THE CODE MACHINE

THE CODE MACHINE

The CODE MACHINE consists of one cassette and this manual. The packaging may contain space for a second cassette to allow you to store a back-up copy.

The CODE MACHINE was written for the CPC 464 but is compatible with the CPC 664 with an external cassette recorder.

Copyright 1985 by Picturesque
All rights reserved. This book and the accompanying computer program are copyright. No part of either this book or the accompanying computer program may be reproduced, copied, lent, hired, or transmitted by any means without the prior written consent of the publishers.

Published by:
Picturesque,
6 Corkscrew Hill,
West Wickham,
Kent BR4 9BB.

CONTENTS

INTRODUCTION

SECTION 1 — THE MONITOR/DISASSEMBLER "AMMON"

1.1 General Description
1.2 Loading AMMON
1.3 Access to AMMON
1.4 The prompt and Cursor
1.5 AMMON COMMANDS
- 1.5.1 M — Inspect and change memory contents (Hex)
- 1.5.2 ESC — Escape
- 1.5.3 $ — Inspect and change memory contents (ASCII)
- 1.5.4 I — Insert
- 1.5.5 D — Delete
- 1.5.6 A — Area Move
- 1.5.7 F — Fill
- 1.5.8 P — Print Hex/Graphics Dump
- 1.5.9 Z — Disassembler
- 1.5.10 N — Number Converter
- 1.5.11 E — Access to Editor Assembler
- 1.5.12 Example program for Running and Debugging commands
- 1.5.13 B — Breakpoint
- 1.5.14 J — Jump and Execute
- 1.5.15 K — Breakpoint restore
- 1.5.16 R — Registers display
- 1.5.17 C — Breakpoint continue
- 1.5.18 T — Trace (Single Step)
- 1.5.19 S — Search
- 1.5.20 L — Lower ROM access
- 1.5.21 U — Upper ROM access
- 1.5.22 Y — Return to Basic

1.6 Interrupts
1.7 ROM states and the STACK
1.8 Screen modes
1.9 Keyboard
1.10 The Monitor in practice
1.11 Summary of Commands

SECTION 2 — THE EDITOR/ASSEMBLER "AMMAS"

2.1 General description
2.2 Loading AMMAS
2.3 Access to AMMAS
2.4 THE EDITOR
- 2.4.1 AMMAS COMMANDS
- 2.4.2 Screen display
- 2.4.3 Entering a line of Source Code
- 2.4.4 LIST
- 2.4.5 EDIT

2.4.6 AUTO
2.4.7 RENUM
2.4.8 DELETE
2.4.9 COPY
2.4.10 NEW
2.4.11 CLEAR
2.4.12 MODE 1
2.4.13 MODE 2
2.4.14 BASIC
2.4.15 External commands
2.4.16 COPY CURSOR
2.4.17 COMMENTS
2.4.18 ESC
2.4.19 Resetting the Keyboard
2.4.20 MONITOR access
2.5 CASSETTE AND DISC Commands
2.5.1 FILE
2.5.2 SAVE
2.5.3 LOAD
2.5.4 VERIFY
2.5.5 External DISC Commands
2.6 THE ASSEMBLER
2.6.1 The Object Buffer
2.6.2 Numbers
2.6.3 ASCII Characters
2.6.4 Arithmetic
2.6.5 LABELS
2.6.6 Label slicing
2.6.7 JR/DJNZ
2.6.8 Assembling Source Code
2.7 ASSEMBLER DIRECTIVES
2.7.1 ORG
2.7.2 END
2.7.3 EQU
2.7.4 DEFL
2.7.5 DEFB
2.7.6 DEFW
2.7.7 DEFS
2.7.8 DEFM
2.7.9 PRNT
2.7.10 ENT
2.8 MULTI-SECTION SOURCE FILES
2.8.1 Saving a section of Source Code
2.8.2 Loading a section of Source Code
2.8.3 Verifying a section of Source Code
2.8.4 Assembling a multi-section Source file
2.8.5 Assembly from Disc/Tape to Disc/Tape
2.9 ASSEMBLER ERROR MESSAGES
2.10 Summary of Commands

SECTION 3 — USING "AMMAS" and "AMMON"

SECTION 4 — Z80 MNEMONICS

INTRODUCTION

The CODE MACHINE is a full feature Machine Code programming development package consisting of an Editor/Assembler and a separate Monitor/Disassembler. Both programs are entirely self-contained, and are written entirely in Machine Code.

The Editor/Assembler (called **"AMMAS"**) will allow you to enter and edit Source Code listings (mnemonics and label names); to produce Object Code (the actual Machine Code); to print your Machine Code listings onto a printer; and to Save and Load Source Code or Object Code.

The Monitor/Disassembler (called **"AMMON"**) gives you all the commands you need to test and debug your Machine Code programs, as well as facilities to investigate the inner workings of your Amstrad computer.

You will find that both programs are very quick and easy to operate, which makes the CODE MACHINE an ideal programming aid for beginners and for experienced Machine Code programmers alike.

If you are new to Machine Code programming, you will need to purchase a book to teach you how to write in Machine Code — and there are several available to choose from. Machine Code is not an easy language to write, and it is beyond the scope of this manual to teach the subject, but, used in conjunction with a good book, the CODE MACHINE will help you to understand how Machine Code works, and where you are going wrong.

But, to get the most from programming your CPC 464 in Machine Code, we strongly recommend that you purchase "The Concise Firmware Specification" from AMSOFT (SOFT 158) as this will give you all the information you will need about the computer for simple and effective Machine Code programming. Indeed, without this mine of information, your Machine Code programs will be very limited and will never be able to use the many excellent features of the CPC 464.

SECTION 1

The MONITOR DISASSEMBLER "AMMON" Version 1.1

1.1 General Description

The MONITOR allows you to inspect or change the contents of memory locations, and works entirely in Hex, which is the most logical number system to use in Machine Code. A display of memory is also available in ASCII characters to allow you to search for messages or to enter messages.

One of the biggest problems with Machine Code programming is tracking down errors, since there is no built-in error detection as there is in the Basic interpreter. The MONITOR helps you over-come this by allowing you to insert a Breakpoint in your Machine Code, which temporarily stops your program at that point when it is run. Having encountered a Breakpoint, you can use the MONITOR'S commands to check that the values in the CPU registers are correct, that data bytes have the correct values and that your program has not overwritten or corrupted itself.

The MONITOR will give you a display of the contents of all the CPU registers at the Breakpoint, and allow you to disassemble your program (or any part of memory including either ROM). A more precise but more time-consuming method of error detection is available with the Single Step feature. Each instruction is executed in isolation, with a comprehensive screen display of the machine status: You can even Single Step through the ROM routines. Either the Upper or Lower ROM can be enabled or disabled, and can be Disassembled or Single Stepped through. Memory management commands allow you to move the contents of a block of memory to a new location; to insert or delete a number of bytes within a block without re-typing the rest of the block; to fill a specified area with any value; and to search for up to 10 consecutive ASCII characters, or up to 5 consecutive Hex bytes.

To simplify any interchange with Basic, a Dec-Hex-Dec number converter is included, along with printer routines for the disassembler. The MONITOR is totally compatible with Basic, allowing you to return to Basic at any time. Machine Code programs can either be run from Basic, with the CALL function, or from the RUN command within the MONITOR.

The Monitor is fully relocatable in memory so that it can be loaded into the most convenient location for the code you are working on. To allow easy access from Basic, the Monitor is set up as an RSX so that it is accessible via an External Command, and it allows direct access to the Assembler when both are in memory together.

1.2 Loading AMMON

As the Monitor operates in memory as an RSX, it is advisable to load it into a clean machine, so clear the CPC464 by CTRL + SHIFT + ESC. If you have Discs attached then enter the direct command **|TAPE.IN** (ENTER) and type **RUN"** (ENTER) to Load the Monitor. AMMON gives you the option of making a Back-up copy onto Disc or Cassette, and if you have Discs attached, you should enter the direct command **|TAPE.OUT** if you require a cassette back-up. You should also set the cassette write speed from Basic before Loading the program.

AMMON is fully relocatable, and a Basic loader program loads first and asks you

for the start address for the Monitor. Type in this address either as a decimal value or as a Hex value (preceded by '&') and press ENTER. HIMEM will be set to one byte below the entered address, and then the Code will be loaded in. At this point you will be asked whether you want to make a Back-up copy. The program is then run to actuate the relocator. The relocate routine returns to Basic with the normal 'Ready' message.

The lowest address you can normally Load to is &1321 (Hex). However, if before loading the Monitor, you permanently reserve the Disc/Tape buffer, you can force a much lower minimum loading address. To do this, execute the following Basic commands:

OPENOUT "DUMMY":MEMORY HIMEM-1

As HIMEM and the Disc/tape buffer will not now be consecutive in memory, Basic cannot recover the buffer after use, and the Monitor will also use this buffer when it loads its own code. Be aware when using the Monitor that the buffer still exists as far as Basic or your machine code is concerned.

The Basic loader for the Monitor does not contain a SYMBOL AFTER 256 command, so if you need to define other graphic characters after the Monitor has loaded, you should clear the definable graphics with SYMBOL AFTER 256 before loading the Monitor, otherwise HIMEM and the graphic definitions will separate, and Basic will not be able to recover the graphic area of memory. If you see a 'Memory Full' error when you load the Monitor, it is likely that SYMBOL AFTER is the cause, or that you have specified too low an address for Loading.

To locate AMMON to the highest available address, do not enter a start address but simply press ENTER. In this case, AMMON will position itself immediately under the current HIMEM, resetting HIMEM to underneath itself. (This is useful when using the Assembler and Monitor together or when other ROM or RSX software is required).

There is no check made on the Start Address that you specify, and the machine will crash if you overwrite Static Variables, Jumpblocks etc.

AMMON is totally self-contained and not dependant on any Basic. Having loaded and relocated, you can now Load any Basic or any other Machine Code that you require, but remember that if you move HIMEM to a higher address than the Start Address given on Loading the Monitor, you run the risk of corrupting AMMON.

Loading a back-up Copy

To Load your back up copy of AMMON, type RUN "MON" and follow the screen prompts. You will not be offered a further back up option.

WARNING

THIS PROGRAM IS COPYRIGHT AND ONE COPY ONLY may be made for your OWN PERSONAL USE.

It is illegal to sell copies or to give copies to friends.

N.B.
For the purposes of the examples in this part of the manual, it has been assumed that AMMON has been loaded to the highest available part of memory on Loading.

1.3 Access to AMMON

Having Loaded and relocated, you can access AMMON with the External Command **| MON,** or by **CALL nnnn,** where nnnn is the Start Address supplied on Loading. You will be greeted with the screen message "SPACE for MONITOR". On entry to AMMON, you will always see this message along with everything else that is currently on the screen.

Pressing SPACE will clear the screen and allow entry to the Monitor with the Prompt and Cursor displayed on the bottom line.

1.4 Prompt & Cursor

The prompt () indicates that the MONITOR is waiting to be put into a command mode. It does not appear at the start of every line of the display, but only appears when a command routine has ended, and the MONITOR is waiting for a new command instruction. The cursor is visible for the majority of the time, and indicates a request for a keyboard entry, and shows where the result of that keyboard entry will be displayed on the screen.

1.5 AMMON COMMANDS

The range of commands offered by the MONITOR is as follows:—

M	Display a memory location & its contents in HEX, and change its contents.
$	Display a memory location in HEX and its contents in ASCII, and change its contents.
ESC	Escape from a command mode to the Prompt & Cursor.
I	Insert up to 255 bytes into a block of memory.
D	Delete up to 255 bytes from a block of memory.
A	Move the contents of an area of memory to a new location.
F	Fill a specified area of memory with a given byte value.
P	Hex/Graphics memory dump to Screen/Printer.
Z	Disassemble to Screen or Printer.
N	Number conversion: Hex to Dec and Dec to Hex.
E	Access to Editor Assembler (if loaded).
B	Set a Breakpoint.
J	Jump to a specified address and execute the code there.
K	Restore code after a Breakpoint.
R	Register display.
C	Continue execution of program after a Breakpoint.
T	Trace (Single Step) with front panel display.
S	Search for Hex or ASCII values in memory.
L	Lower ROM enable/disable.
U	Upper ROM enable/disable.
Y	Return to Basic.
#	Select screen stream.

All command modes are accessed by a single keystroke denoted by the letter in the left hand column above. All keyboard entries are checked for validity, and at any given time only the permitted entries are accepted. Any other key press is rejected and the keyboard is scanned again.

All numeric inputs and displays are in **HEX** to facilitate the entry and inspection of Z80 Op-codes. The number conversion command simplifies any exchange between Hex and Decimal.

All references to addresses and their contents in these instructions will be in **HEX,** and will be shown as a two or four figure number with no prefix or suffix.

1.5.1 M — Inspect & change memory contents (Hex)

The screen should show the Prompt & Cursor on the bottom line. If this is not so, then press **ESC.**

Type M

An inverse M will appear immediately to the right of the prompt, and the cursor will move along the bottom line by one character space.

Now type in the HEX value of the address that you wish to inspect, say 6000. (You can use the MONITOR to inspect or change any memory location in RAM or ROM, although you cannot alter the ROM values.)

Do not worry if you find different values in these memory locations from those shown.

The address appears on the screen as you type it in, and as soon as you have typed the fourth digit, a two character Hex number appears on the screen to the right of the address, showing the Hex value of the contents of that memory location.

```
>M6000 00 ■
```

The cursor has now moved on, leaving a space after the two contents digits.
Now type FF.

```
>M6000 00 FF ■
```

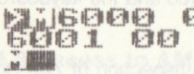
This has loaded the Hex value FF into memory location 6000. The value is loaded into the location automatically after you type the second digit.
Now type ENTER.

```
>M6000 00 FF
6001 00 ■
```

The original line with the prompt has scrolled up one line, and 6001 00 is now displayed on the bottom line. At all times, the MONITOR operates with a scrolling screen, and new information is always displayed on the bottom line.

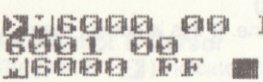
Typing ENTER displays the next memory location and its contents. To enter a value in this new address, enter the two HEX digits, or type ENTER again for the next memory location.

Now let's check that FF really has been loaded into address 6000.
Type M



The screen has scrolled up one line, and the inverse M has reappeared on the bottom line.

Typing M after a HEX address, data entry, or ENTER, allows you to re-enter the M command routine at the start. To re-enter the M command in the middle of typing a Hex address, press **ESC** to escape, and **M** to enter the M command.
Now type 6000



The contents of 6000 are shown as FF.

To summarize the M command so far, you can sequentially step through memory locations using ENTER and change the value of the contents of each address, or, by typing M again, you can define a new starting address.

To re-enter the M command with a partly typed address on the screen, press **ESC** to escape to the Prompt, and then type **M** to enter the M command.

Now type 00 to clear the contents of address 6000. You will notice that the cursor is still visible on the bottom line of the screen to the right of the 00 just entered. Although address 6000 now contains the value 00, (the value was changed immediately you typed the second 0) you can change it again if you wish.

So, if you enter an incorrect value, and realise what you have done before you press ENTER, you can correct your mistake without having to specify the address again. In fact, you can make only two attempts at entering a value before the routine returns you to the prompt and cursor, when you will have to type M to re-enter the M command.

Remember, that in the M command, ENTER **only** has the effect of stepping on to the next memory location, whereas in other commands (where applicable) ENTER causes the operation to be executed.

Imagine that you have just entered a machine code routine from, say 6000 to 6200, and you realise that you need to change the value of one byte somewhere near the middle of the routine, but you do not know the precise address. You could spend a long time guessing addresses and looking at their contents, or repeatedly pressing ENTER until you find the right byte. But if you type M, to re-enter the M command, and look at the contents of an address somewhere near the beginning of the routine, then press ENTER, **and hold it pressed,** after about 1½ seconds the screen will start scrolling quite fast, and will rapidly display successive locations until you release ENTER. In this way you can quickly scan through a routine until you find the byte you are looking for. To effect the alteration, having released ENTER, you will again have to type M xxxx to re-enter the M command at the correct address, and then change the contents of that address.

1.5.2 ESC — Escape from a command

ESC allows you to escape from a command mode, and returns you to the Monitor's Prompt & Cursor.

Press **ESC**

The screen will scroll and the Prompt & Cursor will reappear on the bottom line of the screen. You can now enter any of the Monitor commands. All command routines, with the exception of R (Register display) and K (Breakpoint clear), will accept ESC to escape at any time up to the point of execution.

1.5.3 $ — Inspect & change memory contents (ASCII)

This command operates in a similar fashion to the 'M' command, but allows you to enter text directly from the keyboard. It is by no means a word processor, but it offers a much simpler method of text entry than by converting letters to their character codes, and entering the codes individually with the 'M' command.

The command takes the form: **$ aaaa** where **$** is the command mode, and **aaaa** is the starting address of the text block.

Let us enter a simple message into a free area of RAM.

Type $ (Shift and '4') to enter the $ command.

An inverse $ will appear on the bottom line of the display.

Type 6100

The address is displayed as normal.

As you type in each letter of the message, it is displayed on the screen, to the right of the address and its present contents, and the character code is stored in that address. The screen scrolls automatically, displaying the next address and its present contents. You **do not** need to press ENTER to access the next address.

If there is a valid character code in an address, it will be displayed between the address itself and the cursor, otherwise a question mark is displayed.

All upper and lower case characters can be entered by use of the SHIFT keys. The only exception is $ which is reserved as the character to access the command.

Graphics and User Defined Characters cannot be entered directly.

In other words, any single character that appears on a key top and that can be accessed by a single key press or by **one** level of shift can be entered.

Now type in the following message:

This is "AMMON"

(Use SHIFT + 2 for the " marks). Having typed it in, mistakes included, now review the message.

Type $	(Shift and 4) to re-enter the $ command.
Type 6100	the start address.
Type ENTER	and hold it pressed until the whole message is on the screen.

If the message is correct, you can now press ESC to return to the Prompt.

If you have made a typing error, or if you want to put another message at a new starting address, type $, to re-enter the $ command at the beginning, in the same way that the 'M' command is re-entered. You will have to enter the new address before making your correction, or starting you new message.

The repeating keyboard works as in the 'M' command, to allow you to review a message quickly.

1.5.4 I — Insert

If, having written a machine code routine, or having entered Data into memory, you find it necessary to add extra bytes in the middle of that routine, the Insert command allows you to insert up to 255 bytes at any point, and automatically moves up memory a specified area of RAM by the number of bytes that you wish to insert.

The Insert command takes the form **I aaaa bbbb nn** where **I** is the Insert command mode, **aaaa** is the Hex address of the first byte of the insertion, **bbbb** is the Hex address of the highest byte in RAM of the block of memory to be moved, and **nn** is the number of bytes to be inserted, in Hex.

Example

Press ESC to restore the prompt and cursor to the bottom line, and then, using the M command, enter the following consective values into memory:

```
6000 00
6001 01
6002 02
6003 03
etc.
  .
  .
  .
  .
600A 0A
600B 0B
600C 0C
600D 0D
600E 0E
600F 0F
```

(The values entered into these locations are purely for a demonstration of the Insert and Delete commands, and, if run, will probably cause the CPC464 to crash.

