1 The Firmware

This manual describes the firmware of the Amstrad CPC 464/664/6128 microcomputers. It also describes the disc operating systems (CP/M and AMSDOS). It does not describe either the BASIC language supplied with the system or CP/M. The manual does describe certain aspects of the BASIC where these affect other programs and it uses BASIC in certain example programs when describing some features of the firmware. It also describes how to call the firmware from CP/M.

Three versions of the firmware are described. V1.0 (on CPC464), V1.1 (on CPC664) and V1.2 (on CPC6128). Apart from support for bank switching V1.1 and V1.2 are identical and are referred to as V1.1 throughout this manual. It may be necessary for a program to deduce which firmware is fitted in a computer, and this can be achieved by inspecting the on-board ROM's version number (as described in section 10.2) using KL ROM PROBE. This will return a 0, 1 or 2 depending on the version of firmware.

The firmware is the program that resides in the lower ROM and the disc controlling ROM (see section 2). Its function is to control the hardware of the computer and to provide useful facilities for other programs to use. This avoids every program written having to provide its own facilities.

This manual is expected to be of interest to anyone who would like to know how the system works. It is indispensable for programmers writing machine code programs, particularly system programs (e.g. other languages) and games.

The information presented can be extremely detailed. It covers the operation of the firmware from the lowest level (e.g. driving the sound chip) to the highest level (e.g. running a queue of sounds). It is not necessary to understand all the information given to be able to use the firmware, however, a good grasp of how the system works will aid the programmer in selecting the most appropriate method for performing a particular task.

Two disc operating systems are provided: AMSDOS, which enables BASIC programs to use disc files in much the same way as cassette files; and CP/M 2.2 the industry standard operating system (CP/M Plus in the CPC6128, but we do not discuss the differences between the two in this manual). Both AMSDOS and CP/M use the same file structure and may read and write each others file's.

CP/M is invoked from BASIC by typing |CPM. Part of CP/M (the CCP and BDOS) is loaded from the disc in drive A:. The CP/M BIOS resides in the disc ROM.

AMSDOS is enabled whenever BASIC is first used. This intercepts most of the cassette firmware routines and redirects them to disc. Thus existing BASIC programs which use cassette files can use disc files with little or no modification. AMSDOS also provides a number of external commands for erasing and renaming files and redirecting the cassette firmware routines.

Provided with the disc system are a number of utility programs for formatting and copying discs and for changing various system parameters. These all run under CP/M.

1.1 The Hardware.

The diagram on the following page gives an indication of the different pieces of hardware in the system and how they connect to each other. For more information on how the hardware works see Appendix XII and the relevant manufacturer's data sheets.

The system centres around the CPU (Central Processing Unit) which is a Z80A microprocessor with a 4MHz clock. Next in importance is the gate array which contains miscellaneous logic to control much of the system. In particular, it controls ink colours, screen mode and ROM enabling (see section 10 and Appendix XII). In conjunction with the CRTC (Cathode Ray Tube Controller), which is a 6845 chip, the gate array generates the video signals for the monitor.

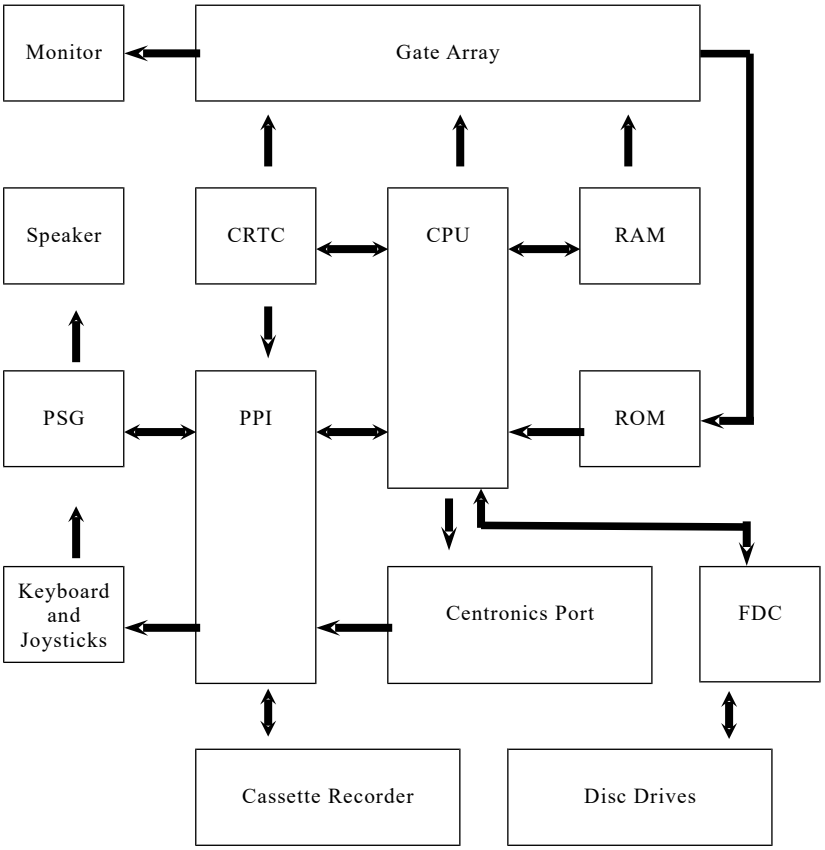
The PSG (Programmable Sound Generator) is an AY-3-8912. This chip has three channels of sound generator, a noise generator, envelope control for each channel and an I/O port. The way the sound generating hardware is used is described in section 7. The I/O port is used in input mode to sense the state of the keyboard and joystick switches.

