number.

2. Code location.

3. Object code. Up to 4 bytes per line are displayed. Directives may cause more than 4 bytes to be assembled, in which case the object code will be listed on more than one line - 4 bytes to a line.

4. Label field.

5. Instruction field.

6. Operand field.

7. Comment field.

```
00001                                           ; Example listing
00002
00003                           output f,s
00004
00005 0100  (0100)              org   &100      ; Start
00006
00007 0100  (0001)      conin   equ   1         ; BDOS console input
00008 0100  (0005)      bdos    equ   5         ; BDOS call entry point
00009 0100  (000D)      return  equ   13        ; Carriage return
00010
00011 0100  21 11 01            ld    hl,buffer ; Start of input buffer
00012
00013 0103  0E 01       loop    ld    c,conin   ; BDOS function number
00014 0105  CD 05 00            call  bdos      ; Get a character
00015 0108  FE 0D               cp    return    ; Is it return?
00016 010A  28 04               jr    z,done    ; Yes, go to done
00017 010C  77                  ld    (hl),a    ; No, store in buffer
00018 010D  23                  inc   hl        ; Increment pointer
00019 010E  18 F3               jr    loop      ; Get another character
00020
00021 0110  C9          done    ret
00022
00023 0111  (0100)      buffer  rmem  &100      ; Reserve &100 bytes
00024
00025 0211  (0211)              end             ; End of program

Errors: 00000  Warnings: 00000
```

Several directives cause a number to be printed after the address in parentheses. The directives are as follows:

```
END       : the storage location.
EQU       : the value assigned.
IF,IFNOT  : the value of the conditional expression.
LET       : the value assigned.
ORG       : the storage location.
RMEM      : the number of bytes reserved.
UNTIL     : the value of the conditional expression
```

PLEN <expression>

Without a PLEN command the listing is continuous with no page breaks. PLEN defines the number of lines per page. To use this make sure the printer is at the top of the page (exactly where the first line is to be printed). Set PLEN to the exact number of lines per page. This is not the number of lines to be printed - a few blank lines are automatically left at the bottom of the page.

The value of the expression may be either 0 or between 40 and 255. PLEN 0 tells the assembler to revert to continuous listing.

Examples:
```
PLEN 66     for 11" paper
PLEN 72     for 12" paper
```

PAGE (<expression>)

The command PAGE causes a page eject. The page length will be used to calculate the number of blank lines to be printed so the new page starts at the right place on the paper.

The expression is optional. If supplied this will be used as the new page number. This may not exceed 255. If no number is given the page number will be one more than the previous page number.

PAGE is ignored if listing is disabled.

TITLE <string>

This defines a title to printed at the top of each page. For this to be printed on the first page, the TITLE command must appear before the first directive or mnemonic. The title will be printed starting in column 1, so to centre the title include the necessary number of spaces in the string.

TITLE with no string will cancel the titling option, whereas TITLE "" will give a blank title line.

WIDTH <expression>

This sets the number of characters per line in the listing. The default setting is the current screen width (40 or 80), but it may be set to any value between 40 and 255. WIDTH 0 causes the default setting to be restored.

Example: WIDTH 132

External commands

External commands can be included in a source file by prefacing them with a *. The commands supported are:

*ERASE <filename>
*DRIVE <name>
*GROUP <number>
*ACCESS <filename>
*PROTECT <filename>

Where wildcards are normally accepted they can be included in these external commands.

Example: *ERASE *.BAK

All external commands are executed on both passes by default. If this is not required you can select exactly which commands are executed and when using IF, IFNOT, IF1, IF2 and so on.

Example:

```
IF1
*ERASE *.BAK
ENDIF
IF2
*RENAME <newname> <oldname>
ENDIF
```

LINKING SEPARATELY ASSEMBLED SOURCE FILES

MAXAM II allows you to link files in many ways. The following shows how you can link two programs produced by the assembler. It is also possible to link programs with Arnor C object code. How to do this is explained at the end of the section.

SYM

The SYM command saves all the symbols used in your program to disc. This symbol table can be used later to link programs together or can be loaded into the monitor along with a program allowing symbolic debugging.

SYM should only be entered after the last symbol has been defined. The best place to put it is just before END statement.

```
Example:        SYM progname.sym
                END
```

LINK

LINK enables you to link two programs together. A program to be linked has all its symbols saved, unlike with other assemblers where only the symbols declared PUBLIC are saved. This ensures that no symbols are missing. Also, a program to be linked is not relocatable but this is not important as will be seen below.

The command LINK must be entered before any symbols are defined.

A program that is going to be linked must follow this format:

```
                ORG  &101
                WORD endofprog-$
                <main program>
                ...

endofprog       SYM progname.sym
                END
```

This is to pass the length of the program to the link routine. The bytes following WORD are replaced on linking with a jump address which points to the start of 2nd program.

A program that is linking in another must follow this format:

```
            ORG  &100
            JP   start
            LINK "progname.com" "progname.sym"
start       <main program>
            ...

            END
```

So each time you link, the previously assembled program is always loaded in at &0103. Execution will begin with the last assembled section of code.

This makes it easy for you to create a library of subroutines which you use all the time, assemble it once and thereafter link in the assembled code, thus saving a lot of unnecessary re-assembly.

All three Maxam II programs were written this way.

LINKING ASSEMBLER AND ARNOR C PROGRAMS

WRITE

Specifying a write filename with the extension .L creates an Arnor C link file. When this happens the following two commands become available.

PUBLIC <list of symbols>

PUBLIC causes only those symbols following the command to be stored in the link file. These symbols can then be accessed by the C linker.

EXTERN <list of symbols>

The symbols specified by an extern command will have been created by the C compiler. You must declare the symbols you will be using for them to become available to your program.

A link file must be fully relocatable. For this reason there are a number of rules you must remember when creating a link file.

1. If you are creating a link file you must not use ORG.

2. If you are not creating a link file you can not use PUBLIC or EXTERN.

3. Your code length may not exceed 32k. If this occurs an error message will be displayed. The code should then be split into two or more sections.

4. Care must be taken to ensure code is relocatable. All 2 byte operands with a value between &100 and &80FF (the highest allowed code address) are considered relocatable. If an operand is a constant it should not be used as a 16 bit value - instead of LD HL,constant use LD H,highbyte:LD L,lowbyte. Relocatable 8 bit values should be avoided - instead of LD A,address MOD 256 use LD BC,address:ld A,C

An explanation of how to use an assembler link file with Arnor C is included in the C manual.

DRW <list of expressions>

Define relocatable word. This is the same as the WORD directive except that the values of the expressions are considered relocatable.

MACROS

Macros are ideal for making your source code more legible to other people (and yourself when you come back to it at a later date), reducing the size of your source code and building libraries of routines which assemble according to the parameters passed. Here is an example of a listing including macros:

```
PRINT "Macro examples"

MLON

                ORG &100

                MACRO addhla
                add a,l
                ld l,a
                jr nc,addhla1
                inc h
addhla1         MEND

                MACRO swap $reg1 $reg2
                push $reg1
                push $reg2
                pop $reg1
                pop $reg2
                MEND

                MACRO mulhl128
                rr h:ld h,l
                rr h:ld l,0
                rr l
                MEND

                MACRO divhl128
                rl l:ld l,h
                rl l:ld h,0
                rl h
                MEND
```

```
                MACRO data $num $a
LET             counter=0
REPEAT
                text "$a"
LET             counter=counter+1
UNTIL           counter=$num
                MEND

start           ld hl,&1234
                ld bc,&5
                ld a,&6
                addhla
                swap hl bc
                mulhl128
                divhl128
                data 3  12345678
                data 10 01010101
                end
```

Macros MUST be defined before they are referenced, otherwise macro calls will be considered as references to addresses. If you have a number of macros it might be useful to save them in a separate file and read them in when required.

Macro names

Macro names and their parameters must be on a line of their own, so must the accompanying MEND - except that it can be preceded by a local label.

MLON/MLOFF

Looking at the start of the above listing you will see the command MLON. This informs the assembler to list macros when they are used. MLOFF turns macro listing off - this is the default setting. Macros are always listed (if listing is turned on) when they are defined.

Local macro labels

Labels within a macro are considered local. This means they adopt a different value each time a macro is used. They may not be accessed from outside a macro.

```
Example:        MACRO addhla
                add a,l
                ld l,a
                jr nc,addhla1
                inc h
addhla1         MEND
```

However, global labels can be accessed from within a macro but only if there is not a local label with the same name. The assembler looks for a local variable first, if one is not found it looks for a global variable.

```
Example:        MACRO length?
                ld hl,endofprog
                ld bc,startofprog
                or a:sbc hl,bc
                MEND
```

Parameters and textual substitution

Variables passed to macros simply follow the macro name (eg. swap hl bc). Within a macro references to parameters must be preceded by a "$". Each parameter is passed as a string of characters and is referred to by the argument name in the macro definition line. Whenever a macro is used, any occurrence of the name of an argument preceded by a "$" is replaced by the corresponding parameter string.

This "textual substitution" is a powerful method of passing parameters as any part of the code can be variable. The parameter can represent an operand, label, instruction or even any combination or part of these.

To pass the string 123 to macro fred you need only type: fred 123. But if your string contains spaces it must be enclosed in quotes.

In the case of the above macro DATA the variable $a is enclosed in quotes. This is because, although the string is textually substituted, when it is assembled 12345678 will be evaluated as a number unless enclosed in quotes.

THE MONITOR

MAXAM II MONITOR

The Maxam II Monitor offers the most flexible system of debugging and tracing CP/M programs available for the CPC 6128 and PCW micros. It includes a disassembler, one pass assembler, automatic or manual single stepping, evaluated conditional breakpoints, memory editor and relocator.

The monitor also supports bank switching, all the major disc functions, printer control, expression evaluation, user breakpoint routines and symbolic debugging.

There are two versions of the monitor; small and large. The small version is cut down with many of the least used functions removed. This has been included to enable larger programs to be debugged.

The front panel

On entry to the monitor you are presented with a panel display as shown in figure 1. This panel is divided into 4 windows.

```
MAXAM II MONITOR Ver 1.96   HIMEM 6033 TOP F606 QUICK BRK UBRK OFF BANK 00
   (c) 1987 Armor Ltd.      LOMEM 0100 BOT 0100 SPCHK -ON PROG << No File >>

00D0  09 C3 4B 09  C3 0F BB C3  .CK.C.;C   AF 0000 (.) - - - - - - - - -
00D8  90 02 C3 A6  02 C3 AD 02  ..C&.C-.
00E0  C3 A3 02 C3  70 01 C3 63  C#.Cp.Cc   BC 0000 01 89 7F ED 49 C3 91  ...mIC.
00E8  01 C3 04 01  C3 19 02 C3  .C..C..C   DE 0000 01 89 7F ED 49 C3 91  ...mIC.
00F0  00 02 C3 2F  02 C3 0C 02  ..C/.C..   HL 0000 01 89 7F ED 49 C3 91  ...mIC.
00F8  C3 9A 03 C3  42 02 C3 67  C...CB.Cg   IX 0000 01 89 7F ED 49 C3 91  ...mIC.
0100  02 C3 72 05  E5 D5 11 4F  .Cr.eU.O   IY 0000 01 89 7F ED 49 C3 91  ...mIC.
0108  BE D5 21 5E  01 01 05 00  >U!^....   SP F29F FF FF FF FF FF FF FF  .......
0110  ED B0 0E 03  21 F4 BD ED  m0..!t=m
0118  B0 E1 22 F5  BD FB D1 01  0a"u={Q.   PC 0100 02        .   LD   (BC),A
0120  FF 80 21 48  BE CD EF BC  ..!H>Mo<      0101 C37205    Cr.  JP   &0572
0128  01 32 00 50  59 21 42 BE  .2.PY!B>      0104 E5        e    PUSH HL

a>
```

Figure 1. The Front Panel

The top window is the status window. In this you will find various information about the computer such as Himem, Lomem and the current program being debugged. Himem is the highest address in memory your programs can use and lomem is set to the end of your programs. Memory directly following lomem is often used by the monitor when cataloguing discs and so on.

The window in the middle on the left is the memory window. Here memory can be edited in either Ascii or Hexadecimal, or you can disassemble both backwards and forwards and assemble with the one pass assembly option.

The right-hand window in the middle is the register window. Here the contents of all the registers and the memory addressed by them is displayed. There is also a disassembly from the Program Counter.

The bottom window is the main window. This is where you type in all commands and results from cataloguing discs, listing memory and so on are displayed. The height of this window can be changed to suit your requirements, and it can also be enlarged to cover the whole screen, replacing the memory and register windows.

There is also a fifth window which appears at the bottom of the main window when you define any breakpoints. Any text you have in the main window is scrolled up accordingly when the breakpoint window is created.

Finally, at the bottom of the screen there is the CP/M command line. Most error messages are displayed here. Also on PCWs you have usual access to printer operations on this line.

EDITING MEMORY

There are a number of ways you can edit memory using the monitor. The first and simplest is to press ESC or STOP in command mode. This takes you directly into the memory window at the address pointed to by the memory pointer (MP).

The large version of the monitor allows you to set the value of MP to an expression. For example, typing MP PC causes the MP to follow the program counter each time the window is updated. Or, typing MP &234 will keep MP pointing to location &234.

When you enter the memory window having pressed ESC or STOP you will be in either Hex or Ascii mode, depending on the mode you last used. You can easily switch between the two but let's assume we've entered in Ascii mode first.

Depending on the height of the main window there are a number of lines of Hexadecimal characters with the address the MP is pointing to on the middle line. This is where your cursor starts.

You can now move around inside the window using the cursor keys. If you are at the top or bottom of the window and try to move out of it, the memory displayed scrolls past. In this way we can truly think of the memory window as a window into the computer's memory.

You can also use the cursor keys in conjunction with the SHIFT or CONTROL/ALT keys. The possible combinations are shown in table I.

	◄	►	▲	▼
NORMAL	Left a char	Right a char	Up a line	Down a line
SHIFT	Far left	Far right	Scroll up	Scroll down
CONTROL/ALT	Top left	Bottom right	Up a page	Down a page

Table I Cursor editing in Hexadecimal mode

Once you have selected the area of memory you wish to edit you can simply type over the bytes already there and they will be immediately stored in memory in the currently selected bank.

You can also edit using Ascii by pressing TAB. This takes you over to the right-hand side of the window where all the cursor controls are identical but pressing any other key will store the corresponding value in memory.

To quit from the memory window simply press ESC or STOP at any time.

If you need to see a disassembly of the memory you are inspecting you press COPY. The window is then cleared and replaced with disassembly of the code with the address pointed to by the MP in the centre and your cursor on top of the first character.

This is not always the case because the monitor allows you two forms of disassembly: one intelligent and the other simple. By default you enter in the intelligent mode. What this does is to inspect the code you are disassembling and place you on the first legal instruction following the address pointed to by the MP.

There are two reasons why this will not always work:

1) You may be trying to disassemble some data and of course the disassembly will be meaningless.

2) Certain opcodes, when read backwards, can be ambiguous. For example: the opcode for CALL &3D05 is &CD &05 &3D. However, &3D is DEC A and &05 is DEC B. So, if you disassemble from half way through this instruction it will not always be correct.

Nevertheless, the intelligent disassembly usually gets it right. If there is, however, any ambiguity you can solve the problem by going into simple mode by pressing TAB. Here, the address being disassembled is shown at the top of the window from the exact address specified. Pressing TAB again takes you back to intelligent mode.

Moving around in Ascii or disassembly mode is similar to moving in Hexadecimal mode with the exception that the left and right cursor keys are not used. The keys required are shown in table II.

	◄	►	▲	▼
NORMAL	Nothing	Nothing	Up a line	Down a line
SHIFT	Nothing	Nothing	Up a line	Down a line
CONTROL/ALT	Nothing	Nothing	Up a page	Down a page

Table II Cursor editing in Ascii mode

If you are using the large version of the monitor and press any other character the line you are on is highlighted in inverse video and that character is entered on the line. From now on whatever you type will be assembled when you press RETURN.

Should you just want to edit the instruction you press RETURN on its own and can then use the cursor keys to edit the line in the usual way.

All instructions are supported by the assembler including the directives BYTE, WORD and RMEM Labels can also be defined by placing a full stop before them. Labels previously defined and variables such as Himem and Lomem can be used too.

Any errors reported when assembling a line are shown on the line below. The monitor then beeps and waits a moment to give you time to read the message before continuing.

Macros are not supported.

As previously mentioned, the assemble option is not available in the small version of the monitor, but you can disassemble both backwards and forwards.

Returning to Hexadecimal mode is done by pressing COPY again. Each time you enter the memory window using ESC/STOP or E (for edit) you will enter in the last mode that was used. Pressing ESC or STOP while in Ascii mode will also quit immediately from the memory window.

To recap on changing modes:

```
In Ascii mode:          TAB  = Intelligent/Simple
                        COPY = Change to hexadecimal mode

In Hexadecimal mode:    TAB  = Hexadecimal/Alphanumeric
                        COPY = Change to Ascii mode
```

In both case ESC or STOP quits.

EDITING THE REGISTERS

The Register window displays the contents of all the registers and the locations in memory to which they point. Also the contents of the A register is shown to the right of it as an Ascii character in brackets followed by the states of the flags. At the bottom of the window there is a disassembly from the PC, depending on the height to which you have defined your main window.

There are two ways of editing the contents of the registers. The first is by a direct command such as: BC &1234 or BC PC (where the contents of BC is set to the contents of the PC). You can also type A= 42 or A= H to set a single register. These commands are not available in the small version but the second and perhaps the easiest method is. This is to type ER (Edit Registers) in command mode, which takes you straight into the register window.

Here you can proceed in a similar fashion to editing in the memory window using the cursor keys. You can move up and down from register to register with the up and down cursor keys, or left and right to each single register using the left and right cursor keys.

If you are on the top line, moving right takes you onto the flags which can be edited singly. To do this move to the flag you wish to change and press any key. Each time you press a key, whilst the cursor is on top of one of the flags, causes that flag to alternate between being set and reset. To get back to editing the registers either move all the way left to the A register or move up or down.

If you have switched off stack checking be very careful when you change the contents of SP as you may overwrite some important memory with your stack.

Pressing ESC or STOP at any time quits and returns you to the command mode.

THE EXPRESSION EVALUATOR AND THE MONITOR

You will already have seen some examples of entering addresses in the form of an expression in the monitor - full details are given in the appendix on the expression evaluator. This can only be done when using the large version of the monitor. This is because the expression evaluator has been left out of the small version to allow you more room for your programs. Addresses are entered in the same way with the small version with the exception that the input must be either a decimal or hexadecimal number. (If hexadecimal it must be prefaced by "&".)

The expression evaluator has been included to make it as simple as possible for you to find the areas of memory you are interested in and to allow conditional breakpoints.

Most commands will take an expression or a constant as their parameters. Some may prompt you with a "&" to allow quick entry of hexadecimal numbers. However, if you wish to enter an expression in place of a number just press DEL or ◄DEL once to remove it and type in your expression. If you cannot remove the "&" then the monitor is asking for a number only. This occurs in places where it is safer to enter a number, as an incorrect expression might be fatal. One example of this is when you are asked the load address of a file. If you are unsure of the address required you can work this out in the large version using the PRINT COMMAND.

One of the main uses you will probably make of the expression evaluator is with conditional breakpoints. Using the command SC you can set a condition under which the monitor will halt execution in any of the three single step modes. Conditional breakpoint expressions can follow the form:

SC HL>42 and [B+E]/2=9 or (PC+23)<(PC-4)

Although that's a bit of a mouthful you can see how a small expression can be set to cause your programs to stop under the most complicated or unusual of conditions. And, as you know, most bugs seem to occur under unusual conditions!