In the above example, the start of the imaginary routine is 6000 and the end is 600F. We will now insert 5 bytes, the first new byte to be at 6004.

Press ESC	to restore the prompt and cursor.
Type I	to get into the Insert mode.
Type 6004	the address of the first byte of the insertion.
Type 600F	the address of the highest byte to be moved.
Type 05	the number of bytes to be inserted (Hex).

At any time up to this point, you can press ESC to escape from this command mode, as no change to RAM has occured yet. Indeed, if you make a typing error at any time, you must press ESC and start the command again.

If you have entered the Insert example correctly, you can now press ENTER to effect the insertion. The screen will scroll up one line, and the prompt and cursor will return to the bottom line. The insertion has been completed.

Using the M command, check through the 21 locations from 6000. Addresses 6004 to 6008 inclusive will now contain the value 00, and 6009 to 6014 will contain the values 04 through to 0F.

When using the Insert command on machine code routines, any absolute addresses in the remainder of the routine that referred to the area that has been moved will need to be changed to maintain correct operation of the routine.

1.5.5 D — Delete

This command has the opposite effect to Insert, and takes the form **D aaaa bbbb nn,** where **D** is the Delete command mode, **aaaa** is the address of the first byte to be deleted, **bbbb** is the address of the highest byte to be moved down RAM, and **nn** is the number of bytes to be deleted.

Assuming that the result of the Insert example is still in memory,let us now move the area of RAM from 6009 to 6014 back to its original place.

Press ESC	to restore the prompt and cursor.
Type D	to enter the Delete command.
Type 6004	the start of the area to be deleted.
Type 6014	the end of the area to be moved.
Type 05	the number of bytes to be deleted (Hex).
Type ENTER	to effect the deletion.

The prompt and cursor will reappear on the bottom line, and the deletion will be complete.

Now check through addresses 6000 to 6014, using the M command. The contents of these locations will be as they were **before** the Insert example, and locations 6010 to 6014 will have been loaded with the value 00.

Again, any absolute addresses relating to the area of RAM that has been moved by Delete, will now need to be changed.

1.5.6 A — Area Move

This command will Block Move a specified area of RAM, and takes the form **A aaaa bbbb cccc** where **A** is the Area Move command; **aaaa** is the present start address, **bbbb** is the

present end address of the area to be moved; and **cccc** is the new starting address.

Assuming that the example used for the I and D commands is still in memory, let us now move the whole area from 6000 to 600F up memory, to start at 6200.

Press ESC	to restore the prompt and cursor.
Type A	to enter the Area Relocate mode.
Type 6000	the start address of the area to be moved.
Type 600F	the end address of the area to be moved.
Type 6200	the new start address.
Type ENTER	to effect the move.

The screen scrolls up one line, and the prompt and cursor return to the bottom line. The move is complete.

Using the M command, check that addresses 6200 to 620F have been loaded with the same values as those still remaining in addresses 6000 to 600F.

This routine will allow you to move up or down memory, from any original starting address to any new starting address, even if the new area overlaps the original area. The original area (unless over-written by the move) is not changed.

Try moving the area from 6000 to 600F to a new start address of 6008 and then move it back again.

A word of warning. Most AMMON commands that allow you to alter the values in memory locations will also operate on the area of memory containing AMMON. Always check carefully that you are not about to overwrite AMMON which uses nearly 7K from the Start Address given on loading. See Section 1.10 for more details of how much memory is used by AMMON.

1.5.7 F — Fill

The Fill routine allows you to enter the same byte value into a given area of RAM, and takes the form: **F aaaa bbbb xx,** when **aaaa** and **bbbb** are the start and end addresses respectively of the specified area, and **xx** is the value to be entered.

Press ESC	to restore the prompt and cursor.
Type F	to enter the F command.
Type 6020	the start address.
Type 6100	the end address.
Type AA	the value to be entered. (Hex).
Type ENTER	to effect the Fill.

The screen scrolls, and the prompt and cursor appear on the bottom line of the screen. The fill is complete.

Now use the M command, with ENTER kept pressed, to verify that each byte from 6020 to 6100 **inclusive** has a value of AA.

1.5.8 P — Print Hex/Graphics Dump

This command allows you to produce a Hex Dump of the contents of any section of memory onto the screen or printer, or to produce a graphics/ASCII Dump to the screen only. It takes the form **P aaaa bbbb** where **P** is the command name; **aaaa** is the Hex Address of the first byte to be printed; and **bbbb** is the Hex address of the last byte to be printed. The display is shown thus:

```
P 0000 0017
  Hex or $ H
PRINTER? (Y/N)
0000 01 89 7F ED 49 C3 80 05
0008 C3 82 B9 C3 7C B9 C5 C9
0010 C3 16 BA C3 10 BA D5 C9
```

Each line shows the Hex contents of eight successive locations, with the Hex address of the first byte shown in the left column. The routine will only print complete lines, and if the end address that you specify is part of the way along a line, it will print up to the end of that line.

If the Prompt is not visible on the bottom line of the screen, press ESC, otherwise access the command by pressing **P** followed by the address from which you want to start the display. If you want an open-ended screen display only, then press ENTER; but if you want Printer output, you must define the address at which the display will stop. Enter that Hex address following the start address, and press ENTER.

You will now be asked to select Hex or $ (ASCII) by pressing either 'H' or SHIFT + 4. If you only specified a start address, both options will only give a screen output of up to 16 lines and then wait for ENTER to continue. If you typed a start and an end address, the Hex option then asks for printer requirements, but the $ (ASCII) option will only give a screen display as ALL 256 ASCII and Graphic characters are displayed, and most printers will not cope with the graphics or will try to interpret the control codes.

If you want a printer output of the Hex Dump, press **y** in response to the question **PRINTER (Y/N).** While using the printer, you can stop the command before the specified end address by pressing ESC at any time.

1.5.9 Z — Disassembler

This command will disassemble any part of RAM or ROM, either to the screen or to the Printer. It provides a display that includes the Hex address of the first byte of the instruction, the Hex values of the bytes that relate to that instruction and the Z80 mnemonic for that instruction. The full set of Z80 mnemonics can be disassembled.

For access to the ROM contents, see section 1.5.20 and 1.5.21 on ROM enabling.

The command takes the form: **Z aaaa bbbb** where **Z** is the command mode, **aaaa** is the hex starting address and **bbbb** is the hex end address of the part of memory you wish to disassemble.

Type Z	to access the command.
Type 0000	the start address.

You can now either type in an end Hex address where you want the disassembly to stop, or you can press ENTER to achieve an open-ended disassembly in the same way that the **P** command operates.

If you specify an end address, you will be given the same Screen/Printer options as described in the **P** command above. If you attempt to disassemble any part of AMMON, an error message **"INVALID ADDRESS"** is displayed.

The disassembly will appear as follows:

```
>0000 0020
PRINTER? (Y/N)
0000  01897F     LD    BC,7F89
0003  ED49       OUT   (C),C
0005  C38005     JP    0580
0008  C382B9     JP    B982
000B  C37CB9     JP    B97C
000E  C5         PUSH  BC
000F  C9         RET
0010  C316BA     JP    BA16
0013  C310BA     JP    BA10
0016  D5         PUSH  DE
0017  C9         RET
0018  C3BFB9     JP    B9BF
001B  C3B1B9     JP    B9B1
001E  E9         JP    HL
001F  00         NOP
0020  C3CBBA     JP    BACB
  ENTER for more; <ESC> for END
```

When using the screen display, 16 lines of disassembly are shown followed by the message:

ENTER for more, ESC for end

Pressing ENTER will display the next 16 lines unless the end address is reached, when the Prompt & Cursor will be returned.

Pressing ESC in response to the above message will also return you to the Prompt & Cursor.

If you are disassembling to a printer, the routine continues uninterrupted until it reaches the end address. The printer can be stopped early by pressing ESC at any time.

All disassembled addresses and values are in Hex. Relative jumps show the address to which the jump will go, with the offset value shown with the hex coding for that instruction.

In both the **Z** and **P** commands, if the printer is not connected, is off line or otherwise appears BUSY, then the printer option fails after 3 seconds and the normal prompt and cursor reappear.

1.5.10 N — Number Converter

This routine will convert Hex numbers to Decimal or vice versa.

Type N The Number command.

The screen will display:

NUMBER H/D?

Type H (for Hex) or D (for Decimal) to indicate the number system of the number you wish to convert.

Type H to convert a Hex number to Decimal.

The screen will scroll, and show 'H' (followed by the cursor). Now enter **four** Hex digits. (You must enter leading zeros when entering a Hex number).

Type 4000
Type ENTER

The display will now show:

H 4000 = 16384

with the prompt and cursor on the bottom line.

To convert from Decimal to Hex, Type D instead of H in response to "NUMBER H/D?", and enter your decimal number, **without** leading zeros. Again type ENTER to produce an answer.

1.5.11 E — Access to Editor Assembler

Press ESC to restore the Prompt & Cursor to the bottom line of the screen and then type **E** followed by ENTER. If the Assembler is resident in memory it will be accessed, otherwise the Monitor's Prompt & Cursor is displayed. A full check of the Assembler is impossible, and it is up to the user to ensure that the Assembler and its associated tables and buffers have not been overwritten by Monitor commands. If in doubt, you should reload the Assembler from Disc or Tape.

1.5.12 Example program for running & debugging commands

To demonstrate the next five Monitor commands, use the **M** command to enter the following short program at 6000 Hex, or at any other convenient location.

N.B.
If you have loaded AMMON on its own into your CPC464 and have positioned it as high in memory as possible by pressing ENTER in response to "Start Address?" when loading, then you should find that the memory locations at 6000 Hex will be free to accept this example.

```
6000    01 00 00    LD   BC,0000   ;  Clear BC
6003    11 00 00    LD   DE,0000   ;  Clear DE
6006    21 00 00    LD   HL,0000   ;  Clear HL
6009    03          INC  BC        ;  BC = BC + 01
600A    13          INC  DE        ;  DE = DE + 01
600B    23          INC  HL        ;  HL = HL + 01
600C    C9          RET            ;  Return
```

Start 6000
End 600C

Having entered Hex codes, go back to 6000 and check that the codes are correct. (Type M 6000 and check the contents of each location).

It is recommended that you would normally Save a machine code program before running it in case it crashes, which it is certainly likely to do unless you are an experienced machine code programmer. In this case, there is no real point in Saving the program, but if you wish to do so, refer to the section on "The Monitor in practice".

The last line of the routine is a RETURN instruction which it is usual to use at the end of a Machine Code routine, to return you to the Calling program (usually Basic).

When you are satisfied that you have entered the above program correctly, press ESC to restore the Prompt & Cursor.

1.5.13 B — Breakpoint

This command allows you to temporarily interrupt a machine code program at any point, and return control to the MONITOR, so that you can inspect the values in the CPU registers, and in RAM, and make corrections as necessary.

It takes the form **B aaaa,** where **B** is the Breakpoint command mode, and **aaaa** is the address of the instruction that the break will replace. (aaaa must be the address of the **first** byte of a multi-byte instruction).

The Op. Codes in the three addresses **aaaa, aaaa + 1,** and **aaaa + 2** are automatically stored by AMMON, and these addresses are then loaded with the values **CD OD BF** which constitutes a CALL to the entry point of AMMON. It must be a CALL to maintain correct operation of the Stack.

On entering AMMON at this address, the values in the CPU Registers are stored within AMMON; the Stack Pointer is set to the Monitor's Stack; and the message "SPACE for Monitor" is displayed on the bottom line of the screen, in addition to the screen display that your program has created. The Monitor will now wait until you press SPACE, when it will clear the screen and display the Prompt & Cursor.

You will now be able to use any of the MONITOR commands to check or alter the routine, before returning control to the routine at the point at which the break occured. As the MONITOR uses its own integral Stack separate from the Program Stack, there is no danger of over-writing the Program Stack during a Breakpoint.

Before running the example in section 1.5.12 enter a Breakpoint at address 6009. This will have the effect of stopping the program after the Registers BC DE and HL have been cleared, but before they are incremented.

If the Prompt is not visible on the bottom line of the screen, press ESC, otherwise

Type B	the breakpoint command mode
Type 6009	the breakpoint address

There is no need to type ENTER, as the Breakpoint is set after typing the fourth digit. The screen will scroll, and the prompt will appear on the bottom line.

Using the M command, check that 6009 to 600B now contain CD OD BF in place of 03 13 23, the latter having been stored for later replacement.

1.5.14 J — Jump & Execute

The Jump command allows you to jump out of the control of the MONITOR to the starting address of any routine that you write, and it takes the form **J aaaa** where **J** is the Jump command mode, and **aaaa** is the start address of your program.

You can run your Machine Code programs either with the Monitor's **J** command, or by returning to Basic and using the CALL command. Either way, all the Monitor's facilities are available to you after a Breakpoint.

In this example, we will use the 'J' command.

Press ESC	to restore the prompt and cursor
Type J	to enter the Jump command.
Type 6000	the start address.
Type ENTER	

The screen is cleared and the routine will run, and then access the Monitor with the screen message "SPACE for Monitor".

The sequence of events on executing a **J** command is:

i) the screen is cleared
ii) your program's screen Mode is set
iii) Interrupt status is restored
iv) the Stack Pointer is set to the program Stack
v) the start address is put into the Program Counter and the program is executed.

The Monitor uses its own integral Stack (see Section 1.7) which is set on entry to AMMON, therefore the program Stack which is set by the initialisation routine when your CPC464 is switched on must be reset before your program can be run. This is done for you by the **J** command. The use of two Stacks helps to make AMMON invisible to your programs.

Having run, the example program will have encountered the Breakpoint at address 6009, and "SPACE for Monitor" will be displayed on the screen. Press the SPACE bar to access AMMON.

The first operation after a Breakpoint should always be to restore the correct byte values to the addresses where the break occured.

1.5.15 K — Breakpoint Restore

This command restores the correct values into the three bytes overwritten by the Breakpoint command.

Type K

The screen will show K 6009 and will scroll up one line, displaying the prompt on the bottom line. There is no need to type ENTER. Using the M command, verify that the original codes have been replaced in addresses 6009 to 600B.

Only **ONE** Breakpoint can be entered at any time, so a Breakpoint Restore (K) command must be executed before the next Breakpoint (B) is set, and it is recommended that a Breakpoint Restore (K) command is keyed immediately after a Breakpoint has been encountered.

If you enter an incorrect breakpoint with the B command, type K immediately afterwards to restore the original values to the incorrect breakpoint address, and then re-type the Breakpoint.

The K command can only restore the **last entered** Breakpoint.

Let us now inspect the CPU registers, to make sure that the program is working as we expect.

1.5.16 R — Register display

If the prompt is not visible on the bottom line of the screen, press ESC, otherwise

Type R

The screen scrolls up, automatically displaying the CPU register contents thus:

```
>
IR      007E
              SZ■i■PNC
A'F'    8D  10001100
B'C'    7F8D
D'E'    BF02
H'L'    AE74
              SZ■i■PNC
AF      13  01101000
BC      0000
DE      0000
HL      0000

IX      BFFE
IY      8DEB
SP      BFFA
PC      6009
```

There is no need to type ENTER.

As you will see, the Program Counter contains 6009, the address at which the Breakpoint occurred. The BC, DE and HL register pairs will all contain 0000. In this example, these are the only registers that we are interested in.

The FLAGS registers are shown in BIT form, with the purpose of each flag indicated above. If a flag is SET, a 1 is indicated, and if it is reset, a 0 is shown.

The CPU registers are displayed with their contents shown in Hex.

When the Monitor is entered at a Breakpoint, the values in the Registers immediately prior to the Breakpoint are stored, so that the operation of your routine can be checked, and corrections to the routine can be made before continuing.

Having a) encountered one Breakpoint, b) restored the correct values after the break, and c) verified that the CPU registers have their correct values, we will now enter another Breakpoint, and continue the routine.

If the prompt is not visible on the bottom line of the screen, press ESC, otherwise,

Type B 600B

This will set a new Breakpoint after the BC and DE register pair have been incremented, but before the HL pair is incremented.

1.5.17 C — Breakpoint Continue

The command allows you to Continue from a Breakpoint, and is executed by typing C followed by ENTER. You can escape to the Prompt by pressing ESC before ENTER. The program will continue as if nothing had stopped it. The only information that is lost to the program is the contents of the Screen RAM.

The screen is cleared; the program Stack is reset; and the CPU registers are re-loaded from their data block before the Breakpoint address is put into the Program Counter, and execution is resumed.

Type C
Type ENTER

The routine will run on until it reaches the next Breakpoint, and will then display "SPACE for Monitor".

When the Prompt appears after pressing SPACE,

Type K	to restore the bytes occupied by the Breakpoint.
Type R	to display the registers.

You can now verify that the Program Counter contains 600B, the BC and DE register pair contain 0001, having been incremented, and the HL pair still contains 0000.

When a routine encounters a Breakpoint, it returns control to the MONITOR with a CALL operation, the return address being stored on the **Program** Stack, for use by the Breakpoint Continue (C) command. Having encountered a Breakpoint and studied the CPU registers and/or memory locations, one of two situations will occur:

1) Everything will be as you expect, and the program is correct to that point. In this case, you would normally restore the Breakpoint bytes ('K' command) and use the Breakpoint Coninue ('C' command) to continue the program to a new Breakpoint.

or

2) An error will become evident, in which case you would track down the error and correct it, and then, leaving the current Breakpoint set, use the 'J' command to re-run the program up to the same Breakpoint, to check that your correction is successful.

The Program Stack operation of the MONITOR allows you to do this providing that, at the Breakpoint, there have been an equal number of PUSHes and POPs, or CALLs and RETs. If the Program Stack is not balanced at the Breakpoint, you will have a cumulative stack imbalance every time you use the 'J' command after a Breakpoint (but not if you use the 'C' command). In this case to restore the Stack to normal once you have traced an error, RETURN to Basic ('Y' command) and re-access the monitor from the beginning, then use the 'J' command to run your program up to the Breakpoint again.

Having set a Breakpoint in a program, you can either use the **J** command to run the program, or you can RETURN to Basic, and run the program from the CALL command in Basic. For example, if you have written some Machine Code that is to be accessed from a Basic program, you can set a Breakpoint in the Machine Code using AMMON, and then RETURN to Basic and run the Basic program. When the Breakpoint is reached in the Machine Code, AMMON will be accessed, and the Breakpoint Continue command (C) will allow the Machine Code to resume and eventually RETurn to the Basic program that CALLed it.