The FDC (Floppy Disc Controller) is an NEC µPD765A chip. Only two disc drives are supported, since the US1 line from the µPD765A is ignored. This results in the two disc drives being accessed as drives 0 and 1 and again as 2 and 3. The FDC supports both single and double sided and single and double density mini-floppy disc drives. Note that the clock frequency supplied to the µPD765A CLK pin is 4MHz rather than the 8 MHz used with larger disc drives.

Each disc drive takes a single 3” floppy disc. Either side of the disc may be used, depending on which way up the disc is inserted into the drive. The disc interface contains a 16K expansion ROM, 8K of which contains the disc driving software, the remainder being used by DR LOGO.

The PPI (Parallel Peripheral Interface), which is an 8255 chip, is used to control the remainder of the system. It has three ports. Port C is used as an output port to control the cassette recorder motor, to write data to the cassette, to strobe data in or out of the PSG and to select rows of the keyboard. Port B is used as an input port to sense the frame flyback signal, the Centronics port busy signal and various option links and to read data from the cassette. Port A is used to communicate with the PSG and is set into input or output mode as required.

Accesses to memory are synchronised with the video logic - they are constrained to occur on microsecond boundaries. This has the effect of stretching each Z80 M cycle (machine cycle) to be a multiple of 4 T states (clock cycles). In practice this alters the instruction timing so that the effective clock rate is approximately 3.3 MHz.



1.2 The Division of the Firmware.

The firmware is split into 'packs' each dealing with a particular part of the system, usually a hardware device. Each pack has a section of this manual devoted to it where its operation is explained in detail. The system components and their associated packs are:

Keyboard:	Key Manager.
Screen:	Text VDU, Graphics VDU, Screen Pack.
Cassette/Disc:	Cassette Manager/AMSDOS.
Sound:	Sound Manager.
Operating System:	Kernel, Machine Pack, Jumper.

a. Key Manager

The Key Manager is more fully described in section 3. It deals with scanning the keyboard, generating characters, function keys, testing for break and scanning the joysticks.

b. Text VDU

The Text VDU is more fully outlined in section 4. It deals with putting characters on the screen, the cursor and obeying control codes.

c. Graphics VDU

The Graphics VDU is more fully presented in section 5. It deals with plotting points, testing points drawing lines and filling areas on the screen.

d. Screen Pack

The Screen Pack is more fully detailed in section 6. It interfaces the Text and Graphics VDUs with the screen hardware and deals with aspects of the screen that affect both of these packs, such as screen mode or ink colours.

e. Sound Manager

The Sound Manager is more fully discussed in section 7. It deals with queueing, enveloping, synchronising and generating sounds.

f. Cassette Manager/AMSDOS

The Cassette manager is more fully explained in section 8. It deals with reading from tape, writing to tape and cassette motor control.

AMSDOS is explained more fully in section 9. It deals with reading from disc, writing to disc and the disc motor control.

g. Kernel

The Kernel is more fully described in sections 2, 10, 11 and 12. It is the heart of the operating system and deals with interrupts, events, selecting ROMs and running programs.

h. Machine Pack

The Machine Pack is more fully documented in section 13. It deals with the printer and the low level driving of the hardware.

i. Jumper

Jumper, or rather, the main firmware jumpblock is listed in section 14. The entries in the jumpblock are described in detail in section 15. Jumper sets up the firmware jumpblock.

1.3 Controlling the Firmware.

The firmware is controlled by the user calling published routines rather than by the user setting the values of system variables. This will allow the firmware's variable layout to be changed in major ways without the user being affected.

The addresses of the routines the user is to call need to remain constant if the firmware is altered. This is achieved by using jumpblocks (see below).