If you wanted to you could take levels of indirection to ridiculous extremes in the following way:

SC (((HL)))=BC

Which means:

"Stop if the contents of the address pointed to by the contents of the address pointed to by the contents of the address pointed to by the contents of HL equals the contents of BC"

It would be a rare case when you wanted to do that, but it's there if you need it.

ERROR TRAPPING

To help you debug programs the monitor has a number of built-in error trap routines. These include moving the stack pointer out of its boundaries, calls to warm boot, automatic execution of BDOS functions (preventing bank switching during single stepping - which usually causes a crash) and the facility to enter your own error traps.

These error traps are on by default unless you have reconfigured the monitor. However, it is possible for you to disable any or all of the trap routines, but do so at your own peril!

Stack checking

The error trap you will most often need to disable is stack checking. Many programs require the stack to be moved to a specific location, if that location is outside the default stack area an error will be reported. Typing IC (Ignore stack checking) will allow you to put your stack anywhere. But before you do so always ensure that the chosen area does not overwrite any of the monitor's or CP/M's memory.

Warm boot

Most programs terminate execution with a warm boot by jumping to &0000. As mentioned in the appendix on Programming Using CP/M Plus, program execution can also be terminated by a BDOS call (function 0) or a RET to system. In all these cases the monitor will trap the warm boot and return you to command mode in the main window. The only form of program termination that is not trapped is a Chain to Program call.

Warm boots are trapped by storing &F7 (BRK) in location &0000. If you change this then warm boot trapping will no longer occur. Normally location &0000 contains &C3 (JP).

Quick calls

The system variable TOP is set to the start of the BDOS routines by default, and BOT is set to &0100. Any calls or jumps outside this area will be executed at full speed. In the case of BDOS calls this is essential as many of the BDOS functions use bank switching to access various parts of memory outside the TPA. However, if there is a routine you wish to single step outside of these boundaries you can set TOP and BOT to any location. Again beware, as it is very easy to cause a crash doing this.

Finally as an extra precaution, on start-up and reset all memory between LOMEM and HIMEM is filled with &C9 (RET) so that should your code branch to an unexpected address it will be caught, in the majority of cases, before any harm is done.

SINGLE STEPPING

Single stepping is a means by which you can follow the execution of a program step by step with the contents of the flags, registers and memory being updated after each instruction.

There are 3 modes of single stepping available in the monitor. These are:

SQ — Step quickly. This mode is similar to making a jump directly to your code. However, the monitor retains control allowing you to press ESC or STOP at any time to terminate execution. It also allows you to use a previously defined conditional breakpoint expression which will be evaluated after each instruction. If the expression is evaluated as true (non zero) then execution terminates.

SS — Step slowly. Similar to the above but the contents of the flags and registers and the next instruction to be executed are displayed and updated in the main window.

S — Single step manually. This mode gives you even more control over the execution of a program via the addition of single key commands.

The extra commands available during manual single stepping are:

SPACE — Execute the instruction shown at the program counter and stop at the next one.

TAB — Skip the next instruction.

RETURN — If the instruction to be executed is a call and RETURN is pressed it will run at full speed and return to the instruction following the call. Also if you make a jump outside the top and bottom limits set for fast calls by pressing RETURN the monitor will assume you are returning from a subroutine by jumping to some code outside this area which has its own RET.

Note: TOP and BOT are monitor variables. They are used to limit the area inside which single stepping will show every instruction. Outside these limits execution is at full speed.

The usual case is where you would enter:

```
JP clearwindow
```

in place of:

```
CALL clearwindow
RET
```

CLR/DEL	This command redraws the whole panel. It is useful when your code has overwritten part or all of the screen.
CTRL/ALT U	Updates the memory window to display the contents of memory around the program counter
CTRL/ALT W	Execute next instruction, then wait for a key press.
CTRL/ALT C	Clear the screen, execute the next instruction and then wait for a key press.
COPY	Toggles between fast and slow manual single stepping. In the slow mode all registers, flags, contents of memory and current instruction are displayed. In the fast mode only the contents of the main window are updated.

Any other key returns you to command mode. There is further information on single stepping in the section "Debugging a CP/M Plus Program".

BREAKPOINTS

Breakpoints are one of the most powerful ways of debugging a program. The monitor offers several types of breakpoints which can be used in different ways.

Hard breakpoints

First of all there are "hard" breakpoints. No matter how you are running a program, execution will always stop when a hard breakpoint is reached. These can be set and cleared using the SB and CB commands or, if in disassembly mode in the memory window, you can set and clear breakpoints by pressing CTRL or ALT-B.

If you wish to test a piece of code without breakpoints taking effect you can disable them using the IB (Ignore Breakpoints) command. This has the effect of keeping your breakpoints intact for later use. When breakpoints are ignored they will appear in inverse video at the bottom of the front panel.

Hard breakpoints can be re-enabled by typing EB.

Conditional breakpoints

Conditional breakpoints are only available in the large version of the monitor. They are set using the SC (Set Condition) command. Conditional breakpoints are only tested for in the single stepping modes. This is done by emulating each Z80 instruction then evaluating the condition and stopping if it is true.

User conditional breakpoints

User conditional breakpoints are most useful in the small version of the monitor which does not have normal conditional breakpoints. These work by setting a monitor variable to point to the start of a user routine you have written which can be as long as necessary. Your routine then does all the required checking and should return at the end with the carry flag clear if execution is to stop, or with it set if execution is to continue.

A user conditional breakpoint routine is passed the contents of all registers and flags, including the alternate set. The registers can be corrupted by the routine but the carry flag must return the correct condition.

To initialise a user routine enter UB <address> where address is the entry address. This informs the monitor of the routine's presence. To use the routine you then have to enable it by typing EU (Enable User). The routine can be disabled by the IU (Ignore User) command.

JUMP and RESUME

It must be stressed that confusing JUMP and RESUME can cause major problems. This is because JUMP always puts a return address on the stack so that control can return to the monitor. If you have jumped to some code and hit a breakpoint the return address will still be on the stack. In this case you must use RESUME in future to ensure that another return address is not pushed on the stack.

The only time RESUME puts a return address on the stack is if it is empty. This is to ensure that control returns to the monitor on program termination.

RR (Repeat RESUME) is useful when you have put a hard breakpoint inside a loop. Quite often you will wish this loop to be executed a number of times until you have reached the correct point in a program. To save having to type R RETURN several times you can enter RR <number>. Execution will then resume the specified number of times only stopping at a breakpoint after all resumes have occurred.

If you set a breakpoint which will only be used once, when you reach it you can clear the breakpoint and resume execution using one command, RC (Resume and Clear breakpoint).

MONITOR CONFIGURATION

The monitor can be configured to your individual requirements. The way to do this is to simply set up any of the options below to the states or values required in the normal way. Then type SF to save the configuration file. Thereafter, each time you run the monitor, it will use the file to re-configure itself - assuming the configuration file is in a drive at start-up.

If no configuration file is present at start up, the settings below will assume default values.

1.	Length of symbol table - set by the DSL command	
2.	Current editing address in memory window	
3.	Memory pointer expression	
4.	Bank selected	
5.	BOT	
6.	TOP	
7.	All flags, registers and alternate registers	
8.	Window size	
9.	Exec filename - set by the DX command	Filename/Ignored
10.	Edit mode	Ascii/Hexadecimal
11.	Disassembly mode in memory window	Normal/Intelligent
12.	Manual single stepping	Fast/Slow
13.	Breakpoints	Enabled/Ignored
14.	Wait at breakpoint	On/Off
15.	Stack checking	On/Off
16.	AF register pair	Standard/Alternate
17.	BC, DE and HL register pairs	Standard/Alternate

You also have the option to save breakpoints and the filename of the program you are currently debugging. To do this you type QS rather than Q when you quit. Then, the next time you use the monitor, your breakpoints and the file being debugged will be automatically loaded in at start-up.

If you type Q to quit then all breakpoints and your current program's filename are wiped from the disc.

Neither Q nor QS affect the configuration file.

DEBUGGING A CP/M PLUS PROGRAM

At the beginning of the manual we had a brief look at writing and debugging a simple CP/M Plus program. The example was necessarily simple as it was used as an overview of the program development process. In this section we shall take an in-depth look at the facilities offered by the monitor.

It is recommended that you type in each of the examples and try them out for yourself to gain a clearer insight into the monitor's workings. The line numbers shown in the listings should not be entered. They are there as a guide to explaining the programs.

A simple program

```
 1 PRINT "Program I"
 2
 3                ORG &0100
 4 RESET          equ &0000
 5 BDOS           equ &0005
 6 PRINTSTRING    equ &0009
 7 READCONSOLE    equ &000a
 8
 9 start          ld c,printstring        ; Print string function
10                ld de,m.name            ; String to print. Must terminate
11                                        ; with "$"
12                call bdos               ; Call BDOS
13
14                ld a,20                 ; Maximum length of input
15                ld (buffer),a           ; Poke it into (buffer+0)
16                sub a                   ; Set A to zero; Characters already
17                                        ; in the buffer
18                ld (buffer+1),a         ; Poke it into (buffer+1)
19                ld c,readconsole        ; Read console function
20                ld de,buffer            ; Input buffer start
21                call bdos               ; Call BDOS
22
23                ld a,(buffer+1)         ; Find the length of the input
24                and a                   ; Was it zero?
25                jr z,start              ; Yes, try again
26
27                ld hl,buffer+2          ; No, find start of string
28                ld c,a:ld b,0           ; Put length in BC
29                add hl,bc               ; Add offset to get the end
30                ld (hl),"$"             ; Poke the end with "$" to
```

```
31                                      ; provide a terminator
32
33              ld c,printstring        ; Print string function
34              ld de,m.hello           ; String to print
35              call bdos               ; Call BDOS
36
37              ld c,printstring        ; Print string function
38              ld de,buffer+2          ; Re-print user's input
39              call bdos               ; Call BDOS
40
41              jp reset                ; Jump to warm boot
42
43 m.name       text 13,10,"What is your name? $" ; First message
44 m.hello      text 13,10,"Hello $"              ; Second message
45 buffer       rmem 22                           ; User's input buffer
46
47              END                     ; End of program
```

Program 1 takes an input from the user and prints a message. The first line is a command to the assembler to print "Program 1" on the screen. This is usually helpful when you write a program in several parts which are read in one at a time during assembly.

Lines 4-7 set labels to certain values. This is not necessary but it makes programs more legible for other people to read.

The label 'start' on line 9 is there as we may need to jump back there later in the program.

Lines 9-12 print out the message following 'm.name' (line 43). Note that you must always end a string which is to be printed using BDOS function 9 (print string) with a "$" to mark the end of the string.

In lines 14-21 we meet BDOS function &0a (read console). This is an example of a function which requires more than one parameter to be passed to it. Firstly DE must point to a buffer in which the input will be stored and secondly, before you call BDOS, you must set the first byte of this buffer to the maximum length of input allowed and the second byte to the number of characters already input. In most cases this will be zero. Entering a non zero number will have the effect of returning any characters already in the buffer to the input line as if they had been typed into the keyboard. The string entered in the buffer is stored from the third byte onwards.

On return from the call the second byte of the buffer contains the new length of the string. This is checked in lines 23-24. If the length is zero then we return to 'start' at line 25.

If a valid string was entered then in lines 28-29 we take the length of the string and add it to the start to find the end. Immediately following the end we put a "$" as the end-of-string identifier.

Having done this lines 33-35 print out the string following 'm.hello' (line 44) and lines 37-39 reprint the user's input.

Finally line 41 jumps to location &0000 which causes a warm boot.

This is a fairly straight-forward program and (which we hope) has no bugs in it. We will be using it as the basis for the next examples so you should save it now as "PROG1" then type:

```
ASM PROG1
```

If all is well the program will successfully have assembled. Try it out by typing:

```
*PROG1
```

The message "What is your name ?" should appear on the screen. You can now type in up to twenty characters. Try this by entering your name. The message "Hello <your name>" should now appear on the screen followed by the CP/M prompt below it. If this does not happen go back and check that you typed the program in exactly as it is printed (ignoring the line numbers) and re-assemble it.

Re-enter the editor by typing:

```
APED
```

Improving on Program 1

Assuming you now have a working program we'll go on to look at how we can on improve it. The first thing that comes to mind is that if you are using a large number of BDOS calls in a program a lot of memory will be used up simply setting up the parameters. Also, poking a "$" at the end of each string to be printed is wasteful on memory.

Another thing to note with BDOS is that, apart from any registers which return parameters, all other flags and registers should be assumed to be corrupt on return from a BDOS call. That's not a problem with Program I but larger programs may wish to keep register values. To solve these problems it's a simple matter to modify Program I and at the same time make the main part of the program much shorter.

```
 1 PRINT "Program II"
 2
 3              ORG &0100
 4 RESET        equ &0000
 5 BDOS         equ &0005
 6 PRINTSTRING  equ &0009
 7 READCONSOLE  equ &000a
 8
 9 start        ld hl,m.name              ; String to print
10              call print
11
12              ld a,20                   ; Maximum length of input
13              ld hl,buffer              ; Input buffer
14              call input                ; Get input
15
16              ld a,(buffer+1)           ; Find the length of the input
17              and a                     ; Was it zero?
18              jr z,start                ; Yes, try again
19
20              ld hl,buffer+1            ; No, find start of string
21              ld c,a:ld b,0             ; Put length in BC
22              add hl,bc                 ; Add offset to get the end
23              set 7,(hl)                ; Set the top bit of the last char
24              ld hl,m.hello             ; String to print
25              call print                ; Print it
26              ld hl,buffer+2            ; User's input
27              call print                ; Print it
28
29              jp reset                  ; Jump to warm boot
30
31 .print       push af:push bc:push de ; Push all flags and registers
32              push ix:push iy:push hl ; on the stack
33
34              ld bc,tempbuf             ; Get ready to copy string
35
36 print1       ld a,(hl):ld (bc),a       ; Copy from string to tempbuf
37              bit 7,a                   ; Is the top bit set?
```

```
38              jr nz,print2              ; Yes, carry on
39              inc hl:inc bc:jr print1   ; No, copy next character
40
41 print2       inc bc:ld a,"$"           ; Get terminator
42              ld (bc),a                 ; Poke terminator at end of string
43              ld c,printstring          ; Print string function
44              ld de,tempbuf             ; Get start of tempbuf
45              call bdos                 ; Call BDOS
46
47              pop hl:pop iy:pop ix      ; Pop all flags and registers
48              pop de:pop bc:pop af      ; off the stack
49              ret                       ; return
50
51 input        push af:push bc:push de   ; Push all flags and registers
52              push ix:push iy:push hl   ; on the stack
53
54              ld (hl),a:inc hl          ; Put max length at start of buffer
55              ld (hl),0:dec hl          ; Put start length next
56              ld c,readconsole          ; Read console function
57              ex de,hl                  ; Put start of buffer in DE
58              call bdos                 ; Call BDOS
59
60              pop hl:pop iy:pop ix      ; Pop all flags and registers
61              pop de:pop bc:pop af      ; off the stack
62              ret                       ; Return
63
64 m.name       byte 13,10
65              str  "What is your name ?"      ; First message
66 m.hello      byte 13,10
67              str  "Hello "                   ; Second message
68
69 buffer       rmem 22                         ; User's input buffer
70 tempbuf      rmem &100                       ; Temporary buffer
71
72              END                             ; End of program
```

If you take a look at Program II you'll see that the main program only takes up lines 9-29 (compared with 9-41 in Program I). Perhaps the first thing you'll notice is the change of parameters required in line 9. Here we simply load HL with the start of the string to be printed and call 'print'. Looking at 'print' (line 31) you'll see that it is preceded by a ".". This is because print is an assembler directive, but because its a sensible word to use in this case it has been turned into a label by adding the ".".

At line 31 the first thing we do is push all the flags and registers on the stack so that they will be intact when we return from the subroutine. Then lines 36-39 copy the string to be printed into a special buffer called 'tempbuf' (line 70). We do this so that we can then add the "$" to the end of the string in the buffer, saving us from having to place it at the end of every string in memory (a saving of many bytes). This also allows us to process an output string in any way we want.

The way we determine the end of the string is by checking whether the top bit of a byte is set. In the case of the string being printed from line 9 ('m.name') the top bit of the last character has been set by using the command STR in line 67. STR simply places the string following it in memory at the program counter and sets the top bit of the last character.

So, having copied the string into 'tempbuf' and found the end of it, lines 41-42 place a "$" after the last character. Finally lines 43-45 load register C with the required BDOS function number, load DE with the start of the string and call BDOS. To return from the routine lines 47-48 pop the stored contents of the flags and registers off the stack and line 49 returns.

We're now back at line 12. Here we wish to input a string and again we use a subroutine. First we load register A with the maximum length of the input string (this saves us having to poke it anywhere in memory) then we set HL to the start of the input buffer and call 'input' in line 14.

'Input' is similar to 'print' in that it first stores the contents of the flags and registers. It then stores the maximum length of the input string at the start of the buffer and stores the number of characters already in the string (zero) in the second location in the buffer. Having done this it sets itself up for the BDOS call, calls it, restores the registers and returns.

It's now a simple matter at lines 16-18 to check whether a string was entered and jump to 'start' if one wasn't.

Then lines 20-22 find the end of the string in the input buffer and line 23 sets its top bit. Lines 24-25 now print the word "Hello " and lines 26-27 reprint the user's input. Finally line 29 returns to the CP/M command mode via a warm start.

A good exercise to help you practise the editor's facilities would be to adapt Program I turning it into Program II, or you may prefer to type the whole program in. Either way, (again having ignored the line numbers) save Program II as PROG2 and assemble it.

When you run it you'll find it works in exactly the same way as Program I except that the last character of each string is incorrect. Oh dear, a bug. However, it looks fairly simple to fix. If you remember we were setting the top bit of the last character of each string to be printed, so the problem is probably something to do with this. Looking at the code it may not be obvious exactly where the error lies but it is probably somewhere between lines 36-44.

Single stepping Program II

To debug Program II go into the large version of the monitor by typing:

```
MM PROG2
```

This loads in the monitor which then automatically loads in PROG2.COM for you.

Before single stepping we'll set up a few functions. First type:

```
ASC
```

to change the display in the memory window to disassembly format. Now type:

```
MP PC
```

This causes the memory window to always show the memory around the program counter. And now, because we are fairly sure the problem is to do with setting the top bit of a byte, type:

```
SC A>&7F
```

This sets a conditional breakpoint which tells the monitor that if the contents of register A gets higher than &7F (its top bit gets set) to stop.

Finally, to get quickly to the problem, type:

```
SS
```

This means single step slowly. You will see lines of information scrolling by in the main window. These show the contents of the flags, registers and the current instruction being executed. After a few seconds the monitor should beep and the next instruction to be executed, shown next to PC in the register window, should be LD (BC),A.

Looking in the memory window you'll see that the previous instruction was LD A,(HL) from line 36 in Program II. Also, looking next to the register pair AF in the register window you'll notice that the contents of A (shown in brackets) is a question mark with the top bit set. So, we've found the last character in the string: "What is your name ?".

Right, let's follow execution through until the string is printed on the screen. Press ESC or STOP and type:

S

This means single step manually. If the word at the top right of the main window (in inverse video) is "FAST" press COPY once. During manual single stepping the COPY key alternates between showing only the summary of information in the main window and complete updates of all the windows. As you might think, the slow mode is a lot slower but it provides you with much more information.

To single step press the space bar 8 times until CALL &0005 appears next to PC in the register window. This is the BDOS call where the string will be printed. Press ALT or CTRL C (This means clear screen, execute next instruction then wait for a key press) and you should see the string printed on the screen with the question mark replaced by the TM (Trademark) symbol. Now press any key, then ESC or STOP.