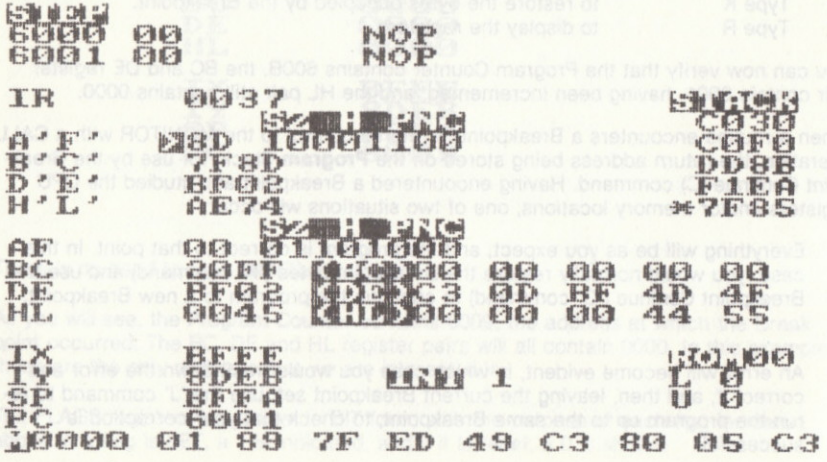
The MONITOR has been carefully designed to allow this free interchange between Basic and machine code, without upsetting the Stack.

1.5.18 T — Trace (Single Step)

The Trace command allows you to execute Machine Code in ROM or RAM one instruction at a time, or in certain specified blocks. But at ALL times, the execution of the Machine Code is strictly under the control of the Trace command, and a crash is almost impossible.

Commands such as LDIR (a self repeating block move) can cause a crash by overwriting AMMON, as can any instruction that writes into the memory occupied by AMMON. Altering the value of the Stack Pointer could also overwrite the Monitor. AMMON uses nearly 7K from the Start Address given on loading, but see Section 1.10 for fuller details.

The comprehensive screen display throughout the Trace command gives a permanent display of the CPU register contents, a disassembly of the current and the next instructions, the contents of the last five stack locations, and the contents of specified memory locations.



The Trace command is accessed from the Main Monitor by typing **T** (when the prompt and cursor are visible on the bottom line of the screen) followed by the **4**-digit Hex address of the instruction from which single stepping is to start. If you do not specify a Hex address, but press ENTER immediately after typing T, the Trace command is accessed, and the address shown against PC in the register display is used as the starting address. To return to the Main Monitor from the Trace command, press ESC. The screen will clear, and the Main Monitor's prompt and cursor will appear on the bottom screen line. All register values are passed between the Main Monitor and the Trace command, and you can therefore use all the available commands of the Monitor to debug your machine code.

The Trace command contains its own set of commands which are separate from the main Monitor commands, and allow control over

a) executing instructions singly;
b) running on to a Breakpoint (not the same as a Breakpoint set by the main Monitor);

and

c) skipping to the end of a subroutine.

Any section of code that will only operate correctly when run at full speed (e.g. timing loops, sound output or interfaces to external equipment) will not operate correctly in the Trace command as each instruction is decoded and executed separately. The main Monitor commands for setting a Breakpoint and running the code should be used instead.

To demonstrate the trace command, enter the following short routine using the **M** command in the main Monitor to enter the Hex code shown in bold type.

(If you have already accessed the Trace command, press ESC to exit to the main Monitor before entering the example program).

```
6400                0005          ORG   6400H
6400 212C64         0010          LD    HL,642CH
6403 113412         0015          LD    DE,1234H
6406 7A             0020          LD    A,D
6407 CD1064         0025          CALL  6410H
640A 7B             0030          LD    A,E
640B CD1064         0035          CALL  6410H
640E 00             0040          NOP
640F 00             0045          NOP
6410 4F             0050 SUB1     LD    C,A
6411 E6F0           0055          AND   0F0H
6413 1F             0060          RRA
6414 1F             0065          RRA
6415 1F             0070          RRA
6416 1F             0075          RRA
```

```
6417 CD2164   0080          CALL  6421H
641A 79       0085          LD    A,C
641B E60F     0090          AND   0FH
641D CD2164   0095          CALL  6421H
6420 C9       0100          RET
6421 C630     0105 SUB2     ADD   30H
6423 FE3A     0110          CP    3AH
6425 3802     0115          JR    C,+2
6427 C607     0120          ADD   7
6429 77       0125          LD    (HL),A
642A 23       0130          INC   HL
642B C9       0135          RET
642C 00       0140          DEFB  0
642D 00       0145          DEFB  0
642E 00       0150          DEFB  0
642F 00       0155          DEFB  0
              0160          END
```

The code from 6400 to 640D converts the value held in the DE register pair into four ASCII character codes representing that value, and puts the four character codes into memory at 642C (this location held in HL). The code starting at 6410 and at 6421 are subroutines called during this process.

Having entered the code press ESC to restore the main Monitor Prompt and then type **T** followed by 6400. The Trace display will be formed in a similar fashion to that shown above. The **STEP** mode is automatically selected, and is shown at the top of the screen. Below that is a disassembly of the next instruction to be executed. The Register display shows the current register contents (which could be anything as no instructions have been executed yet). The Stack display shows the last five pairs of bytes on the Stack, with the last value placed on the Stack indicated by the *.

The **M** display line at the bottom of the screen gives a window onto memory locations, and the starting address can be changed at any time by typing **M** followed by a four digit Hex address. In the example above, it would be useful to see a display of the memory that will eventually contain the four ASCII characters, so type M642C. The **M** display is updated as soon as the fourth digit is entered and you do not need to press ENTER.

While in the Trace command, the address of the next instruction can be changed at any time by typing **S** followed by the new Hex address followed by ENTER. This restores STEP mode and updates the display ready for execution of the next instruction at the new address.

To execute the first instruction (LD HL,642CH) simply press ENTER. The display will now update and show 642C in HL. Alongside that value are shown the contents of the bytes at and immediately following that address. You will see that the disassembly of the first instruction has scrolled up one line, and that the next instruction (LD DE,1234H) is

displayed on the second disassembly line. The upper of the two Disassembly lines is always the instruction just executed, and the lower is always the instruction about to be executed.

Continue pressing ENTER and STEP through the whole routine, stopping when the NOP instruction at address 640E is displayed on the second disassembly line, and observe the effect on the display of each instruction. The register display of the PC value will ALWAYS show the address of the NEXT instruction to be executed, and the Stack display will only change when the Stack is used (in this case the CALL and RET instruction).

At any time you can exit from the Trace command back to the main Monitor by pressing ESC. The screen will clear and the normal Monitor Prompt will be displayed at the bottom of the screen. You may find that you have to press ESC more than once to do this. For example, if you were in the process of typing in a command within the Trace facility, the first ESC will clear that command, and the second ESC will access the main Monitor.

You may have found that single stepping through the two subroutines became tedious after you had stepped through them once, and proved that they did work. To speed up this process the Trace command contains two functions that can be used to automatically step through sections of your program that have already been proved. They are:

(i) **BREAKPOINT**

Do not confuse this Breakpoint function with the BREAKPOINT command in the main Monitor. Setting a Breakpoint while in the Trace command does not alter your program, as it does in the Main Monitor. The Breakpoint address is stored, and after each instruction is executed by the Trace command, the address of the next instruction in your program is compared with the Breakpoint address. If the program address is not equal to the Breakpoint, the next instruction is decoded and automatically executed, until the program address is equal to the Breakpoint address, whereupon the whole display is updated and the STEP mode is accessed.

To use the above example to demonstrate the use of the Breakpoint, press ESC to access the main Monitor and use the **M** command to write 00 into the four memory locations from 642C Hex. (You do not need to do this to use the Breakpoint, but in this example it will help to clarify what is happening by clearing the ASCII codes already written there). Re-enter the Trace command by typing **T6400** and then type **B640E,** followed by ENTER. This has put the Trace command into BREAKPOINT mode and this is displayed at the top of the screen.

Pressing ENTER again will cause the program to be executed under the control of the Trace command until the Breakpoint is reached. The display is then fully updated and STEP mode accessed.

While the program was being executed, the only part of the display to be updated was the value of PC in the Register display. Should you have set a Breakpoint at an address that for some reason is never reached (perhaps a conditional instruction that has jumped elsewhere) the constantly changing PC value will indicate that the command is still running. In this case you can escape by pressing ESC, which will access the STEP mode again and update the display showing the address at which the Break occurred.

You may find that this leaves unwanted addresses or register contents on the program stack, and an incorrect value for SP in the register display. This will not affect the operation of the Monitor as it uses its own stack. The simplest way to restore the Stack Pointer to its normal value (usually as set by Basic) is to return to Basic (Key Y) and then to re-access the Monitor with | MON.

(ii) **SKIP to RET**

This function can be accessed at any time while in the STEP mode. Type **R** and press ENTER. The top line of the display will now show **STEP SKIP to RET.** Pressing ENTER again will cause the Trace command to execute your program automatically, and stop (in the STEP mode) when it has executed the RET instruction associated with the **next** CALL instruction. While the subroutine is being executed, the display of the PC register value is the only part of the display to be updated. On completion of the subroutine the whole display is updated and the normal STEP mode is accessed.

To demonstrate, type **S6400** (assuming you are still in the TRACE command) followed by ENTER. The display will update and you will be ready to single step from address 6400. Press ENTER three times to execute the first three instructions of the example program. The next instruction will now be CALL 6410. Press **R** followed by ENTER, to access the SKIP to RET mode.

Press ENTER again, and the subroutine at 6410 will be executed in its entirety, including the two CALL instructions within it, and the display will update showing the LD A,E instruction at 640A as the next instruction on the second disassembly line, and the original CALL on the line above, The normal STEP mode will also be indicated at the top of the screen.

You can now continue stepping through the program using the ENTER key. So press ENTER to execute the LD A,E, and then press ENTER again to execute the CALL 6410. The next instruction will be shown as LD C,A at address 6410 and is the first instruction in the subroutine. If you now press **R** and ENTER to access the SKIP to RET mode, a second press on ENTER will execute the program up to the RET associated with the **next** CALL. The **next** CALL is at address 6417 (CALL 6421), and execution will stop showing LD A,C at address 641A as the next instruction. Follow this through by referring to the program listing to clarify what has happened.

If you use the SKIP to RET function, and a RET instruction is found **before** a CALL instruction, that RET instruction will cause automatic execution to stop and the STEP mode to be accessed. This can be demonstrated by single stepping through the example program using the ENTER key in normal STEP mode, until an instruction in the subroutine at 6421 is shown as the next instruction. Using the SKIP to RET mode at this point will automatically execute the program until the RET at 642B. The routine will stop in the STEP mode, showing the next instruction as being the one following the CALL that called the subroutine.

To summarise **SKIP to RET,** automatic execution continues up to and including the RET instruction associated with the next CALL, OR to the next RET if a CALL is not encountered in the meantime.

In the same way as the **Z** (Disassemble) command in the main Monitor gives an 'INVALID ADDRESS' error if you try to disassemble the Monitor, you cannot single step through the Monitor program, the same error message being displayed, If this error condition occurs while you are in the Trace command, an automatic exit to the main Monitor is made, and the main Monitor's Prompt is shown at the bottom of the screen.

REGISTER POINTER

You will notice that an inverse video ( ) is displayed immediately to the left of the register contents column. This is the Register Pointer and is used to enable the contents of a register pair to be easily changed from the TRACE command. The pointer can be moved up or down by using the cursor control keys. To alter the contents of a register, move the Pointer until it indicates the required register pair, and then press the COPY key. The current value is shown on the bottom line of the Trace display.

You can now either press ENTER to restore the same value or type in a **4** digit Hex number to change the register value.

This facility only operates in the STEP mode of the Trace command and can easily be used while debugging software using the main Monitor. Access the Trace command by typing **T** followed by ENTER. All register values are passed to the Trace command. and after making your register alterations, exit to the main Monitor by pressing ESC. Again, all register values are passed back to the main Monitor.

ROM and INT status

The Trace screen displays the staus of ROMs.

The display **ROM 00** shows the currently selected Upper ROM
and **U 0**
L 0 shows the enable status of the ROMs. (0 = OFF)

These will change if a different Upper ROM is selected, or if an OUT (C),r command to Port 7F hex (i.e. B' = 7F) is stepped through. Amsoft's FIRMWARE SPECIFICATION (SOFT 158) gives more details on ROM paging and selection.

The INT display will change if an EI or a DI instruction is stepped through.

If your program uses routines that must be run at full Machine Code speed to operate (e.g. timing loops or sound output), you will need to escape from the Trace command by pressing ESC and use the main Monitor commands to run those routines properly. All the register values are passed back to the main Monitor, and you can then set a normal Breakpoint (using the main Monitor **B** command) and use the **C** (Continue) command to continue executing your program in real time from the point at which Trace stopped.

Having encountered the Breakpoint, use **K** to clear the Breakpoint, and access the Trace command by typing **T** followed by ENTER. All the Register values and the address of the next instruction are passed back to the Trace command ready for you to continue Single Stepping from the address shown against PC in the register display.

1.5.19 S — Search

This command will search for up to 5 consecutive Hex bytes or 10 consecutive ASCII characters, and display the address of the first byte of the sequence each time it occurs.

Type **S** followed by a Hex start address, plus an optional Hex end address. If no end address is given and ENTER is pressed after the start address, then FFFF is assumed as the end address. The ROM enable states (set by **U** and **L** described below) are taken into account when searching.

Having entered a start address and an end address/ENTER, select H (Hex) or $ (ASCII)

option to define what is to be searched for. Pressing H will allow up to 5 Hex values (2 digits each) to be specified. Do not press ENTER between each value, but only when all values are entered. The Search starts automatically after the fifth value. Pressing SHIFT + 4 ($) will allow up to 10 ASCII characters to be defined. Again ENTER starts the search if less than 10 characters are defined.

If more than 16 locations are found to be successful, the Search stops and waits for ENTER before continuing.

1.5.20 L — Lower ROM access

Press ESC to access the prompt and cursor if it is not already on the bottom screen line. Then press L. An inverse video "L" is displayed, along with '0' or '1' to indicate the current lower ROM enable state (1 = enabled : 0 = disabled). If you do not want to change the state, ESC will return you to the prompt and cursor; but pressing '1' or '0' will set that ROM enable state for the purposes of the Monitor. If you enable the Lower ROM, the Monitor commands that access memory (e.g. M. Z, $, P, S etc.) will acess the lower ROM and not the RAM when addressing locations between 0 and 3FFF.

1.5.21 U — Upper ROM access

This works in a very similar way to L, but for the currently selected upper ROM, Press U and the currently selected ROM number (in Hex) is displayed along with '0' or '1' to give its enable status. Pressing '1' or '0' will set that enable status.

To change the current Upper ROM selection press ESC to restore the prompt and cursor, and then press CTRL + U. This will display the currently selected Upper ROM number (in Hex). If you do not want to change the ROM number, press ESC, but to effect a change, type in the new ROM select number as two Hex digits (e.g. 07). This will select the new ROM if it exists, and enable it.

When either Basic or the Assembler are accessed, the currently selected ROM defaults back to ROM 0.

Please also read the section on ROM STATES for more information on ROM/RAM states.

1.5.22 Y — Return to Basic

The Monitor allows you to return to Basic so that you can debug machine code that is accessed as a subroutine to Basic by the CALL command.

Press ESC to restore the prompt and cursor.
Press Y, which will display "Return to Basic"
Press ENTER to exit from the Monitor.

To re-access the Monitor from Basic, use the External Command | MON as described in section 1.3.

1.6. INTERRUPTS

As well as the Interrupt status being shown in the Trace display, the Interrupt status is maintained throughout the Monitor. In other words, if at a Breakpoint the Interrupts are disabled, then they will maintain that state when a 'C' command is used to continue after the Breakpoint, unless you have single stepped through an EI instruction.

Interrupts are enabled on exit to Basic or to the Assembler.

To allow Breakpoints in a routine that has diverted the Interrupt routine (i.e. changed the JP instruction at 0038). the normal system default values at 0039 and 003A are stored in the Monitor when it is first loaded. This default value is restored on exit to Basic or the Assembler.

1.7 ROM states and the STACK

Although the CPC464 requires that the BC' registers contain the current ROM state and Port number at ALL times when interrupts are enabled or when O/S Calls are made, the Monitor has been written in such a way that Breakpoints can be entered in your programs at points where this is not true. Equally the TRACE routine will operate with ANY value in ANY register without crashing. To do this, and to allow ROM paging when the Monitor resides below 4000 (i.e. underneath the Lower ROM), certain Monitor operations MUST be RAM based.

To avoid conflicting with programs that occupy the normal free area of RAM, part of the Machine Stack area has been 'taken over' by the Monitor for its RAM based routines.

The stack area reserved by the CPC464 is from BF00 to BFFF, and the Monitor uses the area from BF00 to BF2F for these RAM based routines. The Monitor also uses its own Stack separate from the program Stack, and this is also based at the lower end the Stack page, from BF30 to BF96. The remaining Stack area from BF97 to BFFF is free for your program to use, and although this is a reduced area there are some 104 bytes (52 pairs of bytes) for program use. If your program should use more than this then the Monitor will not crash, but the bottom of your program Stack may be corrupted by the Monitor.

As it is possible to enter the Monitor from a Breakpoint with a non-standard value in BC' there is now way of checking whether your program has enabled or disabled either of the ROMs before that Breakpoint. Therefore, at a Breakpoint, the stored ROM state as shown by the U and L commands is not updated. However, in the TRACE command, if an OUT (C),r command to Port 7F is detected as the current instruction, this is simulated, and the ROM state display is updated.

1.8 Screen Modes

On entry to the Monitor, pressing SPACE will not change the screen mode from that which your program had set. It is possible to change the screen mode from within the Monitor by using **CTRL + 1** for Mode 1 or **CTRL + 2** for Mode 2. Any change made will be effected by use of a routine in the Lower ROM which has the effect of destroying many screen parameters (e.g. cursor locations and screen colours) for all screen windows, and these parameters affect your program as well. The original program screen mode is reset after a **J** or **C** command.

Providing that you can use the Monitor satisfactorily in the same Mode as your program (i.e. without changing Modes), the Monitor is designed to retain as much of your program screen data as possible. This is done by using a different screen window or Stream within the Monitor from your program. Normally the Monitor uses Stream 7 for all its screen output, but any of the Streams (0 to 7) can be defined as the Monitor Stream.

 # (SHIFT + 3) is used to change Monitor Streams, and the existing Stream number is displayed. Type in a single number (0 to 7) to change Streams.

The new Stream is selected, and on that Stream,
the VDU is enabled
the whole screen is selected as the window area
and opaque mode is selected.

The program Stream is reselected after a J or C command and in the Trace command.

If the screen Mode is not changed within the Monitor, all of your program screen parameters should remain unchanged while using the Monitor.

1.9 Keyboard

If, on entry to the Monitor from a Breakpoint or from Basic, the Keyboard has been re-defined such that Monitor command keys are not recognised, then it is possible to completely reset the keyboard from within the Monitor.