The advantage of a routine interface is that it allows a number of different system variables to be altered by the firmware in a consistent way in one operation. If the system variables had to be set by the user then the firmware could be left in an indeterminate state if some variables had been set but not others. Also, the routine type of interface ensures that all the required side effects of a change are taken care of automatically without the user being troubled with all the details. An example of this is changing the screen mode (see section 6.1) - changing the size of the screen requires a number of other people to be informed of the change so that illegal screen positions and inks are not used.

1.4 Jumpblocks.

A jumpblock is a series of jump instructions placed in memory at well-known locations. The jumps are to the various routines in the firmware that the user might want to call. Programs that need to use the facilities provided by the routines in the jumpblock should call the appropriate jumpblock entries.

If the firmware is altered then it is quite likely that the addresses of some of the routines available to the user will change. By keeping the address of the jumpblock constant but altering the entries in the jumpblock so that they jump to the new addresses of the routines, the change is hidden from the user (providing that the user is only calling routines via the jumpblock and is not accessing the firmware directly).

To make the change to the firmware completely hidden from the user it is also necessary to keep the entry and exit conditions of the routines accessed via the jumpblock constant. The greater part of this manual is taken up with the detailed entry and exit requirements of the jumpblock entries.

The jumpblock is placed in RAM so that the user can alter the entries in it. This allows the user to trap particular entries and to substitute a new routine that will replace the standard firmware routine. Provided that the new routine obeys the entry and exit requirements of the firmware routine, the substitution will not upset programs unaware of the change.

There are four jumpblocks. These are all listed in section 14. The first and largest jumpblock is the main firmware jumpblock (see sections 14.1 and 15). This allows the user to call most firmware routines. The second jumpblock is the indirections jumpblock (see sections 14.2 and 16). The entries in this jumpblock are used by the firmware at key moments in order to allow the user to alter the action of the firmware. The last two jumpblocks are rather special. They are to do with the Kernel and allow ROMs to be enabled and routines in ROMs to be called. (See sections 14.3, 14.4, 17 and 18).

Section 1.7 below gives an example of how a jumpblock entry might be changed to alter the action of the firmware.

1.5 Conventions.

a. Notation

Processor instructions are generally referred to by their standard Z80 mnemonics. The exceptions that prove the rule are the restart instructions. The mnemonics RST 0 .. RST 7 are used rather than the more usual Z80 mnemonics RST #00.. RST #38.

The registers are also referred to by their standard Z80 names. The flag register as a whole is referred to as F but the individual flags are called by their full name, e.g. carry. The flags are said to be true when they are set and false when they are clear. Thus a JP NC instruction would jump if carry was false and not if carry was true.

Hexadecimal numbers are indicated by prefixing the number with #, thus #7F is the number 127 in hex. All numbers not prefixed by # are in decimal.

Large numbers are often abbreviated by writing them as a multiple of 1024. For example, 32K bytes means 32 times 1024 (i.e. 32768) bytes.

b. Usage

Routines, where possible, take and return values in registers. Where more information than may be held in registers is to be passed to a routine, the address of a data area is given. The location in memory of these data areas is sometimes critical, see section 2.4.

Where a routine can succeed or fail this condition is normally passed back in the carry flag. Carry true normally implies success, whilst carry false normally implies failure.

The alternate register set, AF' BC' DE' HL', is reserved for use by the system. The user should not execute either an EX AF,AF' or an EXX instruction as these will have unfortunate consequences. (See Appendix XI for a full description.)

c. General

The logical values true and false are generally represented by #FF and #00 respectively. Often, however, any non-zero value is taken to mean true.

The bits in a byte are numbered 0..7, with bit 0 being the least significant bit and bit 7 being the most significant bit.

Where two byte (word) values are stored (in tables etc) they are always stored with the less significant byte first and the more significant byte second, unless a specific indication to the contrary is given. This is in accordance with the standard way the Z80 stores words.

Tables and the like are always laid out with byte 0 being the first byte of the table. When the address of such a table is given this is the address of byte 0 of the table unless otherwise indicated.

When the computer is turned on (or when it is reset) it completely initializes itself before running any program. This initialization is known as early morning startup, abbreviated to EMS from now on.

1.6 Routine Documentation.

Each routine described in this manual has entry and exit conditions associated with it. Where there are other points of interest about the routine these are normally given in a section after the entry and exit conditions. Such points include whether interrupts are enabled and a fuller description of the parameters and side effects of the routine.

There are two reasons for providing this information. Firstly it tells the user what will happen when the routine is called. Secondly it tells the user what a replacement routine is expected to do.

> The entry conditions tell the caller of the routine what the routine expects to be passed to it. When calling a routine all values specified must be supplied. Values may only be left out where the routine documents that they are optional. When providing a replacement routine to fit this interface only information that is specified may be used, although not all of it need be used.
>
> The exit conditions tell the caller what values the routine passes back and which processor registers are preserved. Registers that are documented as being corrupted may be changed by the routine or may not. The user should not rely on their contents. When providing a routine to fit this interface it is extremely important that registers documented as being preserved are indeed preserved and that the values returned are compatible with the original routine. Corrupting a register or omitting a result will usually cause the system to fail, often in subtle and unexpected ways.

Often a routine will have different exit conditions depending on some condition or other (usually whether it worked or not). In these cases the specific differences in the exit conditions are given for each case and all conditions that remain the same irrespective of the case are given in a separate section (marked 'always').

There are abundant examples of routine interfaces in sections 15 to 18.

1.7 Example of Patching a Jumpblock.

The following is an example of how the jumpblocks may be used. At this stage many of the concepts introduced may be unfamiliar to the reader. However, since altering jumpblocks is an important technique for tailoring the system to a particular purpose the example is given here. Later sections will explain the actions taken here.

Suppose an assembler program is being written that is intended to use the printer when it is finished. While this program is being written it would save time and paper if the program could be made to use the screen instead of the printer. However, changing the program itself to use the screen could introduce bugs when it is changed back to using the printer. What is needed is a way of altering the action of the firmware that drives the printer - and this is what a RAM jumpblock is for.

The technique that will be used is to ‘connect’ the printer to a particular text window. This can be achieved by writing a short routine to send the character to the screen and patching the entry in the jumpblock for sending characters to the printer, MC PRINT CHAR, so that it jumps to this routine instead of its normal routine.

The substitute routine will have to obey the entry/exit conditions for MC PRINT CHAR. These can be found in the full description of this entry in section 15. Briefly they are as follows:

MC PRINT CHAR:

Entry conditions:

A contains character to print.

Exit conditions:

If the character was sent OK:
Carry true.
If the printer timed out:
Carry false.

Always:

A and other flags corrupt.
All other registers preserved.

The action of the substitute routine will be to select the screen stream that the printer output is to appear on, to print the character on the stream and then to restore the stream that was originally selected. To do this the substitute routine will need to call the routines TXT STR SELECT and TXT OUTPUT. Once again the full descriptions of these jumpblock entries can be found in section 15. The entry/exit conditions are as follows:

TXT STR SELECT:

Entry conditions:

A contains stream number to select.

Exit conditions:

A contains previously selected stream number.
HL and flags corrupt.
All other registers preserved.

TXT OUTPUT:

Entry conditions:

A contains character to print.

Exit conditions:

All registers and flags preserved.

The code for the substitute routine could be written as follows (stream 7 has been chosen as the stream on which printer output is to appear):

```
PUSH HL
PUSH BC
;
LD B, A                         ;Save the character to print
;
LD A, 7                         ;Printer stream number
CALL TXT_STR_SELECT             ;Select the printer stream
LD C, A                         ;Save the original stream number
;
LD A, B                         ;Get the character again
CALL TXT_OUTPUT                 ;Send it to the screen
;
LD A, C                         ;Get the original stream number
CALL TXT_STR_SELECT             ;Reselect the original stream
;
POP BC
POP HL
SCF                             ;The character was sent OK
RET
```

Note the following points:

1/ MC PRINT CHAR preserves HL and BC. The routine above uses B and C for temporary storage and HL is corrupted by TXT STR SELECT. HL and BC are therefore pushed and popped to preserve them through the substitute routine.

2/ MC PRINT CHAR returns a success/fail indication in the carry flag. Since the routine above can never fail it always sets the carry flag to indicate success.

3/ The routine above does not change which text stream is selected. It selects the stream it is going to print on and restores the previously selected stream when it has printed the character. The firmware is written in such a way as to allow routines to restore the original state when they finish if required.

To use the substitute routine it is necessary to patch it into memory and to change the jumpblock entry for MC PRINT CHAR to jump to it. Assume that some memory at #AB00 has been reserved for the substitute routine and that the routine has been patched into memory. The MC PRINT CHAR entry in the jumpblock is at location #BD2B (as can be seen by inspecting section 13.1.8). The three bytes of the entry should be set to the instruction JP #AB00 by patching as follows:

```
#BD2B    #C3
#BD2C    #00
#BD2D    #AB
```

From now on all text sent to the printer will appear on the screen on stream 7. Of course, stream 7 should have its window set so that it does not interfere with any other streams using the screen.

This redirection will remain in force until the jumpblock entry is restored. This can be achieved by patching the jumpblock back again or by calling JUMP RESTORE or by causing an EMS initialization to take place by resetting the system.