This is where the problem lies. Let's re-trace the above procedure. To do this press ESC or STOP then type:

IN

This stands for initialise, it resets the program counter to &0100 and sets all the registers to their default values. Then go into slow single stepping. We should again reach the instruction LD (BC),A. As the instruction prior to this loads register A with the contents of the location pointed to by HL it means we've copied everything into 'tempbuf', including the top bit set in the last character. Problem solved; before copying characters into 'tempbuf' we must ensure that their top bits are not set.

Having discovered the problem, press ESC or STOP, type:

Q

to quit from the monitor and return to the editor by typing:

APED

All we need to do now is adapt the routine 'print1' at line 36.

The simplest way to do this is shown in Program IIa which shows the replacement lines required.

```
                ; Program IIa

36 print1       bit 7,(hl):push af      ; Is the top bit set? Store result
37              ld a,(hl):and &7f       ; Clear top bit if set
38              ld (bc),a               ; Copy from string to tempbuf
39              pop af:jr nz,print2     ; Restore result. Yes, carry on
40              inc hl:inc bc:jr print1 ; No, copy next character
```

Line 36 checks the top bit of the character in the buffer pointed to by HL BEFORE copying the character, and then pushes the result of the check on the stack. Line 37 then loads the character into A and ANDs it with &7f (binary %01111111). This has the effect of keeping any set bits between 0-6 and masking out the top bit if set.

Then line 38 copies the character with the unset top bit into the buffer and line 39 pops the result of the check off the stack. It is only at this point that we should check if the top bit had been set. If so, line 39 jumps to 'print2', otherwise line 40 increments the two pointers and jumps back to 'print1' where the process repeats.

This is a genuine bug that crept in while writing the example. The debugging procedure followed is the one that found the bug!

Turning program II into a Password routine

Before going any further correct the routine 'print1' in program II and save the result as PROG2A. Try running it to make sure it works. We will be using the print and input routines in the next example.

```
 1 NOLIST
 2
 3 PRINT "Program III"
 4
 5              ORG &0100
 6 RESET        equ &0000
 7 BDOS         equ &0005
 8 PRINTSTRING  equ &0009
 9 READCONSOLE  equ &000a
10
11 start        ld hl,m.prompt          ; String to print
12              call print
13
14              ld a,20                 ; Maximum length of input
15              ld hl,buffer            ; Input buffer
16              call input              ; Get input
17
18              ld a,(buffer+1)         ; Find the length of the input
19              and a                   ; Was it zero?
20              jr z,start              ; Yes, try again
21
22              ld de,buffer+1          ; No, find start of string
23              ld b,7                  ; Put length of password in B
24              ld hl,password          ; Get start of password
25
26 passloop     ld a,(de):set 5,a       ; Convert to upper case
27              cp (hl)                 ; Compare with password
28              jr nz,wrong             ; Same? No, goto wrong
29              djnz passloop           ; Yes, check next
30
31              ld hl,m.welcome         ; Correct input so
32              call print              ; print welcome message
33
34              ; The rest of your program goes here
35              ; .....
37
38              jp reset
39
40 wrong        ld hl,m.invalid         ; Incorrect password so
41              call print              ; tell the user and
42              jp reset                ; jump to warm boot
43
44 .print       push af:push bc:push de ; Push all flags and registers
45              push ix:push iy:push hl ; on the stack
46.
47              ld bc,tempbuf           ; Get ready to copy string
```

```
48
49 print1        bit 7,(hl):push af       ; Is the top bit set? Store result
50              ld a,(hl):and &7f        ; Clear top bit if set
51              ld (bc),a                ; Copy from string to tempbuf
52              pop af:jr nz,print2      ; Restore result. Yes, carry on
53              inc hl:inc bc:jr print1  ; No, copy next character
54
55 print2        inc bc:ld a,"$"          ; Get terminator
56              ld (bc),a                ; Poke terminator at end of string
57              ld c,printstring         ; Print string function
58              ld de,tempbuf            ; Get start of tempbuf
59              call bdos                ; Call BDOS
60
61              pop hl:pop iy:pop ix     ; Pop all flags and registers
62              pop de:pop bc:pop af     ; off the stack
63              ret                      ; return
64
65 input        push af:push bc:push de ; Push all flags and registers
66              push ix:push iy:push hl ; on the stack
67
68              ld (hl),a:inc hl         ; Put max length at start of buffer
69              ld (hl),0:dec hl         ; Put start length next
70              ld c,readconsole         ; Read console function
71              ex de,hl                 ; Put start of buffer in DE
72              call bdos                ; Call BDOS
73
74              pop hl:pop iy:pop ix     ; Pop all flags and registers
75              pop de:pop bc:pop af     ; off the stack
76              ret                      ; Return
77
78 m.prompt     byte 13,10
79              str  "Please enter password ?"  ; First message
80 m.invalid    byte 13,10
81              str  "Invalid password"         ; Incorrect input message
82 m.welcome    byte 13,10
83              str  "Welcome to MegaProg etc.."; Welcome message
84
85 password     str  "TOLKIEN"                  ; Password
87
88 buffer       rmem 22                         ; User's input buffer
89 tempbuf      rmem &100                       ; Temporary buffer
90
91              END                             ; End of program
```

Have a look at program III. The main differences between it and Program II are the new routine 'passloop' starting at line 26, the new strings to be printed from line 78 and the password in line 85. Also note the NOLIST at line 1. This is included because the program is getting fairly long and the assembly process will take quite a bit longer if everything is printed to the screen. Although listing is turned off any errors will still be reported.

Quickly going through 'passloop', starting just before it at line 22, DE is set to point to the start of the input string, register B is loaded with the length of the password and HL is set to point to the start of the password.

The password checking loop starts at line 26 where A is loaded with each character in turn from the input string and then has bit 5 set. This has the effect of converting the contents of A to upper case. This is done so that the case used in the input is not important.

At line 27 we compare the result with the current character in the password pointed to by HL. If it's not the same we branch to 'wrong' which displays an error message and quits.

But if the character matches we continue the process until all characters have been compared. If the input string matches successfully line 31 loads HL with the start of a welcome message and line 32 prints it. After this you can enter the rest of your code, but in this case we simply return to CP/M command mode.

As before, type in the program (or adapt PROG2A), save it as "PROG3", assemble it and run it.

You will find that no matter what you type in reply to the prompt you always get the message "Invalid password". So we have another bug.

Enter the large version of the monitor by typing:

```
MM PROG3
```

Again we'll set up the monitor ready for single stepping. Press ESC or STOP once. This takes you into edit mode in the memory window. Now press COPY, this switches to an Ascii disassembly. Now press the down cursor key until the instruction under the cursor is LD A,(DE). To the left of the cursor you will see that you are at location &011C. At this point press CTRL or ALT B. This sets a breakpoint automatically at address &011C.

What we've done is set a hard breakpoint at which the monitor will always stop. The breakpoint address is the start of the routine 'passloop', the most likely source of the bug. Press ESC or STOP again to take you back into the main window and type:

```
J
```

This means jump to the code pointed to by the program counter, which in this case is the start of the program (&0100). Enter the correct password "tolkien" and press return in reply to the prompt.

The monitor's panel should return. Look at the characters on the far right hand side of the register window next to the DE and HL register lines. You'll see that DE is pointing one character before the start of the input string. Going back to the listing to see where we loaded DE with the input line start you'll see that at line 22 we loaded DE with buffer+1. This must be wrong because that location contains the length of the string, and going back to the register window, next to DE is the number 7, which is the length of the string.

To fix the problem type:

```
DE DE+1
```

This increments DE. You should see that the memory shown next to DE and HL now matches up.

To continue type:

```
S
```

to single step. Press the space bar twice. This loaded A with the first character of the input string (t) and set bit 5. But looking next to the AF register pair in the register window the character in A is still lower case. Of course, we should have RESET bit 5 because the Ascii value of "t" is 116 and the value of "T" is 84 - a difference of 32 - which means masking OUT not SETTING is how we convert to upper case.

To fix this go back into the memory window by pressing ESC or STOP twice (once to quit from single stepping) and move upwards a line with the up cursor key. The instruction shown should be SET 5,A. Press RETURN. The whole line is now highlighted in inverse video. Press CLR or DEL► 3 times to remove the SET and type:

```
RES
```

Then press return. You have now re-assembled the line correctly. Now to try again. Press ESC or STOP, type:

```
IN
```

to re-initialise the registers and type:

```
J
```

to jump to the start of the program. Again answer the prompt and the monitor returns to the main panel. Remembering that DE is pointing to the wrong address increment it again in the same way as before. Now go into single stepping and press space a few times.

You will notice that the character shown next to AF in the register window alternates between a "t" and "T". So we are correctly converting the case of the character but somehow we're not moving onto the next character after each comparison. That's a simple one. We forgot to increment HL and DE at line 29 before repeating the process.

That's 3 bugs so far, maybe a good point at which to update the source code as it looks likely that we've found everything. Go back into the editor and change the following lines:

```
22              ld de,buffer+2

26 passloop     ld a,(de):res 5,a

29              inc de:inc hl
30              djnz passloop
```

Re-save the program as "PROG3", assemble it and run it.

Again when you type "tolkien" you get the message "Invalid password". Oh well, another bug. Go back into the monitor and follow the above process, setting the breakpoint and jumping so that execution stops at the instruction LD A,(DE). Go into single stepping and press space a few times. You'll see that each character in the input string is being converted to upper case, checked and matched successfully with the password. Until, that is, we reach the final "n". When we match it with the password it fails. Comparing the two bytes in the register window we see that in the password "n" has its top bit set. Its our old problem again. But how did it creep in this time?

Well, looking at where the password is defined in line 85 we find that we used the command STR and that caused the top bit of the final character to be set. The problem is now obvious, we should replace the STR with TEXT. Try it and re-assemble Program III.

Success, it works!

Program III was an example of how a number of bugs can occur causing similar problems. Using the monitor however it is easy to trace program execution and spot exactly where a bug occurs, and in most cases correct the bug from within the monitor. This allows you to test for further bugs before returning to the editor.

9 WAYS TO CRASH THE MONITOR!

This may seem like a strange title but there are many ways in which you can crash the monitor and it is well to be aware of them.

1. FILL. Perhaps the easiest, and most obvious, is FILL. The fill routine does no checking so if you overwrite any important memory the monitor will crash. The same goes for MOVE and RELOCATE.

2. BLOAD loads a file into the currently selected bank. If that bank contains important information the monitor will crash.

3. Quick calls. If you jump to a routine above TOP or below BOT which does not return you will be lucky to get control back as the code will be running at full speed with no checks being made.

4. EDIT. Be very careful which areas of memory you edit.

5. Breakpoints. If you set a breakpoint halfway through an instruction there may be unexpected results.

6. User breakpoints. Make sure any user conditional breakpoint routines are fully tested before using them.

7. JUMP. Don't jump when you mean resume as this puts a return address on the stack. The monitor may not crash but strange things will happen to your code.

8. JUMP. Be sure where you're jumping to. By default it will always be the program counter.

9. If you change system pointers such as HIMEM by changing location &0006 be sure that you leave any chain of jumps undisturbed. HIMEM is in fact contained in the 2 bytes following the BDOS jump location at &0005. If this is changed the new value must point to a jump which jumps to the previous location pointed at by &0006. Also, if you change system pointers, ensure that the new jump is set up first.

MONITOR COMMANDS

In this section the monitor's commands are divided into groups. Many of these commands can be abbreviated. Where this is the case the abbreviation follows the command name. The commands are shown in alphabetical order within each section so that they can be quickly found. Parameters may be entered in line, if they are omitted and a default value is not assumed you will be prompted. See Appendix IV for commands which assume default values.

Inline parameters must be separated by a comma and expressions are allowed in most cases in the large version.

There are two versions of the monitor on the disc, large and a small, many of the commands are only available in the large version and these are shown followed by a *.

MEMORY COMMANDS

ASC - Sets the display in the memory window to Ascii format so that disassembly will be displayed.

ASSOCIATED COMMANDS: EDIT, HEX, MDIS, MEDIT, MP

BANK * - This allows you to select a bank configuration. On the CPC 6128 you can select bank 0, 1 or 2. On a PCW you can select further bank configurations up to the capacity of memory available. These extra banks resemble the TPA with the bottom 16k being replaced by a 16k block from the Ram disc. selecting bank 3 places block 9 at the bottom of the TPA. On the PCW 8256 you can select up to bank 9 (block 15). On the PCW 8512 you can have up to 25 bank configurations.

After a bank has been selected all further memory operations such as edit or disassemble will take place in this bank.

ASSOCIATED COMMANDS: BLOAD, BSAVE

BLOAD * **BL** Load a file into Ram in the currently selected bank.

ASSOCIATED COMMANDS: BANK, BSAVE

BSAVE * **BS** Save a file to disc from the currently selected bank.

ASSOCIATED COMMANDS: BANK, BLOAD

COMP * **CP** Compares two blocks of memory with each other byte by byte displaying any differences.

ASSOCIATED COMMANDS: FC

DIS **D** Disassembles memory in the main window from the given address.

ASSOCIATED COMMANDS: ASC, FD, MDIS, OD

DUMP * **DM** Dump a file to screen in Hexadecimal format.

ASSOCIATED COMMANDS: TYPE

EDIT **E** Puts you directly into Edit mode in the memory window, using the default set by the last HEX or ASC command. Pressing ESC or STOP on its own in command mode has the same effect. However, EDIT allows you to specify the address at which to edit. See "Editing Memory".

ASSOCIATED COMMANDS: ASCII, HEX, MDIS, MEDIT, MP

FC * - Compare a file with contents of memory.

ASSOCIATED COMMANDS: COMP

FD * - Disassemble a file from disc using the given start address.

ASSOCIATED COMMANDS: DIS

FILL * - Fills an area of memory with a specified byte.

ASSOCIATED COMMANDS: MMOVE, RELOC

FIND F This command allows you to find a sequence of up to 8 bytes of code or a string of up to 20 characters or assemble a block of code to match. This block is entered an instruction at a time, when RETURN is pressed on its own the search is initiated. The byte or code find will both accept wildcards using "?". The code option is only available in the large version.

ASSOCIATED COMMANDS: (None)

HEX - Sets the display in the memory window to Hexadecimal.

ASSOCIATED COMMANDS: ASC, EDIT, MDIS, MEDIT, MP

LIST L Lists memory in the main window in Hexadecimal between two given addresses.

ASSOCIATED COMMANDS: DM

LOAD - Load a file into the TPA at the given address. If no file extension is given then .COM is assumed.

ASSOCIATED COMMANDS: SAVE

LS * - Load a symbol table. The symbol table will have been created using the assembler SYM directive.

ASSOCIATED COMMANDS: PS

MDIS * **MD** Takes you into Edit mode in the main window allowing disassembly and one pass assembly. The small version of the monitor allows full disassembly, but the one pass assembly option is not included. See "Editing Memory".

ASSOCIATED COMMANDS: ASC, EDIT, HEX, MEDIT, MP

MEDIT **ME** Takes you into Edit mode in the main window allowing Hexadecimal or Ascii editing byte by byte. See "Editing Memory".

ASSOCIATED COMMANDS: ASC, EDIT, HEX, MDIS, MP

MMOVE * **MM** Moves a block of memory of a given length from one address to another. The blocks may be overlapping.
ASSOCIATED COMMANDS: FILL, RELOC

MP - MP stands for Memory Pointer. The memory pointer is the address which is used for display and editing in the memory window.

ASSOCIATED COMMANDS: ASC, EDIT, HEX, MDIS, MEDIT

OD - Exactly the same as DIS except that you can specify an offset from the start of disassembly. This allows you to see what the code would look like if it were relocated. If no start address is given then &0100 is assumed.

ASSOCIATED COMMANDS: DIS, FD

PRINT * **?** Evaluates an expression allowing nested parentheses, register names, system variables, and indirection.

PRINT (HL) prints the address pointed to by the contents of HL.

PRINT [HL+BC]/2 prints the result of adding the contents of HL to the contents of BC and then dividing by 2

This evaluation is integer only and the result is printed in both Hexadecimal and Decimal.

ASSOCIATED COMMANDS: (None)

PS * - Print Symbols. Displays any symbols you have created or loaded.

ASSOCIATED COMMANDS: LS

RELOC * REL This is similar to MMOVE except that jump and call addresses can be relocated to point to the correct places. You are given an option between a simple and an intelligent relocation. The simple option relocates the whole block, whereas the intelligent option allows you to specify areas of data which are not to be relocated.

Note: Intelligent relocation does not move code but relocates it on the spot.

ASSOCIATED COMMANDS: FILL, MMOVE

SAVE - Save a file from the TPA to disc. No extensions are assumed.

ASSOCIATED COMMANDS: LOAD

TAIL - Store the specified command tail in memory at &80. This is useful for testing programs that obtain parameters from the CP/M command line.

RUNNING AND DEBUGGING COMMANDS

BOT - This is the bottom range allowed when you make fast calls while single stepping. It is included so that if you are stepping through a program and do not wish to follow certain subroutines below a certain address, (perhaps because they are executed many times) they will be executed at full speed if they are below BOT.

Initially BOT is set to &0100 as this is the normal start of a program.

ASSOCIATED COMMANDS: TOP

CB - Clears breakpoints. A number of addresses can be entered following the command, or if no address is specified, the cursor moves to the breakpoint window (if any breakpoints have been set). At this point you can select the breakpoint to clear using the left and right cursor keys and then clear that breakpoint by pressing RETURN.

ASSOCIATED COMMANDS: RC, SB

CLS - Clears the main window.

ASSOCIATED COMMANDS: (None)

EB - If you have entered any breakpoints this command will enable them. By default, breakpoints are enabled but may be ignored using the command IB. This is useful when you want to test your code without breakpoints but may wish to use them again later.

ASSOCIATED COMMANDS: IB

EC - Enable stack checking. During single stepping, if the stack pointer is changed to lie outside the allocated area, and stack checking is enabled execution stops and an error message is displayed.

ASSOCIATED COMMANDS: IC

ER - Puts you into Edit mode in the register window where you can change the contents of all registers, the PC and flags.

ASSOCIATED COMMANDS: (None)

EU - Enables a user breakpoint routine if one has been initialised. See UB for how to initialise one.

ASSOCIATED COMMANDS: IU, UB

EXX - Exchanges BC, DE and HL with their alternate register pairs.

ASSOCIATED COMMANDS: XAF

IB - Once this command has been issued, any hard breakpoints will be ignored. Conditional breakpoints and user conditional breakpoints will still work.

ASSOCIATED COMMANDS: EB

IC - Ignore stack checking. Normally your stack is located in a &0100 byte block in common memory above &C000. If your stack moves out of this area, execution will terminate and an error will be reported. If for any reason you wish to move your stack elsewhere you must use this command so that execution can continue.

If you use this command and your stack overwrites part of the monitor's or CP/M Plus's memory then the system will probably crash! You have been warned.

ASSOCIATED COMMANDS: EC

INIT **IN** Initialises all registers and alternate registers to 0. Also resets the PC to &0100, SP to its default value and MP to &0100.

ASSOCIATED COMMANDS: RESET, ZR

IU - If a user breakpoint routine has been initialised this command causes it to be ignored.

ASSOCIATED COMMANDS: EU, UB

JUMP **J** Jumps to code at the given address. If no address is specified, execution will commence at the program counter. This command places a return address on the stack to allow control to pass back to the monitor on termination.

NOTE: To avoid corrupting the stack by placing return addresses on it, this command should only be issued on commencement of execution. Thereafter you should use RESUME.

ASSOCIATED COMMANDS: RESUME, RC, RR

QB - Quick breakpoints. If a breakpoint is met whilst code is being executed from a JUMP or RESUME, control will return immediately to the monitor in command mode.