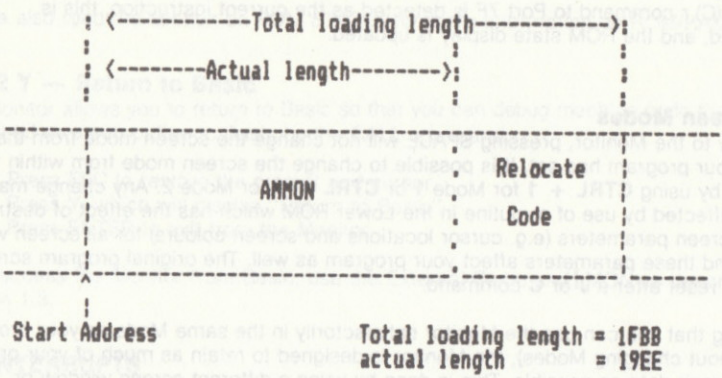
CTRL + @ will reset the keyboard, but the old status of the keyboard will be lost. This will always work, as the '@' key is tested directly by the Monitor as a Key number and not as the character returned by that key.

The ENTER key is set to repeat (which is does not normally do in basic) to facilitate the M and $ commands. Its repeat state on entry to the Monitor is restored after a J or C command, and on exit to Basic or the Assembler.

1.10 THE MONITOR IN PRACTICE

Length and location of AMMON

To enable the most effective use to be made of the relocatable feature of AMMON, a memory map will help to explain memory usage:



When AMMON is loaded, a short Basic program is first loaded which is used to get the Start Address and alter HIMEM to (Start Address - 1). The Machine Code is then loaded into memory at the Start Address and this code is 1FBB Hex long (about 8K). The relocate routine is at the higher memory addresses, and is overwritten with zeros once it has been run. So having relocated, the actual length of AMMON is only 19EE (almost 6.5K).

There is no check made on the validity of the Start Address supplied, and the highest values normally acceptable to the system, without overwriting memory reserved by the Operating System, are:—

Without Discs, but with SYMBOL AFTER 256 = 8C40
with Discs, and with SYMBOL AFTER 256 = 8740

These values will be 80 Hex (128) bytes lower if you do not execute a SYMBOL AFTER 256 before loading.

The simplest method of loading AMMON to the highest available location is to press ENTER in response to "Start Address?" when loading.

Saving & Loading Code
If you have used AMMON to write short Machine Code routines or blocks of data directly into memory without using the Assembler, and you wish to Save this code you should return to Basic and use Basic commands to do this. You can specify the start location and length in Hex or use AMMON's number converter to get the Decimal equivalents.

System Status throughout AMMON
AMMON has been written to preserve as much as possible of the CPC464 machine system status while you are using the Monitor functions.

On entry to AMMON, the sequence of events is:—

i) INT status is stored
ii) All register contents are stored
iii) Stack Pointer is set to AMMON Stack
iv) Current program screen Mode is stored
v) Repeat state of ENTER key is stored: key is set to repeat
vi) Screen is set to AMMON default Stream (usually Stream 7)
vii) Entry message is displayed

On exit from AMMON by way of **J** or **C** commands:—

i) Screen Stream is set to program Stream
ii) Screen Mode is reset to program Mode **only** if Mode has been changed in AMMON
iii) ENTER key repeat status is reset
iv) INT status is reset
v) Register contents are loaded with stored values
vi) Stack Pointer is reset to program Stack

On exit from AMMON to Editor Assembler:—

i) Stack Pointer is set to its default value in program Stack
ii) ENTER key repeat status is reset
iii) ROM O is selected as the current Upper ROM
iv) The Interrupt Vector at 0038 is reset to its default value
v) Interrupts are enabled

On exit from AMMON to Basic:—

i) ENTER key repeat status is reset
ii) ROM O is selected as the current Upper ROM
iii) The Interrupt Vector at 0038 is reset to its default value
iv) Interrupts are enabled
v) Stack Pointer is reset to its default value in program Stack

Single Stepping ROM Routines

If you use the Trace command to Single Step through ROM routines, particularly those in the Lower ROM, you may find that your computer creashes and will not accept any further keyboard input. The Lower ROM routines most likely to cause this effect are those concerned with printing text to the screen. AMMON uses these routines to create the Trace display, and the Lower ROM routines expect to find the data they use for that display intact. By Single Stepping the ROM routines, you will change that data, and thus cause the CPC464 operating system to lock up.

If you escape from the Trace command while Single Stepping a Lower Rom routine, you will probably leave the Lower ROM enabled. Any address between 0000 and 3FFF will then be interpreted by the Main Monitor commands as a ROM location, not RAM. This may become evident if you then use the Monitor to inspect the Assembler's Object Buffer, which normally lives below 3FFF. In this case, it would appear that the Assembler has not assembled your program. In reality you would be looking at ROM, not RAM locations, and you would need to disable the Lower ROM as described in Section 1.5.20.

1.11 Summary of Commands

M aaaa nn
Memory location & contents in Hex
aaaa = address
nn = new contents
ENTER for next location
M re-enters command

ESC
Escape to Prompt & Cursor

$ aaaa letter
Memory location & contents in ASCII
aaaa = address
letter = character from keyboard
ENTER for next location
$ re-enters command

I aaaa bbbb nn
Insert
aaaa = address 1st byte insertion
bbbb = address highest byte to be moved
nn = number bytes to Insert

D aaaa bbbb nn
Delete
aaaa = address 1st byte deletion
bbbb = address highest byte to be moved
nn = number bytes to Delete

A aaaa bbbb cccc
Area Move
aaaa = present start address
bbbb = present end address
cccc = new start address

F aaaa bbbb nn — **Fill**

aaaa = start address of area to Fill
bbbb = end address of area to Fill
nn = value to be loaded into area

P aaaa bbbb/ENTER — **Print Hex/Graphics Dump**

aaaa = address of first byte
bbbb = address of last byte (optional) or ENTER for open ended screen display
Printer options on Hex display if end address given

Z aaaa bbbb/ENTER — **Disassembler**

aaaa = start address
bbbb = end address (optional) or ENTER for open ended screen display
Printer options if end address given.

N H/D number — **Number Converter**

H/D = Hex or Dec number
number = value to be converted

E — **Editor Assembler**

Access to AMMAS if resident in memory

B aaaa — **Breakpoint**

aaaa = address of Breakpoint

J aaaa — **Jump & Execute**

aaaa = address to Jump to

K — **Breakpoint Restore**

Restores last Breakpoint — executes automatically, giving address

R — **Register Display**

C — **Breakpoint Continue**

Continues program execution after a Breakpoint

T aaaa/ENTER — **Trace**

aaaa = address from which Trace starts or ENTER to start Trace from PC value in Register display.

S aaaa bbbb/ENTER — **Hex or ASCII Search**

aaaa = start address
bbbb = end address (optional) or ENTER for default end address of FFFF.

L 0/1 — **Lower ROM access**

Current enable state given. 0/1 to disable/enable ROM

U 0/1 — **Upper ROM access**

Current enable state given. 0/1 to disable/enable ROM

CTRL U nn — **Upper ROM Select**

Current select number given
nn = new ROM number in Hex.
ROM nn is selected & enabled

Y — **Return to Basic**

# n	Screen Stream Current AMMON Stream given n = new Stream for AMMON to use (n is a single decimal number)
CTRL 1	**Screen Mode 1**
CTRL 2	**Screen Mode 2**
CTRL @	**Keyboard reset**

SECTION 2

The EDITOR ASSEMBLER "AMMAS" Version 1.1

2.1 General Description

The ASSEMBLER allows you to write your Machine Code program as a series of instructions (mnemonics) into a listing with line numbers in a similar fashion to a Basic listing. The ASSEMBLER is completely self-contained, and operates in MODE 1 or MODE 2, with a screen display that is automatically tabulated into fields to make your program listing very easy to read. Entering your Machine Code listing is a quick and simple process since the ASSEMBLER'S Line Editor contains facilities to provide Automatic Line Numbering; to Renumber the listing; to Edit and alter any line; and to insert lines or delete unwanted lines. A full-screen Copy Cursor is available operating in a similar fashion to Basic.

The ASSEMBLER accepts all the Z80 mnemonics (plus a number that are not published) and will accept both Decimal and Hex numbers, and an unlimited number of Labels, each containing up to 6 characters.

To simplify programming, a number of Assembler Directives have been included to allow you to define constants, variables, messages, and to set up data tables. These are ORG, END, EQU, DEFL, DEFM, DEFS, DEFB and DEFW. The Directive PRNT allows screen or printer output to be turned ON or OFF during the Assembly process. Arithmetic functions allow the ASSEMBLER to perform addition or subtraction within operands, and any combination of Label names, numbers or single ASCII characters is accepted. This allows the ASSEMBLER to calculate the length of a message or data table, and to accept negative numbers.

When assembling your Machine Code program, the assembled code is written into memory, and can be displayed on the printer, or on the screen with a PAUSE facility to freeze the assembly process while you study the assembled listing on the screen. The fastest assembly mode is with no output to either printer or screen, and in this mode, 1K of finished Machine Code is assembled in approximately 7 seconds. Full error detection is included, and when an error is found assembly stops with a sensible error message. The whole line containing the error is displayed on the screen, and you are left in the EDIT mode, ready to correct the error and re-enter the line into the listing.

The ASSEMBLER contains routines to LIST to a printer and to SAVE, LOAD and VERIFY the program listing or the resulting Machine Code to Disc or Tape. The ASSEMBLER automatically calculates the start address and length of any SAVED information, and the assembled code will LOAD back via Basic.

To allow long programs to be assembled, the Source Code can be stored on tape or disc in up to 26 linked sections. Each section is loaded into memory in turn during the assembly process, and the resulting Machine Code can either be stored in memory, or for very lengthy programs, can be sent directly back to tape or disc. This would allow you to assemble up to 64K of code if you so required, and with discs attached, the whole process is automatic. Using tape, detailed screen prompts guide you through the assembly process.

All commands are available from the keyboard as single keystrokes, and Save/Load file names can be defined to be available from a single keystroke. To simplify access from Basic, the Assembler is set up as an RSX (i.e. it is accessible as an External Command), and it fully supports External Commands to allow the use of Disc commands etc. without the need to return to Basic.

2.2 Loading AMMAS

As the Assembler operates as an RSX, it is advisable to load it into a clean machine, so clear the CPC464 by CTRL + SHIFT + ESC. If you have Discs attached, then enter the direct command **|TAPE.IN** (ENTER) and type **RUN"** (ENTER) to load the Assembler from cassette. AMMAS gives you the option of making a Back-up copy onto Disc or Cassette, and if you have Discs attached, you should enter the direct command **|TAPE.OUT** if you require a cassette back-up. You should also set the cassette write speed from Basic before Loading the program.

A short Basic program loads the code of the Assembler and offers the option of making a back-up copy onto disc (or cassette).

To allow for memory 'grabbed' from the memory pool by discs or other ROM or RSX software, the Assembler is relocatable, and automatically relocates itself to below the current HIMEM.

This creates a very simple method of running the Monitor and Assembler together by loading the Monitor first and locating it to the highest available memory (press ENTER in response to 'Start Address'), and then loading the Assembler. It will then locate itself below the Monitor. You can reserve memory above the Assembler for your own purposes by lowering HIMEM before loading the Assembler. Remember that neither the Monitor or the Assembler do a SYMBOL AFTER 256 in their Basic loader programs, so if you do change HIMEM before loading either program, AND you wish to use further definable graphics, then do a SYMBOL AFTER 256 from Basic BEFORE loading.

To load a back-up copy from disc, type **RUN "ASS".** The Assembler is Called and the usual entry message displayed.

WARNING

THIS PROGRAM IS COPYRIGHT AND ONE COPY ONLY may be made for your OWN PERSONAL USE.

It is illegal to sell copies or to give copies to friends.

2.3 Access to AMMAS

On loading, the Basic program sets HIMEM to a very low value which is passed to the Assembler as the start of the Object Buffer. The Assembler is totally self-contained and not dependant on any Basic, and is compatible with any Basic program that you wish to load at the same time. If you alter HIMEM from Basic after loading the Assembler, you must pass the new HIMEM value to the Assembler next time you access it. The normal means of accessing the Assembler is by **|ASS** but if you have changed HIMEM, then use **|ASS,HIMEM + 1.** The value of the start of the Object Buffer will be set to HIMEM + 1, but the amount of memory available to the Assembler will be altered.

To make the most effective use of AMMAS, you should allow it to access as much

memory as possible. You should make sure that HIMEM is set as high as possible before Loading AMMAS which allows it to load as high as possible in memory. If you need to use Basic while AMMAS is in memory, make sure that HIMEM is set as low as possible. AMMAS will then use the area between these two HIMEM values.

On accessing AMMAS, you will be greeted by the following message:—

EDITOR ASSEMBLER
AMMAS 1.1
© Picturesque

NEW Text or CONTINUE with text (N/C)

If you have no Source Code in the Assembler, press N which will ensure that all the Assembler's buffers are reset. Any Source Code in memory will be lost in this case.

If you already have Source Code loaded into the Assembler, pressing C will change nothing in AMMAS and allow you to carry on with your machine code programming.

2.4 THE EDITOR

AMMAS works in two distinctly separate parts. The **EDITOR** is used to enter and edit your Source Code listings and offers a number of functions to simplify this process, for example Auto line numbering, line re-numbering, and block copy and delete functions. It is a Line Editor that always uses the bottom line of the screen to accept keyboard entries, with screen scrolling when necessary. There is a Copy Cursor facility operating in a similar way to Basic's Copy Cursor.

Having typed a Source Code listing into AMMAS, it can be saved to Cassette or Disc for future use. But before a program represented by that Source Code listing can be run, it must be converted into machine code by the ASSEMBLER. When you Load AMMAS into your CPC464, both the EDITOR and the Assembler are loaded as one program, and the EDITOR is automatically entered. The EDITOR controls the operation of the whole of AMMAS, and contains commands to invoke the ASSEMBLER and the Cassette/Disc/Printer facilities.

When you are entering or editing a Source Code listing, the Editor normally produces capital letters at all times, and will only allow lower case letters to be displayed between quotation marks, when the SHIFT and CAPS LOCK keys operate in the normal way.

You can move the cursor through the Edit line at the bottom of the screen using the left and right cursor control keys. The DELETE key deletes the character to the left of the cursor, and moves the cursor one character to the left. To move the cursor quickly to the left hand end of the edit line, press CTRL and the left cursor key together.

2.4.1 AMMAS COMMANDS

All commands can be typed in character by character, but to simplify access to the commands, they are all available by holding the CTRL key down while pressing one other key as follows:—

Command	CTRL + Key
ASSEMBLE	A
AUTO	+
BASIC	B
CLEAR	X
COPY	C
DELETE	D
EDIT	E
FILE	F
LIST	\
LOAD	L
LABEL	K
MONITOR	M
MODE 1	1
MODE 2	2
NEW	N
RENUM	R
SAVE	S
VERIFY	V

Wherever possible, the initial letter of the command name is used to access that command, but where this is not possible (e.g. LIST, LABEL, LOAD) other keys have had to be used.

Pressing CTRL plus one of the command keys shown above will always print the command name on the screen at the current cursor location, but for the command to be recognised, it must be the first word on the Edit line. In other words it must be at the left hand end of the bottom screen line. If you are unsure that the cursor is in the correct place, press ESC before accessing the command name. There must be a space after the command name and before any operands. Using CTRL plus a key puts the space there automatically.

2.4.2 Screen Display

The Screen display is always tabulated into fields to make your listings easy to follow, and the SPACE bar acts as a TAB function when you are entering lines of Source Code. The cursor control keys automatically detect the boundaries of the fields within the Edit line. You can use Mode 1 or Mode 2 (see sections 2.4.12 and 2.4.13) with any choice of Paper and Pen colours.

2.4.3 Entering a line of Source Code

The machine code program that you wish to write is entered into a listing in a similar fashion to a Basic listing. Only **one** instruction per line is allowed, and the line must contain a line number between 0 and 9999, and the operation name along with the relevant operand(s). A full list of Z80 mnemonics in the correct form for the ASSEMBLER is shown in Section 4. The program line may also contain a LABEL name that will identify that instruction in the program. The screen display is divided into **4** fields as shown below, starting at the left hand end of the line:—

Line number	— 1st 4 characters plus 1 space
Label name	— Next 6 characters plus 1 space
Operation name	— Next 4 characters plus 1 space
Operands	— Remainder of line

```
1045 LABEL  LD   HL,3FFFH
1050        LD   DE,32768
```

Use the "←" and "→" cursor controls to move the cursor along the bottom line. With an empty line, the cursor will jump to the beginning of each field. Use the "←" cursor control to ensure that the cursor is at the left hand end of the line and type in a line number, say 10.

As you type each character, it is displayed, and the cursor moves 1 location to the right. Having entered the number, type SPACE. The cursor will move to the start of the next field, the LABEL field. The SPACE key has several different functions depending upon the cursor position in the line, and these will be explained later. Its main function when entering a line is to advance the cursor to the start of the next field, clearing any characters it passes. The cursor is now at the start of the Label Field. Most of the lines you will enter will not require a Label, so type SPACE again, and the cursor will move to the start of the Operation Field.

Pressing space a third time has no effect, as every line you enter must have an operation name, and the cursor waits for a name to be entered.

All programs you write **must** start with a definition of the address from which the program is to be assembled. This is called the ORIGIN, and is abbreviated to ORG. Type in ORG as the operation. Type SPACE, and the cursor will now move to the start of the Operand Field. The ORG operation now requires the address that will be the address of the first byte of the assembled program. For this example, enter 6000H. This represents the Hex address 6000 (= 24576 decimal). For more information on ORG, see section 2.7.1, and for more about numbers, see section 2.6.2.

The bottom line of the screen should now be:—

```
0010            ORG   6000H
```

Before entering this line into the listing, use the "→" and "←" controls to move the cursor around the line. When the cursor is over a character, pressing an alpha-numeric key will replace the original character with the one typed in, and move the cursor **1** location right. The "→" and "←" controls move the cursor to the next character left or right, within a field, ignoring any spaces. If a field is empty, the cursor moves to the appropriate end of that field.

To DELETE a character, move the cursor to the character to the right of the one to be deleted and press **DEL.** The character to the left of the cursor is deleted and the cursor moves into the deleted location. The rest of the line is **not** moved.

Now press **ESC** The message ***BREAK*** is displayed and the cursor moves to the left hand end of the bottom line of the screen, which will have scrolled up. Pressing ESC will abort the current Editor function (and cancel Auto line numbering) at any time. Press Space again and nothing happens, as the cursor is waiting for a line number or a command name.

Re-type the whole ORIGIN line again, and experiment with the cursor control keys, the SPACE and DELETE keys until you are familiar with the operation of these functions. When you are ready re-type the ORIGIN line again, and this time press ENTER.

The screen will scroll, and the cursor will appear at the left end of the bottom line. The line has been entered into the listing. At the time of entering a line into the listing, checks are carried out on the contents of the line to ensure that each field ends with at least one space. The spaces are essential for the ASSEMBLER to recognise the various

parts of the line. If they do not appear in the line when you press ENTER, the line is not entered, and the cursor appears over the unwanted character.