ASSOCIATED COMMANDS: WB

RC - Resume and clear breakpoint at the PC. If your code has stopped execution at a breakpoint and you no longer have need of that breakpoint, this command removes it and continues execution.

ASSOCIATED COMMANDS: CB, JUMP, RESUME, SB

RESUME R — Similar to JUMP except that no return address is placed on the stack unless it is empty. In this case it is necessary to do so to allow control to pass to the monitor on completion.

ASSOCIATED COMMANDS: JUMP, RC, RR

RR - Repeat RESUME. Causes a RESUME to be executed a specified number of times. If no number is specified RR does one RESUME only. RR is particularly useful when debugging code with a breakpoint inside a loop.

ASSOCIATED COMMANDS: JUMP, RC, RESUME

S - Single step. This takes you into manual single stepping in which you retain more control over your program's execution with the ability to skip instructions, make fast calls - which are not stepped - to routines which you know work, update the contents of all the windows in the panel and select between fast and slow running. See "Single Stepping".

ASSOCIATED COMMANDS: SQ, SS

SB - Set a breakpoint. You can enter a number of addresses following this command. If no address is specified you will be prompted for one.

ASSOCIATED COMMANDS: CB, RC

SC * - Set a conditional breakpoint expression. This command enables you to specify a condition under which your code is to stop execution without having to set an address. Parentheses and indirection plus full integer arithmetic are supported.

This facility is only available during single stepping.

ASSOCIATED COMMANDS: (None)

SQ - Step quickly. Similar to RESUME except that the monitor retains control throughout allowing it to keep track of the stack, user breakpoint routines, and conditional breakpoints. Execution speed is reduced due to the amount of work the monitor has to do, but there is less chance of a crash and you can terminate execution at any time by pressing ESC or STOP. See "Single Stepping".

ASSOCIATED COMMANDS: SS, S

SS - Step slowly. Identical to SQ except the contents of the registers, stack, program counter and current instruction are continuously displayed and updated in the main window. See "Single Stepping".

ASSOCIATED COMMANDS: SQ, S

TOP - Is similar to BOT, except that calls made above TOP will execute at full speed.

ASSOCIATED COMMANDS: BOT

UB - Although the monitor is supplied with a sophisticated

expression evaluator for setting conditional breakpoints there may be times when you wish program execution to stop under complicated conditions. For example you may wish to check whether any memory between &4000 & &4023 has been altered. In this case you can write your own routine which will be called each time an instruction is single stepped.

If your code returns with the carry flag set then no action will be taken. However, if the carry flag is clear the monitor will stop and wait for further instructions as if it had encountered a normal breakpoint.

A user conditional breakpoint can be disabled using the IU command.

ASSOCIATED COMMANDS: EU, IU

WB - Wait at breakpoint. Once this command has been issued, if a breakpoint is met whilst code is executing from a JUMP or RESUME, the monitor will beep and wait for a key press, allowing you to study the screen before returning to the monitor.

ASSOCIATED COMMANDS: QB

XAF - Exchanges the accumulator and flags with the alternate accumulator and flags. Alternate registers are shown with a ' following them in the register window.

ASSOCIATED COMMANDS: EXX

ZR - Similar to INIT with the exception that the SP, PC and MP are unaffected.

ASSOCIATED COMMANDS: INIT, RESET

AF * -
BC * -
DE * -
HL * -
IX * -
IY * -
SP * -
PC * -

Each of these commands sets the associated register pair to a given value. If no value is specified the register pair is set to 0, or in the case of the SP and PC they are set to their default values.

ASSOCIATED COMMANDS: A=, F=, B=, C=, D=, E=, H=, L=

A= * -
F= * -
B= * -
C= * -
D= * -
E= * -
H= * -
L= * -

Similar to the above commands except the register associated with each command is set to the given value, and if no value is specified they will always be set to 0.

ASSOCIATED COMMANDS: AF, BC, DE, HL, IX, IY, SP, PC

CONFIGURATION COMMANDS

CPM - Takes you straight into CP/M command mode no matter where the monitor was called from.

ASSOCIATED COMMANDS: QUIT, QUITS

DSL - Set the default symbol table length. By default you are allocated &0100 bytes of symbol table area for defining your own symbols or loading in a small symbol table. If you will not be using a symbol table you can use this command to set a length of 0, thereby saving a page of memory.

Also, if you know the length of any symbol table you will be using you can set the correct length.

This command is useful because, when you load in a symbol table longer than the default &0100 without having set a new length, the old table will be discarded and a new one created, losing you &0100 bytes of memory.

ASSOCIATED COMMANDS: DX, SF

DX - The monitor allows you to configure itself to your specific requirements. This command allows you to specify the name of a file which will be automatically executed as soon as the monitor is loaded.

The file can include any legal commands which will be acted upon as if you had typed them in at the keyboard.

For this command to be accepted by the monitor you must use the command SF to save the configuration file.

ASSOCIATED COMMANDS: DS, SF

QUIT **Q** Quits to the calling program.

ASSOCIATED COMMANDS: QUITS

QUITS **QS** Quits to the calling program saving the current settings including any breakpoints set, window size and register contents.

ASSOCIATED COMMANDS: QUIT

RESET - This command resets the monitor to its startup state. However if your program has altered system variables such as HIMEM or you have loaded a symbol table these remain unchanged.

ASSOCIATED COMMANDS: INIT, ZR

SF - Save a configuration file. This command saves the complete configuration of the monitor including contents of registers, window status, Himem, Lomem and so on. It also saves any default Exec file name and default symbol table length.

ASSOCIATED COMMANDS: DSL, DX

WS - Enables you to change the size of the main window. On a CPC 6128 this can be a value between 0 and 8, on a PCW it can be between 0 and 15. A value of 0 removes the memory and register windows replacing them with a large main window. Any other value sets the height of the main window.

ASSOCIATED COMMANDS: (None)

THE EDITOR

The Arnor Program Editor is a full implementation of the program mode of the PROTEXT word processor. It operates in two modes. A summary of the conventions used in these modes follows:

Edit mode commands

ALT-[means the key marked 'ALT' and the key marked '['. Wherever a hyphen is used between them, it means that the first key should be held down whilst the second key is pressed. Most of the editing commands take this form.

ALT-V T means that the 'ALT' and 'V' keys should be used as described above, then released and the 'T' key pressed. Note that there is no hyphen between the 'V' and the 'T'.

ALT-SHIFT-H means that all three keys should be pressed at the same time. This sort of command that requires more than two keys to be pressed at a time is rarely used and at least two of the keys are always adjacent to each other.

ALT-(means that the 'ALT' key and the key which has the '(' on it are pressed together. It does NOT mean that SHIFT is required as well. The '(' is merely being used for ease of remembering its function.

```
APED    Program  << No File >> Ok   ESC for command mode   CTRL-H for Help
Ch 1    Line 1   Col 20             No markers set                 Insert

            org &100        ; Start

            conin  equ 1    ; BDOS console input function
            bdos   equu 5   ; BDOS call entry point
            return equ &0d  ; Carriage return

                   ld hl,buffer    ; Start of input buffer

loop               ld c,conin      ; BDOS function number
                   call bdos       ; Get a character
                   cp return       ; Is it return?
                   jr z,done       ; Yes, go to done
                   ld (hl),a       ; No, store it in the buffer
                   inc hl          ; Increment pointer into buffer
                   jr loop         ; Get another character

ARNOR Program Editor v2.05 (c) 1987                    Printer: SIMPLE
a>
```

Command mode commands

Command mode commands are always shown in upper case, though when they are being entered into the computer, they may equally well be entered in upper or lower case. Similarly, when entering filenames to LOAD or SAVE a file, even though they may be shown in upper case, lower case is acceptable and the editor will automatically convert them to upper case if required.

Key variations for the CPC6128

There are a number of differences between the keyboards of the PCW computer and the CPC computer. Throughout this section of the manual PCW key names are the ones which are used, rather than cause confusion by listing both keys on all occasions.

The following keys are direct equivalents:

PCW key	CPC key
STOP	ESC
EXIT	ESC
ALT	CONTROL
DEL►	CLR
◄DEL	DEL

There is no direct equivalent to the EXTRA key, but for most purposes, CTRL-0 (zero) serves the same purpose.

Any other variations, where, for example, there is no directly equivalent key on the CPC6128, are noted at the appropriate point in this section

Full advantage is made of the cursor keys and when used in conjunction with SHIFT, or ALT, the effect becomes increasingly greater. For example: Using the right cursor key on its own will move the cursor one character at a time. Using it with the SHIFT key will move a word at a time, whilst with ALT, it will move to the end of the line.

Similarly, the commands to delete make use of the two DEL keys, which on their own will delete one character, but when used with SHIFT will delete a word and with ALT will delete to the beginning or end of the line.

Command mode

Command mode can be recognised by a broad inverse band about two thirds of the way down the screen and immediately beneath this is the command mode prompt 'a>', followed by the block cursor. Whenever the prompt and the cursor are visible, the editor is waiting for a command. Commands are words which are typed in as instructions and may be followed by one or more parameters, depending on the command.

There are a considerable number of commands, which are covered in detail in the 'Command mode' section, but a few of the most commonly used ones will be mentioned here. Commands and parameters may all be entered on the same line, with the parameters being separated from the command by a space. If more than one parameter is specified, they may be separated either by a space, an equals sign (=) or a comma (,).

Any commands which NEED further parameters can be used just by typing the command and the editor will prompt for the parameters. There are, however a number of commands which optionally may have parameters specified, in which case the parameter should follow the command, separated by a space.

Once the command has been specified, the RETURN key should be pressed and the command will be carried out and when completed, the command mode prompt will return.

Drives may be changed by just typing the drive letter, or alternatively, the DRIVE command may be used (DRIVE B). When a drive is changed, the command mode prompt letter will change to suit the new drive. Drives may be catalogued at any time by entering CAT, optionally with the drive letter as a parameter (CAT B will catalogue the contents of drive B).

Documents may be saved and loaded by use of the SAVE and LOAD commands, with the name specified as the parameter (see section on command mode for details of simple methods and current filenames).

Printing is carried out from command mode. There are two printing commands which enable documents or parts of documents to be printed.

In addition, there is a range of commands which allow 'housekeeping' of disc files, such as renaming, moving and copying of files, as well as formatting and copying of discs and these are all covered in detail in the relevant sections.

EDIT MODE

Throughout this part of the manual, the standard editor command keys are described. The PCW computers have a number of 'special' keys on the right hand side of the keyboard and these have been configured to correspond to their originally intended uses as far as possible, which in most cases is merely a duplication of the equivalent editor command. It should, however, be noted that due to the different methods used by the editor for moving, copying and deleting blocks of text, there is some variation in the way that the CUT, CAN and COPY keys are used.

Editing

Once the editor has been loaded, two lines containing information about the state of the program will be seen at the top of the screen. These are the 'Status lines', the contents of which will be explained later. There is also a thin horizontal line, which always marks the end of the text and about two thirds of the way down the screen is another, broader, line containing further information.

At this stage the program is still in Command Mode, which is described in detail in the next chapter, but pressing the STOP key will put the program into Edit Mode, which is the mode used for all entry and correction of text. The line two thirds of the way down the screen will disappear, leaving the lower part of the screen clear. Pressing the STOP key at any time will return to command mode.

Entering text

Once in edit mode a flashing cursor is positioned beneath the status lines and anything that is typed at the keyboard will appear on the screen at this position and the cursor will be moved forward one position.

Any mistakes made whilst typing, which are noticed at the time, may be corrected by pressing the ◄DEL key, which will cancel the last key pressed.

The cursor can be moved around the screen by pressing the four cursor keys. By using these keys, text may be entered at any position. The cursor moves one line or column for each press of a cursor key. Holding a cursor key down will make the cursor move continuously - release the key and the cursor will stop.

The cursor cannot be moved past the end of text (the thin horizontal line on the screen). To position the cursor further down, the end of text must be moved down by positioning the cursor at the end of the text and pressing RETURN as many times as required.

Upper and lower case

Initially the letter keys produce lower case letters, unless SHIFT is pressed at the same time. If SHIFT LOCK or CAPS LOCK is pressed, upper case letters are always produced, and this is indicated on the status line.

Note: The Amstrad PCW computers are slightly unusual in having a SHIFT LOCK and no CAPS LOCK key. When SHIFT LOCK is on, all the characters on the upper part of those keys which have more than one character on them will also be selected. Caps lock is selected by pressing ALT-ENTER.

The editor has two commands which change the case of a letter. To make a letter upper case, press ALT-/ when the cursor is on the letter. This command only affects letters, so the cursor can be moved quickly over a line to convert all letters to upper case by holding down ALT-/. Similarly, ALT-. (point) will convert upper case letters into lower case.
(CPC6128 equivalent: CTRL-\).

Deleting and inserting

The ability to move the cursor around, permits the correction or alteration of text anywhere on the screen. The cursor should be positioned on the letter to be changed and the DEL► key pressed. This will remove the letter at the cursor position, and move the rest of the line to the left. As many letters as required can be deleted in this way. If the new letter is now entered it will appear on the screen and the rest of the line will move back to the right. Alternatively, pressing ◄DEL will remove the character to the left of the cursor and the text will again move to the left to fill the gap. Repeated pressing of either DEL key will cause further characters to be deleted.

If extra text is to be inserted, the cursor should be positioned where the first new character is to be added and the new text entered.

To insert a new blank line into the text, ALT-I should be used. The cursor will remain where it is and all text from the current line to the end of the document will be moved down a line.

Just as a character can be deleted, so can a word. Pressing SHIFT and DEL► when the cursor is at the start of a word will make the word disappear. If this is done when the cursor is in the middle of a word, only that part of the word at and to the right of the cursor position will be deleted.

Similarly, pressing SHIFT and ◄DEL will remove the word to the left of the cursor, or if positioned in the middle of a word, the characters to the start of the word.

ALT-◄DEL will delete all text from the character on the left of the cursor to the start of the line and ALT-DEL► will delete all text from the cursor to the end of the line. ALT-E also deletes everything from the cursor position to the end of the line. (CPC6128 users: only CTRL-E is available).

ALT-CAN will delete the whole line. The line is removed from the document and the remainder of the text moved up a line. (CPC6128 equivalent: CTRL-CLR).

Note: Pressing ALT-◄DEL followed by ALT-DEL► will delete all the text from a line, but will not remove the empty line from the text, unlike ALT-CAN, which will remove the blank line as well.

Swapping two characters

A common typing mistake, especially when typing quickly, is to type two letters the wrong way round, e.g. 'wrod' instead of 'word'. The ALT-A (Alternate characters) command will put this right. The cursor should be positioned on the first of the two offending characters (on the 'r', in the above example) and ALT-A pressed. The two characters will then be exchanged.

Un-deleting all or part of a line

The editor maintains a buffer which always contains the most recently deleted section of text. If a line or part of a line, more than three characters long, is deleted, the deleted text will be saved in the buffer. If a section of text has been accidentally deleted, it may be restored by pressing ALT-U.

This command can also be put to good use for moving lines or parts of a line to a different position in the text, though this is not the purpose for which it is really intended. The text to be moved should be deleted using one of the word or line delete commands and the cursor moved to the position in the text where the deleted text is to be placed. Pressing ALT-U will then restore the text at the new location.

Note: Only the text removed by the last delete command will be stored in the buffer and any previous contents of the buffer will be lost. It is therefore only possible to un-delete a section of text until such time as any other section of text is deleted.

Insert and Overwrite mode

Initially the editor, by default, is in insert mode and the word 'Insert' is displayed on the status line at the top of the screen to indicate this. This means that when text is typed, the rest of the text on the line is moved along to the right to make room. This is the mode that is preferred by most people for text entry.

Pressing ALT-TAB will change the status line to 'Overwrite'. Selecting overwrite mode can make certain editing jobs easier. The effect of using it is that if the cursor is positioned over an existing piece of text and new text typed in, the existing text will be replaced by the new text, unlike insert mode, where the existing text would be moved to the right.

If an extra character needs to be inserted whilst in overwrite mode (for example if replacing a word by a longer word), this can be done by pressing ALT and the space bar which will move the text to the right to make room.

Moving the cursor more rapidly

So far the cursor has been moved by a character at a time, but there are also various ways to move the cursor more quickly. These are as follows:

(a) Pressing SHIFT-► or SHIFT-◄ will make the cursor jump a word to the start of the next (or last) word. This feature is useful for moving more quickly to a word which needs correction.

(b) Pressing ALT-► or ALT-◄. This moves the cursor to the beginning or end of the line.

(c) Pressing SHIFT-RETURN or ALT-RETURN. This moves the cursor to the beginning of the next line, without causing a new line to be inserted, which would happen if the RETURN key was used on its own.

(d) Pressing ALT-▲ or ALT-▼. This moves the cursor up or down rapidly. By holding down ALT-▲ or ALT-▼ the text can be rapidly scanned. The text will scroll by nearly a screenful at a time, but with a few lines overlap so that the context may more easily be followed. Similar functions are performed by ALT-Q and ALT-Z, except that a full screen is scrolled each time, with no overlap of text.

(e) Pressing ALT-[or ALT-] moves the cursor to the beginning or end of the text resident in memory at that time. Pressing the same key a second time will move the cursor to the beginning or end of the complete document.

(f) Pressing ALT-@ [or ALT-@] will move to the opening or closing block markers, if set.

(g) Pressing ALT-[+] or ALT-[-] will go to the next or previous marker in the document. See 'Place markers'.

(h) Pressing ALT-L moves the cursor back to the last position. This is particularly useful if the cursor has accidentally been moved to another part of the text by using an incorrect command. ALT-L will return the cursor to the position where it was before the incorrect move was made. It will only have any effect if the cursor has been moved with one of the 'jump' commands. Moving the cursor a single space at a time will not affect the use of ALT-L and it can still be used to return to the original position from which the last jump was made.

With care, this facility can be put to good use, by permitting a jump to another part of the text, where one or two alterations or additions may be made, before pressing ALT-L to return to the original place in the text.

Moving to a specified line or column number

Pressing ALT-G will result in a message appearing on the status line, requesting a line or column number. If just a number is entered the cursor will move to that line. If C is followed by a number it will move to that column.

Place markers

A place marker can be put anywhere in the text and is similar in use to a book marker. Ten place markers can be set, numbered 0 to 9. A place marker is set by pressing ALT-@ followed by the number. When a marker has been set, it will appear in the text as the number in inverse and will be shown on the status line, so that by looking at the status line it is easy to see which markers are available. Once a place marker has been set, it can easily be returned to at any time by repeating the ALT-@ command with the same number.

In addition to using ALT-@ and the number to find a place marker, it is possible to jump from one to the next in the document by using ALT-[+] to move on through the document, or ALT-[-] to move backwards. Using these commands will find the next or last marker in the text. All types of markers (place, and block) will be found. They are not treated numerically, but are found in the order in which they occur in the document. (CPC6128 equivalents: CTRL-@ + moves to next marker, CTRL-@ - moves to previous marker).

As an example of the use of a place marker, suppose a long file is being edited and something needs to be added at the top of the text. A place marker can be set and ALT-[typed, to move to the top of the text, and after making the addition, ALT-@ and the place marker number used to move back to the place marker.

Note: Place markers are saved with the text and will be restored when the file is reloaded.

Scrolling

When the text fills the entire depth of the screen, typing further text will cause the screen to scroll up. That is, the top line will disappear and the rest of the screen will move up one line to make room for a new line at the bottom of the screen.

In the same way the text will scroll if the cursor reaches the bottom of the screen but there is more text to come, or reaches the top of the screen when the text has previously scrolled. This is known as vertical scrolling, and is essential for editing text that is longer than a few lines.