Section 4 shows a complete list of mnemonics in the form that the Assembler will recognise, and it is important that you enter them in this form. They all conform to the standard Z80 instruction set.

Using the four fields into which the screen is divided, always enter a line number; enter a label name (maximum 6 characters) **or** leave the label field blank; enter the operation name (LD CALL RET etc.) into the operation field; and enter any operands into the last field. Where an operation has two operands, they **must** be separated by a comma e.g.

```
0020            LD      A,20H
```

2.4.4 LIST

Move the cursor to the left hand end of the line and press CTRL and the\ keys together. (The keys are one above the other.) The word LIST will appear on the screen. Pressing ENTER will produce a listing in numeric line order starting with the lowest line number in the listing. Ten lines of program are displayed whereupon the listing will temporarily stop. Subsequent blocks of ten lines of listing are displayed by pressing SPACE, until the end of the program is reached. After each block of ten lines, any other command may be used before continuing with the listing.

You can specify a line number after LIST, and the listing will start from that line number, or the next line, if that line number does not exist.

For example **LIST 500** will list from line 500 onwards, whereas in Basic it would only list that one line. If you are used to the Basic form of LIST 500— to list on from line 500, then that form is also acceptable to AMMAS.

You may well have RENUMbered your Source Code listing in the process of entering it, and be unsure of the current line number of a particular part of the listing. If this is the case, and you know a label name close to the area you wish to list, you can use the List command to list from that label name with

LIST XXXX where XXXX is the label name.

The Editor will search for the definition of that label name in the label field of your Source Code, and list from the line containing the label definition. If the label specified in the LIST command does not exist in your Source Code, no listing is produced.

If you have a printer connected to your CPC464, you can produce a printout of your listing by typing a / after the LIST command name and before any line number or Label name. For example

LIST /500 will list to the printer from line 500 to the end of the listing. You can pause the printer listing by pressing ESC once, after which, press SPACE to continue or press ESC again to terminate the printer listing.

2.4.5 EDIT

Move the cursor to the left hand end of the line and press CTRL and the E key together. EDIT is displayed on the screen. Pressing ENTER now will display the first line of the listing at the bottom of the screen. To EDIT a specific line, enter the line number after the EDIT command and press ENTER. You can now use the cursor control keys to edit the line and re-enter it into the listing by pressing ENTER.

To delete a line from the listing, type in the line number only and press ENTER.

2.4.6 AUTO

Move the cursor to the left hand end of the line and press CTRL and the key with + engraved on it. AUTO will be displayed on the bottom screen line. When enabled, this command automatically gives you a new line number each time you enter a line into the listing.

The command form is:—

AUTO x, y

where AUTO is accessed from CTRL +
x = the starting line number
y = the step value (between 1 and 99)
x and y must be separated by a comma.

For example, AUTO 1000,5 will start numbering at line 1000 in steps of 5.
and AUTO 10 will start producing line numbers from line 10 with the previously defined step value. If you have not yet defined a step value, the default value is 5.

If AUTO line numbering is about to produce a new line number in excess of 9999, the AUTO facility is aborted to avoid overwriting the start of your listing.

When a new line number has been displayed on the screen, the cursor is positioned at the start of the Label field. If you do not wish to enter a label name, press SPACE once to move the cursor to the start of the operation field.

To stop the AUTO facility, press ESC once. Re-entering AUTO again without specifying any values will use the last step value with a line number following the last one produced by AUTO.

2.4.7 RENUM

Move the cursor to the left hand end of the Edit line and press CTRL and the R key together. RENUM will be displayed on the bottom screen line.

This allows the Source Code line numbers to be renumbered.
The command form is:—

RENUM x, y, z

where RENUM is accessed from CTRL R
x = the step value
y = the line number from which to start renumbering
z = the new value to be given to line y (the routine actually gives a new value of z + x to line y)

EXAMPLE a). RENUM 2,1000,2000 will give a step value of 2, and renumber from the existing line 1000, giving that line a new number of 2002. Subsequent line numbers, through to the end of the listing, will be renumbered in steps of 2.

EXAMPLE b). RENUM 2,1000 will renumber from line 1000 to the end of the listing in steps of 2.

EXAMPLE c). RENUM 2 will renumber the whole listing in steps of 2.

In all cases, if the RENUM parameters would create a line number greater than 9999, the step value is reduced by 1 and the renumber is executed again automatically. In Example b). if the step value has reached 1 and a line number greater than 9999 would be produced, the whole listing is renumbered in steps of 1. If, in Example a). the step value reaches 1 and a line number greater then 9999 would be produced, the value of z is reduced and the renumber is attempted again with a step value of 1 until it is successful.

In all cases, renumbering is effective from the chosen line in the listing through to the end of the listing.

2.4.8 DELETE

Move the cursor to the left hand end of the Edit line and press CTRL and the D key together. DELETE will be displayed on the screen. This command allows a block of lines to be deleted from your Source Code listing.
The command form is

DELETE x, y

where DELETE is accessed from CTRL + D
x = line number of start of block
y = line number of end of block.

Press ENTER, and the block of lines from x to y INCLUSIVE will be deleted.

2.4.9 COPY

Move the cursor to the left hand end of the Edit line and press CTRL and the C key together. COPY will be displayed on the screen. This command will copy a block of Source Code into another part of your listing. The original block of listing is not deleted.

The command form is

COPY x, y, z

where COPY is accessed from CTRL + C
x is the first line number of the block to copy
y is the last line number of the block of copy
z is the new point in the listing to copy to

If Z is between X and Y then a Syntax Error is generated.
If X >= Y then a Syntax Error is generated.
If a line Y exists, that line is included in the Copy.
If a line Z exists, then the block is inserted AFTER it.
If a line Z does not exist, then the block is copied to where line Z would be.

The listing is not re-numbered, and you will have duplicate line numbers.

2.4.10 NEW

This command clears the Text Buffer of all Source Code, clears the Label Table, and resets the Assembler to the same state it was in when you first loaded it.

With the cursor at the left hand end of the bottom line, press CTRL and the N key. The message that you saw when you first loaded the Assembler will appear. if you typed NEW by mistake you can now press **C** and nothing will be lost. If you want to erase the entire program listing, press **N.**

2.4.11 CLEAR

This command is used to Clear specific parts of the Assembler's buffers and tables.

Move the cursor to the left hand end of the Edit line and press CTRL and the X key together. CLEAR will be displayed on the bottom line. You must now specify one of four single parameters to indicate which part of memory is to be cleared.

The command forms are:—

CLEAR L This clears all the labels from the label table, and resets the label table to the # label only.

CLEAR O This clears the Object Buffer to zero length. (use the letter O, not the number zero)

CLEAR T This clears the Text Buffer of your current Source Code, and enables a protection system on the labels currently in the label table. This is useful where you need to assemble two separate sections of Source Code, with the second section having access to the labels created by the first section. The ASSEMBLE command will clear the label table back to its protected size (normally just the # label) before it starts to assemble code.
The protection can be cleared by the following commands:-
CLEAR P; CLEAR L; NEW

CLEAR P This clears the protection applied by CLEAR T to the label table. All the labels remain in the label table, but without protection.

2.4.12 MODE 1

Move the cursor to the left hand end of the bottom screen line and press CTRL and 1 together. MODE 1 is displayed on the screen. This command (and MODE 2 below) change the screen display mode.

If required, two operands may be appended to this command to define the INK colours used by PEN and PAPER. The Assembler always uses PEN 1 : PAPER 0 and the colours associated with INK 1 and INK 0 are set to the values given. The first operand is the PEN ink and the second operand is the PAPER ink. The two must be separated by a comma. The given values are masked to be within the range of 0 to 26, and the Border colour is set the same as the Paper colour.

e.g. **MODE 1 26,0** gives white ink, black paper.

2.4.13 MODE 2

This command form is identical to MODE 1 above, but sets Screen Mode 2. PEN and PAPER inks can be specified if required.

2.4.14 BASIC

This command allows you to return to BASIC from AMMAS. Move the cursor to the left hand end of the Edit line and press CTRL and the B key together. BASIC will be displayed at the bottom of the screen. Press ENTER and you will return to Basic, with the usual "Ready" message.

To re-enter the Editor Assembler see section 2.3.

The Basic Loader program for AMMAS will probably still be in memory, but as AMMAS is completely self-contained, you can safely use the Basic command NEW to remove it. You can also use your own Basic programs while AMMAS is in memory, but remember that if you alter the value of HIMEM you must tell AMMAS its new value when you re-access the Assembler. You should use the External command **| ASS,HIMEM + 1.**

If you need to Load or Save while in Basic, you will have to raise HIMEM to give Basic enough room for its Tape/Disc buffer. AMMAS sets HIMEM to a very low value, and Basic will give a "Memory Full" error if you do try to use Tape/Disc commands without altering HIMEM.

2.4.15 EXTERNAL COMMANDS

When additional ROM or RSX software is in your CPC464, you can access this software through External commands with the form | NAME where NAME is the command name (e.g. | MON gives access to AMMON). The Editor Assembler also contains facilities for accessing External commands without the need to return to Basic to do so.

Move the cursor to the left hand end of the Edit line, and type SHIFT + @ to give the BAR symbol followed immediately by the external command name. A maximum of 5 numeric or string operands can also be specified. Decimal or Hex numbers are accepted, but Hex numbers must be in the AMMAS format (i.e. 4000H : see section 2.6.2).

If you add operands to the External command, follow the command name with a comma. String operands are typed out in full, and not accessed via a variable as in basic.

e.g. **| ERA,FILENAME.BIN** to erase a file from Disc
| REN,FILE2.BIN,FILE1.BIN to rename a Disc file.

If a string operand starts with a number character, you should enclose the whole string operand within quotes to avoid any confusion over the operand being treated as a number by AMMAS. The quotes will not be counted as part of the string.

e.g. **| REN, "FILE2.BIN","FILE1.BIN"** has the same effect as the above example.

2.4.16 COPY CURSOR

When entering a listing, or during Editing, a full screen Copy Cursor is available exactly as in Basic, using SHIFT with the arrow keys to move the Copy Cursor, and COPY to transfer the relevant character into the Edit line.

When a Space is copied into the Edit line at the bottom of the screen, the Main Cursor in the Edit line does not Tab across to the next field, but simply writes a single Space into the Edit line. Because of this, you could find that a line of Source Code is rejected by the Editor because it does not conform to the expected field format.

As the Editor of AMMAS always operates in an "overwrite" mode as opposed to an "insert" mode, the Copy Cursor offers a simple way of inserting characters into a line of listing, for example into a message line. List the Source Code so that the line requiring an insertion is on the screen. Use the COPY CURSOR to copy from the line number up to the insertion point into the Edit Line and then type in the insertion from the keyboard. Finally, use the COPY CURSOR to copy the remainder of the line from the screen listing and ENTER the new line into the listing.

2.4.17 Comment lines

Comments can be inserted into your listing, in a similar way to a REM line in Basic.

Enter the Line Number; move the cursor to the start of the Label Field and enter a semi-colon.

Using the "→" cursor control, move the cursor to the start of the **operand** field and enter your comment, enclosed in quotes, exactly as you would enter a message line. (See DEFM). The semi-colon identifies the line as a comment, and the Assembly routine ignores the whole line.

2.4.18 ESC Key

The **ESC** key can be used at any time to cancel or stop most functions; (e.g. during assembly, listing, display of label table, disc/cassette commands or to cancel unwanted commands before execution). Using the ESC key while typing in a command or a line of Source Code will cause a BREAK straight away, but if a command is running when ESC is pressed, that command freezes and the cursor appears at the bottom of the screen. Pressing SPACE allows the command to continue and a second press on ESC will cause a BREAK.

2.4.19 Resetting the keyboard

If you return to Basic, the command name expansion strings will still operate, and will be available on re-entry to the Assembler. If, however, you have changed any of the main keyboard definitions in Basic, you can reset the Assembler keyboard by **CTRL + @.** This will re-define all the key expansion strings, and will set the **FILE** name (see section 2.5.1) to a null string.

2.4.20 MONITOR access

If you loaded the MONITOR (AMMON) program as well as the Assembler, you can directly access the Monitor in one of two ways. (See Section 3 for loading both programs together).

Firstly, as the Monitor is set up as an RSX, you can use the external command | MON and press ENTER.

Secondly, if you press CTRL and the M key together, you will see MONITOR appear on the bottom screen line. Press ENTER to gain access to AMMON.

2.5. CASSETTE AND DISC COMMANDS

It is possible to Save the contents of both the Source Code buffer (your program listing) and the Object Code buffer (your assembled program) directly from the Assembler. There is no need to specify a start address or length as the Assembler calculates this automatically.

On loading, the Assembler checks to see if DISCS are present, and all cassette based routines will work with the Discs if they are available.

If you wish to use cassettes while the Disc interface is connected to your CPC464, you should use the External commands | TAPE.IN. | TAPE.OUT or | TAPE to re-instate the required cassette facilities. Those external commands maintain their effect until countered by the relevant | DISC external commands. Full details of these external commands are given in Amsoft's Disc User Instructions. A 2K buffer for Cassette/Disc commands is reserved within AMMAS.

N.B.
It is possible to Load and Save a Source Code file from Basic, but if you do, you will either corrupt the Assembler, or the Assembler will not recognise the Source Code.

CASSETTE SYNTAX

The syntax of the cassette commands is very similar to Basic cassette commands. You can LOAD, SAVE and VERIFY Source code or Object code by specifying a file name of up to 16 characters long, and the file name must be enclosed in quotation marks (SHIFT + 2). To change the speed at which the Save is executed, you must return to Basic (see Section 2.4.14) and use SPEED WRITE 0 or SPEED WRITE 1 from Basic. You can then re-access AMMAS and all further Saves will operate at the Speed you have selected.

DISC SYNTAX

When using Discs, the file name specified can contain optional User, Drive and Type parts along with the compulsory Name of up to 8 characters, and the syntax is the same as in Basic. The whole file name, including any optional parts, must be enclosed in quotation marks (SHIFT + 2).

When Saving or Loading with Discs, the Type part of the file name will default to ".SCE" for Source code, or ".BIN" (Binary) or ".COM" (CP/M) for Object code. However, if you specify a different Type part, then that will be used in place of the default characters. All Saves are defined in their header as Unprotected Binary files, except for ".COM" files which are unprotected ASCII.

2.5.1 FILE

You can define a file name for use in Cassette/Disc commands that will then be

available from a single key.

Move the cursor to the left hand end of the line, and press CTRL and the F key together. FILE will be displayed on the bottom line of the screen. Type in the file name, including the opening and closing quotation marks, and press ENTER. The file name will be checked for syntax, as if it were a cassette command with a maximum of 16 characters. Spaces are ignored. If you are using Discs, then a full Disc syntax check is carried out by the LOAD/SAVE/VERIFY commands. Also, if you are using Discs, a Type part is not added by the file command, but is added by the LOAD/SAVE/VERFIY commands. If you wish, you can specify a type part within the **FILE** name you define.

To access the defined **FILE** name, hold down CTRL and Press the 4 key (it has $ engraved on it) and the file name, complete with quotation marks is printed on the screen. For example, to Save a file with the default FILE name, use CTRL + S for SAVE and CTRL + 4 for the file name.

2.5.2 SAVE

With the cursor at the left hand end of the Edit line, press CTRL and the S key together. SAVE will be displayed on the screen.
Type in a file name (or use CTRL + 4 if you have defined a FILE name) and make sure that the file name ends with quotation marks (SHIFT + 2).

You must now specify whether you want to Save Source code or Object code. To Save Source Code that you have entered, follow the file name with T (for Text) and to Save your assembled Object Code, follow the file name with either B (for BINARY Code) or with C (for CP/M COM files).
Then press ENTER.

SAVE "name"T	Saves Source Code.
SAVE "name"B	Saves Object Code as Binary Code.
SAVE "name"C	Saves Object Code as CP/M ".COM" file. (Disc only)

If you are using cassette routines, you will see the usual cassette messages.

If you are using Discs, you will see some new messages. If you Save a file to Disc from Basic, and a file of the same name is already on the Disc, it is automatically made into a Back-up version, with the Type part ".BAK". However, the Assembler gives you the option of making a back-up, or of over-writing the old file, or of abandoning the Save to choose a new file name.

So, when Saving to Disc, the Assembler searches the Disc Directory to see if a file of the name specified already exists, and the first new message you see tells you that a Search is being made. If a file of the same name does exist, a message will tell you so, and will ask you:—

Overwrite :Backup: Abandon (O/B/A)

You must respond with **SHIFT** plus the letter **O** or **B** or **A.** The SHIFT has been added to make your keypress a definite and consious choice. This should help to eliminate overwriting files by mistake. Having made your choice, the Save will go ahead.

If the file name specified in your SAVE command does not exist on the Disc, you will be told so, and the Save will go ahead immediately. With Saves to Disc, you do not need to specify a Type part, as Source code is automatically given ".SCE", and Object code is given ".BIN" or ".COM".

OBJECT CODE

The Assembler always stores the code it assembles in the Object Buffer irrespective of the ORG address supplied in the Source code. (A fuller explanation is in section 2.6.1). When you Save the Object code to Disc, the code is Saved from the Object buffer (probably not its final run location), but the Header information on the disc is made to contain the correct information about the Origin location. This means that you can Load that code back into memory from Disc with the Basic command.

LOAD "name"

and it will Load to the location given as the ORG in your Source code.

However, if you Save Object code to Cassette, there appears to be no way of creating a faked header containing the ORG location. For this reason, you must load Cassette based Object code from Basic with

LOAD "name",address where address = your ORG location.

This will force Basic to Load it to the correct address, but you must keep a note of the ORG location specified in your Source Code.

2.5.3 LOAD

The Assembler will **ONLY load a valid SOURCE CODE file** that has been previously Saved by the Assembler. Any other file will give you a "Wrong file type" error.

With the cursor at the left hand end of the EDIT line, press CTRL and the L key together. LOAD will be displayed on the screen. Type in a file name (or use CTRL + 4 if you have already defined the FILE name). Press ENTER, and the file will be loaded. Loading from Cassette will produce the usual cassette messages, and loading from Disc does not produce any messages.

The command form is

LOAD "name"

Before the new file is loaded, the Assembler will clear any existing Source Code from its buffer, and clear the label table to a null table. In other words, it will NEW the Assembler.

If you wish to retain the labels from a previous Assembly so that a new section of Source code can have access to those labels, you can simulate the CLEAR T command (see section 2.4.11) in a Load command by using the form

LOAD "name"C

This automatically does a CLEAR T before loading to protect the existing labels.

You can also APPEND a section of Source Code onto the end of the Source code currently in memory. If there are any labels currently in the label table, they will be deleted, and the new Source code will be added to the end of any Source code already in the buffer. The command form is

LOAD "name"A

Having Appended a section of Source code, you should RENUMber the whole listing to allow editing of the full listing, and you should also ensure that there is only ONE ORG and ONE END statement in the entire listing.

When loading from DISC, the Assembler will assume a Type part in the file name of ".SCE" unless you have specified a Type part in the LOAD command.

When loading from CASSETTE, LOAD"" will load the next Source file on the tape.

2.5.4 VERIFY

This command does not exist in Basic, but the Assembler allows you to Verify a Saved file against memory. The Verify command knows from the file's header whether it is Source Code or Object Code, and checks the appropriate buffer.

With the cursor at the left hand end of the bottom screen line, press CTRL and the V key together. VERIFY will be displayed on the screen. Now type in a file name (or use CTRL + 4 if you have defined a FILE name) and press ENTER.

The command form is

VERIFY "name"

If you are reading from cassette, the screen messages will imply a LOAD, but the Assembler is only checking the information loaded against the values held in memory.

If you are Verifying from Disc, and do not specify a Type part in the file name, the Verify command will always default to ".SCE". Therefore, to Verify an Object code file (normally Saved as ".BIN") you must specify the ".BIN" in the file name as follows

VERIFY "name.BIN"

2.5.5 External DISC commands

External commands have already been dealt with in detail in section 2.4.15. If you need to **REN**ame or **ERA**se a Disc file, or call up a **DIR**ectory, you can do this from the Assembler without the need to go back to Basic.

However, if you need to CAT a Disc, you will have to return to Basic (CRTL + B) to do this.

2.6 The ASSEMBLER

The ASSEMBLER converts the mnemonics entered in the listing into the Hex code that the CPU can understand. The mnemonics in the listing are called the Source Code and the assembled Hex code is known as the Object Code.

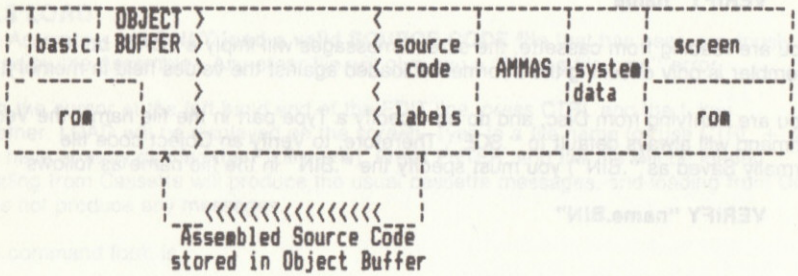
The ASSEMBLER makes two passes through the listing. The first pass checks for correct syntax; calculates the values of the Labels and creates a table of those values: and creates an outline Object File. The second pass calculates fully all the numeric and Label operands, and calculates the offsets of relative jumps. It is not possible to separate the two passes. The results of the second pass can be displayed either to the screen or to a Printer.

The Editor/Assembler resides at the top of memory and sets HIMEM to a very low value as described in Section 2.3, and all its tables and buffers are allocated in the space between HIMEM and the bottom of the Assembler. This allocation of table/buffer space is transparent to the user, which simplifies programming, and a "Memory Full" error is given if two areas are in danger of overwriting each other.

As you enter lines of Source code into the Editor, the Source code buffer expands downwards below AMMAS, and the label table expands downwards below the Source Code during the Assembly process. The Assembled code is stored in the Object Buffer which starts immediately above HIMEM and expands upwards, towards the label table.

2.6.1 The Object Buffer

To allow you to Assemble code that could ultimately reside in any part of memory, and also to give the Assembler the maximum amount of working space at all times, the Assembled code is ALWAYS stored in the Object Buffer, irrespective of the ORGIN location specified in your Source code.



The Assembler will calculate all addresses during assembly based on the ORIGIN address supplied in the Source code; the code is simply stored in the Object Buffer, ready for Saving when the Assembly process is complete. The Machine Code produced by the Assembler is designed to run at the memory location given in the ORG command, which is not normally the Object Buffer area, and the machine code would be loaded back to the correct location using the Basic Load command. At that stage, the Editor Assembler would not be required in memory, thus freeing a large part of the memory for your own program.

When the Assembler is first Loaded, HIMEM is set to a very low value, and this value is passed to the Assembler for it to use as the start of the Object Buffer. If you return to Basic from the Assembler, and alter the value of HIMEM (which you will need to do if you want to Load a Basic program) then you must tell the Assembler the new value of HIMEM next time you access it. To do this, access the Assembler using, |ASS,HIMEM + 1. If you simply use |ASS, then the Assembler will carry on using the old Object Buffer, which will probably corrupt your Basic program.

The Assembler will store the new HIMEM address, and will automatically define a Label with that value. This label is given a single character name of " # ". Every time you Assemble a piece of Source Code, you will always find this label at the end of the Label Table. For most of the time you will not need to use the # label, but, if you load both the Monitor and the Assembler together, you can use it as your Origin and save time on Saving and Loading while you test your Machine Code. See Section 3 for more details.

2.6.2 Numbers

The ASSEMBLER will accept Decimal or Hex numbers, and will default to Decimal. Numbers **must** start with a numeric digit, i.e. 0 to 9, and Hex numbers **must** have a suffix H. If a Hex number starts with a letter (A to F) it must be preceeded by a zero, or the ASSEMBLER will treat it as a Label. i.e. D000 Hex must be written 0D000H, and FF Hex as 0FFH.

Decimal numbers need no suffix. Decimal numbers between 0 and 65,535 are evaluated to their corresponding Hex value. Decimal numbers greater than 65,535, up to 99,999 are evaluated, but the carry produced is ignored. (i.e. 65,536 is evaluated to 0000 Hex and – 1 is evaluated to FFFF Hex).

2.6.3 ASCII Characters

Single ASCII characters are accepted as operands, providing that the character is entered within normal quotes (Shift and 2). This type of operand is accepted by all operations that can accept a numeric operand, except the Assembler Directive ORG.

When you type the first quote, the ASSEMBLER automatically produces lower case letters unless you use the Caps Shift.

Example:— **LD A, "a"** produces the code **3E 61.**

2.6.4 Arithmetic

Addition and Subtraction within an operand is allowed. The sum or difference of Label values, numeric values or ASCII characters in any combination is accepted.

Examples:—

(i) LD A, "A" + 20H produces the code 3E 61

(ii) LD HL, LBL2 — LBL1 calculates the difference between Labels.

(iii) LD DE, START + 100H adds a fixed offset to a Label.

2.6.5 Labels

In Assembly language programming, Labels are used to indentify points in the program. Whereas in Basic you would GO TO a line number, in Assembly language, you would JP (jump) to a Label name. The line numbers in your Source code listings are only there to maintain the correct order of the instructions and to allow editing.

You can also use labels to store the values of constants and variables. For example, to use the Firmware ROM routine to print on the screen, you would CALL OBB5AH. To make your Source Code easier to understand, you could define a label (e.g. TXTOUT) with a value of OBB5AH and then CALL TXTOUT. Using this method also allows you to make changes very quickly by altering the definition of a label. All subsequent uses of that label will also be altered to the new value. Section 2.7.3 and 2.7.4 give details of using labels as constants and variables.

All Labels have a two byte, 16 Bit value. Label names can be defined with a maximum of 6 characters. They **must** start with a non-numeric character and cannot contain spaces. All register names and abbreviations for conditions are reserved names, and cannot be used as a Label name, as the Assembler will try to evaluate them as a register or condition. The first character of a label name must not be a semi-colon, or it will be recognised as a comment line. (See section 2.4.17), nor must it be a '/' as that could be misunderstood as a printer command by LIST. To avoid any possible confusion, always start a label name with a LETTER.

ORDINARY LABELS
Label names put into the Label Field and followed by an operation other than EQU or DEFL are given the value of the current program address for that instruction. They are not re-definable, and can only be defined once in a program. An ordinary label used as an operand must be defined somewhere in the program listing, but can be defined in a line after it is used as an operand. This allows forward jumps to a Label name. Any Label can be used as an operand, providing that its value is within the limits for that operation.

The label table display at the end of an assembly pass can be inhibited, or can be called up independately of an assembly pass. The label table is always created by an assembly pass, and this command only controls its display.

LABEL With the cursor at the left hand end of the bottom screen line, press CTRL and the K key. LABEL is displayed on the screen. Pressing ENTER now will display the label table to the screen (unless it has been disabled). The screen will scroll through to the end of the table display and can be paused by ESC.

LABEL 0 Press CTRL and K as above, followed by the Zero key. This will disable the label table display.

LABEL 1 Press CTRL and K as above, followed by the 1 key. The label table display will be enabled and displayed on the screen.

LABEL / Press CTRL and K as above, followed by the / key. If the label table display is enabled, the table will be printed on the printer.

You can interrogate the label table to find the value given to any specific label by pressing CTRL and K as above, and then typing the name of the label. The value in Hex of that label will be displayed on the screen.

e.g. **LABEL LBL1** will display the value of LBL1.

2.6.6 Label Slicing

All labels are stored as a 16-BIT value even if they are defined as having a value of 255 or less. If their value is 255 or less, then the Assembler will treat them as either 8-BIT or 16-BIT, depending upon the instruction with which they are used.

e.g. If a label, LBL1 is defined as having a value of 80H, it will be stored as 0080H, and could be used as an 8-BIT value, by

```
LD A, LBL1
```

or as a 16-BIT value by

```
LD HL,LBL1
```

If its value is defined as greater than 255, it will always be treated as a 16-BIT number.

It is possible to access the high and low bytes of a label value separately by preceeding the label name when it is used as an operand by < to access the HIGH byte, or by > to access the LOW byte.

EXAMPLE. A label has been defined thus:

```
LBL1 EQU 5CBOH
```

The instruction `LD A,< LBL1`
will take the HIGH byte of the label, and is equivalent
to `LD A,5CH`

The instruction `LD A,> LBL1`
will take the LOW byte of the label, and is equivalent
to `LD A,OBOH`

The direction in which the < or > is pointing indicates which byte is taken.

The label does not have to be defined by EQU, and any label can be 'sliced' in this way. It is important to remember that ALL label names should start with a LETTER. Never start a label name with '>' or '<' as the Assembler will produce errors when that label name is used as an operand.

2.6.7 JR/DJNZ

The relative offsets for these two jump instructions can be defined in a number of ways. Normally you would use a label as the operand;

```
    JR    LOOP
or  JR    Z,LOOP  for a conditional jump
```

where the Assembler will calculate the correct forward or backward offset.

You can use a numeric offset if you wish, but this would cause errors in your program if you change the Source code within the limits of the jump without changing the JR instruction.

```
JR   2   }
         } jump forwards 2 bytes
JR  +2   }

JR  -2   }
         } jump backwards 2 bytes
JR  OFEH }
```

2.6.8 Assembling Source Code

To Assemble the Source Code that is currently in the Source code buffer, make sure that the cursor is at the left hand end of the bottom screen line, and press CTRL and the A key together. ASSEMBLE will be displayed on the screen. If you press ENTER, the Source code will be Assembled into the Object Buffer, and there will be no output to the screen or printer until the end of the Assembly. If you have enabled the label table

display, then the label table is shown on the screen, but if it is disabled (LABEL 0) then you will see the Assembler's cursor re-appear at the bottom of the screen when Assembly is complete.

If you require a full Assembly listing on the screen, then having pressed CTRL and the A key to display ASSEMBLE on the bottom screen line, you should then press the S key on its own followed by ENTER. The Assembler makes two passes through the Source Code listing and the first pass will not produce any screen output. There will be a delay before anything is printed on the screen which will vary according to the length of your Source Code. The second pass of the Assembler will produce a fully assembled listing consisting of the current program address in Hex, the Hex code produced and the line of Source code relating to that code. Producing a screen output will obviously slow down the assembly process quite considerably. The screen output will continously scroll up the screen throughout the assembly process. If there is a part of the listing that you wish to study, you can freeze the Assembly process by pressing ESC once. The Assembler's cursor will appear on the bottom screen line, and the Assembler will wait indefinitely until you either press SPACE to continue the Assembly process, or until you press ESC again to abort the assembly process.

You are more likely to want to produce a fully assembled listing onto your printer rather than the screen, and to achieve this, press CTRL and the A key to display ASSEMBLE on the screen, then press the / key (this key also has ? engraved on it) followed by ENTER. As with screen output above, there will be a delay while the Assembler makes its first pass through the Source Code, then the printer will be accessed to produce the assembled listing during the second assembly pass. If you press the ESC key once during the assembly process, the Assembler will stop and wait for either SPACE to continue or a second ESC to abort.

With both screen or printer output, the label table is displayed if it is enabled (LABEL 1)

To summarise,	**ASSEMBLE**	produces no screen or printer output.
	ASSEMBLE S	produces screen output.
	ASSEMBLE /	produces printer output.

The Object code is always produced, and stored in the Object Buffer.

2.7 ASSEMBLER DIRECTIVES

These are not operation names recognised by the CPU and are included to simplify the process of writing machine code. They are used by the ASSEMBLER, at assembly time, to create messages, data bytes, etc. They are entered into the listing, with a line number, by writing the Directive name into the operation field. They can be identified with labels and all require an operand except END.

Two of these Assembler Directives are compulsory in every program you write; namely ORG and END. An error message is produced if either is missing.

2.7.1 ORG

Defines the address of the first byte of the assembled code and therefore the base address from which all other label values are calculated. It must be defined in the listing before any other instruction. Comment lines can precede it, but it is good practice to define it in the first line you enter.

The Hex code produced by the ASSEMBLER is written into the Object Buffer from where it is Saved to cassette. Label # defines the start of this Buffer, and is set by the ASSEMBLER. The operand for ORG can be a number; a Label, so long as it has already been defined; or Label # . Only **ONE** ORG per program is allowed.

If a program is assembled with ORG # , the code in the Object Buffer is correctly assembled for the Buffer addresses, and can be run there (See Section 3). Any other ORG address will produce code in the Object Buffer designed to run from a different address.

2.7.2 END

This signifies the end of the program. Although there may be lines of program listing after the END statement, the ASSEMBLER will ignore them.

2.7.3 EQU

This Assembler Directive assigns a value to a Label name. It does not put any code into the Object Buffer. Labels are normally given a value equal to the current program address (based on your ORG), but EQU and DEFL will force a label to have a specified value. Once a Label is assigned a value by EQU, it cannot be re-defined. Any numeric value, ASCII character or other Label name is accepted as an operand, but if a Label name is used as an operand, it must already have been **defined in a previous line in the listing.**

The Label name is put into the Label field; EQU is put into the operation field and the operand in the operand field. A special operand for use with EQU and DEFL only is $ i.e. LABEL EQU $. This gives the label a value equal to the current program address.

2.7.4 DEFL

This Assembler Directive also allows you to define the Label value, but it also allows you to re-define the value as often as you wish within a program. At each new definition, the previous value is lost. A Label is only re-definable if its **first** definition in line order in the listing is by DEFL. Subsequent definitions can be by DEFL or EQU, but it remains re-definable. You cannot change an existing ordinary Label into a re-definable one. Having established a Label name using DEFL, you cannot use the same name for an ordinary Label to identify part of the program. It can only be re-defined by DEFL or EQU.

Example:

```
0001 ;.
0010          ORG   7000H
0020 LBL1     DEFL  "A"
0030          LD    A,LBL1
0040 ;
0050 ;***** **** ****************************
0060 ;
0070 ;              "Rest of program"
0080 ;
0090 ;***** **** ****************************
0100 ;
0110 LBL1     DEFL  LBL1+20H
```

In the above example, line 20 sets LBL1 to the ASCII value of "A" which is 41 H, and uses that value in line 30. Later in the program, LBL1 has 20 H added to it, which gives the ASCII value of "a", which is used in line 120.

When using other Label names as operands with EQU and DEFL, those Label names **must have been previously defined in the listing.**

As EQU and DEFL Labels are not strictly part of the program, and do not appear in the assembled code, they have no address related to them, and so during the assembly display, their **value** is shown in the left hand column, where you would expect to see an address.

At the end of the Assembly procedure, a table of Label values is produced, and DEFL labels are flagged by an asterisk.

2.7.5 DEFB

Assigns a value to the single byte at the current assembly address. The value must be less than 256 (0100H) and the operand can be a number, a single ASCII character, or a Label with a value less then 256.

It is useful for defining data bytes within a program.

Multiple operands are allowed, separated by commas, e.g.

```
DEFB    10,0E3H,255,3CH
```

The assembled value of each operand is displayed to the screen or printer at assembly time (if screen or printer output is requested). The first operand value is displayed in the normal position along with the program address and the line from the listing. The second and subsequent operand values are displayed on the next line.

2.7.6 DEFW

Assigns a value to the next two bytes at the current assembly address. The value is stored with the LSB first, followed by the MSB; i.e. in the normal way for a two byte value.

It is also useful for defining data bytes within a program.

Multiple operands are allowed, separated by commas, e.g.

```
DEFW    40000,7C80H,3CH,LBL2
```

The assembled value of each operand is displayed to the screen or printer at assembly time (if screen or printer output is requested). The first operand value is displayed in the normal position along with the program address and the line from the listing. The second and subsequent operand values are displayed on the next line.

2.7.7 DEFS

This creates a number of blank bytes from the current assembly address. The space created has its byte values set to 0. Its operand can be a number or a label name, providing that the **label has been defined in a previous line in the listing.**

You can use **DEFS** to create space for tables or buffers that your program will use when it is run. (e.g. Disc/cassette buffers). You can also use it to create variable amounts of blank space. For example, your program may require that a certain section of code starts on a page boundary. (A page is 256 bytes, and page boundries are multiples of 256 bytes. The LSB of the address of a page boundary will always be zero; e.g. 7F00 Hex). To ensure that a given instruction in your program does start on a page boundary, use the following line of Source code before that instruction:—

line no. **PAGE DEFS 0100H – > PAGE**

The label PAGE is the current program address, e.g. 7D5A Hex.
Using label slicing, > PAGE is the LSB of this address. e.g. 005A Hex.
So the blank space created is 0100H – 005AH = 00A6H.
The next instruction will be at PAGE plus the DEFS value
which is 7D5AH + 00A6H = 7E00H

This DEFS instruction will always ensure that the next instruction starts on a page boundary, irrespective of your ORG, and of any changes you make to the Source code before the DEFS.

2.7.8 DEFM

Allows messages to be entered into the listing as a string of ASCII characters. DEFM calculates the Hex code for each character, and puts that code into the current Object Buffer address. The operand must be a string of ASCII characters enclosed within quotes. When you type the first quote into the listing, the EDITOR automatically displays lower case characters unless CAPS SHIFT is pressed. When the cursor is positioned between string quotes, lower case letters are automatically produced. The first character of the operand **must** be a quote.

2.7.9 PRNT

Allows screen or printer output to be turned on and off within an assembly pass. PRNT is entered into the listing (in the operation field) at the points where the display is required. It does not affect the machine code that the ASSEMBLER produces.

If, for example, you have altered part of a listing, and require a print out of the section that has been altered, you can enable screen or printer output for that part of the Assembly process only, without sacrificing the increased speed of having no output for most of the assembly process.

PRNT S turns on screen output from the next line in the listing.

PRNT / directs the Assembly output to the Printer.

PRNT OFF turns off the Print facility.

The operands **S,/, OFF** must be placed in the operand field in the Source Code. Use ASSEMBLE (followed by ENTER) to give no screen or printer output until the PRNT directive requests it.