The editor has commands to force the screen to scroll either up or down at any time. This is done by pressing SHIFT-▲ or SHIFT-▼. The cursor will stay on the same line, but the whole text will scroll by one line. This feature is useful if a line is to be edited and it is desirable to see the text beneath or above.

There is another form of scrolling, called horizontal scrolling, which happens automatically when the cursor is moved beyond the right hand limit of the screen. If this is done the text will scroll to the left. This means that the text on the left of the screen will start to disappear as the cursor is moved further to the right of the screen. Horizontal scrolling allows text to be entered in lines that are longer than the screen width. This can be confusing at first and so is best avoided initially. If horizontal scrolling occurs, any of the commands which move the cursor to the left may be used to scroll the text back, or SHIFT and RETURN may be pressed together, which will return the cursor to the start of the next line.

Splitting and joining lines

Lines will often require splitting, or joining together. This is very easy in the editor. There are two different methods of doing this, depending on whether 'Insert' or 'Overwrite' mode is in operation.

To split a line whilst in Insert mode, the cursor should be moved to the character which is to be the first on the new line and RETURN pressed. To join two lines, either move to the end of the first line and press DEL►, or move to the start of the second line and press ◄DEL. The text on the second line will then move up and join onto the end of the text on the first line.

If in overwrite mode, ALT-* will split the line at the cursor and ALT-+ will join the next line to the end of the current line.

Block commands

The editor allows any section of text to be moved or copied to any other part of the text. This is often called 'cut and paste' editing.
This section will describe the ways to use block editing. A block of text is any continuous section of text. It may be of any length and may start at any position in the document and finish at any position. When in block editing mode, all text between these two points will be manipulated in whatever way is chosen.

Defining a block

The first requirement is that the block of text is marked with block markers. The cursor should be moved to the start of the section of text and SHIFT-COPY pressed (alternatively SHIFT-[+] can be used and may be found more convenient). This will set a block marker. The marker will be indicated on the screen by an inverse video square bracket. The cursor should then be moved to the end of the section and SHIFT-COPY pressed again, to set a second marker. The block has now been defined. An opening square bracket is the start marker, a closing square bracket the end marker. When markers are defined, this will be indicated on the status line, where the message 'No markers set' will be replaced by 'Markers [ ]', showing that both the start and end markers are set.

The markers can be set in either order, and can be at any position in the text. The first marker set will be displayed as an opening bracket, but if the second marker is positioned earlier in the text than the first marker, this will change to a closing bracket. If the marker is put in the wrong place, pressing SHIFT-COPY again while the cursor is still on the marker will remove it. Either or both block markers can be cleared at any time, by pressing ALT-K or CAN. Often a block will consist of a number of complete lines. To define a block like this, the first marker should be positioned at the start of the first line, and the second marker at the start of the line following the last line of the block.

If an attempt is made to set a marker when both are already set, a beep will sound and an error message will be displayed on the status line. Pressing STOP will resume editing and ALT-K can be used to clear the markers.

Moving or copying a block

Once a block has been defined, it can be moved to any point in the text simply by moving the cursor to the required position and pressing ALT-M or alternatively on the PCW, the PASTE key. The markers will move with the text. The cursor must not be within the block at the time; if it is an error message will be displayed on the status line. Pressing STOP will return to edit mode and the cursor can be moved to the correct position.

The block can also be copied, leaving the original text intact. This is done by pressing ALT-COPY or just COPY (PCW only). The markers will be moved with the block, which makes it easy to see clearly where the new copy of the block is and also to copy the block again if required. The cursor must not be within the block. (CPC6128 equivalent: CTRL-COPY).

Deleting a block

The section of text to be deleted must be defined in the usual way. Pressing the CUT key will delete the block. If the block is larger than a certain size (see below) a beep will sound and a warning message will be displayed on the status line, requesting confirmation that the block is to be deleted. The block will only be deleted if 'Y' is selected. (CPC6128 equivalent: CTRL-DEL).

Un-deleting a block

If a block of text is accidentally deleted, it may often be recovered by use of the ALT-U command. When text is deleted, the editor retains the deleted block in a buffer and ALT-U will restore it to the document.

Note: A block can only be restored until such time as further text is deleted, after which time the buffer will contain only the most recently deleted text.

FIND AND REPLACE

Note: CPC6128 users should note that there is some variation in the use of keys in this chapter, as the CPC6128 does not have the special [+] and [-] keys. The COPY key serves the same purpose as the [+] key and CTRL-@ @ is the substitute for [-]. SHIFT-f2 and CTRL-f2 generate REPLACE and FIND respectively instead of the PCW EXCH and FIND keys.

Two functions, FIND and REPLACE are provided, which permit searching through text for any string of characters and, if specified, replacing them with a second string.

Pressing FIND or EXCH whilst in edit mode will cause the editor to enter command mode, with a request for the 'String to find'. Alternatively, typing FIND or REPLACE from command mode will have the same result.

The string to find is requested first, followed by the replacement string (if the REPLACE option was selected). After entering the string or strings, one or more of a number of options may be selected by typing the appropriate letters one after another (in any order). Each option is either a single letter abbreviation or a number (these are listed on the screen). Pressing RETURN on its own will cause no options to be selected.

The options available are as follows:

G Global search. If selected the whole text is searched from the start, otherwise only the text from the current cursor position to the end of the text.

C Case specific search. If selected all letters will only match letters that are the same case, otherwise either capitals or lower case letters will be treated as being the same.

W Find string only if it appears as a complete word. For example to find occurrences of the word 'and' without finding 'hand', 'England' etc.

B Search backwards. Searches from the end of the document to the beginning.

A Find or replace all strings automatically. REPLACE will change all occurrences of the string with the new one, without requesting confirmation and return a figure of the total number of replacements made. In the case of FIND being used, it will simply return the total number of occurrences of the string.

n Find or replace the nth occurrence. n should be a number between 1 and 255. This option has a number of uses, but a simple example might be to check that every set of quotation marks has a matching closing set, in which case FIND would be used to find '"' and '2G' would be specified as options, to search globally for every second occurrence.

If no options are selected the search will be forwards, from the current cursor position to the first occurrence of the string, ignoring the case of letters, finding the string even if it occurs as part of a longer word, and asking for confirmation before replacing a string.

Any number of wildcards are allowed in the string. A wildcard is a character that matches any character in the text, except the return character. It is entered in the string by typing a question mark (?).
A tab character may be entered simply by pressing the TAB key. It is displayed as a right pointing arrow.

There are various characters that cannot be entered directly, but that it may be useful to include in a search string. Provision has been made for including these in a string, by means of an 'escape character'. The 'escape character' (!) should be typed in, followed by a symbol, number or letter, as appropriate.

The full list of characters that are entered by this means is:

question mark	!?
exclamation mark	!!
return	!.
search for code	!<number>

Using FIND

Once the string and any options have been selected, edit mode is entered and the cursor placed on the first character of the first occurrence of the string. To find the next occurrence of the string, the [+] key, positioned to the left of the space bar, should be pressed. This need not be done immediately. Editing can be carried out first and when complete, the search may be continued by pressing [+]. At any stage, [-] can be used to search back towards the beginning, if necessary.

As with other commands, FIND can be used by typing the string on the same line as the command name, followed by any options. Thus the command 'FIND word GWC' will search for the string 'word' from the start of the document, selecting only those occurrences where it is a complete word with all letters in the same case as specified. If no options are specified, the default options will be used.

If the A option is selected, the editor will return the total number of occurrences found, when the search is complete.

Using REPLACE

The cursor will be positioned on the first character of the string and a message, 'Replace (y/n)?',will be displayed on the status line. Pressing 'Y' will replace the string with the new one and the cursor will move to the next occurrence. Pressing 'N' will leave the string untouched and move the cursor to the next occurrence. Alternatively STOP may be pressed and normal editing resumed. At a later time, [+] may be pressed to resume the find and replace operation. Alternatively [-] may be used to resume the search in the reverse direction, which may be found useful if an occurrence of the string is passed over by pressing 'N' in error.

If option A is selected then all occurrences of the string are replaced without prompting and the program will remain in command mode. When complete a count of the total number of changes made will be displayed.

Examples

1. To find all occurrences of the word 'text' in lower case only, starting at the cursor position.

   ```
   FIND string: text
   Options: CW
   ```

2. To convert all occurrences of 'rom' or 'Rom' to 'ROM', asking for confirmation of each replacement.

   ```
   FIND string: rom
   REPLACE with: ROM
   Options: GW
   ```

3. To insert a blank line after each line.

   ```
   REPLACE !.  !.!.  AG
   ```

COMMAND MODE

All entry of text is carried out in Edit mode, but in order to carry out operations such as saving, loading or printing, 'command' mode must be entered. This can be done at any stage of editing simply by pressing STOP. Pressing STOP a second time will return to edit mode.

When STOP is pressed, the bottom part of the screen will be cleared and the command mode banner line will appear, displaying the editor version number. The cursor will be positioned next to a '>' symbol. This symbol is the 'command prompt' and indicates that commands may be entered. The currently selected drive is indicated by the letter prefixing the > and if any 'group' other than group 0 is selected, this number will also be indicated.

The output of all commands will be displayed in this window at the bottom of the screen. Many commands produce more output than will fit in the window in which case the screen will automatically scroll as necessary.

General information

Before studying the individual commands in detail, there are a number of points connected with the entry of commands which are of general interest and are listed below.

Command entry

The editor has a special feature which permits the entry of commands in a simplified fashion. For example, to save a text file it is only necessary to type 'SAVE' and the editor will prompt with 'SAVE filename:' and wait for entry of a name for the text file.

Alternatively, the parameters of a command may be entered on the same line as the command name, e.g. 'LOAD source', 'SAVE prog'. In this way the commands may be used without the prompts for the parameters appearing, which is often more convenient when familiar with the syntax of the commands.

Note: All commands which **require** a parameter will prompt for them if the command is used on its own. Commands which have optional parameters require these to be entered at the same time as the command.

The editor provides a sophisticated line editing facility which is in operation whenever commands are being typed in. If a mistake is made the cursor can be moved back and the mistake corrected in the same way as in edit mode. The following editing commands are available in command mode:

PCW8256/8512	CPC6128	
◄	◄	Move cursor left one character.
►	►	Move cursor right one character.
ALT-◄	CTRL-◄	Move to start of line.
ALT-►	CTRL-►	Move to end of line.
DEL►	CLR	Delete at cursor.
◄DEL	DEL	Delete before cursor.
ALT-A	CTRL-A	Alternate characters.
ALT-◄DEL		Delete to beginning of line.
ALT-DEL►	CTRL-E	Delete to end of line.
ALT-TAB	CTRL-TAB	Switch between insert and overwrite modes.
CAN	CLR	Clear screen (if cursor at the start of a line)
STOP	ESC	Abandon entry of current command.

Pressing COPY or [+] when the cursor is at the start of a line recalls the last command line used that was 4 or more characters in length, and positions the cursor at the end of the command. This has a number of uses, such as carrying out multiple saves of the same file, or repeating a load command which failed because the wrong disc had been inserted. Short commands such as 'A', 'CAT', 'SW' do not affect the command recalled.

Abbreviations

Many of the commands can be abbreviated. For example, there is no need to type 'LOAD' in full, typing 'L' will serve the same purpose. Similarly 'S' for 'SAVE' and 'P' for 'PRINT'. A full list of the commands, abbreviations and their parameter syntax, is given later in this chapter.

The current filename

After a file has been loaded, or once a piece of text has been saved, the name of the file will be displayed on the status line. This becomes the 'current filename' and is remembered by the editor until changed, either by saving with another name, by use of the NAME command, or by loading a new file. Once a file possesses a current filename the name may be omitted when saving a file. Entering the SAVE command, and just pressing RETURN when the prompt 'SAVE filename:' appears, will save the file with the current filename. Care must be taken to ensure that it is indeed the correct name, to avoid accidentally erasing something else. If SAVE is typed and the name displayed is incorrect it can be edited as described above ('Command entry').

EDITOR COMMANDS

This section gives full details of the commands available in command mode. Details of the syntax used and what the command does are given together with any optional extensions to the basic command.

Many of the commands allow the use of ambiguous filenames. An ambiguous filename is one which contains 'wildcards'. The editor has two types of wildcards, which may be used in the same way as with CP/M commands.

?	may be used to mean 'any single character'.
*	may be used to indicate 'any number of characters'.

For example:

DATA?.TXT	Any filename beginning with 'DATA' and having one further character (which may be blank), with the suffix 'TXT'.
B*.*	Any filename beginning with 'B', of any length and any suffix.
.	Any file.

Note: Only one '*' may be used in each part of the filename and suffix.

LOAD (L)

Syntax: LOAD <filename>

Description: A document will be loaded into memory from a disc file of the specified name. A warning message will be given if the text currently in memory has not been saved. Press 'Y' to confirm that this text is to be discarded.

Note: If only the command name is entered, the editor will prompt for a filename. Once loaded, the specified filename will become the 'current filename'

MERGE (MER)

Syntax: MERGE <filename>

Description: This is similar to LOAD but whereas LOAD clears any existing text from memory and then loads the file in, MERGE inserts the new file into the existing text at the current cursor position.

Note: Care should be taken to ensure that the cursor is in the required position before using this command.

Note: The current filename is NOT changed.

NAME (N)

Syntax: NAME <filename>

Description: Permits the name of the document in memory to be changed. The new name becomes the 'current filename'.

SAVE (S)

Syntax: SAVE <filename>

Description: The complete document in memory will be saved to a disc file with the name specified.

Note: If only the command name is entered, the editor will prompt for a filename. If the file already has a 'current filename', then pressing RETURN will result in the file being saved with the same name. Alternatively, a new name may be specified, which will then become the current filename.

SAVEB (SB)

Syntax: SB <filename>

Description: This is the same as SAVE except that only the text within the block defined by the block markers is saved.

Note: The current filename is NOT changed.

SWAP (SW)

Description: Swaps between two documents in memory. All settings of the files and cursor, block markers etc are retained. See 'Two file editing' for full details.

Printing options

The following commands determine the form that printing will take.

BM

Syntax: BM <number>

Description: Bottom Margin. Specifies the number of lines to be left blank at the bottom of each page.

PL

Syntax: PL <length>

Description: Sets the length of each page in lines.

PRINT (P)

Syntax: PRINT (num)

Description: This command prints the document in memory.

PRINTB (PB)

Description: Only the section of text defined by the block markers will be printed.

PRINTER (PR)

Syntax: PRINTER (name)

Selects the printer driver you will be using.

TAB

Syntax: TAB <column(s)>

Description: Sets a tab stop at the specified column or columns. The last number in the list may be preceded by '*'. This causes tabs to be set at equal intervals up to column 128.

Example:

TAB 8,15,*5 sets tabs at 8 15,20,25,30,.....

TAB without any parameters sets default tabs at every 8th column.

MISCELLANEOUS COMMANDS

CLEAR

Description: Clears the text currently in memory. A request for confirmation is made before this is done. On the PCW computer the same effect is achieved by typing ALT-SHIFT-CAN in edit mode.

CPM

Description: Quits straight to CP/M.

FIND (F)

Syntax: FIND <text> (<parameters>)

Description: The document will be searched for the first occurrence of the specified text, according to any parameters specified and the cursor positioned on the first character.

See chapter on Find and Replace for full details.

GOTO (G)

Syntax: GOTO <line number>/<column number>

Description: Moves the editing cursor to the specified line or column, eg. GOTO L125 or GOTO C70.

NUMBER (NUM)

Description: The purpose of this command is to add line numbers to, or remove line numbers from, the beginning of every line of text. This command will prompt for whether numbers are to be added or removed from the document. If the choice to add line numbers is selected, a starting line number and the value by which each subsequent number is to be incremented will be requested.

This provides a convenient method of writing Basic programs, amongst other uses, using the editor's full screen editing facilities and finally adding line numbers prior to saving a program.

NUMBERB (NUMB) This command is similar to NUMBER but only adds or removes numbers within a marked block.

QUIT (Q)

Description: Quits the editor and returns to CP/M command mode. If a document is in memory and any changes have been made to it since it was loaded or last saved, a caution will be issued, warning that the document has not been saved and asking for confirmation of the desire to continue.

REPLACE (R)

Syntax: R <text> <newtext> (<parameters>)

Description: The document will be searched for the first occurrence of the specified text, according to any parameters specified, and the cursor positioned on the first character.

External commands

External commands call other utility programs from disc. The program files specified must be available at the time the command is used. They may be on any disc drive - the editor will search all drives to find the file. The following utility programs are designed so that on completion of their task, a return is made to the editor with any text that was in memory at the time the command was called still intact.

Note: It is possible to call other programs from within the editor command mode and they should be prefixed with an asterisk (*). When this is done, the text in memory will be saved to a temporary file and the name passed as a parameter to the program being called. For example, typing '*BCPL' will call the compiler called BCPL (i.e. the program file BCPL.COM), which will then compile the source code which was in memory. Unless these programs have been written specifically for the purpose, they will not return to the editor automatically and this must be done by typing 'APED' from CP/M command mode. The temporary file will automatically be loaded back into the editor.

PROGRAMMING COMMANDS

ASM — Assembles the file in memory, or the file specifed as a parameter.

MA — As above, except if no filename is given, the assembler prompts for a filename.

MON — Runs which ever version of the monitor it finds first, passing the name of the file being edited. If this file has been assembled the object code is automatically loaded in to the monitor.

MM — As above but runs the large version of the monitor.

MSM — As above but runs the small version of the monitor.

(The last two commands will not attempt to load in the object code of the program being edited.)

AC — Compiles links and runs a C program. If there are any compilation errors the program is not run.

RUNC — Enters the C run time system.

***** — Runs a specified CP/M Plus program. For example *BCPL will run the BCPL compiler.

LARGE FILES

The editor is capable of handling large files very efficiently and the only limit on the size of the files which can be edited is the capacity of the disc drives. It must be remembered that under CP/M, large files cannot be totally loaded into memory at one time, and as editing continues and progress is made through a long document, the editor will automatically save parts of the document as temporary files.

As a result, it is preferable to start editing a large file with as empty a disc as possible. With the PCW computers, drive M is normally used as the drive on which these temporary files are stored. On the CPC6128, which does not have a memory drive, the temporary files are saved onto the text file disc. This would normally be drive B on a two drive system.

In the event that the document becomes so large that there is no room left for the temporary files to fit, a warning will be issued with the option to delete the original file. If 'Y' is selected, then editing will continue as before. If 'N' is selected, then it will usually be possible to delete one or two files from the disc to make room, before continuing. For example the disc might have copies of the help files on it, in which case deleting these would give more space.

Note: If the decision to delete the original file is made, it should be borne in mind that if an accident happens to the file which is being edited, the original will no longer be available. The solution to this is to make sure that a back up copy of the file is saved on another disc before editing commences.

Other than the points mentioned above, editing of large documents is exactly the same as editing any other document. It should also be remembered that the ALT-[and ALT-] commands move to the start and end of the text in memory, not the start and end of the whole document. With a small document this will be the same thing, but if the start or end of a long document is required, then ALT-[or ALT-] should be pressed a second time.

Important notes on large file editing.

1. It is important to ensure that a disc is present in the selected drive at all times and that it is not changed for another disc during the course of editing the document.

2. The editor saves temporary files with various names commencing with 'APED' Under NO circumstances must any of these files be deleted. When the document is completed and saved, the editor will automatically delete the temporary files which are no longer required.

Are large files necessary?

Even though the editor can handle 'unlimited size' files, this is perhaps a suitable place to consider whether it might be more convenient and efficient to work with a number of smaller files. Rarely is there any NEED for a long document to be in one piece. For example: A long program can be broken down into a number of sections or subroutines.

Whilst it may appear that there are advantages to being able to work on one long file so that it can all be viewed and edited at the same time, there are a number of points which should be considered.