2.7.10 ENT

Defines the ENTry point of your program. It requires one operand, which can be a number or a label. When the assembled code is saved to Disc, the ENT address is loaded into the file header as the Run address so that the resulting Code can be loaded and run from Basic with RUN "name". If no ENT address is specified in your Source Code, then the ORG address is taken as the ENT address and put into the file header. As file headers cannot be overwritten when saving to cassette, (see Section 2.5.2 on Object Code), the ENT directive is only of use when saving Object Code to Disc. If you are using cassettes to save Object Code, and you require a Run address to be specified, you will need to Save the Object Code from the Assembler and load it back into its correct location from Basic as explained in Section 2.5.2, and then Save it again from Basic, adding a Run address.

2.8 MULTI-SECTION SOURCE FILES

To overcome the problems of writing very large programs, where there is insufficient memory for the total Source Code and Object Code in memory together, it is possible to store the Source Code in sections on cassette or disc, and to assemble the sections into one long Object Code file.

A maximum of 26 sections of Source Code is permitted, each with the same file name that also includes an identifying section letter. The first section must contain an ORG directive, and the last section must contain an END directive. Intermediate sections must contain neither. Labels in any section can be accessed from any other section, exactly as if the whole Source Code was in memory at the same time.

The principle of the operation is that each section of the Source Code is loaded into the Text Buffer in turn, with the Assembler executing the first of the two assembly passes on each section. Then each section of Source Code is again loaded into the Text Buffer for the second assembly pass. Therefore, with a limited size of Text Buffer, a very large Object Code buffer can be created.

For maximum effectiveness, each section of the Source Code should be around 750 to 1000 lines in length. It is impossible to give definite guidelines, but 10K of Object Code could be produced from 6 sections each containing around 1000 lines of Source Code.

When using Discs to store the sections of Source Code, the multiple assembly process is automatic, and the normal options of assembly output to the screen or printer or neither are available. Source Code sections can be stored on more than one Disc if required.

If a cassette recorder is used, prompting messages are displayed on the screen, indicating when to start, stop or rewind the tape to access the correct section. Each section of the Source Code on the cassette does not need to follow on immediately from the last, as the assembly process allows you time to wind the cassette to the correct location before continuing.

2.8.1 SAVING A SECTION OF SOURCE CODE

The Multi-Section Source Code commands for cassette and disc, are almost the same as for normal Source Code. (Section 2.5.2). The main difference is that in place of the suffix T after the file name, a section letter is specified thus

SAVE "name" # A
where "name" is the file name as normal
# indicates a multi-section Source Code file
A is the section identifier.

If you defined "name" as a **FILE** name, you can enter this command by typing

CTRL S for SAVE
CTRL 4 for "name"
CTRL 3 for " # "
followed by the Section letter.

N.B.
The " # " symbol is available from both SHIFT 3 and CTRL 3.

The Assembler will take the file name "name" (which must be identical for all sections of the same program) and add " # A" to the end of the name. This effectively reduces the maximum number of characters permissible in the file name by two.

When Saving the various sections, you must make sure that the section identifying letters are in order, and are consecutive letters. You cannot miss out letters in the sequence. You must also always identify the first section as " # A". When Saving to Disc, a Type part of ".SCE" is given unless you specify otherwise in the file name.

When you are making changes to various sections of Source Code as you refine your program, it is a great help to keep a record of the section name and section letter in a comment line at the start of the listing for each section. This will help to avoid the possibility of overwriting the wrong section, particularly on Disc, when you save the modified section of Source Code.

e.g. 0005 ; "name" # C
0006 ;
0010 (Source Code for Section C follows)

2.8.2 LOADING A SECTION OF SOURCE CODE

Alterations to any section of a Multi-Section Source Code file can be made by loading that section, and subsequently Saving it again with the same file name and section identifying letter. If you are using Discs, you can Overwrite the old file to save Disc space. The Load commands follow the normal rules described in Section 2.5 and 2.5.3 but you must specify the Section letters as follows:

LOAD"name" # A
Where "name" is the file name as normal
specifies a multi-section Source code file
A is the section letter

If you have defined "name" as a **FILE** name, you can enter this command by typing

CTRL L for LOAD
CTRL 4 for "name"
CTRL 3 for " # "
followed by the Section letter.

The Assembler will put the " # A" into the correct place in the file name for you.

2.8.3 VERIFYING A SECTION OF SOURCE CODE

Providing that a section of Source code has been Saved and is also the current section in memory, you can Verify the Save with

VERIFY "name" # A

2.8.4 ASSEMBLING A MULTI-SECTION SOURCE FILE

To Assemble multi-section Source Code files, with no screen or printer output, use the command

ASSEMBLE # "name"

Where ASSEMBLE is accessed by CTRL and the A key
# indicates multi-section Source files
"name" is the file name used when Saving the Source Code.

The Assembler automatically adds the identifying letters to the file name for each section.

If you have defined a **FILE** name with the "name" of the multi-part Source Code files, this command can be accessed by three key strokes:—

CTRL A	for ASSEMBLE
CTRL 3	for ". # "
CTRL 4	for "name"

If the Source Code files are stored on cassette, follow the screen prompts to set the tape to the required section and to press PLAY. The assembly process waits for you to wind the cassette to the correct place, so the sections of Source Code do not need to be consecutively recorded on the cassettes.

If the Source Code is stored on Disc, the whole process is automatic with screen prompts to indicate which section is being assembled. If you have stored your Source Code on more than one Disc (or on two sides of the same Disc), the Assembler will indicate that it has not found a particular section, and will wait for Discs to be changed. You will be offered the options to "Retry or Cancel". Press R to try to load that section again, or C to cancel the whole assembly.

If you require screen or printer output of the fully assembled listing, then you should type S (for Screen) or / (for Printer) before the # "name" in the Assemble command i.e.

ASSEMBLE S # "name"	for screen output
ASSEMBLE / # "name"	for printer output

At Assembly time, you may well get an Assembly error such as "jump out of Range". If this happens, amend the appropriate Source Code section and Save it again (see Section 2.8.1 and 2.8.2 for Loading and Saving) ensuring that you use the correct Section identifying letter. Having Saved the amended section, restart the Assembly process with the appropriate ASSEMBLE # command as described above.

To help explain the multi-section assembly process, try the following short example:—

1). Type in the following Source Code

```
0010          ORG   #
0020 TXTOUT   EQU   0BB5AH
0030 START    LD    HL,MSG
0040          LD    B,LEN
```

and Save with the command **SAVE "test"# A**

2). NEW the Source Code, and enter part two thus

```
0010 LOOP     LD    A,(HL)
0020          PUSH  HL
0030          PUSH  BC
0040          CALL  TXTOUT
0050          POP   BC
```

and SAVE with the command **SAVE "test"# B**

3). NEW the Source Code, and enter part three thus

```
0010          POP   HL
0020          INC   HL
0030          DJNZ  LOOP
0040          RET
0050 MSG      DEFM  "A multi-part Assembly"
0060 LEN      EQU   $-MSG
0070          END
```

and Save with the command **SAVE "test"# C**

4). Type in the command **ASSEMBLE S # "test"**

which will invoke the multi-part assembly and display the results on the screen.

The whole of the Object Code will be stored in memory in the Object Buffer and you will need to Save it to Disc/Tape as described in Section 2.5.2.

2.8.5 ASSEMBLY FROM DISC/TAPE TO DISC/TAPE

It is possible to extend the multi-part Assembly process to store the Object Code onto Disc/Cassette in 2k blocks as it is produced. This limits the size of the Object Buffer to just over 2k in length, which allows you to use up to 26 sections of Source Code that can each be made much longer than if the whole Object Buffer had to reside in memory.

The command form is:—

ASSEMBLE # "name","namecode"B for Binary code files
ASSEMBLE # "name","namecode"C for CP/M ".COM" files

where ASSEMBLE # "name" is identical to that described in section 2.8.4,
"namecode" is the name given to the Object Code file to be Saved,
B indicates a Binary code file (saved as "namecode.BIN"),
or C indicates a CP/M ".COM" file (saved as "namecode.COM").

If you are using **cassettes,** the screen prompts will be given as normal for a multi-part assembly, but during the second Assembly pass, you will occasionally be asked to **"Press PLAY and REC then any key".** This indicates that 2K of Object Code is ready to be Saved. When this message appears, make sure you have your **OBJECT CODE CASSETTE** loaded before recording. It is a good idea to knock out the record protect tab on your Source Code cassette before starting. You can always cover it with adhesive tape if you do need to re-record a Source Code file. This will eliminate the danger of over-writing your Source Code when asked to save a block of Object Code. When a block of Object Code has been saved to cassette, remove the cassette, but do not wind it in either direction. Replace the Source Code cassette and carry on.

If you have **Discs,** you can use any combination of DISC IN or OUT with TAPE IN or OUT. If you use DISC IN and OUT the only limitation is that the Disc drive storing the Object Code must NEVER have its disc changed during assembly. The DOS will produce an error if you do change a disc with an open file on it, and the assembly will be aborted. Therefore, if you only have one disc drive, the whole of the Source Code and the Object Code must be able to reside on the one side of one disc. This limits the maximum amount of Object Code to around 21K, which requires about 145K of Source Code. This will fill one side of a blank disc.

If you have two drives, you should specify the Drive parts in the filenames, with Source Code on one drive, and Object Code on the other;

e.g. **ASSEMBLE # "A:NAME","B:NAMECODE"B**

This will take Source from drive A and store Object on drive B, and will allow you to change discs in drive A if the Source Code will not fit onto one disc. Theoretically, this would allow you to produce up to 64K of Object code if you wished!!!

You can preceed the " # " with S for screen output, or / for printer output.

The size of the Object Buffer in memory is only ever just over 2K long, and it is dumped to disc or tape when the 2K is full. The 2K Cassette/Disc buffer at the end of the Assembler is used for the loading of the Source Code, and a separate 2K Cassette/Disc buffer is required for the Object code. The buffer is coincident with the Object Buffer, both buffers sharing the same 2K block of memory.

The first pass of the Assembler will happen as normal for a multi-part assembly, but before the second pass, the output file is opened with the usual "Overwrite / Abandon / Backup" options if the file already exists on disc.

At the end of Assembly, the Object buffer only contains the last 2K block to have been Saved, so it is not possible to Verify the Object Code or to run it in the Object Buffer.

2.9 ASSEMBLER ERROR MESSAGES

The Editor part of the Assembler will produce error messages under certain conditions as follows:—

BREAK

(i) If the ESC key is pressed while entering or editing Source Code.
(ii) If the ESC key is pressed during Loading/Saving operations.
(iii) If the ESC key is pressed TWICE during Listing or Assembly. (one press of ESC pauses Listing or Assembly — SPACE continues)

PRINTER OFF LINE

If printer output is requested and the printer is off line or otherwise appears BUSY for more than 3 seconds. If printer is put on line, printing commences. ESC ESC will exit if printer is not required.

MEMORY FULL

(i) If there is no room to enter a line of listing into Source Code.
(ii) During Assembly if there is no room to expand the Object Buffer.
(iii) During multi-part Assembly if there is not enough room to load the next section of Source Code.

SYNTAX ERROR

If a command is entered incorrectly, or with out-of-range parameters.

UNKNOWN COMMAND

If an External Command cannot be found in any ROM or RSX.

Disc/Tape errors are normally produced by the operating system and are documented in the AMSOFT manuals.

FILE ALREADY EXISTS

If a named file for Saving already exists on Disc.
Options given are Overwrite/Backup/Abandon.

FILE DOES NOT EXIST
Retry, Cancel

If a named file for Loading is not found on the current Disc.
Change Disc and **R**etry or **C**ancel to abort.

VERIFICATION FAILED

Data error on verifying a Saved file.

WRONG FILE TYPE

If non-valid Source Code is attempted to be loaded into the Assembler.

During the Assembly process, the following errors may be displayed on the screen. They will be followed automatically by a display in the EDIT mode of the line that contains the error. The cursor will be at the right hand end of the line, as if you had just EDITed the line.

INVALID ORG

(i) If there is not an ORG at the start of the program listing.
(ii) If there is more than one ORG in the listing. (The second occurence produces the error)

INVALID NUMBER

If a decimal number contains a non-numeric character, or if a Hex number contains an invalid character.

INVALID OP

(i) If an operation name is not recognised.
(ii) If a numeric operand has a value out of range for the particular operation (e.g. loading a single register with a Label whose value is greater than 255).
(iii) If an invalid mnenomic is entered. (e.g. if there is an incorrect number of operands).

LABEL NOT DEFINED

(i) If an operand is not a recognised register or condition and no label of that name has been defined.
(ii) If a Hex number does not commence with a number.
(iii) If EQU, DEFL or DEFS have an operand that is a label which has not been defined in a previous line of the listing, or in a previous section. (These directives cannot refer forward to labels).

LABEL ALREADY DEFINED

If a Label name is defined more than once and the first definition is not DEFL. The line containing the **second** definition is displayed.

NO END INSTR.

If there is not a line containing the END instruction.

JUMP OUT OF RANGE

If a relative jump (JR or DJNZ) is asked to jump to an address with an offset of more than +127 or −128.

2.10 SUMMARY OF COMMANDS

ASSEMBLE **CTRL A** Assemble Source code currently in memory into Object Buffer. No screen or printer output. ESC will pause; SPACE to continue; ESC again to abort.

ASSEMBLE S As above with assembled listing on screen.

ASSEMBLE / As above with assembled listing on printer.

ASSEMBLE # "name" Assembles Source code stored on DISC/TAPE with file name "name". Max 26 sections. No screen or printer output. ESC will pause; Space to continue; ESC again to abort. N.B. ESC during a LOAD will abort.

ASSEMBLE S # "name" As above with assembled listing on screen.

ASSEMBLE / # "name" As above with assembled listing on printer.

ASSEMBLE # "name","namecode"B or C Assembles Source code stored on DISC/TAPE with file name "name". Automatically stores Object Code back to DISC/TAPE in 2K blocks with file name "namecode.BIN", or "namecode.COM" No screen or printer output. ESC as above.

ASSEMBLE S # "name","namecode"B or C		As above with assembled listing on screen.
ASSEMBLE / # "name","namecode"B or C		As above with assembled listing on printer.
AUTO	**CTRL +**	Auto line numbering. Step value as last specified; from last auto line number.
AUTO x		Auto line numbering from line x with last specified Step value.
AUTO x,y		Auto line numbering from line x with Step value of y.
BASIC	**CTRL B**	Return to Basic
CLEAR T	**CTRL X**	Clears Source Code Buffer: Retains and protects all current labels.
CLEAR L		Clears all labels, including protected labels.
CLEAR O		Clears Object Buffer to zero length.
CLEAR P		Removes protection applied to labels by CLEAR T. Labels remain in table without protection.
COPY xx,yy,zz	**CTRL C**	Copies lines xx to yy inclusive in Source code and inserts them at or after line zz.
DELETE xx,yy	**CTRL D**	Deletes Source code from line xx to line yy inclusive.
EDIT	**CTRL E**	Displays first line of Source Code in Edit line at bottom of screen.
EDIT xx		Displays line xx of Source code in Edit line at bottom of screen.
ESC	**ESC**	While Editing or entering a command, aborts current Edit line. While Assembling or Listing, first press on ESC pauses function; SPACE to continue; ESC again to abort.
FILE "name"	**CTRL F**	Define a default File name for Save/Load functions.
LIST	**CTRL **	List Source code from start of listing onto screen. 10 lines displayed. SPACE gives next 10 lines.
LIST xx		List Source code to screen from line xx for 10 lines. SPACE give next 10 lines.

LIST xx –		As above.
LIST /		List Source Code to printer from start of listing. Continuous to end of listing.
LIST / xx		List Source Code to printer from line xx to end of listing.
LIST / xx –		As above.
LIST LABEL		List Source Code to screen from label LABEL for 10 lines. Space gives next 10 lines.
LIST /LABEL		List Source code to printer from label LABEL through to end of listing.
LOAD "name"	**CTRL L**	Loads Source code file from DISC/TAPE with file name "name" (only if it is valid Source code file). Deletes all existing Source code and Labels.
LOAD "name"N		As above.
LOAD "name"C		Loads valid Source code file from DISC/TAPE with file name "name'. Deletes previous Source Code but retains and protects current Labels.
LOAD "name"A		Appends valid Source Code file from DISC/TAPE with file name "name". New Source Code added to end of existing Source Code. Deletes all labels.
LOAD "name" # X		Loads section X of a multi-section Source file. Deletes all previous Source Code and labels.
LABEL	**CTRL K**	Displays label table to screen if enabled. ESC to pause: ESC again to abort, or SPACE to continue.
LABEL /		As above with printer output.
LABEL O		Disables label table display.
LABEL 1		Enables label table display, and displays table to screen.
LABEL NAME		Displays to screen the Hex value of label NAME, if it has been defined.
MONITOR	**CTRL M**	Access to Monitor AMMON if it is in memory.
MODE 1 X,Y	**CTRL 1**	Sets screen Mode 1. Optional screen colours: X = INK colour (0 to 26) Y = PAPER colour (0 to 26)

MODE 2 X,Y	**CTRL 2**	Sets screen Mode 2. Optional screen colours as above.
NEW	**CTRL N**	Deletes all Source Code and labels. Resets assembler buffers to null state.
RENUM X	**CTRL R**	Renumbers entire Source Code with step value of X.
RENUM X,Y		Renumbers Source Code from line Y to end in steps of X.
RENUM X,Y,Z		Renumbers Source Code from line Y to end in steps of X, giving line Y a new value of Z+X.
SAVE "name"T	**CTRL S**	Saves Source Code with filename of "name". (or "name.SCE" to Disc).
SAVE "name"B		Saves Object Code as Binary file with filename "name", ("name.BIN" to Disc).
SAVE "name"C		Saves Object Code as ASCII file with filename "name", ("name.COM" to Disc).

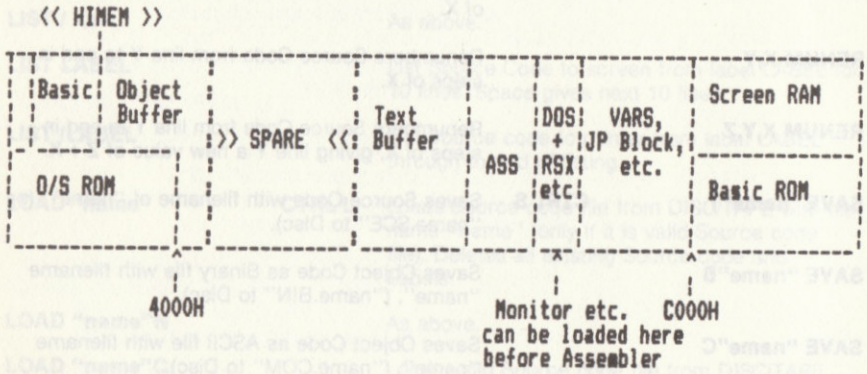
SAVE "name" # X		Saves a section of multi-part Source Code as "name # X", ("name # X.SCE" to Disc).
		X = Section suffix: First section must be # A; subsequent letters must be consecutive.