1. In the event of a catastrophe, such as a power failure, or accidentally deleting a file from a disc, if the text is in one long file, the complete file may be lost.

2. Due to the limited amount of memory available under CP/M, it is not possible to have the whole of a large file in memory at the same time and as progress is made forwards and backwards through the file, parts of it have to be saved to temporary files and other parts loaded. The editor has specially written routines which do this more efficiently than other programs, but it can still take a short time to jump from one part of a file to another, whereas with a smaller file this will to all intents and purposes be instantaneous.

3. It is usually easier to locate specific sections of text in a smaller file.

4. Usually only a relatively small part of a file will actually be worked on at a time and it is considerably quicker to load, and save smaller files.

TWO FILE EDITING

The editor provides the facility to work on two files at the same time. These files are maintained quite separately and are loaded and saved individually. Any operation can be carried out on one file without affecting the other, the cursor location and all markers being maintained for each file. Blocks of text can be copied between one file and the other.

This is an extremely powerful function and is controlled by only three commands, one of which is used from command mode and the other two from edit mode.

```
SWAP (SW)       : Command mode - Swap between two files in memory
ALT-O           : Edit mode    - Copy block over from the other file
ALT-Y           : Edit mode    - same function as SWAP
```

To load a second file, 'SW' should be entered from command mode and the current file will be switched, leaving an empty file. The second file should be loaded in the normal way. Switching between the two files will cause the information on the status lines to change to suit the current file, enabling easy recognition of which file is being worked on.

In edit mode, ALT-Y performs exactly the same purpose as 'SW', enabling quick switching between files.

The ALT-O (letter o) command is extremely useful, as it enables any part of the text of either file to be copied over to the other.

Before a block of text can be copied over, the block should be marked out using the markers in the normal fashion. Typing ALT-Y will swap files and the cursor should then be positioned where the text is required. If ALT-O is then pressed, the block will be copied across.

If the original text is no longer required, ALT-Y should be pressed again, to return to the original file, followed by CUT, to delete the original text.

Two file editing is also very convenient as a means of keeping notes, for later attention, during the course of editing a file. Press ALT-Y, make the note and ALT-Y again, to return to the original file.

Another use for ALT-O is for transferring text from one file to another - load the first file, type SWAP, load the second file and use ALT-O to copy the blocks required into the first file, before re-saving it. This is quicker than using SB (save block), loading the other file and merging the saved block of text into the file and finally resaving it.

SPECIAL CHARACTERS

The editor is capable of being used with most non-English languages and fully supports the use of accents and characters such as c-cedilla.

Characters containing accents may be typed in during the course of editing and will appear correctly on screen.

There are seven main accents which are required to cover the usual range of European languages and these may be obtained in the following way.

The base character should be entered first and then immediately followed by EXTRA and the number key which contains the required accent (the accents and their keys are listed below). The accent will then be positioned over the character. Accents may be used with any character, which permits the use of the editor with a number of languages which normally are not catered for. Welsh and many of the Eastern European languages are covered.

Note: CPC6128 users should note that CTRL-1 to CTRL-7 are used to obtain accents, instead of the EXTRA key and a number key. The special characters are obtained by pressing CTRL-0, followed by the appropriate letter key, or by pressing one of the function keys with either SHIFT or CONTROL (see below).

If an accent is required by itself, press space followed by the accent key. Should any of the accent characters be required frequently it is possible to re-define the keys to give just the accent.

Accents supported

PCW key	CPC key	Accent
EXTRA-2	CTRL-3	Umlaut
EXTRA-5	CTRL-5	Ring
EXTRA-6	CTRL-7	Acute accent
EXTRA-7	CTRL-6	Circumflex
EXTRA-8	CTRL-1	Grave accent
EXTRA-0	CTRL-4	Inverted circumflex
EXTRA-hyphen	CTRL-2	Tilde

Note: The keys used on the PCW are the same ones used under CP/M, with the exception that EXTRA-0, the 'Inverted circumflex', which is used by a number of Eastern European languages, is an additional accent.

In addition to these accents, which may be used on any character, a number of phrases are initially defined as special 'non-English' characters and in the case of the PCW, these are the same keys as are used in CP/M. A list of the keys to be pressed is given below.

Note: The phrases may be redefined and care should be taken when selecting phrases, if any of these characters are required.

Summary of special characters available from the keyboard

The command LPHRASES displays all the characters available using the EXTRA key (or function keys on a CPC 6128).

PCW key	CPC key	Character
f3	CTRL-f1	Lower case o slash
f5	CTRL-f4	Lower case diphthong
f7	CTRL-f7	Lower case c cedilla
SHIFT-f3	SHIFT-f1	Upper case O slash
SHIFT-f5	SHIFT-f4	Upper case diphthong
SHIFT-f7	SHIFT-f7	Upper case C cedilla
EXTRA-A	SHIFT-f5	Superscript a
EXTRA-C	CTRL-f0	Copyright
EXTRA-O	SHIFT-f6	Superscript o
EXTRA-P	CTRL-f8	Paragraph symbol
EXTRA-S	SHIFT-f0	Eszett
EXTRA-Y	SHIFT-f3	Yen sign
EXTRA-?	CTRL-f5	inverted ?
EXTRA-!	CTRL-f6	inverted !
EXTRA-<	SHIFT-f8	French open quotes
EXTRA->	SHIFT-f9	French close quotes
SHIFT-ALT-◄	CTRL-V◄	Left arrow
SHIFT-ALT-►	CTRL-V►	Right arrow
SHIFT-ALT-▲	CTRL-V▲	Up arrow
SHIFT-ALT-▼	CTRL-V▼	Down arrow

SETPRINT AND CONFIG

These two utilities were originally written for PROTEXT. To retain compatibility between the editor (APED) and PROTEXT they are included on the disc. This means that many of the functions are not relevant to the program editor.

SETPRINT allows you to create or modify a printer driver. This is done by simple selection of choices from menus. Default and previous settings are always shown. Printer drivers have the extension .PTR.

CONFIG enables you to change such things as the temporary text drive to configure the editor to your own requirements. As with SETPRINT this is easy to do as you are guided through the program with menus and default and previous options are shown. Config files have the filename PROTEXT.CFG.

It is recommended that you get used to using the editor before attempting to change its configuration. If you create a configuration you don't like you can either delete PROTEXT.CFG or re-enter CONFIG to adapt it.

It may be found useful to change the following options with CONFIG:

Temporary text drive - the drive used to store temporary files. On a PCW computer this should be M, on a CPC 6128 A or B may be used.

Key translations - the keyboard may be fully re-defined.

Default printer driver - the printer driver loaded automatically. Set this to EPSON.PTR if using an Epson or compatible printer such as the DMP 2000.

Autoexec file - an EXEC file that is loaded automatically when the editor is started. This may usefully contain configuration commands such as TAB, PL and BM.

COMMANDS COMMON TO THE MONITOR & EDITOR

COMMANDS COMMON TO THE EDITOR, MAXAM II MONITOR AND C

The following commands can be used within the Maxam II monitor the editor and the C Run Time System. In some cases their applications will be slightly different. A note is made where these differences occur. Those commands which are only available in the large version of the monitor are shown followed by a *. All these commands are available in the editor and C.

Note: To save room in the small version of the monitor, many commands such as DRIVE or GROUP will not prompt you for input. Parameters should be entered in-line, i.e. following the command.

ACCESS (ACC)

Syntax: ACCESS <ambiguous filename>

Description: Sets the status of a file or files to 'Read-write'. Wildcards are permitted. See PROTECT for details of the reverse operation.

CAT (DIR)

Description: Performs a catalogue of the files on a disc. By default, with no parameters, it will catalogue all the files on the currently selected group of the currently selected drive.

Extensions: Filenames, drive letters and group numbers.

Syntax:
CAT <ambiguous filename>
CAT <drive letter>
CAT <group/user number>

Description: Either another group OR another drive may be specified. Alternatively a filename may be specified using wildcards, optionally with a drive letter prefix.

Example: CAT B:*.LTR will catalogue all the files with a LTR suffix on drive B and group 0.

The files are listed in alphabetical order with the size of each file shown. The amount of free disc space is also shown. If this last figure becomes too small it will be often be necessary to erase backup files in order to save a file. The catalogue also displays a symbol by certain files:

Note: * indicates a protected file (see PROTECT, below)

COPY *

Syntax: COPY <oldname> <newname>
COPY <ambiguous filename> (<group>) (<drive>)

Description: There are two variations of this command, the first of which will copy a file giving it a new name. The filenames may be prefixed with the drive letter to copy a file from one drive to another with a different name.

The second variation permits the use of 'wildcards', but the names cannot be changed. This allows the transfer of one or a number of files with some common feature, from the current group on any drive to any group on any drive. Either <group> or <drive> or both may be specified.

Examples: COPY B:OLDNAME NEWNAME
COPY *.TXT 1
Copies all files with suffix 'TXT' into group 1.
COPY B:*.* 2M
Copies all files on drive B (current group) to drive M group 2.

Note: Any existing file with the same name, in the destination drive/group, will be renamed with a '.BAK' suffix.

Note: Copying does not erase the original files, so if they are no longer required, ERASE must be used after the copying process.

DCOPY

Description: Calls an Arnor utility program which copies the contents of one disc onto another. This command will copy single sided single density (CF2) discs only. See 'Utility Programs' for full description.

Note: The original contents of the disc to which the files are being copied will be erased.

Note: In order to copy CF2DD double sided discs, as used in drive B on the PCW8512, it is necessary to leave the editor and use the DISCKIT program which is on the System Utilities disc supplied with the computer.

Note: DCOPY can not be run from within the monitor. To use it you will have to go into CP/M or the editor.

DFORM *

Description: Formats a disc to either CF2 or CF2DD format, depending on which drive is selected. On the PCW computer a disc in drive B will be formatted to CF2DD format and a disc in drive A to CF2 format. On the CPC6128 data format is always used.

Note: Both sides of a CF2DD disc are formatted at the same time, but when formatting a disc in A to CF2 format, it is necessary to format each side separately.

DFORMD *

Description: This command will format a disc as CPC6128 Data format.

Note: PCW users should use this command if the disc will also be used on a CPC6128. On the CPC6128 this command has exactly the same effect as DFORM.

DRIVE (DR)

Syntax: DRIVE <drive letter>

Description: Selects the specified drive. This command will accept drives between A and P, and an error message will be given if the requested drive does not exist, or if it does exist but there is no disc in the drive.

Note: If any special drives are installed, such as a hard disc, which use a drive letter other than A, B, C, D, or M, then this command may be used to select the drive.

ERACOPY (ECOPY) *

Syntax: ECOPY <oldname> <newname>
ECOPY <ambiguous filename> (<group>) (<drive>)

Description: This is the same as COPY in every respect but one - if a file already exists with the same name as the file being copied, this file is erased before copying, whereas COPY renames this file as a backup file.

Example: ECOPY B:*.* A

ERASE (ERA)

Syntax: ERA <ambiguous filename>

Description: All files which meet the criteria of the filename will be erased. Wildcards are permitted and the drive letter may be specified as a prefix to the filename.

Note: This is a potentially destructive command and should be used only with care. One very useful version is to use ERASE *.BAK to erase all back up files from the disc in the current drive. ALT-f7 on the PCW (and CTRL-f9 on the CPC) will perform 'ERA *.BAK'.

EXEC (X) *

Syntax: EXEC <filename>

Description: This command causes the contents of the specified file to be treated as if they were being typed in at the keyboard (See Chapter on EXEC Files for full details).

GROUP (USER)

Syntax: GROUP <number>

Description: Selects the specified group number as the one which will be used by CAT, LOAD, SAVE etc.

HELP (H)

Description: Provides a summary of the available commands.

INFO

Syntax: INFO <filename>

Description: The info command provides information about files. The filename used can include wildcards, so INFO *.SRC will give you information on all files with the extension .SRC. The information displayed is the file length, file type (ie. document, program, system..) and its read or write state. RW means Read and Write. RO means Read Only.

INTERNAL (INT) *

Description: Resets the printer output to the normal printer supplied with the PCW range.

PARALLEL (PAR) *

Description: Selects the parallel (Centronics) printer port for the output of all printing.

PAUSE

Syntax: PAUSE

Description: This command is primarily intended for use in an EXEC file. See section "EXEC files".

PRINTON (PRON) *

Description: All output to the screen will also be echoed to the printer, after this command has been used, until PRINTOFF is used to cancel it. One particular use is to provide a printed copy of a disc catalogue. Or, in the monitor to disassemble or list code on the printer.

PRINTOFF (PROFF) *

Description: Cancels the echoing of screen output to the printer, which has previously been initiated by use of the PRINTON command.

PROTECT (PROT)

Syntax: PROTECT <ambiguous filename>

Description: Sets the status of a file or files to 'Read-only'. Wildcards are permitted. Files which have read only status can not be overwritten by subsequent files of the same name. An error message will be given if an attempt is made to do so. Protected files are indicated in the catalogue by an asterisk following the filename. See ACCESS for details of the reverse operation.

Note: PROTECT cannot stop files being erased if the disc is reformatted, or a complete disc is copied onto the disc, either with DISCKIT or the DCOPY command.

RENAME (REN)

Syntax: RENAME <newname> <oldname>

Description: This command renames files on a disc. It does not move or change the file, merely renames it.

Note: If a file requires moving to another disc and renaming, the COPY command should be used and then the original file erased with ERASE.

SERIAL (SER)

Description: Redirects all printed output to the serial interface, for use with a serial interfaced printer.

SPOOL (SPON) *

Syntax: SPOOL <filename>

Description: All output to the screen will also be sent to a file on disc with the specified name until the file is closed with the SPOOLOFF command.

SPOOLOFF (SPOFF) *

Description: Cancels the echoing of all screen output to a file, having first closed the file.

SYMBOL (SYM)

Syntax: SYMBOL <char>,<n1,n2,n3,n4,n5,n6,n7,n8>

Description: The SYMBOL command allows you to redefine a character as it will appear on the screen. The first number following the command is the character to be redefined. Thre eight numbers following are the bytes making up the character. See your Basic manual for more information on SYMBOL as this command is identical.

TYPE (T) *

Syntax: TYPE <filename>

Description: Used to 'type' the contents of an editor or ASCII file to the screen. The file is not loaded into memory, merely the contents displayed on the screen. This can provide a convenient means of viewing the contents of a file without loading it into memory. Whilst the file is being typed pressing STOP will pause the display. Pressing STOP a second time will cancel the command and any other key will resume.

A: (A)

Description: Select drive A as the currently selected drive. Optionally, the colon may be omitted.

B: (B)
C: (C)
D: (D)

Note: D: only in the monitor

Description: Select drive B, C, or D as the currently selected drive. Optionally, the colon may be omitted. Only valid on machines with the requested drive fitted.

M: (M)

Description: Select drive M as the currently selected drive. Optionally, the colon may be omitted. Only valid on the PCW computers.

PHRASES AND FUNCTION KEYS

Phrases are pieces of text which can be stored and used at any time with a single key press. The keys used to recall phrases are the keys marked 'A to Z' on the main keyboard when used in conjunction with the EXTRA key. Function keys are essentially the same, but use the special function keys on their own and in conjunction with the ALT, SHIFT and EXTRA keys.

There are 31 expansion tokens which by default are allocated to the keys EXTRA-A to EXTRA-Z and a number of other keys. Several of these tokens are also allocated to the function keys, duplicating a number of the letters. Some of these tokens are already defined and cannot be changed, leaving 26 tokens which may be defined by the user. By default, many of these tokens are pre-defined to give a variety of European characters, such as 'C, cedilla' and 'AE diphthong', but may be redefined by the user if not required for that purpose.

Each of these tokens can be allocated a string of text or codes up to 200 characters long.

Predefined tokens

The following tokens are predefined by the editor and may not be changed. Each of these selects command mode and executes a command.

PCW	6128	Definition
f1	CTRL-f3	CAT
ALT-f7	CTRL-f9	ERA *.BAK
FIND	CTRL-f2	FIND
EXCH	SHIFT-f2	REPLACE
EXTRA-ENTER	CTRL-ENTER	EXEC EXFILE

Phrases and function key definitions

As far as the editor is concerned, there is no difference between phrases and function key definitions. They are both merely strings (of text or codes) and any difference would be in the use to which they were put, rather than their format. For example, function keys would probably be used to carry out tasks or functions, whereas phrases would be used to store text to be incorporated into documents, though there is no reason why they should not be used for other purposes.

A string has a maximum length of 255 characters, subject to the total buffer size and the free space remaining in it. It may contain any characters and control codes. Any normal text may be typed in from the keyboard as usual, but in order to be able to enter control codes, an escape code must be used to inform the editor that the characters which follow constitute a control code. The escape code used by the editor is the upwards pointing arrow (↑). This is obtained by pressing EXTRA-: (colon) on the PCW. It is used to allow entry of the following:

↑<number>↑	Inserts the code specified by the number. The code may be entered in decimal, eg. ↑13↑, or hexadecimal, in which case it must be prefixed by &, e.g. ↑&0D↑.
↑<letter>	is translated as a control code between 1 and 26 e.g. ↑A would be translated as Ascii code 1, ↑B as 2, etc.
↑↑	is translated by PROTEXT as a single up arrow. This must be used if an arrow is required in the string.

Note: When specifying a code as a number, it must be both prefixed and suffixed with the escape code character (↑), but in other cases, it is only necessary to prefix the character with the escape code. This is because a number could consist of from one to three characters.

As an example of how one would use a control code, if a key was to be defined so that when it was pressed it automatically did a catalogue of drive A followed by a catalogue of drive B, the following string would be used:

CAT A↑13↑CAT B↑13↑

'13' is the code for a carriage return, which would normally be given when the RETURN key is pressed. As CAT requires the RETURN key to be pressed, the codes are inserted into the string. Alternatively ↑M could be used instead of ↑13↑.

Details of the most useful codes are given in an appendix at the end of the manual, but in the unlikely event that a full key translation list is required, this is available from Arnor on request.

Phrase commands

There are two commands which are directly connected with phrases and are used from command mode:

PHRASE (KEY) *

Syntax: PHRASE <letter> <string>
KEY <letter> <string>

Description: PHRASE and KEY are alternative names for the same command. This command allows temporary strings to be created at any time. The command is used from command mode and the letter must be a letter between A and Z, followed by the string of text or codes, which should be wrapped in quotation marks.

Note: If it is required to cancel a key definition, this can be done by using a null string ("") following the key letter in the parameters. This may prove useful when a number of phrases have been defined and the buffer is too full to take any further definitions. Any phrases which are no longer required can be discarded in this way.

LPHRASES (LP) *

Description: Use of this command will list the contents of all the defined phrases between A and Z. Where a phrase contains a code between 0 and 31 or between 192 and 255 this will be displayed in escape code form.

Storing phrases for regular use

The command PHRASE, which enables temporary phrases to be defined has already been described and is very useful for quickly defining a phrase during the course of editing a document, but once the program is left, these phrases will be lost and would require re-entering the next time that the editor was used.

The editor has another method of defining phrases and function key definitions, which enables users to keep one or more files of definitions on disc and to load them as and when required. This is done through the use of an EXEC file.

A set of phrases should be saved with an appropriate name. Only those keys required and their definitions need to be in the phrase file and any existing phrases will not be changed or deleted unless redefined by the new ones. When they are required, it is only necessary to go into command mode and use the following command:

EXEC <filename>

where <filename> is the name of the file containing the phrases. This will automatically allocate them to the specified keys.

Using phrases and function keys

Once a phrase or function key has been defined by either of the above processes it may be used within the monitor or editor at any time by pressing the appropriate key. Any of them may be used either when in edit mode, or command mode. The most convenient arrangement would probably be to use the function keys for commands which would be used in command mode and the keys 'A' to 'Z' for strings of text to be used in documents.

Phrases are called by pressing EXTRA and one of the letter keys between 'A' and 'Z', which gives 26 different possibilities. The function keys may be used either on their own, or in conjunction with SHIFT, ALT, EXTRA and SHIFT-ALT, which gives 20 possible combinations on the PCW.

When a phrase key or function key is pressed, the contents of the string will be entered into the document (if in edit mode), or the command line (if in command mode), as if it had been typed in at the keyboard, and any control codes will be acted upon.

On the CPC6128 the 10 function keys f0 to f9 may be used either with SHIFT or with CONTROL. The phrases are obtained in edit mode by typing CTRL-0 (zero) followed by a letter. Only the function keys may be used in command mode, but these are set up to provide the various European symbols.

Note: Most of the function keys are allocated the same tokens as the keys A to Z and redefining one will also change its equivalent. It is not possible to have different contents in each of them. To set up a function key the KEY command is used with the corresponding letter. A table listing the function keys and the corresponding letters is given as an appendix.

EXEC FILES

What is an EXEC file?

An EXEC file is a file which may contain text, commands and codes and which, when called with the EXEC command, will be read by PROTEXT and the contents treated and acted on, as if they had been entered at the keyboard.

They are created in just the same way as any other text file, but what makes them different is the content of the file and the way it is used later.

Creating an EXEC file

Creating an EXEC file is extremely simple and is done by just typing the required text in, as would be done with any document, but there are a number of special features which permit codes to be inserted into the text, which the editor will understand to be instructions to do certain things.

In addition to ordinary text, any of the editor's command mode commands may be used as well as any valid code between 0 and 255.

Codes must be entered in a special way, otherwise the editor will consider them to be ordinary text. A special 'Escape character' is used to tell the editor that the following character(s) is/are a code and the escape character used is the vertical bar. This is obtained by pressing EXTRA-. (full stop) on the PCW. The escape codes are exactly the same as used in phrase definitions except that the bar is used instead of the arrow.

The easiest way to describe the sort of uses to which an EXEC file might be put, is to give one or two examples. The examples given are intended to show the sort of things that can be done, rather than be particularly useful:

Example to change every occurrence of a certain word to another word in a number of files.

```
L file1
R "buffer" "BUFFER" GA
S|13|
L file2
R "buffer" "BUFFER" GA
S|13|
L file2
R "buffer" "BUFFER" GA
S|13|
L file2
```

Note: When a new line is used, the editor will take this to mean that a carriage return character (CR) is required, as would normally be given by pressing RETURN after entering the command. In the case of the lines concerned with saving, escape characters have been used to insert an extra CR code into the file. The reason for this is that if a file is to be saved with the same name, then RETURN is pressed once after entering the 'S' and again to confirm the same filename.

When complete, the file should be saved with an appropriate name. Entering 'EXEC <filename>', from command mode will automatically execute the file of that name whenever required.

Creating a phrase file

This is easily done by using the PHRASE (KEY) command in an EXEC file. This is an example of a phrase file, to be used to define phrases and function keys in the editor:

```
KEY C ""
KEY G ""
KEY B "This is a remark which can be inserted into the text"
KEY D "The EXTRA key and the appropriate letter should be pressed"
KEY A "CAT A ↑13↑CAT B↑13↑"
```

In the above brief example, keys B and D are straightforward examples of text to be inserted when the appropriate key is pressed.

Keys G and C are defined as null strings. This will have the effect of removing any existing definition from keys C and G. This may be desirable if a second phrase file is being loaded, when a number of keys are already defined, otherwise the phrase buffer may become full before all the new definitions are loaded.

Key A is an example of the sort of definition which would be used for a function key and in this case would perform a catalogue of drive A, followed by drive B when SHIFT-f1/2 was pressed.

It should also be noted that up arrow has been used, rather than the vertical bar, as these commands are simulating entry of the phrases at the keyboard.

Note: Because the EXEC file is executing a command to define a string, it is necessary to specify the CR at the end of the command if one is required when the function key is pressed, as the CR which is implied at the end of each line of an EXEC file will serve only to execute the KEY command.

It is recommended that phrase files should be saved with a suitable suffix to identify them, say '.PHR'.

Commands related to EXEC files

EXEC (X) Execute a file

Syntax: X <filename>

Description: The file specified will be opened for reading and the contents read will be treated as if they were input from the keyboard, until the end of the file is reached, at which time normal operation will continue.

PAUSE Cause the editor to go into a 'waiting' condition.

Syntax: PAUSE

Description: When this command is read by an EXEC file, the program will halt until a key is pressed. Optionally a message will be displayed. This is useful during the course of an EXEC file being executed, as it will permit discs to be changed and messages to be displayed before continuing execution.

Using EXEC files

EXEC files may be used at any time by typing EXEC from command mode, followed by the name of the file to be executed. If a file called 'EXFILE' is present on the currently selected drive, it may be executed at any time by pressing EXTRA-ENTER (CTRL-ENTER on the CPC6128).

There is one further feature which can be very useful. If the 'less than' symbol is used to prefix an EXEC filename when the editor or monitor is called, it will be taken to mean that all input should be taken from the specified file, until the end of the file is reached. For example, 'APED textfile <exfile' would load a text file into memory and then execute the EXEC file, which if required could then carry out operations on the text file, such as replacing text.

APPENDICES

APPENDIX I

Bibliography

Programming the Z80	Rodnay Zaks (Sybex)
CP/M Plus Handbook	Digital Research/Amstrad (Heinemann)
The Amstrad CP/M Plus	Andrew Clarke and David Powys-Lybbe (MML)
The Lord of the Rings	J.R.R. Tolkien (George Allen and Unwin)

Acknowledgement

We thank George Allen and Unwin for permission to quote the extract from "The Lord of the Rings".

APPENDIX II

a) Assembler Directives

BYTE	put byte string in object code
CODE	cancel NOCODE
DB	same as BYTE
DEFB	same as BYTE
DEFL	same as LET
DEFM	same as BYTE
DEFS	same as RMEM
DEFW	same as WORD
DRW	define relocatable word
DS	same as RMEM
DW	same as WORD
ELSE	assemble otherwise
END	end assembly
ENDIF	end IF block
EQU	equate
EXTERN	declare symbols as external
IF	assemble if
IFNOT	assemble unless
IF1	assemble if pass 1
IF2	assemble if pass 2
LET	define symbol
LINK	link in an object file
MACRO	define a macro
MEND	end of defined macro
NOCODE	suppress writing of code
ORG	define code origin
PUBLIC	declare symbols as public
READ	define source file
REPEAT	assemble the following code until expression following UNTIL is true
RMEM	reserve block of memory
STOP	abandon read file
STR	define a string
SYM	save a symbol table
TEXT	same as BYTE
UNTIL	terminate REPEAT loop if expression true
W8080	report non-8080 instructions
WORD	put 2-byte numbers in object code
WRITE	define object filename

b) Assembler Commands

BEEP	sound a beep
DUMP	dump symbol table
INKEY	wait for a key press and return value to a variable
LIST	enable listing to selected output devices
MLOFF	disable macro listing
MLON	enable macro listing
NOLIST	disable listing
OUTPUT	select output listing devices
PAGE	start new page
PAUSE	wait for key press
PLEN	set printer page length
PRINT	display string on screen
TITLE	define title
WIDTH	set printer page width

c) **Assembler Fatal Errors**

1. An ORG directive with an undefined expression.
2. An EQU directive with an undefined expression.
3. An RMEM directive with an undefined expression.
4. An IF or IFNOT directive with an undefined expression.
5. A badly nested IF block.
6. A line longer than 255 characters.
7. The assembler runs out of memory for the symbol table or file buffer.
8. A disc I/O error occurs, e.g. 'disc full', 'file not found'.
9. Stack overflow during symbol dump.
10. Too many nested READs (more than 14).
11. Attempt to nest REPEAT/UNTIL loops.
12. UNTIL without REPEAT.

APPENDIX III

Z80 Instructions

mnemonic	name	operand formats		
ADC	add with carry	A,n	A,r	HL,rh
ADD	add	A,n	A,r	HL,rh
		IX,rx	IY,ry	
AND	and with A	A,n	A,r	
BIT	test bit	b,r		
BRK	MAXAM breakpoint	-		
CALL	call subroutine	nn	cc,nn	
CCF	complement carry flag	-		
CP	compare to A	A,r		
CPD	compare & decrement	-		
CPDR	block compare & decrement	-		
CPI	compare & increment	-		
CPIR	block compare & increment	-		
CPL	complement A	-		
DAA	decimal adjust A	-		
DEC	decrement	r	rr	
DI	disable interrupts	-		
DJNZ	decrement B & jump if not zero	e		
EI	enable interrupts	-		
EX	exchange registers	AF,AF'	DE,HL	(SP),ra
EXX	exchange alternate registers	-		
HALT	halt CPU	-		
IM	set interrupt mode (do not use)	0	1	2
IN	input (do not use A,(n) form)	r,(C)	A,(n)	
INC	increment	r	rr	
IND	input & decrement (do not use)	-		
INDR	block input & decrement (do not use)	-		
INI	input & increment (do not use)	-		
INIR	block input & increment (do not use)	-		
JP	jump	nn	cc,nn	(ra)
JR	jump relative	e	c,e	
LD	load	r,n	r,s	s,r
		A,(nn)	A,(BC)	A,(DE)
		(nn),A	(BC),A	(DE),A
		rr,nn	rr,(nn)	(nn),rr
		A,I	A,R	
		I,A	R,A	
LDD	load & decrement	-		

LDDR	block load & decrement	-	
LDI	load & increment	-	
LDIR	block load & increment	-	
NEG	negate A	-	
NOP	no operation	-	
OR	or with A	A,r	
OTDR	block output & decrement (do not use)	-	
OTIR	block output & increment (do not use)	-	
OUT	output (do not use (n),A form)	(C),r	(n),A
OUTD	output & decrement (do not use)	-	
OUTI	output & increment (do not use)	-	
POP	pop register pair	rp	
PUSH	push register pair	rp	
RES	reset bit	b,r	
RET	return from subroutine	-	cc
RETI	return from interrupt	-	
RETN	return from NMI (do not use)	-	
RL	rotate left	r	
RLA	rotate A left	-	
RLC	rotate left with branch carry	r	
RLCA	rotate A left with branch carry	-	
RLD	rotate left decimal	-	
RR	rotate right	r	
RRA	rotate A right	-	
RRC	rotate right with branch carry	r	
RRCA	rotate A right with branch carry	-	
RRD	rotate right decimal	-	
RST	restart	(see below)	
SBC	subtract with carry	A,r	HL,rh
SCF	set carry flag	-	
SET	set bit	b,r	
SLA	shift left arithmetic	r	
SRA	shift right arithmetic	r	
SRL	shift right logical	r	
SUB	subtract from A	A,r	
XOR	exclusive or with A	A,r	

Key: r means one of A, B, C, D, E, H, L, (HL), (IX+d), (IY+d)
s means one of A, B, C, D, E, H, L
d means an integer in the range (-128,127)
rr means one of BC, DE, HL, SP, IX, IY
rh means one of BC, DE, HL, SP
rx means one of BC, DE, IX, SP
ry means one of BC, DE, IY, SP
rp means one of BC, DE, HL, AF, IX, IY

ra means one of HL, IX, IY
n means a single byte constant
nn means an address or two byte constant
b means a bit number between 0 and 7
e means an address within the range ($-126,$+129) where $
   is the address of the current instruction
cc means one of C, NC, Z, NZ, M, P, PE, PO
c means one of C, NC, Z, NZ
- means no operands (implicit addressing mode)

Note: with the following instructions the first parameter may be omitted if it is A: ADC ADD AND CP OR SBC SUB XOR
e.g. 'OR B' is equivalent to 'OR A,B'

APPENDIX IV

Monitor commands and syntax

All the monitor's commands can be typed on their own and, unless a default value is assumed, you will then be prompted for further input. You can also type all parameters in-line - that is to say following the command.

Where a default value will be assumed that value is shown following the command concerned. Those commands shown followed by a * are only available in the large version of the monitor.

COMMAND		SYNTAX	DEFAULT
ASC	-	-	-
BANK *	-	<number>	1
BLOAD *	BL	<filename>,<address>	-
BOT	-	<address>	&0100
BSAVE *	BS	<filename>,<address>	-
CB	-	<number>	Use cursor, RETURN & ESC/STOP
CLS	-	-	-
COMP *	CP	<start>,<end>,<2nd>	-
CPM	-	-	-
DX	-	<filename>	-
DIS	D	<start>,<end>	-
DSL	-	<length>	0
DUMP *	DM	<filename>	-
EB	-	-	-
EDIT	E/ESC/STOP	<address>	Address in memory window
EC	-	-	-
ER	-	-	-
EU	-	-	-
EXX	-	-	-
FC *	-	<filename>,<start>,<end>	Default start = &0100
FD *	-	<filename>,<start>	Default start = &0100
FILL *	-	<start>,<end>	-
FIND	F	<start>,<end>	You will be prompted for Ascii, Hex or Code, then asked for string, bytes or mnemonics
HELP	H	-	-
HEX	-	-	-
IB	-	-	-
INIT	IN	-	-

IC	-	-	-
IU	-	-	-
JUMP	J	<address>	Program counter
LS *	-	-	-
LIST	-	<start>,<end>	-
LOAD	LD	<filename>,<address>	Default address = &0100
MDIS *	MD	<address>	Address in memory window
MEDIT	ME	<address>	Address in memory window
MP	-	<address>	Previous address
MMOVE *	MM	<start>,<end>,<to>	-
OD	-	<start>,<end>	You will be prompted for the offset
PRINT *	?	<expression>	-
PS *	-	<first>,<last>	A-Z
QB	-	-	-
QUIT	Q	-	-
QUITS	QS	-	-
RC	-	<address>	Program counter
RELOC *	REL	-	You will be prompted for Simple, start, end and data
RESET	-	-	-
RESUME	R	<address>	Program counter
RR	-	<number of times>	Once
SQ	-	<address>	Program counter
SS	-	<address>	Program counter
SAVE	SV	<filename>,<start>,<end>	Default start & end = &0100-LOMEM
SF	-	-	-
SB	-	<addr1>,<addr2>,etc..	-
SC *	-	<expression>	-
S	-	<address>	Program counter
TAIL	-	<string>	-
TOP	-	<address>	Start of BDOS
UB	-	<address>	-
WB	-	-	-
WS	-	<size>	6 on CPC6128, 13 on PCW
XAF	-	-	-
ZR	-	-	-
AF *	-	<number>	0
BC *	-	<number>	0
DE *	-	<number>	0
HL *	-	<number>	0
IX *	-	<number>	0
IY *	-	<number>	0
SP	-	<address>	Default top of stack

PC *	-	<address>	Program counter
A= *	-	<number>	0
F= *	-	<number>	0
B= *	-	<number>	0
C= *	-	<number>	0
D= *	-	<number>	0
E= *	-	<number>	0
H= *	-	<number>	0
L= *	-	<number>	0

APPENDIX V

Useful BDOS functions

Note: (...) indicates the address of.

Function number/name	Input parameters	Returned values
0 System Reset	-	-
1 Console Input	-	A=Char
2 Console Output	E=char	A=&00
3 Auxiliary Input	-	A=Char
4 Auxiliary Output	E=char	A=&00
5 List Output	E=char	A=&00
6 Direct Console I/O	E=&FF/&FE/&FD/char	A=Char/Status/-
7 Auxiliary Input Status	-	A=&00/&FF
8 Auxiliary Output Status	-	A=&00/&FF
9 Print String	DE=(string)	A=&00
10 Read Console Buffer	DE=(buffer)	Characters in buffer
11 Get Console Status	-	A=&00/&01
12 Return Version Number	-	HL=Version (&31 for CP/M+)
13 Reset Disc System	-	A=&00
14 Select Disc	E=Disc Number	A=Error flag
15 Open File	DE=(FCB)	A=Dir code
16 Close File	DE=(FCB)	A=Dir code
17 Search for First	DE=(FCB)	A=Dir code
18 Search for Next	-	A=Dir code
19 Delete File	DE=(FCB)	A=Dir code
20 Read Sequential	DE=(FCB)	A=Dir code
21 Write Sequential	DE=(FCB)	A=Dir code
22 Make File	DE=(FCB)	A=Dir code
23 Rename File	DE=(FCB)	A=Dir code
25 Return Current Disc	-	A=Current disc number
26 Set DMA Address	DE=(DMA)	A=&00
32 Set/Get User Code	E=&FF/User number	A=Current user/&00
33 Read Random	DE=(FCB)	A=Error code
34 Write Random	DE=(FCB)	A=Error code
35 Compute File Size	DE=(FCB)	ro,r1,r2 A=Error flag
37 Reset Drive	DE=Drive vector	A=&00
44 Set Multi-sector count	E=Number of sectors	A=Return code
152 Parse Filename	DE=(PFCB)	See definition below

Note: The DMA is the Direct Memory Address which is set to &80 by default.

FCB stands for File Control Block. An FCB must be set up for each file operation. There are two ways of doing this the first is by using BDOS function 152 (Parse Filename).

Function 152 requires DE to be pointing to a 4 byte block of memory. The first two bytes should point to a string containing the filename, extension and any drive prefix. The string must be terminated by a zero byte. The second two bytes should point to the address at which you wish the FCB to be put. The FCB area should be 36 bytes long.

Example:

```
PFCB            WORD    filename
                WORD    FCB

filename        TEXT "b:myprog.com",0
FCB             RMEM 36
```

The second way you can create an FCB is to do it yourself following this format:

```
BYTE      0     Drive field. 0=default, 1=A, 2=B etc...
BYTES   1-8     Specified filename.
BYTES  9-11     Specified filetype.
BYTES 12-15     All zero.
BYTES 16-23     Password field. If not required set the first byte to
zero.
BYTES 24-31     Reserved. Do not use.
```

If an error occurs function 152 returns &FFFF in HL.

APPENDIX VI

PROGRAMMING WITH CP/M PLUS

Bank organisation

Under CP/M Plus the CPC 6128 or PCW's memory is divided into three separate banks:

Bank 0 is the BDOS bank which contains the banked portions of the BIOS (Basic Input/Output System) and BDOS (Basic Disc Operating System). It also contains the screen memory, the extended BIOS jump block and some disc buffers.

Bank 1 is the TPA (Transient Program Area) bank in which all application programs are run.

Bank 2 contains a copy of the CCP (Console Command Processor), disc hash tables and data buffers. On the PCW it also contains parts of the BIOS. These are not included on the CPC 6128 because many of the routines already exist in its Operating System Rom.

On the PCW there is also the screen environment which is similar to bank 0 with parts of the BIOS and BDOS being replaced by further screen Ram.

The top 16k of each bank is called Common memory as it remains the same in all banks. As well as containing the top part of the TPA it holds the resident portions of the BIOS and BDOS.

The 16k blocks allocated to each bank are shown in figures 1a and 1b.

The CCP is a transient program that the BIOS loads into the TPA at System Cold and Warm start. For further information on transient programs see the section on transient programs a little further on. The Loader program is used by the CCP to handle program loading. This is loaded in at the same time.

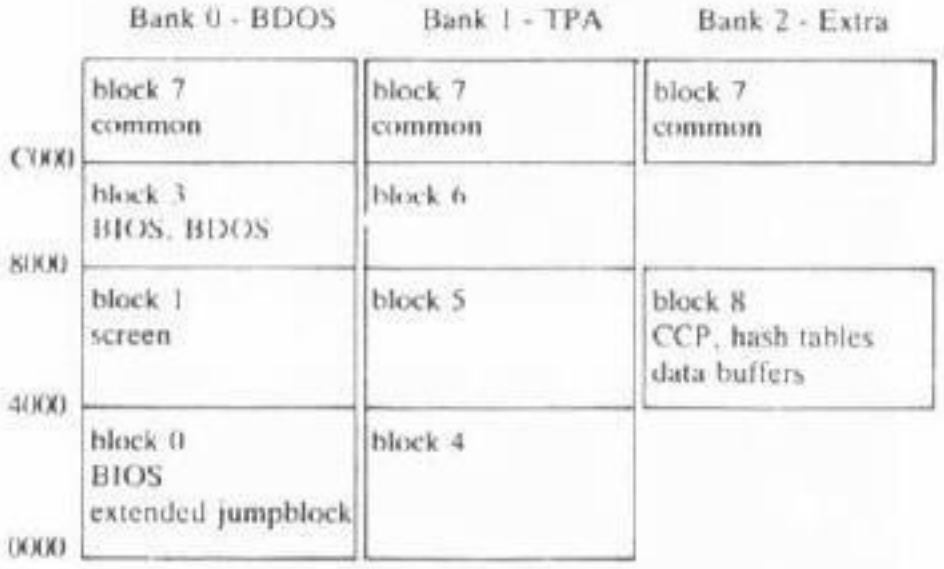

Figure 1a PCW 8256/8512

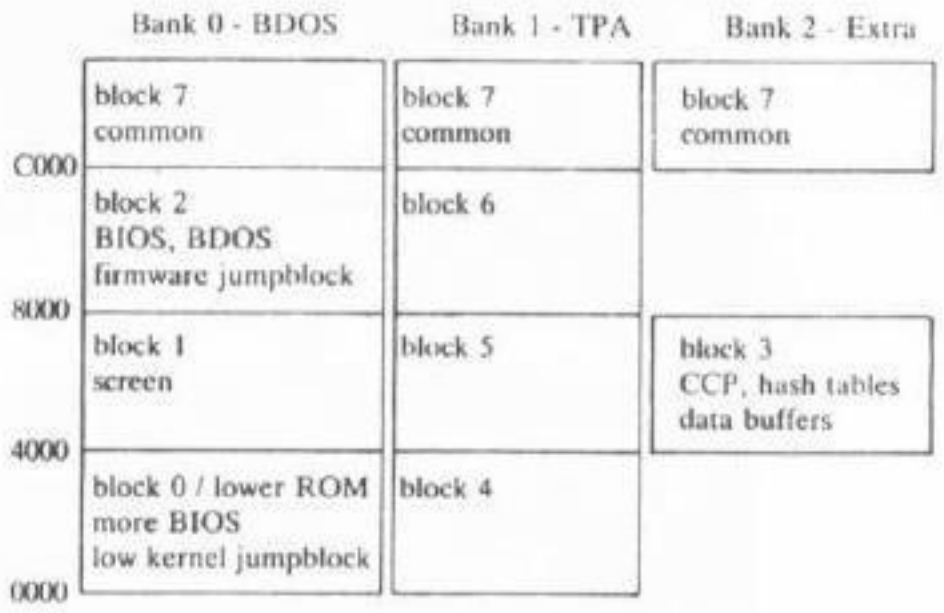

Figure 1b CPC 6128

Cold start

A Cold start is executed immediately the PCW is turned on or CP/M is selected on the CPC 6128. On the CPC 6128 the first sector of a system disc is loaded in and given control. This function is performed by a bootstrap Rom on PCWs.

The bootstrap loads the directory and searches for the first file with the extension .EMS. If such a file is found it is loaded and given control. The .EMS file contains CPM3.SYS, CCP.COM, the Loader and the BIOS.

If the .EMS file is not present on the CPC 6128 the message "Cannot find EMS file" is displayed. On a PCW the machine beeps. In either case you must press a key to restart.

Console Command Processor

The CCP provides access to CP/M Plus facilities when transient programs are not running. It reads command lines typed in and acts on them, loading in transient utilities where required or calling built-in routines.

On selection of a transient program, control is passed first to the Loader and then to the program. On termination of a program the CCP is then reloaded into the TPA and the process repeats.

Transient Program Area

A transient program, as the name suggests, is one that is not system resident, ie. it is loaded from disc each time you need to use it. PIP, DISCKIT and RENAME are transient programs as are word processors, databases and the Maxam II Assembler and Monitor. (Although the Monitor relocates itself to the top of the TPA to allow you to load your own programs at &0100.)

Transient programs generally communicate with the operating system through BDOS function calls. This is done by loading the Z80's registers with the appropriate parameters and calling location &0005.

There are three main ways in which a transient program can terminate execution. These are:

a) A jump to location &0000 causing a warm boot

b) A BDOS system reset call

c) Making a BDOS Chain To Program call

It can also RET to the system if the stack is still intact.

The first two methods pass control to the BIOS warm start entry point which loads the CCP into the TPA and calls it. Chain To Program allows a program to specify the next command to be executed before terminating its own execution by passing this command directly to the CCP immediately after the warm boot.

Resident System Extensions

RSXs (Resident System Extensions) are programs which can be attached to the operating system to modify or extend the BDOS functions. You may rarely need to use them but this short introduction will help you understand what is going on when other programs use them.

When a program is loaded into the TPA any attached RSXs are also loaded. When RSXs are resident the Loader resides directly below the BDOS and and the RSX modules stack downwards from it. The most recently loaded RSX has precedence over any others.

When the CCP regains control after a Warm start it removes all RSXs from memory that have their remove flags set to &FF.

As an example, the GET utility has an attached RSX. When GET.COM is loaded GET.RSX is also loaded with it. GET then performs a number of operations including opening the Ascii file specified in the GET command line. It then makes a BDOS call function 60 to initialise the RSX and then terminates execution. After this, GET.RSX intercepts all console input calls and returns characters from the file specified in the GET command line until it reaches the end of the file. Having done this it sets the remove flag and stops intercepting console input. On the following Warm boot the CCP removes the RSX from memory.

BDOS

As previously mentioned BDOS calls are made through location &0005. The following program is an example of using BDOS calls. It reads characters continuously until it encounters an asterisk and then terminates execution by returning to the system.

```
                BDOS equ &0005

                org &100        ; Start of program

loop            ld c,1          ; BDOS console input function
                call bdos       ; Return character in A
                cp "*"          ; Is it a star?
                jr nz,loop      ; No, try again
                ret             ; Yes, return
                end             ; End of program
```

Command tails

After the CCP has loaded a transient program into the TPA any parameters passed to this program in the command tail are placed at location &0080. So, when you type M2 MYPROG to assemble a program, M2 is loaded into the TPA and the string MYPROG is stored at &0081 with the length of the string at &0080. The character following the last in the command tail is set to 0. Command line characters are preceded by a leading blank and are translated to Ascii upper case.

This allows the assembler to decide what action to take with the command tale. In this case it will assemble MYPROG.

You can include as many parameters as you wish in the command tail up to 128 characters in length. However, if each parameter is a new program it is up to you to update the command tail for each program.

Page zero

Page Zero contains a number of CP/M Plus calls and information for transient programs. The following is a breakdown of the more useful locations in Page Zero.

LOCATION		CONTENTS
From	To	
&0000	&0002	Contains a jump instruction to the BIOS Warm start entry point. The address at location &0001 can also be used to make direct BIOS calls to the BIOS console status, console input, console output and list output primitive functions.
&0005	&0007	Contains a Jump instruction to the BDOS, the Loader or the most recently added RSX. It serves two purposes. Firstly, JP &0005 provides the primary entry point to the BDOS and LD HL,(&0006) places the address field of the Jump instruction into the HL register pair. This value -1 is the highest address in memory available to the TPA.
&0050	-	Identifies the drive from which the transient program was loaded. A value of 1 to 16 identifies drives A - P.
&005C	&007B	Default file control block, FCB, area 1 initialised by the CCP from the first command tail operation of the command line - if it exists.
&006C	&007B	As locations &005C - &007B except for this memory contains the FCB area 2. Note that this area overlays the last 16 bytes of FCB area 1. To use the information in this area a transient program must copy it to another location before using FCB area 1.
&0080	&00FF	Default 128 byte disc buffer. This buffer is also filled with the command tail when the CCP loads a transient program.

APPENDIX VII

THE EXPRESSION EVALUATOR

The expression evaluator is a powerful program used by both the Assembler and Monitor. It allows nested parentheses and indirection as well as logical operators and operator precedence. Numbers can be entered in 3 bases - Binary, Decimal and Hexadecimal. Binary numbers must be prefaced by "%" and Hexadecimal numbers by "&" or "#". During assembly a "$" represents the current program counter.

All expressions are evaluated as a 16 bit number.

The operators provided are, in order of precedence:

1	[ ]	Brackets for nested expressions
	( )	Brackets for indirection
2	*	Multiply
	/	Divide
	MOD	Mod
3	+	Plus
	-	Minus
4	=	Equals
	<>	Not equal to
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
5	<<	Logical shift left
	>>	Logical shift right
6	NOT	Not
7	AND	And
8	OR	Or
9	XOR	Exclusive or

Unless nested in parentheses the expression with an operator of the highest precedence (1=high 9=low) will be evaluated first. So, 5-2*3 is -1 (or &FFFF) not 9 but [5-2]*3 is 9.

The round brackets are used for indirection. For example in the monitor you might want to determine the memory address pointed to by the contents of HL plus the contents of A. To do this you would type PRINT (HL+A).

The reason square brackets have been used for precedence and the round brackets for indirection is to retain the syntax of round brackets used for indirection in Z80 assembly language.

All labels and symbols defined within a program can be included in an expression. This is also the case within the monitor where you can load in a symbol table or define new symbols.

In the monitor you can also use system variable names such as HIMEM and LOMEM and all register names in expressions. The conditional breakpoint routine included in the large version of the monitor makes full use of the expression evaluator to assess a program's state and determine whether to terminate execution or continue. For full information on using the expression evaluator in the monitor see the section "The Expression Evaluator And The Monitor".

APPENDIX VIII

KEY TRANSLATIONS

These are the codes which must be used in a phrase definition to string command sequences together.

PCW	CPC	code	PCW	CPC	code
ALT-@	CTRL-@	0	ALT-[	CTRL-[	27
ALT-A	CTRL-A	1	ALT-.	CTRL-Ω	28
ALT-B	CTRL-B	2	ALT-]	CTRL-]	29
ALT-C	CTRL-C	3	--	CTRL-£	30
ALT-D	CTRL-D	4	--	CTRL-0	31
ALT-E	CTRL-E	5	ALT-<	CTRL-<	218
ALT-F	CTRL-F	6	ALT->	CTRL->	219
ALT-G	CTRL-G	7	ALT-(	CTRL-)	220
ALT-H	CTRL-H	8	ALT-)	CTRL-(	221
ALT-I	CTRL-I	237	ALT-*	CTRL-*	222
ALT-J	CTRL-J	10	ALT-+	CTRL-+	223
ALT-K	CTRL-K	11	ALT-/	CTRL-/	231
ALT-L	CTRL-L	12	ALT-hyphen	CTRL-hyphen	227
ALT-M	CTRL-M	238	ALT-space	CTRL-space	235
ALT-N	CTRL-N	14			
ALT-O	CTRL-O	15	up	up	240
ALT-P	CTRL-P	226	down	down	241
ALT-Q	CTRL-Q	17	left	left	242
ALT-R	CTRL-R	18	right	right	243
ALT-S	CTRL-S	19	SHIFT-up	SHIFT-up	244
ALT-T	CTRL-T	20	SHIFT-down	SHIFT-down	245
ALT-U	CTRL-U	21	SHIFT-left	SHIFT-left	246
ALT-V	CTRL-V	22	SHIFT-right	SHIFT-right	247
ALT-W	CTRL-W	23	ALT-up	CTRL-up	248
ALT-X	CTRL-X	24	ALT-down	CTRL-down	249
ALT-Y	CTRL-Y	25	ALT-left	CTRL-left	250
ALT-Z	CTRL-Z	26	ALT-right	CTRL-right	251
TAB	TAB	9	RETURN	RETURN	13
SHIFT-TAB	SHIFT-TAB	228	SHIFT-RETURN	SHIFT-RETURN	236
ALT-TAB	CTRL-TAB	225	ALT-RETURN	CTRL-RETURN	236
DELr	CLR	16	ALT-CAN	CTRL-CLR	230
SHIFT-DELr	SHIFT-CLR	229	ALT-CUT	CTRL-DEL	232
ALT-DELr		5	ALT-COPY	CTRL-COPY	234
lDEL	DEL	127	[+]	COPY	224
SHIFT-lDEL	--	211	STOP/EXIT	ESC	252
ALT-lDEL	--	212	Enter edit mode		253
SHIFT-COPY	SHIFT-COPY	233	Enter command mode		254

APPENDIX IX

SYSTEM ERROR MESSAGES

This section lists error messages relating mainly to disc operation. These are termed system error messages because they do not relate to any particular program or command, and may occur at any time.

Disc missing or read fail - Retry Ignore or Cancel?

(i) The disc being used has been taken out of the drive. Insert the disc and press R.

(ii) The disc is not formatted.
Press C to cancel the operation. The disc must be formatted using DFORM before it can be used.

(iii) PCW only. This error is given if a CF2DD disc is put into drive A, or the wrong way round in drive B.

(iv) The disc may be faulty or corrupted. Press R to retry. If the error persists try re-formatting the disc.

Drive not ready - Retry Ignore or Cancel?

The disc being used has been taken out of the drive. Insert the disc and press R.

Disc error - Retry Ignore or Cancel?
Seek fail - Retry Ignore or Cancel?
Data error - Retry Ignore or Cancel?
No data - Retry Ignore or Cancel?
Missing address mark - Retry Ignore or Cancel?
Media changed - Retry Ignore or Cancel?
Media change occurred - Retry Ignore or Cancel?

If any of these errors occur the disc may be faulty or corrupted. Press R to retry. If the error persists try re-formatting the disc.

Write protected - Retry Ignore or Cancel?

The write protect tab is pushed in. Remove the disc, slide the tab out, and press R.

Disc unsuitable for drive - Retry Ignore or Cancel?

PCW only. The disc in drive B is formatted as a CF2 disc, and so may only be written to in drive A.

Bad format - Retry Ignore or Cancel?

The disc is not formatted, or is of a non-Amstrad format, or the disc may be faulty.

File is read only

The chosen file cannot be written to because it has been protected. Use ACCESS to unprotect the file.

Directory full

The maximum number of files allowed on a disc has been reached. There may still be room on the disc, so any unwanted files should be deleted.

Disc full

The storage capacity of the disc has been reached. Often this can be remedied by erasing backup files. Type ERASE *.BAK or press ALT-f7 (PCW) or CTRL-f9 (CPC6128).

File not found
File does not exist

There is no file of the chosen name on the disc. Check that the name was typed correctly, that the correct disc is in the drive, and that the correct drive is selected.

Bad filename

The combination of characters chosen as a filename is not allowed. Valid names consists of up to 8 characters, followed optionally by a full stop and an extension of up to 3 characters. Certain characters are not allowed.

Maximum number of files open

There is a limit to the number of files that can be open at the same time. In normal use this limit will never be reached.

Insufficient memory for program

The program has used all the available computer memory. This error will not occur in normal use.

EXEC file read error

A disc error occurred when reading commands from an exec file. This could be caused by removing the disc containing the exec file, or by a faulty disc.

This program will only run under Amstrad CP/M Plus

Arnor CP/M Plus programs will only work on Amstrad computers with CP/M Plus. In particular they will not run on other CP/M systems. This is because special use is made of Amstrad specific features to attain the best performance.

If this message occurs when using an Amstrad computer, which is possible if some other software is installed, all may not be lost. Contact Arnor for help.

GLOSSARY

GLOSSARY OF TERMS

ADDRESS
A number representing the position of a byte in memory.

ARNOR
"The land of the King". In the Third Age of Middle Earth Arnor was known as the "lost realm of the North". The kingdom was re-established by Elessar after the War of the Ring.

ASCII (American Standard Code for Information Interchange)
1. punctuation symbols, etc.
2. The form of representation of a program using no special tokens, only ASCII codes.

ASSEMBLER
1. A program which converts assembly language mnemonics into binary machine code.
2. Another name for assembly language.

ASSEMBLY LANGUAGE
The set of mnemonics which correspond to the operations the Z80 processor is capable of performing.

BANKED MEMORY
The CPC 6128's and PCW's memory is divided into blocks of 16k. Under CP/M Plus these blocks are switched in 4 at a time for each of the 3 bank configurations.

BINARY
The base 2 number system, in which all numbers are represented using just 2 digits, 0 and 1.

BIT
A binary digit, 0 or 1.

BREAKPOINT
A debugging aid. A program stops at a breakpoint allowing you to see whether it is working correctly.

BYTE
8 bits. The unit of memory usually used for data transfer.

CODE ORIGIN
The address of the start of the object code.

CODE LOCATION
While assembling, the address where the next byte of code is to be assembled.

COMMAND
1. An instruction to the assembler which affects the listing in some way.
2. An instruction to the assembler or monitor to do something.

COMPILER
A program which converts a high level language such as C or BCPL into binary machine code.

CONDITIONAL ASSEMBLY
A feature of the assembler which allows code to be assembled differently depending on the setting of variables.

CONDITIONAL BREAKPOINT
A breakpoint which only occurs when a certain condition is true, eg. the contents of A is greater than 48.

CP/M
The operating system used on Amstrad Z80 computers.

DELIMITER
A special character which tells the computer where a string starts and ends.

DIRECTIVE
A instruction to the assembler which affects the object code in some way.

DISASSEMBLE
Convert binary machine code to assembly language mnemonics.

ENTRY POINT
The address to begin execution of a machine code program.

FIRMWARE
1. The operating system.
2. Any program contained in ROM.

HEXADECIMAL (HEX.)
The base 16 number system, where letters A to F represent 10 to 15.

IDENTIFIER
A string of characters which is the name of a symbol.

INSTRUCTION
In the assembler, a Z80 mnemonic or a directive or a command.

LABEL
A symbol which represents a position within a program.

LINK
The operation of joining two separately compiled or assembled programs to form one program.

LISTING
The output produced by the assembler on the screen or printer, showing the source code, object code, and addresses at which the code has been assembled.

MACHINE CODE
A sequence of binary numbers which the Z80 processor interprets as simple operations.

MACRO
A sequence of instructions (for example to an assembler) represented by a single name.

MARKER
In the editor, a pointer to a particular location in the text.

MNEMONIC
A string of characters which represents a Z80 operation.

MONITOR
A program used to help with program debugging.

OBJECT CODE
The machine code program produced by the assembler.

OPCODE
The binary number representing a Z80 operation.

OPERAND
The data which an operation acts on, often a memory address.

OPERATING SYSTEM
The machine code program which accesses the hardware directly and is called by user programs.

RAM (Random Access Memory)
The main memory of the computer which can be written to and read from.

REGISTER
A 1 or 2 byte memory location within the Z80 processor which is accessed very quickly, and is used by Z80 operations.

RELOCATE
Take a machine code program and change the address references throughout it so it will run at a different memory address.

ROM (Read Only Memory)
Memory which can only be read from.

SINGLE STEPPING
A means of tracing the execution of a program step by step showing the contents of memory, flags and registers after each instruction.

SOURCE CODE
The assembly language program, consisting of mnemonics, directives, and commands.

STORAGE LOCATION
While assembling, the address where the next byte of code is to be stored.

STRING
A sequence of characters.

SYMBOL
A variable used when assembling.

SYMBOL TABLE
The list of symbols maintained by the assembler.

Z80
The central processor (CPU) of the Amstrad CPC 6128 PCW 8256 and PCW 8512.

INDEX

INDEX