VERIFY "name"	**CTRL V**	Verifies incoming Source or Object Code with appropriate buffer (cassette). With Discs, Object code's Type part must be specified in name. i.e. VERIFY "name.BIN".
VERIFY "name" # X		Verifies incoming Source Code with current Source Code in memory for a section of multi-part Source Code.
CTRL @	**CTRL @**	Resets keyboard expansion strings. Also resets FILE name to null string.
CTRL left cursor		Moves cursor to left hand end of Edit line at bottom of screen.

SECTION 3

USING "AMMAS" and "AMMON" TOGETHER

Memory Map



LOADING ASSEMBLER & MONITOR TOGETHER

Both programs use a Basic loader program, so make sure that you do not have any important Basic in memory before loading. As both programs are set up as RSXs, the CPC464 should be as near to its EMS (switch on) state as possible. If you wish to reserve a block of memory at the top of the memory pool for your own machine code or RSX, then alter HIMEM before loading our programs. (Don't forget to do a SYMBOL AFTER 256 if you want to create further defined characters.)

Type RUN" (cassette) or RUN "MON" (Disc) to load the Monitor, and when asked for the "Start address?" reply by pressing ENTER. This will locate the Monitor at the highest available memory location, set HIMEM to immediately below the Monitor, set up the RSX and return you to Basic.

Type RUN" (cassette) or RUN "ASS" (Disc) to load and run the Assembler at the new highest possible memory location. HIMEM will be set very low and the Assembler will be entered.

In this way, there will be no conflict of memory between the Assembler and the Monitor, but the memory available to the Assembler will be reduced.

Use of the # label to test a program

When you are using **AMMAS,** it stores the value of HIMEM + 1 as the value of the " # " label, and uses this value as the start of the Object Buffer. This label can be useful when you are developing a Machine Code program with both **AMMAS** and **AMMON** in memory together.

Defining the ORG in your Source Code as **ORG #** will instruct AMMAS to take the first byte of the Object Buffer as its Origin when you assemble the Source code. The

resulting assembled code stored in the Object Buffer will, therefore, be assembled correctly to run in the Object Buffer location.

Having assembled the Source code, you can then access AMMON and use the Monitor commands to test your program. (You should already have Saved your Source Code in case your program crashes irretrievably). If you find an error in your programming, you can re-enter AMMAS, make a change and re-assemble the code, and test it again with AMMON. When you are sure that your program is error free, go back to AMMAS once more, and change the ORG address to the location where your Machine code will ultimately live, and re-assemble it. You can now Save the Object Code to Disc or Cassette, ready to use on its own, without requiring AMMAS or AMMON to be in memory.

SECTION 4

Z80 MNEMONICS

A full list of Z80 Mnemonics acceptable to AMMAS

```
ADC   A,(HL)
ADC   A,(IX+5)
ADC   A,(IY+5)
ADC   A,A
ADC   A,B
ADC   A,C
ADC   A,D
ADC   A,E
ADC   A,H
ADC   A,L
ADC   A,20H
ADC   HL,BC
ADC   HL,DE
ADC   HL,HL
ADC   HL,SP
ADD   A,(HL)
ADD   A,(IX+5)
ADD   A,(IY+5)
ADD   A,A
ADD   A,B
ADD   A,C
ADD   A,D
ADD   A,E
ADD   A,H
ADD   A,L
ADD   A,32
ADD   HL,BC
ADD   HL,DE
ADD   HL,HL
ADD   HL,SP
ADD   IX,BC
ADD   IX,DE
ADD   IX,IX
ADD   IX,SP
ADD   IY,BC
ADD   IY,DE
ADD   IY,IY
ADD   IY,SP
AND   (HL)
AND   (IX+5)
AND   (IY+5)
AND   A
AND   B
AND   C
AND   D
AND   E
AND   H
AND   L
AND   32
BIT   0,(HL)
BIT   0,(IX+5)
BIT   0,(IY+5)
BIT   0,A
BIT   0,B
BIT   0,C
BIT   0,D
BIT   0,E
BIT   0,H
BIT   0,L
BIT   1,(HL)
BIT   1,(IX+5)
BIT   1,(IY+5)
BIT   1,A
BIT   1,B
BIT   1,C
BIT   1,D
BIT   1,E
BIT   1,H
BIT   1,L
BIT   2,(HL)
BIT   2,(IX+5)
BIT   2,(IY+5)
BIT   2,A
BIT   2,B
BIT   2,C
BIT   2,D
BIT   2,E
BIT   2,H
BIT   2,L
BIT   3,(HL)
BIT   3,(IX+5)
BIT   3,(IY+5)
BIT   3,A
BIT   3,B
BIT   3,C
BIT   3,D
BIT   3,E
BIT   3,H
BIT   3,L
BIT   4,(HL)
BIT   4,(IX+5)
BIT   4,(IY+5)
BIT   4,A
BIT   4,B
BIT   4,C
BIT   4,D
BIT   4,E
BIT   4,H
BIT   4,L
BIT   5,(HL)
BIT   5,(IX+5)
BIT   5,(IY+5)
BIT   5,A
BIT   5,B
BIT   5,C
BIT   5,D
BIT   5,E
BIT   5,H
BIT   5,L
BIT   6,(HL)
BIT   6,(IX+5)
BIT   6,(IY+5)
BIT   6,A
BIT   6,B
BIT   6,C
BIT   6,D
BIT   6,E
BIT   6,H
BIT   6,L
BIT   7,(HL)
BIT   7,(IX+5)
BIT   7,(IY+5)
BIT   7,A
BIT   7,B
BIT   7,C
BIT   7,D
BIT   7,E
BIT   7,H
BIT   7,L
CALL  C,0584H
CALL  M,0584H
CALL  NC,0584H
CALL  NZ,0584H
CALL  P,0584H
CALL  PE,0584H
CALL  PO,0584H
CALL  Z,0584H
CALL  0584H
CCF
CP    (HL)
CP    (IX+5)
CP    (IY+5)
CP    A
CP    B
CP    C
CP    D
CP    E
CP    H
CP    L
CP    32
CPD
CPDR
CPIR
CPI
CPL
DAA
DEC   (HL)
DEC   (IX+5)
DEC   (IY+5)
DEC   A
DEC   B
DEC   BC
DEC   C
DEC   D
DEC   DE
DEC   E
DEC   H
DEC   HL
DEC   IX
DEC   IY
DEC   L
DEC   SP
DI
DJNZ  2EH
EI
EX    (SP),HL
EX    (SP),IX
EX    (SP),IY
EX    AF,A'F'
EX    DE,HL
EXX
HALT
IMO
IM1
IM2
IN    A,(C)
IN    B,(C)
IN    C,(C)
IN    D,(C)
IN    E,(C)
IN    H,(C)
IN    L,(C)
INC   (HL)
INC   (IX+5)
INC   (IY+5)
INC   A
INC   B
INC   BC
INC   C
INC   D
INC   DE
INC   E
INC   H
INC   HL
INC   IX
INC   IY
INC   L
INC   SP
IN    A,(32)
IND
INDR
INI
INIR
JP    0584H
JP    HL
JP    IX
JP    IY
JP    C,0584H
JP    M,0584H
JP    NC,0584H
JP    NZ,0584H
JP    P,0584H
JP    PE,0584H
JP    PO,0584H
JP    Z,0584H
JR    C,2EH
JR    NC,2EH
JR    NZ,2EH
JR    Z,2EH
JR    2EH
LD    (BC),A
LD    (DE),A
LD    (HL),A
LD    (HL),B
LD    (HL),C
LD    (HL),D
LD    (HL),E
LD    (HL),H
LD    (HL),L
LD    (HL),32
LD    (IX+5),A
LD    (IX+5),B
LD    (IX+5),C
LD    (IX+5),D
LD    (IX+5),E
LD    (IX+5),H
LD    (IX+5),L
LD    (IX+5),32
LD    (IY+5),A
LD    (IY+5),B
LD    (IY+5),C
LD    (IY+5),D
LD    (IY+5),E
LD    (IY+5),H
LD    (IY+5),L
LD    (IY+5),32
LD    (0584H),A
LD    (0584H),BC
LD    (0584H),DE
LD    (0584H),HL
LD    (0584H),IX
LD    (0584H),IY
LD    (0584H),SP
LD    A,(BC)
LD    A,(DE)
LD    A,(HL)
LD    A,(IX+5)
LD    A,(IY+5)
LD    A,(0584H)
LD    A,A
LD    A,B
LD    A,C
LD    A,D
LD    A,E
LD    A,H
LD    A,I
LD    A,L
LD    A,32
LD    A,R
LD    B,(HL)
LD    B,(IX+5)
LD    B,(IY+5)
LD    B,A
LD    B,B
LD    B,C
LD    B,D
LD    B,E
LD    B,H
LD    B,L
LD    B,32
LD    BC,(0584H)
LD    BC,0584H
LD    C,(HL)
LD    C,(IX+5)
LD    C,(IY+5)
LD    C,A
LD    C,B
LD    C,C
LD    C,D
LD    C,E
LD    C,H
LD    C,L
LD    C,32
LD    D,(HL)
LD    D,(IX+5)
LD    D,(IY+5)
LD    D,A
LD    D,B
LD    D,C
LD    D,D
LD    D,E
LD    D,H
LD    D,L
LD    D,32
LD    DE,(0584H)
LD    DE,0584H
```

```
LD    E,(HL)
LD    E,(IX+5)
LD    E,(IY+5)
LD    E,A
LD    E,B
LD    E,C
LD    E,D
LD    E,E
LD    E,H
LD    E,L
LD    E,32
LD    H,(HL)
LD    H,(IX+5)
LD    H,(IY+5)
LD    H,A
LD    H,B
LD    H,C
LD    H,D
LD    H,E
LD    H,H
LD    H,L
LD    H,32
LD    HL,(0584H)
LD    HL,0584H
LD    I,A
LD    IX,(0584H)
LD    IX,0584H
LD    IY,(0584H)
LD    IY,0584H
LD    L,(HL)
LD    L,(IX+5)
LD    L,(IY+5)
LD    L,A
LD    L,B
LD    L,C
LD    L,D
LD    L,E
LD    L,H
LD    L,L
LD    L,32
LD    R,A
LD    SP,(0584H)
LD    SP,HL
LD    SP,IX
LD    SP,IY
LD    SP,0584H
LDD
LDDR
LDI
LDIR
NEG
NOP
OR    (HL)
OR    (IX+5)
OR    (IY+5)
OR    A
OR    B
OR    C
OR    D
OR    E
OR    H
OR    L
OR    32
OTDR
OTIR
OUT   (C),A
OUT   (C),B
OUT   (C),C
OUT   (C),D
OUT   (C),E
OUT   (C),H
OUT   (C),L
OUT   (32),A
OUTD
OUTI
POP   AF
POP   BC
POP   DE
POP   HL
POP   IX
POP   IY
PUSH  AF
PUSH  BC
PUSH  DE
PUSH  HL
PUSH  IX
PUSH  IY
RES   0,(HL)
RES   0,(IX+5)
RES   0,(IY+5)
RES   0,A
RES   0,B
RES   0,C
RES   0,D
RES   0,E
RES   0,H
RES   0,L
RES   1,(HL)
RES   1,(IX+5)
RES   1,(IY+5)
RES   1,A
RES   1,B
RES   1,C
RES   1,D
RES   1,E
RES   1,H
RES   1,L
RES   2,(HL)
RES   2,(IX+5)
RES   2,(IY+5)
RES   2,A
RES   2,B
RES   2,C
RES   2,D
RES   2,E
RES   2,H
RES   2,L
RES   3,(HL)
RES   3,(IX+5)
RES   3,(IY+5)
RES   3,A
RES   3,B
RES   3,C
RES   3,D
RES   3,E
RES   3,H
RES   3,L
RES   4,(HL)
RES   4,(IX+5)
RES   4,(IY+5)
RES   4,A
RES   4,B
RES   4,C
RES   4,D
RES   4,E
RES   4,H
RES   4,L
RES   5,(HL)
RES   5,(IX+5)
RES   5,(IY+5)
RES   5,A
RES   5,B
RES   5,C
RES   5,D
RES   5,E
RES   5,H
RES   5,L
RES   6,(HL)
RES   6,(IX+5)
RES   6,(IY+5)
RES   6,A
RES   6,B
RES   6,C
RES   6,D
RES   6,E
RES   6,H
RES   6,L
RES   7,(HL)
RES   7,(IX+5)
RES   7,(IY+5)
RES   7,A
RES   7,B
RES   7,C
RES   7,D
RES   7,E
RES   7,H
RES   7,L
RET
RET   C
RET   M
RET   NC
RET   NZ
RET   P
RET   PE
RET   PO
RET   Z
RETI
RETN
RL    (HL)
RL    (IX+5)
RL    (IY+5)
RL    A
RL    B
RL    C
RL    D
RL    E
RL    H
RL    L
RLA
RLC   (HL)
RLC   (IX+5)
RLC   (IY+5)
RLC   A
RLC   B
RLC   C
RLC   D
RLC   E
RLC   H
RLC   L
RLCA
RLD
RR    (HL)
RR    (IX+5)
RR    (IY+5)
RR    A
RR    B
RR    C
RR    D
RR    E
RR    H
RR    L
RRA
RRC   (HL)
RRC   (IX+5)
RRC   (IY+5)
RRC   A
RRC   B
RRC   C
RRC   D
RRC   E
RRC   H
RRC   L
RRCA
RRD
RST   0
RST   8H
RST   10H
RST   18H
RST   20H
RST   28H
RST   30H
RST   38H
SBC   A,(HL)
SBC   A,(IX+5)
SBC   A,(IY+5)
SBC   A,A
SBC   A,B
SBC   A,C
SBC   A,D
SBC   A,E
SBC   A,H
SBC   A,L
SBC   A,32
SBC   HL,BC
SBC   HL,DE
SBC   HL,HL
SBC   HL,SP
SCF
SET   0,(HL)
SET   0,(IX+5)
SET   0,(IY+5)
SET   0,A
SET   0,B
SET   0,C
SET   0,D
SET   0,E
SET   0,H
SET   0,L
SET   1,(HL)
SET   1,(IX+5)
SET   1,(IY+5)
SET   1,A
SET   1,B
SET   1,C
SET   1,D
SET   1,E
SET   1,H
SET   1,L
SET   2,(HL)
SET   2,(IX+5)
SET   2,(IY+5)
SET   2,A
SET   2,B
SET   2,C
SET   2,D
SET   2,E
SET   2,H
SET   2,L
SET   3,(HL)
SET   3,(IX+5)
SET   3,(IY+5)
SET   3,A
SET   3,B
SET   3,C
SET   3,D
SET   3,E
SET   3,H
SET   3,L
SET   4,(HL)
SET   4,(IX+5)
SET   4,(IY+5)
SET   4,A
SET   4,B
SET   4,C
SET   4,D
SET   4,E
SET   4,H
SET   4,L
SET   5,(HL)
SET   5,(IX+5)
SET   5,(IY+5)
SET   5,A
SET   5,B
SET   5,C
SET   5,D
SET   5,E
SET   5,H
SET   5,L
SET   6,(HL)
SET   6,(IX+5)
SET   6,(IY+5)
SET   6,A
SET   6,B
SET   6,C
SET   6,D
SET   6,E
SET   6,H
SET   6,L
SET   7,(HL)
SET   7,(IX+5)
SET   7,(IY+5)
SET   7,A
SET   7,B
SET   7,C
SET   7,D
SET   7,E
SET   7,H
SET   7,L
SLA   (HL)
SLA   (IX+5)
SLA   (IY+5)
SLA   A
SLA   B
SLA   C
SLA   D
SLA   E
SLA   H
SLA   L
SRA   (HL)
SRA   (IX+5)
SRA   (IY+5)
SRA   A
SRA   B
SRA   C
SRA   D
SRA   E
SRA   H
SRA   L
SRL   (HL)
SRL   (IX+5)
SRL   (IY+5)
SRL   A
SRL   B
SRL   C
SRL   D
SRL   E
SRL   H
SRL   L
SUB   (HL)
SUB   (IX+5)
SUB   (IY+5)
SUB   A
SUB   B
SUB   C
```

```
SUB  D          SUB  32         XOR  A          XOR  E
SUB  E          XOR  (HL)       XOR  B          XOR  H
SUB  H          XOR  (IX+5)     XOR  C          XOR  L
SUB  L          XOR  (IY+5)     XOR  D          XOR  32
```

Extra Z80 Instructions.

There are a number of Z80 instructions that are not published or documented by Zilog, but which appear to work on all Z80s. They have been found by experiment on some of the apparently missing Hex code sequences. They are mainly operations on the individual halves of the IX and IY register pairs.

AMMAS will accept these instructions in the forms shown below, but AMMON will decode them to their HL equivalents. Generally, the MSB of IX is called XH and the LSB is called XL; the MSB of IY is called YH and the LSB YL. There is one extra arithmetic instruction, SLL, which Shifts the byte to the left by one BIT, puts BIT 7 into the CARRY and SETs BIT 0. The flag results for SLL are as for SRL.

The extra instructions are:—

```
SLL  (HL)
SLL  (IX+5)
SLL  (IY+5)
SLL  A
SLL  B
SLL  C
SLL  D
SLL  E
SLL  H
SLL  L
INC  XH
INC  YH
DEC  XH
DEC  YH
LD   XH,32
LD   YH,32
INC  XL
INC  YL
DEC  XL
DEC  YL
LD   XL,32
LD   YL,32
LD   B,XH
LD   B,YH
LD   B,XL
LD   B,YL
LD   C,XH
LD   C,YH
LD   C,XL
LD   C,YL
LD   D,XH
LD   D,YH
LD   D,XL
LD   D,YL
LD   E,XH
LD   E,YH
LD   E,XL
LD   E,YL
LD   XH,B
LD   YH,B
LD   XH,C
LD   YH,C
LD   XH,D
LD   YH,D
LD   XH,E
LD   YH,E
LD   XH,XL
LD   YH,YL
LD   XH,A
LD   YH,A
LD   XL,B
LD   YL,B
LD   XL,C
LD   YL,C
LD   XL,D
LD   YL,D
LD   XL,E
LD   YL,E
LD   XL,XH
LD   YL,YH

LD   XL,A
LD   YL,A
LD   A,XH
LD   A,YH
LD   A,XL
LD   A,YL
ADD  XH
ADD  YH
ADD  XL
ADD  YL
ADC  XH
ADC  YH
ADC  XL
ADC  YL
SUB  XH
SUB  YH
SUB  XL
SUB  YL
SBC  XH
SBC  YH
SBC  XL
SBC  YL
AND  XH
AND  YH
AND  XL
AND  YL
XOR  XH
XOR  YH
XOR  XL
XOR  YL
OR   XH
OR   YH
OR   XL
OR   YL
CP   XH
CP   YH
CP   XL
CP   YL
```

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES