Understanding and Expanding YOUR

AMSTRAD CPC-464/664/6128

Alan Trevennor

SIGMA PRESS

Understanding and Expanding Your Amstrad CPC 464, 664 and 6128

Alan Trevennor

Σ
SIGMA
PRESS

Copyright © Alan Trevennor, 1986

All Rights Reserved

No part of this book may be reproduced or transmitted by any means without prior permission of the publisher. The only exceptions are for the purposes of review, or as provided for by the Copyright (Photocopying) Act or in order to enter the programs herein onto a computer for the sole use of the purchaser of this book.

ISBN 1-85058-040-5

Published by:
SIGMA PRESS
98A Water Lane
Wilmslow
Cheshire
U.K.
Printed by Interprint Limited

Distributors:

U.K., Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester
West Sussex, England

Australia:
JOHN WILEY & SONS INC.
GPO Box 859, Brisbane
Queensland 40001
Australia

Acknowledgements: CPC–464, CPC–664 and CPC–6128 are the Trade Marks of Amstrad Consumer Electronics plc.

PCB and kits service

Kits of parts for any or all of projects in this book, including professionally designed high quality printed circuit boards, are available in ready made or kit form. These include an eight slot extended expansion bus motherboard including power supply – as detailed in appendix five and chapter three.

The sole supplier of the kits, special printed circuit boards, and ready made projects is:

Halstead designs,
5 High St,
Halstead,
Essex.

Telephone (0787) 477408

Software supply service

A complete set of all the software in this book is available on disc only from:

Reader software service
Amsoft
Brentwood house
Brentwood
Essex CM14 5YA

Telephone (0277) 230222

The disc supplied will also contain several programs not listed or described in this book, along with some text files.

A special note for CPC 664 and 6128 users.

This book was started when only the CPC 464 was available. Because the amount of developement work involved for this book was considerable it took rather longer to complete than either the publisher or myself anticipated. In the interim, Amstrad launched the CPC 664 and the 6128. It is a measure of the consistency of Amstrad computer products that these newer machines maintain compatability with the CPC 464 in most respects. This means that nearly all the material contained in this book can be applied to the newer machines: 664 and 6128 users can expect the projects detailed within these covers to be for the most part compatible. There are instances where the application of some software (like the cats printer program in Chapter 5) will not be applicable for non cassette machines, but these are marked up for CPC 464 owners only.

Descriptions of disc related subjects have deliberately been avoided as another Sigma book, "The Amstrad disc companion" by Simon Williams, is available on the Amstrad disc subsystem and its supporting software.

PREFACE

This book has been written with the aim of giving its readers information in at least two areas. Firstly, information to provide a good understanding of the hardware inside the Amstrad CPC family of home computers. (The disk hardware and software is not detailed – another Sigma book, "The Amstrad Disk Companion" by Simon Williams, does this.) Secondly, some practical projects which have been developed specifically as CPC peripherals. Nearly all these projects have had driving software written for them, and full details of this are included. The projects will not only broaden the scope of your computing activities, but several of them are designed to make the computer faster or more easy to use and provide yet more facilities on it.

Readers who have so far concentrated their attention on software, are welcomed to the wonderful world of hardware, you are assured that within this book is enough information to get you started as a chip freak, as well as presenting a veritable feast of information about software to keep the chips under control!

Much of the slog of hardware construction has been abolished by the design of some Printed Circuit boards (PCBs). These are designed and are to be available from Halstead Designs Ltd who are also preparing component kits and ready assembled versions of the projects. Amsoft are to sell complete sets of the software via their user software service. (Full details of these services may be found on page three of this book). The availability of these makes it a far more easy process to build any of the described projects which takes your fancy.

I do not recommend any particular reading order, except to readers who know little or nothing about computer hardware, who should definitely read appendix six first. Appendix six is an introduction to some of the basic concepts involved.

I would like to thank all the people who have contributed in one way or another to the completion of this book. to the many people at Amsoft Brentwood whose brains I have picked, most especially to William Poel for his goodwill and cooperation. To Geoff Melvin and Wally Maynard – my bosses at Systime who assisted in so many ways. To Graham Beech at Sigma for his encouragement, and for his patience as each new completion deadline passed. Thank you all.

But the biggest thank you must go to my wife, Wendy, who has put up with spending a great many long evenings alone, with only the distant clicking of a keyboard, and the odd faintly heard cry of despair, to remind her that she still has a husband! Thanks love.

Dedicated to Wendy with love.

Alan Trevennor December, 1985.

CONTENTS

Chapter One: Hardware Overview 1
The chips inside the CPC 1
The Z80A 1
The 8255 PPI 1
The ROM 7
The RAM chips 8
The VDU Controller chip 11

Chapter Two: Z80 Microprocessor Primer 16
Signal Timings 20
Z80 Programming 22
Z80 Instruction Set in Alphabetical Order 26
Z80 Instruction list 27

Chapter Three: Hardware Projects 43
The Sigma Power Supply Unit 43
Project 1: RS232 Interface – Version "A" 46
Multiproject Board 58
Project 2: Speech Synthesiser 58
Project 3: Adding an External Keyboard 71
Project 4: Key Matrix Port 77
Project 5: Sense Register 86
Project 6: Parallel Transfer Control 89
Project 7: RS232 Version "B" Interface 99
Project 8: EPROM Programmer 128
Project 9: ROM Disk 145
Expansion ROM Board 183

Chapter Four: Using Amstrad Firmware 191
Hardware control features in Locomotive BASIC 191
Interactions between BASIC and Hardware 193
Interrupts are our Friends! 193

Chapter Five: Developing Assembler Language Programs 195

What is an Assembler Program? 196
Non machine-code elements of assembly language 197
Getting going 198
Writing programs in assembly language 199
Software for hardware project five 201
Developing programs from flowcharts 208
Wot! No Assembler? 225

A useful program written in assembly language . . . 227
Another confession – shock-horror probe! . . . 232
Do not use these instructions . . . 235

Chapter Six: An Advanced Project . . . 238
Local Network Project . . . 238
CPCNET Facilities and Limitations – in detail . . . 239
Limitations . . . 239
Transmission headers . . . 239
Detailed description of the software in the CPCNET background ROM . . . 240
The CPCNET Hardware . . . 252
CPCNET Applications . . . 253
CPCNET ROM Listing . . . 254
CPCNET Conclusion . . . 265

Appendix One: Circuit Symbols and General Constructional Notes . . . 266

Appendix Two: Further Reading . . . 269

Appendix Three: Useful Chip Data . . . 271

Appendix Four: Miscellaneous Data – pin connections and ASCII character set . . . 277

Appendix Five: The CPC Expansion Bus . . . 279

Appendix Six: Begin Here – for readers new to computers or new to hardware . . . 286

Appendix Seven: The Printer Port . . . 317

Index . . . 319

CHAPTER 1

Hardware overview

The CPC hardware consists of two chunks. Chunk one is the monitor. The monitor comes in two types, the colour or the monochrome. In both cases the monitor houses not only the Cathode Ray Tube (CRT) assembly, the front of which is the screen, but it also contains the power supply for the second chunk.

Chunk two is the keyboard unit. This contains the clever bit of the machine, the Z80A microprocessor. Also inside the keyboard unit are the memory (able to hold 65536 bytes of data – or if you prefer, think of it as 65536 alphabetic characters), the chips which allow communication with the keyboard, the chip which makes all those lovely sound effects and musical noises, and the cassette recorder and its associated electronics. Additionally there is a custom designed chip, unique to the CPC, which drastically reduces the number of chips which would otherwise be required. There is the video display controller chip. Finally, there is the ROM (Read Only Memory) which has had permanently written into it the operating system, and the BASIC interpreter which controls all the hardware, and allows you to program the computer in the BASIC programming language.

In this first chapter we will take a look at these various chips and see how they are used in the CPC and also at some of their operating characteristics. I do not propose to give exhaustive descriptions of each chip, as this would serve no practical purpose. Only such information as might be useful in adding hardware to the machine, and understanding the hardware function has been included.

The chips inside the CPC

The Z80A

This is the microprocessor chip which is the central processor for the whole machine. That is all I want to say about this chip here since all of the next chapter is dedicated to it.

The 8255 PPI

The 8255 was first designed by Intel Corporation. It is called a Programmable Peripheral Interface (PPI) chip. It comes in a 40 pin Dual In Line (DIL) package, and is a highly versatile chip for use in computer Input Output (I/O). The 8255 provides a possible 24 lines to interface the computer to the outside world. Thus it forms a channel via which data can be passed in or out of the computer. In fact the PPI provides up to three eight bit channels into, or out of the computer. These three channels are referred to as port A, port B and port C.

When powered on the PPI is placed in a reset state. In this state the software can place instructions in the PPI mode definition register which will define the PPI operating characteristics. There are a great many ways to configure the chip. It can be configured as three output ports, three input ports, or any mixture in between. Additionally there are special modes to allow certain bits of port C to be used as an input strobe to latch data into port A or port B, or as handshaking signals for data transfers, as well as interrupt related signals. As previously stated the exact function of the chip is defined by the program which initialises it after power on. Refer to the data sheet of the 8255 for more details of configurations available. Let us get down to some specifics about the PPI.

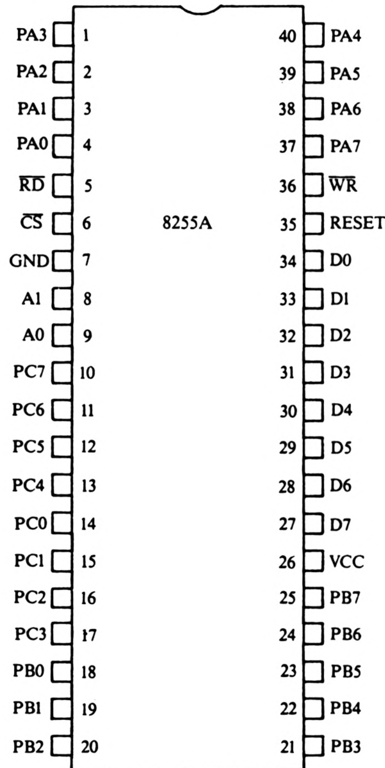


Fig 1.1: 8255A Chip Pinout Details

The 8255 pinout is shown in Figure 1.1. It requires only a +5 volt supply and a ground (0 volts) connection. The three ports A,B, and C, each have 8 bits numbered on the pinout as PA0:PA1:PA2:PA3:.. for port A, PB0:PB1:PB2: PB3.. for port B, and so on. The 8255 occupies four addresses in the I/O space of the CPC 464. The first address is for port A, the second is for port B, and the

third is for port C. Any of these three addresses can be read from or written to. The fourth address is write only, you cannot read from it, this is the address to write the mode definition into. The truth table for addressing the chip is shown in Figure 1.2

A1	A0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Function
0	0	0	1	0	Read Port A
0	1	0	1	0	Read Port B
1	0	0	1	0	Read Port C
0	0	1	0	0	Write Port A
0	1	1	0	0	Write Port B
1	0	1	0	0	Write Port C
1	1	1	0	0	Write control byte
X	X	X	X	1	Chip deselected

X = doesn't matter

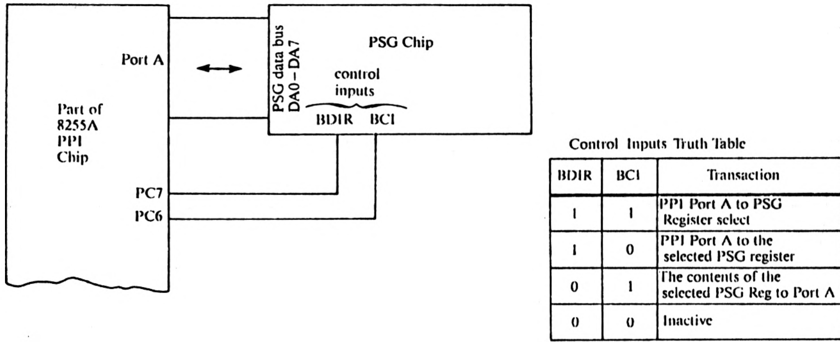
Note: Reading from the control register will always return rubbish, as no read facility is available on it. Ports A, B and C can all be read – even when programmed to be outputs.

Fig 1.2: 8255A addressing truth table

Regard the 8255 as a big chunk of I/O logic which can be programmed to meet the specific requirements of almost any application requiring input/output.

Now let us have a look at how the 8255 is used inside the CPC 464 computer. The registers of the 8255 reside at I/O port addresses Hex F4xx, F5xx, F6xx, and F7xx. The xx section of these numbers means that you can use any last two digits for the address, so long as the first two are as shown. This is due to the fact that the I/O addresses in the CPC 464 are not completely decoded– of which more in the final section of this chapter. The addresses above refer to port A, port B, port C, and the control register of the 8255 respectively.

So what are the three ports used for? Port A is used for accessing the Programmable Sound Generator chip (PSG). Because all transfer of the



BDIR	BCI	Transaction
1	1	PPI Port A to PSG Register select
1	0	PPI Port A to the selected PSG register
0	1	The contents of the selected PSG Reg to Port A
0	0	Inactive

Fig 1.3: Block diagram showing how the PSG chip is connected to the 8255A chip inside the CPC

```
1000    CLS: PRINT "This program POKEs a machine code program into memory":
             PRINT "starting at memory address 30000. This small machine" :
             PRINT "program will appear to hang up the machine until the "
1010         PRINT "CAPS LOCK button on the keyboard is pressed. See also":
             PRINT "the assembly listing for this program, and also the text":
             PRINT "for which this is an example"
1020    MEMORY 25000: RESTORE: FOR I=30000 TO 30099: READ R: POKE I,R: NEXT:
        PRINT STRING$(3,10): INPUT "Press blue ENTER to proceed"; R$:
        CLS: PRINT "Now the machine code routine will be CALLed - press "
1030    PRINT "CAPS LOCK when you have satisfied yourself that pressing any":
        PRINT "other key has no effect": CALL 30000
30000   DATA 197 , 245 , 243 , 62 , 130 , 205 , 133 , 117 , 62 , 0
30010   DATA 205 , 127 , 117 , 62 , 7 , 205 , 139 , 117 , 62 , 192
30020   DATA 205 , 127 , 117 , 62 , 0 , 205 , 127 , 117 , 205 , 139
30030   DATA 117 , 62 , 128 , 205 , 127 , 117 , 62 , 0 , 205 , 127
30040   DATA 117 , 62 , 14 , 205 , 139 , 117 , 62 , 192 , 205 , 127
30050   DATA 117 , 62 , 0 , 205 , 127 , 117 , 62 , 146 , 205 , 133
30060   DATA 117 , 62 , 72 , 205 , 127 , 117 , 1 , 0 , 244 , 237
30070   DATA 120 , 203 , 119 , 32 , 250 , 241 , 193 , 251 , 201 , 1
30080   DATA 0 , 246 , 237 , 121 , 201 , 1 , 0 , 247 , 237 , 121
30090   DATA 201 , 1 , 0 , 244 , 237 , 121 , 201 , 0 , 0 , 0
```

Fig 1.4: BASIC listing for the CAPS LOCK key interrogation program

information to and from the PSG takes place via port A, it must be constantly changed between an input and an output port. The situation is further complicated by the fact that the PSG itself has an I/O port which is used to help scan the QWERTY keyboard. Thus the software must not only suit the flow of data via port A to the current transaction with the PSG, but it has to use two of the bits of port C, to feed into the PSG control inputs, a code which will define what that transaction is. This is all highly confusing in abstract, but perhaps figure 1.3 will illustate the idea more clearly. What? still confused? Well then, figures 1.4 and 1.5 are listings of two functionally similar programs which will load all the appropriate registers of the PPI and PSG with the values required to see if the "Caps lock" key on the keyboard is pressed. Yes, I know you can use the INKEY function of BASIC to do this, but we are going to do it the hard way– just this once! Figure 1.4 is the BASIC version of the program, which has built into its DATA statements the machine code

program which actually does the job. Figure 1.5 is the assembly listing of that machine code program. DON'T PANIC if you cannot understand these programs at this stage, we will be covering assembly language programming in chapter 5, and included will be a description of how to incorporate machine code into your BASIC programs. The program ends when the CAPS LOCK key is pressed. As you can see from the listings low level keyboard driving on the CPC is no sinecure for the programmer, but using the PSG I/O port has saved the addition of another chip to scan the keyboard. You could use the program to scan any key on the keyboard, see the key layout in the description of port C usage.

```
100 ; ** Program to show how to access the keyboard via the PPI
200 ; ** the PSG chips.
300 ;
400          ORG  30000
500          ENT  $
600          PUSH BC ; Save regs
700          PUSH AF
800          DI
900          LD   A,#82
1000         CALL CONOUT ; Set PPI mode
1100         LD   A,0        ; Port C
1200         CALL COUT    ; gets sent zero to make PSG inactive.
1300         LD   A,7     ; Then port A = 7
1400         CALL AOUT
1500         LD   A,#C0 ; now send port C Hex C0 to latch data on
1600         CALL COUT  ; as a PSG register address.
1700         LD   A,#0  ; and then return PSG to inactive
1800         CALL COUT
1900         CALL AOUT  ; Port A=0 as well
2000         LD   A,#80 ; write zero into PSG reg 7
2100         CALL COUT  ; put a write command into port C
2200         LD   A,0
2300         CALL COUT ; And port C inactive again
2400         LD   A,14  ; Now we put the PSG I/O register number
2500         CALL AOUT  ; into PSG via port A
2600         LD   A,#C0
2700         CALL COUT ; and output a latch address command to Port C
2800         LD   A,0 ; and then return port C to inactive.
2900         CALL COUT
3000         LD   A,#92 ; Now we must change PPI mode
3100 ; this involves changing port A to a set of inputs.
3200         CALL CONOUT
3300         LD   A,#48 ; Tell port C to set PSG to output its REG #14
3400 ; also select the key rank on which CAPS LOCK key lives
3500         CALL COUT
3600         LD   BC,#F400 ; BC=address of port A
3700 NOPRESS:     IN         A,(C)
3800         BIT  6,A ;
3900         JR   NZ,NOPRESS ; Jump if no press
4000         POP  AF
4100         POP  BC
4200         EI   ; Interrupts on again
4300         RET
4400 COUT:   LD   BC,#F600 ; Port C address into BC
4500         OUT  (C),A
4600         RET
4700 CONOUT: LD   BC,#F700 ; BC=PPI control reg address
4800         OUT  (C),A    ; Do the output
4900         RET  ; And go back
5000 AOUT:   LD   BC,#F400 ; BC=address for port A
5100         OUT  (C),A
5200         RET
```

Fig 1.5: Assembly language version of CAPS LOCK interrogation program

Port B of the PPI is used as an 8 bit input port. Bit 0 is used to sense the state of the frame flyback sync pulse from the VDU controller chip. Bits 1–4 are used to sense whether links one to four are installed on the CPC PCB. The purpose of most of these links is not defined in published Amstrad documentation, however, link 4 (LK4) which is attached to bit 4 of port B must not be installed if you are plugged into a 50 Hertz mains supply, and must be installed if you are running your CPC 464 on a 60 Hertz supply (as in the USA). If you type the BASIC command:

```
PRINT BIN$(INP(&F500))
```

BASIC will print the state of each bit at port B. On my machine I get 111110. When a bit whose state is controlled by a link reads as a one, then the link is out. Therefore – as bits 1–4 are all ones in the case of my machine– all the links in my machine are out. Bit 5 of port B is a sense input which is available at the expansion bus. It is, therefore, not unreasonably called EXP. This can be used to allow software to tell whether something is plugged into the expansion bus or not. Bit 6 is the BUSY signal from the Centronics printer port. If this signal is high (logic one) then the printer is signalling the computer not to send any more characters for printing. (See appendix 7 for more details about the printer port). Bit 7 of port B is used to receive data from the cassette data recovery electronics. The signal actually presented to bit 7 port B is the digital version of the data coming from the cassette tape. The software has to decode this signal, using various complex timing routines, into valid data. All the inputs to port B except bits 0 and 7 have 2.2K ohms pull up resistors fitted to them.

Port C is used as all outputs. Bits 0–3 feed into a 74LS145 chip. The code presented to bits 0–3 of port C is decoded by the 74LS145 into ten outputs. That is to say whichever of the ten outputs of the 74LS145 is active depends on the code at its input.

Together with the I/O register of the PSG chip, the 74LS145 outputs form an X,Y matrix across which the keys of the keyboard are connected. Figure 1.6 shows where in this matrix each key on the keyboard is connected. When the indicated keys are pressed one of the outputs from the 74LS145 is momentarily connected to one of the inputs of the PSG I/O port. Given that there are ten outputs from the 74LS145, and eight inputs to the PSG, this gives a possible 80 keys which could have been connected in the matrix. There are nine intersections (or crosspoints as they are more often called) which are unused on the QWERTY keyboard. These are used when connecting joysticks to the paddle port at the rear of the CPC 464.

Bit 4 is used to control the cassette motor relay. If this bit is set high then the cassette motor is switched on, via a small relay located with the datacorder. Prove this to yourself by typing in the program shown in Figure 1.7. This switches the motor relay on and off every two seconds. DO NOT try to make the relay switch any more frequently than this, as you will damage the transistor which drives the relay under bit 4 control. Bit 5 of port C is used to

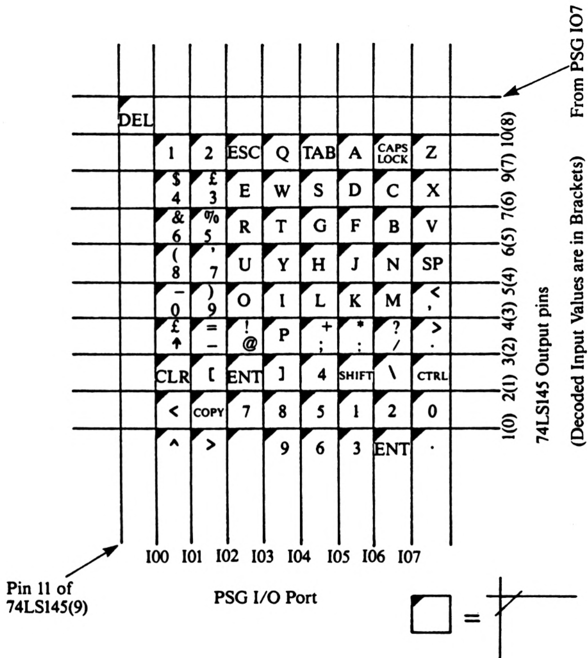


Fig 1.6: CPC keyboard matrix layout

```
1000    '** Program to demonstrate how port C of the PPI controls
         ** the motor relay for the cassette datacorder.

1100    ON BREAK GOSUB 10000
1200    EVERY 100,0 GOSUB 5000: MODE 2: LOCATE 1,10
1300    GOTO 1300
5000    OUT &F600,((INP(&F600) XOR 16)):
        PRINT "Relay state changed";CHR$(24);CHR$(13);: RETURN
10000   OUT &F600,((INP(&F600) AND &E0)): PEN #0,1: PAPER #0,0: CLS:
        PRINT CHR$(13);"Relay switched off": END '** Make sure motor is off
        ** and pen and paper are normalised when ESC is pressed.
```

Fig 1.7: Relay control demonstration

send data for writing onto cassette tape to the datacorder electronics. Bits 6 and 7 are used to control transactions between the PSG and port A – as previously discussed in the description of port A usage.

The ROM

The BASIC interpreter and the operating system software have been permanently programmed into a 32K byte ROM (Read Only Memory). Each of these two vital pieces of software occupies exactly half of the ROM (that is 16K bytes each) with the BASIC interpreter in the top half. Although they

reside in the same physical chip, it is best to picture the ROM as being split into two halves. Throughout the Amstrad documentation, and in this book, there are references to the lower ROM and the upper ROM. The address decoding built into the gate array chip actually arranges things so that the lowest half of the ROM is seen as starting at address 0000, whilst the upper half is seen as starting at address Hex C000. The memory map is further complicated by the fact that the ROM can be switched out, and RAM be left in its place. At the end of this chapter there is a special section explaining how this duality works. The ROM chip itself has 15 address inputs, 8 data I/O lines, an output enable pin, a chip select pin, and requires only the +5 volt supply and 0 volts. The pinout is the same as for the 27128 EPROM which you will find in the description of project Eight, on page 131.

The RAM chips

The RAM consists of 8 chips. These are all 64K by 1 bit dynamic RAM chips, type HM4864. Whilst they offer far higher capacity per chip than a static RAM (for example the 6116 RAM which is a 2K by 8 bit RAM which you may have seen used in magazine projects a lot), dynamic RAM chips need to be continuously reminded of what they are remembering! This is because the memory elements are actually tiny capacitors, which if they are not topped up will lose their charge. This top up function is called refreshing them. (No Lager jokes please!). The trick is to perform the RAM refresh when the memory is not being accessed. In fact the Z80A has the facility to automatically perform refresh, but in the CPC the gate array has been chosen to do it. This is because the memory is also accessed by the VDU controller, and the gate array has to ensure that the memory is not required by either processor or VDU controller before refresh can occur. The memory refresh process consists of pulsing two input lines to all the RAM chips. These are called CAS (Column Address strobe) and RAS (Row address strobe). These are pulsed whilst the row and column addresses are passed into the RAM sequentially. The 4864 chips used have a cycle time of 200 nanoseconds.

Potentially the RAM occupies all the memory map, but as we shall see in the last section of this chapter the somewhat complex address decoding in the CPC can switch out large sections of RAM in order to let in the ROM sections.

Programmable Sound Generator

The Programmable Sound Generator (PSG) is used in a rather odd way in the CPC. The AY−3−8912 was originally designed to be directly connected to a microprocessor bus. As we saw when looking at the way in which the 8255 PPI chip is used, the PSG is not connected to the Z80A bus at all, it receives all its data and commands via the PPI. This does not stress the chip, but certainly stresses those programmers who want to access the PSG directly. I cannot imagine that there will be many people who, having tried out the SOUND,ENV, and ENT commands in the Locomotive software BASIC, will want to program the PSG in machine code. Using the PSG from BASIC is complex enough! However, a fairly detailed examination of the PSG and its internal registers might perhaps reveal how BASIC manages to keep the beast under control.

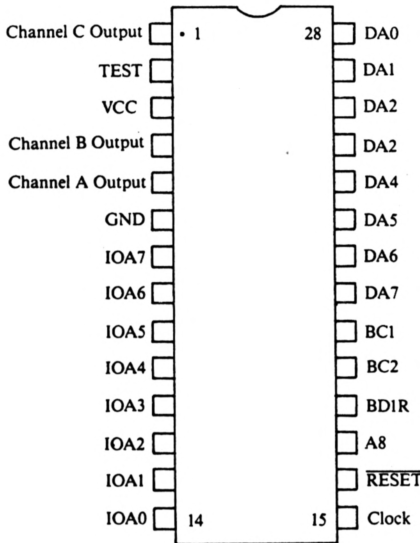


Fig 1.8: PSG Chip Pinout

The PSG chip pinout is shown in Figure 1.8. Its power supply requirements are for +5 volts and 0 volts. The DA0–DA7 lines are bi–directional tri–state lines which are intended for connection to a microprocessor data bus. The lines marked IOA0–IOA7 are a general purpose I/O port – pretty similar to one of the PPI ports. This I/O port is the one we saw when discussing the way the PPI and PSG interact to scan the QWERTY keyboard. Pins 1,4, and 5 are the three pins from which comes the sound which the PSG has been programmed to produce. As you will know from having used the SOUND command of BASIC, you can define what comes out of each channel independently of the others. Some rather clever software is built into the Locomotive BASIC to synchronise the output of these three channels. The RESET pin on the PSG will, when taken low, place the contents of all internal registers of the PSG in the inactive condition. The A8 input is unused in the CPC. The TEST input is not for general use. The CLOCK input is the master frequency input which will determine the centre frequency of the sounds the PSG is capable of making. In the CPC this is fed a one megahertz signal from the gate array. VSS is the 0 volts or ground pin. Finally the BC1,BC2, and BDIR inputs form the command inputs to the PSG. BC2 is tied high inside the CPC. If you refer back to figure 1.3 of this chapter you will see how to encode the function on the BC1 and BDIR pins.

Now the internal registers of the PSG. The BASIC method of driving the PSG works by the user program supplying a list of parameters to the `SOUND` command. As in:

```
10 FOR I=35 TO 4 STEP -1: SOUND 1,I,5 SOUND 2,I*4,5: SOUND 4,I*3,5:
NEXT: GOTO 10
```

Broadly speaking, a similar method is used when machine code programming the PSG. The important difference is that the parameters which are actually used must be placed into each register of the PSG at the appropriate time. For your information here is the list of the PSG internal registers and their purposes.

The three sound channels within the PSG are called channels A,B, and C. The frequency of the sound coming from each of these channels is defined by six registers, two per channel. These lay out as follows:

Register 0 = An 8 bit register which represents the lower 8 bits of the frequency to be produced by channel A
Register 1 = A 4 bit register which represents the upper 4 bits of the frequency to be produced by channel A
Register 2 = As for Reg. 0 except that it relates to channel B
Register 3 = As for Reg. 1 except that it relates to channel B
Register 4 = As for Reg. 0 except that it relates to channel C
Register 5 = As for Reg. 1 except that it relates to channel C

The values placed in registers 0–5 are used purely to define the division factor of the master CLOCK. As such, the frequencies obtained for any given values will depend on the frequency of this master clock. Register 6 defines how long the noise feature of the chip is turned on for. This noise is useful in explosions for games etc. R6 is a 5 bit register. R7 is the I/O port enable and mixer control register. Bit 0 ,bit 1, and bit 2, when set low enable the tone generators for channels A,B and C respectively. Bits 3,4, and 5 enable the noise generator to be output on channels A,B and C respectively. Bit 6 would be set high to make the I/O port an output– not advisable in the CPC configuration – and set low to make the I/O port an input. Bit 7 of register 7 is unused. Registers 8,9, and 10 are the amplitude controls for channels A,B, and C respectively. Bits 0 to 3 of these registers control the volume of each channel, and bit 4 controls the amplitude mode. Registers 11,12, and 13 are used to control the envelope of the generated sounds. Registers 11 and 12 are used as a 16 bit value to define how long the function selected in register 13 is to last. The functions which can be selected in register 13 are:

Hold (Bit 0)
Alternate (Bit 1)
Attack (Bit 2)
Continue (Bit 3)

The remaining bits of R13 are unused. Finally as previously mentioned register 14 is the I/O register – used exclusively as an input register in the CPC, for the purpose of scanning one side of the QWERTY keyboard.

The PSG is a highly complex chip. To really master it you need to spend a great deal of time experimenting with the envelope settings and the extras provided for its control in Locomotive BASIC.

The Gate Array

This is the custom designed chip which was mentioned at the start of this chapter. Its development, specially for Amstrad, saved the inclusion of a great many other chips, which would otherwise be required to build an equivalent configuration of hardware. Details about this chip are understandably hard to get, it probably cost a lot of money to have developed. From its position in the circuit however we can tell most of the things that it does. As already mentioned the gate array performs the memory refresh. Also we have already seen that the gate array must contain a programmable address decoder, so that ROM and RAM can be switched in and out of the address space. (See the last section of this chapter for more details on the memory map.) Very importantly the gate array also produces the RGB (Red Green Blue) LUM and SYNC signals which are fed into the monitor section of the CPC 464 to provide the information to the CRT unit to make it produce a text or graphic display.

Another function of the gate array is interrupt generation. The whole time the CPC is switched on the gate array is generating interrupts to the processor, the interrupt routines are for things like updating the "Clock" which counts the time since the CPC was switched on, as used in the BASIC TIME variable, and also for things like keyboard scanning which is going on the whole time that the machine is switched on – unless you have explicitly disabled interrupts in an assembler program.

The gate array also processes the master clock for the whole machine – turning the 16 Mhz crystal pulses into a 4Mhz signal to be the CPU clock for the Z80A. (don't fall into the trap of regarding the Z80A as running at 4Mhz however. Internal logic in the gate array stretches the duration of memory and I/O request signals which slows the Z80A down a little to around the equivalent of a 3.5Mhz clocked device). The gate array produces control signals to open and close the buffers between the RAM and the Z80A data bus. It also controls whether the VDU controller or the CPU addresses the RAM.

The VDU Controller Chip

The CPC screen hardware consists of a monochrome or colour monitor, with integral power supply. The only connections from the keyboard unit to the monitor are the +5 volt power supply, which is provided to the keyboard unit from the monitor unit, and the monitor signals sent from the VDU controller chip inside the keyboard unit, to the monitor.

In conjunction with the video gate array chip (see above) the VDU controller produces all the required signals to create the display you see when you look at your monitor. As you will know this can be text, animated graphics or whatever else you program the machine to do.

The VDU controller chip is an MC6845. As you will see from its data sheet, reproduced in appendix 3 the 6845 was originally produced to work with the Motorola 6800 family of microprocessors, but it is readily adaptible to use in other systems. The 6845 can share an area of main memory with a microprocessor, and from the memory contents will create a display on the monitor. In the CPC application the video gate array controls the colours which are actually displayed, whilst the 6845 controls the shape and position of what is actually displayed. Like all TV pictures the display on the CPC machines is made up of dots. Each dot is called a pixel, and is the smallest addressable unit of the display. That is to say that the smallest effect you can have on the display is to alter one pixel. Try this out by typing the following, which is a string of direct BASIC commands:

```
MODE 2: FOR I=1 TO 500: POKE (&C000+(INT(RND*&4000)),128: NEXT
```

This will make a star matt, as used in so many spacewar type games, appear on the screen. Each pin prick of light on the screen is one pixel. What we are actually doing here is setting bit eight of a random location in the screen memory, which begins at Hex address C000, to make a pixel "bright up". Try the command sequence twice more, but substitute modes zero and one for mode two. Did you notice that the pixels are not any bigger in the vertical direction, yet they are wider in mode zero than they are in modes one and two? This is due to the way the 6845 organises the display.

When you are using the screen to display text there is a character font inside the ROM containing a list of the pixel patterns which are required to make up each displayable character, think of this as a list of dots which must bright up to make a particular letter, symbol or number appear on the screen. When a character is to be displayed on the screen, the pixel pattern for it is located in the list and placed into the screen memory at the appropriate address. As well as having a fixed character font in the lower ROM the CPC firmware also allows you to define customised characters in a RAM table. Thus you can redefine the character which is displayed for any given ASCII code. This facility is accessible with the SYMBOL keyword in Locomotive BASIC.

The 6845 is continuously reading the screen memory contents and outputting them in a serial stream to the gate array, which modifies them, adding colour information from its internal Pallette memory. The gate array then outputs them to the monitor for display. The whole thing is synchronised so that a dot will appear at the same exact point on the screen during every cycle, this giving the illusion of a static display. When you use flashing colours the appropriate codes for the inks which you have defined are alternately placed into the pallette memory at the required intervals to make the locations flash. This happens as part of the interrupt routines mentioned previously.

How about the cursor? When you are editing, word processing, or entering programs, you need to have a visual indication of where the next character you type will appear on the screen. This of course is what a cursor is for. The cursor is actually generated on the 6845. It can be big, or it can be small, it can

blink, be static, or be turned off altogether. The 6845 sports several registers to control the cursor and its attributes, refer to the data sheet for details.

There is another feature of the 6845 which is of especial use to people who want to go in for interactive graphics, it is the light pen input. This is a pin on the 6845 which is extended to the expansion bus (see appendix 5). To use this you get a light sensor, installed inside an old ballpoint pen with the nib and ink tube removed, and connect it via a simple interface circuit to the light pen input. As each pass of dot drawing proceeds to build up the display on the face of the monitor tube, the light from the cathode ray tube momentarily shines onto the light sensor when the pen is held with its tip onto the screen. The 6845 has some counters inside it which show exactly which memory location it is currently transferring to the screen. Asscotiated with the light pen input are two eight bit registers. These are loaded with the contents of the screen memory location counters when a high going signal occurs at the light pen input. This means that once you have built a light pen, which is very easy and cheap, you then only need software which continually reads the state of the light pen address registers, and writes a one into the screen memory which they indicate. You can then move your light pen around the screen and draw with it.

The 6845 is a very capable and versatile chip. When used in the CPC machines in conjunction with the gate array the result is a very flexible text VDU, and can be some stunning graphics. To use an old cliche, with the colour version of the CPC the usage of the screen is only limited by your imagination.

The printer port
In pure hardware terms the printer port consists of an eight bit latch chip. This is a 74LS273, and it lives at I/O address Hex EF00. There are some special considerations to apply when using the printer port, and printers connected to it. These are discussed in appendix 7.

The memory map (and how it can be altered)

As already mentioned several times the memory map of the CPC is somewhat difficult to be dogmatic about due to the programmable address decoder inside the gate array. If you probe about from BASIC you can PEEK and POKE into any location of memory, and it will always look like RAM! Logic tells us that this cannot be so, there has to be some ROM located somewhere to contain both the BASIC interpreter and the machine firmware. Crafty old BASIC arranges for the ROMs to be switched out of the memory map whenever you PEEK or POKE a memory location. The truth of the matter is that the gate array has a register inside it called the ROM select register. We shall now look at how this functions.

Warning: If you know nothing about the Z80A microprocessor, save your sanity by returning to this section after reading chapter two

In chapter two we shall be looking at the Z80A microprocessor in a little detail, but briefly it provides two discrete addressing spaces these are called the memory address space and the I/O address space. In the CPC address allocation scheme the I/O address space is used for I/O to peripheral devices, and also for access to the gate array. In the I/O address space an access to a specific address will always read or write from or to the same device or register. The memory address space on the other hand is sectioned into three distinct areas. These are:

1) The lower ROM area addresses 0000 to 03FF

2) The middle RAM area addresses 4000 to BFFF

3) The upper ROM area addresses C000 to FFFF

The ROM select register, which is loaded by writing into I/O address 7F80, can make either the upper or lower ROM be replaced with RAM. This is due to the programmable address decoder inside the gate array. When the ROM select register shows that, for instance, the Upper ROM is disabled, the decoder will enable the RAM when addresses in the range C000 to FFFF are decoded. If the upper ROM is enabled then the decoder enables the ROM to be read from, and the RAM remains deselected unless the middle RAM area is addressed. The lower ROM can be switched in or out in the same way. The scheme is further complicated by the fact that no matter what ROMs are enabled, writing to a location always writes into the RAM. This makes a kind of sense, since allowing a write into ROM would be futile! When a ROM is enabled, the RAM to which you can still write is said to be "Underneath" the ROM.

Remember that although we talk about the upper and the lower ROMs they are actually two halves of one chip. Which half is actually accessed is again controlled by the gate array.

The Datacorder (CPC 464 only)

A standard cassette mechanism is built into the CPC 464 and some special electronics are attached to it. Place a cassette containing music into the datacorder, press down the PLAY key and type the CAT command. Turn up the volume control on the side of the unit. Not exactly Hi–Fi is it? Try playing a tape containing speech, it sounds dreadful, as if it were coming through an exceptionally bad CB radio. The reason for the bad reproduction is that the electronics which processes the signal from the datacorder head is set up to optimise only the frequencies at which data is recorded, not a broad range of frequencies as you need for normal music playback. (Take the music or speech cassette out of the datacorder now – we don't want to overwrite it!)

When writing data to tape the CPC firmware has limited control over tape motion in the datacorder by means of a stop/go relay, whose contacts make or

break the supply of +5 Volts to the cassette motor. The electronics of the datacorder has an input, and an output line – as with an audio recorder. Data to be written to tape is fed serially into the input from one line of PPI port C. Data read from tape is presented as a squared up waveform at the output. The datacorder electronics contains no logic circuitry at all, all encoding and decoding, serialising and deserialising the data is done by software.

Disk units

Extensive details of the disk unit are not provided here, as another SIGMA book is available* which concentrates on the disk version of the CPC 464 as well as the CPC 664 and 6128.

The disk units used on the CPC machines are three inch floppy units. The disks are double sided, and provide 169K of storage per side of the diskette. On the CPC 464 the controller is supplied in an external plug in unit which contains not only the floppy disk controller chip, but also the Background ROM which contains software to provide some extra commands to control disk i/o. As if this were not enough, the disk package also comes with a copy of the C/PM operating system, and the Dr Logo programming language. The CPC 664 contains the same things, but they are all built into an expanded keyboard unit, which has different coloured keys.

Conclusion

The hardware of the CPC is very good, and very solid. All the chips used except the gate array, have been available for some time, which to some degree makes them proven products. The gate array, in common with every other gate array I have ever encountered, runs very hot. To help dissipate some of this heat it is fitted with a small metal heatsink. You should bear in mind this heat factor when using the machine, especially on hot days. Try not to position the machine where direct sunlight can shine in through the louvres at the rear of the machine. Do not obstruct the cooling louvres on either the keyboard, or on top of the monitor unit. Overheating will seldom do lasting damage (unless you really overdo it) but whilst the machine is over temperature it may repeatedly crash (seize up in laymans lauguage!) or do the unexpected. Before you take your CPC back to your Amstrad dealer one summer's day for miscellaneous malfunctions, ensure that you were not causing it to overheat by obstructing its cooling vents.

In the next chapter we shall look at the central chip in the machine, the Z80A microprocessor.

**"The Amstrad Disk Companion", by Simon Williams.*

CHAPTER TWO

Z80 Microprocessor Primer

At the heart of your Amstrad CPC is the Z80A microprocessor. The Z80 is made, and was designed, by Zilog inc. It has now been available for about eight years. The design of the Z80 is really an expansion of an earlier microprocessor, the 8080, which is a product of Intel corporation, who subsequently designed and marketed the very successful 8086 microprocessor. By using the basic 8080 architecture, but adding more internal registers, and greatly expanding the instruction set, Zilog managed to produce a powerful, and ultimately cheap microprocessor. (At the time of writing you can buy a Z80 for under £3). The version of the Z80 used in the CPC machines is the Z80A. (The only difference between a Z80 and a Z80A, is the speed at which the chip can run, the Z80A can run from a 4MHz master clock frequency, whilst the Z80 can only run from a 2.5 MHz clock. This means that the Z80A runs almost twice as fast.)

This chapter is not intended to give you an exhaustive knowledge of the Z80 microprocessor. There are already many books available which concentrate exclusively on the Z80 microprocessor, its instruction set, and how to apply them. Some of these titles are listed in the bibliography, appendix 2 The intention of the author in including this "Primer", is to give such information about the Z80 as will assist readers in understanding the software and hardware presented in the next chapter. In view of this the presentation of the complete Z80 instruction set could be seen as superfluous, but it has been included as a ready reference to assist readers wishing to study or modify assembly language programs in later chapters.

Z80A pinout and signal description.

The Z80 microprocessor comes in a 40 pin DIL (Dual In Line) package. Like most other microprocessor chips the Z80 produces all the signals required to address, and read from, or write to, memory chips external to it. Additionally, the Z80 produces signals which are for a specific type of memory called Dynamic RAM (Random Access Memory). Dynamic RAM offers far greater memory capacity per chip, but has to have a continual stream of refresh commands, otherwise it loses its contents. The Z80 has the capability to supply these refresh command signals, thus obviating the need for additional devices to perform the function but, the CPC dynamic RAM chips in the CPC are not refreshed by the Z80. Instead this is performed by the gate array chip, which was introduced in chapter one. The Z80A pinout is shown in Figure 2.1. We will now briefly look at the purpose of each of the signals, or group of signals, which are shown on the pinout diagram.

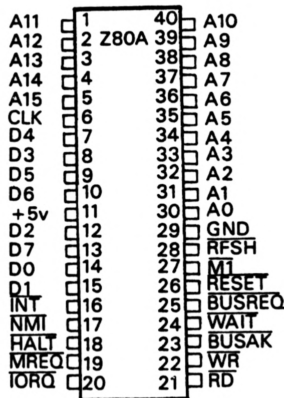


Fig. 2.1: Z80A Pin Connections

Pins 1 to 5 and 30–40: A0–A15. These lines contain the address of the external memory, or I/O (Input/Output) location which the microprocessor is currently accessing. With 16 address lines, up to 65536 (Decimal) locations can be addressed.

Pin 6: Clock. This is the pin to which is applied the clock signal for the microprocessor. The frequency of this clock determines how quickly instructions are executed. For the Z80A the maximum frequency is 4MHz.

Pins 7–10 and 12–15: D0–D7. These are the data lines over which data is transferred from, or to, the Z80. These lines obviously have to be bi-directional, to allow data to flow into, and out of, the microprocessor.

Pin 11: VCC. This is the positive voltage supply pin. The Z80A needs +5 volts at a maximum of 200 milliamps.

Pin 16: INT bar. This is the primary interrupt signal input, it is negative edge triggered. When a negative going edge has occurred at this input, and software has previously allowed interrupts, by executing an EI (Enable Interrupts) instruction, an interrupt sequence is entered upon completion of the instruction currently being executed. None of the projects described in this book use interrupts. The CPC firmware makes extensive use of interrupts for timing of firmware events. The Z80 provides comprehensive and complex interrupt handling facilities. We shall not deal with them in detail as they are not applicable to the projects in hand; readers who need to know more about interrupts handling are referred to the books on the Z80 in the bibliography.

Pin 17: NMI bar. This is the Non–Maskable Interrupt signal. As its name implies, it is an interrupt pin whose function cannot be software disabled. (that is, masked off.) When a negative going edge occurs at this input, an interrupt sequence occurs after completion of the current instruction.

Pin 18: HALT bar. This output signal from the Z80 goes low whenever a HALT instruction is executed.

Pin 19: MREQ bar. This output signal is low whenever the Z80 wants to read from, or write to, a memory address, as opposed to an I/O address. In the next subsection we shall look at the sequence of signals for a memory read and write.

Pin 20: IORQ bar. This output is taken low to indicate that the Z80 wants to read or write to an I/O address. In the next subsection we shall look at an I/O read and an I/O write cycle.

Pin 21: RD bar. This is the read strobe signal output. This line is pulsed low to strobe data presented by memory or I/O devices into the Z80.

Pin 22: WR bar. This is the write strobe signal output. This line is pulsed low to strobe data from the Z80 into memory or I/O devices.

Pin 23: BUSAK bar. This output signal goes low to indicate to external devices that the Z80 address and data lines are in a high impedence state. When this occurs, other devices can use the address and data bus lines. An example of another device which might share the bus in this way is a disk controller with DMA capability .

Pin 24: WAIT bar. This input signal is used to extend read or write cycles indefinitely, to facilitate using slow devices or memory chips with the Z80.

Pin 25: BUSRQ bar. This input signal is taken low by devices requiring bus access, which must then wait until the Z80 completes it current instruction, whereupon it will reply with BUSAK.

Pin 26: RESET bar. An input signal which, when taken low, resets the microprocessor completely.

Pin 27: M1 bar. Execution of all microprocessor instructions involves at least one so called fetch cycle. This is when the microprocessor is fetching the next instruction to be executed. It is followed by the execute cycle. This output signal goes low as the Z80 is entering a fetch cycle. When the Z80 is fetching two byte instructions, the M1 signal will be low until both bytes have been fetched.

Pin 28: RFSH bar. This signal when taken in conjunction with the MREQ signal is used for dynamic RAM refresh. In the CPC this pin is unconnected.

Pin 29: GND. This is the zero volts supply pin, or ground.

Now, having looked at the signals passing in and out of the Z80, we shall see how these are used. Please be aware that in terms of the internal function of the microprocessor the following examples are simplifications.

The Z80 microprocessor internal sequencing is split into time slots, which are called "T cycles". In broad terms we need not concern ourselves with these, because we are only interested in how the Z80 relates to external hardware. Readers keen to find out more about the Z80 should choose a book from the bibliography. The cycle sequences presented here are a satisfactory basis upon which to learn how to interface external hardware to a Z80.

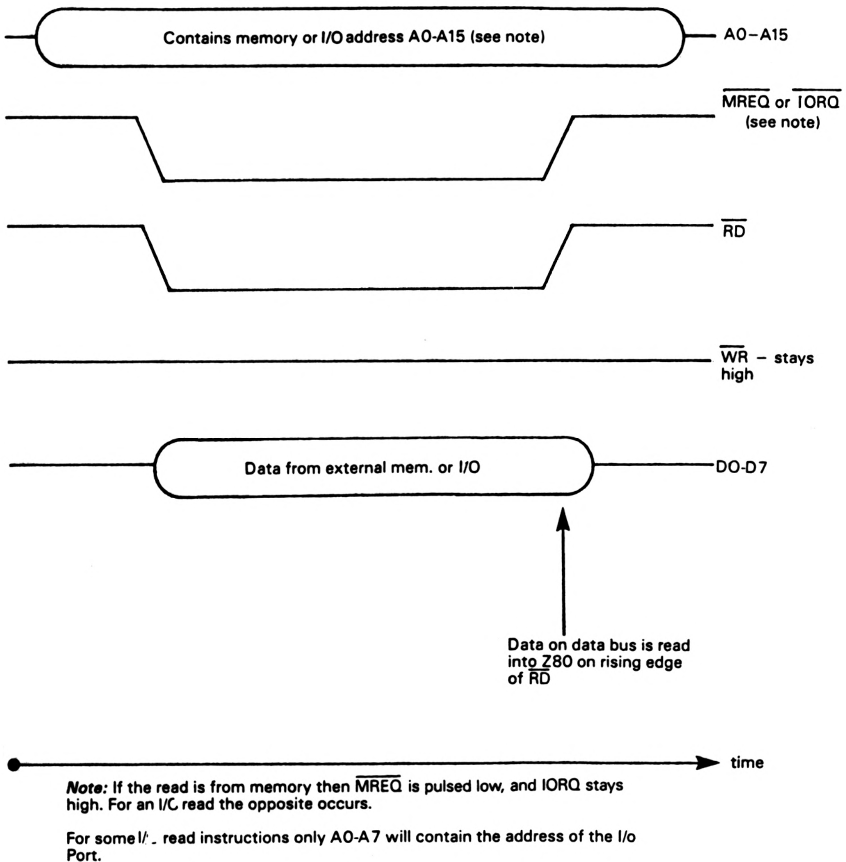


Fig. 2.2: Shows the signal sequence of a Z80 read cycle.

Signal timings

Figure 2.2 shows the timing for the Z80 read cycle. This occurs when the Z80 is reading a byte of data from memory into one of its internal registers. In this case the sequence of signals is as follows:

1) The Z80 places the address of the location which it wishes to access on the address lines A0–A15.

2) After a very short delay, to give the address lines time to stablise, the RD bar strobe is pulsed low. If the read is to a memory address then the MREQ bar signal goes low at the same time as RD bar. If the read is from the I/O address space then it is the IORQ bar signal which goes low at the same time as RD bar.

3) During the time that the RD bar signal is low the contents of the memory location or I/O register which are addressed, must be placed on the data bus lines D0–D7. Just before the RD pulse ends, the data presented on the data bus is read into the Z80.

4) RD and MREQ (or IORQ) return to inactive high.

5) The address lines are released. The read cycle is now ended.

To any readers who are complete newcomers to microprocessors, it must seem that this sequence will be complex enough to make the Z80 slow. Not so. In the Z80A configuration used in the CPC, the above sequence will occur in 750 nanoseconds, that is 0.750 microseconds, or 0.000750 milliseconds, or one 0.000000750th of one second! To give you a comparison, the average human reaction to a random event takes about 10 milliseconds, or one 0.010th of a second.

The cycle for a memory write differs from an I/O write cycle, therefore, the two will be considered separately. Figure 2.3 shows the Z80 memory write cycle. This takes place in the following steps:

1) The Z80 places the address of the memory location it wishes to write into, on the address lines A0–A15.

2) The Z80 takes the MREQ bar line low.

3) The Z80 places the data which is to be written on the data bus lines D0–D7.

4) The Z80 pulses the write strobe line, WR bar.

5) Upon termination of the WR strobe pulse,the MREQ bar line is returned to its high (inactive) state.

6) Finally, the address and data lines are released.

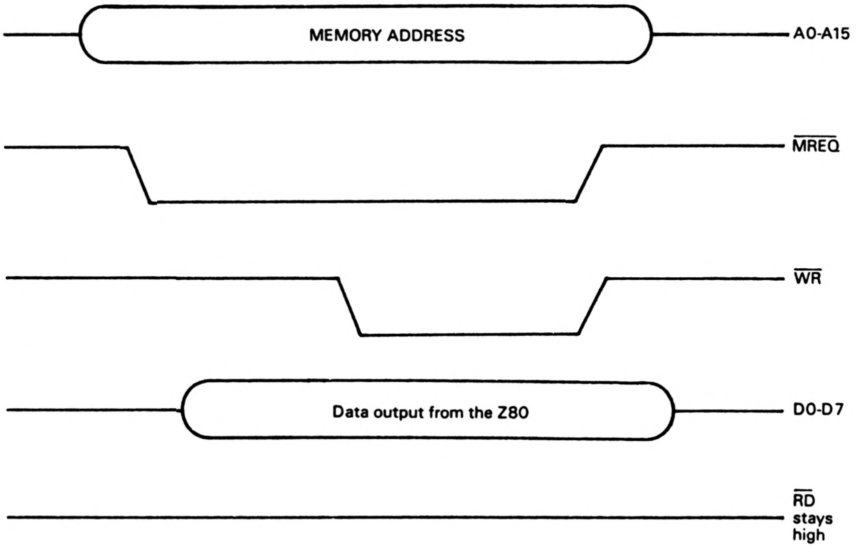


Fig. 2.3: Illustrates the sequence for a memory write.

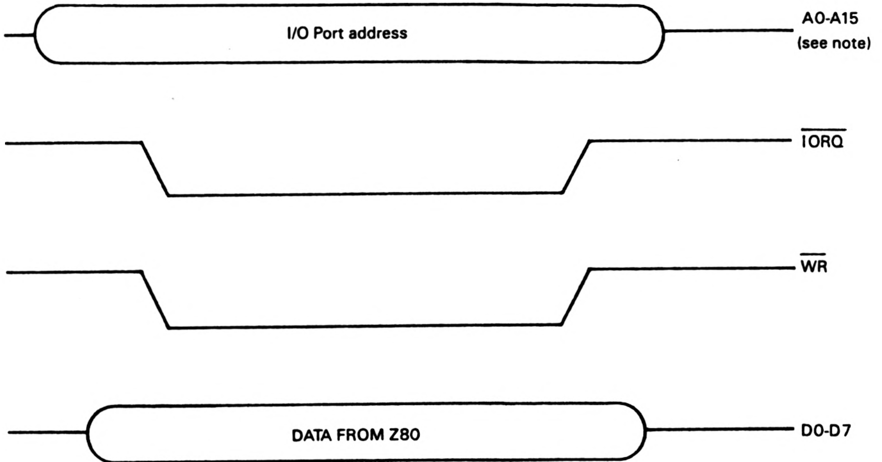


Fig. 2.4: Shows the sequence timing for a Z80 I/O write.

Now the sequence for an I/O write cycle. This occurs in the following steps; (Refer to Fig 2.4.)

1) The Z80 outputs to the address bus the address of the I/O port it wants to write into.

2) The Z80 places the data it wishes to write onto the data bus lines, D0–D7.

3) The Z80 takes the IORQ bar and WR bar signals low simultaneously.

4) The WR bar and IORQ bar lines are returned to their inactive high level.

5) The data and address buses are released.

These then are the sequences of signals for reading and writing data with the Z80. Now, we must look at the Z80 with a programmer's eye, soldering irons off please!

Z80 Programming.

To the programmer, the Z80 microprocessor looks as shown in Fig 2.5. The registers available to the programmer will now be described.

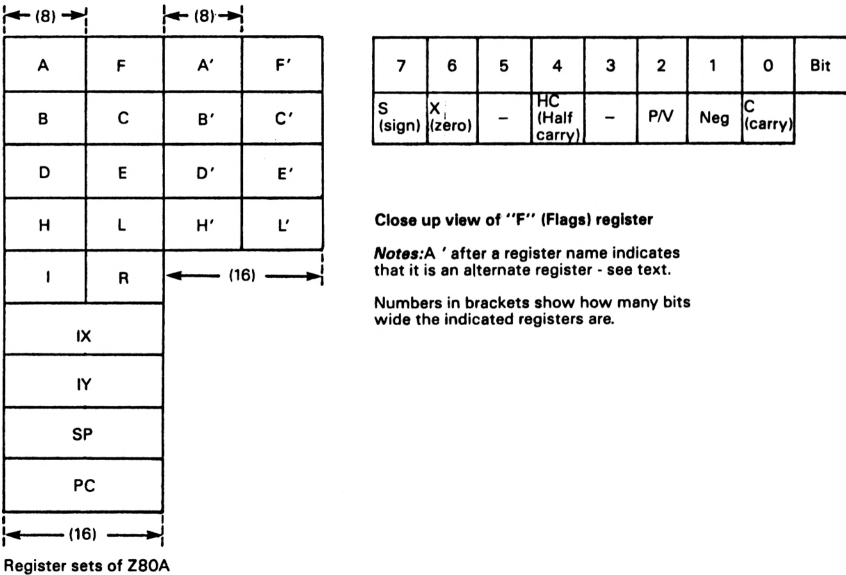


Fig. 2.5: Shows the Z80 register map.

The A register or Accumulator: This register is the one into which the results of operations are always placed. The operations in question being things like additions, subtractions, and logical operations like OR, AND, XOR and NOT.

Six of the registers are available for use singly, or can be used in pairs to hold 16 bit data, or addresses. These are the B,C,D,E,H and L *general purpose registers.* When used as 16 bit registers they are specified as the BC, DE, and HL registers with the choice of instruction used determining the mode of use.

Next comes the *flags register,* or "F" register. This is where the Z80 keeps its condition codes. Condition codes reveal everything a program could need to know about the result of a test or arithmetic operation. For example: did it generate a carry? was the result zero? and so on. This allows pre examination of conditions before executing a conditional jump or branch instruction. (if some of these terms are strange to you don't worry, the explanations will be encountered shortly in the instruction set list.)

All microprocessors have the next register, which is the *PC.* PC stands for Program Counter. This sixteen bit register always contains the address of the next instruction to be executed.

The *SP register* is the stack pointer. In order to properly understand the function and purpose of the SP register, you must understand the concept of a stack.

> A *stack* is a last in – first out type of data structure. A range of read/write memory locations is set aside by software, as an area for use as a stack. The stack pointer register is loaded with the address of the location which is at the highest point in this range. When programs need to save data from the CPU registers, during program execution, they use instructions which place the saved data on the stack. These are called PUSH and POP instructions. In a push operation the data to be saved is written into the memory location pointed to by the SP register, the CPU then automatically decrements, (that is, subtracts one from the value of), the SP. The POP operation is the opposite of the PUSH. When executing a POP instruction, the CPU first increments the SP register contents (that is, adds one to them), then loads the contents of the address pointed to by the SP,into whichever register the instruction specified. The register to be PUSHed or POPped is specified in the instruction. Fig 2.6. shows a program to PUSH the "A" register onto a stack, then POP it, alongside the physical contents of the "A" register, the stack memory and the SP at each stage of the execution of the program. The program is shown in a format called assembly language. Assembly language allows you to program the Z80 directly, refrencing the actual instruction set. In chapter five, assembly language programming will be detailed and illustrated, and in the next section of this chapter, we shall be looking at all the Z80 instructions, of which PUSH and POP are only two.

The "I" register is used to store an interrupt vector. This is used during an interrupt sequence, and its contents form the upper 8 address bits of the interrupting devices software handler. As previously stated, the add-on projects presented in this book do not use interrupts, so readers will probably not have occasion to use this register when writing programs for them.

Registers "IX" and "IY" are both index registers. They are especially useful for operations where a group or range of memory locations, are to be copied to another group or range. The contents of these registers are used as addresses, and there are instructions to add displacements, and also to increment and decrement their contents. As we shall see later in the book, this allows simpler programing of some complex operations.

Finally the "R" register. This contains the refresh address for the transparent refresh option of the Z80. In the CPC machines this is not used. The Z80 constantly increments the lowest seven bits of this register.

There are hidden depths to the Z80, in that the A,B,C,D,E,H,L, and F registers are all physically duplicated on the Z80 chip! These duplicate registers are known as the alternate register set. The way the scheme is used is that you use the same instructions for both sets of registers, but you can only use one set at any one time. In order to begin using the second set of registers (or to get back to the main set) the programmer must use the EXX instruction. There is also an instruction to switch to the alternate A and F registers, whilst continuing to use the same general purpose registers. Through all changes in register sets the SP,IY,IX, R,I, and PC registers remain unchanged.

Having looked at the Z80 register set, let us now go on to look at the instruction set.

The instruction set of a microprocessor, is the list of instructions which it will execute. In this context an instruction means a code which the microprocessor interprets to mean things like; "Move the contents of register A to register H", or another code might mean, "add the contents of register A to the contents of register L and place the result in register A". These codes are called operation codes, commonly abbreviated to opcodes. When we place many such instructions into memory and tell the processor to begin executing them we are writing, then running, a program. All programs run on a microprocessor ultimately consist of a list of opcodes in memory. This holds true whether the program is written in a language like the Locomotive BASIC in your CPC, or whether you do it instruction by instruction with an assembler,like the DEVPAC assembler which we will be learning in chapter 5. From the foregoing it should be obvious that the more instructions are available, the more powerful the microprocessor becomes. The Z80 has become virtually unchallenged as the top 8 bit microprocessor in the world,

```
100 ; Stack pointer demo program
200   ENT #
300   LD A,#55 ; Put a known value into the A register.
400   PUSH AF  ; Push the A and F registers onto the stack.
500   ; The AF register at this point is now available for
600   ; the program to use.  When the program has finished
700   ; using them then it merely has to POP and A and F
800   ; registers to restore their original contents.
900   POP AF   ; Pop them off again.
1000  RET      ; Then we return to the caller.
```

The program above is written in assembly language. DEVPAC is one of the assembler packages which is available for the CPC. An assembler takes a program like the one above, and converts it into machine code instructions, rather like the BASIC interpreter built into your CPC. In assembly language each line has a number – as with BASIC. Anything which comes after a ";" character, is a comment – analogous to a REM statement in BASIC. The "ENT $" at line 200 is an assembler directive. This particular one directs the assembler to regard line 300 as the start – or ENTry point, of the program. You cannot run this program, unless you have previously used an assembler program to convert it into machine code. (Chapter 5 deals with the methods used to do this.)

In the following explanation, "(SP)−1" means the memory address below the one pointed to by the SP register, Whilst "(SP)" means the address pointed to by the SP register. And "(SP)+1" means the address one location above the one pointed to by the SP register, and so on.

Refer to the stack operation description in the text. The contents of the A register, the SP register, and the memory pointed to by the SP register, will be as follows, AFTER execution of each numbered line:

Line 200: A and F= xx (Where xx means unpredictable). SP assumed to contain Hex FC00. (SP), (SP)+1 contain the saved PC value of the program which executed a CALL instruction to call this stack demo subroutine. These will be called RET PC since they are the address to be placed in the PC when the next RET instruction is executed.
Line 300: A = Hex 55. F = xx. SP = FC00. (SP)+1,(SP)+2 = RET PC
Line 400: A = 55. F = xx. SP = FBFE. (SP) = xx (saved F reg.). (SP)+1 = 55 (Saved A red.). (SP)+2, (SP)+3 = RET PC
Line 900: A = 55. F = xx. SP = FC00. (SP)+1 = RET PC.
Line 1000: A = 55. F = xx. SP = FC02. (SP),(SP)+1 = xx.

Fig. 2.6: Stack operation demonstration program, plus comments.

mainly on the strength of its instruction set. The Z80 has 158 types of instruction. In the following pages we will look briefly at them all. Because this is only a Z80 primer, we will not go into great detail about them, readers wishing to do so are urged to read one of the Z80 books mentioned in the bibliography.

Z80 Instruction Set in Alphabetical Order

The following is a complete list of the Z80 instruction set. The format below is used for each instruction;

Number) Instruction mnemonic : Instruction description
FLAGS: list of affected flags.

The number preceding each instruction in the list, is to allow us to easily refer to specific instructions in other chapters. Many abbreviations and symbols are used in the instruction set description and these have the following meanings;

s = A place to get or put data. All the possibilities will be listed in the description of the instruction.

r = any of the registers A,B,C,D,E,H,L

rp = any of the register pairs BC,DE,HL

b = a bit number, 0–7 only.

xx = any hex value

(r) = the memory address pointed to by the register r. EG (B) means the address pointed to by the B register, whilst (HL) means the address pointed at by the HL register pair.

(IY + xx) = The address pointed to by the result of the calculation IY + xx. xx will normally be a third byte located after the opcode.

FLAGS affected, – means that the list of flags which follows may or will be altered by the instruction. Flags not mentioned will never be affected by the instruction.

Many instructions perform actions if certain conditions are true, in the descriptions the "cc" section should be replaced with one of the condition mnemonics, which have the following meanings:

Condition	*meaning*
Z	The zero flag is set.
NZ	The zero flag is not set.
C	The carry flag is set.
NC	The carry flag is not set.
PO	The P/V flag is not set.
PE	The P/V flag is set.
P	The S (sign) flag is set.
M	The S flag is not set.

To give an example JP cc,xxxx (Jump conditional), instruction 61 in the list, could have its cc field modified to any of the following:

JP Z,xxxx	= Jump to location xxxx if the zero flag is set.
JP NZ,xxxx	= Jump to location xxxx if the zero flag is not set.
JP C,xxxx	= Jump to location xxxx if the carry flag is set.
JP NC,xxxx	= Jump to location xxxx if the carry flag is not set.
JP PO,xxxx	= Jump to location xxxx if the P/V flag is not set.
JP PE,xxxx	= Jump to location xxxx if the P/V flag is set.
JP P,xxxx	= Jump to location xxxx if the S flag is set.
JP M,xxxx	= Jump to location xxxx if the S flag is not set.

Z80 INSTRUCTION LIST.

1) *ADC A,s :* The operand s and the carry bit are added to the A register. The operand s may be any of the following; xx,r,(HL) (IX + xx), or (IY + xx). Result in A. ,H,P/V,C.

FLAGS:ALL.

2) *ADC HL,rp :* The two register pairs HL and rp are added to one another, and the carry bit. Result in HL. rp can be BC,DE,HL, or SP.

FLAGS: S,Z,H,P/V,C

3) *ADD A,(HL):* The contents of the location pointed to by the HL register are added together. The result is placed in the A register.

FLAGS: S,Z,H,P/V,C

4) *ADD A,(IX+xx):* The memory location whose address is obtained by adding xx to the IX register contents, is added to the A register contents. Result in A. IX unchanged.

FLAGS: S,Z,H,P/V,C

5) *ADD A,(IY+xx):* As per instruction 4 above except IY is used in place of IX

6) *ADD A,n:* Add the byte of data n to the A register. Result in A register.
FLAGS: S,Z,H,P/V,C

7) *ADD A,r:* Add the contents of the register r to A. Result in the A register.
FLAGS: S,Z,H,P/V,C

8) *ADD HL,rp:* Add the contents of the register pair rp to the HL register pair, and place the result in the HL register pair. rp can be BC,DE,HL, or SP.
FLAGS: H,N,C

9) *ADD IX,rp:* Add the IX register contents to the contents of the register pair rp. Result in IX. rp can be BC,DE,IX,or SP.
FLAGS: H,N,C

10) *ADD IY,rp:* As per instruction 9, except read IY instead of IY.
FLAGS: H,N,C

11) *AND s:* The A register and the operand s are logically ANDed. The result is placed in the A register. s may be any of; r,xx,(HL),(IX+xx), or (IY+xx).
FLAGS: All.

12) *BIT b,(HL):* The specified bit of the memory location pointed to by the HL register pair is tested. The Z flag is cleared or set to show the result. Most other flags are affected too.
FLAGS: S,Z,H,P/V,N

13) *BIT b,(IX+xx):* The specified bit of the memory location pointed to by the calculation (IX+xx) is tested, and the Z flag set or cleared to indicate its state.
FLAGS:S,Z,H,P/V,N

14) *BIT b,(IY+xx):* As per instruction 13 above, except that IY is used instead of IX.
FLAGS:s,Z,H,P/V,N

15) *BIT b,r:* The specified bit of register r is tested, and the Z flag is set accordingly to indicate the state of the tested bit.
FLAGS:S,Z,H,P/V,N

16) *CALL cc,xxxx:* The specified condition is tested, and if found to be true the contents of the PC are pushed onto the stack, then the value xxxx is placed in the PC, to effect the subroutine call. Use RET to return from a subroutine called using this instruction. See the introduction to

this section for the list of conditions which can be tested for the cc part of this instruction.

FLAGS: NONE.

17) CALL xxxx: Very similar to the instruction above, but the call always happens, it is not conditional.

FLAGS: NONE.

18) CCF: Complement (Invert) the state of the carry flag.

FLAGS:H,N,C

19) CP s: Compare the contents of the A register with the operand s. The operand s may be any one of r,xx, (HL),(IX+xx),(IY+xx). Result of the compare is used to set the flags only, the actual result is not stored anywhere.

FLAGS: ALL.

20) CPD: The contents of the memory location which is pointed to the HL register are subtracted from the contents of the A register. The result is not stored. Then both the HL and BC registers are decremented.

FLAGS: S,Z,H,P/V,N,C

21) CPDR: This instruction is almost the same as CPD with an important variant, which is that the instruction automatically executes repeatedly, until a point where BC is decremented to zero, OR the A register matches the memory location pointed to by HL. A little known fact is that BC is decremented before the instruction executes.

FLAGS: S,Z,H,P/V,N

22) CPI: The contents of the memory location which the HL register pair points to are subtracted from the A register. The result is not stored. The HL register is then incremented, and the BC register decremented.

FLAGS: S,Z,H,P/V,N

23) CPIR: This is the automated version of CPI. Its function is similar in all respects except that it will be automatically executed repeatedly until the BC register pair content is zero, or until the content of the memory location pointed to by the HL register pair matches the contents of the A register. BC is incremented before execution.

FLAGS: S,Z,H,P/V,N

24) CPL: Complement (Invert) the contents of the A register.

FLAGS: H,N

25) *DAA:* This instruction changes the contents of the A register to BCD (Binary Coded Decimal) equivalent. that is, it decimally adjusts the A register contents.
FLAGS: S,Z,H,P/V,C

26) *DEC s:* s is decremented, that is, has one subtracted from its value. s may be : r,(HL),(IX+xx),or (IY+xx).
FLAGS: S,Z,H,P/V,N

27) *DEC rp:* The register pair selected are decremented. rp may be any of BC,DE,HL, or SP.
FLAGS: NONE.

28) *DEC IX:* The IX register is decremented.
FLAGS: NONE.

29) *DEC IY:* The IY register is decremented.
FLAGS: NONE.

30) *DI:* Disable interrupts. After execution of this instruction, interrupt requests arriving at the INT pin of the Z80 will be ignored. Interrupt requests presented at the NMI pin will still be actioned.
FLAGS: NONE.

31) *DJNZ xx:* The B register is decremented. If this does not result in the B register containing zero, then the value of the xx field is added to the value of the updated PC, thus effecting a jump. The addition is done using two's complement arithmetic, this means that the top bit of the xx byte is taken as being positive if clear, and negative if set. The bottom 7 bits are then added or subtracted with the PC as appropriate.
FLAGS: NONE.

32) *EI:* Enables the Z80 to respond to interrupt requests at the INT pin. The interrupt enable flip flop is set AFTER the Z80 executes the instruction following the EI.
FLAGS: NONE.

33) *EX AF,AF':* The alternate A'and F' register contents are exchanged with the main A and F registers. This effectively selects the alternate A' and F' registers for use.
FLAGS: ALL.

34) *EX DE,HL:* The two register pairs DE and HL contents are exchanged.

FLAGS: NONE.

35) *EX (SP),HL:* Exchange the L register contents with the contents of memory location pointed to by the SP register, and exchange the contents of the H register with the contents of the memory location at (SP + 1).

FLAGS: NONE.

36) *EX (SP),IX:* The low 8 bits of the IX register are exchanged with the memory location pointed to by the SP register, whilst the highest 8 bits of the IX register are exchanged with the contents of the memory location pointed to by (SP + 1).

FLAGS: NONE.

37) *EX (SP),IY:* The low 8 bits of the IY register are exchanged with the memory location pointed to by the SP register, whilst the highest 8 bits of the IY register are exchanged with the contents of the memory location pointed to by (SP + 1).

FLAGS: NONE.

38) *EXX:* The contents of all the general purpose registers are exchanged with the contents of the alternate register set. The registers exchanged are BC,DE, and HL.

FLAGS: NONE.

39) *HALT:* The CPU is halted. It idles, executing NOP instrucions until an interrupt occurs, or until it is externally reset.

FLAGS: NONE.

40) *IM 0:* Sets the interrupt mode to mode 0.

FLAGS: NONE.

41) *IM 1:* Sets the interrupt mode to 1.

FLAGS: NONE.

42) *IM 2:* Sets the interrupt mode to 2.

FLAGS: NONE.

43) *IN r,(C):* Reads the contents of the I/O port, whose 16 bit address is contained in the BC register, into the A register. Register C supplies the address bits A0–A7, and register B supplies A8–A15.

FLAGS: S,Z,H,P/V,N

44) *IN A,(xx):* The contents of the I/O register pointed to by the xx byte are loaded into the A register. Becuase the xx field is only a byte, and is used to supply address bits A0–A7, the address bits A8–A15 are supplied from the

original contents of the A register. In the context of the CPC 464 it is better to use the IN r,(C) instruction.

FLAGS: NONE.

45) *INC r:* Increment the register r.
FLAGS: S,Z,H,P/V,N

46) *INC rp:* Increment the selected register pair rp. The rp can be BC,DE,HL, or SP.
FLAGS: NONE

47) *INC (HL):* The contents of the memory location pointed to by the HL register are incremented, and stored back at the contents of the memory location pointed to by the HL register.
FLAGS:S,Z,H,P/V,N

48) *INC (IX+xx):* Increment the contents of the memory location pointed to by (IX + xx). The result is stored back at the same location.
FLAGS:S,Z,H,P/V,N

49) *INC (IY+xx):* Increment the contents of the memory location pointed to by (IY + xx). The result is stored back at the same location.
FLAGS:S,Z,H,P/V,N

50) *INC IX:* Increment the contents of the IX register.
FLAGS:NONE.

51) *INC IY:* Increment the contents of the IY register.
FLAGS:NONE.

52) *IND:* The I/O address which is addressed by the C register is written into the the memory location pointed to by the HL register pair. Then both the B register and the HL register pair are decremented.
FLAGS:S,Z,H,P/V,N

53) *INDR:* Similar to instruction above, except that the instruction repeats itself until the B register contains zero.
FLAGS:S,Z,H,P/V,N

54) *INI:* The contents of the I/O register pointed to by the contents of the C register, are written into the memory location pointed to by the HL register contents. Then the B register is decremented, and the HL register is incremented. During the addressing of the I/O register the C register provides address bits A0–A7, and the B

register, which here is used as a counter, supplies the A8–A15 bits. This severely limits the usefulness of this instruction within the context of the CPC 464.

FLAGS: S,Z,H,P/V,N

55) INIR: This is the automated version of INI. The contents of the I/O register pointed to by the C register are transferred to the memory location pointed at by the HL register. Then the HL register is incremented, and the B register is decremented. If the B register does not contain zero the instruction automatically executes again.

FLAGS: S,Z,H,P/V,N

56) JP cc,xxxx: If the specified condition is found to be true then a jump to the address xxxx occurs. The cc field can be any of the conditions listed at the start of this section.

FLAGS: NONE.

57) JP xxxx: Jump to location xxxx. The jump is not a conditional one, it always occurs.

FLAGS: NONE.

58) JP (HL): Jump to the memory location pointed to by the contents of the HL register.

FLAGS:NONE.

59) JP (IX): Jump to the address pointed to by the IX register.

FLAGS:None.

60) JP (IY): Jump to the address pointed to by the IY register.

FLAGS: NONE.

61) JP cc,xx: If the condition cc is true then jump to the address obtained by two's complement adding xx to the updated PC. (See instruction 31).

FLAGS: NONE.

62) JR xx: Twos complement add xx to the PC and jump to the address thus obtained. (See instruction 31).

FLAGS: NONE.

63) LD rp,(xxxx): Load the contents of the memory location pointed to by the value of xxxx, into the register pair rp. The rp pair can be any one of BC,DE,HL, or SP.

FLAGS: NONE.

64) LD rp,xxxx: The xxxx field values are loaded into the register pair rp. The pair rp can be any one of BC,DE,HL, or SP.

FLAGS: NONE.

65) *LD r,xx:* The register r is loaded with the value xx. r can be any one of A,B,C,D,E,H, or L.
FLAGS: NONE.

66) *LD r,r':* The register r' contents, are loaded into the register r. r and r' can be any one of A,B,C,D,E,H, or L.
FLAGS: NONE.

67) *LD (BC),A:* the contents of the A register are written into the memory location pointed to by the BC register pair.
FLAGS: NONE.

68) *LD (DE),A:* The contents of the A register are written into the memory location pointed to by the DE register pair.
FLAGS: NONE.

69) *LD (HL),xx:* The value xx is written to the memory location pointed to by the HL register pair.
FLAGS: NONE.

70) *LD (HL),r:* The contents of the register r are written to the memory location pointed to by the HL pair.
FLAGS: NONE.

71) *LD r,(IX + xx):* The contents of the memory location pointed to by (IX + xx) are loaded into register r.
FLAGS: NONE.

72) *LD r,(IY + xx):* The contents of the memory location pointed to by (IY + xx) are loaded into register r.
FLAGS: NONE.

73) *LD (IX +xx),xx:* The memory location pointed to by (IX + xx) is loaded with the value of the xx field.
FLAGS: NONE.

74) *LD (IY +xx),xx:* The memory location pointed to by (IY + xx) is loaded with the value of the xx field.
FLAGS: NONE.

75) *LD (IX+xx),r:* The memory address (IX+xx) has the contents of r written into it.
FLAGS: NONE.

76) *LD (IY+xx),r:* The memory address (IY+xx) has the contents of r written into it.
FLAGS: NONE.

77) *LD A,(xxxx):* Load the A register from the memory location xxxx.
FLAGS: NONE.

78) LD (xxxx),A: The contents of the A register are written into the memory location pointed to by xxxx.
FLAGS: NONE.

79) LD (xxxx),rp: The memory location pointed to by xxxx, has the contents of the lowest order register of the register pair rp written into it. Then the memory location immediately after xxxx has the contents of the higher order register of the pair rp written into it.
FLAGS: NONE.

80) LD (xxxx), HL: The contents of the L register are loaded into the memory location xxxx, then the contents of the H register are loaded into memory location xxxx+1.
FLAGS: NONE.

81) LD (xxxx),IX: The contents of the memory location xxxx with the contents of the low byte of the IX register, and the memory location xxxx+1 with the contents of the high byte of the IX register.
FLAGS: NONE.

82) LD (xxxx),IY: The contents of the memory location xxxx with the contents of the low byte of the IY register, and the memory location xxxx+1 with the contents of the high byte of the IY register.
FLAGS: NONE.

83) LD A,(BC): Load the A register with the contents of the memory location pointed to by the BC register pair.
FLAGS: NONE.

84) LD A,(DE): Load the A register with the contents of the memory location pointed to by the DE register pair.
FLAGS: NONE.

85) LD A,I: Load the A register with the contents of the interrupt vector register I.
FLAGS: S,Z,H,P/V,N

86) LD I,A: Load the interrupt vector register I with the contents of the A register.
FLAGS: NONE.

87) LD A,R: Load the A register with the contents of the R register.
FLAGS: S,Z,H,P/V,N

88) LD HL,(xxxx): Load the contents of the memory location pointed to by xxxx into the L register, and then load the contents of the memory location pointed to by xxxx+1 into the H register.
FLAGS: NONE.

89) *LD IX,xxxx:* The value of xxxx is loaded into the IX register.
FLAGS: NONE.

90) *LD IX,(xxxx):* The low byte of the IX register is loaded with the contents of the memory location pointed to by xxxx, and then the high byte of the IX register is loaded with the contents of the memory location xxxx +1.
FLAGS: NONE.

91) *LD IY,xxxx:* The value of xxxx is loaded into the IY register.
FLAGS: NONE.

92) *LD IY,(xxxx):* The low byte of the IY register is loaded with the contents of the memory location pointed to by xxxx, and then the high byte of the IY register is loaded with the contents of the memory location xxxx +1.
FLAGS: NONE.

93) *LD R,A:* Load the A register contents into the R (Refresh) register.
FLAGS: NONE.

94) *LD SP,HL:* Load the contents of the HL register pair into the stack pointer register, SP.
FLAGS: NONE.

95) *LD SP,IX:* Load the stack pointer with the contents of the IX register.
FLAGS: NONE.

96) *LD SP,IY:* Load the stack pointer with the contents of the IY register.
FLAGS: NONE.

97) *LDD:* Loads the contents of the memory location pointed to by the HL register into the memory location addressed by the DE register, and then decrements the contents of BC,DE, and HL.
FLAGS:H,P/V,N

98) *LDDR:* Another automatic instruction. This one does the same as LDD above, except that it repeats until the BC register pair contains zero.
FLAGS:H,P/V,N

99) *LDI:* The memory location addressed by HL is written into the memory location addressed by DE. Then DE and BC are incremented, whilst BC is decremented.
FLAGS:H,P/V,N

100) LDIR: This is the automatic version of LDI. The instruction has the same effect, but it is executed repeatedly, until the BC register has been decremented to zero.
FLAGS:H,P/V,N

101) LD r,(HL): The contents of the memory location pointed to by the HL register pair, are loaded into register r.
FLAGS: NONE.

102) NEG: This instruction negates the contents of the A register.
FLAGS:ALL.

103) NOP: No operation, this instruction is used to make the Z80 idle, or to pad out timing loops for time critical applications.
FLAGS: NONE.

104) OR s: A logical OR between s and the A register is performed. s can be any one of: r,xx,(HL),(IX+xx), or (IY+xx).
FLAGS:ALL.

105) OTDR: The B and HL registers are decremented. Then the contents of the memory location addressed by the HL register pair are output to the I/O port addressed by the contents of the C register.
FLAGS:S,Z,P/V,H,N

106) OTIR: The HL register is decremented, and the B register is incremented. Then, the contents of the memory location pointed to by the HL register is output to the I/O port addressed by the C register.
FLAGS:S,Z,H,P/V,N

107) OUT (C),R: The contents of register r are output to the I/O port pointed to by the BC register pair.
FLAGS: NONE.

108) OUT(xx),A: The contents of the A register, are output to the I/O port whose 8 bit address is specified by the value of xx.
FLAGS: NONE.

109) OUTD: The contents of the memory location pointed to by the HL register pair are output to the I/O port addressed by the contents of the C register. After this, the HL register pair and the B register are decremented.
FLAGS:S,Z,H,P/V,N

110) OUTI: The contents of the memory location addressed by the

HL register are output to the I/O port addressed by the C register contents. The B register is then decremented and the HL register pair is incremented.

FLAGS:S,Z,H,P/V,N

111) POP rp: The contents of the memory address pointed to by the SP register are loaded into the lowest byte of the register pair rp, then the SP register is incremented, and the memory location which it now points to is loaded into the high byte of the register pair rp.

FLAGS: NONE.

112) POP IX: The contents of the memory location currently addressed by the SP, are loaded into the low byte of the IX register. Then the SP is incremented and the contents of the memory location to which it now points are loaded into the high byte of the IX register.

FLAGS: NONE.

113) POP IY: The contents of the memory location currently addressed by the SP are loaded into the low byte of the IY register. Then the SP is incremented and the contents of the memory location to which it now points are loaded into the high byte of the IY register.

FLAGS: NONE.

114) PUSH rp: The SP register is decremented, and then the contents of the higher of the two registers in the pair rp are written into the memory location pointed to by SP. Then the SP is decremented again, and then the lower of the registers in the pair is written to the location at which the SP now points. rp may be any one of, BC,DE,HL,or AF.

FLAGS: NONE.

115) PUSH IX: The SP register is decremented, and then the contents of the highest byte of IX are written into the memory location pointed at by SP. Then , SP is decremented again, and the lowest byte of IX is written into the memory location at which SP now points.

FLAGS: NONE

116) PUSH IY: The SP register is decremented, and then the contents of the highest byte of IY are written into the memory location pointed at by SP. Then SP is decremented again, and the lowest byte of IY is written into the memory location at which SP now points.

FLAGS: NONE.

117) *RES b,s:* Clear (reset) bit b of the location s. The b specifier should be a number from 0 to 7, whilst s should be one of the following: r,(HL) (IX+xx), or (IY+xx).

FLAGS: NONE.

118) *RET:* Return from a subroutine. The program counter has its new contents popped from the stack (see pop explanations), and this effects a return to the program which pushed the old PC onto the stack.

FLAGS: NONE.

119) *RET cc:* IF the condition cc is met then a return from subroutine is executed. (See start of this section for a list of cc conditions).

FLAGS: NONE.

120) *RETI:* Return from interrupt. The return PC is popped from the stack as previously described. As this instruction should always be at the end of an interrupt service routine, it is nearly always preceded by an EI enable interrupts instruction. This allows further interrupts to once again pull an interrupt on the main program.

FLAGS: NONE.

121) *RETN:* Return from am NMI interrupt.

FLAGS: NONE.

122) *RL s:* Rotate left through the carry bit of the F register. The contents of s are shifted to the left by one bit, and the contents of bit 7 are shifted into the carry bit of the F register, and the previous state of the carry bit is shifted into bit 0 of s. The s operand can be any of the following: r,(HL),(IX+xx),or (IY+xx).

FLAGS: ALL.

123) *RLA:* Rotate the A register left through carry. This instruction functions identically to RL s, except that s is always the A register.

FLAGS:H,N,C

124) *RLCA:* The A register is rotated left and the carry bit in the F register always follows the state of bit 7 before shifting.

FLAGS:H,N,C

125) *RLC r:* As for RLCA, except that the register rotated can be any of the registers A,B,C,D,E,H, or L.

FLAGS:ALL.

126) *RLC (HL):* A rotate left in the manner described for RLCA is

performed on the memory location pointed to by the HL register pair.

FLAGS:ALL.

127) RLC (IX+xx): A rotate left in the manner described for RLCA is performed on the contents of the memory location pointed to by (IX+xx).

FLAGS:ALL.

128) RLC (IY+xx): A rotate left in the manner described for RLCA is performed on the contents of the memory location pointed to by (IY+xx).

FLAGS:ALL.

129) RLD: Rotate left decimal. This instruction swaps the low nibble (a nibble being four bits, that is half of a byte!), of the contents of the address pointed to by HL into the high bits, then the original high 4 bits of the memory location are moved to the low nibble of the A register, and finally the original low nibble of the A register is moved to the memory location low nibble !

FLAGS. S,Z,H,P/V,N

130) RR s: The selected location s is rotated right through the carry bit of the F register. The rotate is done as follows: the selected location is shifted by one bit to the right, with bit 0 going into the carry bit of the F register, whilst bit 7 of the selected location takes the previous state of the carry bit. s can be any one of; r,(HL),(IX+xx) or (IY+xx).

FLAGS: ALL.

131) RRA: Rotate the A register right through the carry bit. The rotate is done as in RR s, the difference being that A takes the place of s as the rotated location in this instruction.

FLAGS:H,N,C

132) RRC s: Rotate s right, with branch carry. The selected location contents are rotated right in the following manner: The contents of the selected byte are shifted right. Bit seven takes the state previously held by bit 0. The carry bit always takes the state of bit zero before the shift. s can be any of: r,(HL),(IX+xx), or (IY+xx).

FLAGS:ALL.

133) RRCA: Rotate the A register right with branch carry. As for RRC s, except that the place of s is taken by the A register.

FLAGS:H,N,C

134) RRD: Rotate right decimal. The low nibble of the A register is moved to the high nibble of the memory location pointed to by HL. In that same memory location the high nibble replaces the low nibble, whilst the original low nibble in the memory location replaces the A registers low nibble.

FLAGS: H,N,Z,S,P/V

135) RST p: Restart at p. This instruction causes the PC contents to be pushed onto the stack, then the specified p value is placed into the PC register. The next instruction is then fetched from location p which can be any of the following HEX values; 00,08,10,18,20,28,30,or 38.

FLAGS: NONE.

136) SBC A,s: The carry bit plus the specified s location are subtracted from the contents of the A register and the result is placed into the A register. s may be any one of; r,xx,(HL),(IX+xx), or (IY+xx).

FLAGS:ALL.

137) SBC HL,rp: The contents of the specified register pair rp are added to the carry bit, then the total is subtracted from the contents of the HL register pair. The result is stored back into the HL register pair.

FLAGS:ALL.

138) SCF: Set the carry flag.

FLAGS:H,N,C

139) SET b,s: The selected location s has bit b set. The value for b may be from 0 to 7. The selected location s may be any one of: r,(HL),(IX+xx) or (IY+xx).

FLAGS: NONE.

140) SLA s: Arithmetic shift to the left in the selected location s. The selected location contents are shifted left, and bit zero is set to zero, whilst bit seven of the selected location is shifted into the carry bit of the F register. s may be any one of r,(HL),(IX+xx),or (IY+xx).

FLAGS:ALL.

141) SRA s: Shift right arithmetic. The selected location s has its lowest seven bits shifted to the right. The original state of bit zero is transferred into the carry bit. s may be any of the locations listed above in SLA s.

FLAGS: ALL.

142) SRL s: Logical shift to the right of the selected location, s. The

contents of s are shifted one bit to the right. Bit seven is set to zero, and the state of bit zero is shifted into the carry bit of the F register. s may be any of the locations listed in SLA s above.

FLAGS:ALL.

143) SUB s: Subtract s from the A register. The s operand is subtracted from the A register, and the result is stored back into the A register. s may be any one of; r,xx,(HL),(IX+xx),or (IY+xx).

FLAGS: ALL.

144) XOR s: Exclusive OR the A register and the selected location, s. The result is stored back into the A register. s may be any of; r,xx,(HL),(IX+xx), or (IY+xx).

FLAGS:ALL.

Conclusion.

As you can see, the Z80 instruction set is really large especially when you take the variants of each standard instruction into account. Use this section as a reference, when you come to examine some of the programs which are presented later in the book for driving hardware add–on projects. One final point about the Z80: there are a few of these instructions which it is not wise to use in the context of the CPC. This is either because they do not do what you expect them to, or because they interfere with the firmware program inside the CPC, usually crashing it. You cannot damage your computer using these instructions, but it will go haywire, and you will have to turn it off and start again! The instructions concerned will be highlighted in chapter 5, which covers assembler language programing, using the DEVPAC assembler. That concludes the Z80A description.

CHAPTER 3

Hardware Projects

In this chapter we will look at some hardware projects which you can build to enhance and extend the usefulness of your CPC computer. At least one of each project has been built in preparation for this book. Therefore the circuit diagrams represent a finished form. See page iii for details of parts available.

Readers who are unfamiliar with computer hardware and its terminology, are urged to read Appendix 6. This is entitled "Begin here!", and will hopefully give you a good grounding in the subject matter of this chapter.

THE SIGMA POWER SUPPLY UNIT.

The CPC internal power supply produces +5 Volts only. This is because the CPC utilises integrated circuits which require only a power supply of +5 volts DC. There are some integrated circuits used in the projects about to be described however, which need +5 volts and also +12 volts DC. Some need −12 volts DC as well. Because of this it becomes necessary to provide an external power supply unit which will make available the following;

+5 volts at 3 Amps.
+12 volts at 1 Amp .
−12 volts at 1 Amp .

The +5 volts is provided to save any risk of overloading the CPC internal 5 volt supply. If you do not feel up to constructing the external power supply unit, commercially produced units can be bought. The amperage ratings are only nominal so any unit capable of producing something close to the above ratings will suffice.

Figures 3.1 and 3.2 show the circuit diagram for the external power supply unit, which, (for reasons which will become apparent if you look at the cover of this book) I have named the SIGMA power supply unit.

The unit is designed along very conventional lines. The mains voltage is transformed by T1, which is a multi−tapped transformer. This means that it has several secondary windings to produce several different AC (Alternating Current) voltages. This kind of transformer is invaluable when designing a power supply, from which you wish to obtain several different DC voltages. (Incidentally a multi voltage power supply, is often reffered to as a multi rail power supply. Nothing to do with trains I'm afraid, the term "Rail" here means a power supply voltage, as in, the +12 Volt rail.)

Fig. 3.1: SIGMA PSU diagram # 1

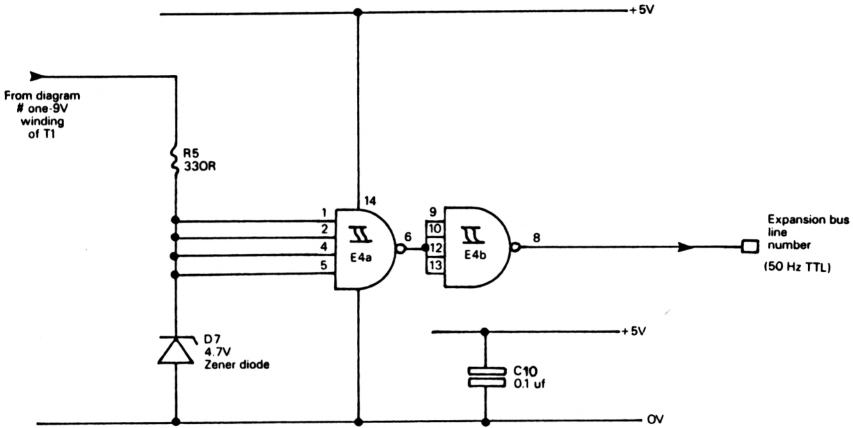


Fig. 3.2: SIGMA PSU diagram # 2

Each of the AC voltages is rectified by bridge rectifiers BR1 to BR4, into raw DC. In this case we use the word raw to mean that although, at this stage the voltage is now DC, it has not yet been regulated, or smoothed. The smoothing comes next. The smoothing is done by the capacitors C1 to C6. These capacitors store a charge so that any small shortfall in the voltage coming out of the Bridge rectifier, can be met from the stored charge in the capacitors. Capacitors do not function in the same way as batteries, they can hold enough of a charge to meet the full supply demand for a few Milliseconds only. The smoothed voltages are then fed into the three voltage regulators.

The function of a regulator is to ensure a constant supply voltage, no matter how heavy or light the load on its output. In digital circuitry the power consumption can fluctuate quite sharply, depending upon what activity is taking place. This makes it essential that computer power supplies be well regulated. The regulator outputs are all further smoothed, and diodes D1–D5 are fitted to prevent the three voltage rails going to the opposite potential, during power up or power down. The regulators can usually look after themselves. They are internally protected against short circuits and overheating. Thus if they overheat, or if their outputs are shorted to ground, they will shut themselves down. (Don't try out the short circuit protection, you will create a lot of nasty voltage spikes on the supply lines, which may damage any connected circuitry. It's only there as protection in case of accidents!) The resistors R2, R3, and R4 "bleed" the smoothing capacitors, so that within a few seconds of powering the unit down, any residual voltages have drained away.

The 21 volt regulator formed by R1, TR1, D6,D8 and smoothed by C1 is a simple series pass regulator. The two zener diodes hold the base of TR1 at around 21 volts and therefore its emitter cannot rise above 21.6 volts – easily

close enough for what we need. In practise the supplied voltage is about 21.5 volts after connector losses.

SW1 is the mains switch, and FS1 protects the transformer against overloads – it should be a slow blow 1 amp type.

The parts list is shown in Figure 3.3. You will see that RS components part numbers have been given for many items in the list. These are provided to give an easy way of identifying the exact type of component recommended. Having said that however, none of the components are critical, equivalents should work perfectly well. The transformer shown has two of its secondary windings connected in paralell to give the required Amperage. Enclose the unit in a metal box , and don't forget that the mains earth should be bonded to the chassis of the box. This gives you the satisfaction of safety. Fuse FS1 should NOT be omitted, though switch SW1 may be.

Parts list for the SIGMA power supply unit

Capacitors			
C1	100 mfd 63 Volts	(EG RS 103–705)	
C2	1000 mfd 25 Volts	(RS 103–610)	
C3	1000 mfd 25 Volts	(RS 103–610)	
C4	1000 mfd 25 Volts	(RS 103–610)	
C5	4700 mfd 25 Volts	(RS 103–632)	
C6	100000 mfd 40 Volts	(RS 102–409)	
C7–C12	0.1 mfd 25 Volts	(RS 124–140)	(6 required)
Resistors			
R1	10K Ohms 1 Watt	(RS 143–488)	
R2–R4	5.6K Ohms 1 Watt	(RS 143–450)	(3 required)
R5	330 Ohms 1Watt	(RS 143–286)	
Semiconductors			
D1–D5	400V 3 Amp. Diodes	(RS 261–306)	
D6	10 Volt Zener diode	(RS 282–684)	
D7	11 Volt Zener diode	(RS 282–690)	
D8	4.7 Volt Zener diode	(RS 283–003)	
BR1 & BR4	Bridge rectifiers 1 Amp	(RS 261–328)	(2 required)
BR2 & BR3	Bridge rectifiers 6 Amp	(RS 262–084)	(2 required)
E1	1 Amp +12 Volt regulator	(RS 305–894)	
E2	5 Amp + 5 Volt regulator	(RS 307–301)	
E3	1 Amp – 12 Volt regulator	(RS 306–055)	
E4	TTL 7413 chip	(RS 305–541)	
TR1	TIP 31A Transistor	(RS 294–205)	
Transformer			
T1	Special transformer.	Available only from Halstead Designs Ltd – see notes below.	
Miscellaneous			
FS1	Fuseholder (PCB mounting)	(RS 412–784)	
Fuse	1 Amp 20mm quick blow	(RS 412–144)	
Printed circuit board (see note 2 below)			
Solder			
Cable			

NOTES:
1) Note that the RS part numbers quoted may be supplied in packs containing more than one part.
2) Complete kits of parts are to be available from Halstead Designs Ltd.

Fig. 3.3: SIGMA PSU parts list

Project 1 RS 232 interface – Version "A"

This first project is a relatively simple one to convert the parallel data which is output to the printer port of the CPC, into a serial data stream which will enable you to output ASCII text to printers, VDUs and a number of other devices which are equipped to receive data in a serial form. This is the output

only version; there will be another version of RS 232 interface later in this chapter which will provide two way communication. If you only want to get a serial representation of the data presented at the CPC printer port, then use this project – the version "A" RS232 interface. Here are the specifications for the version "A" RS 232 interface;

1) Takes data from the CPC printer port and converts it to a serial data stream at one of the following baud rates (where one baud is roughly equal to 1 cycle per second .

   75 baud
   150 baud
   300 baud
   600 baud
   1200 baud
   2400 baud
   4800 baud
   9600 baud

2) Serial data is output at full level swings minus and plus 12 volts to drive most serially interfaced devices. No modem type signals are provided (EG. DTR,DSR,CD), the user must obtain these from static logic levels if they are needed.

3) Will respond to XOFF (ASCII DC3, HEX 13) and XON (ASCII DC1, HEX 11) control signals – see below.

4) Standard version derives baud rate clocks from the master clock of the CPC. (An optional section of circuitry is also detailed in case of this not being satisfactory).

5) Visual indication is provided of the state of the "busy" signal state. The "busy" signal shows when the serial interface is ready to take another byte of data from the CPC. The indicator is an LED (light emitting diode) which lights to indicate that the serial interface is signalling the CPC that it is "busy".

If you understand the purpose of the XOFF and XON signals mentioned above, please skip the following two paragraphs.

When a printer receives data from a computer, be it serially or otherwise, the speed at which data is received can easily outstrip the speed at which the printer can actually place the characters on paper. To partially alleviate this problem, many manufacturers provide printers with an internal buffer. This consists of some local memory which acts as a first–in first–out queue. Fig 3.4 shows this arrangement. As each new character comes in from the computer it is placed at the start of the queue. The contents of the queue constantly bubble upwards, the character at the head of the queue at any

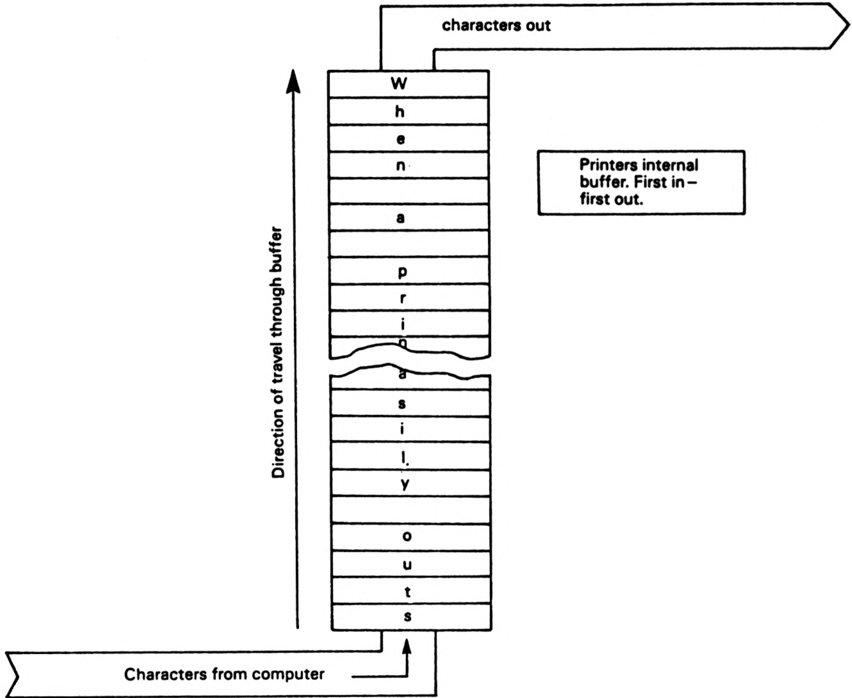


Fig. 3.4: FIFO printer buffer diagram (1 of 2)

given moment is always the next one to be printed. Fig. 3.5 shows the state of the queue contents after the character placed in it in figure 3.4 has moved to the head of the queue, by means of the printer itself having emptied enough space in the queue to allow it to bubble up to the top.

This representation of a printer buffer is simplified, but correct. A common size for a printer buffer is 2048 bytes. (Usually abbreviated in equipment manufacturers literature to 2Kb). For short bursts of printing 2Kb will be ample to store all the characters to be printed. But what happens when larger amounts of printing take place causing the printers internal buffer to be full? The computer must somehow be told to stop sending characters, or else data will be lost. The method used in most serial schemes is called XOFF/XON.

When the printer wants the computer to stop sending characters for printing, it sends the XOFF character (Hex 13) to the computer. When it is able to accept more characters, having cleared or reduced the contents of its buffer, the printer will send the XON (HEX 11) character to the computer, to tell it to resume sending characters. If the printer has a keyboard attached, you can usually generate the XOFF character by typing CTRL/S, and an XON character by typing CTRL/Q. Some VDU units also use the XOFF/XON protocol when run at high speeds such as 9600 baud. Note that not all printers

have a buffer; in this case some transmit XOFF after reception of just one character, then XON when that character has been printed.

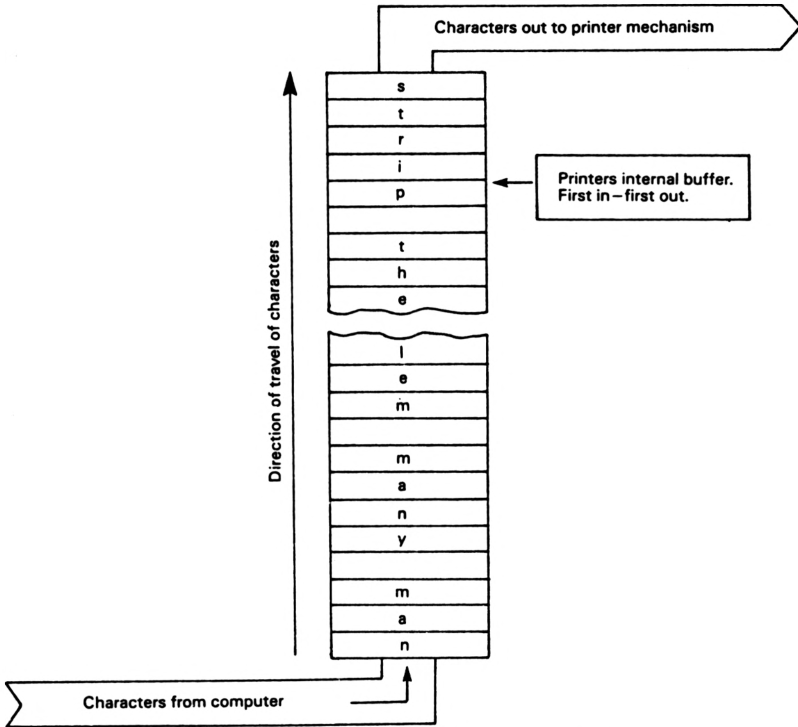


Fig. 3.5: FIFO printer buffer diagram (2 of 2)

Now to the version "A" RS 232 circuit diagrams. (If the serial transmission terminology used here confuses you, please refer to the brief description of serial data transmission in appendix 6.) Figures 3.6 and 3.7 show the hardware diagrams for the interface. Fig 3.6 shows the UART chip. (UART stands for Universal Asynchronous Receiver Transmitter). Fig. 3.6 also shows the various gates, and the data send and reception circuits. The CPC outputs a character to its printer port, then briefly pulses low the printer port strobe line output. This indicates to the outside world that the data available at the port represents a character to be printed. In the case of this circuit the strobe line latches the data into the transmit register of the UART. This register is used to hold the character currently being sent serially. The UART now automatically begins to transmit the character, and simultaneously its TBMT line goes low. The TBMT (Transmitter Buffer Empty) signal, when low,indicates that the UART is unable to accept any fresh characters for transmission. When TBMT is high the UART is able to accept another character. The TBMT signal suitably gated is sent back to the CPC "busy" line. When the CPC "busy" line is taken high the computers software will not send

any more characters to the printer port, until it sees the "busy" line go to a logic low level. So TBMT is inverted by gate E2c, to give the correct logic level.

The gate chips E3, E4, and E2d form two detectors. One of these detects when an XOFF character has been received from the printer, and the other detects reception of an XON character. These produce two active low signals which are called, naturally enough, XOFF BAR and XON BAR. These two signals control the bistable latch formed by gates E2a and E2b. This latch is used to remember which was the last received from the printer, XOFF or XON. Characters received other than XOFF or XON have no effect. Thus the logic level at pin 3 of gate E2a indicates which was the last received. If pin 3 is high then XON was last received, whereas if pin 3 is low then XOFF was last received.

In the overall scheme of the circuit then, the "busy" signal is taken to its true condition if either TBMT is true, or if XOFF was the last character received from the printer. As we saw previously the "busy" signal being true prevents any further output from the CPC printer port.

Also on Fig. 3.6 is the 1488 RS 232 transmit chip E5a. This takes the standard TTL logic levels and effectively changes them to corresponding −12 volts and + 12 volt levels. This greatly increases noise immunity on the cable carrying the data to the printer. (Up to 50 feet seems to work perfectly well). It also allows your printer to be tucked away from the computer and, more importantly, out of earshot! I did not want to add a 1489 chip, which performs the opposite function to the 1488, as only a quarter of a 1489 would be used. Instead two diodes, D1 and D2, and a resistor, R1, limit the incoming voltage to between +5.6 volts, and −0.6 volts, and together with transistor TR1 form the data receptor. This is acceptable because the standard transmit chips now used in just about all printers and terminals, feature current limiting. The transistor TR1 inverts the incoming data stream− a job also usually performed by a 1489 chip. This inverted data stream is fed into pin 20 of the UART, which is its serial input.

Now, let us examine the purpose of link LK1. (And in this connection all readers should perhaps peruse appendix 7 after completing this section.) Let us also clarify another important point about the CPC printer port. LK1 is provided to allow you to use a printer or VDU which transmits only a seven bit code set, with the eighth bit being used as a parity bit, or always being set to a logic high. If you are using a seven bit printer or terminal, then link LK1 from pin 5 of gate E3c to 0 volts. If you are using an eight bit device then LK1 should be jumpered from pin 5 of gate E3c to pin 5 of E1. I could not find any mention in the CPC 464 user manual, but the printer port is definitely a seven bit port. Consulting the technical manual reveals that bit seven is tied to logic low inside the machine so, no matter what data you output to the printer port, the top bit (that is the highest order bit), will always be received by the device connected to the port as a logic low. Because of this, transferring such things as machine code via the printer port cannot be done successfully.

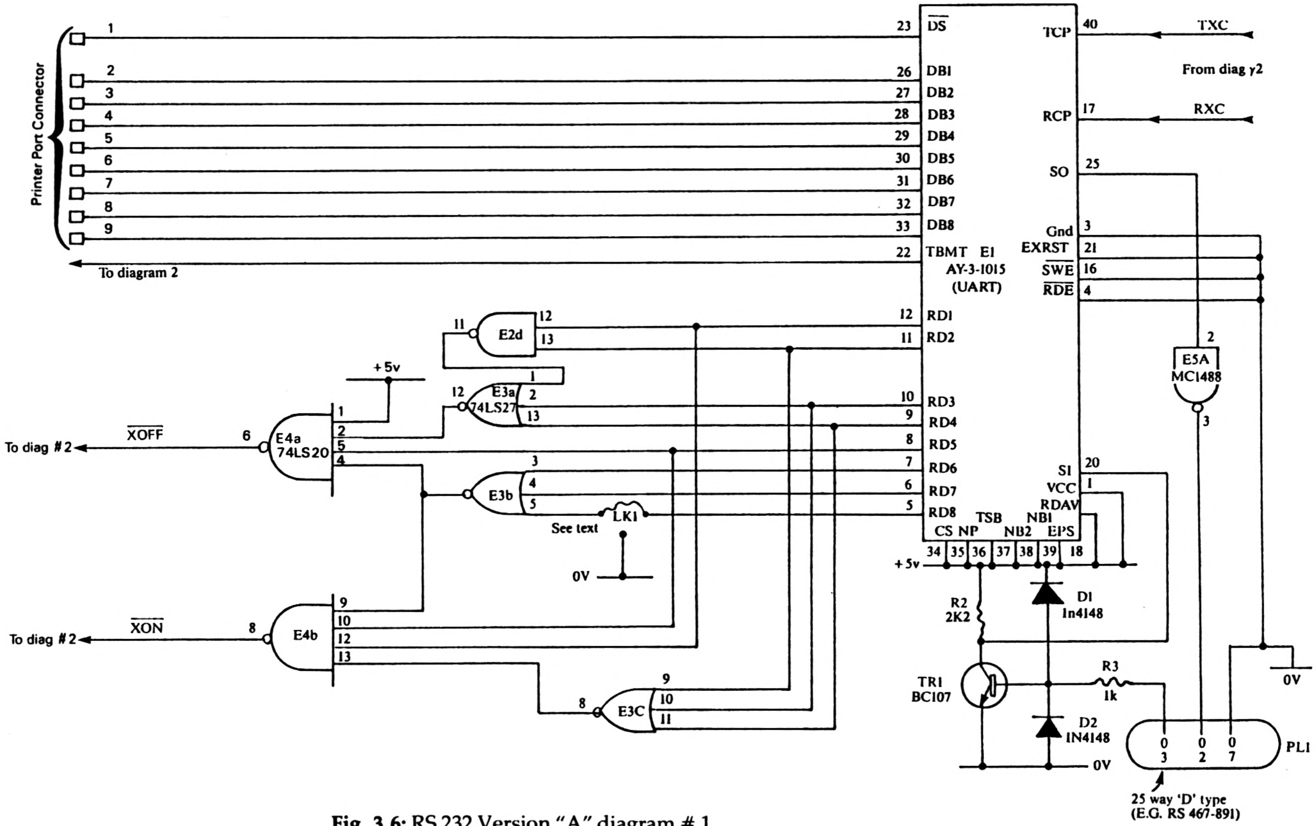


Fig. 3.6: RS 232 Version "A" diagram # 1

Looking at Fig 3.6. you can see that many leads from the UART are tied low or high. A great many of these are used for selection of the serial data characteristics. Using the connections as shown, serial data will be sent with;

8 data bits
2 stop bits
No parity checking.

If you need to change these characteristics at all, you are referred to appendix A where the data sheet for the UART is reproduced, detailing all the options available and how to select them.

Finally in Fig 3.6, the two clock signals coming into the top right hand side of the diagram – named TXC and RXC– are the clock pulses for the transmit and receive sections of the UART. The data transmission and reception rates are decided by the frequency of the signal which is fed into these two inputs. To further complicate matters the UART divides these two signals by sixteen, so these two lines should contain a frequency which is sixteen times the desired baud rate. (As we shall see a little later in the version "B" RS 232 interface, other serial transmitter receiver chips are somewhat more programmable.)

Looking now at Fig. 3.7 the clock signals can be derived from the master clock of a CPC 464, which runs at a frequency of 4 megaHertz, (4 million cycles per second). If you are going to use the standard circuit, and feel that you can proficiently wield a small sized soldering iron, here is a modification which you can make to a CPC 464 only, which will save you having to have a connector on both the printer port and the expansion bus when using only the version "A" RS 232 interface. The purpose of this change is to provide the 4 Mhz clock signal at the printer port, as well as at the expansion port. PLEASE NOTE: If you perform this modification you will invalidate your warranty. If you do not want to perform this modification you can still use the version "A" RS 232 interface, but you will have to use a connector to the 50 way expansion bus to pick up the 4 Mhz signal from line number 50 of the expansion bus.

Ensure that both leads from monitor to keyboard unit are unplugged before you begin.

The modification consists of the following simple steps. Firstly, remove the main CPC 464 circuit board from the keyboard. To do this, simply remove the six recessed posidrive screws in the base of the keyboard unit. Then very carefully remove the plugs which connect the cassette unit and keyboard to the board. Carefully solder a small gauge wire from the inner most end of finger number 17 on the printer port. (Use the diagram in appendix five page 2 of the CPC 464 user manual to locate finger 17.) Then, cut the wire to length so that it will reach the end of resistor R141 which is nearest the 50 way expansion port connnector. Fig. 3.9. shows the whereabouts of these points on the component side of the main CPC 464 board. Be careful to ensure that

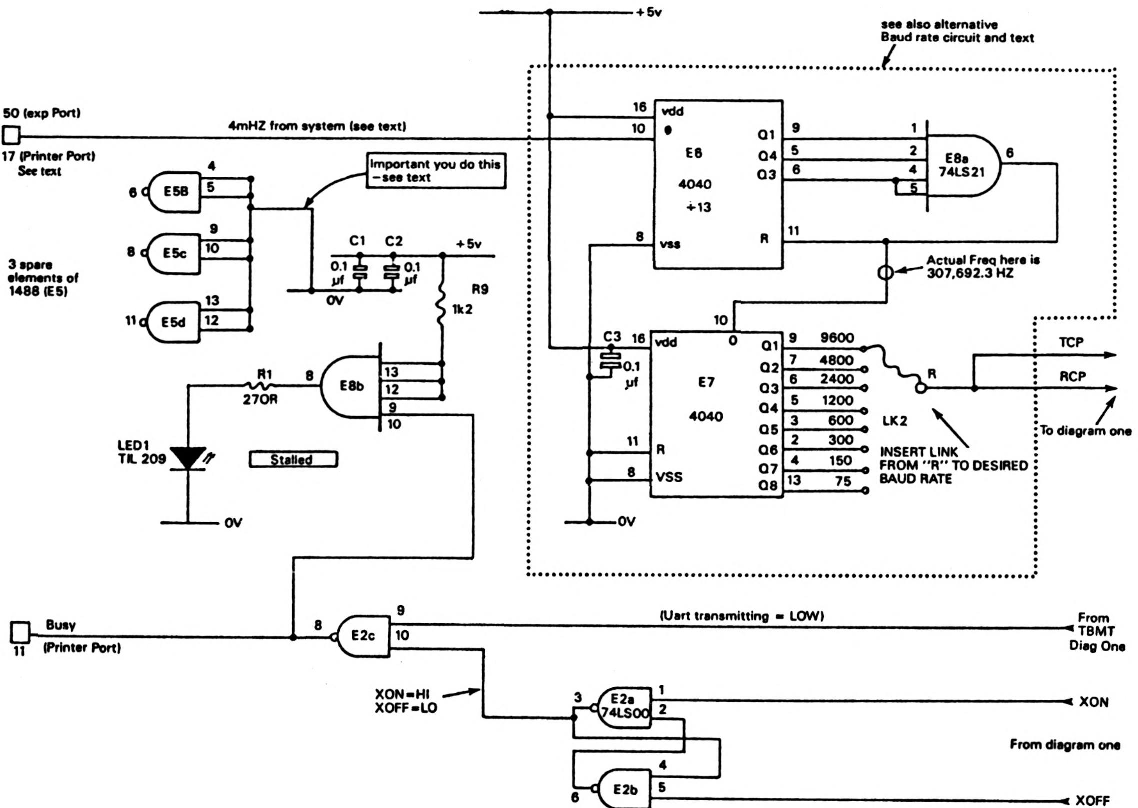
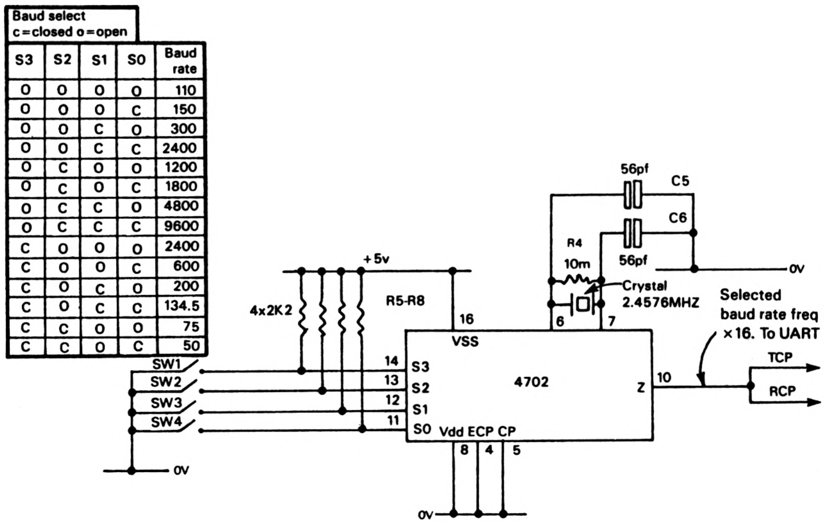


Fig. 3.7: RS 232 Version "A" diagram # 2

no solder flows onto any other part of the edge connector finger, especially the part which forms a wiping contact with an edge plug. Ensure that no solder splashes have found their way onto any other part of the board. The modification is now completed, reassemble the unit in the reverse order to that used to take it apart.

In the standard version (as shown in Fig. 3.7) the 4MHz signal from the CPC 464 is fed into first one divider chip then another. Each of these are 4040 CMOS 12 stage divide by two counters. The first divider has a count length of 13 (HEX D) – detected by gate E8a. When 13 is reached, E8a resets the first divider, and also adds one to the second divider count. Thus, we feed 4mHz into E6, and, after division, we get a frequency of 307692.3 Hz. This is divided by two in the successive stages of the E7 divider, so that at pin 9 of E7 we get a frequency of 153846 Hz – which when the UART has divided it by 16, will make 9616 Hz in round figures. This is easily close enough to 9600 Hz to be usable.



Baud select
c=closed o=open

S3	S2	S1	S0	Baud rate
o	o	o	o	110
o	o	o	c	150
o	o	c	o	300
o	o	c	c	2400
o	c	o	o	1200
o	c	o	c	1800
o	c	c	o	4800
o	c	c	c	9600
c	o	o	o	2400
c	o	o	c	600
c	o	c	o	200
c	o	c	c	134.5
c	c	o	o	75
c	c	o	c	50

Fig. 3.8: Alternative Baud rate generator circuit

I have tried the RS 232 Version "A" interface with several different printers and terminals, and at both 9600 and 600 baud never had a single bad character noted. I therefore have no reserve in saying that for nearly all applications the standard method of baud clock derivation, as shown in Fig. 3.7 will be fine. If, when you try it out, you get any extreme problems, you could resort to the alternative arrangement show in Fig. 3.8. (Which was in fact going to be the standard, until the cheaper method was devised!). In the alternative BAUD

rate derivation scheme, a Ferranti F 4702 BPC baud rate generator chip is used, and a four way code is presented at the baud rate select inputs of the chip. The circuitry of Fig. 3.8 is used in place of all the components shown within the dotted lines in Fig 3.7.

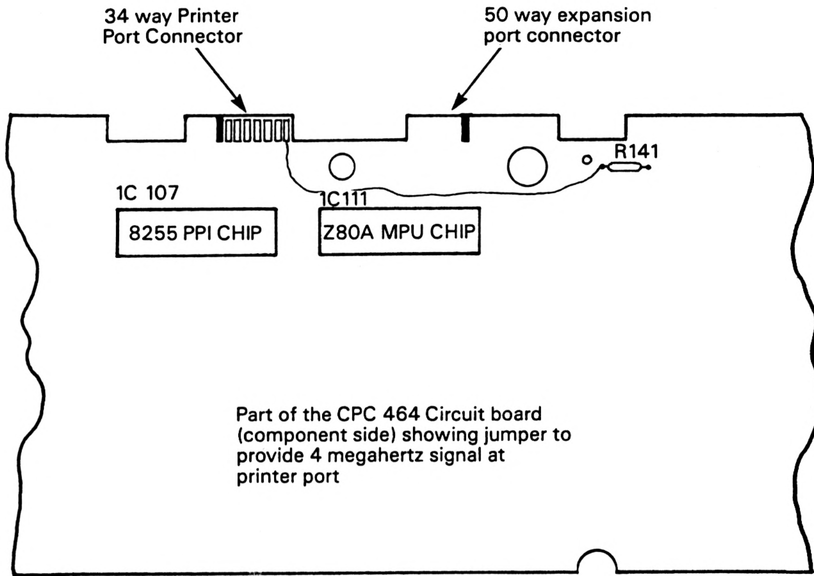


Fig. 3.9: Location of modification wire for 4mHz pick up.

Having used the standard scheme to derive the frequency for 9600 baud, the rest follow down in sequence through each stage of second divider– E7. Because the baud frequencies are all simultaneously available using this scheme, you may elect to connect a jumper wire from the required output of E7, to the TCP and RCP inputs of the UART. (Pins 40 and 17 respectively).

The remaining circuitry on Fig 3.7 consists of: E8b which is used to drive an LED (Light Emitting Diode). This visually indicates the state of the "busy" line back into the CPC 464 . When the LED is lit, the computer is being told not to send; when the LED is off the computer is free to send a character. It is VERY IMPORTANT to tie all the unused inputs of E5 to zero volts. If this is not done, the chip can become very hot due to noise at the inputs making the spare elements unstable. In one extreme case I saw a 1488 which had a burn hole in it; remember, there is a 24 volt potential in the chip! Gate E2c takes the "busy" signal true if either XOFF is send from the printer, OR if TBMT is low.

When you have made this RS 232 interface you will need to check its correct function. Throughout the computer industry the most popular method of printer testing is to output an ASCII swirl to the printer to be tested. This consists of all the printable ASCII characters being printed in a rotating pattern. (See appendix four for ASCII character set.) This ensures that all characters can be printed at all positions on each line, and has the added advantage that any major problems with data transmission to the printer, show as breaks in the regular sloping pattern of the print. Fig. 3.10 shows a program for the CPC 464 which will output such a test pattern, along with a sample of a swirl test pattern print out

```
50  '** TEST THE PRINTER WITH 60 LINES OF ASCII SWIRL PATTERN **
100 MODE 2: STRM=8 ' Set up scrn mode and set printer stream to 8
105 INPUT "How many print columns <80>";W: IF W > 132 THEN
    105 ELSE IF W=0 THEN W=80
110 FOR I=32 TO 32+79: C$=C$+CHR$(I): NEXT I:
    IF LEN(C$) < W THEN 110
115 C$=LEFT$(C$,W): CLS
120 PRINT "PRINTING...";: LOCATE 16,1: PRINT "Of 60 lines"
130 FOR I=1 TO 60: LOCATE 12,1: PRINT I;: PRINT #STRM,C$
140 C$=MID$(C$,2,LEN(C$))+LEFT$(C$,1): NEXT
10000 END
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno 
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !
'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"
()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#
)*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$
*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%
+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&
,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'
-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'(
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()
/0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+
123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,
23456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,-
3456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,-.
456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,-./
56789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,-./0
6789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,-./01
789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,-./012
89:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno !"#$%&'()*+,-./0123
```

Fig. 3.10: ASCII test swirl generator program, and sample printout.

The exact method of connecting to the SIGMA power supply unit will be left to the reader, as it really depends upon how you actually box the project, and also how many others you intend to build from this book, – Read on!

Finally, the voltage supply pins of all the chips used are shown in Fig. 3.11

'E' Number	Device description			Power Connection Pins				
	Device type	Alternative device(s)	Functional description	+5V	+12V	−12V	Ground	others
E1	AY-3-1015	–	UART	1	–	–	3	–
E2	7400	–	Quad 2 input NAND gates	14	–	–	7	–
E3	74LS27	–	Triple 3 input NOR gates	14	–	–	7	–
E4	74LS20	–	Dual 4 input NAND gates	14	–	–	7	–
E5	MC1488	SN 75188	Quad RS 232 Line driver	–	14	1	7	–
E6	4040B	–	12 stage divider	16	–	–	8	–
E7	4040B	–	12 stage divider	16	–	–	8	–
E8	74LS21	–	Dual 4 input AND gate	14	–	–	7	–
E9	4702 BPC	–	Baud rate generator	16	–	–	8	–

Fig. 3.11: Voltage supply pins for chips used in version "A" RS 232.

Parts list for the RS232 version "A" interface

Capacitors

```
C1-C3     0.1 mfd disc ceramic 16 Volt          (RS 124-178) (3 required)
C5-C6 *   56 pf polystyrene bead 16 Volt        (RS 114-676) (2 required)
```

Resistors

```
R1        270 Ohm half    Watt                  (RS 132-159)
R2        2K2 Ohm quarter Watt                  (RS 131-299)
R3        1K  Ohm quarter Watt                  (RS 131-255)
R4 *      10M Ohm half    Watt                  (RS 133-330)
R5-R8 *   2K2 Ohm quarter Watt                  (RS 131-299) (4 required)
R9        1K2 Ohm quarter Watt                  (RS 131-261)
```

Semiconductors

```
E1        AY-3-1015 UART chip
E2        TTL 74LS00 Quad two i/p NAND gates    (RS 307-480)
E3        TTL 74LS27 Triple 3 i/p NOR  gates    (RS 309-060)
E4        TTL 74LS20 Dual 4 input NAND gates    (RS 307-553)
E5        MC 1488 line driver chip.             (RS 309-587)
E6 &E7    CMOS 12 stage divider chip            (RS 307-187) (2 required)
E8        TTL 74LS21 Dual 4 i/p NAND gates      (RS 305-210)
```

```
E9 *      Baud rate generator chip                 (RS 303-517)
Crystal*  2.4576 Mhz crystal                       (RS 303-359)
LED 1     .2 inch LED                              (RS 586-475)
TR1       BC107 transistor                         (RS 293-527)
D1 & D2   Diodes 1N4148 500mw silicon diodes                  (2 required)


Miscellaneous

Sockets for all chips
Printed circuit board - see note 2 below
SW1-SW4 * Rate select switches for optional       (RS 334-548)
          Baud generator. (4 way DIL switch)
PL2       25 way "D" connector female.             (RS 467-891)

          34 way cable and IDC connector to
          connect to the CPC printer port

          Suitable plastic case.
      -------------------------------------------------------------

NOTES:
        1) Note that the RS part numbers  quoted  may  be  supplied  in
           packs containing more than one part.

        2) Complete kits of parts are to  be  available  from  Halstead
           designs LTD.
```

Fig. 3.12: RS232 version "A" interface parts list.

MULTI PROJECT BOARD
(Projects 2 to 6)

The speech synthesiser and the four projects which follow it are all designed to be on one board. This is because these five projects all make use – in one way or another – of the two PPI chips on the board. (See chapter one and appendix three for details of the PPI chip). Having them all on one board cuts down on the amount of address decoding required as well, with only a single decode required. The block diagram for the multi–project board is shown in Fig. 3.13. Parts lists for each project will be given separately, listing only those parts which you would need to build it in isolation. Because some parts are shared between projects take care not to buy two of a part, when you only need one. At the end of the description of the multiproject board a complete list of multiproject board parts will be given.

Project Two: Speech synthesiser.

The digital synthesis of speech has only recently been made cheap enough to be applicable to home computers. It is now possible to buy integrated circuits for less than ten pounds, which will reproduce intelligible speech. There are several approaches to synthesising speech electronically.

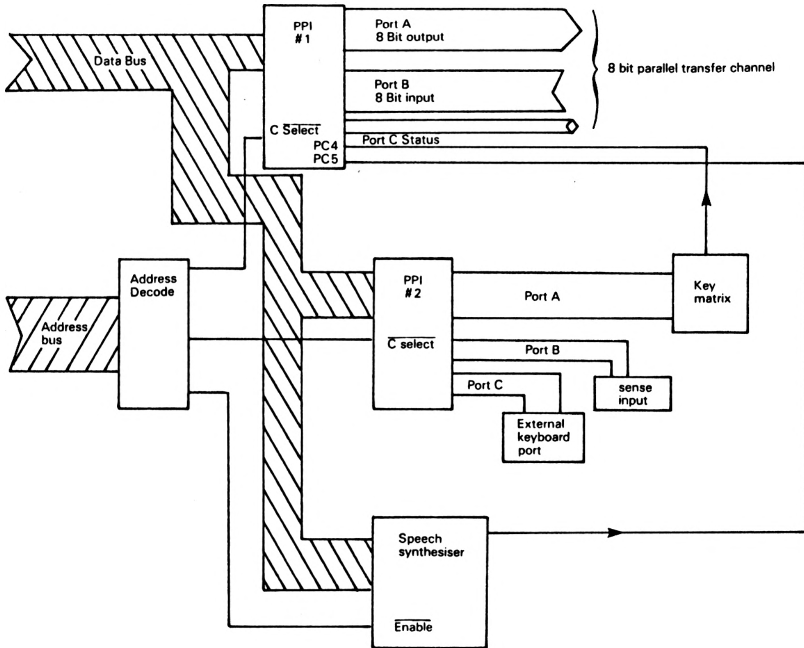


Fig. 3.13: Multiproject board block diagram.

The first technique, and the most straightforward, is to connect a microphone to an analogue to digital converter circuit. The resulting output from this A/D converter is stored in a large amount of memory. When the stored speech is to be reproduced, the digital representation is output to a Digital to Analogue (D/A) converter, at the same rate it was "recorded". This turns the digital data back into speech. In small computers this technique is hardly ever used, due to the fact that one second of speech, stored in this way, requires about 12,000 bytes of memory storage. This would consume the 64K of memory in the CPC 464, in a little over five seconds! Another way to store speech is to break it down into its smallest components. The smallest component of speech is known as an allophone. You can then store the information required to reproduce these allophones in an integrated circuit. This method is far more practical for small systems. All the microprocessor now has to do, is feed the appropriate commands into the speech chip, to make it reproduce a string of speech components, which will form the word or words desired. A third variant, which is quite similar to the previous one, is to store the information for a limited vocabulary in a chip. This is less flexible than the previous method, but the quality of the stored speech is usually slightly better.

Obviously, the speech synthesiser chip you choose should fit your application. In cases where there are only a certain number of words to be

spoken, the extra clarity of the fixed vocabulary can be chosen. If you want to have a speech synthesiser which can speak words which are not available in a ready made vocabulary, then you must use the chip which reproduces each individual allophone under control of the microprocessor. This is the method which is used in the speech synthesiser, about to be presented.

The speech synthesiser chip used in this design is the General Instruments SPO 256. This can reproduce a useful range of allophones, and has the great virtue of simplicity. Programming the SPO 256 is analogous to sending data to a USART. The SPO 256 takes in a 6 bit code, representing the required allophone, on the negative going edge of its ALD bar signal. The allophone is then reproduced, and when the chip is ready to take another command, its LRQ and SBY lines go true. The pinout for the SPO−256 AL is reproduced in appendix three. The block diagram for the circuit we are about to look at is shown in figure 3.14.

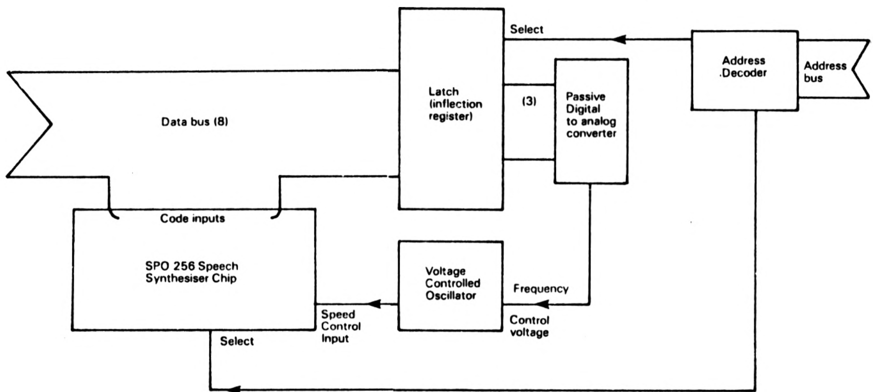


Fig. 3.14: Block diagram of speech synthesiser.

Figure 3.14 shows that the speech synthesiser has been fitted with an inflection register. This allows the software to program the synthesiser to use one of seven tones of voice when speaking. Needless to say this greatly adds to the natural quality of the speech, and to some extent disposes of the monotone for which computer speech has always been derided. The inflection register is possible because the tone of the speech reproduced by the SPO 256 is dependent upon the frequency presented to it at its OSC1 input. (You can also use the SPO 256 with a 3.12 Mhz crystal – see data sheet in appendix three.) In this design the output of the inflection register– which is really a latch feeding into a passive digital to analogue convertor – is fed into the frequency control input of a voltage controlled oscillator chip. The frequency at the output of this chip is adjusted by varying the voltage at its frequency control input. No separate amplifier has been added to this design.

This is partly in the interests of economy and simplicity, but also because the speech quality obtained by feeding the speech signal back into the CPC 464 internal amplifier was very acceptable for so simple a unit.

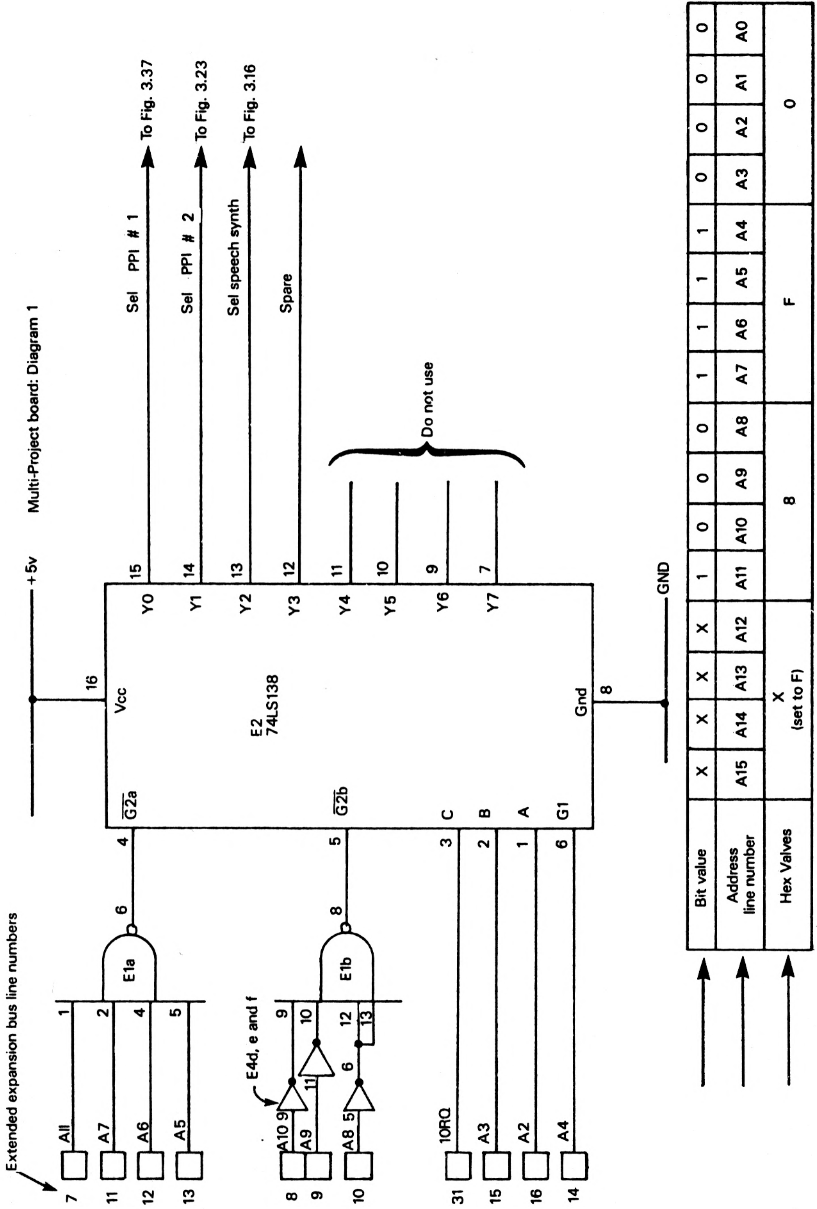


Fig. 3.15: Circuit diagram of multiproject board address decoder.

Figure 3.15 is the circuit diagram for the address decode of the multiproject board. This is mainly self explanatory, and places the speech synthesiser register at I/O address Hex F8F8. When the address decoder – set up as discussed in chapter 1 – detects that an address in the I/O page which lies between Hex F8F8 and F8FB is on the address bus, the Y2 output of chip E2 (marked SEL SPEECH SYNTH on the diagram) will go low. Then moving onwards to the actual synthesiser section of the project, shown in fig. 3.16, the Y2 signal is passed through the conditioning logic comprised of gates E8a–E8d and inverters E4b and E4c. This conditioning logic ensures that the CLK line into the inflection register latch, E10, is only taken high when I/O address Hex F8F9 is present on the address bus. Further the ALD bar input of the SPO 256 chip is only pulsed low during the time that a write to I/O address Hex F8F8 is occurring. (Address load is an input which latches the data bus contents into the SPO 256 when it is pulsed low.) All this means that once you have constructed and connected the unit the command:

```
OUT &F8F8,15
```

will cause the speech synthesiser to spring to life. This is the allophone whose code is 15, which is the AX sound, to be used as part of a word like sUCCeed. The commands:

```
FOR I=0 TO 7: OUT &F8F9: FOR P=1 TO 1000: NEXT: NEXT
```

will cause the noise to start low and rise through eight different pitches. This brings out a very important point to always remember about the speech chip. Unless you tell it to be quiet, it will go on making the sound of the allophone which you last commanded it to make. Fortunately the allophone codes 0–4 are defined as silences of various lengths, so before it drives you mad type the command:

```
OUT &F8F8,0
```

This will shut Arnold up! So the address decoder circuit places the speech chip register at I/O address Hex F8F8. However, this register is write only, that is to say that although you can place values into it, reading it with a command like:

```
PRINT INP(&F8F8)
```

will always print "255". This is because the speech chip does not respond when the processor reads I/O address F8F8, and the effect is the same as reading from an empty address in the I/O space. So how do we read the status of the speech chip? There are only two status signals of concern to us for this application, these are called LREQ bar and SBY. Either of these signals could be used to allow the processor to know when the speech chip is ready for another allophone code. The LREQ bar line is an output which goes low when the chip is ready for another code, whilst the SBY output goes high to indicate the same thing. LREQ bar has been used in this design, and it is fed into one of the two PPI chips – E3 pin 12. Chip E3 is labelled within this

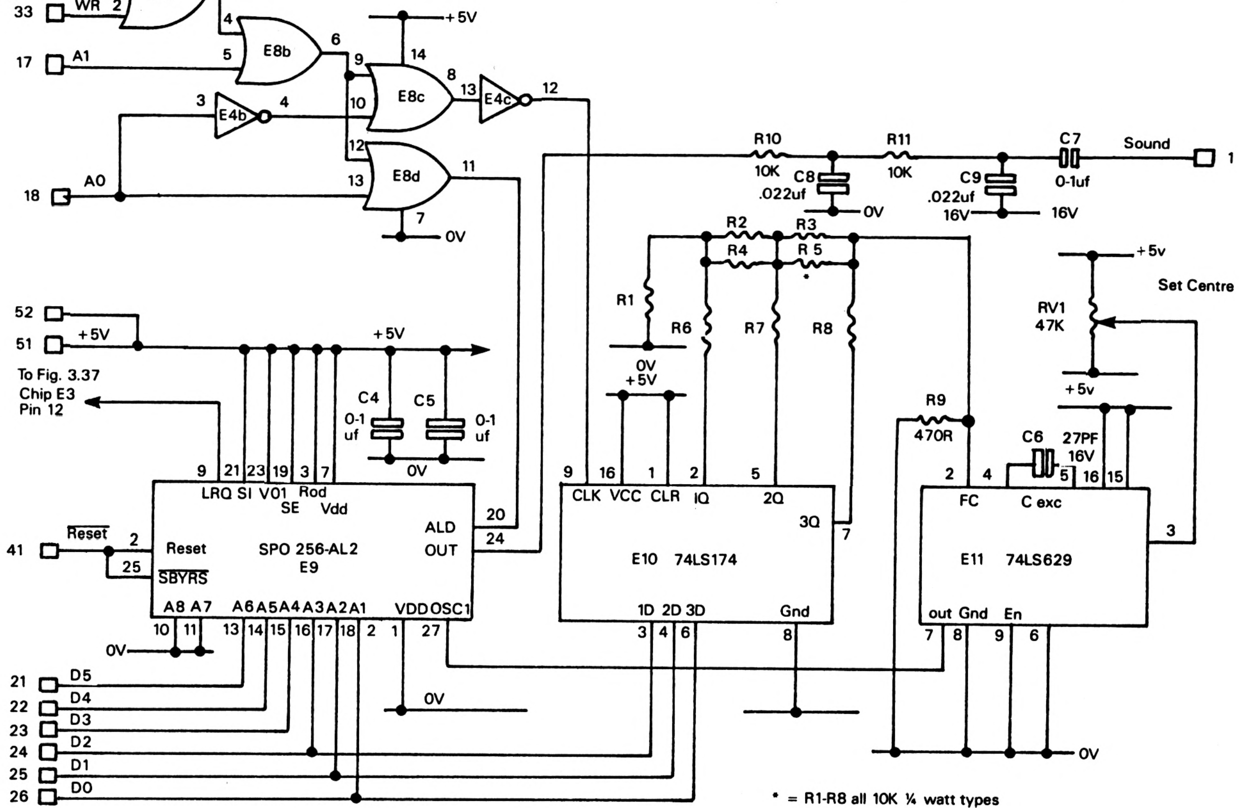


Fig. 3.16: Speech synthesiser circuit diagram.

design as PPI number one. (abbreviated on the diagrams to PPI #1). Without going into the operation of the PPI chips in this design (which will be happening in the description of the next project) the PPI is set so that its pin 12 is an input to bit 5 of port C on PPI one. Software merely has to read port C of PPI one, AND out bit 5, and if it is low then the speech chip is ready for another allophone code. We will soon be looking at a practical program for driving the speech synthesiser, together with a list of allophone codes.

The speech synthesiser chip is clocked not by a crystal oscillator circuit, but by a 74LS629 Voltage Controlled Oscillator (VCO) chip. The potentiometer RV1 sets the centre frequency of the oscillator, and is best set midway when setting up the circuit. The timing capacitor C6 is the other factor in the frequency determination. The voltage applied to the Frequency Control (FC) input to the 74LS629 increases or decreases the oscillator frequency. A low voltage slows it down, whilst a higher voltage will allow it to run faster. In this design the software places a value from 0 to 7 into the inflection register formed by E10, this is converted to one of eight possible voltage levels by the R2R network made up of R1–R8. The selected voltage level is applied to the FC input of the VCO, altering the frequency of its output. Because the speech chip depends entirely on the VCO output for all its timings, altering the VCO output frequency alters the speed and pitch at which the SPO 256 speaks.

The final part of the circuit consists of the small C/R filter made up from R10,R11,C8, and C9. This makes the speech output more suitable for input to the CPC 464 internal amplifier. The audio speech signal is fed out onto the expansion bus SOUND bar input which goes straight into the cassette amplifier. On a CPC 464 it is possible to use the datacorder for saving or loading programs whilst the speech synthesiser is actually speaking – so there would seem to be no interaction between the cassette and the synthesiser.

A small amount of setting up will be needed in order to ensure the correct operation of the speech circuit. This mainly consists of setting RV1 to a point where the SPO 256 seems happy with the inflection register loaded with either zero or seven. If the speech synthesiser crashes, due to being clocked too quickly or too slowly it emits a mind numbing din, which sounds a bit like Niagara falls in full spate. The only way to stop it is to power everything off and start again. The setting up is largely a matter of trial and error if you do not have an oscilloscope available, but it usually only takes a couple of minutes. If you have the use of a fast 'scope, you can easily set the circuit up. Put a value of 3 or 4 into the inflection register and monitor pin 27 of the speech chip. Adjust RV1 so that you see a frequency of about 3 megahertz. Then the set up is complete.

Construction notes
Beyond the normal safeguards against static which you should take when handling MOS integrated circuits, such as the SPO 256, there should be no difficulty in assembling the speech synthesiser– provided that you are using the PCB. (see notes at the beginning of this book for PCB details.)

Dec	Hex	Phonetic	Sample	Dec	Hex	Phonetic	Sample
00	00	Silence	10 mils	32	20	aw	Bout
01	01	Silence	30 mils	33	21	dd2	Doom
02	02	Silence	50 mils	34	22	gg3	Rig
03	03	Silence	100 ms	35	23	vv	Void
04	04	Silence	200 ms	36	24	gg1	Gone
05	05	Oy	Toy	37	25	sh	Shine
06	06	i	Fry	38	26	zh	Azure
07	07	eh	Send	39	27	rr2	Bronze
08	08	KK3	Coast	40	28	ff	Fail
09	09	pp	Pat	41	29	kk2	Scott
10	0A	jh	Lodge	42	2A	kk1	Cost
11	0B	nn1	Win	43	2B	zz	Zulu
12	0C	ih	Wit	44	2C	ng	Anger
13	0D	tt2	Trill	45	2D	ll	Lazy
14	0E	rr1	Roast	46	2E	ww	West
15	0F	ax	Suck	47	2F	xr	Prepare
16	10	mm	Much	48	30	wh	Where
17	11	tt1	Post	49	31	yy1	Yes
18	12	dh1	They	50	32	ch	Chain
19	13	iy	Tree	51	33	er1	Better
20	14	ey	Rage	52	34	er2	Bird
21	15	dd1	Good	53	35	ow	Show
22	16	uw1	Stew	54	36	dh2	Thought
23	17	ao	Bought	55	37	ss	Roast
24	18	aa	Hot	56	38	nn2	No
25	19	yy2	Yawn	57	39	hh2	How
26	1A	ae	Platt	58	3A	or	Bore
27	1B	hh1	Hurl	59	3B	ar	Arm
28	1C	bb1	Rib	60	3C	yr	Beer
29	1D	th	Thirst	61	3D	gg2	Gate
30	1E	uh	Crook	62	3E	el	Meddle
31	1F	uw2	Brood	63	3F	bb2	Boil

Fig. 3.17: List of Allophone codes for the SPO 256 chip.

Figure 3.17 is a list of the available allophones which can be reproduced by the speech chip, and the decimal or hexadecimal codes you have to load in to make it reproduce each one. If when you have built this circuit you want to make it even more comprehensive, you could fairly easily construct your own board for a more sophisticated unit which would include the extra ROM which is available from GI. This extra ROM greatly extends the number of allophone codes available. See the reproduced data sheet in appendix three, which shows a sample circuit for an expanded speech synthesiser.

Software for the speech synthesiser

Referring once again to Fig 3.17 you can see that the codes available for allophones run from decimal 0 to 63 (Hex 0 to 3F). This means that in this design the top two bits of the bytes sent to the synthesiser are not used. Bear this in mind for the moment.

In its most simple form a program to make the speech synthesiser speak must perform the following steps:

1) Initialise the PPI chip via which the status of the speech synthesiser is read back.

2) Set up the inflection register with a required pitch code.

3) Wait until the speech chips LREQ bar line is low.

4) Send the code for the next allophone required to be spoken.

5) Check to see if the code just sent was the last. Go back to step 3 if not; otherwise end.

This is shown in flowchart form in Fig 3.18 and the BASIC program shown as figure 3.19 is an implementation of the flowchart. It includes some DATA statements to make the circuit say – "I am CPC 464, but you can call me Arnold". Note the monotonous tone of the speech.

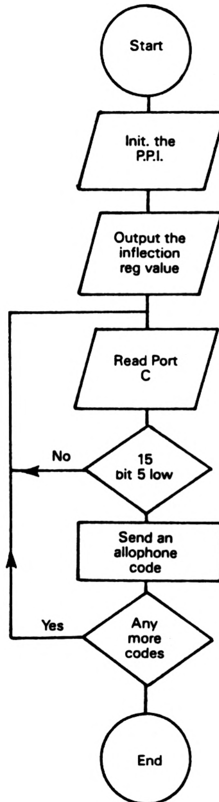


Fig. 3.18: Flowchart for speech synthesiser software.

```
100     ** Speech synthesiser demo program - manual inflection select. **
200     OUT &F8F8,0 ' Shut up Arnold....
300     RESTORE: MODE 2: ON BREAK GOSUB 1400:
        PRINT "Speech synth demo program with manual inflection select.";
        STRING$(4,10): INPUT "Select inflection pitch (0 to 7)"; SPR
400     IF SPR < 0 OR SPR > 7 THEN 300 ELSE OUT &F8F9,SPR 'Set up the
        inflection register.
500     OUT &F8F3,&AF ' Set up PPI to let us look at busy bit of
        speech chip on PC5.
600     READ D: IF D > 63 THEN 1300 ELSE WHILE (INP(&F8F2) AND 32) <> 0:
        WEND: OUT &F8F8,D: GOTO 600
700     DATA 4,4,6,4,26,16,3,55,19,3,9,19,3,55,19,3,40,58,4,55,55,
        12,12,41,55,3,40,58,4,4,4
800     DATA 4,4,28,15,17,3,25,53,4,42,26,11,4,42,23,45,3,16,19,4
900     DATA 59,56,23,45,33,4,4,4,4,4,4
1000    DATA 4,4,4,4,6,3,46,24,11,13,3,49,31,3,23,45,3,13,31,3,56,
        53,3,3,4,29,26,17,3,4,6,3,26,16,4,40,19,45,12,11,61,4,35,7,
        47,19,3,33,12,9,14,7,55,55,13,4,4,4,4,4,4,4
1100    DATA 4,4,4,4,4,4,4,53,4,4,46,15,11,4,4,13,31,4,29,14,19,4,
        40,58,4,40,6,35,4,55,55,12,12,41,55,4,55,55,7,39,15,11,4,20,
        2,17,4,56,6,12,4,13,7,11,4,4
1200    DATA 4,4,27,26,45,53,46,4,4,4,4,4,4,4,27,26,45,53,46,4,4,4,4,4,
        4,4,64
1300    MODE 1: LOCATE 10,10: PRINT "Arnold has spoken!": T=TIME:
        WHILE TIME < T+400:  WEND: CLS: LOCATE 10,10:
        PRINT "Hush! he speaks again....": RESTORE: GOTO 600
1400    OUT &F8F8,0 'Shut Arnold up to avoid insanity. Next we END
1500    MODE 2: : PRINT "The mighty one is silenced.....";STRING$(4,10):
        END
```

Fig. 3.19: BASIC program to drive speech synthesiser.

Remember those two unused bits? (Yes you do, wake up at the back there!), you know, the ones which are always unused in the allophone codes. We can put those to good use in a more sophisticated program which will interpret each item in a list of DATA statements as one of four things:

1) An allophone code

2) A value to be loaded into the inflection register.

3) A value to be used in a delay loop, to allow the current allophone to continue being spoken as long as the loop lasts

4) A value which means that the end of the allophone codes list has been reached.

So we can use those top two unused bits as instructions to the program that does the READ of the items in the DATA statements. (NOTE: If you intend ever to extend the allophone set, by use of the add on ROM which was previously mentioned, all 8 bits of the allophone value are used.) We can draw a small table to show what each possible combination which can exist on those top two bits will be taken to mean.

Bit 7	*Bit 6*	*Meaning*
0	*0*	*Bottom 6 bits are an allophone code.*
0	*1*	*End of list of allophones.*
1	*0*	*Bottom 3 bits are a value for inflection reg.*
1	*1*	*Prolong last allophone. Bits 0–6 are multiplied by 5, and the result is used as a delay counter. During the delay the current allophone continues to sound.*

```
50 '    **          Speech synthesiser demo program - Auto inflection version **

100     RESTORE: MODE 2: ON BREAK GOSUB 1800:
        PRINT "Speech synth demo program auto-inflection version";
        STRING$(4,10)
200     OUT &F8F8,0 ' Shut up Arnold....
300     OUT &F8F3,&AF ' Set up PPI to let us look at busy bit of
        speech chip on PC5.
400     READ D: CTRL=((D AND &C0)/64): IF CTRL = 0 THEN
        GOTO 900
500     '
        **          Routine to handle speech codes where the top two bits   **
        **        of the speech code are NOT zero. These are held in CTRL   **
600 '   **        and have the following usages.  00 = Illegal: 01 = finish**
        **        02 = AND code with 07 and send to inflection register:
        **        03 = Prolong last allophone. Bits 0-6 of D*2 = loop value**
700     IF CTRL = 1 THEN GOTO 1700 ELSE IF CTRL=2 THEN
         OUT &F8F9,(D AND 7) ELSE IF CTRL = 3 THEN FOR I=0 TO
         ((D AND 63)*5): NEXT I
800     GOTO 400
900     IF (INP(&F8F2) AND 32) <> 0  THEN 900
1000    OUT &F8F8,D: GOTO 400
1100    DATA 4,4,133,6,4,132,26,16,3,55,19,3,9,19,3,55,19,3,40,
        58,4,55,55,12,12,41,55,3,40,58,4,4,4
1200    DATA 4,4,28,15,&F8,17,3,25,53,&F8,4,42,26,11,&F8,4,42,23,
        &FF,45,&CF,3,16,19,4
1300    DATA 133,59,&FC,56,23,45,33,4,4,4,4,4,4
1400    DATA 4,4,4,4,131,6,3,46,24,11,13,3,49,31,3,23,&F0,45,3,13,31,3,130,56,
        53,255,3,3,4,29,26,17,3,4,6,3,26,16,4,40,19,45,12,11,61,4,129,35,7,47,
        19,3,33,12,128,9,&F0,14,7,55,55,13,4,4,4,4,4,4,4,4
1500    DATA 4,4,4,4,4,4,4,4,&82,53,4,4,46,15,&C8,11,&C8,4,4,13,31,4,29,14,
        19,&FF,4,40,58,4,40,6,35,4,55,55,12,12,41,55,4,55,55,7,39,15,11,4,
        20,2,17,4,56,6,12,4,13,7,11,4,4
1600    DATA 4,4,&87,27,26,&EF,&82,45,53,&FF,&F0,46,4,4,4,4,4,4,4,4,
        &80,27,26,&EF,&85,45,53,&FF,&F0,46,4,4,4,4,4,4,4,4,64
1700    MODE 1: LOCATE 10,10: PRINT "Arnold has spoken!": T=TIME:
        WHILE TIME < T+400:  WEND: CLS: LOCATE 10,10:
        PRINT "Hush! he speaks again....": RESTORE: GOTO 400
1800    OUT &F8F8,0 'Shut Arnold up to avoid insanity. Next we END
1900    MODE 2: : PRINT "The mighty one is silenced.....";STRING$(4,10):
        END
```

Fig. 3.20: Improved BASIC program to drive speech synthesiser.

Fig. 3.20 is a program which uses this scheme, it includes the codes to make the synthesiser speak with a little more feeling, the first few DATA statements are the same as before, but more has been added to give you a better demonstration of the inflection register action.

The program is fairly self explanatory, but a few points may need clarification. At line 300 the PPI mode is set to Hex AF. This sets the PPI into a mode where the port C line PC5 can be used as an input. As you will find from fig 3.14 and 3.15 the LREQ bar line is made available for interrogation via PC5. The CTRL variable is derived from the R variable, which contains the most recently read value from the DATA list. After execution of line 400 the CTRL variable contains the top two bits of the contents of R, but these have been shifted to become the lowest two bits in the byte. This is done by ANDing out the top two bits of R and dividing the result by 64. As in:

```
X=&40: X=(X/64): PRINT X
```

Which will first set up X to be = Hex 40 (Binary 01000000). Then X is divided by 64, which has the effect of shifting all the bits down by 6 places – with zeroes being shifted in from the left and, as the shift takes place, the rightmost bit value drops off the end and disappears! Applying this to the binary pattern 01000000 in successive stages we get:

Shift #1: 00100000
Shift #2: 00010000
Shift #3: 00001000
Shift #4: 00000100
Shift #5: 00000010
Shift #6: 00000001 =1; So that X is now = Hex 01.

E number	Device description			Power connection pins			
	Type	Equivs	Function	+5V	+12V	-12V	0V
1	74LS20	-	Dual 4 I/P NAND	14			7
2	74LS138	-	3/8 line decode	16			8
3	8255A	-	PPI chip	26			7
4	74LS04	-	Hex inverters	14			7
5							
6							
7							
8	74LS32	-	QUAD 2 I/P OR	14			7
9	SPO-256	-	Speech synth	7			1
10	74LS174	-	Hex data latch	16			8
11	74LS629	-	V.C.O chip	16			8

Fig. 3.21: Voltage supply pins for chips used in speech synthesiser.

After line 400 has extracted the top two bits and adjusted them into a more usable form, in the manner just described, the result is checked to see if it is zero. If it is then this means that the bottom 6 bits of the R variable contain an allophone code, so a GOTO 900 takes place. If the result is non zero, then lines 700 and 800 implement the required action according to the previously described scheme. BREAKs are trapped so that the speech synthesiser can be silenced before the program is ended. In my example DATA statements I have used the decimal values for the allophones, except where the code is not an allophone, where I have used the Hex notation for it. I find that this has two advantages: it makes control codes easier to locate in the list, and it makes their value more easy to calculate.

```
Capacitors

C1        50 mfd 16 Volt electrolytic          (RS 103-979)
C4-C5     0.1 mfd disc ceramic 16 Volt         (RS 124-178) (2 required)
C6        27 pf polystyrene bead 16 Volt       (RS 114-654)
C7        0.1 mfd any type 16 Volt             (eg RS 115-102)
C8 & C9   0.22 mfd 16 Volt                     (RS 115-118) (2 required)

Resistors

R1-R8     10K Ohm quarter Watt                 (RS 131-378) (8 required)
R9        470 Ohm quarter Watt                 (RS 131-211)
R10 & 11  10K Ohm quarter Watt                 (RS 131-378) (2 required)
RV1       47K pcb mounting potentiometer       (RS 186-889)

Semiconductors

E1        TTL 74LS20 Dual 4 input NAND gates   (RS 307-553)
E2        TTL 74LS138 3 to 8 line decoder      (RS 307-648)
E3        8255A PPI chip                       (RS 309-363)
E4        TTL 74LS04 Hex inverters             (RS 307-503)
E8        74LS32  Quad 2 i/p OR gates          (RS 307-569)
E9        SPO-256                                   -
E10       74LS174 Hex data latches             (RS 307-682)
E11       74LS629 Voltage controlled oscillator     -

Miscellaneous

Sockets for all chips
Expansion  bus connector                                 (RS 468-119)
Printed circuit board - see note 2 below


      -----------------------------------------------------

NOTES:
        1) Note that the RS part numbers  quoted  may  be  supplied  in
           packs containing more than one part.

        2) Complete kits of parts are to  be  available  from  Halstead
           designs LTD.
```

Fig. 3.22: Parts list for speech synthesiser.

This concludes the description of the speech synthesiser, figure 3.21 shows the power connection pins for all the chips used. Fig 3.22 is the parts list for the project.

Project 3: Adding an external keyboard

If you scan the pages of your favourite electronics magazine you will nearly always find at least one advertisement for computer keyboards. This is partly due to computer companies selling off their spares holdings, and partly due to the rate at which new units are being introduced. If you know someone who works in the computer industry, they may well be able to get you a keyboard from a scrap bin or a computer salvage company.

Why would you want a second QWERTY keyboard? There are lots of applications which spring to mind. There are certain kinds of computer games where several players can take part, and both can see the screen but neither must see what the other one types on the keyboard. This would be secret codes, passwords, co–ordinates in a battleships type of game etc.

Another intriguing possibility, given that the CPC 464 screen is very easy to partition into 2 parts would be to use a split screen and two keyboards, with some software– to be written in BASIC– to effect a very cheap implementation of a two user machine. Granted there might be difficulties for some applications, but for text entry this would be pretty easy.

Another use for an add on keyboard port could be to add on a special keyboard for use by a disabled typist, either with super large key legends, or keyboards for use with some other part of the body than the hands. I have not developed such a unit, but it would be fairly easy to build a speech synthesiser into a stand alone unit containing the external keyboard, which is then connected to the CPC 464, this would be a useful aid to the blind, speaking the name of the key pressed.

These are just a few of the possible applications which spring to mind. I have no doubt that readers will be able to envisage many more.

Those nice cheap keyboards advertised in the electronics magazines (Down to about £15 or so) fall into two catagories. These are called encoded and unencoded. In this context encoded means that the keyboard unit has enough on–board electronics to present the code for any key pressed to the computer. An unencoded keyboard – like the one in the CPC 464 is scanned by software. This second method means that the computer must continuously interrogate every key on the keyboard – to see if it is pressed, and the process must go on the whole time that the machine is switched on. This is not as bad as it might seem however, because of the disimilarity of the speed of the computer and even the fastest typist.

There are many chips available which perform keyboard scanning as a hardware function. These chips are usually found inside a unit sold as a fully encoded keyboard. What they do is debounce a keypress and then send out the code – usually ASCII – for that key, to the computer. (see appendix 4 for a chart of ASCII codes.) The term debounce will now be explained.

Because electronic circuitry is so fast, it is sometimes difficult to interface switches and relays to it. Although you may think that a microswitch, or a relay has a snap action, a slow motion look at the contacts closing would show you that the contacts actually bounce as they meet. This means that they open and close several times, before they actually close for good. In applications like switching on a light this doesn't matter, because the bounce duration is only a millisecond or so. However when you realise that some logic chips can count up to 25 million pulses per second, you will see that the situation is somewhat more critical when the switch is connected to a computer. If the effect were not dealt with, you might press a keyboard key and have it register four or five times. How would that beeeee? To combat the effects of contact bounce many techniques have evolved. The hardware approach usually involves putting a time delay CR network on the logic input. This has the effect of ironing out the bounce. The software approach is to begin a short delay loop when the first contact is made, then wait for a sufficient time to elapse for the bouncing to stop. This is usually called a settle time, or debounce delay loop. The situation is not so bad when using keyboard switches, as they are specially designed for minimal contact bounce, though even with these switches the problem still exists.

When choosing a keyboard you need to find out three things.

1) Do you get a connectional diagram with it? If not don't buy it, as you will have to guess the connections – not easy.

2) Is it encoded –does it have an encoder chip on it? If it does, is it an ASCII encoder? If it is not then interfacing it to your CPC 464 will be more difficult – since you will have to write code conversion software.

3) What power supply voltages does it require? All will need +5V DC and 0V, but some also need –12 volts. If you have built the SIGMA power supply this will be no problem.

So to sum up, you need an ASCII encoded keyboard, with connectional details and power supply requirement details.

Fig 3.23 is the circuit diagram for three of the projects on the multi project board. These are the external keyboard port (currently being detailed) the key matrix port: and the sense register. The last two will be described in the next two sections.

In Fig 3.23 the central device is the second 8255 PPI chip. This is programmed so that Port A is an 8 bit output port, whilst ports B & C are 8 bit input ports.

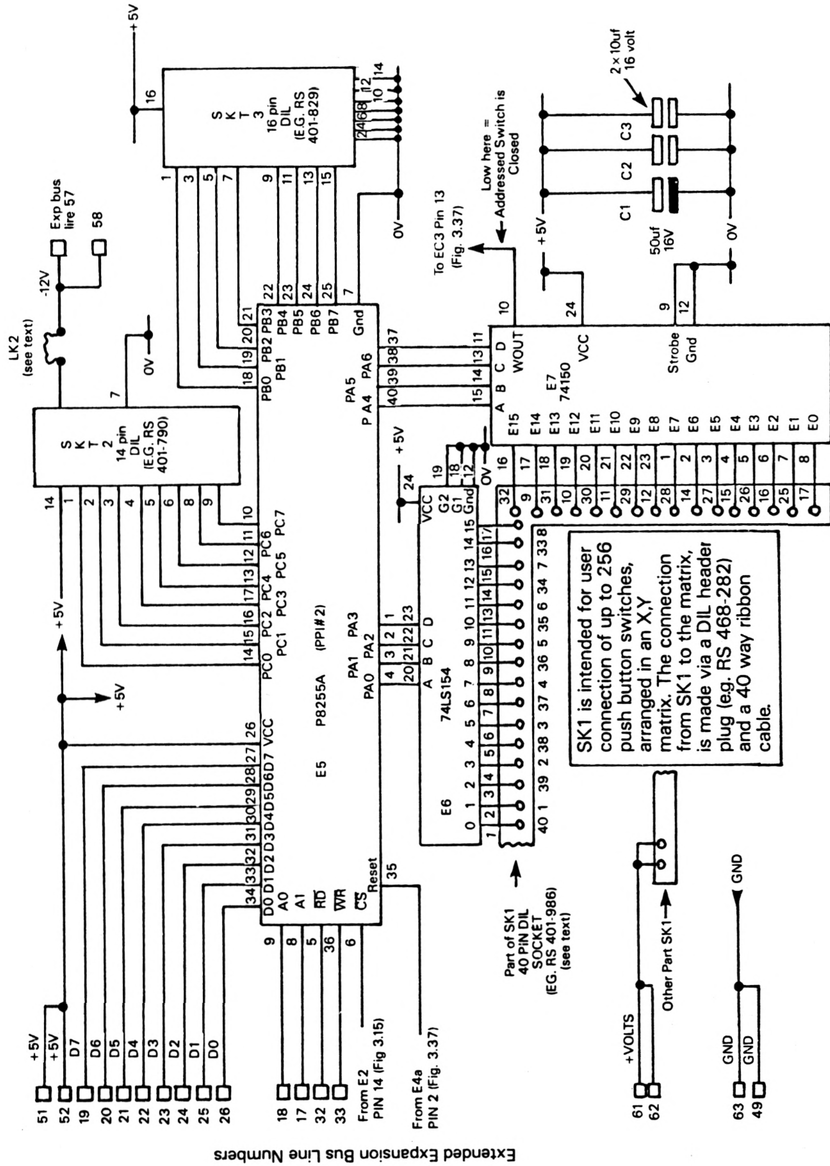


Fig. 3.23: Circuit diagram for second part of multiproject board.

Port B is used as the add–on keyboard port. We saw the address decoder for the multi project board in figure 3.15, referring to this briefly, you can see that the Y1 output of the decoder chip E2 will go low to select the second PPI. This places the PPI #2 registers at the following I/O addresses.

PPI #2 registers

Port A	Hex F8F4
Port B	Hex F8F5
Port C	Hex F8F6
CTRL register	Hex F8F7

This means, for example, that the commands:

```
PRINT INP(&F8F6): OUT &F8F6,10
```

will respectively read and write at port C. Port C is the port we are going to connect our add–on keyboard to.

You will find that 99% of the keyboards you can buy will have seven data lines and a strobe output. The seven bits of data contain the ASCII code for the key pressed, and the strobe line will be pulsed to indicate that a new code is available. If, as in the case of this design, these lines are all fed into an unlatched port the software must regularly check for an active strobe line from the keyboard.

The usual arrangement is to use bit 7 of the port for the strobe line and the lower order 7 bits as the ASCII code input. The BASIC program to scan the keyboard port then looks as shown in figure 3.24 The PPI is set to the desired mode of operation– in this case the control byte value is Hex 8B. Then port C is repeatedly scanned for a logic level change at bit 7 of the port. This approach allows both negative and positive going strobe signals to be accommodated – though most strobe lines from keyboards are active low. Finally, when the strobe is seen as active the value obtained by reading port C is read into the variable V, and the strobe bit is set to a zero. The value thus obtained is the ASCII code for the character typed at the external keyboard. The value is then printed to the CPC screen– note that the semicolon after the print statement suppresses a carriage return, which would ruin the character layout by inserting a carriage return and line feed pair after every character printed. Pressing any key on the CPC keyboard aborts the program.

Referring again to Figure 3.23: the external keyboard is connected onto the multi project board via SKT2 which is a standard 14 pin DIL chip socket. These make very good, and very cheap, connectors. They are usually only used where the connection is not unplugged very often, but where it must periodically be detached. REMEMBER to put a mark on both plug and socket so that accidentally plugging in the wrong way round cannot occur. To connect your external keyboard you buy the appropriate length of ribbon cable (about 3 metres maximum) and connect it into a DIL header plug – see

parts list. Then connect the appropriate conductors in the ribbon cable to the keyboard. No details about the method of connection at the other end of the ribbon cable are provided, since this will obviously depend entirely on what kind of connector the keyboard comes with. The DIL connector SKT2 pin usages are as follows:

```
10    '** Small BASIC program to echo all characters typed on    **
       ** an add on keyboard unit.

20     OUT &F8F7,&9B ' ** Initialise the PPI to what we need
30     STROBE =128 '** this is for high pulsing strobes, for low pulsing
       ** strobes set STROBE to zero.
40     WHILE (INP(&F8F6) AND 128) <> STROBE: WEND '** Wait till the
       strobe condition occurs.
50     PRINT CHR$(INP(&F8F6) AND 127); : GOTO 40
32767 END
```

Fig. 3.24: Add on keyboard echo program in BASIC.

SKT 2 Pin number allocations

Pin	*Usage*	*Pin*	*Usage*
1	Data bit 0	8	Data bit 6
2	Data bit 1	9	Strobe from keyboard
3	Data bit 2	10	unused
4	Data bit 3	11	unused
5	Data bit 4	12	− 12 volts supply to KB.
6	Data bit 5	13	unused
7	0 volts (Gnd)	14	+5 volts supply to KB.

The link shown on the diagram − LK2 is for you to extend, or not extend, the −12 volt supply to the keyboard. Many keyboard units do not require any other than just +5 volts and ground. If the one you get is one such then leave LK2 out. This will completely preclude the possibility of any damage occurring to the keyboard electronics if the −12 volt line is connected wrongly.

Constructional notes

Handle all the chips with care. As previously mentioned the exact method of making the ten or eleven connections from the port to the keyboard will depend entirely upon what kind of keyboard unit you get. So long as you get the data about the keyboard at the time of purchase, there should be no difficulty in interfacing the port to the keyboard. Fig 3.25 shows the power pins for the devices used. Fig 3.26 is the parts list for the keyboard port.

E number	Device		description	Power connection pins			
	Type	Equivs	Function	+5V	+12V	-12V	0V
1	74LS20	--	Dual 4 i/p NAND	14			7
2	74LS138	--	3/8 Line decode	16			8
3							
4	74LS04	--	Hex inverter	14			7
5	8255A	--	PPI chip	26			7
6							

Fig. 3.25: Voltage supply pins for add on keyboard project.

This is a list of the parts which you will require to build only the external keyboard port section of the Multiproject board

Capacitors

C1	50 mfd 16 Volt electrolytic	(RS 103-979)
C1-C2	0.1 mfd disc ceramic 16 Volt	(RS 124-178) (2 required)

Semiconductors

E1	TTL 74LS20 Dual 4 input NAND gates	(RS 307-553)
E2	TTL 74LS138 3 to 8 line decoder	(RS 307-648)
E4	TTL 74LS04 Hex inverters	(RS 307-503)
E5	8255A PPI chip	(RS 309-363)

Miscellaneous

Sockets for all chips
SK2 14 pin DIL socket (RS 401-790)
Expansion bus connector (RS 468-119)
Printed circuit board - see note 2 below

NOTES:

1) Note that the RS part numbers quoted may be supplied in packs containing more than one part.

2) Complete kits of parts are to be available from Halstead designs LTD.

Fig. 3.26: Parts list for add on keyboard project.

Project 4: Key matrix port

This project is a very flexible one to allow the connection of a great many switches to the computer without a great deal of electronics to interface them. This project will find many applications in computerised systems which have a requirement for scanning a large number of switches. Examples of this are customised keyboards, unencoded ASCII keyboards, burglar alarms – where switches are usually fitted to windows and doors, and also various detectors must be connected to the computer. Another application which appeals to me greatly, but which I have never had the time, money, or opportunity to try, is to buy an old pinball machine, clear out all the electro–mechanical parts and completely refurbish it, using an electronic system whose central element would be a microprocessor. In such a project I would scan the ball detect switches with a circuit like this. Another application is for games, and indeed a small program called "Minefield" will be listed for use with this project. Another application close to my heart is to scan various train detect, and point state switches on a large model railway. Suffice it to say that if you have a requirement to repeatedly scan a large number of simple two contact switches, you can do it with this circuit.

Residing as it does on the multi project board, the Key Matrix Port (KMP) uses the address decoder in an identical way to the keyboard port in the last project. When you access I/O address Hex F8F4 you are reading or writing at port A on PPI #2. The KMP lives on port A which is set up as an 8 bit output port. Referring again to figure 3.23 we see that port A has two chips connected to it. These are E6 and E7. Chip E6 is a 4 to 16 line decoder, to use this you feed a four bit binary code into its A,B,C and D inputs, then the output number matching the binary code goes low. Chip E7 is a 16 channel data selector. It has 16 inputs – E0 to E15, and an output called WOUT. This chip also has A, B,C and D inputs, but in this case they select which of the E inputs is routed to the WOUT pin.

The ABCD inputs to chip E6 are fed with binary codes from the lowest four bits of port A, whilst the ABCD inputs of E7 are fed from the upper four bits of port A. This means that if we initiate a FOR NEXT loop from 0 to 255 and as part of each loop we output the loop value to port A, we will cause each of the E6 outputs to go low in turn, and every time the bottom four bits go back to zero we will have added one to the count value on the top four bits, therefore updating which of the E7 inputs is routed to the WOUT pin of E7. (Note the WOUT pin always contains the inverse logic level of the selected input.) If we connect, via the extending socket SK1, the required number of switches with one terminal connected to an output from E6, and the other terminal connected to an input of E6, and then press that switch, we should be able to detect the fact by stopping the scan when we see WOUT go to a low level. How do we see WOUT? It is connected to bit 4 of port C on PPI #1. This is at I/O address Hex F8F2. For example, if we wanted to read the state of a switch which had one terminal connected to to Y row 8 (Pin 9 of E6), and the other connected to X row 3 (pin 5 of E7), we would use the following commands to initialise the PPIs' and read the state of the addressed switch:

```
OUT &F8F3,&AE: OUT &F8F7,&8B: OUT &F8F4,&83: STATE=INP(&F8F2):
IF (STATE AND &10)=0 THEN PRINT "Not pressed" ELSE PRINT "Pressed"
```

The 40 pin DIL socket SK1 is provided for connection to an external key matrix unit, or a distribution point if the switches are not local. One important thing to note is that for some applications it might be a good idea to fit some pull–up resistors – value about 10Kohms– from the X lines to the +5 volt supply. This will ensure that for long runs of cable to remote switches any noise picked up will not be a problem. This is the reason for extending out the +5 volts line to the distribution. The pin connections for SK1 are as follows:

Pin 1	Y1	Pin 21	+5 volts
Pin 2	Y3	Pin 22	––
Pin 3	Y5	Pin 23	––
Pin 4	Y7	Pin 24	––
Pin 5	Y9	Pin 25	X1
Pin 6	Y11	Pin 26	X3
Pin 7	Y13	Pin 27	X5
Pin 8	Y15	Pin 28	X7
Pin 9	X14	Pin 29	X9
Pin 10	X12	Pin 30	X11
Pin 11	X10	Pin 31	X13
Pin 12	X8	Pin 32	X15
Pin 14	X6	Pin 33	Y14
Pin 15	X4	Pin 34	Y12
Pin 16	X2	Pin 35	Y10
Pin 17	X0	Pin 36	Y8
Pin 18	––	Pin 37	Y6
Pin 19	––	Pin 38	Y4
Pin 20	+5 Volts	Pin 39	Y2
		Pin 40	Y0

As this project, and all the multi project board designs use at least one of the PPI chips, this is perhaps a good place to give you a means of testing the PPI to CPU interface.

Fig 3.27 is a program which will test a PPI chip to a limited extent. It will ensure that the PPI is capable of taking and retaining any value when all three of its ports are set up as outputs. The program prints out the details of any failures. This allows you to spot spurious bit dropping or raising, or other problems which may occur due to bad solder connections etc. Please read the REMs of the program, as some of them make important points.

Fig 3.28 is a program to use when testing, or familiarising, yourself with the KMP. It draws each of the possible 256 keys as a small square on the screen. When a key is pressed the square is coloured in, and the most recently pressed key's square slowly flashes. The program also sounds a high pitched

bleep for the time it sees a key pressed, and a low pitched boop for every time it has scanned all the keys and found none pressed. You will notice that the program, because it is written in BASIC, is sometimes rather sluggish to respond. This can be rectified by using a BASIC section to do the screen layout and a machine code subroutine to do the actual scan function. This solution has been adopted for the Minefield game to be presented shortly.

```
10 '        Peripheral Interface Adaptor (PPI) Test program. Written
         to test a PPI whose first register address is specified as the
         PPIBASE variable at line 30.
11 '     Tests that any value can be output to any of the three PPI ports.
         ** NOTE: The program sets all three ports to the OUTPUT. mode. **

12 ' VERSION 2 ·
20 MODE 2              'SET UP ME FAVOURITE SCREEN MODE.
30 PPIBASE=&F8F0       'SET UP THE PPI BASE ADDRESS HERE.
40                     'PROGRAM NOW TAKES PPIBASE AS THE PORT A ADDRESS, AND
                        PPIBASE+1 AS B PORT, PPIBASE+2 AS C PORT ADDRESS, AND
                        PPIBASE+3 AS CONTROL REGISTER ADDRESS

45 OUT PPIBASE+3,&80: PRINT "All ports have been set up as outputs": PRINT
50 PRINT "WHICH TEST ? (S)OAK TEST   OR (I)NDIVIDUAL TEST ";
60 T$=INKEY$: IF T$="" THEN 60 ELSE PRINT : IF UPPER$(T$)="S" THEN 100
70 INPUT "WHICH PORT TO SEND VALUES TO ",P$: P$=UPPER$(P$):
   IF P$="A" THEN OA=PPIBASE ELSE IF P$="B" THEN OA=PPIBASE+1 ELSE IF P$="C"
   THEN OA=PPIBASE+2 ELSE PRINT: PRINT "You must say one of A,B, or C" :
   GOTO 70
80 INPUT "Decimal value to place in port ";VALUE: IF VALUE > 32767 THEN 80
   ELSE OUT OA,(VALUE AND 255): PRINT: PRINT "PORT:","A","B","C":PRINT:
   PRINT "CONTENT:";,
90 FOR I=PPIBASE TO (PPIBASE+2): X=INP(I): PRINT CHR$(8);X;
   "(HEX=";HEX$(X);")",;: NEXT I: PRINT STRING$(50,95): PRINT:  GOTO 70
100 ' This section tests the selected PPI ports as outputs. It does
.     this by writing a count pattern into them. This tests the PPI
      logic, but a 'scope, DVM, or multimeter must be used to ensure
110   'that the outputs actually show the changes. This would not be the
       case when an output transistor (Inside the PPI) was defective.
120  CLS: LOCATE 25,1: PRINT "Soak test commencing"
130  LOCATE 8,5:  PRINT "Test value =": FOR I=0 TO 255: FOR X=0 TO 2 :
     OUT PPIBASE+X,I: RD=INP(PPIBASE+X): IF RD <> I THEN GOSUB 2000
     ELSE LOCATE 20,5:  PRINT I;CHR$(13);
135  NEXT X: NEXT I: PASS=PASS+1: LOCATE 1,3: SOUND 1,30,10:
     SOUND 2,60,10,14: SOUND 4,120,11,15: PRINT "Pass";PASS;
     "completed";CHR$(20);CHR$(13);: GOTO 130
2000 '
                        Error printing routine

2100 SOUND 1,200,50,15: FAIL=FAIL+1: LOCATE 1,10: PRINT STRING$(80,&2A):
     PRINT TAB(20);"Error report for failure number";FAIL;".": LOCATE 1,13:
     PRINT "Port ";CHR$(65+X);":I/O address "; HEX$(PPIBASE+X);" under test."
2200 PRINT "Details:  Wrote";I;"(HEX) ";HEX$(I):
     PRINT "Read value";RD;"(HEX) ";HEX$(RD):
     PRINT "Press any key to continue"
2300 IF INKEY$="" THEN 2300 ELSE LOCATE 1,6: PRINT CHR$(20): RETURN
65535 END
```

Fig. 3.27: PPI test program.

```
500     '** This program is a test/demo program for the Key Matrix Port
        ** project, see the book text for more details about the KMP.
1000    OUT &F8F3,&99: OUT &F8F7,&80 ' INIT THE PPIs
1100    ON BREAK GOSUB 2800
1200    INK 2,24,1: SPEED INK 10,25: COL=1: MODE 1
1300    FOR O=64 TO 304 STEP 16: FOR I=64 TO 304 STEP 16
1400    MOVE I,O: DRAWR 16,0,COL: DRAWR 0,16,COL: DRAWR -16,0,COL:
        DRAWR 0,-16,COL: NEXT I: NEXT O ' This line draws one box, whose
        bottom left corner is at the graphics screen co ordinates I,O
1500    L=10: U=18: COL=2: GOSUB 2200: COL=1: LOCATE 25,11:
        PRINT "=Last pressed"
1600    LOCATE 5,2: PRINT "M . S . D (HEX)": LOCATE 5,5: PRINT STRING$(16,245)
        FOR I=0 TO 15: LOCATE I+5,4: PRINT HEX$(I): NEXT: FOR I=15 TO 0 STEP -
        LOCATE 3,6+(&F XOR I): PRINT HEX$(I);CHR$(152): NEXT
1700    LOCATE 1,8: PRINT "L": PRINT: PRINT ".": PRINT: PRINT "S": PRINT:
        PRINT ".": PRINT: PRINT "D": FIRST=255
1800    GOSUB 2500 ' Go to the key scanner subroutine. Which will
        not come back to the next line until a key is pressed. The
        number of the key will be in the I variable when it does.
1900    FOR XX=1 TO 3: SOUND 1,50,5,3: SOUND 1,20,5,3: NEXT XX: IF FIRST <> 0
        THEN 2000 ELSE IF I=LASTKEY THEN GOTO 1800 ELSE COL=1: GOSUB 2200
        'Sound to tell user press found. See if first pass-skip if so
2000    U=(I AND &F0)/16: L=I AND &F
2100    FIRST=0: COL=2: LASTKEY=I: GOSUB 2200: GOTO 1800 '
        Reset first flag, ink flashing,remember this key, flash box
        then re-scan.
2200    '
        ** This is a subroutine to fill in a box. The ink used is
           indicated by the variable COL. If this = 2 box will flash. **
2300    MOVE 64+(U*16),64+(L*16): FOR XX=0 TO 16: DRAWR 16,0,COL: MOVER 0,1:
        DRAWR -16,0,COL: NEXT XX
2400    COL=1: RETURN
2500    '
          loop until we find that bit 4 of port C on PPI #1 (See diagram)
          is = Logical one. This means that key number I is pressed

2600    FOR I=0 TO 255: OUT &F8F4,I: IF (INP(&F8F2) AND (&10)) <> 0
        THEN RETURN
2700    NEXT I: SOUND 1,200,20,3: GOTO 2600 ' The SOUND statement indicates
        a complete scan with no closures found. Delete it if it irritates.
2800    MODE 2: END
```

Fig. 3.28: Key matrix port (KMP) familiarisation program.

One final point about the hardware side of the KMP. Although the design allows connection of 256 switches in the matrix, you need only add as many as you need. So if you only need sixteen switches that is all you need have. When you connect less than a full complement of switches though, connect the ones which are installed to the lowest numbered inputs of E7. To take the previous example of sixteen switches only being installed, connect one side of all the switches to pin 17 of SK1 (X0), and connect the other side of each switch to the appropriate Y line, according to the table for SKT1 connections above. When you use less than 256 switches don't forget to modify the software at the points indicated in the REMs, this will save checking uninstalled switches, and will make the scanner part of the programs run a little faster.

Fig. 3.29: Connectional details for a 16 switch matrix on the KMP.

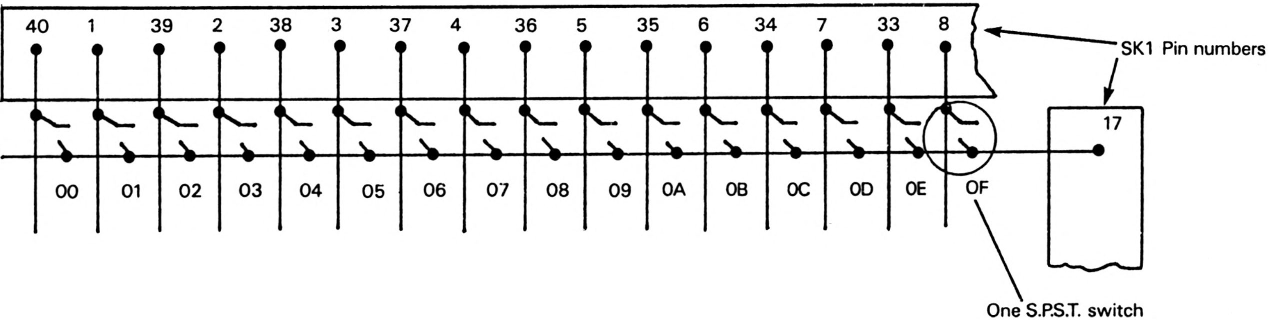
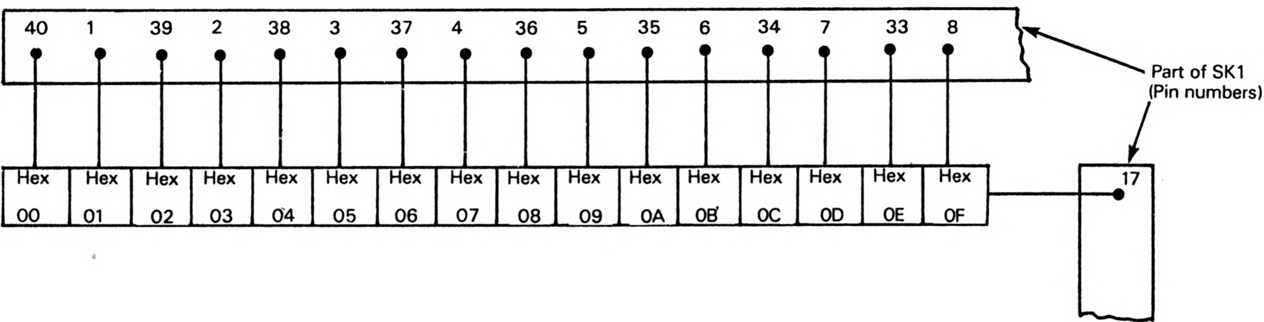


Fig. 3.30: Hex value used when reading each of 16 switches.



Now the Minefield game. You will need to have bought, or constructed a small key matrix in a suitable box, with a minimum of 16 switches on it. The switches are connected to the KMP via a ribbon cable, and DIL header. Connections to make to SK1 are listed in figure 3.29. You can build a bigger keymat than this, and the points in the software to modify in order to include these are mentioned in REMs in the listing. To add more switches just connect them so that they connect to X1 – X15 as required. If you follow Fig 3.29 when connecting up the key matrix your key matrix will correspond to fig 3.30 which shows the hex value which must be written into port A of PPI # 2 to read the state of each of the matrix switches.

The incredibly hackneyed scenario is that the two players have been thrown onto an enclosed minefield by an arch fiend (wouldn't you know it!). They cannot escape, but must go around the minefield until one of them is blown up! Once this has happened the fiend will set the survivor free. The object is for each player to travel around the minefield without being blown up. The minefield is a set of boxes on screen, each representing a switch in the matrix. The program plants random mines, the number planted is a fifth of the total number of boxes specified at line 1300 of the program. The boxes number from 1 in the bottom left hand corner. Each player can jump to any box on the grid, by pressing the corresponding switch on the matrix. There is a brief pause, (to allow you to fall asleep!), and then the square either explodes, or fills with the player's colour to show that there was not a mine. Each safe move scores five points. In order that the game can be played on both the monochrome and colour versions of the CPC, the position of player one is marked by a flashing square, and player two by a non flashing square. That is the game, and the listing for it is shown in fig 3.31.

```
500     '**            C 1985 Kernow computers                          **
        **                    MINEFIELD!                                **

1000    OUT &F8F3,&99: OUT &F8F7,&80: MEMORY 28999: GOSUB 29000
1100    ON BREAK GOSUB 32767
1200    INK 2,20,1: SPEED INK 50,5: COL=1: MODE 1: INK 3,12,3:
        INK 4,12,24
1300  SW=16 '** This is the variable assignment which tells the program
            ** How many switches are in your matrix. You can have any
            ** number up to 256. Numbers less than 8 will cause funnies
1400    IF SW > 256 THEN 32767 ELSE IF SW > 190 THEN SIZ=16 ELSE IF
        SW > 100 THEN SIZ=20 ELSE IF SW > 48 THEN SIZ=24 ELSE SIZ=32
       '** We limit switches to 256, then set the box size on SW
1410    DIM MINES(256): IF SW > 32 THEN YOFF=32 ELSE YOFF=64
1500    XOFF=YOFF: FOR I=0 TO SW-1: GOSUB 27000: GOSUB 26000: NEXT
1510    '** Draw the boxes - one for each installed switch

1600    RANDOMIZE TIME: P(1)=INT(RND*(SW+1)-1): P(2)=INT(RND*(SW+1)-1):
        IF P(1)=P(2) OR P(1)<1 OR P(2)<1 THEN 1600
       '** Select random - different - start positions for players 1 & 2
1700    FOR LOOP=1 TO 2: I=(P(LOOP)): GOSUB 27000: COL=LOOP: GOSUB 25000:
        NEXT: PEN #0,2: LOCATE 1,3: PRINT CHR$(143);: PEN #0,1:
        PRINT "=Player 1:": PRINT: PRINT CHR$(143);"=Player 2:"
1800    FOR LOOP=1 TO (INT(SW/5))
1900    RANDOMIZE TIME: XX=INT(RND*(SW+1)): IF MINES(XX) <> 0 THEN 1900
        ELSE MINES(XX)=255: NEXT LOOP: FIRST=255
2000    CURPLYR=INT(RND*3): IF CURPLYR=0 THEN 2000 ELSE LOCATE 10,24:
        PRINT "Player";CURPLYR;"to begin";: '** Choose which player begins
```

```
2100      GOSUB 28000: LOCATE 10,24: PRINT CHR$(20);:
         '** Go do keypress s/r.  K=switch pressed on ret.
2200      IF FIRST <> 0 THEN 2300 ELSE IF K=LASTMOV THEN LOCATE 15,24:
          PRINT "Move rejected": GOTO 2100 '** See if first go - if so
          ** then its ok to press switch zero. If the move is same as
2250     '** last move then we brutally reject it, and say so!
2300      FIRST=0: I=P(CURPLYR): GOSUB 27000: COL=0: GOSUB 25000:
          IF MINES(K) <> 0 THEN 3000 '** Clr FIRST. Erase old
          position, See if there's a mine at new position.
2400      I=K: COL=CURPLYR: GOSUB 27000: GOSUB 25000: P(CURPLYR)=K:
          IF CURPLYR=1 THEN LOCATE 15,3 ELSE LOCATE 15,5:
         '** Re-draw player on screen, then position cursor
2500      SCOR(CURPLYR)=SCOR(CURPLYR)+5:
          PRINT "Score="; SCOR(CURPLYR);CHR$(18):
          IF CURPLYR=1 THEN CURPLYR=2 ELSE CURPLYR=1
2600      LASTMOV=K: LOCATE 10,24: PRINT "Player";CURPLYR;"to move":
          GOTO 2100:
3000      I=K: GOSUB 27000:MOVE (XOFF+(X*SIZ))+2,(YOFF+(Y*SIZ))+2:
          SPEED INK 2,2: FOR I=1 TO 100 STEP 2: DRAWR I+(INT(RND*15)),0,4:
          DRAWR 0,I,3: DRAWR -(I+INT(RND*15)),0,4: DRAWR 0,-(I+(INT(RND*3))),3
3010      SOUND 1,1,1,15,1,1,15: SOUND 2,2000,1,12,1,1,14:
          SOUND 2,1,1,15,1,1,15: NEXT: FOR I=15 TO 1 STEP -1: INK 2,I:
          SOUND 1,1,30,I,1,1,15: INK 3,I: NEXT: SPEED INK 1,255
3100      LOCATE 1,3: PRINT CHR$(18);STRING$(2,10);CHR$(13);CHR$(18):
          LOCATE 1,24: PRINT CHR$(18);: LOCATE 1,3: IF CURPLYR=1 THEN WIN=2
          ELSE WIN=1
3105      PRINT "Player";WIN;"won: Score was";SCOR(WIN)
3110      T=TIME: WHILE (T+1500) > TIME: WEND: CLS: CLEAR: RUN
3120
         '** The program has ended, the following are subroutines **

25000    '** This subroutine fills in the box whose x,y coordinates
          ** it receives in the varaibles x and y. It adds the screen
          ** offsets XOFF and YOFF to them in order to get an absolute
25010    '** position for them. Because it wants to colour IN the box
          ** and not destroy the box frame there are lots of -1 and +1
          ** values to be found in the expressions for the actual DRAWs.
25020    '** Finally the COL variable contains the colour ink to use
          ** when drawing. The routine set COL back to 1 before it
          ** returns, this saves confusion. COL=2 will draw a flashing fill
25030     MOVE (XOFF+(X*SIZ))+2,(YOFF+(Y*SIZ))+2: FOR XX=1 TO SIZ-3:
          DRAWR SIZ-3,0,COL: MOVER 0,1: DRAWR (-SIZ)+3,0,COL: NEXT XX
25040     COL=1: RETURN
26000    '** This subroutine draws one box, whose bottom left hand corner
          ** is located at the graphics screen coordinates passed by the
          ** caller in variables x and y. The routine adds the XOFF and YOFF
26010    '** variables to these to translate them into real graphic screen
          ** locations. The pen colour used is passed in COL. The actual
          ** dimension of each box (In pixels) is passed in SIZ so if SIZ
26015    '** were equal to 32 then a box 32 pixels square would be drawn.
26020     MOVE (X*SIZ)+XOFF,(Y*SIZ)+YOFF: DRAWR SIZ,0,COL: DRAWR 0,SIZ,COL:
                    DRAWR -SIZ,0,COL: DRAWR 0,-SIZ,COL: RETURN
27000    '** This little routine converts a switch number into X,Y
          ** coordinates for use in locating the screen box which shows
          ** that switch. Assumes that the switches are fitted contiguously
27010    '** The switch number to be converted is passed to it in the
          ** I variable.
27020     Y=INT(I/16): X=I-(Y*16): RETURN
28000    '** This subroutine CALLs the KMPSCAN machine code routine
          ** which scans all possible switch position of the KMP and
          ** returns the code of the first one it finds pressed.
28005    '** if memory location 30001 is zero then no switch was pressed
          ** but if it is non zero then address 30000 contains the number
          ** of the switch pressed.
28010     CALL 30002: R$=INKEY$: IF PEEK(30001)=0 AND R$="" THEN
          28010 ELSE  IF R$=CHR$(5) THEN GOTO 32767 ELSE
          K=PEEK(30000): RETURN' If CTRL/E typed - end game
29000    '** This routine POKEs the KMPSCAN machine code subroutine into
```

```
        ** memory begining at address 30000. CALLs to KMPSCAN are made
        ** by the s/r at line 28000. (Listing for KMPSCAN in chapter 5)
29010   FOR I=30000 TO 30055: READ D: POKE I,D: NEXT
29020   RETURN
29050  DATA 224 , 255 , 213 , 245 , 197 , 1 , 244 , 248 , 22 , 0
29060  DATA 237 , 81 , 14 , 242 , 237 , 120 , 230 , 16 , 32 , 13
29070  DATA 14 , 244 , 20 , 32 , 241 , 50 , 48 , 117 , 50 , 49
29080  DATA 117 , 24 , 18 , 237 , 120 , 230 , 16 , 32 , 250 , 61
29090  DATA 32 , 253 , 122 , 50 , 48 , 117 , 62 , 255 , 50 , 49
29100  DATA 117 , 193 , 241 , 209 , 201 , 0 , 0 , 0 , 0 , 0
32767   MODE 2: END
```

Fig. 3.31: Listing for the Minefield game.

Conclusion

The KMP has a great many possibilities which have not been explored here. For example,if you built the main elements of the design onto a prototyping board, changed the 74151 for a 74LS159 and fit some 7404's or transistors to the outputs of the 74LS154, you could change the whole idea around and fit up to 256 individual LEDs' in place of the switch matrix. This would give you a large 16 row by 16 column dot display, which you could use as a special display or light show for a disco, or as a large equipment status display able to be read from a distance. In both cases some fresh multiplexing software would need to be written.

E Number	Device		description	Power connection pins			
	Type	Equivs	Function	+5V	+12V	−12V	OV
1	74LS20	–	Dual 4I/P NAND	14	–	–	7
2	74LS138	–	3 to 8 decode	16	–	–	8
3	8255A	–	PPI Chip	26	–	–	7
4	74LS04	–	Hex inverter	14	–	–	7
5	8255A	–	PPI Chip	26	–	–	7
6	7415A	–	4 to 16 line decode	24	–	–	12
7	74150	–	16 line selector	24	–	–	12
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							
24							
25							

Fig. 3.32: KMP project power connection pins.

Constructional notes

All of the chips used are TTL or TTL compatible, which means that unless you really go to town, they are unlikely to be damaged by static. The greatest problem is liable to be that of wiring up the external switch matrix in whatever box you choose to house it. I hope that the explanations given will provide you with enough information to stop even this being a problem. When you have wired your switches use the program in figure 3.28 to ensure that you have wired the switches correctly. Finally, the power connection pins for all the chips used are listed in fig 3.32, and the parts list for the key matrix port is shown in figure 3.33.

```
   This is a list of the parts which you will require  to  build  only
the key matrix port project section of the Multiproject board


Capacitors

C1        50 mfd 16 Volt electrolytic          (RS 103-979)
C2-C3     0.1 mfd disc ceramic 16 Volt         (RS 124-178) (2 required)


Semiconductors

E1          TTL 74LS20 Dual 4 input NAND gates  (RS 307-553)
E2          TTL 74LS138 3 to 8 line decoder     (RS 307-648)
E3          8255A PPI chip                      (RS 309-363)
E4          TTL 74LS04 Hex inverters            (RS 307-503)
E5          8255A PPI chip                      (RS 309-363)
E6          74154   4 to 16 line decoder        (RS 306-493)
E7          74150   16 input data selector           -

Miscellaneous

Sockets for all chips
SK1       40 pin DIL socket                                (RS 401-986)
Expansion bus connector                                    (RS 468-119)
Printed circuit board - see note 2 below


       -----------------------------------------------------------

NOTES:
          1) Note that the RS part numbers  quoted  may  be  supplied  in
             packs containing more than one part.

          2) Complete kits of parts are to  be  available  from  Halstead
             designs LTD.
```

Fig. 3.33: Parts list for the KMP project.

Project 5: Sense register

This is a very simple project, which once again uses the PPI to provide inputs to the computer. It is a general purpose 8 bit input port. The reason for calling it a sense register is that we shall look at a way to use it as a means of sensing the state of some switches.

The sense register lives on the multi project board, and uses port B of PPI #2. The port can be read with the command:

```
OUT &F8F3,&8B: PRINT INP(&F8F2)
```

The `OUT` command writes the mode of operation into the PPI control register. Fig 3.34 is the section of the earlier figure 3.23 which deals with the sense register. Port B is extended out to a 16 pin DIL socket – SKT3. A ribbon cable terminated on a DIL header can be plugged into this socket, and the other end of the ribbon cable connected to the signals to be sensed. NEVER connect any signal which is capable of a voltage greater than +5 volts, or less then 0 volts to this port. If you do so you will certainly destroy the PPI chip, and will probably damage your CPC as well. You *can* directly connect signals from other logic circuits to the port, but all other signals will have to be converted to logic levels by external circuitry before being presented. That is about all there is to the sense register hardware, now let us look at some applications.

Fig. 3.34: Sense register section of fig 3.23 reproduced.

Perhaps the most simple application for the sense register is to connect some DIL switch packs to it. These are widely used in nearly all computer peripherals to allow certain features to be switched on and off. The microprocessor which runs the unit reads the state of these switches when it is powered on, and acts accordingly.

Another application is to count pulses from switches connected at the port. This could be useful in data logging applications, where a set of eight switches are provided. Each switch when pressed causes a program running inside the computer to increment a counter assoctiated with that switch. This kind of application is useful for gathering information of a simple statistical nature. For example, a bird watcher might make a small switch box to do a count of how many: sparrows, starlings, wrens, crows, blue tits, blackbirds, robins, and finches were seen during a watch on a particular garden. If the computer is already available, the cost of making a switch box for this kind of work is very modest. Of course this assumes that the computer can be plugged into a mains outlet at the spotting site.

Let us take a look at some software to allow us to count how many times we press eight switches connected to the sense register. Firstly, we need a small box which can be hand held, and enough cable to connect from it to the expansion bus box for whatever application we are carrying out. The switches in the hand held box should be connected to SKT3 as shown in fig 3.35.

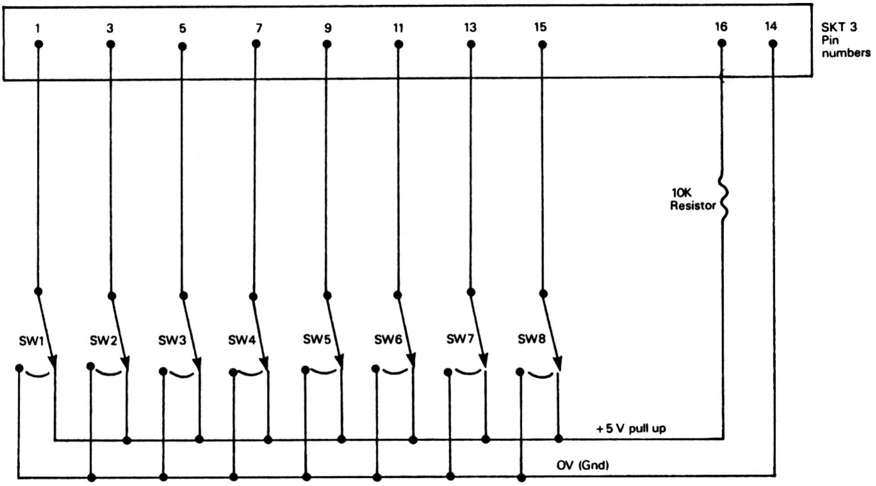


Fig. 3.35: Circuit diagram of switch box for use with sense register.

What we now need is a program to repeatedly read the sense register, and to implement debounce delays when a switch is pressed. (See the text of project three for an explanation of debouncing.) Further this program should count how many times each switch has been pressed during the time it has been running, and maintain an on–screen display of the count values. An additional facility is that the statistics gathered can be stored in a cassette file by pressing the "#" key (Shift 3) on the CPC keyboard. This would allow a further program to be written which would produce a report containing averages of the contents of several of these files. The counters are to be 16 bit counters, which means that count values from zero to 65535 can be obtained. Fig 3.36 is a listing of a BASIC program to do this job. Later, in chapter 5, we shall return to this application, and develop an assembly language program to perform a similar function.

```
100     ' **            Data logger - BASIC version                        **
          **                                                                 **
200     ' ** This software accompanies the book "Understanding and         **
          ** expanding your Amstrad CPC 464/664" by Alan Trevennor.        **
300     ' ** The data logger counts presses of each switch connected **
          ** to the sense register detailed as a hardware add on         **
          ** project in chapter three of the book.                        **
400       CLEAR: MODE 2: FOR I=0 TO 7: LOCATE ((10*I)+1),1: READ T$:
          PRINT T$;: NEXT: OUT &F8F7,&AF: PASS=1: DIM C(20)
500     ' ** Line 400 prints the types of bird we might be spotting    **
          ** but you can change this to anything you like - EG car     **
          ** types, types of people, etc etc.                          **
600     ' ** Now we scan the switches repeatedly and increment the     **
          ** counter in the C matrix which is associated with that     **
          ** switch. If user presses # key (Shift 3) then we dump to **
700     ' ** a tape or disk file as appropriate for the machine type.**

800       R=0: I$="": WHILE R=0 AND I$<> "#": R=INP(&F8F5): I$=INKEY$:
          GOSUB 5000: WEND: IF I$="#" THEN 1000 ELSE FOR I=0 TO 7:
          IF (R AND (2^I)) <> 0 THEN C(I+1)=C(I+1)+1
900       NEXT I: FOR DB=1 TO 50: NEXT: GOTO 800
        ' ** The for next loop of DB above is a debounce delay loop    **
1000    ' ** Write data to disk or tape file routine                   **

1100      OPENOUT "DATLOG."+STR$(PASS): FOR I=1 TO 8: PRINT #9,STR$(C(I)):
          C(I)=0: NEXT: CLOSEOUT: PASS=PASS+1: LOCATE 1,2: PRINT CHR$(20);:
          GOTO 800' ** We create DATLOG.xx file (Where xx is sample no.)**
5000    ' ** Small subroutine to update counter display on screen      **

5100      FOR I=0 TO 7: LOCATE ((I*10)+1),2: PRINT C(I+1);: NEXT:
          LOCATE 1,5: RETURN
32000     DATA SPARROW,OWLS,WRENS,CROWS,TITS,CURLEW,ROBINS,FINCH
```

Fig. 3.36: BASIC program to run a data logger.

Conclusion

We have looked at one particular application for the sense register, in statistics gathering. This most basic of computer I/O building blocks has many, many, more uses.

Construction notes
The only thing to say about building the sense register is to stress what has already been said. DO NOT connect any signals to the PPI which exceed the normal logic levels. Ignoring this warning will inevitably result in damage to your CPC.

Project 6: Parallel transfer channel

This is another project which lives on the multi project board. The purpose of this project is to allow fast transfer of data from one CPC to another, or if there is suitable hardware available, between the CPC and other computers.

With the advent of computer networks it is common for computer users to demand ways to exchange data and programs between machines. This is especially true in the field of mini computers and up–market micro computers. Home computer users have only recently taken an interest in swapping data and programs with one another over networks, usually with the best of motives. The parallel transfer channel hardware presented here, will allow a number of CPC machines to be connected in a small local network configuration. This will involve the construction of the hardware presented in this section, and also of the EPROM programmer, and the expansion ROM board. The network driving software will be presented as an advanced project in chapter six.

Let us look at some undeveloped ideas for the use of the parallel transfer channel as it stands. The Parallel transfer channel gives us a way to transfer data a byte at a time from our CPC to another byte oriented device. We also gain the capability to transfer data at a far greater rate than would be possible with an RS 232 serial link. We should therefore be looking at bulk transfers of data into, or out of, the CPC, rather than just messages etc.

One frequent application for electronics in the scientific field is data logging devices. The logging device might be a black box which has been left to record temperatures, humidity, and amount of sunlight in a weather study. It might be a traffic logging device which has been recording traffic density over a period of time. These devices usually have built in battery operated memory, or a paper tape punch or cassette recorder to record the raw data.

The common factor with data logging devices is that sooner or later they are returned to the laboratory, classroom, or office. They are then often connected to a computer so that the raw data can be read into the computer, which either stores or analyses it. The parallel transfer channel can be used to input such data to the CPC for storage or analysis.

Another possible application is in interfacing to printers. There are some printers which it will prove impossible to connect to the printer port of the CPC, because they need 8 bit character codes,

whereas the CPC printer port can only transfer 7. (See Appendix 7).

Parallel transfers to large computers from your CPC are quite possible. All mini and mainframe computer manufacturers produce at least one parallel data interface, which could connect to the one presented here. If you have access to a large machine, and can get such an interface rigged up, you would find it possible to produce letters, programs, and textual data on your CPC at home. You could then take your CPC to the large machine and transfer your homework into it very quickly. No reasonable management or administration could refuse you this hi–tech way of taking work home! If you demonstrate that the idea gets results, you might find that a dedicated CPC would be bought to be permanently connected to the large machine! In comparison with the price of mini or mainframe computers the price of a second CPC would be minsicule, so this is quite likely to happen in a wealthy company or institution.

Parallel transfer channel hardware

Once again the project is heavily dependent on one of the PPI chips. In fact apart from the address decode and the PPI, this is really a software project. PPI #1 ports A,B and half of port C are used for a parallel transfer channel. The PPI is run in mode 1, control byte value Hex AF. This sets up port A as an 8 bit output port, port B as an 8 bit input port. Bits 6 and 7 of port C are the status signals for port A, and port C bits 1 and 2 are the status signals for port B. Bits 0 and 3 of port C are available for interrupts, but remain unused in this design. Bits 4 and 5 of port C are set up as inputs for use by the speech synthesiser and Key Matrix port projects, which were described earlier in this chapter. PPI #1 and its associated logic and connectors are shown in Fig. 3.37.

Referring to Fig. 3.37 PPI #1 is shown with all the lines required for the parallel transfer channel being brought to the offboard connector PLG 1, which is a 25 way "D" type connector. The pinout list for PLG 1 will be listed after the meanings of the signals have been detailed. E4a is an inverter which inverts the bus RESET signal from the CPC. This is because the 8255 needs a normally low input on its RESET input, it must only go high when the chip is being reset. The CS bar input of PPI #1 comes from the Y0 output of the address decoder (refer back to Fig. 3.15). This places the PPI at the following Hex I/O addresses:

Port A	Hex F8F0
Port B	Hex F8F1
Port C	Hex F8F2
Control register	Hex F8F3

Therefore, the BASIC command:

```
OUT &F8F3,&AF
```

will initialise the PPI to the required mode as detailed in the first paragraph of this section.

The mode of operation selected for of the PPI is ideal for a Parallel transfer channel application. The two intercommunicating devices are connected by a 25 way cable. This can be a ribbon type, but a shielded conventional cable

would be better. The maximum length of this interconnecting cable should be no more than 30 feet in a favourable environment (that is, where the cable is not run alongside any mains cables or near electrical noise generators like large motors or transformers.) In adverse circumstances about ten feet of cable should be considered as a maximum. Trial and error are the best judges of this decision. A test program, along with a circuit to enable you to soak test the Parallel transfer channel will be given at the end of this section.

The input port B as the 8 data input lines, and also two other signals. These are the strobe input, and the "input buffer full" (IBF) output. The strobe input is pulsed low to command the input port to latch in the data being presented from the remote device. The IBF output goes high to show that the input port has taken the data presented. IBF goes low when the processor has read the data which has been input. After this has happened the next byte can be transferred.

The output port A has 8 data output lines, and two status lines associated with it. These are the Output Buffer Full output (OBF bar), and the ACK bar input. The OBF bar output goes low when the processor has loaded a byte of data into the output port. the ACK bar input is pulsed low by the remote device to tell the port that it has read the data being presented out of the port. Fig. 3.38 shows how this translates into the practical connection of two parallel transfer channels. Perhaps a slightly humanised explanation of a transfer sequence will clarify the use of these signals:

STROBE: (from the remote device) when pulsed low means – "I say CPC old chap, I've just put a byte into your input port".

IBF: (from the CPC) when taken high means – "Oh so you have, half a mo' and I'll find somewhere to store it".
OBF: (from the CPC) when taken low means – "Hey there remote device! I have just put a byte onto my output port which is addressed to you".

ACK: (from the remote device) when pulsed low means – "I have read your byte, and will treasure it always!".

As briefly mentioned above, when the PPI is programmed to operate in this mode part of port C is used for the status and control signals. These are:

Port C bit 7 = OBF bar
Port C bit 6 = ACK bar
Port C bit 2 = STB bar
Port C bit 1 = IBF
Port C bits 3 and 0 = can be used as interrupt signals though they are not used in this design.

The beauty of using the 8255 in this way is that the processor is relieved of any detailed responsibility for the transfers, because the 8255 takes care of all the handshaking between itself and the remote device. (Handshaking is the technical term for the signal sequencing needed to synchronize data transfers between computer devices.) All it needs to do is write the byte to be sent into port A, and then execute a wait loop until it reads OBF bar as being high. For input the processor need only check the state of IBF (Port C bit 1). If this is high then a byte from the remote device is available at port B.

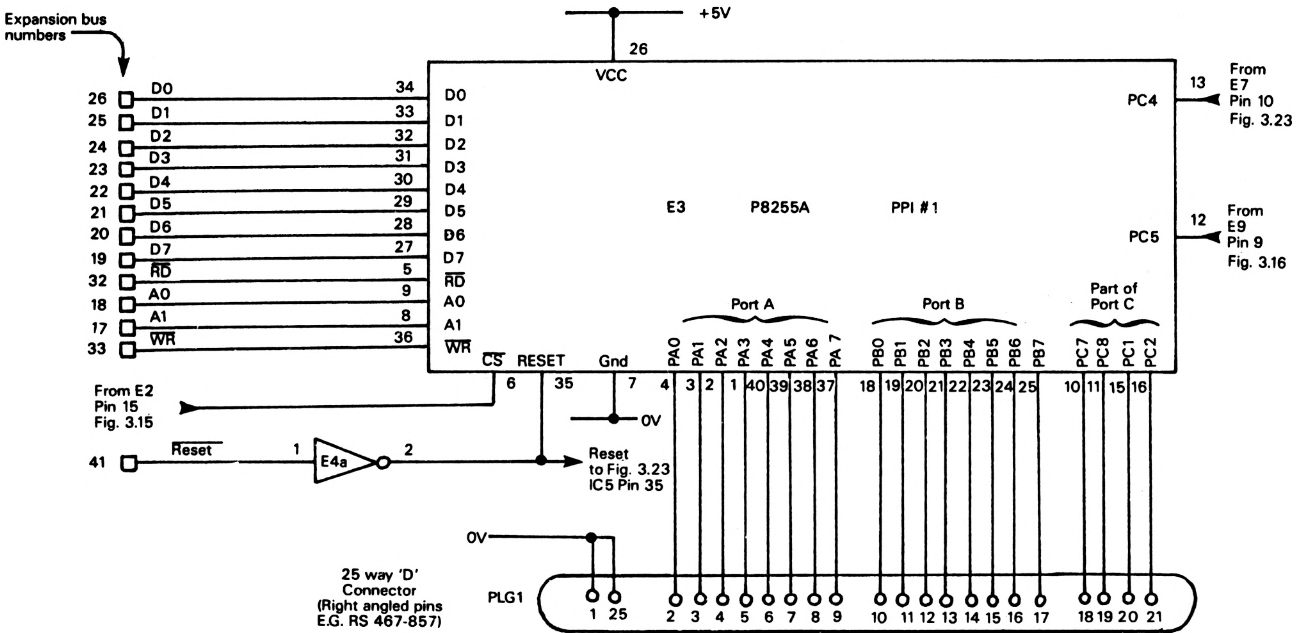


Fig. 3.37: PPI #1 circuit diagram showing the paralell transfer channel.

Now that we have looked at all the signals involved, here is the list of pin connections to the connector PLG 1, shown on diagram 3.37.

Pin usages for connector PLG 1

Pin	Signal name	Pin	Signal name
1	0 Volts	14	In data bit 4
2	Out data bit 0	15	In data bit 5
3	Out data bit 1	16	In data bit 6
4	Out data bit 2	17	In data bit 7
5	Out data bit 3	18	OBF bar (PC7)
6	Out data bit 4	19	ACK bar (PC6)
7	Out data bit 5	20	IBF (PC1)
8	Out data bit 6	21	STB bar (PC2)
9	Out data bit 7	22	Spare
10	In data bit 0	23	Spare
11	In data bit 1	24	Spare
12	In data bit 2	25	0 Volts
13	In data bit 3		

Figure 3.38 lists both a BASIC program to soak test the parallel transfer channel electronics, along with the list of pins to link on a 25 way female D plug to connect the output port to the input port. If after running this program error free for at least two hours you find that you get corruptions when using the channel to another machine, then it may be that your interconnecting cable is too long, or too near an electrical noise source. If one bit is in error all the time, a badly soldered joint on PLG 1 or the cable connections may be to blame. The solutions are to check your soldering, re route or shorten the cable, fit a screened cable if not already fitted, or to make a small board up at each end of the cable for fitting special long line driver chips on to. (eg TTL type 74LS07 or 74LS128).

```
                 Test connector details for PTC testing


      On a 25 pin "D" connector the following pins  should  be
 connected to one another to make the test connector  required
 to run the soak test program listed below:

 Connect    2 to 10
 Connect    3 to 11
 Connect    4 to 12
 Connect    5 to 13
 Connect    6 to 14
 Connect    7 to 15
 Connect    8 to 16
 Connect    9 to 17
 Connect   18 to 21   (OBF bar to STB)
 Connect   19 to 20   (IBF to ACK bar)

   Linking the indicated pins will loop the  send  channel  to
 the receive channel, the program then sends all possible byte
 values to the send channel and ensures that it receives  them
 back correctly with comprehensive error reports  if  it  does
 not.
 -------------------------------------------------------------
```

Soak test program

```
1000    MODE 2: HED$=STRING$(79,&2A): PRINT HED$: PRINT TAB(17);
        "Parallel transfer channel soak test program": PRINT HED$
2000    LOCATE 1,5: PRINT "Test set up phase": LOCATE 1,7:
        INPUT "Is the test plug plugged into PLG 1 <No>";R$:
        R$=UPPER$(LEFT$(R$,1)): IF R$ <> "Y" THEN LOCATE 1,7 ELSE 2200
2100    PRINT "The test plug detailed in the book must be plugged":
        PRINT "into the PLG 1 connector on the Multi project board":
        GOTO 32767
2200    LOCATE 1,5: PRINT CHR$(20);"PTC soak test underway": LOCATE 1,8:
        PRINT "Test commencing: Pass xx  completed:  Errors so far = xx "
2300    OUT &F8F3,&AF: FOR P=1 TO 999: FOR T=255 TO 0 STEP-1:
        OUT &F8F0,T  : WHILE (INP(&F8F2) AND 2) = 0: WEND: R=INP(&F8F1):
        IF R <> T THEN GOSUB 10000
2400    NEXT T: LOCATE 22,8: PRINT P;: NEXT P: GOTO 32767
10000   LOCATE 1,17: PRINT CHR$(7);HED$: PRINT TAB(20);"Error report":
        PRINT HED$: PRINT: PRINT "Sent Hex ";HEX$(T);": Received Hex "
        HEX$(R): PRINT: BITERR= ( T XOR R)
10010   PRINT TAB(50);"Sent  ";SPACE$(8-LEN(BIN$(T)));BIN$(T):
        PRINT TAB(50);"Rec   ";SPACE$(8-LEN(BIN$(R)));BIN$(R)
10015   LOCATE 1,22: PRINT "Bits in error are:  ";: FOR B=0 TO 7:
        IF (BITERR AND (2^B)) <> 0 THEN PRINT B;":";
10020   NEXT: FAILS=FAILS+1: LOCATE 55,8: PRINT FAILS: LOCATE 1,25:
        PRINT "Press any key to continue";: R$="": WHILE R$="":
        R$=INKEY$: WEND: LOCATE 1,17: PRINT CHR$(20);: RETURN
32767   LOCATE 1,20: PRINT "Program run finished"
```

Fig. 3.38: Soak test program for the paralell transfer channel.

Parallel transfer channel software

The software to enable the parallel transfer channel to be used in a local area network will be detailed in chapter 6. However, a listing for two machine code programs to perform the fundamental operations of the channel are listed as figures 3.39 and 3.40. (As Z80A machine code programming has not yet been introduced, you may not understand these programs, but put in a book mark and come back to them.) The "Send" program in figure 3.39 is a machine code subroutine which will send the data whose length is indicated by the value it finds in the DE register pair, and the first address of which it finds in the HL register pair. The "RX" program listed in fig 3.40 is a subroutine which takes the same arguments in the same registers, but uses them as parameters of how many bytes to receive. If at any time during the reception the user types CTRL/A at the keyboard, the reception is aborted.

Figure 3.41 is a BASIC program to receive data and write it to tape or screen. Beware of writing non textual data to the screen, it can have some very strange effects, and you may lose the program, so if you try it SAVE the program first. Finally, figure 3.42 is a BASIC program to take data from the cassette or keyboard and send it out to the parallel transfer channel.

```
100 ;
110 ;
120 ; ** SEND.  This machine code subroutine will send a number **
130 ; ** of bytes out over the Parallel Transfer Channel (PTC). **
140 ; ** On entry the HL register points to the first byte to   **
150 ; ** to be sent, and DE contains the number of bytes to be   **
160 ; ** be sent. AF corrupted on return.                        **
170 ;
180 PTCSND: EQU  #F8F0 ; Define the addresses for the PTC regs.
190 PTCREC: EQU  #F8F1
200 PTCCNT: EQU  #F8F2 ; The control signals for the channels
210 PPICTL: EQU  #F8F3 ; The control reg for the PTC's PPI chip
220 ;
230 ;
240 ;
250 ;
260         ENT  $          ; Start execution here please!
270 SEND:   PUSH BC
280         PUSH DE
290         PUSH HL
300 SEND00: LD   BC,PTCCNT    ; Make BC point to PTC status register
310         IN   L,(C)        ; Get status into L
320         BIT  7,L          ; See if send channel is ready
330         JR   NZ,SEND01    ; Jump if yes
340         CALL #BB09        ; see if user has typed anything
350         JR   NC,SEND00    ; Loop if not
360         SCF  ; If so Set carry
370         JR   SEND04       ; And end up
380 SEND01: LD   BC,PTCSND    ; Now BC points to send channel
390         LD   A,(HL)       ; Get a byte to A
400         OUT  (C),A        ; output it
410         LD   A,#0         ; Now see if count reached zero
420         CP   E            ; compare with E reg
430         JR   NZ,SEND05    ; loop if not zero in low byte
440         CP   D            ; test D register
450         JR   Z,SEND06     ; jump to exit if DE now = 0000
460 SEND05: DEC  DE           ; if DE not yet zero subtract one
470         INC  HL           ; and update buffer pointer
480         JR   SEND00       ; from DE and do the loop again
490 SEND06: AND  A            ; Ensure carry flag is clear
500 SEND04: POP  HL           ; POP the stack
510         POP  DE
520         POP  BC
530         RET  ; return
540 ;
550 ;
```

Fig. 3.39: M/C program "SEND" for the PTC.

```
100 PTCSND: EQU  #F8F0 ; Define the addresses for the PTC regs.
110 PTCREC: EQU  #F8F1
120 PTCCNT: EQU  #F8F2 ; The control signals for the channels
130 PPICTL: EQU  #F8F3 ; The control reg for the PTC's PPI chip
140 ;
150 ;
160 ; ** RX: This routine receives bytes from the parallel
170 ; ** transfer channel (PTC). It places them in a buffer
180 ; ** which is pointed to by the HL register on entry,
190 ; ** and it will return after receiving the number of
200 ; ** characters indicated by the entry contents of the
210 ; ** DE register. When bit one of PTCCNT goes high
220 ; ** a byte is available. While RX is waiting it also
230 ; ** keeps calling a firmware routine to see if CTRL/A has
240 ; ** been typed at the keyboard. If yes, RX returns with
250 ; ** the carry flag set. AF register corrupt on return
260 ;
```

```
270 ;
280             ENT  $    ; Start execution here.
290 RECEEV: PUSH BC
300         PUSH HL
310         PUSH DE
320 RECEE2: LD   BC,PTCCNT    ; BC points to control channel
330         IN   A,(C)        ; Get control status
340         BIT  1,A          ; See if IBF is true
350         JR   NZ,RECEE1    ; Jump if so
360         CALL #BB09        ; else check for CTRL/A's
370         JR   NC,RECEE2    ; If no key pressed then loop
380         CP   #1           ; else see if CTRL/A
390         JR   NZ,RECEE2    ; loop if no
400         SCF  ; if yes then set carry
410         JR   RECOUT       ; and go  out
420 RECEE1: LD   BC,PTCREC    ; BC points at receive channel
430         IN   A,(C)        ; get the byte
440         LD   (HL),A       ; Buffer the byte
450         LD   A,#0         ; Get zero to the A reg
460         CP   E            ; See if its zero yet
470         JR   NZ,RECEE3    ; loop again if not
480         CP   D            ; See if D is zero yet
490         JR   NZ,RECEE3    ; Jump if not
500         AND  A            ; Ensure carry is clear
510         JR   RECOUT
520 RECEE3: DEC  DE           ; DE gets one taken away
530         INC  HL           ; Advance HL (buffer pointer)
540         JR   RECEE2       ; Get another character.
550 RECOUT: POP  DE
560         POP  HL
570         POP  BC
580         RET  ; And go back
590 ;
```

Fig. 3.40: M/C program "RECEEV" for the PTC

```
100  '** Program to receive characters from the PTC and
      ** put them in a disk or tape file.
110 MODE 2: INPUT "Data to screen or tape"; R$: R$=UPPER$(LEFT$(R$,1)):
    IF R$="S" THEN S=0 ELSE S=9
120 IF S=9 THEN INPUT "Name of the tape or disk file";F$:
    IF INSTR(F$,".") <> 0 THEN 120 ELSE OPENOUT F$:
    PRINT "File opened"
130 ON BREAK GOSUB 150
140 WHILE (INP(&F8F2) AND 2) = 0 : WEND: PRINT #S,CHR$(INP(&F8F1));:
    GOTO 140
150 CLOSEOUT: IF S <> 0 THEN PRINT "Reception aborted - file ";
    F$;" created"
32767 END
```

Fig. 3.41: BASIC reception program to receive data and write it to tape.

```
1000    ' ** Program to send data to the PTC from tape or memory
1010    ON BREAK GOSUB 10000: MODE 2: INPUT
        "Data source: (K)eyboard or (F)ile";R$: R$=UPPER$(LEFT$(R$,1)):
        IF R$ ="F" THEN 2000 ELSE IF R$="K" THEN 3000 ELSE 1010
2000    CLS: CAT: INPUT "Which file to send";FF$:
        INPUT "Filetype - Binary or ASCII";T$: T$=UPPER$(LEFT$(T$,1)):
        IF T$="A" THEN 2010 ELSE 2100
2010    OPENIN FF$: WHILE EOF =0: LINE INPUT #9,R$:
        FOR I=1 TO LEN(R$): WHILE (INP(&F8F2) AND &80) = 0: WEND:
        OUT &F8F0,ASC(MID$(R$,I,1)): NEXT: WEND: GOTO 10000
2100    MEMORY &3FFF: LOAD FF$,&4000: PRINT "16K from Hex 4000 will be sent":
        FOR I=1 TO 16384: WHILE (INP(&F8F2) AND &80) = 0: WEND:
        OUT &F8F0,PEEK(&4000+I): NEXT: GOTO 10000
3000    CLS: PRINT "Type characters to be sent, end with ESC":
        FOR I=1 TO 65000: X$="": WHILE X$="": X$=INKEY$: WEND: PRINT X$;:
        OUT &F8F0,ASC(X$): NEXT
10000   CLS: PRINT "Transfer to PTC ended"
32767   END
```

Fig. 3.42: BASIC program to send data from tape to PTC.

```
Capacitors

C1       50 mfd 16 Volt electrolytic         (RS 103-979)
C2-C3    0.1 mfd disc ceramic 16 Volt        (RS 124-178) (2 required)


Semiconductors

E1       TTL 74LS20 Dual 4 input NAND gates  (RS 307-553)
E2       TTL 74LS138 3 to 8 line decoder     (RS 307-648)
E3       8255A PPI chip                      (RS 309-363)
E4       TTL 74LS04 Hex inverters            (RS 307-503)


Miscellaneous

Sockets for all chips
PLG 1    25 way D connector with right angled pins     (RS 467-857)
Expansion bus connector                                 (RS 468-119)
Printed circuit board - see note 2 below

        ------------------------------------------------------------

NOTES:
        1) Note that the RS part numbers quoted may be supplied in
           packs containing more than one part.

        2) Complete kits of parts are to be available from Halstead
           designs LTD.
```

Fig. 3.43: Parts list for the PTC.

This parts list is for all the projects on the Multi-project board. If you only want to build some of the projects on it please refer to the individual parts list for the projects you are building

Capacitors

C1	50 mfd 16 Volt electrolytic	(RS 103-979)	
C2-C5	0.1 mfd disc ceramic 16 Volt	(RS 124-178)	(4 required)
C6	27 pf polystyrene bead 16 Volt	(RS 114-654)	
C7	0.1 mfd any type 16 Volt	(eg RS 115-102)	
C8 & C9	0.22 mfd 16 Volt	(RS 115-118)	(2 required)

Resistors

R1-R8	10K Ohm quarter Watt	(RS 131-378)	(8 required)
R9	470 Ohm quarter Watt	(RS 131-211)	
R10 & 11	10K Ohm quarter Watt	(RS 131-378)	(2 required)
RV1	47K pcb mounting potentiometer	(RS 186-889)	

Semiconductors

E1	TTL 74LS20	Dual 4 input NAND gates	(RS 307-553)
E2	TTL 74LS138	3 to 8 line decoder	(RS 307-648)
E3	8255A PPI chip		(RS 309-363)
E4	TTL 74LS04	Hex inverters	(RS 307-503)
E5	8255A PPI chip		(RS 309-363)
E6	74154	4 to 16 line decoder	(RS 306-493)
E7	74150	16 input data selector	-
E8	74LS32	Quad 2 i/p OR gates	(RS 307-569)
E9	SPO-256		-
E10	74LS174	Hex data latches	(RS 307-682)
E11	74LS629	Voltage controlled oscillator	-

Miscellaneous

Sockets for all chips

PLG 1	25 way D connector with right angled pins	(RS 467-857)
SK1	40 pin DIL socket	(RS 401-986)
SK2	14 pin DIL socket	(RS 401-790)
SK3	16 pin DIL socket	(RS 401-829)
Expansion bus connector		(RS 468-119)
Printed circuit board - see note 2 below		

NOTES:

1) Note that the RS part numbers quoted may be supplied in packs containing more than one part.

2) Complete kits of parts are to be available from Halstead designs LTD.

Fig. 3.44: Complete parts list for the multiproject board.

Construction notes
If you are using the multi project board PCB (see page three for details of PCB service) there should be no difficulty with this project. The only area of difficulty might come in making up a cable to connect the parallel transfer channel to other devices. Even this should not be a problem since with the connectors specified the pin numbers are marked on the plug mouldings, and so as long as the points just presented about cable length and routing have been heeded, no problems should occur. As noted in the parts list given in figure 3.43 the 8255 should be an "A" version, I have found that many straight 8255 chips do not work satisfactorily. As always mount all chips in sockets.

Conclusion
The parallel transfer channel represents a way of greatly expanding your computing activities. This is because it enables you to exchange data with other nearby computers, though not over communications lines (see next project). It also allows you to receive from data capture units, and to develop your own ideas for battery operated devices which can bring back data to the CPC 464 for anaylsis.

Multi project board conclusion

That concludes the projects which are on the multi project board. The final figure – 3.44 – is a complete parts list for the multi project board circuitry.

Project 7: RS 232 version "B" interface

In project six we looked at a way to transfer data quickly between the CPC machines and other computers or devices. As was mentioned in the description of that project the interconnecting cable between two machines is prone to noise problems, and where the signals are standard TTL logic levels this places a limit on the length over which data can be transmitted. You may find that you get from one room to another, but as for any further, that is pushing your luck!

This project gets around many of the problems associated with TTL level transmission of data over longer distances. If you have read appendix 6 (entitled "Begin here") you will have found out why we use RS 232 links between computers and terminals, or between computers. The main reason is that the signals travel over the interconnecting cable at two or three times TTL voltages, which greatly increases noise immunity. Other reasons are that the computer industry long ago siezed on RS 232 as the major standard interface for serial data transmission. That means that very nearly all computers, whether home micros or mainframes, have an RS 232 channel either built in or available as an optional extra.

Sounds great doesn't it? Before we get too carried away let us look at the drawbacks of using RS 232. Well actually these aren't too severe. First the

bytes transferred are sent down a (typically) three conductor line: one to send data: one ground: one to receive data. This means that circuitry within the serial interface must serialise the data, and when it arrives at the other end it must be de–serialised, that is, converted back into bytes from a series of precisely timed pulses. This is no problem because there is a plethora of chips available to perform this function for you. The problem is that of time, it takes longer to transfer a given number of bytes down a three wire serial link than down a parallel link, and it always will, technology cannot change it. The other disadvantage is that more electronics is needed to implement a serial scheme, but we can live with that.

In its fullest form this project provides four eight bit RS 232 channels for your CPC. Why would you need four channels? You probably don't. Four channels have been included because once the address decoder is built, and you have got the RS 232 transmit and receive chips, and the Baud rate generators, it is a simple matter to add the serial transmit/receiver chip for another channel. So the design allows a maximum of four channels, but you only have to install one, if that is all you need.

What about the uses for this project? It would be quicker to list the uses it CANNOT be put to. It will not make the coffee, or make the bed, or wash up. (Do you ever get tired of sceptical friends and relatives asking you if your CPC can wash up?) But seriously, here is a list of some things you CAN use it for:

1) To allow you to use your CPC as a serial terminal. So that if you get a MODEM (look at the small ads. in the hobby press) you can dial into many of the networks which are popping up all over the place.

2) To allow you to use your CPC for data preparation, you can then downline the data to a bigger machine when it is ready.

3) Using software presented later in this section you can connect your CPC as a VDU to up to four systems, with keyboard selection of features in a set–up mode, just like a "real" VDU – (just joking Amstrad!)

4) You can directly connect to a similarly equipped CPC, or other lesser micro computer. This will allow you to have a closed circuit dialogue with another owner in the same building.

5) You can use the CPC as a cheap test bed for testing printers or VDU's in a workshop environment. This will involve writing software to send specific patterns – not difficult.

And there are many, many, more applications (but I'm afraid it definitely will NOT wash up).

The first of the circuit diagrams for the RS 232 version "B" interface is figure 3.45. This shows the address decoder, and the baud rate latches and generators.

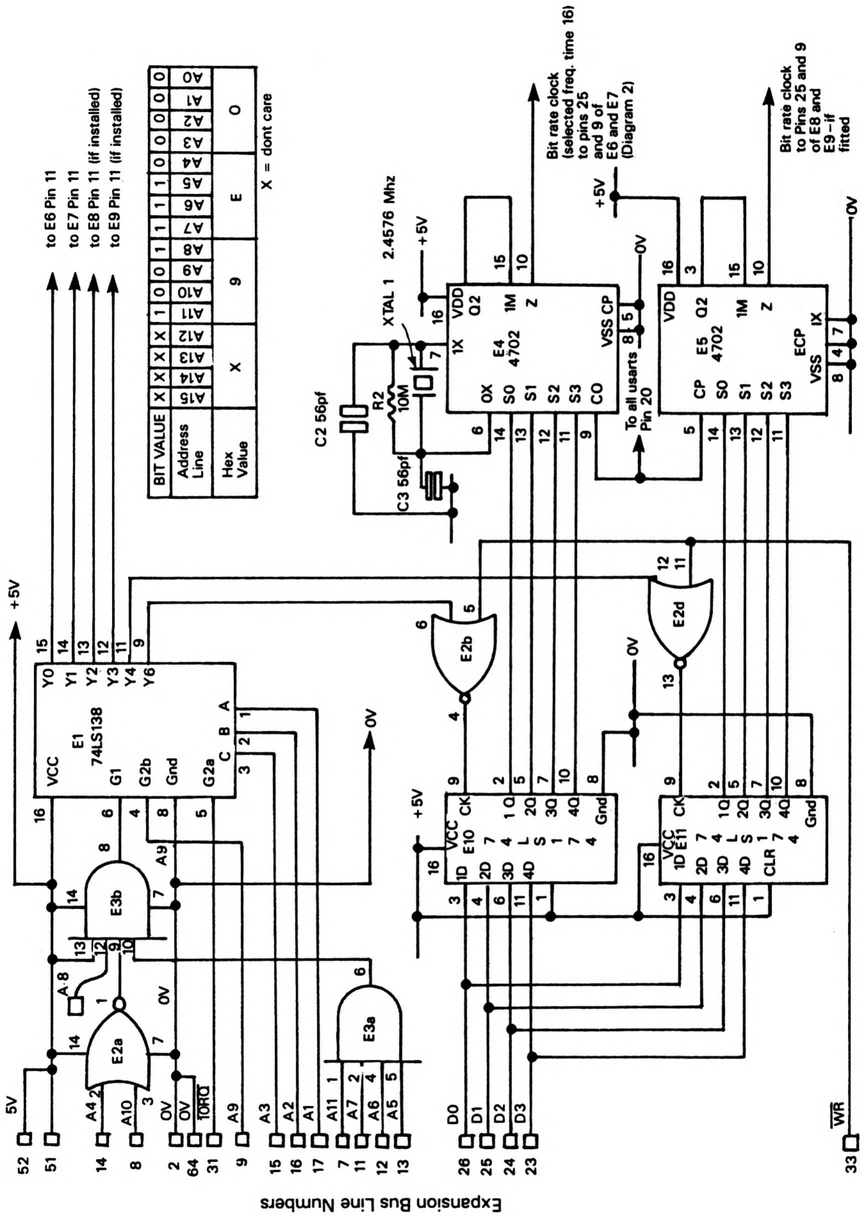
They are shown on a later diagram, but each channel has a chip called a USART (Universal Synchronous Asynchronous Receiver Transmitter). These are sometimes known as PCI chips (Programmable Communications Interface). We will stick with USART. Each USART has a control register and a data register. When we get onto the second diagram we will look a little closer at these useful chips.

Turning first to the address decode circuit shown in fig 3.45, The base address of the interface is I/O address Hex F9E0. The interface occupies the next I/O addresses after this as follows:

F9E0	USART 1 data register.
F9E1	USART 1 control register.
F9E2	USART 2 data register.
F9E3	USART 2 control register.
F9E4	USART 3 data register.
F9E5	USART 3 control register.
F9E6	USART 4 data register.
F9E7	USART 4 control register.
F9E8–B	Baud rate register for channels 1 & 2
F9EC–F	Baud rate register for channels 3 & 4

The bit pattern for F9E0 is shown on Figure 3.45. The gates E2a,E3a, and E3b enable E1 (a 74LS138) when an address beginning with F9E, the IORQ bar strobe and a low on address line 9 occur simultaneously. E1 then decodes address lines 1,2 and 3 into one of six possible chip select strobes. The gates E2b and E2d NOR gate the select strobes to the baud rate registers, with the write strobe, to produce the latch load clock signals. The baud rate registers are write only. The other four chip select signals go off to the USARTs, with the A0 address line selecting either the data or control register on the selected USART.

Returning to the baud rate select registers, the value put into these registers determines what baud rate is selected for each pair of USART chips. The baud rate generator chips are 4702 types, and have their select inputs connected to the four bit outputs of the baud rate registers. The value placed into the registers corresponds to a baud rate according to the following table:



BIT VALUE	X	X	X	X	1	0	0	1	1	1	1	0	0	0	0	0
Address Line	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
Hex Value	X				9				E				0			

Fig. 3.45: Circuit diagram for the RS 232 version B address decoder.

Hex Value in register.	*Baud rate selected*
0	19200
1	19200
2	50
3	75
4	134.5
5	200
6	600
7	2400
8	9600
9	4800
A	1800
B	1200
C	2400
D	300
E	150
F	110

So that for example, the direct command

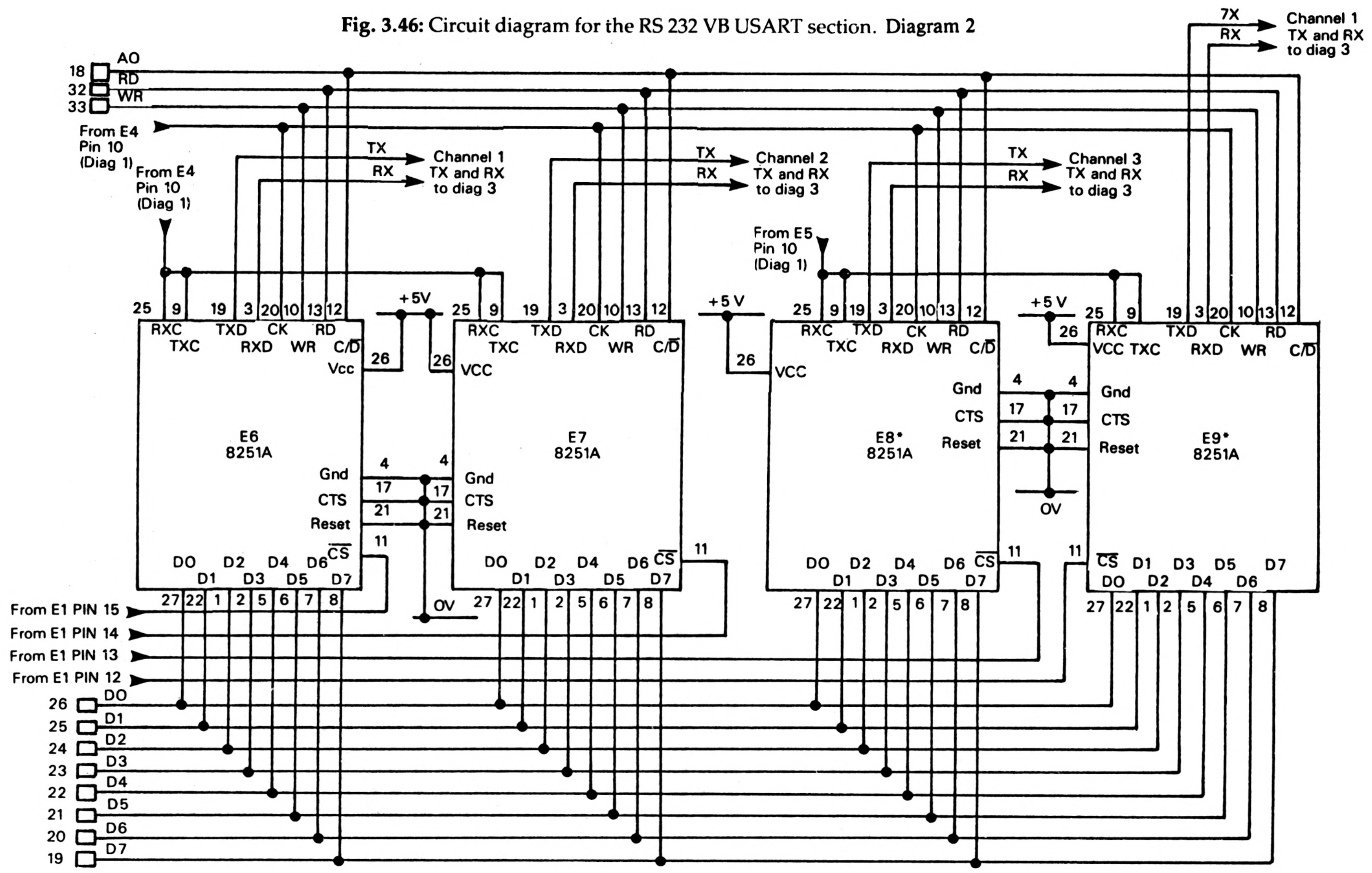
```
OUT &F9E8,&D
```

sets the baud rate generator output for channels one and two to 300 baud. It is actually a bit more complicated than that, because the Baud rate generator chip produces a frequency which is sixteen times the actual rate. This is to compensate for the fact that most UART chips always divide the baud clock they receive by sixteen. We can program the USART to do this too, so for all purposes– except 'scope testing and programming– we can ignore this little foible.

The master frequency for the baud rate generators is from the crystal XTAL 1. The first 4702 – E4 – uses the crystal and an on chip oscillator circuit to produce a 2.4576 Mhz clock signal This is used internally by E4 to produce the baud clocks, but it is also buffered and passed on to the second 4702 – E5. This saves the need to have two crystals and associated components. The selected baud rate clock emerges on pin 10 of each 4702. Chip E4 feeds USARTs one and two, whilst E5 feeds USARTs three and four. In the parts list for this project, given at the end of the hardware and software descriptions, there is a small checking chart to see which components you will need for any given number of channels.

Fig 3.46 shows the USARTs. These all have their D0–D7 lines connected to the data bus. The CS bar (Chip Select active low) input to each USART comes from the address decoder as just described. Each USART has its C/D bar line connected to address line A0. The C/D bar line decides whether the control register or the data register is accessed when the CS bar line is low. The receive and transmit clock signals are connected to baud rate generator E4 for USARTs one and two, and to E5 for USARTs three and four. No provision has been made for split speed working (that is, transmitting at one speed and

Fig. 3.46: Circuit diagram for the RS 232 VB USART section. Diagram 2



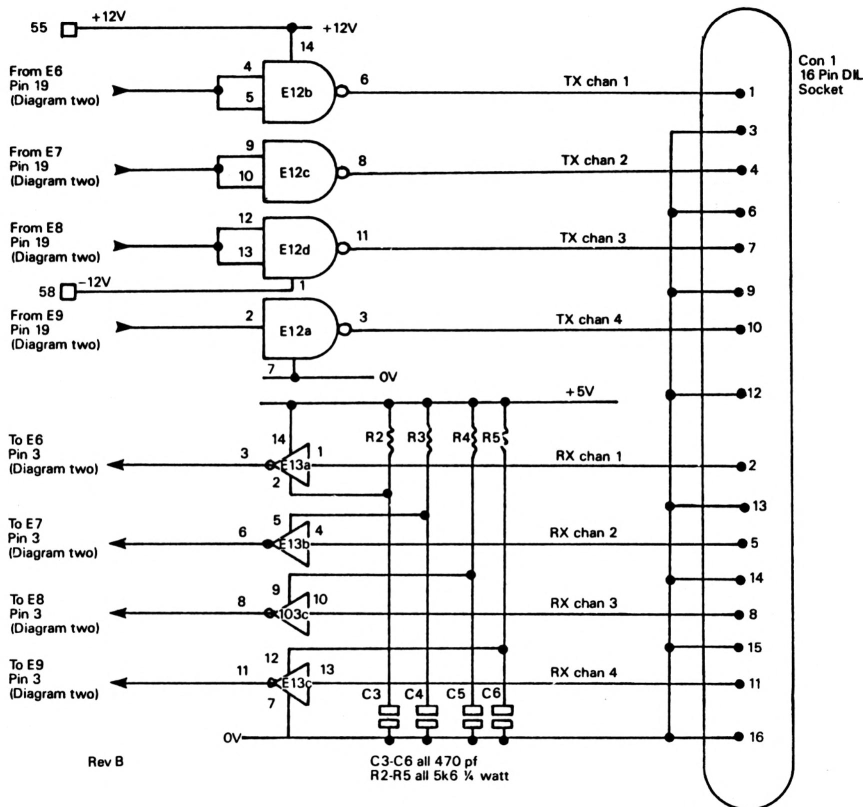


Fig. 3.47: 1488–1489 chips section of RS 232 VB circuitry.

receiving at another) as this is a comparatively rare requirement nowadays, and would have involved the inclusion of two more 4702s' and two more latches. Pin 20 on each USART is its master clock input. The technical specifications for the 8251A state that the frequency presented to this pin must be at least thirty times that presented at the transmit and receiver clock inputs. This is achieved here by connecting all the USART's master clock inputs to the baud rate generators master clock, of 2.4567 Mhz. All USARTs are connected to the read and write strobes, and their RESET inputs are grounded.

Fig 3.47 shows the RS 232 circuits. The two chips used are the ubiquitous 1488 and 1489 chips. The C/R network on each element of the 1489 (E13) are to keep its response frequency to a useful limit, thus avoiding any risk of UHF signals causing spurious receptions. In essence the 1488 takes TTL level signals in, and changes them to RS 232 levels (that is, plus and minus voltages for logic zero and one respectively). The 1489 does the opposite. All the send and

receive lines are brought to the offboard connector CON 1. After this you can really choose whatever method of connection suits you best, but I would suggest a short length of ribbon cable split up into four, then soldered onto four D type sockets.

That concludes the hardware description of the RS 232 version "B" interface. Now let us look at some software to test it, and some software to allow the use of a CPC as a serial terminal.

Software for the RS 232 version B interface

In order to use the interface some special software is needed. This is becuase there is no provision in the CPC firmware for driving a serial channel.

Before the description of the serial driver software,the reader should understand the operation of the 8251A USART. The USART has two registers. Register one is a data register, which bytes for transmission are written into, and where data received from the serial line can be read. The second is the control register, which actually has three functions. The first of these functions is to have written into it the mode byte. The mode byte can only be written into the control register after the 8251A has been fully reset, either by a pulse at its RESET pin, or by a programmed internal reset (see later). The mode byte selects the operational characteristics of the USART until it is next reset. The things selected are the data length in bits, parity settings, and the number of stop bits. The other thing which can be effected via the mode byte is a modification of the baud rate. This allows the two USARTs connected to the same 4702 baud rate generator chip to operate at different rates.

The layout of the mode byte is shown in figure 3.48. As you will see the baud rate clock can divided by: 1 (that is, left alone), by 16 (normally this will be selected since the 4702 actually supplies 16 times the selected rate), or by 64 – which allows a little scope for setting two different baud rates for USARTs connected to the same 4702. When programming the mode byte you should consult with the manual of the equipment you are connecting to, or its owner. As a general rule of thumb if you do not know what setting to use, try eight bits, no parity, and 2 stop bits. (This combination is sometimes called the DEC standard, due to its widespread use in serial transmission from Digital Equipment Corporation mini computers.) if this fails and nobody can tell you the settings, then I am afraid that trial and error is really the only way to find out if you have got it right!

The other two uses for the control register address are to read the 8251A status, and to exercise control over it. Fig. 3.49 shows the read image of the control register. we shall now look at the function of each bit:

Bit 7: DSR – Shows the state of pin 22 of the USART (Data Set Ready).

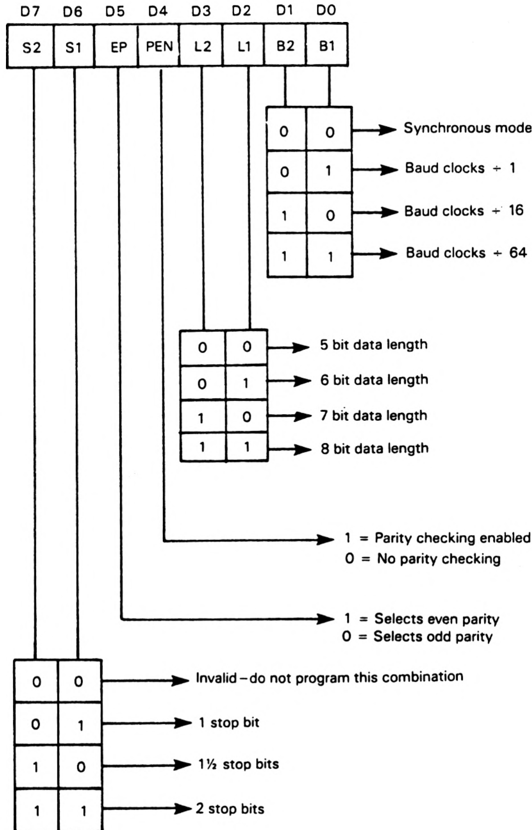


Fig. 3.48: Layout of mode byte for an 8251A USART chip.

Bit 6: Sync detect – used only in synchronous mode, therefore of no interest here.

Bit 5: Framing error. Set when the USART logic detects that the character it has received has the wrong number of stop bits, usually due to line noise or incorrect mode byte.

Bit 4: Overrun error. Set when a previously received character was not read by your program before the next incoming one replaced it – in other words, you missed one!

Bit 3: Parity error. The computed and received parity do not match. Usually due to line noise, unstable Baud clocks, or incorrect mode byte causing the sender of the data and the USART to use different parity checks.

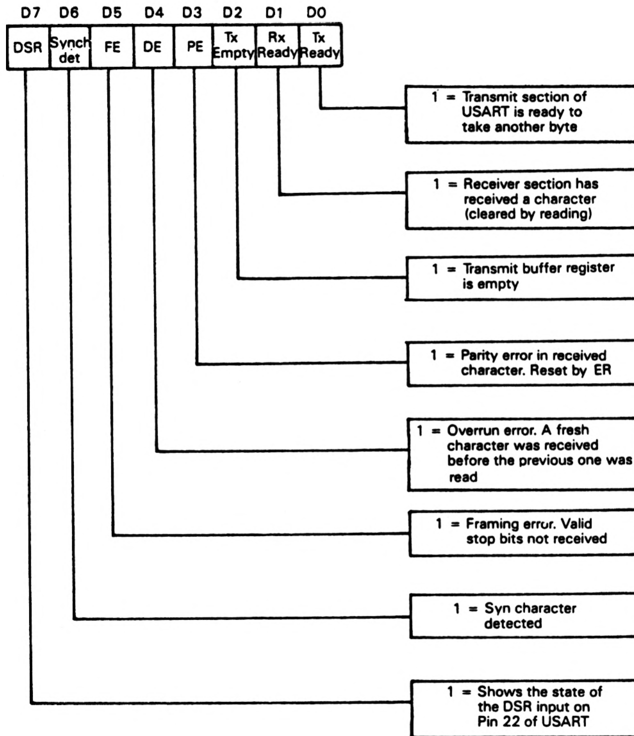


Fig. 3.49: Read image of 8251A control register.

Bit 2: TX empty. When this is a one then the USART transmitter buffer – which is two bytes deep – is empty.

Bit 1: RX ready. When set, this indicates that a character has been received, and is now available for reading from the data register.

Bit 0: TX ready. Shows that the transmitter buffer is ready to take another character for sending.

Now the bit assignments for the write image of the control register, shown in Fig. 3.50:

Bit 7: EH. Enable Hunt – used only in synchronous mode, not used here.

Bit 6: IR. Internal Reset. When you write a one into this bit the USART will be fully reset as if the RESET pin had been pulsed, and the device will take the next thing written into its control register as a mode byte.

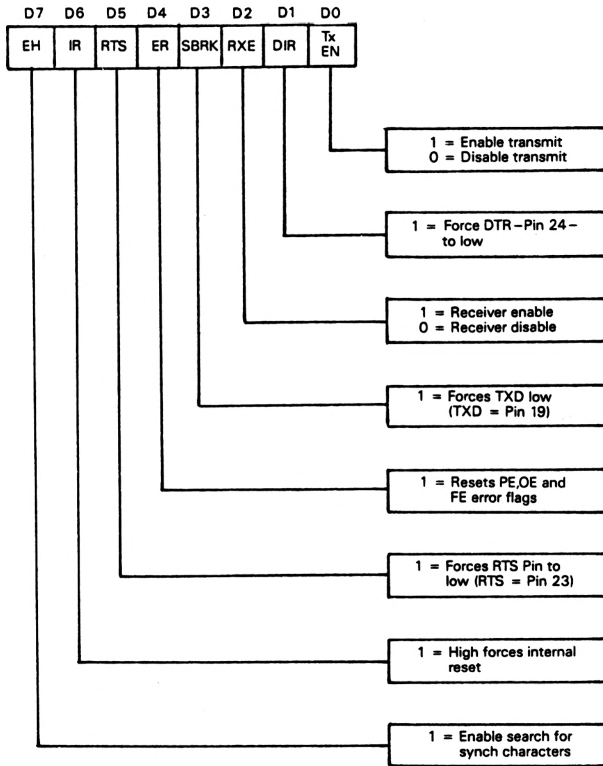


Fig. 3.50: Write image of the 8251A control register.

Bit 5: RTS. The inverse of the value written into this byte will appear on pin 23 of the USART. (if you write a zero the RTS pin will go high, and vice versa.)

Bit 4: ER. Error reset. This must be programmed after detection of one of the error flags being set. The fact that an error flag is set does not inhibit operation of the USART, but unless you reset the error flags, verification of incoming data will not be possible.

Bit 3: SBRK. Programming a one into this bit forces pin 19 low.

Bit 2: RXE. Receiver enable. Programming a one into this bit will enable the receiver section of the USART.

Bit 1: DTR. Writing a one into this bit will force pin 24, DTR low.

Bit 0: TX EN. Programming a one into this bit enables the transmitter section of the USART.

When initialising a USART after power on it is wise, though not essential, to perform a worst case initialisation. Thus ensuring total reset, and that the mode programmed has been set up correctly. This consists of sending the control register six bytes in succession. These bytes are (Hex values):

Byte 1: 00 —Three ASCII NUL characters
Byte 2: 00
Byte 3: 00
Byte 4: 40 —Program an internal reset
Byte 5: CE —(or whatever mode byte you need for your requirments) This selects 2 stop: No parity: 8 bits: Divide by 16
Byte 6: 15 —To reset error flags and enable both TX and RX sections of the USART.

If you want to do a minimal initialisation of the USART then leave out the first three bytes and start from byte 4.

When you have constructed the serial card you will need to test it. The best method of doing this is to connect the send line of each channel you have constructed, back to its receive line. This will allow you to send characters out and make sure that you can receive them back correctly. This will ensure not only the proper function of the USART and associated electronics, but also that the USART and microprocessor in the CPC are communicating with one another correctly.

Fig 3.51 is the listing of a program to do a long test of this kind, also known as a soak test. When you have looped the send and receives of the channel to be tested you should run this program. It displays the number of times that the USART error flags are found to be set. The program also shows how many data errors have occurred, which is the number of times a byte differed from the one which was sent. To be sure of the correct function of the serial channel, the program should be run for 30 minutes or more.

```
100    ' ** This is a soak test program for a selected channel on **
         ** the Serial I/O card. You first select which channel to**
         ** be tested, 0-3, then that channel is initialised and  **
110    ' ** tested. NOTE: You must loop the channel back to itself**
         ** IE connect the transmit wire to the receive wire.     **
         ** The program then sends every possible character over  **
120    ' ** the loop, and checks to see if it is received back OK.**
         ** The on screen error counters show how many errors of  **
         ** each possible type have occurred. The error types are **
130    ' ** Parity error, Framing error, overrun error, and data  **
         ** error. All but the last error are detected by the     **
         ** USART logic. The Data error means that the program    **
140    ' ** sent one character but received a different one.      **

150    DIM BD(16),EQUIV(16) ' BD array will hold textual representations
       of BAUD rates available. EQUIV will contain actual value to be placed
       in the BAUD select register for USART selected for test.
160    MODE 2: ON BREAK GOSUB 470: PRINT
       "This program does a continuous test of one of the serial I/O channels"
       PRINT "which are on the SIO board";STRING$(5,10)
170    USARTBASE=&F9E0: GOSUB 320:
```

```
        INPUT "Which channel to test (0 to 3)-default is zero";C:
180     IF C < 0 OR C > 3 THEN 170 ELSE PRINT: PRINT :
        USARTBASE=USARTBASE+(C*2)
190     INPUT "Select BAUD rate (Type enter for list)";B
200     IF B <= 0 THEN GOSUB 330: GOTO 190
210     FOR I=1 TO 16: IF BD(I) <> B THEN 230 ELSE IF C < 2 THEN
        OUT &F9E8,EQUIV(I) ELSE OUT &F9EC,EQUIV(I) ' Put the BAUD
        select value in the appropriate BAUD rate select register.
220     GOTO 240
230     NEXT I: MES$="Unknown BAUD rate": GOSUB 330: GOTO 190
240     CLS: FOR I=1 TO 3: OUT USARTBASE+1,0: NEXT I:
        OUT USARTBASE+1,&40: OUT USARTBASE+1,&FE:
        OUT USARTBASE+1,&15 'do worst case init of USART
250   '** Worst case init is to send 3 zero bytes, followed by an **
        ** Internal reset command, then mode word.(See accompanying**
        ** diagrams.) Then we reset the error flags and enable TX and RX **
260     GOSUB 360 ' Draw the display screen
270     T=TIME: P=0: OV=0: FR=0: PASS=0: EVERY 3000,1 GOSUB 450
      ' ** Remember when we began for run time display, zero counters **
        ** and initialise the minutes of runtime display interrupt.   **
280     FOR I=0 TO 255: WHILE (INP(USARTBASE+1) AND &1)=0: WEND:
        OUT USARTBASE,I: WHILE (INP(USARTBASE+1) AND &2)=0: WEND:
        IF INP(USARTBASE) <> I THEN DAT=DAT+1: GOSUB 420: GOTO 310
290     UER = (INP(USARTBASE+1) AND &38): IF UER AND &8 <> 0 THEN
        PAR=PAR+1 ELSE IF UER AND &10 <> 0 THEN OV=OV+1 ELSE IF UER
        AND &20 <> 0 THEN FR=FR+1
300     GOSUB 420: IF (INP(USARTBASE+1) AND &38) <> 0 THEN
        OUT (USARTBASE+1),&15 'Clear any error flags if set
310     NEXT I: PASS=PASS+1: GOTO 280
320     RESTORE: FOR I=1 TO 16: READ BD(I),EQUIV(I): NEXT: RETURN
330   '
         **      S/R to show what BAUD rates are available       **

340     CLS: LOCATE 10,1: PRINT "The BAUD rates are as follows"
        STRING$(2,10): FOR I=1 TO 16: PRINT BD(I): NEXT: PRINT:
        IF MES$<> "" THEN PRINT MES$+CHR$(7)"-";B: MES$=""
350     RETURN
360     LOCATE 20,1: PRINT "Serial I/O channel soak test":
        FOR I=39 TO 339 STEP 300: MOVE I,100:  DRAWR 230,0:
        DRAWR 0,230: DRAWR -230,0: DRAWR 0,-230: NEXT I
370     LOCATE 15,6: PRINT "Error log": LOCATE 7,10: PRINT "Parity errors=  0":
        LOCATE 7,12: PRINT "Overrun errors= 0": LOCATE 7,14:
        PRINT "Framing errors= 0": LOCATE 7,16: PRINT "Data errors="
380   ' ** The error counts are at horizontal position 27 on lines **
        ** 10 (Parity) 12 (Overrun) and 14 (Framing).              **
390     LOCATE 52,6: PRINT "Test details": LOCATE 45,8:  PRINT
        "Selected channel is";C: LOCATE 45,12: PRINT "BAUD rate set to";B:
        LOCATE 45,14: PRINT "Current test value ="
400     LOCATE 45,16: PRINT "Pass count =": LOCATE 45,18: PRINT
        "Run time (mins.) =": LOCATE 45,10: PRINT "Channel I/O address= ";
         HEX$(USARTBASE)
410     RETURN
420   '
        ** Status display update **

430     LOCATE 22,10: PRINT PAR: LOCATE 22,12: PRINT OV: LOCATE 22,14:
        PRINT FR: LOCATE 22,16: PRINT DAT: LOCATE 65,14: PRINT I;"  ":
        LOCATE 65,16: PRINT PASS
440     RETURN
450     L=POS(#0): R=VPOS(#0): LOCATE 65,18: PRINT INT(((TIME-T)/300)/60):
        LOCATE L,R: RETURN '** Small s/r to print the time EVERY minute
        - see line 1110. Remembers where cursor was and puts it back  **
460     DATA 19200,0,19200,1,50,2,75,3,134.5,4,200,5,600,6,2400,7,
              9600,8,4800,9,1800,&A,1200,&B,2400,&C,300,&D,150,&E,
              110,&F,-1
470     LOCATE 6,22: r=REMAIN (1): PRINT
        "Program aborted by ** BREAK **": END
```

Fig. 3.51: Soak test program in BASIC for the RS 232 VB four channel interface.

Now having looked at operating and testing the serial channels on the RS 232 version B interface, let us look at actually putting it to work.

The long running serial

Consider the case where you wish to use only one of the four channels, connected to a 300 baud MODEM line. This would be the case where you wanted to use your CPC to get on line to one of the many TV shows, dial in notice boards, or networks which provide such a facility. The BASIC program shown in Fig 3.52 will happily run channel zero at 300 baud. In this case the assumption is made that you will only use the channel at 300 baud, and that the channel in use is always zero. No notice is taken of any activity on the other channels – if fitted. As it says in the listing this is a no frills program, which just puts every incoming byte on the screen with no error checking or vetting to ensure that it is a printable character, beyond setting bit 7 of the character to zero. This means that if you set the data rate wrongly then lots of rubbishy characters will be sent to the CPC screen including some control characters, sometimes with rather strange results.

```
100  '** This is a no frills program to drive the CPC 464 as a serial
      ** VDU attached to a 300 BAUD comms line. To run it at more than
      ** 300 use the larger program    - found later.
200  USARTBASE=&F9E0: CLS: FOR I=1 TO 3: OUT USARTBASE+1,0: NEXT I:
     MODE 2: OUT USARTBASE+1,&40: OUT USARTBASE+1,&CE: OUT &F9E4,&D:
     OUT USARTBASE+1,&15 'do worst case init of USART and set 300 BAUD
300  '** Set USART 1 to divide its clocks by 16, to accomodate the
      ** 4702 chips output which is 16 times baud rate.
400  D$=INKEY$: IF D$="" THEN 600 ELSE  OUT &F9E0,(ASC(D$))
500  '** Line 200 sees if a key has been pressed, if not then it jumps
      ** to line 300. If so then it outputs the character
600  IF ((INP(&F9E1)) AND &2) =0 THEN 400 ELSE R=(INP(&F9E0)AND &7F):
     PRINT CHR$(R);: GOTO 400

700  END '** Line 300 sees if a character has been received by USART 1.
     **  If yes then it fetches it. NOTE: in this no frill version the
     **  received character is assumed ok, the error flags are not sampled.
```

Fig. 3.52: Minimal BASIC program for driving a single serial channel

The no frills program will be adequate for many applications of the serial I/O channels, but it has serious limitations for usages requiring faster baud rates greater then 300, greater flexibility, or multi channel operation. The serial driver program described in the next section will provide all these functions.

Serial driver software

Let us now look at how we can use the hardware as a multichannel VDU.

The program about to be presented was originally written entirely in BASIC. Unfortunately in spite of the Locomotive BASIC being one of the fastest interpreted BASICs around, the resulting program could not keep up with data rates above about 1200. So the program had to be partially re written, with the time critical sections being written in assembler generated machine code. Don't worry if you cannot make head nor tail of the assembly language

listings on the following pages, you can come back to them after you have read chapter 5, which forms an introduction to assembly language programming.

The serial driver program provides the following features and facilities:

1) VDU type operation on the selected channel (see later)
2) Limited message reception on the deselected channels.
3) Operation at up to 19200 Baud
4) A special set up mode where the following characteristics for any channel can be set:

   A) Baud rate
   B) Word length
   C) Number of stop bits
   D) Parity disable/enable and Odd/Even select

   Additionally the set up menu also displays the current settings and indicates when messages are pending on the deselected channels.

Because the software is envisaged to be used with many different systems no control code interpretation is carried out, this is not an emulation package for any particular type of VDU – though I think it could easily be developed into such a thing. Becuase specialised control codes are not obeyed, the host system should be made to treat the CPC as a dumb terminal.

The serial driver is listed in fig 3.53. This is in BASIC with embedded machine code in DATA statements. (the technique for adding a machine code section to a BASIC program is explained in chapter 5.) The driving instructions for the program are simplicity itself. You first load the program from tape or disk, then you RUN it. It comes up with a blank screen with only a cursor –as per most commercially available VDUs. Then if you do not wish to change from the default settings (listed below) you establish a connection to your host computer and then use the CPC just as you would a normal VDU.

To enter the set–up menu you press either CTRL/P or the CLR key located above the large ENTER button.

If you enter the program as shown in the listing of Fig 3.53. the default settings when you first run the program are as follows:

Baud rate =9600   Stop bits =2   Parity =None
Divide factor =16   Word length =8   Characters pending =NO

Becuase we can only display one full size screen at a time (unless we went for splitting the screen into four very small windows) we need to ensure that data from only one channel at a time will be viewed. This channel is referred to as the selected channel, and it can be set to be any channel from zero to three by entering the set–up menu. On the menu screen the channel settings are all shown and the selected channel is marked.

```
500     '**       Serial driver version 4F:   April 1985

1000    '** This is the all singing/dancing 4 channel RS232 version B
         ** serial interface driver.  It is written in a mixture of
         ** BASIC and assembler.
4000    '** For full details of memory usage see the book text, but this
         ** is briefly as follows: BASIC's HIMEM is set to be at Hex 57FF
         ** The buffers are located from Hex 6000 to Hex 63FF: USARTINIT
5000    '** is loaded at Hex 6440: There are specific memory locations
         ** which are used for intercommunication between the BASIC
         ** and m/c section. The program will not co-reside in memory.
7000    '**             Begin

7005    MEMORY &57FF: MODE 1: LOCATE 10,5: DIM BAUD$(16): BORDER 5,10:
        SPEED KEY 15,1:  PRINT "Initialising serial I/O: Please wait":
        PRINT
7010    WHILE R$ <> "USARTINIT": READ R$: WEND: FOR M=&6440 TO &64C0:
        READ TMP: POKE M,TMP: NEXT M: PRINT "USARTINIT    installed"
       '** Poke USARTINIT into memory at Hex 6440
7030    BORDER 26: RESTORE: WHILE R$ <> "MACCODE": READ R$: WEND:
        FOR M=&6500 TO &65C0: READ TMP: POKE M,TMP: NEXT:
        PRINT "Machine code installed": INK 2,24,10: SPEED INK 10,20
8010    RESTORE: FOR T=&6400 TO &6413: READ TMP: POKE T,TMP:
        NEXT '** This transfers the default settings in the DATA
        ** statements to the params buffer used by the USARTINIT routine.
8020    CALL &6440: PRINT "Defaults     installed": PRINT: PEN #0,2: PRINT
        "Press CTRL/P or CLR to enter set up mode": PEN #0,1
8030    FOR I=26 TO 1 STEP -1: FOR P=1 TO 75: BORDER I: NEXT: NEXT
       '** DO some pretties while user reads the info.
8040    POKE &6003,&FF: POKE &6103,&FF: POKE &6203,&FF: POKE &6303,&FF
       '** Init the channel data buffers, with EOB characters.
8060    SELCHAN=0: POKE &6414,0 '** select channel 0 to use first.
8070    MODE 2: CALL &6500 '** go out to the KERNEL code
10000  '** This routine prints up the settings of the currently
        ** selected channel, and also prints up the error
        ** counters for all channels. It allows a new channel
10010  '** to become the selected channel, and indicates if
        ** any unviewed data has been received on the deselected
        ** channels.
10015   OLDSEL=SELCHAN: R$=CHR$(19): FOR I=0 TO 3: SELCHAN=I:
        GOSUB 11020: NEXT I: SELCHAN=OLDSEL
       '** Send an XOFF to every channel while we're in menu
10020   BORDER 5: CLG: MOVE 1,1: FOR I=100 TO 300 STEP 100: MOVE 1,I:
        DRAWR 600,1: MOVER 0,1: DRAWR -600,1 :NEXT: FOR I=0 TO 3:
        LOCATE 1,2+(I*6)
10030   PRINT CHR$(24);" Channel";I;CHR$(24);
        "       ERRORS: Parity=     : Framing=     : Overrun=     ":
        IF I=SELCHAN THEN PRINT: PRINT "* Selected *"
10040   LOCATE 17,4+(I*6): PRINT
        "SETTINGS:  Speed=       : Bits=       : Parity=": PRINT TAB(18);
       "       Stop bits=      : Received characters pending = "
10045   NEXT
10050  '** Now having printed up the titles of all the data we now
        ** have to print up the item values.
10055   IF BAUD$(0)="" THEN GOSUB 42000
10060   FOR I=0 TO 3: LOCATE 35,(I*6)+2: PRINT PEEK(&6000+(I*&100)):
        LOCATE 49,(I*6)+2: PRINT PEEK(&6000+(I*&100)):
        LOCATE 65,(I*6)+2: PRINT PEEK(&6000+(I*&100))
10070  '** The line above prints out the error counters.
10080   LOCATE 35,(I*6)+4: PRINT BAUD$(PEEK(&6400+(I*5))):
        LOCATE 47,(I*6)+4: PRINT 5+(PEEK(&6402+(I*5))):
        LOCATE 65,(I*6)+4
10090   IF (PEEK(&6403+(I*5)) AND 1)=0 THEN PRINT "No" ELSE
        PRINT "Yes- ";: IF (PEEK(&6403+(I*5)) AND 2)<>0 THEN PRINT
        "EVEN" ELSE PRINT "ODD"
10100   LOCATE 34,(I*6)+5: R=(PEEK(&6404+(I*5))): IF R=3 THEN PRINT
        2 ELSE PRINT R
10105   LOCATE 72,(I*6)+5: IF PEEK(&6003+(I*&100)) =255 THEN PRINT
```

```
        "NO" ELSE PRINT "YES"
10110   NEXT: BORDER 1: LOCATE 1,25: R$="": PRINT CHR$(24);
        " Type R to return to VDU or S to change the set ups ";
        CHR$(24);: WHILE R$ <> "R" AND R$ <> "S"
10120   R$=UPPER$(INKEY$): WEND: IF R$="R" THEN GOTO 10500
       '** If R was chosen then we return to VDU: otherwise on we
        ** go into the changes menu
10125  '**        This section allows users to change any set up        **
10130   CLS: TITLE$="Serial channel set-up changes": GOSUB 41000
10140   LOCATE 4,5: MES$="The currently selected channel is":
        PRINT CHR$(20);MES$; SELCHAN: LOCATE 4,7:
        INPUT "Number of channel to select - just ENTER for no change";R$:
10150   IF R$="" THEN GOTO 10160 ELSE IF VAL(R$) > 3 OR VAL(R$) < 0 THEN
        GOTO 10140 ELSE SELCHAN =INT(VAL(R$)): POKE &6414,SELCHAN:
        LOCATE 4,5: PRINT CHR$(20);MES$;SELCHAN '** New selchan updat mem
        table
10160   LOCATE 4,7: BASE=(&6400+(SELCHAN*5)): PRINT "Parity ";CHR$(20);:
        IF (PEEK(BASE+3) AND 1)=0 THEN PRINT "<No>"; ELSE
        PRINT "<Yes>";
10170   INPUT R$: IF UPPER$(LEFT$(R$,1))="Y" OR R$="" THEN 10180
        ELSE POKE BASE+3,0: GOTO 10190
10180   LOCATE 4,7: PRINT CHR$(20);: IF (PEEK(BASE+3) AND 1)=0 AND UPPER$
        (LEFT$(R$,1))  <>"Y" THEN 10190 ELSE INPUT "Odd or Even ";R$:
        IF UPPER$(LEFT$(R$,1))="E" THEN POKE BASE+3,3 ELSE POKE BASE+3,1
10190   LOCATE 4,7: PRINT CHR$(20);"Number of bits per character <";
        (5+(PEEK(BASE+2)));">";: INPUT R$: R=VAL(R$): IF R=0 THEN
        R=(5+(PEEK(BASE+2))) ELSE IF R >8 OR R<5 THEN 10190
10200   POKE BASE+2,R-5
10210   LOCATE 4,7: PRINT CHR$(20);"BAUD RATE <";BAUD$(PEEK(BASE));">";:
        INPUT R$: IF R$="" THEN GOTO 10230 ELSE FOR I=0 TO 15:
        IF R$=BAUD$(I) THEN 10220 ELSE NEXT I: GOTO 10210
10220   IF PEEK(&6414) > 1 THEN 10225 ELSE POKE &6400,I: POKE &6405,I:
        GOTO 10230
10225   POKE &640A,I: POKE &640F,I
10230   LOCATE 4,7: PRINT CHR$(20);"Number of stop bits: 1 or 2 <2>";:
        INPUT R$: IF R$="" THEN 10240 ELSE IF VAL(R$) <=0 THEN 10230 ELSE
        IF VAL(R$) > 2 THEN 10230
10235   IF INT(VAL(R$))=1 THEN POKE BASE+4,1 ELSE IF INT(VAL(R$))=2
        THEN POKE BASE+4,3 ELSE 10230
10240   CALL &6440: GOTO 10000 '** That concludes the set-ups. To keep
        it simple we only allow 1 or 2 stop bits. Now that the set ups
        are done we install the changes and restart the display rtne.
10500   CLS: SELCHAN=OLDSEL: R$=CHR$(17): FOR I=0 TO 3: SELCHAN=I:
        GOSUB 11020: NEXT I: SELCHAN=OLDSEL: GOTO 8070
          '** Tell everyone we are back with an XON, and Finish.
11000  '** Routine to send a character typed at the CPC 464 keyboard
        ** out on the selected channel. If character typed is = 16
        ** then we do the menu subroutine. In half DUP we echo it too.
11020   WHILE INP(&F8E1+(SELCHAN*2)) AND &2 = 0: WEND:
        OUT (&F8E0+(SELCHAN*2)),ASC(R$)
11090   RETURN
20000  '** The following DATA statements represent the parameters
        ** for each of the 4 serial channels. These parameters are
20010  '** fully defined in the text. The values shown in this list
        ** will set these parameters to the following: 9600 Baud:
        ** Divide TX and RX clocks by 16: 8 bit transfers: Parity
20020  '** disabled: 2 stop bits. DATA statement 20030 sets channel
        ** 1: DATA statement 20040 sets channel 2 and so on...

20030   DATA 8,2,3,0,3
20040   DATA 8,2,3,0,3
20050   DATA 8,2,3,0,3
20060   DATA 8,2,3,0,3
20100   DATA "BAUD RATES"
20110   DATA 19200,19200,50,75,134.5,200,600,2400,9600,4800,1800,
            1200,2400,300,150,110
30999  DATA "USARTINIT"
31000  DATA 221 , 229 , 213 , 229 , 245 , 197 , 46 , 0 , 62 , 3
```

```
31010  DATA 205 , 176 , 100 , 46 , 64 , 62 , 1 , 205 , 176 , 100
31020  DATA 221 , 33 , 0 , 100 , 17 , 5 , 0 , 1 , 232 , 249
31030  DATA 221 , 126 , 0 , 237 , 121 , 1 , 236 , 249 , 221 , 126
31040  DATA 10 , 237 , 121 , 1 , 225 , 249 , 46 , 4 , 221 , 126
31050  DATA 1 , 221 , 102 , 2 , 203 , 4 , 203 , 4 , 180 , 221
31060  DATA 102 , 3 , 203 , 4 , 203 , 4 , 203 , 4 , 203 , 4
31070  DATA 180 , 221 , 102 , 4 , 203 , 4 , 203 , 4 , 203 , 4
31080  DATA 203 , 4 , 203 , 4 , 203 , 4 , 180 , 237 , 121 , 45
31090  DATA 40 , 6 , 3 , 3 , 221 , 25 , 24 , 206 , 46 , 21
31100  DATA 62 , 1 , 205 , 176 , 100 , 193 , 241 , 225 , 209 , 221
31110  DATA 225 , 201 , 30 , 4 , 1 , 225 , 249 , 87 , 237 , 105
31120  DATA 21 , 32 , 251 , 29 , 200 , 3 , 3 , 24 , 244 , 0
31999  DATA "MACCODE"
32000  DATA 245 , 213 , 197 , 229 , 205 , 129 , 187 , 205 , 98 , 101
32010  DATA 58 , 20 , 100 , 246 , 96 , 103 , 46 , 3 , 203 , 126
32020  DATA 32 , 9 , 126 , 205 , 90 , 187 , 205 , 72 , 101 , 24
32030  DATA 232 , 205 , 9 , 187 , 48 , 227 , 254 , 16 , 32 , 10
32040  DATA 205 , 132 , 187 , 225 , 193 , 209 , 241 , 201 , 24 , 206
32050  DATA 95 , 1 , 225 , 249 , 58 , 20 , 100 , 203 , 39 , 129
32060  DATA 79 , 237 , 120 , 230 , 1 , 40 , 250 , 13 , 237 , 89
32070  DATA 24 , 191 , 229 , 245 , 58 , 20 , 100 , 230 , 3 , 246
32080  DATA 96 , 103 , 46 , 3 , 203 , 126 , 32 , 7 , 35 , 126
32090  DATA 43 , 119 , 35 , 24 , 245 , 225 , 241 , 201 , 245 , 197
32100  DATA 229 , 213 , 1 , 225 , 249 , 33 , 0 , 96 , 22 , 4
32110  DATA 237 , 120 , 11 , 254 , 255 , 40 , 36 , 203 , 79 , 40
32120  DATA 32 , 230 , 56 , 32 , 48 , 125 , 198 , 3 , 111 , 203
32130  DATA 126 , 32 , 3 , 35 , 24 , 249 , 237 , 120 , 230 , 127
32140  DATA 119 , 35 , 54 , 255 , 62 , 255 , 189 , 32 , 4 , 46
32150  DATA 3 , 54 , 255 , 21 , 32 , 5 , 209 , 225 , 193 , 241
32160  DATA 201 , 46 , 0 , 124 , 198 , 1 , 103 , 121 , 198 , 3
32170  DATA 79 , 24 , 193 , 203 , 95 , 40 , 1 , 52 , 35 , 203
32180  DATA 103 , 40 , 1 , 52 , 35 , 203 , 111 , 52 , 40 , 1
32190  DATA 52 , 62 , 21 , 3 , 237 , 121 , 11 , 24 , 210 , 0
32200  DATA 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
41000  '** Routine to print TITLE$ in a box at the top line of screen
41010  SYMBOL 255,&E0,&E0,&E0,&E0,&E0,&E0,&E0,&E0: SYMBOL 254,&7,&7,&7,&7,&7,&7
       &7,&7
41020  TITLE$=CHR$(255)+" "+TITLE$+" "+ CHR$(254)
41030  LEFT%=(40-(LEN(TITLE$)/2)): LOCATE LEFT%,2: PRINT TITLE$
41040  MOVE (8*(LEFT%-1)),384
41050  DRAW 8*(LEFT%+ (LEN(TITLE$)-1))-1,384
41060  MOVE XPOS,(YPOS-18): DRAW 8*(LEFT%-1),YPOS
41070  RETURN
42000  '** This routine sets up the BAUD$ array to contain
       ** the textual representations of the Baud rates
42010  RESTORE: WHILE R$ <> "BAUD RATES": READ R$: WEND:
       FOR I=0 TO 15: READ BAUD$(I): NEXT: RETURN
```

Fig. 3.53: Four channel interface driver.

The three deselected channels can still receive up to 253 characters from whatever system or device they are connected to and upon a channel becoming the selected channel, any pending messages will be displayed. Additionally the set up menu indicates when any characters are pending for a channel.

Changes made in the set up screen are implemented on exiting to the menu screen again.

Let us now take an overview of the program:

The program uses fixed address locations in memory as buffers, and to house the machine code modules. These are:

HEX 6000–60FF = Received data buffer for channel 0
HEX 6100–61FF = Received data buffer for channel 1
HEX 6200–62FF = Received data buffer for channel 2
HEX 6300–63FF = Received data buffer for channel 3
HEX 6414 = Number of selected channel.
HEX 6440–64C0 = USARTINIT machine code module.
HEX 6500–65C0 = MACCODE machine code module

Each channel has a received data buffer, whose length is 256 bytes. Only 253 bytes are used for character storage however, the first three bytes are used as error counters for the channel. These are allocated as follows:

First byte (eg for channel 0 address Hex 6000)= Parity errors count. Second byte= Framing errors count. Third byte= Overrun errors.

The fourth byte onwards is the actual data buffer. Becuase this software is only intended to handle ASCII codes– which use only the lower seven bits of each byte– we can use the top bit as a flag to show the buffer handling software where the end of buffer is. So if a byte in the data section of the buffer has bit seven set, that means it is the last character in the buffer.

NOTE: If whilst a channel is deselected it receives more than 253 characters, then the 254th character will be placed in the first byte of the buffer again, and the previous contents of the buffer will be lost. This is because the off screen receive feature is only intended to be for message reception, not for normal data reception.

The Parameters buffer consists of the default parameters which the program uses in the mode definition byte to each USART during the run of the USARTINIT routine. The parameters buffer is located at address Hex 6400 to address Hex 6413. There are 5 parameters asscotiated with each channel, which are set from values in the parameters buffer. In order these are:

1) Value to be placed in the baud register for that channel
2) Number of stop bits.
3) Division factor to apply to the USART – (see USART mode byte)
4) Word length
5) Parity settings
6) Number of stop bits

All these values except the one for the Baud rate are used by USARTINIT to construct a mode byte to be placed into the USART after reset. Refer to fig. 3.48 for details of the mode byte. Let us look at this area in a little more detail. Line 20030 of the program reads as follows:

```
20030 DATA 8,2,3,0,3
```

This sets the default parameters for channel zero. The eight is the value to be placed in the Baud select register to select 9600 Baud. The two is used by

USARTINIT as the lower two bits of the mode word to select divide by sixteen. The first three is to select 8 data bits. The zero disables parity checking. And the second three selects 2 stop bits. Reference to figure 3.48 will make this scheme obvious.

Line 20030 sets channel zero's parameters, Line 20040 sets channel ones's and so forth.

After installing the machine code program and setting up the default parameters buffer, the BASIC section calls USARTINIT to initialise the USARTS. The assembly listing for the USARTINIT program is given in Figure 3.54. Once again readers not familiar with assembler are urged to skip this listing now and return to it after reading chapter 5.

```
Pass 1 errors: 00

                        100 ;     **  USART INITIALISATION PROGRAM
                        200 ;
                        300 ; ** This software is for the 4 channel RS232 interface,
                        400 ; ** details of which are in the book"
                        500 ;
                        600 ; ** This software will initialise the 4 channels - and
                        700 ; ** uses parameters supplied by the BASIC serial driver.
                        800 ;
                        900 ;
6440                   1000        ORG  #6440        ; Define where we are running
6440                   1100 INITUS ENT  $
6440   DDE5            1200        PUSH IX
6442   D5              1300        PUSH DE
6443   E5              1400        PUSH HL
6444   F5              1500        PUSH AF
6445   C5              1600        PUSH BC ; Make some room
6446   2E00            1700        LD   L,#0      ; L is the character to send
6448   3E03            1800        LD   A,#3      ; A is how many times to send it
644A   CDB064          1900        CALL SENDIT ; Call the sender routine
644D   2E40            2000        LD   L,#40 ; Now send Hex 40 just once
644F   3E01            2100        LD   A,#1  ; which will reset each USART
6451   CDB064          2200        CALL SENDIT ; and send the reset command
                       2300 ;
                       2400 ; Now having got each USART to the stage where it is
                       2500 ; expecting a mode byte (which is described in the text)
                       2600 ; we now set about putting the parameters for each channel
                       2700 ; - which have been put into a small buffer starting at
                       2800 ; memory address Hex 6400 by the main BASIC program - into
                       2900 ; each of the USART's.
                       3000 ;
6454   DD210064        3100 MODSET LD   IX,#6400    ; Set IX to base of param buffer.
6458   110500          3200        LD   DE,#5       ; DE=update value for stepping IX
645B   01E8F9          3300        LD   BC,#F9E8    ; Load chans 0 and 1 BAUD rate reg
645E   DD7E00          3400        LD   A,(IX)      ; Load the BAUD rate value
6461   ED79            3500        OUT  (C),A       ; Put it into the register.
6463   01ECF9          3600        LD   BC,#F9EC    ; Ld chans 2 & 3 Baud rate address
6466   DD7E0A          3700        LD   A,(IX+10)   ; Get the rate value
6469   ED79            3800        OUT  (C),A       ; And put into register
646B   01E1F9          3900 SETUP  LD   BC,#F9E1    ; Now the rest of the params
646E   2E04            4000        LD   L,4         ; Set up four loops
6470   DD7E01          4100 SETLOP LD   A,(IX+1)    ; A=divide factor for this chan.
6473   DD6602          4200        LD   H,(IX+2)    ; H=numb bits which we must
6476   CB04            4300        RLC  H           ; move left by 2 bits
6478   CB04            4400        RLC  H
```

```
647A  B4        4500         OR   H          ; Set result into A
647B  DD6603    4600         LD   H,(IX+3)   ; Get parity enable & odd even
647E  CB04      4700         RLC  H          ; Move it up by 4 bits
6480  CB04      4800         RLC  H
6482  CB04      4900         RLC  H
6484  CB04      5000         RLC  H
6486  B4        5100         OR   H         ; Set parity selection into A
6487  DD6604    5200         LD   H,(IX+4)  ; Get the stop bits setting
648A  CB04      5300         RLC  H         ; move it left six times
648C  CB04      5400         RLC  H
648E  CB04      5500         RLC  H
6490  CB04      5600         RLC  H
6492  CB04      5700         RLC  H
6494  CB04      5800         RLC  H         ; And when that is done set it
6496  B4        5900         OR   H         ; into A.   A now = mode instruction
6497  ED79      6000         OUT  (C),A     ; so output it to the current USART
6499  2D        6100         DEC  L         ; Decrement L to see if all the
649A  2806      6200         JR   Z,ENALL   ; USART's are done - jump if so
649C  03        6300         INC  BC        ; Make BC point to next USART
649D  03        6400         INC  BC        ; register.
649E  DD19      6500         ADD  IX,DE     ; Move IX to next entry in buff
64A0  18CE      6600         JR   SETLOP      ; And do the loop again
                6700 ;
                6800 ;       We have now initialsed all USARTs. We now need to
                6900 ;  enable them all. We do this using SENDIT again.
64A2  2E15      7000 ENALL   LD   L,#15    ; We send Hex 15 to the USART
64A4  3E01      7100         LD   A,#1     ; and we only want to send it once
64A6  CDB064    7200         CALL SENDIT   ; Call sendit
                7300 ;
64A9  C1        7400         POP  BC
64AA  F1        7500         POP  AF
64AB  E1        7600         POP  HL
64AC  D1        7700         POP  DE
64AD  DDE1      7800         POP  IX ; Restore entry registers
                7900 ; Now the USARTs are ready to use
64AF  C9        8000         RET
                8100
                8200 ; End of USARTINIT.
                8300
                8400
                8500 ; This is the SENDIT routine. It sends the character it
                8600 ; receives in L to the four USARTs control registers n
                8700 ; times - where n is the value of A. It also uses DE.
                8800 ; On exit, A,BC,D,E register corrupt
64B0  1E04      8900 SENDIT LD    E,#4   ; Set up for four USARTs
64B2  01E1F9    9000        LD    BC,#F9E1 ; Make BC = 1st USART stat register
64B5  57        9100 SAV000 LD    D,A    ; Save count in D
64B6  ED69      9200 SAV001 OUT   (C),L ; output the character
64B8  15        9300        DEC   D      ; decrement loop count
64B9  20FB      9400        JR    NZ,SAV001 ; If not done then loop
64BB  1D        9500        DEC   E      ; see if all USARTS are done
64BC  C8        9600        RET   Z      ; return if so
64BD  03        9700        INC   BC     ; if not then make BC point at the
64BE  03        9800        INC   BC     ; next USART control reg.
64BF  18F4      9900        JR    SAV000 ; and go round again

Pass 2 errors: 00

ENALL  64A2  INITUS 6440  MODSET 6454  SAV000 64B5
SAV001 64B6  SENDIT 64B0  SETLOP 6470  SETUP  646B


Table used:   115  from   522
Executes: 25664
```

Fig. 3.54: Assembly listing for USARTINIT.

When the USARTINIT has completed control returns to the BASIC section, which immediately calls the main machine code module MACCODE. MACCODE actually consists of four routines. These are called KERNEL, INCOME, SEND and SHUFFLE. We shall look at these four routines in a little more detail shortly.

In broad outline the main module of the program continually scans the installed USART chips, to see if they have received any characters. If they have then the error flags are checked, with the error counters for that channel being updated as appropriate if any error flags are set. If the character is good then it is placed in memory, in the next available byte in the channels received data buffer. The other continuous function performed by the program is to call a CPC firmware routine to see if any characters have been typed at the keyboard. If any have then they are sent to the USART of the currently selected channel, presuming that it is ready to transmit a character. The only exception to this is if the character typed was CTRL/P or CLR (both of which generate the code 16), in which case a return to BASIC is effected.

When the machine code section returns control to BASIC then the flow drops through to the menu set up routine. This prints up the settings for all channels, and you have then to type "R" to Return to the machine code section and to carry on using the CPC as a VDU. You may also type "S" to go to the detailed set up questions. These are self explanatory when you run the program, all values input as new channel parameters are fully checked so that no bad values can be set. After going through a question and answer session about the new settings, the BASIC section then places the new values into the parameters buffer, then it calls USARTINIT to implement any changes that may have been made. Then it goes back to display the menu again. During the time that the program is in menu or set ups, an XOFF character is sent to all channels so that no characters are sent and therefore lost. (for a description of the meanings of XON and XOFF see section 3.2 – project one.)

There are a couple of other routines in the BASIC section of the program. These are routines to do with good screen layout or giving information. The remarks on the listing explain their purpose.

```
Hisoft GENA3 Assembler.

Pass 1 errors: 00

                   100 ;
                   200 ;
                   210 ;        **   Serial driver module MACCODE - Version 4H
                   211 ;
                   300 ;        * The machine code section of the 4 channel
                   400 ;        * serial interface driver. It is described in the
                   500 ;        * text of the book.
                   600 ;
                   700 ;        * There are four routines KERNEL,INCOME,SEND, and
                   800 ;        * SHUFFL. Each routine is described in the text c
                   900 ;        * the book
                  1000
6500              1100         ORG  #6500 ; We start just after USARTINIT
```

```
6500             1200         ENT  $
6500  F5         1300 KERNEL PUSH AF ; Do some pushes
6501  D5         1400        PUSH DE
6502  C5         1500        PUSH BC
6503  E5         1600        PUSH HL
6504  CD81BB     1700        CALL #BB81 ; Turn the cursor on please
6507  CD6265     1800 K1     CALL INCOME     ; Go do the incoming chars checker
650A  3A1464     1900        LD   A,(#6414) ; Get No. of currently selctd chan
650D  F660       2000        OR   #60
650F  67         2100        LD   H,A       ; Cvert it to buffer address in HL
6510  2E03       2200        LD   L,#03     ; Buffs start at 6x03 (see text)
6512  CB7E       2300        BIT  7,(HL)    ; EOB marker on front of buffr?
6514  2009       2400        JR   NZ,KEYCHK ; Jump if so
6516  7E         2500        LD   A,(HL)    ; else get & print the character
6517  CD5ABB     2600        CALL #BB5A
651A  CD4865     2700        CALL SHUFFLE   ; call shuffle the buffer up
651D  18E8       2800        JR   K1        ; and then loop round again
651F  CD09BB     2900 KEYCHK CALL #BB09     ; Call the KM READ CHAR s/r
6522  30E3       3000        JR   NC,K1     ; Loop again if none pressed
6524  FE10       3100        CP   #10       ; else see if CTRL/P
6526  200A       3200        JR   NZ,SEND   ; If not then SEND char
6528  CD84BB     3300        CALL #BB84 ; Turn our cursor off please
652B  E1         3400        POP  HL
652C  C1         3500        POP  BC
652D  D1         3600        POP  DE
652E  F1         3700        POP  AF  ; Else its time to go back to BASIC
652F  C9         3800        RET  ; so goodbye - for now
6530  18CE       3900        JR   KERNEL ; With a run again option
                 4000
                 4100 ;  *          Send routine.
                 4200 ;  ** Send the character found in A out to the
                 4300 ;  ** selected channel USART.
                 4400 ;
6532  5F         4500 SEND   LD   E,A       ; Save the typed character
6533  01E1F9     4600        LD   BC,#F9E1  ; Load address of 1st USART stat.
6536  3A1464     4700        LD   A,(#6414) ; Get selctd chan No.
6539  CB27       4800        SLA  A         ; Double it
653B  81         4900        ADD  A,C       ; Add it to low byte of address
                 5000 ; This means that if for example the selected channel
                 5100 ; were 1. After doubling it becomes 2, and when BC has
                 5200 ; had this added, BC will contain F9E3. Ie stat reg addr
                 5300 ; for channel 1 - see text.
653C  4F         5400        LD   C,A   ; Put final value into C

653D  ED78       5500 SEND01 IN   A,(C) ;
653F  E601       5600        AND  #1     ; Get STAT reg value - see if TX RDY set
6541  28FA       5700        JR   Z,SEND01 ; Jump if not
6543  0D         5800        DEC  C      ; Decrement C to point at DAT reg.
6544  ED59       5900        OUT  (C),E ; Output the character which was typed.
6546  18BF       6000        JR   K1     ; and then go back to KERNEL
                 6100
                 6200 ; **                    SHUFFLE
                 6300 ; ** This routine moves the contents of the serial channel
                 6400 ; ** buffer up by one, ie it deletes the character at the
                 6500 ; ** head of the buffer and moves the remaining contents
                 6600 ; ** forward one place towards the top of the buffer.
                 6700 ; ** The channel number is found in absolute memory
                 6800 ; ** location Hex 6414.
                 6900 ;
                 7000 ; ** This is a module of the 4 channel serial I/F dvr.
                 7100
                 7200
6548  E5         7300 SHUFFL PUSH HL
6549  F5         7400        PUSH AF
654A  3A1464     7500        LD   A,(#6414); Get the channel number
654D  E603       7600        AND  #3        ; A=channel number: Limit 0-3
654F  F660       7700        OR   #60       ; and make it six something
6551  67         7800        LD   H,A       ; Mult Hex 100 to make buffer addr
```

```
6552  2E03      7900         LD    L,03       ; and make L=pointer address low.
6554  CB7E      8000  SHUF01 BIT   7,(HL)     ; see if its EOB (bit seven set)
6556  2007      8100         JR    NZ,GOUT    ; If it is then we've done so jump
6558  23        8200         INC   HL
6559  7E        8300         LD    A,(HL)     ; Get next byte
655A  2B        8400         DEC   HL         ; Re-adjust HL
655B  77        8500         LD    (HL),A     ; and move it up one
655C  23        8600         INC   HL         ; Inc pointer
655D  18F5      8700         JR    SHUF01     ; and go again
655F  E1        8800  GOUT   POP   HL
6560  F1        8900         POP   AF
6561  C9        9000         RET   ; And goodnight my brave boys
                9100
                9200
                9300
                9400  ;        * This is the INCOME routine. It scans the USART
                9500  ;        * chips to see if any incoming characters have
                9600  ;        * been received, and updates the error counters
                9700  ;        * and buffers incoming data in one of 4 dedicatec
                9800  ;        * buffers - see text for details of these.
                9900
               10000  ;         ** Version 2b **
6562  F5       10100  INCOME PUSH AF    ; Save registers we are going to use
6563  C5       10200         PUSH BC    ;          "
6564  E5       10300         PUSH HL    ;          "
6565  D5       10400         PUSH DE    ;          "
6566  01E1F9   10500         LD    BC,#F9E1  ; Make BC point to first USART stat
6569  210060   10600         LD    HL,#6000  ; HL points at buffer base
656C  1604     10700         LD    D,#4      ; Set up loop counter
656E  ED78     10800  INC001 IN    A,(C)     ; Get the USART status
6570  0B       10900         DEC   BC        ; Make BC point at dat reg.
6571  FEFF     11000         CP    #FF       ; If we read Hex FF - no USART
6573  2824     11100         JR    Z,INC003  ; Jump if no USART installed.
6575  CB4F     11200         BIT   1,A       ; SEE IF RxRDY is set
6577  2820     11300         JR    Z,INC003  ; If not then jump
6579  E638     11400         AND   #38       ; See if any error bits are set
657B  2030     11500         JR    NZ,ERROR  ; Call the error logger if so
657D  7D       11600  FNDEOB LD    A,L       ; Get low byte of pointer
657E  C603     11700         ADD   A,#3      ; into A. Add 3 to it
6580  6F       11800         LD    L,A       ; Put it back updated.
6581  CB7E     11900  INC004 BIT   7,(HL)     ; See if top bit set
6583  2003     12000         JR    NZ,INC002  ; Jump if so  - else not EOB so
6585  23       12100         INC   HL         ; Inc HL and try again.
6586  18F9     12200         JR    INC004     ; loop till we find EOB (Hex FF)
6588  ED78     12300  INC002 IN    A,(C)      ; Get the byte.
658A  E67F     12310         AND   #7F       ; Limit to ASCII
658C  77       12400         LD    (HL),A     ; Buffer the byte
658D  23       12500         INC   HL         ; now make HL point at new EOb
658E  36FF     12600         LD    (HL),#FF    ; Write new EOB
6590  3EFF     12700         LD    A,#FF         ; A=# FF
6592  BD       12800         CP    L             ; See if HL shows buff overflow
6593  2004     12900         JR    NZ,INC003     ; Jump if not
6595  2E03     13000         LD    L,#03         ; else set buffer ptr. to base.
6597  36FF     13100         LD    (HL),#FF      ; Write a new EOB - erase buffe1
               13200  ; This forces the buffers to be circular if they
               13300  ; overflow.
6599  15       13400  INC003 DEC   D             ; Decrement D
659A  2005     13500         JR    NZ,INC005     ; Jump if all USARTs not done
659C  D1       13600         POP   DE
659D  E1       13700         POP   HL
659E  C1       13800         POP   BC
659F  F1       13900         POP   AF             ; Clean the stack and.....
65A0  C9       14000         RET   ; return to caller all unchanged.
               14100
65A1  2E00     14200  INC005 LD    L,#0          ; Zero low byte of HL
65A3  7C       14300         LD    A,H           ; Move H into A
65A4  C601     14400         ADD   A, #1         ; Effectively add #100 to HL
65A6  67       14500         LD    H,A           ; put back high byte
```

```
65A7  79      14600          LD   A,C
65A8  C603    14700          ADD  A,#3       ; Make BC point at next stat reg.
65AA  4F      14800          LD   C,A        ; which points it at next USART
65AB  18C1    14900          JR   INC001     ; and go round again
65AD  CB5F    15000 ERROR    BIT  3,A        ; See if parity error
65AF  2801    15100          JR   Z,OVERCK   ; If not then jump to next check
65B1  34      15200          INC  (HL)       ; Else inc parity error count
65B2  23      15300 OVERCK   INC  HL         ; Bump HL to point at next cntr
65B3  CB67    15400          BIT  4,A        ; see if any overrun error
65B5  2801    15500          JR   Z,FECHK    ; jump if not - else we increment
65B7  34      15600          INC  (HL)       ; the overrun error count.
65B8  23      15700 FECHK    INC  HL         ; Make HL point at next counter
65B9  CB6F    15800          BIT  5,A        ; See if any framing error.
65BB  34      15900          INC  (HL)       ; now increment next counter
65BC  2801    16000          JR   Z,ERR1     ; Finished if not
65BE  34      16100          INC  (HL)       ; Else increment the FE count.
65BF  3E15    16200 ERR1     LD   A,#15      ; Load the code to reset errors
65C1  03      16210          INC  BC         ; Now BC points at stat reg
65C2  ED79    16300          OUT  (C),A      ; flags, and output it.
65C4  0B      16310          DEC  BC         ; BC back to dat reg.
65C5  18D2    16400          JR   INC003     ; And off we go
              16500 ;         * The slate is now clean. USART can now receive
              16600 ;         * another character.
              16700
Pass 2 errors: 00

ERR1    65BF  ERROR  65AD  FECHK  65B8  FNDEOB 657D
GOUT    655F  INC001 656E  INC002 6588  INC003 6599
INC004  6581  INC005 65A1  INCOME 6562  K1     6507
KERNEL  6500  KEYCHK 651F  OVERCK 65B2  SEND   6532
SEND01  653D  SHUF01 6554  SHUFFL 6548

Table used:    248   from    878
Executes: 25856
```

Fig. 3.55: Assembly listing for MACCODE.

The MACCODE modules in more detail

If you know nothing about assembly language programming skip this section and come back to it after reading chapter 5. Fig 3.55 shows the assembly listing for the MACCODE section of the serial driver. The following few paragraphs give a brief description of the actions and decisions taken by each module:

KERNEL: As its name suggests KERNEL is the inner routine of the machine code section. When it is entered it does a system call to make the cursor visible on screen. (BASIC always turns the cursor off at program run time.) Next KERNEL calls another of its companion routines, INCOME. This routine will be detailed shortly, but it checks all USARTs for incoming characters and buffers them. Then KERNEL does a little sequence to check to see if there are any characters pending for the selected channel, if there are then it prints them to the CPC screen using another call to the firmware. After outputting a character to the screen KERNEL proceeds to call another companion routine – SHUFFL. This routine moves the contents of the selected channel buffer up by one. Finally KERNEL checks to see if any characters have been typed at the CPC keyboard. If not then it starts itself again, if so then it checks to see if the character code is the one for CTRL/P or

CLR. If it is then a return to BASIC is made. If not then the SEND routine is called, and on return from SEND KERNEL is restarted.

SEND: This routine sends the character which is passed to it in the A register out on the selected channel. It picks up the number of the selected channel, and converts it into a USART address. It then waits for the transmitter section of the USART to become ready and sends the character.

SHUFFLE: is a routine which shuffles all the characters in a received character buffer forward by one character. It is used by all routines which take the character at the front of the buffer to move the remaining characters forward, and make some room at end of the buffer. Thus as one character is removed for display at the front end of the buffer, then the buffer contents are shuffled up, and a byte becomes available at the end of the buffer space.

INCOME: The INCOME routine is called by KERNEL to check for incoming bytes of data on the USARTs. This is also the routine which updates the error counters for each channel. The purpose of providing the error counters is to enable you to keep a check on the number of data errors which occur in character reception. An excessive amount of data errors would indicate a hardware, or connecting line problem, which should be dealt with. These errors always occur in real world applications, but one or two here and there should be the maximum tolerated.

INCOME takes correctly received characters and places them in the appropriate buffer. The new character replaces the End of buffer (EOB) marker –value Hex FF– and INCOME writes a new EOB in the byte after the new character, unless the buffer has overflowed in which case the buffer is re–initialised and the whole of its contents are lost. Readers needing larger received character buffers should not find it very difficult to modify the software to their needs.

Because INCOME monitors all the installed USARTs all the time, no characters should be missed. The hardware and software were tested by connecting all four channels to four serial lines of a VAX super mini computer and sending out a 200 byte file to all four channels at 2400 Baud simultaneously. This worked with no problem, so there should be no trouble with less intensive applications.

General points

Because the four channel interface is a fairly complex add on for your CPC there are a couple of points to bear in mind when using it.

First the fact that there are two channels connected to each baud rate generator means that when you change the rate for one channel, you will also change the rate for the other. When you are using the serial driver software the baud rate for a pair of channels will be the one most recently selected in the set up menu. For example if you go into the set up menu and choose

channel 3 to be the selected channel and then set it to 300 Baud, then the driver will also set channel two for 300 Baud. This might result in errors if channel two is connected to, say, a 9600 Baud line. So bear in mind things like this if you seem to suffer from a high error rate. You would usually choose to have each pair of channels connected to data channels with the same rate of course, but there may be some situations where this cannot be so.

If you use the interface to allow you to use your CPC as a multichannel VDU you should have no problems with missed messages. If you are using it as a VDU with three message channels then you should regularly enter the set up menu to ascertain if there are any messages waiting on the other channels, if you do not do this then you will miss messages when the buffer is full.

When receiving large amounts of data at the higher rates it may happen that overrun errors will occur on the selected channel. This can occur when the screen has to be scrolled by the CPC firmware. What appears to happen is that the firmware retains control of the processor during the time it takes to scroll the screen, and therefore INCOME gets no chance to collect received characters, resulting in an overrun error. I only experienced this fault when sending continuous line feeds at speeds greater than 2400 baud, it did not affect normal operation. If this error occurs the flag is reset and one character will be missed.

Some typical applications

There are several applicational configurations which the four channel RS232 board could be used in. It may be helpful to enumerate these.

1) Using the CPC as a multichannel VDU in a commercial or educational establishment with several large computers. Figure 3.56 shows how such an application would be wired.

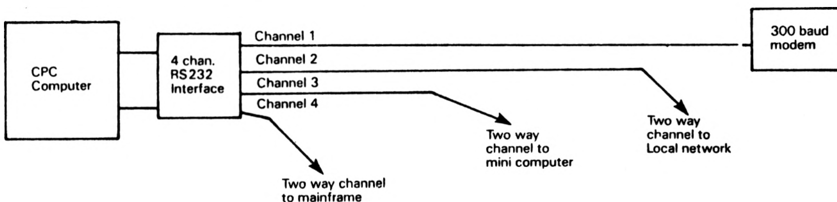


Fig. 3.56: Block diagram showing how to use the CPC computer and RS 232 four channel interface as a multichannel Visual Display Unit (VDU).

2) Using the CPC as a VDU, but with message reception capability from other VDU or similarly equipped CPC computers. This would allow you to use the CPC as a normal VDU, but to have messages sent from other offices in the same building, for you to view later. Fig 3.57 shows how this might be set up.

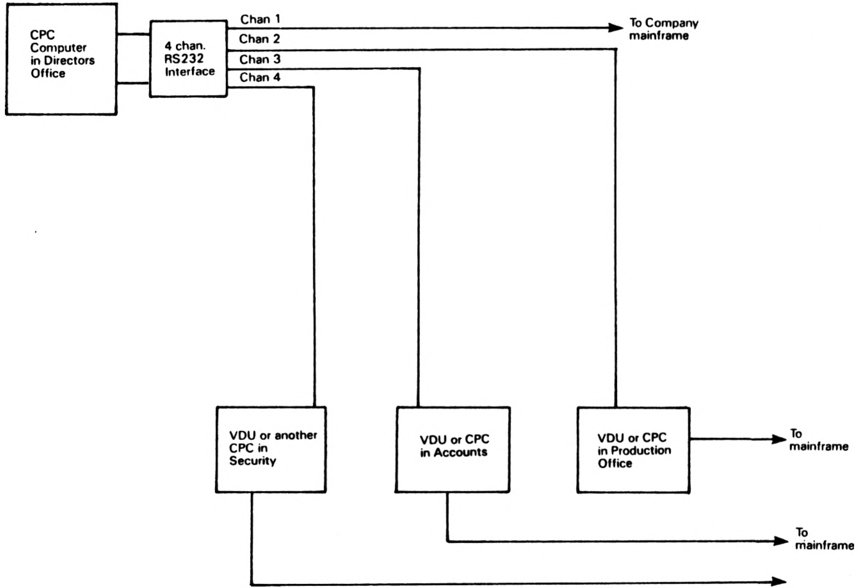


Fig. 3.57: Using the CPC and four channel interface as a VDU and message receptor station.

3) Using the CPC purely as a message reception terminal, as a kind of forget me not intercomm. That is, if it is on the screen it can't slip your mind! See fig 3.58.

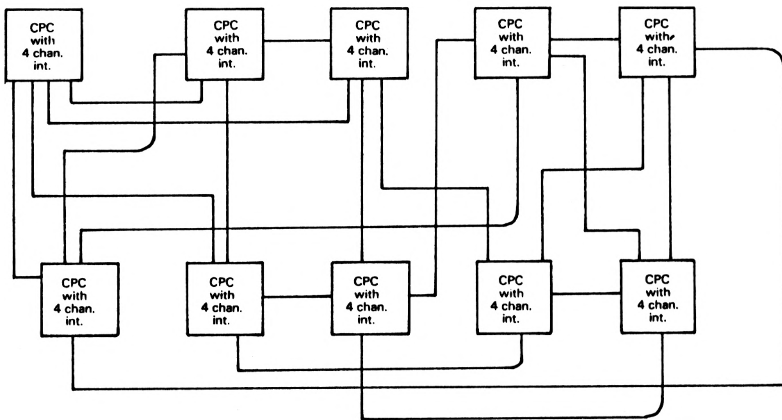


Fig. 3.58: Using the CPC and four channel interface as message terminal.

4) As a multichannel test rig. This would be used in a computer terminal repair shop, where printers, or VDU units were mended. The CPC could be used as a console to send selected typed messages to the units under test. With a little extra software (Namely modifications to the KERNEL routine) you could make the CPC send a continuous stream of characters to all channels. (you could also modify the soak test program to do this).

5) To interface to a serially equipped printer which needs 8 bits of data supplied and therefore cannot be run from the centronics port on the CPC.

I daresay that someone somewhere will find yet more applications for the hardware, but these will probably be the major areas of usefulness.

```
             RS232 version B info sheet

   Time taken at all possible Baud rates to transmit 2048 bytes.
  -------------------------------------------------------------------

19200 Baud took                1.2 seconds
50    Baud took 7 minutes and  5 seconds
75    Baud took 5 mintes  and .5 seconds
134.5 Baud took 2 minutes and  9 seconds
200   Baud took 1 minute  and  9 seconds
600   Baud took               37.6 seconds
2400  Baud took                9.4 seconds
9600  Baud took                2.3 seconds
4800  Baud took                4.7 seconds
1800  Baud took               12.6 seconds
1200  Baud took               19   seconds
300   Baud took  1 minute     15.2 seconds
150   Baud took  2 minutes    30.5 seconds
110   Baud took  3 minutes    27.5 seconds
```

Table showing Baud rates available by use of the divide by 1, 16, or 64 feature of the 8251A USART. This information may enable you to obtain two required rates on USARTs connected to the same Baud rate generator. For example if you were to load six into Baud rate register one you can program USART 1 to divide by 16 to make channel 1 run at 600 Baud, and program USART 2 to divide by 64 to have channel 2 run at 150 Baud.

Division factor programmed into USART mode byte

Value in Baud rate register.	X1	X16	X64
0	307200	19200 *	4800 *
1	307200	19200 *	4800 *
2	800	50 *	12.5
3	1200 *	75 *	18.75
4	2152	134.5 *	33.625
5	3200	200 *	50 *
6	9600 *	600 *	150 *
7	38400 *	2400 *	600 *
8	153600	9600 *	2400 *
9	76800	4800 *	1200 *
A	28800	1800 *	43.75
B	19200	1200 *	300 *
C	38400	2400 *	600 *
D	4800 *	300 *	75 *
E	2400 *	150 *	37.5
F	1760	110 *	27.5

* denotes a standard rate.

Fig. 3.59: Information sheet on the RS 232 VB four channel interface

Conclusion

The version B RS 232 interface represents a very useful piece of hardware to build for your CPC. It gives you the opportunity to interface to a very wide range of devices, and opens up the world of dial in networks, inter computer communication, and all kinds of data transfer. Figure 3.59 is a table of information about the version B interface, which shows the baud rates you can get by using the divide by 64 feature of the 8251A via the defaults set up of the driver program. It also shows a table of time taken to transfer 2K bytes of data at all the major baud rates. Figure 3.60 is the power pins list for the project and Fig. 3.61 is the parts list for it. Finally, readers are referred to appendix four which shows the RS232 connections, and to appendix six for a description of serial transmission.

E Number	Device		description	Power connection pins			
	Type	Equivs	Function	+5V	+12V	-12V	OV
1	74LS138	–	3 to 8 line decode	16	–	–	8
2	74LS02	–	Quad 2 I/P NOR	14	–	–	7
3	74LS21	–	Dual 4 I/P AND	14	–	–	7
4	4702B	–	Baud rate gen.	16	–	–	8
5	4702B	–	Baud rate gen.	16	–	–	8
6	8251A	–	USART Chip	26	–	–	4
7	8251A	–	USART Chip	26	–	–	4
8	8251A	–	USART Chip	26	–	–	4
9	8251A	–	USART Chip	26	–	–	4
10							
11							

Fig. 3.60: Power connection pins for chips used on the RS 232 Version B project.

Project 8: EPROM programmer

One of the most useful things to have when extending the hardware of any home computer is an EPROM programmer. If you already know what an EPROM is, and how to use it, please skip the next two paragraphs.

EPROM is an acronym for Erasable Read Only Memory. As you may already know there are two kinds of computer memory chip, RAM (Random Access Memory) which loses its contents when powered off, and ROM which retains its contents forever. The EPROM is nearer to a ROM than to a RAM, but unlike a ROM it can have data written into it. The way that the kind of EPROM we are about to use works is that a special high voltage is applied to the EPROM chip to put it into a special mode whereby, with a little support logic, it can have data "blown" into it.

Parts list for RS232 version "B". Four channel RS232 interface.

Resistors

R1 = 10M Ohms Half Watt resistor (eg RS 133-330)
R2-R5 = 5K6 Ohms Half Watt resistor (eg RS 132-674)

Capacitors

C1-C2 = 56 PicoFarad 16 volts (eg. RS 124-746)
C3-C6 = 470 PicoFarad 16 Volts (eg. RS 124-897)

Semiconductors

E1 = 74LS138 3 to 8 line decoder. (eg. RS 307-648)
E2 = 74LS02 Quad triple input NOR gate. (eg. RS 307-496)
E3 = 74LS21 Dual four input AND gate. (eg. RS 305-210)
E4-E5 = 4702 B Baud rate generator (eg. RS 303-517) **
E6-E9 = 8251A USART chip (eg. RS 309-357) **
Must be 8251A: Most 8251 chips will not work with this design.

E10-E11= 74LS174 Hex latch chip (eg. RS 307-682) **
E12 = 1488 RS232 transmit chip (eg. RS 309-587)
E13 = 1489 RS232 receiver chip (eg. RS 309-593)

Miscellaneous

Sockets for all chips
Printed circuit board (See note 3 below).
XTAL 1 = 2.45676 Megahertz crystal (eg. RS 303-359)
16 way DIL connector (eg. RS 402-311)

NOTES:

1) Components marked with double asterisk are not all required if you do not build all four channels. The following yes/no table indicates which of the components will be required for any number of channels between two and four.

Integrated circuit numbers

Channels Installed	E4	E5	E6	E7	E8	E9	E10	E11
2	Yes	No	Yes	Yes	No	No	Yes	No
3	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
4	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

2) Note that the RS part numbers quoted may be supplied in packs containing more than one part.

3) Complete kits of parts are to be available from Halstead designs LTD.

Fig. 3.61: Parts list for the RS 232 VB project.

The exact sequence is that first the programming voltage is applied to the VPP pin. In the case of the larger EPROMs (such as the two for which a programmer is about to be described) this is 21 volts DC, for smaller EPROM types it is often 25 Volts. Secondly the address is applied to the address pins of the EPROM. Next the data to be programmed into the addressed byte is placed on the data bus. Then a 50 millisecond (0.050 second) TTL low level pulse is applied to the PGM pin of the chip. When this terminates the address and data lines are released, and the byte has been programmed (or "blown") with the data. These steps are repeated until the required number of bytes have been programmed. You can program any number of bytes in the EPROM, you don't have to do all of them. Once the desired bytes have been programmed, the chip behaves like a ROM, keeping its contents intact through power downs. The other way in which an EPROM differs from a ROM is that it can be erased using certain Ultra Violet light sources. All EPROM chips have a small window in the top of them which allows the light from a special UV lamp to shine on to the actual chip. The EPROM can only be "blown" and erased in this way a certain number of times, usually three, after which it becomes unreliable.

EPROM chips are used in a huge range of applications where their cheapness, and the fact they are reusable, make them a no–contest winner over customised ROM chips. Their very existence has made it possible for hobbyists and low volume manufacturers to produce units which would once have been impossible to make economically. The Amstrad computers allow you to add expansion ROMs which provide extra software which is instantly accessible. An expansion ROM can, as we shall see, be an EPROM with your own extra software blown into it.

Currently there are a great many EPROM chips available, but two devices which are available at almost all electronics shops are the 2764, and the 27128. These chips are the ones which this project is designed to "blow" data into. Both devices come in a 28 pin DIL package, and have identical pin connections except for pin 26, which is not connected on the 2764, and is used as address line 13 on the 27128. The 2764 has 8192 locations, each eight bits wide. The 27128 has 16384 locations, again each is eight bits wide. Figure 3.62 is a reproduction of a very useful data sheet from Intel corporation who are the prime source for the 27128 ,the 2764, and all the other chips in the 27xxx range. Fig 3.62 is especially useful because it shows the pinouts of all the 27xxx range of EPROM chips from the 2716 upwards.

There are a great many different ways to build EPROM programmers. Perhaps the most simple is to enter the data into the EPROM from switches, setting up the binary code required to be in each byte on the switches. This method was barely acceptable in the days of the 2708 EPROM (1K x 8 bits), but as we are now dealing with potentially 16 times that amount of data to be blown into a EPROM, it is obviously out of the question for all practical purposes.

27128
128K (16K x 8) UV ERASABLE PROM

- **250 ns Maximum Access Time . . . HMOS*-E Technology**
- **Compatible to High-Speed 8 MHz iAPX 186 MPU . . . Zero WAIT State**
- **Two-Line Control**
- **Pin Compatible to 2764 EPROM**
- **Industry Standard Pinout . . . JEDEC Approved**
- **± 10% V_{CC} Tolerance Available**
- **Low Power Characteristics**
  - **—100 mA Max. Active Current**
  - **—40 mA Max. Standby Current**

The Intel 27128 is a 5V only, 131,072-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The standard 27128 access time is 250 ns which is compatible to high-performance microprocessors, such as Intel's 8 MHz iAPX 186. In these systems, the 27128 allows the microprocessor to operate without the addition of WAIT states.

An important 27128 feature is the separate output control, Output Enable (OE) from the Chip Enable control (CE). The OE control eliminates bus contention in multiple-bus microprocessor systems. Intel's Application Note AP-72 describes the microprocessor system implementation of the OE and CE controls on Intel's EPROMs. AP-72 is available from Intel's Literature Department.

The 27128 has a standby mode which reduces the power dissipation without increasing access time. The active current is 100 mA, while the standby current is only 40 mA. The standby mode is achieved by applying a TTL-high signal to the CE input.

±10% V_{CC} tolerance is available as an alternative to the standard ±5% V_{CC} tolerance for the 27128. This can allow the system designer more leeway in terms of his power supply requirements and other system parameters.

The 27128 is fabricated with HMOS*-E technology, Intel's high-speed N-channel MOS Silicon Gate Technology.

*HMOS is a patented process of Intel Corporation.

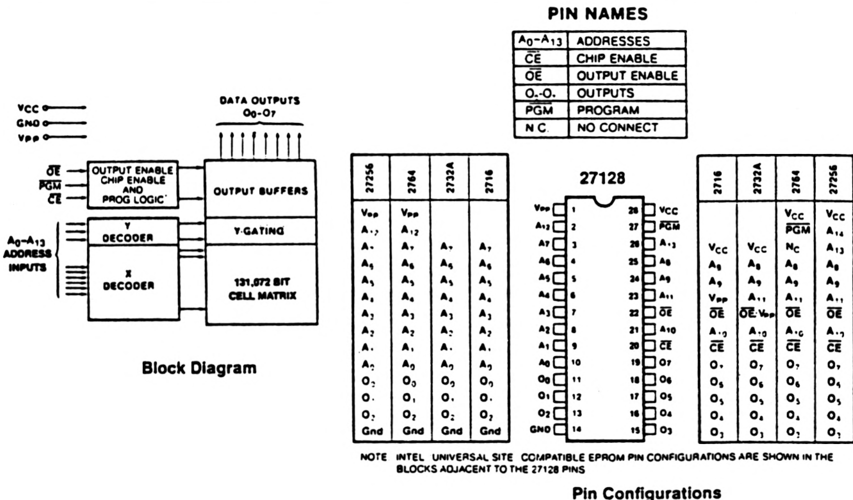


Fig. 3.62: EPROM data sheet showing the pin usages for all 27xx EPROMs.

Reproduced by kind permission of Intel Corp.

The second method relies on some of the computers addressable memory space being unoccupied, with the EPROM being mapped into this memory area. This again was fine when EPROM and home computer memory sizes were smaller, but now there might be some considerable difficulty in finding 16K of memory space spare. On the CPC machines you could switch upper ROMs – substituting the EPROM for BASIC, but this would make the software involved more complex, since you cannot normally read the top 16K of ROM to check that the EPROM was blown with no errors, but it could be done.

Yet another method of EPROM programmer design uses a link between your computer and a programmer unit to transfer the data to EPROM. (this method could be used on the CPC, making use of the parallel transfer channel described earlier). A microprocessor, or similar large logic building block then does the actual "blowing" inside the external unit. This method is good where you need to save processor time, as in a multi-user, or multi-tasking, system but the extra hardware involved is not really justifiable where the host processor supplying the data to be programmed is twiddling its silicon thumbs while the data is being blown anyhow.

The final method is the one which has been selected for the EPROM programmer project. This method involves the use of yet another PPI chip, the ports of which are connected to the address and data lines of the EPROM being programmed. As usual the PPI looks to the system like four I/O registers. A fifth, read only, register supplies EPROM programmer status information. The register addresses and read/write register images are shown, along with the block diagram of the programmer, in Figure 3.63. The LOCAL BUS (or sideways bus) concept referred to on the diagram will used in the next project as well.

The advantages of this approach to EPROM programmer design are twofold. Firstly there always exists a small possibility that accidents will occur with a directly mapped EPROM. This might be that you leave the programming voltage switched on, and inadvertently program some rubbish into an EPROM, or that the hardware develops a fault and places a high voltage on the Z80 bus. In this scheme the PPI gives some protection to the rest of the computer. Unlikely happenings perhaps, but possible. The greatest advantage to this method however, is that you have to allocate just five I/O addresses to the programmer. This gets round the need to elbow something else out of memory to make room to map the EPROM. To be sure it is not so fast to have to load the low address byte, high address byte, and the data, into a PPI, but blowing an 8K or 16K EPROM is a slow business anyway. (Total it up, 0.050 times 16384 = approx. 819 seconds– or thirteen minutes – and that is just for the program pulses).

The circuit diagram for the EPROM programmer address decode and control logic is shown in Figure 3.64. The programmer registers have addresses from Hex F8E0 to F8E7. (The bit pattern of this address is shown on the diagram.) As with all expansion bus peripherals the upper four address bits can be set to

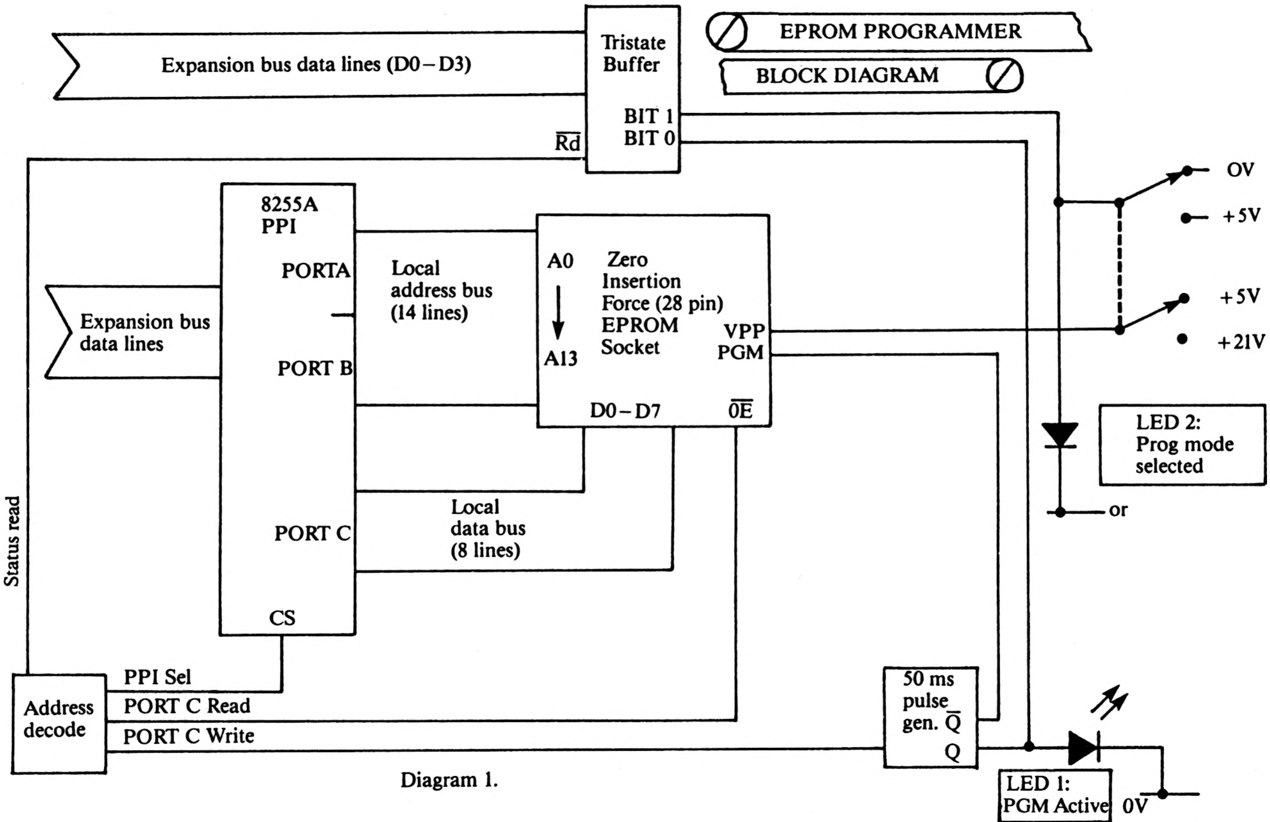


Fig. 3.63: Block diagram of the EPROM programmer.

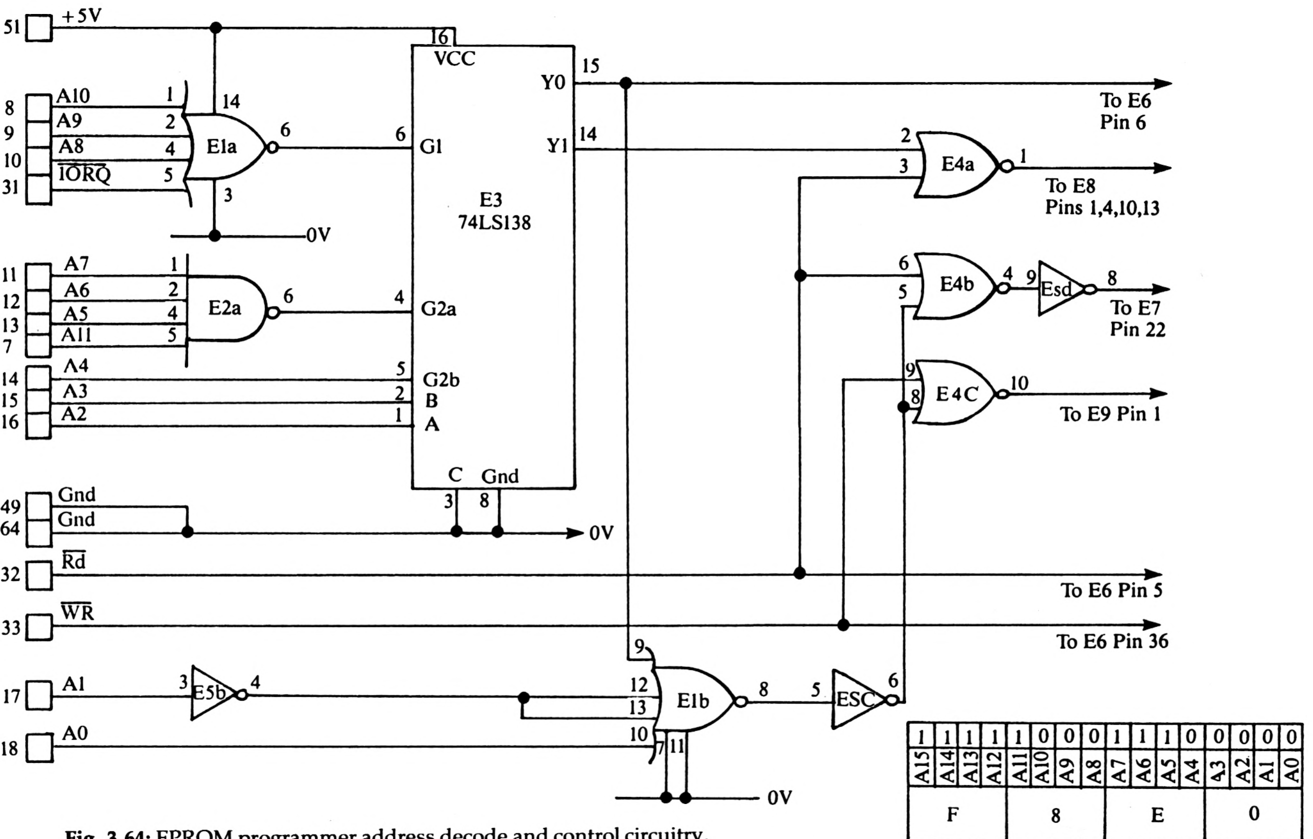


Fig. 3.64: EPROM programmer address decode and control circuitry.

any value, as long as address bit A10 is low, so A12–A15 are not decoded. The address is decoded by E1a and E2a, which enable the address decoder E3. Only two of the decoder outputs are used in the design as it stands, but the design is readily adapted by readers who may like to build the design onto a stripboard or microboard and add a second PPI, and programming socket to convert the project into an EPROM copier. Address lines A2 and A3 feed into E3 and are used to make it decode whether the PPI or the status register is being addressed. If the PPI is being addressed then the internal decoding logic of the 8255A selects the appropriate register within the PPI. The three ports of the PPI are used as follows:

PORT A – Always an output port – connected to the 8 lower order address lines of the EPROM being programmed.
PORT B – Always an output port – connected to the 8 high order address lines of the EPROM being programmed.
PORT C – Is set up as an output port when programming an EPROM, but as an input port when checking its contents. Port C is connected to the data lines of the EPROM being programmed.

Gate E1b detects when port C is being accessed and gates E4b and E4c will trigger the program pulse generator if the access is a write, or output enable the EPROM if the access is a read. Gate E4a enables the status register if a read occurs from an I/O address in the range F8E4 to F8E7.

Figure 3.65 shows the PPI chip – E6 – and the EPROM socket. This socket is a Zero Insertion Force (ZIF) socket. ZIF sockets have a special lever which releases the chip from the socket, thus circumventing the problem of contact wear which would occur when repeatedly plugging and unplugging chips from a normal socket. Figure 3.65 also shows the READ/PROGRAM select switch. This is a double pole two way switch. The two separate poles are very important. One is used to feed either a logic one or a logic zero (+5 Volts and 0 volts) into the status register, and also has the program mode warning light (a flashing LED) connected to it. The second pole is used to connect either +21 volts or +5 volts to the VPP input of the EPROM. Program pulses from the program pulse generator (see Figure 3.66) are fed into the EPROM ZIF socket on pin 27 – which is the PGM input. Obviously great care must be taken not to connect between the poles (that is the centre contacts) of the switch. Finally, the inverter E5e inverts the bus reset pulse to the polarity required by the 8255A. The PPI is chip selected by the signal applied to its pin 6 from the address decode circuit. As shown on the diagram the PPI mode byte should be Hex 80 when blowing some bytes into EPROM, and Hex 89 for all other uses.

Figure 3.66 shows the remaining elements of the hardware. These are the status read register, and program pulse generator. The status read register is used by programs to ascertain two things: If the program pulse generator is active, and what position the READ/PROGRAM switch is in. The read image of the register is drawn on Figure 3.66. The program pulse generator is comprised of a monostable, with an output pulse length adjustable with VR1

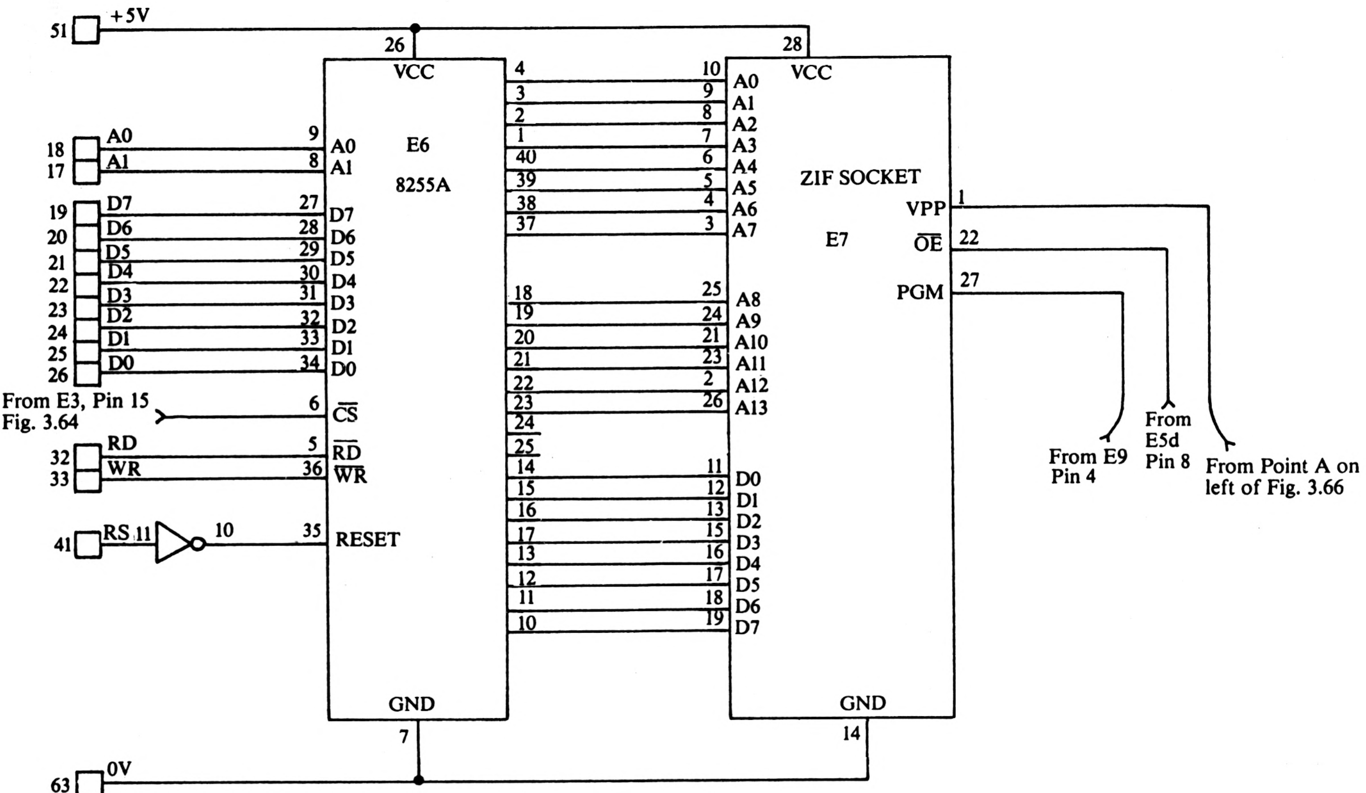


Fig. 3.65: EPROM programmer PPI chip and ZIF socket.

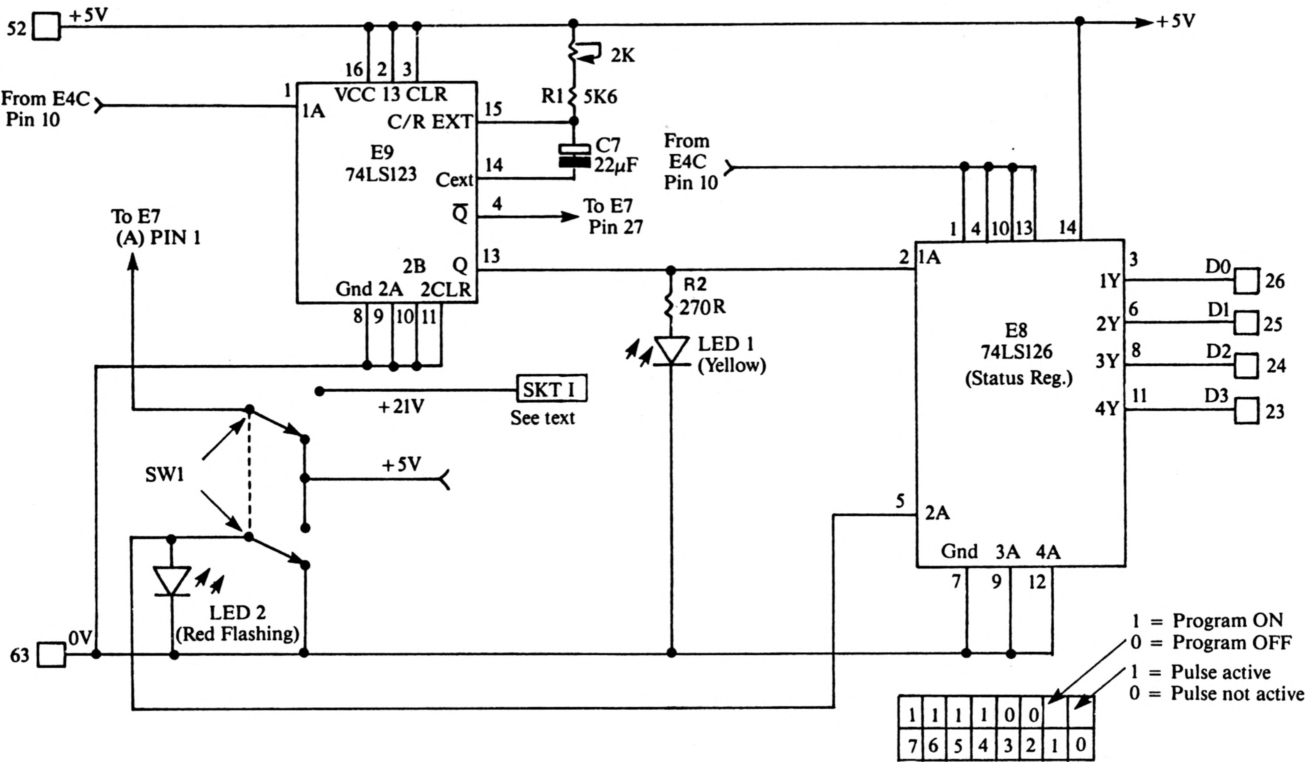


Fig. 3.66: Remainder of EPROM programmer circuitry.

from about 45 up to 60 milliseconds. (Set up details below). Please skip the next paragraph if you already know what a monstable is, and how it can be used.

A great deal of computer electronics relates to generating and detecting voltage pulses. A monostable is a circuit available on a chip like the 74LS123 used here to generate pulses. A monostable usually has two outputs. These are the Q and the Q bar outputs. The Q output is normally low, but pulses high when the monostable is triggered. The Q bar output is normally high but pulses low when the monostable is triggered. The duration of the pulse which is output when the trigger condition occurs can be set by the connection of a capacitor and resistor to the timing pins CEXT and RC/EXT of the monostable. As is the case here, the pulse length can be made adjustable by making part of the resistor value variable. The conditions required to trigger a 74LS123 chip (which, incidentally, contains two separate monostables – one remains unused here) can be a negative going pulse, or a positive going pulse. For a fuller description of the 74LS123 you are advised to consult a TTL data book.

The Yellow "Program pulse active" LED is connected via a resistor to the Q output of the monostable. The Q output also feeds into the input of E8 which is the tri–state buffer chip used as the status register. The monostable is triggered on the negative going edge at its A input. This happens when port C is written into and gate E4c (shown on Figure 3.63) pulses its output into pin 1 of the monostable. Thus triggered, the Q bar output of the monostable feeds into the PGM input of the EPROM as already described.

Before we look at software to do all the setting up and driving of the programmer, here is a detailed step by step explanation of how a byte is programmed into EPROM in this design.

1) Place an EPROM into the ZIF socket and close the locking lever. Ensure that the READ/PROGRAM switch is in the READ position. Connect the 21 Volt flying lead from the SIGMA PSU to PL1. on the programmer board.

2) Power on the machine and SIGMA power supply.

3) Initialise the PPI with mode byte Hex 80, using the BASIC command:
```
OUT &F8E3,&80
```

4) Now output the low byte of the LOCAL address to port A using:
```
OUT &F8E0,&00
```

5) Next output the high byte of the LOCAL address to port B:
```
OUT &F8E1,&00
```

6) Now place the READ/PROGRAM switch in the PROGRAM position. The PROGRAM indicator LED starts to flash.

7) Output the value to be "blown" into the EPROM to port C. For example to program the Hex value C9 use:
`OUT &F8E2,&C9`

8) The PROGRAMMER ACTIVE LED (the Yellow one) will briefly flash. Now change the READ/PROGRAM switch back to READ.

9) Change the PPI to mode Hex 89 by typing:
`OUT &F8E3,&89`

10) Now re–output the LOCAL address as you did in steps 4 and 5 above. (This has to be done becuase the PPI ports are all reset when the PPI mode is changed).

11) Now type:
`PRINT HEX$(INP(&F8E2))`
which will print the value (in Hex) of the byte in the EPROM which you have just programmed. (in the example used in step 7 above this should be C9)

12) Power off the machine, then disconnect PL1.

All that just to program one byte! The above is making a long job of it, becuase software to do all this for you will now be presented, but I hope that the step by step description will make the software more easy to understand.

```
100     '** EPROM programmer project: Monostable set up program. This       **
         ** program retriggers the program pulse monostable constantly      **
         ** to allow you to set up the pulse length using an oscilloscope.  **
150      ON BREAK GOSUB 32700: W=25
200      MODE 2: INPUT "Is there an EPROM chip in the programmer";R$:
         IF UPPER$(LEFT$(R$,2)) <> "NO" THEN CLS:
         PRINT "Power off, remove chip then run program again": GOTO 32767
300      PRINT "Press ENTER to increase  time between triggers":
         PRINT "or space bar to decrease time between triggers":
        '** Thats the way to do it!
400      OUT &F8E2,&FF
410      FOR I=1 TO W: R$=INKEY$: IF R$="" THEN 420 ELSE
         IF ASC(R$)=13 THEN W=W+1 ELSE IF ASC(R$)=32 THEN W=W-1
420      NEXT I: GOTO 400
32700    MODE 2: PRINT "Program aborted by *Break*"
32767    LOCATE 10,10: END
```

Fig. 3.67: BASIC program to continually trigger program pulse generator.

Setting up the program pulse length

The setting up of the circuit consists of adjusting VR1 to get a 50 millisecond pulse length out of the monostable. A program to continually trigger the monostable is listed in Figure 3.67. If you have an osciloscope available then this is a simple matter (50 milliseconds plus or minus 2 milliseconds is accurate enough). If no 'scope is available to you, then you should use the

program listed in Figure 3.68. This program displays on screen how many CPC 300th second ticks have occurred during the time the program pulse was active. There is no direct correlation between the numbers and 50 milliseconds due to delays incurred by the programs processing. You should run this program for about a minute, to allow stabilisation, and then adjust VR1 until only the numbers 15 or 16 are displayed on the left side of the screen. (mostly 15). The set up is then completed.

```
100    '** EPROM programmer project: Monostable set up program. This **
       ** program allows you to set the program pulse width without **
       ** the use of an oscilloscope. **
150    ON BREAK GOSUB 32700
200    MODE 2: INPUT "Is there an EPROM chip in the programmer";R$:
       IF UPPER$(LEFT$(R$,2)) <> "NO" THEN CLS:
       PRINT "Power off, remove chip then run program again": GOTO 32767
300    CLS: WINDOW #1,10,80,1,25: WINDOW #0,1,5,1,25: LOCATE #1,10,10:
       PRINT #1,"Run for 1 minute, then adjust as described in the book,":
       LOCATE #1,10,12: PRINT #1,"until only 15 or 16 are displayed"
400    OUT &F8E2,&FF: T=TIME: WHILE (INP(&F8E4)AND 1) <> 0: WEND:
       PRINT TIME-T: GOTO 400
32700  MODE 2: PRINT "Program aborted by *Break*"
32767  LOCATE 10,10: END
```

Fig. 3.68: EPROM programmer set up program.

Software to run the EPROM programmer

The programmer driving software will write into the EPROM the contents of an area of RAM, or data contained in an ASCII file. (for example a BASIC program saved using the ",A" format, or a file created using AMSWORD or other word processors) The program can tell if it runs out of EPROM before all the bytes of data are written, and will prompt you to place another chip in the programmer, into which the next section of data will be written. Becuase it can sense the state of the READ/PROGRAM switch the program also prompts you to turn the switch to the appropriate position. The program may seem somewhat slow, this is becuase it reads back each byte straight after writing it, which involves changing the PPI mode after every byte has been "Blown". This method was implemented in preference to a separate read pass, so that any failure would immediately halt the program.

The program is listed in Figure 3.69. It first prints up its features, parameters etc. Then it asks you which type of EPROM you wish to "Blow". You must make one of three responses to this, just press ENTER to default to a 2764, or you can actually type – "2764" , or you can type "27128". Any other response is rejected, and the question is asked again. The program then tells you the extents of the LOCAL address for the EPROM type you have selected, and asks you for the LOCAL address to begin programming from. (Silly replies to this question receive short shrift!). With the preliminaries sorted out the program then goes on to establish where the data to be programmed into the EPROM is coming from. You are next asked whether the data is in memory or in an ASCII file. If you select memory as the data source then you are asked for the start address (Remember that you can enter Hex values to an INPUT statement by prefixing with the "&" character). Then the program asks you

for the last address to be copied into EPROM. At this point you can put the actual finish address of the data, or if you know the length of the data you can reply with "+xx" where xx is the number of bytes in Hex or decimal as required, the program then adds the xx to the start address. If you specify that an ASCII file is the data source then the filename is asked for and an attempt is made to open the file on tape or disk.

The subroutine located at lines 5000 to 5080 is the one which does the actual transfer of data to EPROM. It receives the byte to be tranferred to the EPROM in the variable BLOW. The source data fetch routines use the PEEK and LINE INPUT statements to fetch the source data a byte at a time and pass them on to this subroutine in the BLOW variable. After programming the byte the subroutine read checks the EPROM to ensure a successful transfer, if a failure occurs then the error log subroutine at line 6000 is called. This prints out the details of the error —including the binary patterns of the byte which it wrote and the different one which it read back. At this point you can command the program to abort or resume. The listing is fairly heavily annotated, so that you can modify it if you need to.

```
500 ' **          EPROM programmer software, program number 1          **
      **  This software accompanies  the text in "Understanding and **
      **  expanding your CPC 464/664" by Alan Trevennor.             **
900   MEMORY 15000: BBYTES=0
1000  MODE 2: OUTCOME$="successfully": PRINT STRING$(79,"#"):PRINT TAB(15);
      "EPROM blowing program  - full details in the book":
      PRINT STRING$(79,"#"): LOCATE 1,5: PRINT "This program will:": PRINT
1010  PRINT "1) Take data from memory, or from an ASCII file":
      PRINT "   and blow it into an EPROM.": PRINT: PRINT
      "2) Advise of any failures, with optional continuation."
1020  PRINT: PRINT "3) Allow you to continue into another EPROM   -":
      PRINT "   if your data will not fit into one EPROM chip": PRINT:
      PRINT: PRINT TAB(10);"Notes on usage:"
1030  PRINT: PRINT "You may use 2764 (8K x8) or 27128 (16K x8) EPROM chips.":
      PRINT: PRINT "The number of bytes to blow specification is inclusive of"
1040  PRINT "the first and last bytes in the specified range":
      LOCATE 1,25: PRINT "Press any key to continue":
      WHILE INKEY$="": WEND: LOCATE 1,5: PRINT CHR$(20);:
1050  INPUT "EPROM type to be programmed: 2764, or 27128 <2764>";ETYP:
      IF ETYP <> 27128 AND ETYP <> 2764 AND ETYP <> 0 THEN 1050 ELSE
      LOCATE 1,5: PRINT CHR$(20);: IF ETYP=0 THEN ETYP=2764
1060  PRINT "The first byte in the EPROM is referred to as Local address zero.'
      PRINT "The last address of this EPROM is referred to as local address";:
      IF ETYP=27128 THEN PRINT 16384 ELSE PRINT 8191: PRINT
1070  INPUT "Local address of EPROM bytes to begin programming <0>";LOCAL:
      IF LOCAL <0 OR LOCAL > 16384 THEN PRINT "Dont be silly!": GOTO 1070
1075  IF LOCAL > &1FFF AND ETYP=2764 THEN PRINT
      "Illegal Local address for a 2764 - try again": GOTO 1070
1080  IF (INP(&F8E4) AND 2) =0 THEN PRINT
      "Place the PROG/READ switch in the PROG position": WHILE (INP(&F8E4)
      AND 2) = 0: WEND: PRINT "Thank you."
1090  OUT &F8E3,&80: OUT &F8E1,(INT((LOCAL/256)) AND 255):
      OUT &F8E0,(LOCAL AND 255)
1100 '** Now the initialisation is done. We must now go on to **
      ** finding out where we are to fetch the source data    **
      ** from.                                                **
1110 '**            Source data select phase                  **
1120  LOCATE 1,5: PRINT CHR$(20);
1130  PRINT "Now you must specify the source of the data to be blown":
      PRINT "into the"STR$(ETYP)". Data can be in memory, or in a file-":
      PRINT "the only allowable file type is ASCII - see book text."
1140  LOCATE 1,10: PRINT "fetch the source data from (M)emory or (F)ile? ";:
```

```
      WHILE R$="": R$=INKEY$: WEND: R$=UPPER$(LEFT$(R$,1)): IF R$="M" THEN
      PRINT "Memory" ELSE IF R$="F" THEN PRINT "File" ELSE R$="": GOTO 1140
1150  SOUND 1,115,25: SOUND 2,100,25: IF R$="M" THEN 1200 ELSE
      LOCATE 1,12: PRINT CHR$(20);: INPUT "What is the filename";F$:
      IF F$="" OR LEN(F$) > 16 THEN 1150 ELSE F$=UPPER$(F$)
1160  PRINT: OPENIN F$: LOCATE 1,14: PRINT "File opened"+CHR$(20): WHILE EOF =(
      LINE INPUT #9,R$: FOR I=1 TO LEN(R$): BLOW=ASC(MID$(R$,I,1)): GOSUB 5010
      NEXT
1165  BLOW=&D: GOSUB 5010: BLOW=&A: GOSUB 5010: BLOW=&FF: GOSUB 5010:
      WEND: GOTO 32760
1170 '** Line 1160 opens the ASCII file (error if file type <> ASCII) **
      ** it tells the user that the file was opened, then it gets each**
      ** string and passes each character in it for blowing, + CR/LF  **
1175 '** When everything is BLOWN an #FF byte is programmed to show   **
      ** that this is the end of the text.                            **
1200 '** This section feeds source data from memory into the BLOW **
      ** subroutine. It first gets the start address and length   **
      ** of the data.                                             **

1210  LOCATE 1,12: PRINT CHR$(20);:
      INPUT "Address of first byte of source data";FIRS:
      INPUT "Address of last  byte of source data";LAS$
1215  IF LEFT$(LAS$,1)<> "+" THEN LAS =VAL(LAS$) ELSE LAS=
      FIRS+(VAL(MID$(LAS$,2)))
1220  IF LAS < FIRS THEN 1210 ELSE IF FIRS > 65535 THEN 1210 ELSE IF
      LAS > 65535 THEN 1210 ELSE PRINT "Parameters accepted": PRINT
1230  PRINT "Data starts";TAB(30);"Data ends";TAB(50);"Currently programming":
      PRINT "HEX ";HEX$(FIRS);TAB(30);"HEX ";HEX$(LAS),
1240  FOR I=FIRS TO LAS: BLOW=PEEK(I): LOCATE 50,17: PRINT HEX$(I);:
      GOSUB 5010: NEXT: GOTO 32760
5000 '
      **       This is the actual BLOW subroutine. The contents of      **
      **  BLOW variable are programmed into the EPROM and verified      **
5010  LOCATE 1,20: PRINT "BV ";HEX$(BLOW);TAB(10);"LA "; HEX$(LOCAL);"    ":
      WHILE (INP(&F8E4) AND 1)<> 0: WEND: OUT &F8E2,(BLOW AND 255):
      WHILE (INP(&F8E4) AND 1)<> 0: WEND
5020  LOCAL=INP(&F8E0)+(256*(INP(&F8E1))): OUT &F8E3,&89 '
      ** Save the local address value, change PPI mode ready to verify **
5030  OUT &F8E0,(LOCAL AND 255): OUT &F8E1,INT(LOCAL/256): CHK=INP(&F8E2):
      IF CHK <> (BLOW AND 255) THEN GOSUB 6000
5040  OUT &F8E3,&80: LOCAL=LOCAL+1: IF LOCAL > &1FFF AND ETYP=2764 THEN
      5060 ELSE IF LOCAL > &3FFF AND ETYP=27128 THEN 5060 ELSE
      OUT &F8E0,(LOCAL AND 255): OUT &F8E1,INT(LOCAL/256)
5050  BBYTES=BBYTES+1: RETURN
5060  LOCATE 1,18: PRINT CHR$(20)+"EPROM filled up, please place another";
      ETYP;"in the programmer": PRINT "Press any key to resume run - or $ key"
      " to abort": X$="": WHILE X$="": X$=INKEY$: WEND: IF X$="$" THEN 5080
5070  OUT &F8E0,0: OUT &F8E1,0: LOCATE 1,18: PRINT CHR$(20): GOTO 5050
      LOCAL=0: '** Restart with new chip in pgmr, and local address **
              ** set to zero                                        **
5080  LOCATE 1,18: PRINT CHR$(20);:INPUT "Sure you want to abandon the run";
      R$: R$=UPPER$(LEFT$(R$,1)): IF R$ <>"Y" THEN 5060 ELSE
      OUTCOME$="-Abandoned at user request": GOTO 32760
6000  LOCATE 1,20: PRINT TAB(10);"** Failure at EPROM local address";LOCAL
      ;"**": LOCATE 1,22: PRINT "Details as follows: Wrote byte value";
      BLOW;"(" BIN$(BLOW) ")": OUTCOME$="" '** Clear the brag string
6010  PRINT "Byte value read from EPROM was ";CHK;"(";BIN$(CHK);")":
      PRINT: PRINT "Do you wish to continue?": X$="": WHILE X$="": X$=INKEY$:
      WEND: X$=UPPER$(LEFT$(X$,1)): IF X$ <> "Y" THEN 32760
6020  LOCATE 1,20: PRINT CHR$(20): RETURN
32760 LOCATE 1,5: PRINT CHR$(20);"Place READ/PROG switch into READ position":
      WHILE (INP(&F8E4) AND 2) <> 0: WEND: LOCATE 1,5: PRINT CHR$(20);
      "Thank you - **  Program run completed ";OUTCOME$;" **"
32765 PRINT CHR$(20);CHR$(24);BBYTES;"bytes blown -Write this down NOW! ";
      CHR$(24)
32767 END
```

Fig. 3.69: BASIC program to "Blow" data into an EPROM in the programmer.

Notes on usage

The EPROM programmer is a very handy thing to have for your CPC. It will enable you to develop applications which would otherwise be impossible. It will, for example, enable you to develop a stand alone project on your CPC. We are working towards customised ROM chips – which is really what an EPROM amounts to.

Another application is to blow code conversion ROMs for use in various hardware projects. For example you could create a hardware configuration whereby you have to plug in a special EPROM to descramble data back to text in security sensitive equipment. The way this might work is that you enter a code on the address lines of the EPROM, and the encoded equivalent comes out of the data lines of the EPROM. A frequent requirement is to convert ASCII to another of the various international codes. Figure 3.70 shows a rough diagram of how to use a specially programmed EPROM in code conversion.

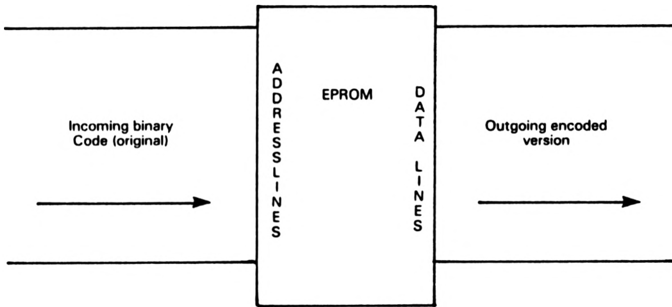


Fig. 3.70: Diagram showing the concept of an EPROM as a code convertor.

You could program an EPROM with data for use in an expert system, where the EPROM contained the various questions and actions which a program (maybe in another EPROM) would access and print to the user of the expert system.

An EPROM containing codes to be fed into a sound synthesiser chip like the AY–8192 in the CPC could be blown. If you use a ZIF socket you have the basis of an electronic musical box.

You can store relatively static information in an EPROM, like a list of telephone numbers and addresses. This would then be always instantly available for printing out to your screen.

You can build a ROM disk...... But now we are jumping ahead on to the next project!

The parts list for the EPROM programmer is given in Figure 3.71, and the voltage connection table for the chips used in the design is Figure 3.72.

Capacitors

C1	22 mfd	10 Volt Radial	(E.G RS components 402-248)
C2-C5	0.1 mfd	10 Volt disc	(RS 124-178 four required)

Resistors

R1	5K6 Ohms	quarter Watt	(RS 131-340)
R2	270 Ohms	quarter Watt	(RS 132-359)
VR1	2K Ohms	Variable resistor	(RD 162-215)

Semiconductors

LED1	Flashing red LED.	(RS 587-080)
LED2	Yellow LED	(RS 586-497)
E1	TTL 7425 Dual four i/p NOR + strobe	(RS 304-122)
E2	TTL 74LS20 Dual four i/p NAND gates	(RS 307-553)
E3	TTL 74LS138 3 to 8 line decoder	(RS 307-648)
E4	TTL 74LS02 Quad two i/p NOR gates	(RS 307-496)
E5	TTL 74LS04 Hex inverters	(RS 307-503)
E6	8255A PPI chip	(RS 309-363)
E8	TTL 74LS126 quad tristate buff chip	(RS 308-247)
E9	TTL 74LS123 Dual retrig monostable	(RS 307-632)

Miscellaneous

<u>Sockets for all chips</u>

28 pin ZIF socket.	(RS 402-248)
SW1 Double pole 3 position slide switch, right angled connection pins.	(RS 334-117)

Printed circuit board. (See note two below)

SKT 1	Small clip on connector.	(RS 434-144)
	Pins for SKT 1	(RS 434-138)
Expansion bus connector		(RS 468-119)

NOTES:

1) Note that the RS part numbers quoted may be supplied in packs containing more than one part.

2) Complete kits of parts are to be available from Halstead designs LTD.

Fig. 3.71: Parts list for EPROM programmer project.

E number	Device		description	Power connection pins			
	Type	Equivs	Function	+5V	+12V	−12V	OV
1	74LS25	–	Dual 4/nput Not gate	14			7
2		–	Dual 4 I/P NAND	14			7
3	74LS138	–	3 to 8 line decoder	16			8
4	74LS02	–	Quad 2 I/P NOR	14			7
5	74LS04	–	Hex Invertor	14			7
6	8255A		PPI chip	26			7
7	–		Zif Socket	28			14
8	74LS126	–	Quad tristate buff	14			7
9	74LS123	–	Dual monostable	16			8
10							
11							

Fig. 3.72: Voltage supply pins for chips used in EPROM programmer.

Conclusion

Many retailers sell EPROM erasers in boxes. These contain the correct type of UV lamp or tube, and usually a timer to turn on the UV light source for long enough to erase any EPROMs which you put in the box. It would have been nice to describe how to build such a unit in this book, but the space available dictates a limit on the number of projects which can be included. Another point to bear in mind regarding erasure is that direct sunlight shining into an EPROM window can eventually erase parts of it, so don't store your programmed EPROM chips on the window sill or in a car!

Project 9: ROM disk

With the appearence of large EPROM devices like the 16K x 8 bit 27128 EPROM, the possibility of replacing a read only disk with a number of EPROMS has been created. A moments thought about the nature of home computing will show that you probably load the same few programs into memory time and again. These may be games, assemblers, or word processing programs, and so on. Loading from cassette is very time consuming, the more so the bigger the program gets. Loading from disk is fast, but a ROM disk is faster. Over a period of time using a ROM disk can save disk wear and tear – since for some programs you may not even need to load a disk into the drive. As to cost, diskettes are cheaper than EPROM chips, and can be reused far more times, but the initial outlay for a ROM disk is lower than for a disk drive unit (even at the very low prices for Amstrad drives!), so it will be attractive to some readers on that score alone. The EPROM approach really comes into its own for storing medium sized amounts of static information.

The ROM disk concept is that you have a board containing lots of EPROM chips, each of which has a regularly used program blown into it. Becuase of the modular way in which the CPC software is written, it is pretty easy to patch in extra commands contained in an expansion ROM routine which cause the loading of software from the EPROMs into main memory.

The ROM disk design which we will now look at uses the sideways or LOCAL bus concepts outlined in the EPROM programmer. Becuase you need a way to blow EPROM chips with the required programs you will obviously have to build the EPROM programmer as well as the ROM disk board, and the expansion ROM board to house the software which will implement the new commands.

The design objectives for the ROM disk are as follows:

1) To be able to accomodate up to eight EPROM chips.
2) Expansion to readers own EPROM boards to be catered for.
3) An expansion ROM contains various routines to allow easy access to the files stored on the ROM disk.
4) To use up as little of the I/O address space as possible.

The second point above is included because it can easily be imagined that some readers would want more than eight programs stored in the ROM disk. This means that there would be a requirement for a means of adding another EPROM board. The design which has been developed allows you to connect up your own EPROM board, built on a strip board. Using this method a total of up to 255 EPROM chips can be added! This gives the potential of over four megabytes (four million bytes plus) of storage, using 255 of the 27128 chips. Figure 3.73 shows this arrangement in block diagram form.

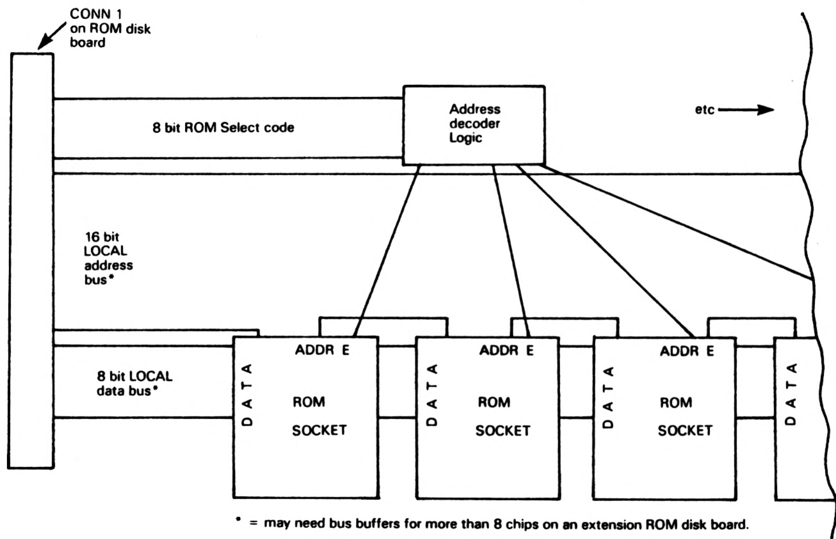


Fig. 3.73: Block diagram showing how extra ROM boards can be connected to the ROMdisk.

EPROMs are LSI (Large Scale Integration) chips and therefore consume a relatively large amount of amperage from the +5 Volts line. Before you actually decide how many EPROM add on boards you want to have, make sure that you will not be overloading the +5 volt line of the SIGMA PSU. Refer to the table in Figure 3.74 which shows the +5 volt consumption of each of the EPROM chips we are using here, and how much you would draw for each 4 additional chips. (The published figures for the 2764 and 27128 are just about identical on power consumption.) These are the figures for when a chip is deselected (IE its CE input is high). When an EPROM is actually selected it consumes just over twice as much, but since the design ensures that only one can ever be selected at a given instant this is not a major consideration.

EPROM current consumption table

Chips	AMPS	Chips	AMPS	Chips	AMPS	Chips	AMPS
8	0.4	72	3.6	136	6.8	200	10
12	0.6	76	3.8	140	7	204	10.2
16	0.8	80	4	144	7.2	208	10.4
20	1	84	4.2	148	7.4	212	10.6
24	1.2	88	4.4	152	7.6	216	10.8
28	1.4	92	4.6	156	7.8	220	11
32	1.6	96	4.8	160	8	224	11.2
36	1.8	100	5	164	8.2	228	11.4
40	2	104	5.2	168	8.4	232	11.6
44	2.2	108	5.4	172	8.6	236	11.8
48	2.4	112	5.6	176	8.8	240	12
52	2.6	116	5.8	180	9	244	12.2
56	2.8	120	6	184	9.2	248	12.4
60	3	124	6.2	188	9.4	252	12.6
64	3.2	128	6.4	192	9.6	256	12.8

Fig. 3.74: Current consumption table for 1 to 255 EPROM chips.

We need allocate only five addresses in the I/O address space to the ROM disk. The ones which have been chosen, and their uses are as follows:

Hex FAE0 = PPI port A – used to drive bits 0–7 of LOCAL addr bus
Hex FAE1 = PPI port B – used to drive bits 8–15 of LOCAL addr bus
Hex FAE2 = PPI port C – used to drive bits 16–23 of LOCAL addr bus
Hex FAE3 = PPI control byte – All ports are outputs.
Hex FAE8 = Data read register, reads state of the LOCAL DATA bus

As we shall see in the section on ROM disk software, the first ROM must always be installed, and is used as a directory for the ROM disk.

ROMdisk circuitry

The ROMdisk electronics is shown in block diagram form in Figure 3.75. This shows that the ROMs data and address lines are connected to the locally generated address and data busses. The local address bus is generated by the PPI, and the data lines of all ROMs are buffered onto the main CPC data bus by a tri state buffer chip. Additionally the local data bus and local address bus

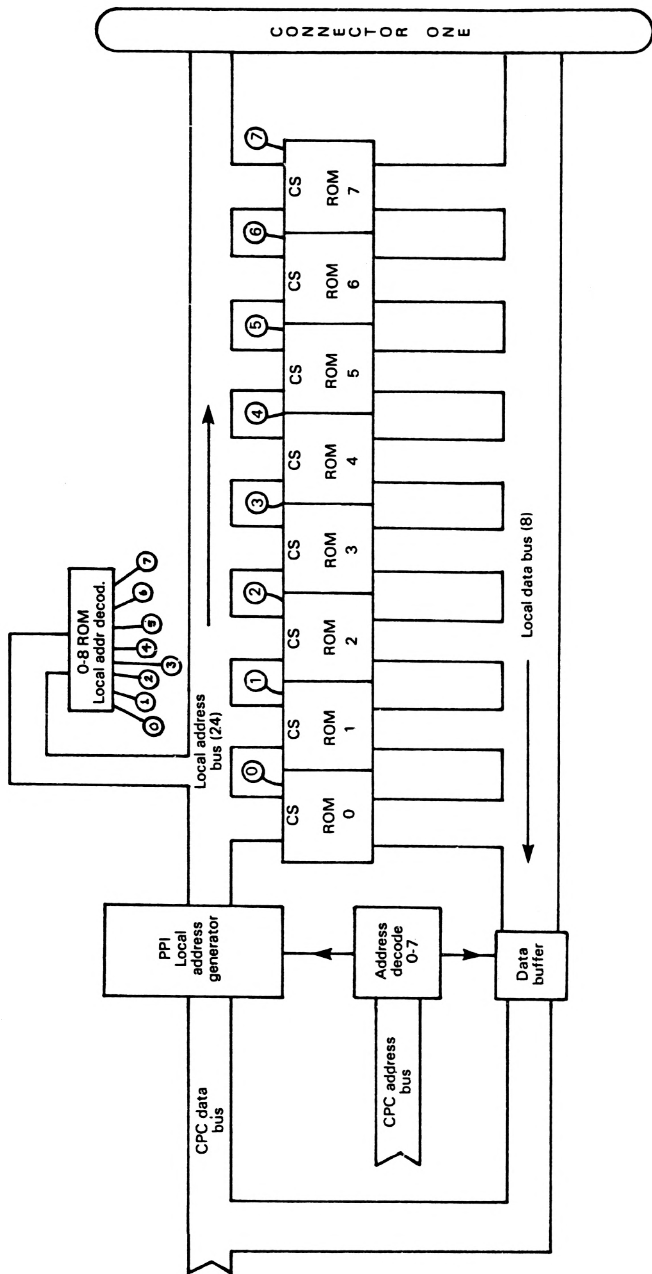


Fig. 3.75: ROMdisk block diagram.

Fig. 3.76: ROMdisk address decoder.

are continued out to connector one which will allow you to build your own extensions to the ROMdisk, should you need to. Figure 3.76 is the first section of the actual circuitry. This shows the address decoder. This particular decoder is all done with gates (or mirrors or something!). The bit states required to form the lowest three digits of the address are shown in the boxed diagram at the bottom of the figure. The net effect of all this gating is that pin 3 of gate E3a will go low when any of the PPI registers are addressed, and pin 4 of inverter E4b will go low when the separate data register is addressed. The purpose of the gating around the transistor TR1 is to detect when data is being read from the ROMdisk, and turn on the transistor thus pulling the expansion bus signal DOUT to its active low level. (See appendix five for a description of the expansion bus signals— including DOUT). Incidentally chip E4 is not a 74LS04 inverter chip, but an open collector version of the same thing, thus the pull up resistors on its outputs.

Moving onward now to Fig 3.77 which shows the PPI chip and part of the expansion connector— CONN one (of which more later). The select signal sent from the address decoder is connected into the CS bar input of the 8255A. The WR bar and RD bar signals from the expansion bus are fed straight into the matching inputs on the PPI. The RESET signal from the expansion bus is inverted by E4c before going into the PPI RESET input. This is because the PPI needs an active high reset signal. The data lines into the PPI are simply connected to the expansion bus data lines, and the A0 and A1 inputs to the PPI are connected to the expansion bus A0 and A1 address lines. This means that once the address decoder has decoded an address which begins with something A E , and address bit 3 is low, then which internal PPI register is selected depends entirely on the state of the A0 and A1 address lines. As we saw earlier the three ports A,B, and C are used in ascending order to generate the 24 bit local address. The local address lines are connected out of CONN 1 as well, so that extensions to the ROMdisk can be easily made.

The next ROMdisk diagram is the local address decoder, this is shown in Fig 3.78. Because the EPROM chips we are using have only twelve or thirteen address lines we have to decode the top eight bits of the local address to produce chip select signals to be passed to the required EPROM. The invertors E4d—E4f are connected together in a wired OR configuration, so that if any one of them has a low output then the output of all the others will be low as well. If the inputs of these invertors are low then the outputs are all turned off and the resistor R1 can pull the G1 input to E6 high. The local address lines LA16—LA18 are fed into the A,B, and C select inputs of the decoder, and so long as LA19 and LA20 (fed into the G2 inputs of the decoder) are low too, then the appropriate Y output will go low to select one of the eight on board EPROM chips.

The final diagram for the ROMdisk hardware is Fig. 3.79. This shows the local data bus buffer chip E7, and typical EPROM socket wiring. To begin with the bus buffer, when the Z80A addresses I/O address xAE0—xAE3 then E7 is enabled, by pulling its G bar input low, to pass through the contents of the

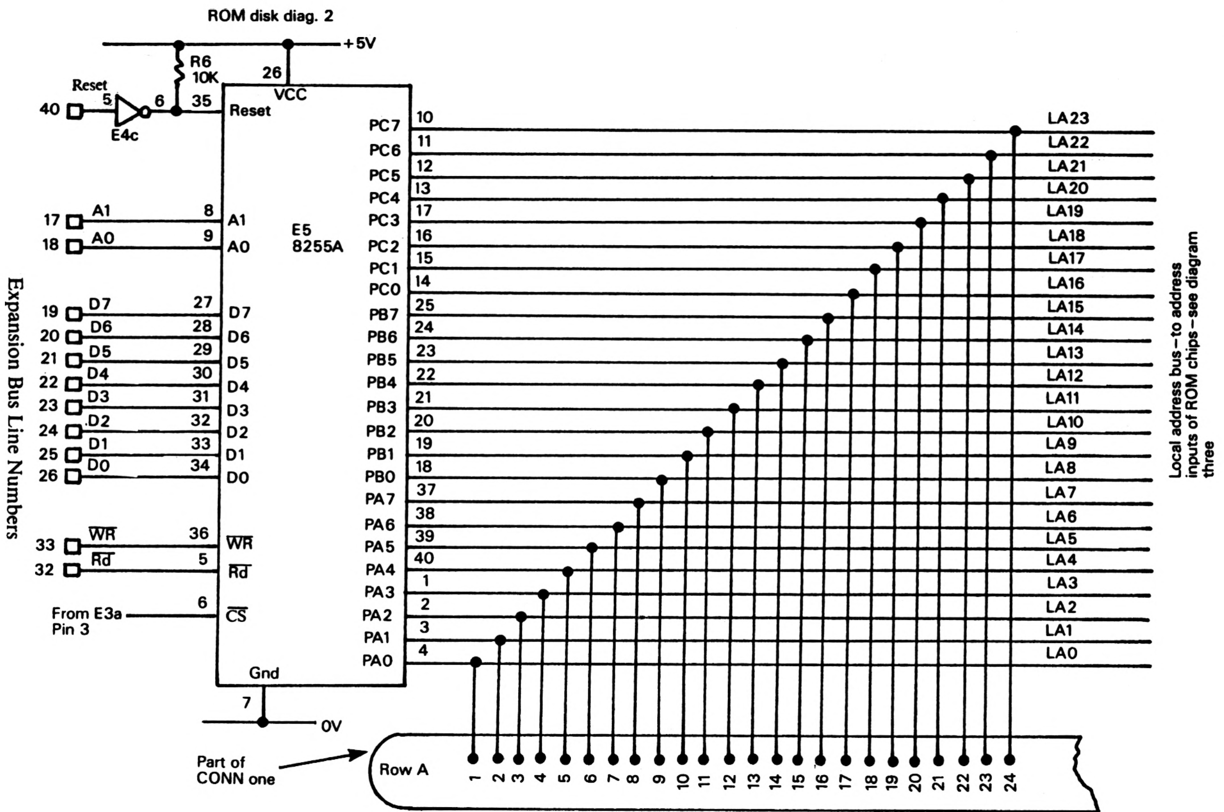


Fig. 3.77: ROMdisk PPI chip and expansion connector.

local data bus. This happens when the address decode logic detects a read from the ROMdisk. The EPROM sockets are connected in parallel, except for their CE bar inputs. Pin 20 of each EPROM is connected to the local address decoder. This means that since the local address decoder is only capable of outputting one enable signal at a time, only one EPROM can ever be enabled at a given time. The local data bus is continued out to CONN one as well, so that extra boards of EPROMs can be added when all the eight sockets on the ROMdisk board are in use. The capacitors C1–C5 decouple and smooth the supply lines to the circuit.

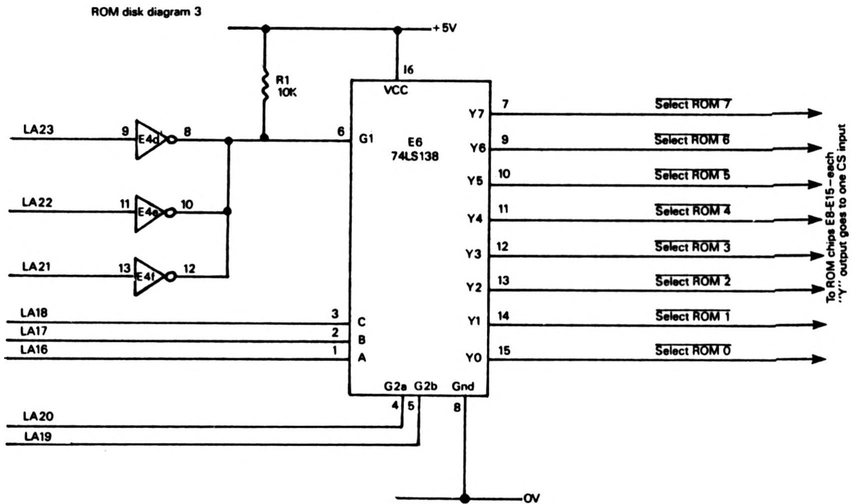


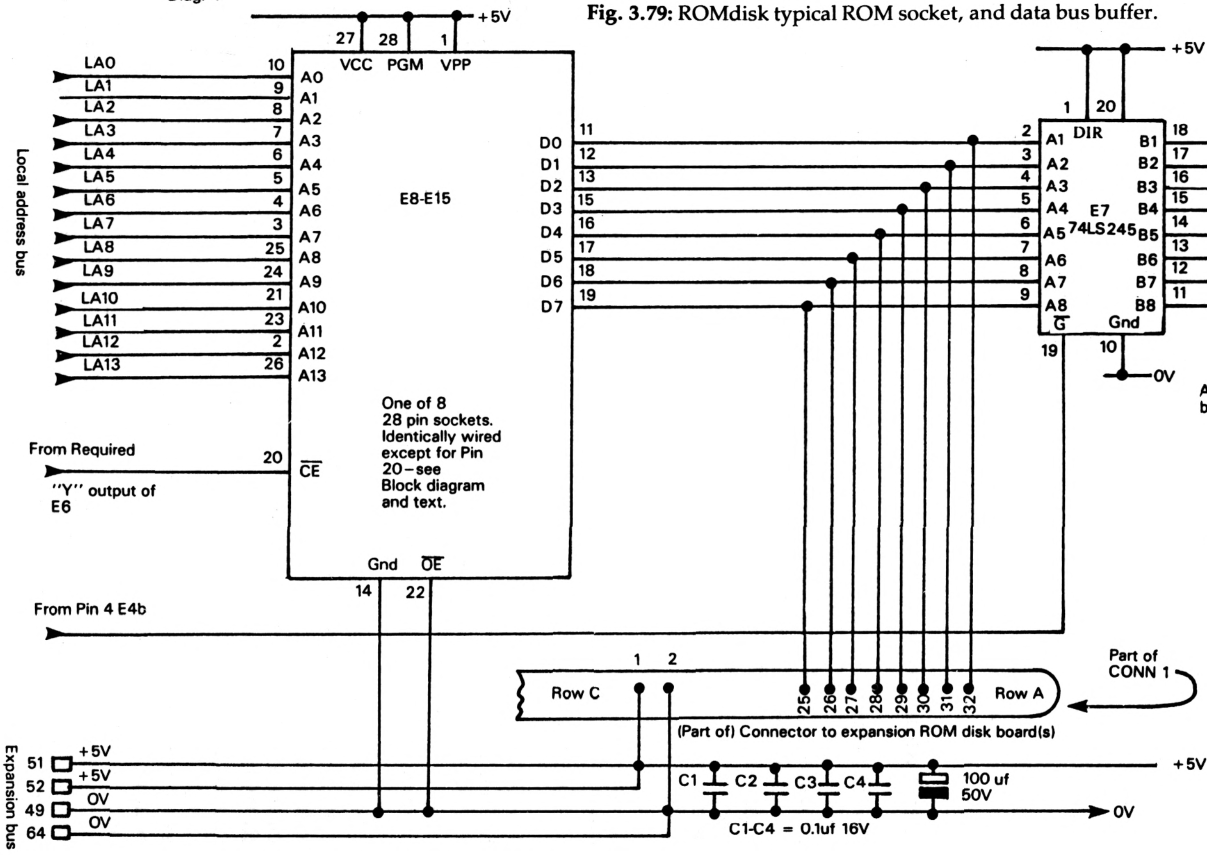
Fig. 3.78: ROMdisk LOCAL address decoder.

ROMdisk extension board requirements

As mentioned previously the connector CONN 1 is to allow the connection of your own ROMdisk extension boards. This will be of use where the eight on boards sockets are not sufficient for all the things you wish to store in the ROMdisk.

Extension ROMdisk boards need only contain a local address decoder to select the required ROM numbers (For a first add on board this will mean continuing on from the last ROM on the main board– ROM 7, so the first one on the extension board should be ROM 8). To allow the addition of further extension boards after the first, each extension board should ideally have an equivalent of CONN 1 – that is the local address and data buses should be continued. Please note that no PCB is expected to be available for extension boards, so readers will have to make their own from stripboard, or using other construction methods.

Fig. 3.79: ROMdisk typical ROM socket, and data bus buffer.



ROMdisk software

The ROMdisk software will be very difficult to understand if you are not conversant with the CPC firmware. This is detailed in a publication sold as SOFT 158 by Amsoft. If you do not have access to details of the firmware you can still "load and go" with this software but you will have difficulty understanding it in detail.

Becuase of the fact that the ROMdisk software is so deeply bound up with the CPC firmware, the detailed description will assume that you have access to SOFT 158. The general description will be sufficient to give readers details on the extra commands and facilities provided by the addition of the ROMdisk software.

General description

The ROMdisk software is contained in what is called a background ROM. This is an EPROM with software blown into it to run the ROMdisk and provide several extra commands, which can be included in programs, or typed at the keyboard. This background ROM actually has to live on the expansion ROMs board which forms the next project. In broad terms each background ROM can be switched, under CPC firmware control, into the address space normally occupied by BASIC, that is Hex C000 to Hex FFFF. In spite of occupying the same address space BASIC and the ROMdisk firmware can communicate with each other, via routines which are in the lower ROM. As with all extra commands the vertical bar "|" (shift @) should precede them. The new commands are introduced alongside the descriptions of the features they provide below.

The ROMdisk will provide an extra file storage medium for binary and ASCII format files on a CPC machine. In practical terms this means that you can load BASIC programs which have been blown into an EPROM (using the EPROM programmer). You can load machine code programs (like DEVPAC) or your own data from the ROMdisk. Let us look at how each of these options works.

CPC BASIC has the capability to load programs stored as ASCII text and then convert them into its own internal format, after which they can be executed. When you save a program with the format– SAVE "PROGRAM",A – you cause an ASCII version of your program to be saved to disk or tape. What the ROMdisk firmware does is intercept requests from BASIC for loading of files, caused by you typing a LOAD RUN MERGE or OPENIN command, and feed programs saved in textual form from the ROMdisk back to BASIC. In esscence the ROMdisk firmware replaces certain of the CPC firmware routines and BASIC is unaware of the difference. This kind of diversion is only possible because of the open software policy which Amstrad have followed. To create the diversion use the command "|RD.OPEN " this opens the ROMdisk to BASIC. When you want to go back to loading from disk or tape the command "|RD.CLOSE" will remove the diversion.

Loading machine code (or binary data) is accomplished with the "|RD.LOAD,R,A" command. Where R is the ROM number to load from, and A is the address to begin loading at. So, for example, the command "|RD.LOAD,5,1000" would load ROM 5 into memory beginning at address 1000. As ROM zero contains a directory of the ROMdisk, the RD.LOAD software knows how many bytes are used in each EPROM chip and loads just that many bytes into CPC memory. RD.LOAD will tell you if there is not enough memory to allow the selected ROM to be loaded into memory. If the directory entry for the EPROM socket is not created, you will never be able to use RD.LOAD to load its contents into memory, as RD.LOAD will give you the "** that ROM slot is not populated **" error message. Similarly if you specify an illegal ROM number in the command an error message is issued to tell you so.

You can find out what is actually present on the ROMdisk by use of the final of the new commands associated with the ROMdisk software. This last one is the command "|RD.DIR". This produces a one line entry, displayed on the CPC screen, for each entry found in the directory ROM – plugged into the ROM zero socket. If you need a printout of this directory, then you should use the command "|RD.DIR,P" which not only displays the directory on the CPC screen, but also outputs a copy to the printer port as well. (see discussion of the directory ROM below). From the information provided by use of this command you can tell which ROM to load a binary program from. This is useful as they cannot be accessed by filename, but only by the ROM number where it resides.

ROMdisk directory

The usual type of information which you will want to store in the ROMdisk will be of the sort from which you do not need to modify or remove anything. When you want to add new ROMs you must update the contents of ROM zero, which is the directory for the whole thing. Since the need to update the directory ROM should be relatively infrequent, it should be Ok to use a standard DIL socket for ROM zero, as with the others. Plugging in and unplugging ROM zero from its socket should not normally cause any problem, but if you intend to do it a lot it may be best to make the ROMdisk ROM 0 socket a ZIF socket. If you are thinking in terms of regularly updating the contents of your ROMdisk, that is OK but you should bear in mind the following points.

1) Use a ZIF for ROM position zero on the ROMdisk board.
2) make sure that you have the contents of ROM zero saved to tape or disk in a binary file.
3) Use the program listed in Figure 3.80 to add new entries. This will require an EPROM blower. (see previous project).

If you bear these points in mind then there should be no problem with regular updates to the ROMdisk contents. The most important thing to remember at all times is that the directory ROM should always be accurate. This is your responsibility, you must update it when you add another EPROM.

```
100 '          **    ROMdisk directory ROM update program    **

150 '** This program adds new entries to the directory ROM on
     ** the ROMdisk board. The directory ROM has to be unplugged
     ** from the ROMdisk board, and plugged into the EPROM programmer.
200     MODE 2: PRINT STRING$(79,&2A):       PRINT TAB(20);
        "ROMdisk directory update program": PRINT STRING$(79,&2A)
250     LOCAL=0: GOSUB 10000: IF RED <> 0 THEN 300 ELSE LOCATE 1,5:
        PRINT "Place the directory EPROM in programmer then press ENTER":
        WHILE R$ <> CHR$(13): R$=INKEY$: WEND: GOTO 250
300     LOCATE 1,5: PRINT CHR$(20): LOCAL=&20: ENTRY=1 '
        ** Clear screen and make LOCAL = first entry local address
        ** and ENTRY points to entry one (Zero being reserved)
400     GOSUB 10000: IF RED <> &21 THEN 500 ELSE ENTRY=ENTRY+1:
        LOCAL=ENTRY*32: IF ENTRY < 256 THEN 400 ELSE PRINT
        "This Directory ROM is full !!": GOTO 32767
500     PRINT "Create new entry for ROM slot";ENTRY;"<Yes> ";:
        INPUT R$: IF R$="" OR UPPER$(LEFT$(R$,1)) ="Y" THEN 600
510     INPUT "Which slot number (1 to 255)";ENTRY :
        IF ENTRY > 1 AND ENTRY < 256 THEN 520 ELSE LOCATE 1,5:
        PRINT CHR$(20);"Out of range - try again": GOTO 510
520     LOCAL=(ENTRY*32): GOSUB 10000: IF RED <> &21 THEN 600 ELSE
        LOCATE 1,5: PRINT CHR$(20);"That entry is already used":
        GOTO 510
600    '** When we get here ENTRY contains the number of the entry
        ** to be created, and LOCAL contains the local address of
        ** its first byte.
610     LOCATE 1,5: PRINT CHR$(20);"ROM contents description entry:"
620     LOCATE 1,7: PRINT CHR$(18);"Filename of the data in ROM";ENTRY;:
        INPUT FIL$: IF LEN(FIL$) > 0 AND LEN(FIL$) < 17 THEN 630 ELSE
        LOCATE 1,22: PRINT "Filename must be 1 to 16 printable characters"
625     GOTO 620
630     LOCATE 1,7: PRINT CHR$(20);CHR$(13);:
        FIL$=FIL$+SPACE$(16-LEN(FIL$)): IF A <> 0 THEN 800
640     PRINT "Preferred loading address";CHR$(18);: INPUT PLADD:
        IF PLADD > 0 AND PLADD < 65536 THEN 650 ELSE  LOCATE 1,22:
        PRINT "That is not a memory address": LOCATE 1,7: GOTO 640
650     LOCATE 1,7: PRINT CHR$(20); "Now you must specify filetype:":
        RESTORE: PRINT: PRINT "Type one of:": PRINT: FOR I=1 TO 5:
        READ T$(I): PRINT TAB(10);T$(I): NEXT: INPUT "Filetype";TYP$
660     IF LEN(TYP$) < 3 THEN 650 ELSE TYP$=UPPER$(TYP$): FOR I=1 TO 5:
        IF TYP$ = UPPER$(LEFT$(T$(I),LEN(TYP$))) THEN 670 ELSE NEXT I:
        PRINT CHR$(7): GOTO 650
670     TYP=(I-1)
680     LOCATE 1,7: PRINT CHR$(20);"Now enter the file length in bytes."
        " If you do not": PRINT "know it just press the ENTER key";:
        INPUT LENGTH: IF LENGTH > 0 THEN 800 ELSE IF LENGTH < 0 THEN 680
690    '** This routine attempts to find out how much data is in the
        ** ROM whose number it finds in ENTRY. If the ROM is not yet
        ** plugged into the ROMdisk, or the ROMdisk is absent then it
700    '** defaults to a full 27128 - that is 16384 bytes as the byte
        ** count for the contents. The EPROM blow program tells you
        ** how many bytes it has blown, so always note it down!
710     OUT &FAE3,&80: OUT &FAE2,ENTRY: IF INP(&FAE8) <> 255 THEN 720 ELSE
        INPUT "Program cannot auto-size ROM contents, default to 16384";R$:
        LENGTH =16384: IF UPPER$(LEFT$(R$,1)) ="Y" THEN 800
715     LENGTH =8192 : INPUT "Default to 8192";R$:
        IF UPPER$(LEFT$(R$,1))="Y" THEN 800 ELSE GOTO 680
720    '** When we get here we have found ROM installed. Now we do a
        ** top down search of the ROM for the end of the FF's.
725     PRINT "Auto-size in progress": TOP=&1F '** We set top address to
        &1FFF for a 2764 first. If top address is used we set up for a 27128.
        Note:if EPROM is exactly filled this routine always fails.
730     FOR H=TOP TO 0 STEP -1: OUT &FAE1,H: FOR I=255 TO 0 STEP -1:
        OUT &FAE0,I: IF INP(&FAE8) <> 255 THEN 740 ELSE NEXT: NEXT
740     IF TOP <> &3F THEN 750 ELSE TOP=&1F: GOTO 730
750     LOCATE 1,7: PRINT CHR$(20);"Data size appears to be";(H*256)+I;:
        INPUT "bytes. Do you accept this estimate";R$: R$=UPPER$(LEFT$(R$,1)):
```

```
        IF R$="" THEN 750 ELSE IF R$="Y" THEN 760 ELSE 680
760     LENGTH=(H*256)+I
800    '** Now we have input all the required information. We now
        ** Print it all up on screen to verify it.
810     LOCATE 1,7: PRINT CHR$(20);: PRINT "Entry details": PRINT
        "1) Entry describes ROM in socket number";ENTRY:    PRINT
        "2) ROM contents known by filename ";FIL$
820     PRINT "3) Preferred load address ";PLADD: PRINT
        "4) File type is ";T$(TYP+1): PRINT
        "5) Data length is Hex ";HEX$(LENGTH);" -Dec";LENGTH;CHR$(10)
830     PRINT "Press ENTER to accept details or any other key to re-enter":
        R$="": WHILE R$ = "": R$=INKEY$: WEND: IF R$=CHR$(13) THEN 850 ELSE
        CLEAR: GOTO 200
850    '** Now we are ready to BLOW the info into the ROM
860     LOCATE 1,4: PRINT CHR$(20): PRINT "EPROM program phase:":
        IF (INP(&F8E4) AND 2) = 0 THEN PRINT
        "Turn PROG/READ switch on EPROM programmer to PROG"
870     WHILE (INP(&F8E4) AND 2) =0: WEND: PRINT "Thank you":
        OUT &F8E3,&80 '** PPI initialised and in PROG mode now.
880     BLOW =ASC("!"): GOSUB 11000: FOR XX=1 TO 16: BLOW=ASC(MID$(FIL$,XX,1)):
        GOSUB 11000: NEXT: BLOW=(PLADD AND 255): GOSUB 11000:
        BLOW=(INT((PLADD/256)) AND 255): GOSUB 11000: BLOW=TYP: GOSUB 11000
890     BLOW=(LENGTH AND 255): GOSUB 11000: BLOW=(INT((LENGTH/256) AND 255)):
        GOSUB 11000
900     LOCATE 1,5: PRINT CHR$(20);"Put switch back to READ":
        WHILE (INP(&F8E4) AND 2) <> 0: WEND: PRINT "Thank you.": GOTO 32767

10000 '** S/R to load a byte from the current local address
        ** of the ROM in the EPROM programmer (normally dir ROM).
10010  OUT &F8E3,&89: OUT &F8E0,(LOCAL AND 255):
       OUT &F8E1,(INT((LOCAL/256)) AND 255): RED=INP(&F8E2): RETURN
11000 '
       **       This is the actual BLOW subroutine. The contents of     **
       **  BLOW variable are programmed into the EPROM and verified     **
11010  WHILE (INP(&F8E4) AND 1)<> 0: WEND: WHILE (INP(&F8E4) AND 1)<> 0:
       WEND
11015  OUT &F8E3,&80: OUT &F8E0,(LOCAL AND 255):
       OUT &F8E1,INT(LOCAL/256)
11017  WHILE (INP(&F8E4) AND 1)<> 0: WEND: OUT &F8E2,BLOW:
       WHILE (INP(&F8E4) AND 1)<> 0: WEND
11020  OUT &F8E3,&89 '** Change PPI mode ready to verify **
11030  OUT &F8E0,(LOCAL AND 255): OUT &F8E1,INT(LOCAL/256): CHK=INP(&F8E2):
       IF CHK <> (BLOW AND 255) THEN 11100 ELSE LOCAL = LOCAL +1:
       GOTO 11200
11100  PRINT "** Verification failure **";CHR$(7): PRINT "Details:": PRINT
       TAB(10);"LOCAL address =";LOCAL;": Wrote";BLOW;": Read";CHK: PRINT:
       INPUT "Retry this byte";R$: IF UPPER$(LEFT$(R$,1))="Y" THEN 11000
11110  LOCATE 1,3: PRINT CHR$(20);"Program run aborting": GOTO 900
11200  RETURN
32000  DATA "BASIC","Binary","Screen dump","ASCII","Unknown"
32767  END
```

Fig. 3.80: ROMdisk directory update program.

Using the ROMdisk to store data to be worked on by BASIC programs is not difficult. All you need to do is lower HIMEM using the MEMORY statement, and then load all or part of the required ROM contents into memory, by incorporating a "|RD.LOAD,R,A" command in a program. The loaded data can then be read into strings with a FOR–NEXT loop like:

```
1000 FOR XX=25000 TO 25200: A$=A$+CHR$(PEEK(XX)): IF PEEK(XX)=13 THEN
     2000 ELSE NEXT XX
2000 REM When we get here a string has been read into A$
```

Bear in mind that the loaded data in this case should be textual, if there is no carriage return or line feed the maximum string length will soon be reached. With text there should be no problem. I am sure that readers can soon see how this idea can be extended to other methods of data input from the ROMdisk to BASIC.

Limitations

Because Locomotive BASIC has to be treated like a Black box when interfacing to it, it has not proved possible to emulate all the stream open and close functions. This has to do with lack of access to BASIC's stream descriptor tables. In view of this there may be some occassions when problems will occur whilst using the BASIC OPENIN statement, namely that the true end of file may be overshot, and that BASIC sometimes thinks that stream nine (#9) is already open. To get around this second problem it is a good idea to CLOSEIN before attempting to use OPENIN. If there are further releases of the ROMdisk software it is hoped that these small problems will prove possible to solve, but they do not hinder usage of the ROMdisk in any serious way.

The ROMdisk software will be available from the software supplier, along with the rest of the software presented in this book. (see page three for details). As well as the assembler listings for the various routines which comprise the software, a hex dump of the background ROM in which the software lives will be given at the end of the detailed description, so that readers with no assembler, and no money to buy the software, can enter it as machine code if they wish to do so.

Detailed description of ROMdisk software

*** NOTE: This section asssumes access to, or knowledge of the appropriate firmware manual for your CPC ***

The ROMdisk software, as previously stated, lives in a background ROM (actually an EPROM plugged into ROM 6 socket on the expansion ROM board – see next project). On power up the firmware searches for background ROMs and places them on its list of possible command handlers. If a command is typed, or encountered in a BASIC program which is not valid to BASIC, then the command tables of the background ROMs are searched for it. Each entry in the jumpblock at the beginning of the ROM contents corresponds to a command table entry, but the first jumpblock and nameblock entry are reserved for an initialisation routine which the firmware calls when the machine is powered up. So the first entry in the jumpblock in the ROMdisk background ROM, points to a routine which will initialise conditions for operation of the ROMdisk.

The main function of the ROMdisk initialisation routine is to grab some memory for its exclusive use. The CPC firmware will then pass it the base address of this memory area in the IY register every time it calls a ROMdisk routine. Briefly, this feature of the CPC firmware works in the following way:

1) The firmware detects that the background ROM is present during its power up initialisation routines.

2) The firmware calls the routine which is pointed to by the first entry in the jumpblock. This routine receives, in the IY register, the address of the upper limit of the free memory pool (in BASIC this is called HIMEM).

3) The ROMdisk (or other background ROM software – they all do the same) initialisation routine does its initialisation, and lowers the address in IY by the number of bytes it wishes to reserve. Then the routine returns to the CPC firmware with IY now pointing to the new upper memory pool limit. Now whenever the firmware calls a routine in the ROMdisk background ROM, it will pass this base address of the ROMdisk work area to the routine it calls. The arrangement allows flexibility, and imposes a discipline when writing background ROM firmware, since all tables have to be placed in memory at relocatable addresses, usually relative to the entry value of IY.

The ROMdisk initilisation module– RDINIT – is listed in Fig 3.81. This shows the routine to initialise the ROMdisk, and also shows the statements to generate the ROM header table which the CPC firmware expects to see in a background ROM. RDINIT also sets the operational mode of the ROMdisk PPI chip. This is programmed with a mode byte of Hex 80, so that ports A,B, and C are all outputs. As we have seen in the software for the EPROM programmer, whenever the PPI chip mode register is loaded all the output latches are cleared. This means that RDINIT does not have to clear the local address to zero, as the hardware does this when the mode byte is programmed. There are two other jobs which RDINIT does. The first is to assign itself Hex 100 bytes of memory as a work area, as previously discussed. The second job it does is to zero the ROM select register of the ROMdisk, so that after RDINIT has run the local address is set to zero, and ROM zero is selected. Finally RDINIT prints a initialisation message.

```
100          ORG   #C000
120  ;
130  ; ** ROMdisk software
140  ;
150 ADDRLO:  EQU   #FAE0 ; First define the ROMdisk register
160 ADDRHI:  EQU   #FAE1 ; addresses low and high.
170 ROMSEL:  EQU   #FAE2 ; Then the ROM select register
180 CTRLP:   EQU   #FAE3 ; And the PPI control port
190 DATAP:   EQU   #FAE8 ; And the ROMdisk data port.
200  ;
210 ROMNUM:  EQU   6 ; Define the ROM address we expect to live at
220          DEFB  #1 ; BACKGROUND ROM
230          DEFB  #1,#1,#0 ; Mark 1 Vers 1 rev 0
240          DEFW  NAMTAB ; Set pointer to name table.
250          JP    RDINIT ; Power on init routine
260          JP    RDOPEN
270          JP    RDCLOS
280          JP    RDDIR
290          JP    RDLOAD
300 NAMTAB:  DEFM  "RD INI" ; Init entry
310          DEFB  "T"+#80
320          DEFM  "RD.OPE"
330          DEFB  "N"+#80
```

```
340          DEFM "RD.CLOS"
350          DEFB "E"+#80
360          DEFM "RD.DI"
370          DEFB "R"+#80
380          DEFM "RD.LOA"
390          DEFB "D"+#80
400          DEFB 0 ; The end of name table marker.
410 RDINIT: DEC  H ; HL is upper memory limit grab some
420          LD   BC,CTRLP
430          LD   A,#80     ; Set ports A,B, and C to outputs
440          OUT  (C),A     ; Write mode into the ROM disk PPI.
450          DEC  BC        ; Now BC points at ROM select register
460          LD   A,#0      ; Preselect ROM zero
470          OUT  (C),A     ; Zero the register.
480          LD   IX,INIMES ; Flash the message
490          CALL OUTPUT
500          JR   INIDUN    ; And out
510 INIMES: DEFB #0A,#0D
520          DEFM "** ROMdisk V1.2 initialised **"
530          DEFB #0A,#0D,#0A,#FF
540 INIDUN: SCF  ; Set carry flag to show all OK.
550          RET  ; End of RDINIT
```

Fig. 3.81: RDINIT listing.

The 256 byte work area assigned by RDINIT is used to hold saved jumpblock entries, intermediate results, buffer pointers and so on. The complete map of its usage is listed in Fig 3.82. The bytes of the work area cannot be specified as specific addresses, since the values will vary according to what hardware is on your machine, version of BASIC and so on. Consequently all positions in the work area are referenced to the entry value of IY, which, as you will recall, always points to the base address of the work area on entry to ROMdisk routines. Fig 3.83 shows some other ROMdisk information which you should use as a ready reference when delving into the ROMdisk software, or making modifications to it.

The next module of the ROMdisk software is called RDLOAD. This is listed in Fig 3.84. The purpose of this routine is to load the contents of the ROM specified in the command into CPC memory, starting at the address specified in the command.

The first thing which RDLOAD does is to ensure that the command includes both a ROM number and a memory address to load the ROM contents into, if it does not then an error message is output to the screen. The next thing that is checked is that the passed ROM number is legal. That is, it is not zero (since this is reserved for the directory ROM), and it is not greater than 255 – the maximum ROM number. If the ROM number is legal the next check is to see that the entry in the directory ROM, which describes the ROM being loaded, is valid. If the directory entry is not valid then an error message is printed to the screen. The final check is that the ROM contents will fit into memory at the desired address. This involves adding the desired address to the length in the directory ROM entry, if the result exceeds Hex FFFF (the maximum memory address in the CPC), then the error message "** Loading at specified address would overflow memory! **" is output to the screen. Once all these tests are passed the ROM contents are loaded into memory at the user specified address.

```
     When the ROMdisk background ROM is initialised by the CPC firmware
at power on it reserves 256 (Hex 100) bytes of memory for its exclusive
use as a work area. This area of memory will  not  be  altered  by  the
firmware. Whenever an applicable command to the ROMdisk background  ROm
is entered, the firmware calls the appropriate ROMdisk command handling
routine, and passes to it the base address of the work area in  the  IY
register. This is why the usage of the work area is specified  relative
to the entry value of IY.

         WORK AREA USAGE RELATIVE TO IY REGISTER ENTRY CONTENTS
        -----------------------------------------------------------

     All pointers and counters are held low order byte first.

IY+#30-#32 CAS IN OPEN jumpblock entry save area (RD.OPEN uses this)
IY+#33-#35 CAS IN CHAR jumpblock entry save area (RD.OPEN uses this)
IY+#36-#38 contain the far address of OPENIN - placed by RD.OPEN
IY+#39-#3C contain the far address of INCHAR - placed by RD.OPEN


IY+#53        = Used as a flag byte by RDDIR to show  that  it  received
                some command parameters
IY+#54 & #55 = Used by OPENIN to save buffer pointer.
IY+#60        = Buffer into which LENTRY subroutine places the directory
 to             entry it has been requested to fetch.
IY+#7F
                Usage of this area is as follows:
IY+#60        =  Marker byte = "!" if this entry is valid (#FF if not)

IY+#61 to IY+#70 = 16 byte filename
IY+#71 to IY+#72 = Preferred loading address (low byte first)
          IY+#73 = Filetype code:

                                  00=ASCII version of a BASIC program
                                  01=Binary
                                  02=Screen dump
                                  03=ASCII file


          IY+#74 = Length of the data (low)
          IY+#75 = Length of the data (high)
          IY+#76 = Local address count (low) used by RDLOAD
          IY+#77 = Local address count (hi)  used by RDLOAD
          IY+#78 = Buffer contents count (Low) Used by RDINCHAR
          IY+#79 = Buffer contents count (Hi ) Used by RDINCHAR
          IY+#7A = Buffer pointer address (low) , , , , , , , ,
          IY+#7B = Buffer pointer address (hi)  , , , , , , , ,
IY+#7C-#7F Unassigned.
```

Fig. 3.82: ROMdisk work area usage.

The main point to bear in mind when using RDLOAD is that being a binary loader it takes no notice of what memory is already allocated to the CPC firmware, disk firmware, or even BASIC. This places the emphasis on the user to ensure that BASICs HIMEM is lowered enough to allow loading of the required amount of data. If you load over the system variables area the machine will most certainly crash! One of the most effective uses of RDLOAD is to load graphic displays directly into screen memory, this is very fast and with lots of screens saved onto the ROMdisk opens up the possibility of some very sophisticated and fast games. So RDLOAD is not difficult to use, but take care.

Addresses of register

PPI register	Usage	I/O address
PORT A	Local address low	FAE0
PORT B	Local address Mid	FAE1
PORT C	ROM select regsister.	FAE2
Control	PPI control	FAE3
-	Read data register.	FAE8

Port C effectively forms the highest 8 bits of a 24 bit address. This means that over four megabytes of ROM can be addressed in a fully expanded ROMdisk scheme.

ROMdisk data structures

These notes specify the format of an entry in the directory ROM of the ROMdisk, and the usage of the work area which is reserved for use by the ROMdisk modules.

ROMdisk directory ROM entry format

For each of the possible 255 ROMs in the ROMdisk scheme there is a 32 byte entry in ROM 0 which is the directory ROM. As new ROMs are added to the ROMdisk the directory ROM should be updated. Each entry has the following format:

```
Byte  0     = Marker byte. Set to ASCII "!" (Hex value 21)
Byte 1-16   = Name of contents - padded if required with spaces.
Byte 17-18  = Preferred loading address. (Low byte first)
Byte 19     = Data type
                      0=A BASIC program in ASCII
                      1=Binary
                      2=Screen dump
                      3=ASCII file which is not a program
                   4-FF=Not defined
Byte 20-21  = Length of data in the ROM (Low order byte first).
Byte 22-31 Undefined - reserved for future use.
```

Addressing the directory ROM to extract an entry

When the directory ROM is accessed the ROM select register should obviously be loaded with zero. Then the two local address registers should be set up according to the following rules:

```
Address bits 13 14 and 15 should all be set to zero.
Address bits 5 to 12  should be set to the number of the ROM
                      whose entry it is desired to load. This will
                      be a number from 1 to #FF.
Address bits 0 to 4   Should be set to the address within the
                      entry, to select one of the thirty two bytes
                      which make up the entry.
```

Fig. 3.83: ROMdisk information sheet.

```
1800 RDLOAD: CP   #2 ; Entry A is params count, See if 2
1810         JR   Z,LOAD01    ; jump if so
1820         LD   IX,ERR1     ; Else error time
1830         CALL OUTPUT      ; Print the error
1840         JP   LDEXIT      ; And go out
1850 ERR1:   DEFB #0A,#0A,#0D
1860         DEFM "** Need ROM number and load address in command **"
1870         DEFB #0A,#0D
1880         DEFM "** For example |RD.LOAD,3,&4E20 will load ROM 3 **"
1890         DEFB #0A,#0A,#0D,#FF
1900 LOAD01: PUSH HL
1910         PUSH DE
1920         PUSH IX ;        Make room
1930         LD   A,(IX+3) ; Get high byte of ROM number
1940         CP   #0        ; See if zero
1950         JR   NZ,MERR2; Jump if no
1960         LD   A,(IX+2) ; Get low byte of ROM number
1970         CP   #0        ; See if THAT is zero
1980         JR   NZ,LOAD02; Jump if not
1990 MERR2:  LD   IX,ERR2  ; Else error occurs
2000 COMMOP: CALL OUTPUT   ; Do the output
2010         POP  IX
2020         POP  DE        ; Restore stack
2030         POP  HL
2040         JP   LDEXIT   ; And away
2050 ERR2:   DEFB #0A,#0A,#0D
2060         DEFM "** ROM number must be between 1 and 255 **"
2070         DEFB #0A,#0A,#0D,#FF
2080 LOAD02: LD   D,(IX+2) ; Get low byte of ROM number
2090         CALL LENTRY; Get the ROMs directory entry loaded
2100         LD   A,(IY+#60)   ; Get the indicator byte
2110         CP   "!" ;          See if it is valid
2120         JR   Z,LOAD03     ; jump if so
2130         LD   IX,ERR3      ; Otherwise error message three
2140         JR   COMMOP       ; AND use common outputter
2150 ERR3:   DEFB #0A,#0A,#0D
2160         DEFM "** That ROM slot is not populated **"
2170         DEFB #0A,#0A,#0D,#FF
2180 LOAD03: LD   H,(IX+1)     ; Load the address to be loaded into HL
2190         LD   L,(IX)
2200         PUSH DE ;           D contains ROM number - save it
2210         LD   E,(IY+#74)  ; Get length from ROMs dir entry
2220         LD   D,(IY+#75)
2230         ADD  HL,DE        ; See if ROM contents will fit
2240         POP  DE           ; Reload DE whatever
2250         JR   NC,LOAD04    ; If no overflow then all ok
2260         LD   IX,ERR4      ; Else issue error message
2270         JP   COMMOP       ; Use the common output
2280 ERR4:   DEFB #0A,#0A,#0D
2290         DEFM "** Loading at specified address would overflow memory! **"
2300         DEFB #0A,#0A,#0D,#FF
2310 LOAD04: LD   (IY+#76),00
2320         LD   (IY+#77),00 ; Set the local address count to zero
2330         LD   BC,ROMSEL    ; Set the ROM select register
2340         OUT  (C),D        ; with contents of D (ROM number)
2350         LD   H,(IX+1)     ; Make HL the address to load into
2360         LD   L,(IX) ; again.
2370 LOAD05: LD   BC,ADDRLO
2380         LD   A,(IY+#76)
2390         OUT  (C),A
2400         LD   BC,ADDRHI   ; Now output the local address
2410         LD   A,(IY+#77) ; Get high address byte
2420         OUT  (C),A       ; OUT goes high byte
2430         LD   BC,DATAP
2440         IN   A,(C)       ; Get a byte of data from the ROM
2450         LD   (HL),A      ; Buffer it
2460         INC  HL          ; Bump buffer pointer
2470         INC  (IY+#76)    ; Inc local low address
2480         JR   NZ,LOAD06   ; If not zero then jump
```

```
2490            INC   (IY+#77)    ; Inc high byte of local address.
2500 LOAD06:    LD    A,(IY+#76)  ; Get low byte of LOCAL address
2510            CP    (IY+#74)    ; See if local addr same as length yet
2520            JR    NZ,LOAD05   ; No ? loop again then
2530            LD    A,(IY+#77)  ; Get high byte of LOCAL addr.
2540            CP    (IY+#75)    ; See if match
2550            JR    NZ,LOAD05   ; do it again if not.
2560 LOAD07:    LD    IX,DUNNIT   ; When we get here weve finished
2570            JP    COMMOP      ; So output a dunnit message and exit
2580 DUNNIT:    DEFB  #0A,#0D,#0A
2590            DEFM  "** ROM loaded as requested **"
2600            DEFB  #0A,#0D,#FF
2610 LDEXIT:    RET   ; After all is done we return
2620            LD    A,(IY+#77)  ; Else get high local addr
2630            CP    (IY+#75)    ; Compare to High length
2640            JR    NZ,LOAD05   ; Loop if no match
```

Fig. 3.84: RDLOAD program listing.

When the ROMdisk was being designed I thought the best way to implement the cataloguing of what it held would be to have a small header area in each individual EPROM chip. Further reflection on this idea revealed a rather bad side effect to doing this. As stated above, one of the most effective uses for the ROMdisk is very rapid loading of graphic screens. Because the CPC screen memory occupies 16K bytes, it therefore follows that 16K bytes of ROM would be required to store a complete screen of information. If we were to rob, say 32 bytes of this for a ROM header, the effect would be to spoil the display, either by leaving it a few pixels short, or by having nasty random dots on the screen. This led to the alternative of allocating one of the ROM slots on the ROMdisk to hold the directory for all the other ROMs.

The ROMdisk directory entry format, and how they are addressed was detailed in Fig 3.83. The only drawback to having a dedicated directory ROM is that whenever a new ROM is added, the directory ROM has to be updated to show the details of the new ROM. This is not a problem because the kind of data kept in a ROMdisk will always be pretty permanent, and so updating the directory ROM will, after the initial setting up phase, be infrequent. Having said all that we do need a way to use the EPROM programmer to update the contents of the directory ROM chip. Refer back to Fig 3.80 which shows the listing for a BASIC program which will, when the directory ROM has been plugged into the ZIF socket of the EPROM programmer, add an entry for a specific ROM number, then "blow" it into the directory ROM at the correct location.

The directory ROM should be a 2764, since only 8192 bytes (256 times 32 = 8192) are required. As with all the ROMdisk ROMs high speed versions are not essential, so the cheaper versions can be used with no problems.

The reason for explaining all about the ROMdisk directory arrangements at this point in the text is that the next routine of the ROMdisk software is RDDIR, which produces a directory of the ROMdisk based on the contents of the directory ROM. Fig 3.85 is a listing of RDDIR. Two forms of command are available from BASIC. These are "RD.DIR" which produces a directory listing

on the CPC screen. The second form is "RD.DIR,P" which causes a directory listing on both the screen and the printer. In fact you can type anything after the "RD.DIR" to get the printed directory, the program does not analyse the extra characters at all.

```
 560 RDDIR:   PUSH AF ; Make some room
 570          PUSH DE
 580          PUSH BC
 590          PUSH HL
 600          PUSH IX
 610          PUSH AF; Save AF for a moment
 620          LD   A,2 ; Set mode 2 up
 630          CALL #BC0E ; Call SCR SET MODE rtne
 640          POP  AF
 650          LD   E,A ; Save parameters count into E
 660 RDDIR5:  LD   BC,ROMSEL  ; Point BC at ROM select reg.
 670          LD   A,#0
 680          OUT  (C),A      ; Ensure ROM zero selected.
 690          LD   BC,ADDRHI ; Now point BC at ADDRHI register
 700          LD   IX,HEADER ; point IX at the header text
 710          CP   E; see if B was zero on entry
 720          CALL NZ,PRINTR; If not then printer is output
 730          LD   IX,HEADER ; Then repoint IX for screen
 740          CALL OUTPUT    ; else screen is output, use OUTPUT.
 750          LD   D,1 ; Now we use D as the entry to load count
 760 RDDIR1:  CALL LENTRY    ; Get an entry in the dir ROM
 770 FORM:    LD   A,"!"     ; See if the entry is valid
 780          CP   (IY+#60)  ; Compare "!" with first byte
 790          JP   NZ,RDDIR4 ; if not equal then jump
 800          PUSH IY
 810          POP  IX ; Copy IY to IX via stack
 820          LD   BC,#80
 830          ADD  IX,BC     ; Now IX points at o/p buffer
 840          LD   A,D       ; Get dir entry number we're listing
 850          CALL HEXAS     ; Convert to ASCII
 860          LD   (IX),H    ; Get high byte
 870          LD   (IX+1),L ; and low byte into buffer
 880          LD   BC,2
 890          ADD  IX,BC     ; Update o/p buffer pointer
 900          LD   B,6       ; Now six spaces please
 910          CALL SPACE
 920 ; Now IX is at IY+#88 - And buffer which entry was loaded
 930 ; into begins at IY+#60 so IX-#27  points at 2nd byte
 940 ; of the loaded entry, which is the filename.
 950          LD   B,16      ; 16 characters to copy
 960 NAMET:   LD   A,(IX-#27) ; Do a namebyte transfer
 970          LD   (IX),A    ; buffer it
 980          INC  IX        ; Update pointer
 990          DJNZ NAMET     ; Do some more ?
1000          LD   B,7       ; Now 7 spaces please
1010          CALL SPACE
1020          LD   A,(IY+#75); Fetch the length (high byte)
1030          CALL HEXAS     ; Convert it to ASCII
1040          LD   (IX),H    ; Buffer the high byte
1050          LD   (IX+1),L ; and the low
1060          LD   BC,#2
1070          ADD  IX,BC     ; Update  pointer
1080          LD   A,(IY+#74); Get length (Low byte)
1090          CALL HEXAS     ; Convert that too
1100          LD   (IX),H
1110          LD   (IX+1),L
1120          LD   BC,#2
1130          ADD  IX,BC
1140          LD   B,7       ; Now 7 spaces into buffer
1150          CALL SPACE
1160 ; Now we have to output filetype as text
1170          LD   A,(IY+#73); Get the filetype byte
```

```
1180            CP    #0        ; See if it is BASIC
1190            JR    NZ,BINCHK; jump if not
1200            LD    HL,BAS    ; Else make HL point at text
1210            CALL  XFER      ; And get it transferred
1220            JR    RDDIR8    ; then jump out
1230 BAS:       DEFM  "BASIC"
1240            DEFB  #80
1250 BINCHK:    CP    #1 ; See if it is binary
1260            JR    NZ,SDCHK    ; Jump if not
1270            LD    HL,BIN      ; If so HL =text
1280            CALL  XFER        ; Do the transfer
1290            JR    RDDIR8      ; And out
1300 BIN:       DEFM  "Binary"
1310            DEFB  #80
1320 SDCHK:     CP    #2 ; See if its a screen dump
1330            JR    NZ,ASCCHK   ; If no then jump
1340            LD    HL,SD       ; HL points at text
1350            CALL  XFER        ; Do the transfer
1360            JR    RDDIR8      ; And off we go
1370 SD:        DEFM  "Screen dump"
1380            DEFB  #80
1390 ASCCHK:    CP    #3
1400            JR    NZ,WHAT     ; If not ASCII then jump
1410            LD    HL,ASC      ; If yes load text addr to HL
1420            CALL  XFER        ; Do the transfer
1430            JR    RDDIR8      ; and out
1440 ASC:       DEFM  "ASCII"
1450            DEFB  #80
1460 WHAT:      LD    HL,UNKNOW   ; Load HL with text addres
1470            CALL  XFER        ; Place it
1480            JR    RDDIR8      ; And exit
1490 UNKNOW:    DEFM  "Unknown"
1500            DEFB  #80
1510 RDDIR8:    LD    (IX),#0D
1520            LD    (IX+1),#0A
1530            LD    (IX+2),#A0
1540            PUSH  IY ; Copy IY to IX again
1550            POP   IX
1560            LD    BC,#80
1570            ADD   IX,BC     ; Now IX points at o/p buffer again
1580            LD    A,E       ; Get the params flag byte out
1590            CP    #0        ; See if zero
1600            CALL  NZ,PRINTR ; Call printer output if not 0
1610            PUSH  IY
1620            POP   IX
1630            LD    BC,#80 ; Do the same again for
1640            ADD   IX,BC  ; The screen o/p
1650            CALL  OUTPUT ; Call screen output anyway
1660 RDDIR4:    LD    A,1    ; Now inc D (Entry count)
1670            ADD   A,D    ; Add D
1680            LD    D,A ; Put result back
1690            JP    NZ,RDDIR1; loop again if no overflow
1700 RDDIR7:    POP   IX     ; If past 255 then reload regs.
1710            POP   HL
1720            POP   BC
1730            POP   DE
1740            POP   AF
1750            RET   ; And return to BASIC (or whatever)
1760 HEADER:    DEFM  "ROM     Name                    Length"
1770            DEFM  "    Type."
1780            DEFB  #D,#A
1790            DEFB  #FF
```

Fig. 3.85: RDDIR program listing.

The first action performed by RDDIR is to call a firmware routine to set the screen into mode 2. Then it selects ROM zero, and decides whether it is outputting to just the screen or to both screen and printer. Next it prints a header for each entry catagory (Filename Length and so on). Then each of the possible 255 entries are loaded in turn, with the details being printed on screen and printer, as selected. The entries are loaded in descending order. If after loading an entry the first byte of the entry is found to be a "!" character then the entry is said to be valid, and its contents are displayed. A large part of the RDDIR listing relates to deciding on the filetype and outputting a textual type to match it. Refer back to Fig 3.83 for a list of the meaning of each possible byte value in the filetype slot of the entry.

RDDIR uses several routines from the ROMdisk subroutine library. Principal among these are HEXAS, which converts a single byte to two ASCII characters, and LENTRY, which loads into memory the directory entry whose number it receives from its caller in the D register. RDDIR also uses PRINTR and OUTPUT, which output the character they receive in the A register to the printer and screen respectively.

The final commands are all to do with loading and running programs from the ROMdisk to be used by BASIC. It may perhaps be helpful to explain how an ASCII version of a program is loaded by BASIC before we launch into describing the ROMdisk routines RDOPEN, OPENIN, RDCLOS, and INCHAR. I am indebted to Cliff Lawson of Amsoft who assisted greatly in sorting this sequence out.

When you ask BASIC to load a program which has been stored on disk or tape in ASCII, it calls the CAS IN OPEN firmware routine, supplying the address of the filename it wishes to be loaded, along with the address of a 2048 byte buffer where it wishes the first block of the file to be loaded. CAS IN OPEN duly does this and passes back to BASIC the address of a header table which contains all the information about the file. At this stage version 1.0 BASIC seems only to check the filetype passed back to it. Silly or null values in the other fields do not cause error messsages, though they cause hilarious results at the next stage! Next BASIC uses another routine called CAS IN CHAR to feed the contents of the 2048 byte buffer, a byte at a time, back to it. CAS IN CHAR feeds all 2048 bytes to BASIC, and then if the file consists of more than one block, loads the next block from tape (or disk if on a disk equipped CPC 464 or a CPC 664), and repeats the process. When all the blocks have been loaded and the number of bytes indicated in the length entry of the file header have been fed to BASIC, CAS IN CHAR sends an end of file character (Hex 1A) and certain flag values back to BASIC, which then ends its input and calls the CAS IN CLOSE routine.

All the time characters from the file are being sent by CAS IN CHAR a process of conversion to its internal token version is being carried out by BASIC.

The foregoing is not a precise description of how BASIC loads files, but is based on information found in the firmware manual from Amsoft Soft 158,

and by observation and experiment.

```
3750 RDOPEN:  PUSH AF
3760          PUSH HL
3770          LD   A,(#BC77)   ; Get and save the CAS IN OPEN
3780          LD   (IY+#30),A ; jumpblock entries.
3790          LD   A,(#BC78)   ; Save them at IY+#30-#32
3800          LD   (IY+#31),A
3810          LD   A,(#BC79)
3820          LD   (IY+#32),A
3830          LD   A,(#BC80) ; Now do the same for jb entry of
3840          LD   (IY+#33),A ; CAS IN CHAR
3850          LD   A,(#BC81)     ; Save these to IY+#33-#35
3860          LD   (IY+#34),A
3870          LD   A,(#BC82)
3880          LD   (IY+#35),A
3890          LD   A,(#BC7A)   ; Save orig contents of
3900          LD   (IY+#3C),A   ; return locations
3910          LD   A,(#BC83)
3920          LD   (IY+#3D),A
3930          LD   A,#C9       ; And place a RET instruction
3940          LD   (#BC7A),A ; in them.
3950          LD   (#BC83),A
3960          LD   A,#DF
3970          LD   (#BC77),A   ; Now put the restart instructions
3980          LD   (#BC80),A   ; into place in the jumpblock
3990          LD   HL,OPENIN   ; Now we must make far addresses
4000          LD   (IY+#36),L ; In the work area.
4010          LD   (IY+#37),H
4020          LD   H,ROMNUM      ; so the address, then ROM number
4030          LD   (IY+#38),H    ; get placed.
4040          LD   (IY+#3B),H
4050          LD   HL,INCHAR     ; And for RD.INCHAR too
4060          LD   (IY+#39),L
4070          LD   (IY+#3A),H
4080          PUSH IY
4090          POP  HL    ; Copy IY to HL
4100          PUSH DE   ; Now we need DE so save it
4110          LD   DE,#36 ; Use DE to make HL point at
4120          ADD  HL,DE ; first far address
4130          LD   (#BC78),HL ; Put it in jumpblock
4140          LD   DE,#3        ; HL points at 2nd
4150          ADD  HL,DE        ; far address
4160          LD   (#BC81),HL ; Put it in jb
4170          POP  DE ; Then all of a sudden we are done
4180          POP  HL ; so restore
4190          POP  AF ; and the
4200          RET  ; return is done
```

Fig. 3.86: RDOPEN program listing.

The ROMdisk command RD.OPEN causes patching of the jumpblock entries for CAS IN OPEN and CAS IN CHAR (or their disk counterparts). Fig 3.86 shows the RDOPEN routine. The entries are made into calls to the ROMdisk routines OPENIN and INCHAR respectively. Becuase of the way that the ROMdisk routines are accessed a single byte of the jumpblock entries for CAS IN CLOSE and CAS IN DIRECT are also altered, causing any calls to them while the patches are applied to return with no action taken. The previous values are stored in the ROMdisk work area so they can be restored when an RD.CLOSE command is issued. RDCLOS is listed in Fig. 3.87. Both OPENIN and INCHAR utilise a subroutine from the ROMdisk library called TWOKAY. As its name implies this routine loads 2K bytes from the currently

selected ROM. On entry it expects the IY register to point at ROMdisk work area. The HL register should point to the buffer where TWOKAY should load the 2K bytes. It takes the values it finds in IY+#76 and IY+#77 (low byte first) as the local address to begin loading from. After TWOKAY has run, the local address will have been incremented by 2048, as will the contents of the HL register. (TWOKAY is listed in the ROMdisk subroutine library listing at the end of this section.)

```
4210 RDCLOS: PUSH AF ; Save AF
4220         LD   A,(IY+#30) ; Get a byte
4230         LD   (#BC77),A  ; restore it
4240         LD   A,(IY+#31) ; Get a byte
4250         LD   (#BC78),A  ; restore it
4260         LD   A,(IY+#32) ; and so on...
4270         LD   (#BC79),A
4280         LD   A,(IY+#33)
4290         LD   (#BC80),A
4300         LD   A,(IY+#34)
4310         LD   (#BC81),A
4320         LD   A,(IY+#35)
4330         LD   (#BC82),A ; Until all restored
4340         LD   A,(IY+#3C)
4350         LD   (#BC7A),A
4360         LD   A,(IY+#3D)
4370         LD   (#BC83),A
4380         POP  AF ; The reload and return
4390         RET
```

Fig. 3.87: RDCLOS program listing.

After RDOPEN has run, any attempt by BASIC to load from tape or disk will result in the usage of OPENIN and INCHAR to load the file (if it exists), from the ROMdisk, in pretty much the same way that the firmware routines would load from cassette or disk. Fig 3.88 is the listing of OPENIN which loads the first 2048 bytes, INCHAR (listed in fig 3.89) feeds a character at a time to BASIC refilling the buffer as required, repeating the cycle until a byte with bit seven set is encountered whence it signals end of file to BASIC and the load completes.

```
4400 OPENIN: LD   A,B  ; First we see if silly user has given
4410         CP   #0 ; a null filename
4420         JR   NZ,OP001 ; Jump if not
4430         LD   IX,ERR01 ; Else load up an error text
4440         JP   BADEND   ; And share bad exit code
4450 ERR01:  DEFB #0A,#0D,#0A
4460         DEFM "** You MUST specify a filename for ROMdisk access **"
4470         DEFB #0A,#0D,#FF
4480 OP001:  LD   (IY+#54),E ; Save buffer address
4490         LD   (IY+#55),D
4500         LD   D,1 ; Start from ROM directory entry 1
4510 OP002:  CALL LENTRY ; Load the entry
4520         LD   A,(IY+#60)  ; Get marker byte
4530         CP   "!"         ; See if entry is valid
4540         JR   Z,OP005     ; Jump if yes
4550 OP004:  INC  D           ; Else increment dir entry counter
4560         JR   NZ,OP002    ; If not all searched then loop
4570         LD   IX,ERR02    ; If we get here file not found
4580 BADEND: CALL OUTPUT ; Common bad exit code starts here.
4590         LD   A,#0    ; Load zero to A
```

```
4600          ADD   A,#0    ; Ensure carry is clear
4610          CP    #0      ; Ensure zero flag set
4620          RET   ; And go back to our caller
4630 ERR02:   DEFB  #0A,#0A,#0D
4640          DEFM  "** File not in ROMdisk **"
4650          DEFB  #0A,#0D,#FF
4660 OP005:   PUSH  IY
4670          PUSH  DE ; Save entry number
4680          LD    DE,#0061 ; Now make IY point at filename
4690          ADD   IY,DE
4700          POP   DE        ; Reload DE
4710 ; ** Now IY points at loaded filename, and HL still
4720 ; ** points at required filename. B is reqd filename length
4730          CALL  COMPAR ; Call compare s/r
4740          POP   IY ; Reload IY anyway
4750          JR    NZ,OP004 ; Jump if compare failed
4760          LD    A,(IY+#73) ; Get filetype
4770          CP    #0  ; See if ASCII (typ=0 if yes)
4780          JR    Z,OP003     ; If ASCII then jump
4790          LD    IX,ERR03    ; Else error
4800          JR    BADEND      ; and out
4810 ERR03:   DEFB  #0A,#0D
4820          DEFM  "** Filetype error - this file is not ASCII BASIC **"
4830          DEFB  #0A,#0D,#FF
4840 OP003:   LD    BC,ROMSEL
4850          OUT   (C),D ; Select the desired ROM
4860          LD    E,(IY+#54) ; Reload buffer address to DE
4870          LD    D,(IY+#55)
4880          LD    (IY+#71),E ; put it into preferred load addr.
4890          LD    (IY+#72),D
4900          LD    (IY+#7A),E ; Buffer pointer =buff base please
4910          LD    (IY+#7B),D
4920          PUSH  DE ; Copy buffer address to HL
4930          POP   HL
4940          LD    (IY+#76),0 ; Zero LOCAL address count
4950          LD    (IY+#77),0
4960          CALL  TWOKAY    ; Call the 2K loader
4970          PUSH  IY ; Now copy IY to hl via stack
4980          POP   HL
4990          LD    DE,#61
5000          ADD   HL,DE
5010          LD    DE,#170 ; Location =Hex 170 and HL points at header
5020          LD    C,(IY+#74) ; Load BC with file length
5030          LD    B,(IY+#75)
5040          LD    A,#16 ; Put filetype into A
5050          CP    #0      ; Ensure the Z flag is clear
5060          SCF   ; Ensure carry is set
5070          RET   ; And then return
```

Fig. 3.88: OPENIN program listing.

You can use the BASIC commands LOAD, MERGE, CHAIN and OPENIN with the ROMdisk, though as previously stated there will be occasions when using the ROMdisk as stream 9 (#9) will give unexpected results. LOADing and MERGEing and CHAINing will always work for fetching BASIC programs saved as ASCII to the ROMdisk.

The subroutines which are used by various modules of the ROMdisk software are listed in Fig. 3.90. These are largely explained in the remarks on the listings, and in the text.

Finally the assembly for the whole of the ROMdisk software is listed in Fig 3.91.

```
5080 INCHAR: PUSH HL ; Save HL
5090         LD   L,(IY+#7A)    ; HL is now the buffer pointer
5100         LD   H,(IY+#7B)
5110         LD   A,(HL)        ; Get a character from the buffer
5120         INC  HL ; Bump the pointer
5130         LD   (IY+#7A),L ; Save pointer
5140         LD   (IY+#7B),H
5150         DEC  (IY+#78)      ; Decrement how many in buffer
5160         JR   NZ,INCH01     ; If not zero then skip
5170         DEC  (IY+#79)      ; else decrement counter high
5180         JR   NZ,INCH01     ; Jump if not zero yet
5190         LD   L,(IY+#71)    ; Else we make HL =buffer base
5200         LD   H,(IY+#72)    ; And then we
5210         PUSH AF            ; have to call TWOKAY to
5220         CALL TWOKAY        ; refill the buffer again.
5230         LD   A,(IY+#71)    ; Make pointer point at buffer
5240         LD   (IY+#7A),A    ; base address
5250         LD   A,(IY+#72)
5260         LD   (IY+#7B),A
5270         POP  AF            ; reload saved AF
5280 INCH01: BIT  7,A ; See if bit seven of char is set
5290         JR   Z,INCH02      ; Jump if not.. otherwise
5300         LD   A,#1A         ; File all loaded: send EOF back
5310         CP   #10           ; Ensure zero flag is clear
5320         JR   INCH03        ; and go to common exit
5330 INCH02: SCF  ;        Set carry
5340         CP   #FF           ; Ensure zero is clear to show OK
5350 INCH03: POP  HL            ; Common exit - restore HL
5360         RET  ;        And return.
```

Fig. 3.89: INCHAR program listing.

```
2645 ; **               RDLIB - The ROMdisk subroutine library **
2650 LENTRY: PUSH BC ; We need BC HL and DE so clr them
2660         PUSH HL
2670         PUSH DE
2680         LD   BC,ROMSEL ; First ensure ROM zero slectd.
2690         LD   A,0
2700         OUT  (C),A     ; Done.
2710         LD   A,D ; This routine loads one directory entry
2720         RRC  A    ; Entry D is the number of the entry
2730         RRC  A    ; See text for details on how an entry
2740         RRC  A    ; is addressed. Make low byte of address
2750         AND  #E0 ; Mask out unwanted bits
2760         LD   BC,ADDRLO
2770         OUT  (C),A ; Output the low address byte
2780         LD   A,D ; Now reload original ROM number
2790         AND  #F8 ; Mask out the lower three bits
2800         SRL  A    ; Shift  by three bit positions
2810         SRL  A
2820         SRL  A    ; Then we output to high address port.
2830         LD   BC,ADDRHI ; BC points at it
2840         OUT  (C),A ; And do the OUT
2850         PUSH IY    ; Move IY contents into HL via stack
2860         POP  HL
2870         LD   A,L
2880         ADD  A, #60
2890         LD   L,A ; put result back
2900         JR   NC,LENT00 ; If no carry then jump
2910         INC  H
2920 LENT00: LD   A,32      ; now 32 bytes to load- A=loop
2930         LD   BC,DATAP ; Now BC points at data port (C)
2940 LENT01: IN   E,(C)     ; Get a byte
2950         LD   (HL),E    ; Store it
2960         DEC  A         ; See if all 32 done
2970         JR   Z,LENT02 ; Jump if yes
2980         INC  HL        ; Bump buffer pointer
```

```
2990           PUSH HL        ; Save buffer pointer
3000           LD   BC,ADDRLO ; BC points to low address reg
3010           IN   L,(C) ; Get the low address
3020           INC  L       ; Increment it
3030           OUT  (C),L ; re output it.
3040           JR   NZ,LENT03 ; If is non zero then jump
3050           LD   BC,ADDRHI ; Now BC points to high byte of addr
3060           IN   L,(C)     ; Fetch it
3070           INC  L         ; Increment it
3080           OUT  (C),L     ; Put it back
3090 LENT03:   LD   BC,DATAP ; BC points at data reg. for next loop
3100           POP  HL ; Restore buffer pointer
3110           JR   LENT01    ; round again
3120 LENT02:   POP  DE
3130           POP  HL
3140           POP  BC ; Reload
3150           RET  ; and go back out
3160 OUTPUT:   LD   A,(IX) ; Get a byte
3170           BIT  7,A     ; See if EOB marker
3180           RET  NZ      ; Return if so
3190           CALL #BB5A   ; Else output the chracter
3200           INC  IX      ; bump buffer pointer
3210           JR   OUTPUT ; and do it again
3220 PRINTR:   CALL #BD2E   ; See if printer is busy
3230           JR   NC,PRINT1 ; If not then jump
3240           CALL #BB1B   ; Otherwise loop with keyboard check
3250           RET  C       ; If any key was pressed then abort
3260           JR   PRINTR ; Else keep waiting
3270 PRINT1:   LD   A,(IX) ; Get a character
3280           BIT  7,A     ; See if EOB
3290           RET  NZ      ; return if so
3300           CALL #BD31   ; otherwise output the character to ptr
3310           INC  IX      ; Bump pointer
3320           JR   PRINTR ; then go round again
3330 HEXAS:    LD   H,A   ; Save original byte in H
3340           AND  #0F   ; AND out the low nibble
3350           OR   #30   ; Convert to ASCII
3360           CP   #3A   ; see if its alpha
3370           JP   M,HIGH ; If not then jump over adjustment
3380           ADD  A,#7   ; make it HEX
3390 HIGH:     LD   L,A    ; Put finished low digit away in L
3400           LD   A,H    ; Get out original byte
3410           SRL  A      ; Make High nibble low
3420           SRL  A
3430           SRL  A
3440           SRL  A
3450           OR   #30
3460           CP   #3A     ; See if Alpha
3470           JP   M,DONE ; Done if not
3480           ADD  A,#7    ; Adjust as before
3490 DONE:     LD   H,A     ; Save finished thing into H
3500           RET  ; and finish
3510 SPACE:    LD   (IX),#20 ; Place a space
3520           INC  IX         ; Bump pointer
3530           DJNZ SPACE      ; Loop till done
3540           RET  ; Then return
3550 XFER:     LD   A,(HL)
3560           BIT  7,A ; See if EOT market byte
3570           RET  NZ  ; Return if so
3580           LD   (IX),A ; Else place byte in o/p buff
3590           INC  IX      ; Increment O/P pointer
3600           INC  HL      ; And the txt buffr pointer
3610           JR   XFER    ; And round again
3620 COMPAR:   PUSH HL; Save regs
3630           PUSH BC
3640           PUSH IY
3650 CP001:    LD   A,(IY) ; Get a character
3660           CP   (HL) ; Compare it
3670           JR   NZ,CP002 ; If mismatched then exit
```

```
3680            INC   HL ; Otherwise bump pointers
3690            INC   IY
3700            DJNZ  CP001 ; Loop some more if more to do
3710 CP002:     POP   IY ; All done now so reload regs and exit
3720            POP   BC
3730            POP   HL
3740            RET
```

Fig. 3.90: RDLIB program listing.

```
                      1 ; **       ROMDISK software assembly        **
C000                100          ORG  #C000
                    110 *T+ ROMDISK.OBJ
                    120 ;
                    130 ; ** ROMdisk software
                    140 ;
FAE0                150 ADDRLO EQU  #FAE0
FAE1                160 ADDRHI EQU  #FAE1 ;
FAE2                170 ROMSEL EQU  #FAE2 ;
FAE3                180 CTRLP  EQU  #FAE3 ;
FAE8                190 DATAP  EQU  #FAE8 ;
0006                210 ROMNUM EQU  6
C000  01            220        DEFB #1 ; BACKGROUND ROM
C001  010100        230        DEFB #1,#1,#0 ; Mark 1 V1 rev 0
C004  15C0          240        DEFW NAMTAB ;pter to name tab
C006  C339C0        250        JP   RDINIT
C009  C30EC4        260        JP   RDOPEN
C00C  C380C4        270        JP   RDCLOS
C00F  C375C0        280        JP   RDDIR
C012  C3B6C1        290        JP   RDLOAD
C015  52442049      300 NAMTAB DEFM "RD INI" ; Init entry
C01B  D4            310        DEFB "T"+#80
C01C  52442E4F      320        DEFM "RD.OPE"
C022  CE            330        DEFB "N"+#80
C023  52442E43      340        DEFM "RD.CLOS"
C02A  C5            350        DEFB "E"+#80
C02B  52442E44      360        DEFM "RD.DI"
C030  D2            370        DEFB "R"+#80
C031  52442E4C      380        DEFM "RD.LOA"
C037  C4            390        DEFB "D"+#80
C038  00            400        DEFB 0 ; end name tab
C039  25            410 RDINIT DEC  H ;
C03A  01E3FA        420        LD   BC,CTRLP
C03D  3E80          430        LD   A,#80   ;
C03F  ED79          440        OUT  (C),A   ;
C041  0B            450        DEC  BC      ;
C042  3E00          460        LD   A,#0    ;
C044  ED79          470        OUT  (C),A   ;
C046  DD214FC0      480        LD   IX,INIMES ;
C04A  CD9FC3        490        CALL OUTPUT
C04D  1824          500        JR   INIDUN
C04F  0A0D          510 INIMES DEFB #0A,#0D
C051  2A2A2052      520        DEFM "** ROMdisk V1.2 initialised **"
C06F  0A0D0AFF      530        DEFB #0A,#0D,#0A,#FF
C073  37            540 INIDUN SCF  ; Set carry flag to show all OK.
C074  C9            550        RET  ; End of RDINIT
C075  F5            560 RDDIR  PUSH AF ; Make some room
C076  D5            570        PUSH DE
C077  C5            580        PUSH BC
C078  E5            590        PUSH HL
C079  DDE5          600        PUSH IX
C07B  F5            610        PUSH AF;
C07C  3E02          620        LD   A,2
C07E  CD0EBC        630        CALL #BC0E
C081  F1            640        POP  AF
C082  5F            650        LD   E,A
```

```
C083   01E2FA     660 RDDIR5 LD   BC,ROMSEL
C086   3E00       670        LD   A,#0
C088   ED79       680        OUT  (C),A
C08A   01E1FA     690        LD   BC,ADDRHI
C08D   DD2193C1   700        LD   IX,HEADER
C091   BB         710        CP   E
C092   C4ACC3     720        CALL NZ,PRINTR
C095   DD2193C1   730        LD   IX,HEADER
C099   CD9FC3     740        CALL OUTPUT
C09C   1601       750        LD   D,1
C09E   CD47C3     760 RDDIR1 CALL LENTRY
C0A1   3E21       770 FORM   LD   A,"!"
C0A3   FDBE60     780        CP   (IY+#60)
C0A6   C285C1     790        JP   NZ,RDDIR4
C0A9   FDE5       800        PUSH IY
C0AB   DDE1       810        POP  IX
C0AD   018000     820        LD   BC,#80
C0B0   DD09       830        ADD  IX,BC
C0B2   7A         840        LD   A,D
C0B3   CDC4C3     850        CALL HEXAS
C0B6   DD7400     860        LD   (IX),H
C0B9   DD7501     870        LD   (IX+1),L
C0BC   010200     880        LD   BC,2
C0BF   DD09       890        ADD  IX,BC
C0C1   0606       900        LD   B,6
C0C3   CDE5C3     910        CALL SPACE
C0C6   0610       950        LD   B,16
C0C8   DD7ED9     960 NAMET  LD   A,(IX-#27)
C0CB   DD7700     970        LD   (IX),A
C0CE   DD23       980        INC  IX
C0D0   10F6       990        DJNZ NAMET
C0D2   0607      1000        LD   B,7
C0D4   CDE5C3    1010        CALL SPACE
C0D7   FD7E75    1020        LD   A,(IY+#75)
C0DA   CDC4C3    1030        CALL HEXAS
C0DD   DD7400    1040        LD   (IX),H
C0E0   DD7501    1050        LD   (IX+1),L
C0E3   010200    1060        LD   BC,#2
C0E6   DD09      1070        ADD  IX,BC
C0E8   FD7E74    1080        LD   A,(IY+#74)
C0EB   CDC4C3    1090        CALL HEXAS
C0EE   DD7400    1100        LD   (IX),H
C0F1   DD7501    1110        LD   (IX+1),L
C0F4   010200    1120        LD   BC,#2
C0F7   DD09      1130        ADD  IX,BC
C0F9   CDE5C3    1150        CALL SPACE
                 1160 ; Output filetype as text
C0FC   FD7E73    1170        LD   A,(IY+#73)
C0FF   FE00      1180        CP   #0
C101   200E      1190        JR   NZ,BINCHK;
C103   210BC1    1200        LD   HL,BAS
C106   CDEEC3    1210        CALL XFER
C109   1853      1220        JR   RDDIR8
C10B   42415349  1230 BAS    DEFM "BASIC"
C110   80        1240        DEFB #80
C111   FE01      1250 BINCHK CP   #1
C113   200F      1260        JR   NZ,SDCHK
C115   211DC1    1270        LD   HL,BIN
C118   CDEEC3    1280        CALL XFER
C11B   1841      1290        JR   RDDIR8
C11D   42696E61  1300 BIN    DEFM "Binary"
C123   80        1310        DEFB #80
C124   FE02      1320 SDCHK  CP   #2
C126   2014      1330        JR   NZ,ASCCHK  ;
C128   2130C1    1340        LD   HL,SD
C12B   CDEEC3    1350        CALL XFER
C12E   182E      1360        JR   RDDIR8
C130   53637265  1370 SD     DEFM "Screen dump"
C13B   80        1380        DEFB #80
```

```
C13C  FE03      1390 ASCCHK CP   #3
C13E  200E      1400        JR   NZ,WHAT
C140  2148C1    1410        LD   HL,ASC
C143  CDEEC3    1420        CALL XFER
C146  1816      1430        JR   RDDIR8
C148  41534349  1440 ASC    DEFM "ASCII"
C14D  80        1450        DEFB #80
C14E  2156C1    1460 WHAT   LD   HL,UNKNOW
C151  CDEEC3    1470        CALL XFER
C154  1808      1480        JR   RDDIR8
C156  556E6B6E  1490 UNKNOW DEFM "Unknown"
C15D  80        1500        DEFB #80
C15E  DD36000D  1510 RDDIR8 LD   (IX),#0D
C162  DD36010A  1520        LD   (IX+1),#0A
C166  DD3602A0  1530        LD   (IX+2),#A0
C16A  FDE5      1540        PUSH IY
C16C  DDE1      1550        POP  IX
C16E  018000    1560        LD   BC,#80
C171  DD09      1570        ADD  IX,BC
C173  7B        1580        LD   A,E
C174  FE00      1590        CP   #0
C176  C4ACC3    1600        CALL NZ,PRINTR
C179  FDE5      1610        PUSH IY
C17B  DDE1      1620        POP  IX
C17D  018000    1630        LD   BC,#80
C180  DD09      1640        ADD  IX,BC
C182  CD9FC3    1650        CALL OUTPUT
C185  3E01      1660 RDDIR4 LD   A,1)
C187  82        1670        ADD  A,D
C188  57        1680        LD   D,A
C189  C29EC0    1690        JP   NZ,RDDIR1
C18C  DDE1      1700 RDDIR7 POP  IX
C18E  E1        1710        POP  HL
C18F  C1        1720        POP  BC
C190  D1        1730        POP  DE
C191  F1        1740        POP  AF
C192  C9        1750        RET
C193  524F4D20  1760 HEADER DEFM "ROM  Name       Length"
C1A9  20202020  1770        DEFM "     Type."
C1B3  0D0A      1780        DEFB #D,#A
C1B5  FF        1790        DEFB #FF
C1B6  FE02      1800 RDLOAD CP   #2
C1B8  2857      1810        JR   Z,LOAD01
C1BA  DD21C4C1  1820        LD   IX,ERR1
C1BE  CD9FC3    1830        CALL OUTPUT
C1C1  C33EC3    1840        JP   LDEXIT
C1C4  0A0A0D    1850 ERR1   DEFB #0A,#0A,#0D
C1C7  2A2A204E  1860        DEFM "** Need ROM no. + load addr **"
C1E5  0A0D      1870        DEFB #0A,#0D
C1E7  2A2A2045  1880        DEFM "** EG. |RD.LOAD,3,&4E20 loads ROM 3 **"
C20D  0A0A0DFF  1890        DEFB #0A,#0A,#0D,#FF
C211  E5        1900 LOAD01 PUSH HL
C212  D5        1910        PUSH DE
C213  DDE5      1920        PUSH IX
C215  DD7E03    1930        LD   A,(IX+3)
C218  FE00      1940        CP   #0
C21A  2007      1950        JR   NZ,MERR2
C21C  DD7E02    1960        LD   A,(IX+2)
C21F  FE00      1970        CP   #0
C221  2033      1980        JR   NZ,LOAD02
C223  DD2131C2  1990 MERR2  LD   IX,ERR2
C227  CD9FC3    2000 COMMOP CALL OUTPUT
C22A  DDE1      2010        POP  IX
C22C  D1        2020        POP  DE
C22D  E1        2030        POP  HL
C22E  C33EC3    2040        JP   LDEXIT
C231  0A0A0D    2050 ERR2   DEFB #0A,#0A,#0D
C234  2A2A2052  2060        DEFM "** ROM no. must be 1 to 255 **"
C252  0A0A0DFF  2070        DEFB #0A,#0A,#0D,#FF
```

```
C256  DD5602    2080 LOAD02 LD   D,(IX+2)
C259  CD47C3    2090        CALL LENTRYed
C25C  FD7E60    2100        LD   A,(IY+#60)
C25F  FE21      2110        CP   "!"
C261  282C      2120        JR   Z,LOAD03
C263  DD2169C2  2130        LD   IX,ERR3
C267  18BE      2140        JR   COMMOP
C269  0A0A0D    2150 ERR3   DEFB #0A,#0A,#0D
C26C  2A2A2052  2160        DEFM "** ROM slot is not populated **"
C28B  0A0A0DFF  2170        DEFB #0A,#0A,#0D,#FF
C28F  DD6601    2180 LOAD03 LD   H,(IX+1)
C292  DD6E00    2190        LD   L,(IX)
C295  D5        2200        PUSH DE
C296  FD5E74    2210        LD   E,(IY+#74)
C299  FD5675    2220        LD   D,(IY+#75)
C29C  19        2230        ADD  HL,DE
C29D  D1        2240        POP  DE
C29E  3032      2250        JR   NC,LOAD04
C2A0  DD21A7C2  2260        LD   IX,ERR4
C2A4  C327C2    2270        JP   COMMOP
C2A7  0A0A0D    2280 ERR4   DEFB #0A,#0A,#0D
C2AA  2A2A2041  2290        DEFM "** Address would overflow memory! **"
C2CE  0A0A0DFF  2300        DEFB #0A,#0A,#0D,#FF
C2D2  FD367600  2310 LOAD04 LD   (IY+#76),00
C2D6  FD367700  2320        LD   (IY+#77),00
C2DA  01E2FA    2330        LD   BC,ROMSEL
C2DD  ED51      2340        OUT  (C),D
C2DF  DD6601    2350        LD   H,(IX+1)
C2E2  DD6E00    2360        LD   L,(IX)
C2E5  01E0FA    2370 LOAD05 LD   BC,ADDRLO
C2E8  FD7E76    2380        LD   A,(IY+#76)
C2EB  ED79      2390        OUT  (C),A
C2ED  01E1FA    2400        LD   BC,ADDRHI
C2F0  FD7E77    2410        LD   A,(IY+#77)
C2F3  ED79      2420        OUT  (C),A
C2F5  01E8FA    2430        LD   BC,DATAP
C2F8  ED78      2440        IN   A,(C)
C2FA  77        2450        LD   (HL),A
C2FB  23        2460        INC  HL
C2FC  FD3476    2470        INC  (IY+#76)
C2FF  2003      2480        JR   NZ,LOAD06
C301  FD3477    2490        INC  (IY+#77)
C304  FD7E76    2500 LOAD06 LD   A,(IY+#76)
C307  FDBE74    2510        CP   (IY+#74)
C30A  20D9      2520        JR   NZ,LOAD05
C30C  FD7E77    2530        LD   A,(IY+#77)
C30F  FDBE75    2540        CP   (IY+#75)
C312  20D1      2550        JR   NZ,LOAD05
C314  DD211BC3  2560 LOAD07 LD   IX,DUNNIT
C318  C327C2    2570        JP   COMMOP
C31B  0A0D0A    2580 DUNNIT DEFB #0A,#0D,#0A
C31E  2A2A2052  2590        DEFM "** ROM loaded as requested **"
C33B  0A0DFF    2600        DEFB #0A,#0D,#FF
C33E  C9        2610 LDEXIT RET
C33F  FD7E77    2620        LD   A,(IY+#77)
C342  FDBE75    2630        CP   (IY+#75)
C345  209E      2640        JR   NZ,LOAD05
C347  C5        2650 LENTRY PUSH BC
C348  E5        2660        PUSH HL
C349  D5        2670        PUSH DE
C34A  01E2FA    2680        LD   BC,ROMSEL
C34D  3E00      2690        LD   A,0
C34F  C9        2700        RET
C350  7A        2710        LD   A,D
C351  CB0F      2720        RRC  A
C353  CB0F      2730        RRC  A
C355  CB0F      2740        RRC  A
C357  E6E0      2750        AND  #E0
C359  01E0FA    2760        LD   BC,ADDRLO
```

```
C35C  ED79      2770         OUT  (C),A
C35E  7A        2780         LD   A,D
C35F  E6F8      2790         AND  #F8
C361  CB3F      2800         SRL  A
C363  CB3F      2810         SRL  A
C365  CB3F      2820         SRL  A
C367  01E1FA    2830         LD   BC,ADDRHI
C36A  ED79      2840         OUT  (C),A
C36C  FDE5      2850         PUSH IY
C36E  E1        2860         POP  HL
C36F  7D        2870         LD   A,L
C370  C660      2880         ADD  A, #60
C372  6F        2890         LD   L,A
C373  3001      2900         JR   NC,LENT00
C375  24        2910         INC  H
C376  3E20      2920 LENT00  LD   A,32
C378  01E8FA    2930         LD   BC,DATAP
C37B  ED58      2940 LENT01  IN   E,(C)
C37D  73        2950         LD   (HL),E
C37E  3D        2960         DEC  A
C37F  281A      2970         JR   Z,LENT02
C381  23        2980         INC  HL
C382  E5        2990         PUSH HL
C383  01E0FA    3000         LD   BC,ADDRLO
C386  ED68      3010         IN   L,(C)
C388  2C        3020         INC  L
C389  ED69      3030         OUT  (C),L
C38B  2008      3040         JR   NZ,LENT03
C38D  01E1FA    3050         LD   BC,ADDRHI
C390  ED68      3060         IN   L,(C)
C392  2C        3070         INC  L
C393  ED69      3080         OUT  (C),L
C395  01E8FA    3090 LENT03  LD   BC,DATAP
C398  E1        3100         POP  HL
C399  18E0      3110         JR   LENT01
C39B  D1        3120 LENT02  POP  DE
C39C  E1        3130         POP  HL
C39D  C1        3140         POP  BC
C39E  C9        3150         RET
C39F  DD7E00    3160 OUTPUT  LD   A,(IX)
C3A2  CB7F      3170         BIT  7,A
C3A4  C0        3180         RET  NZ
C3A5  CD5ABB    3190         CALL #BB5A
C3A8  DD23      3200         INC  IX
C3AA  18F3      3210         JR   OUTPUT
C3AC  CD2EBD    3220 PRINTR  CALL #BD2E
C3AF  3006      3230         JR   NC,PRINT1
C3B1  CD1BBB    3240         CALL #BB1B
C3B4  D8        3250         RET  C
C3B5  18F5      3260         JR   PRINTR
C3B7  DD7E00    3270 PRINT1  LD   A,(IX)
C3BA  CB7F      3280         BIT  7,A
C3BC  C0        3290         RET  NZ
C3BD  CD31BD    3300         CALL #BD31
C3C0  DD23      3310         INC  IX
C3C2  18E8      3320         JR   PRINTR
C3C4  67        3330 HEXAS   LD   H,A
C3C5  E60F      3340         AND  #0F
C3C7  F630      3350         OR   #30
C3C9  FE3A      3360         CP   #3A
C3CB  FAD0C3    3370         JP   M,HIGH
C3CE  C607      3380         ADD  A,#7
C3D0  6F        3390 HIGH    LD   L,A
C3D1  7C        3400         LD   A,H
C3D2  CB3F      3410         SRL  A
C3D4  CB3F      3420         SRL  A
C3D6  CB3F      3430         SRL  A
C3D8  CB3F      3440         SRL  A
C3DA  F630      3450         OR   #30
```

```
C3DC   FE3A      3460         CP   #3A
C3DE   FAE3C3    3470         JP   M,DONE
C3E1   C607      3480         ADD  A,#7
C3E3   67        3490 DONE    LD   H,A
C3E4   C9        3500         RET
C3E5   DD360020  3510 SPACE   LD   (IX),#20
C3E9   DD23      3520         INC  IX
C3EB   10F8      3530         DJNZ SPACE
C3ED   C9        3540         RET
C3EE   7E        3550 XFER    LD   A,(HL)
C3EF   CB7F      3560         BIT  7,A
C3F1   C0        3570         RET  NZ
C3F2   DD7700    3580         LD   (IX),A
C3F5   DD23      3590         INC  IX
C3F7   23        3600         INC  HL
C3F8   18F4      3610         JR   XFER
C3FA   E5        3620 COMPAR  PUSH HL
C3FB   C5        3630         PUSH BC
C3FC   FDE5      3640         PUSH IY
C3FE   FD7E00    3650 CP001   LD   A,(IY)
C401   BE        3660         CP   (HL)
C402   2005      3670         JR   NZ,CP002
C404   23        3680         INC  HL
C405   FD23      3690         INC  IY
C407   10F5      3700         DJNZ CP001
C409   FDE1      3710 CP002   POP  IY
C40B   C1        3720         POP  BC
C40C   E1        3730         POP  HL
C40D   C9        3740         RET
C40E   F5        3750 RDOPEN  PUSH AF
C40F   E5        3760         PUSH HL
C410   3A77BC    3770         LD   A,(#BC77)
C413   FD7730    3780         LD   (IY+#30),A
C416   3A78BC    3790         LD   A,(#BC78)
C419   FD7731    3800         LD   (IY+#31),A
C41C   3A79BC    3810         LD   A,(#BC79)
C41F   FD7732    3820         LD   (IY+#32),A
C422   3A80BC    3830         LD   A,(#BC80)
C425   FD7733    3840         LD   (IY+#33),A
C428   3A81BC    3850         LD   A,(#BC81)
C42B   FD7734    3860         LD   (IY+#34),A
C42E   3A82BC    3870         LD   A,(#BC82)
C431   FD7735    3880         LD   (IY+#35),A
C434   3A7ABC    3890         LD   A,(#BC7A)
C437   FD773C    3900         LD   (IY+#3C),A
C43A   3A83BC    3910         LD   A,(#BC83)
C43D   FD773D    3920         LD   (IY+#3D),A
C440   3EC9      3930         LD   A,#C9
C442   327ABC    3940         LD   (#BC7A),A
C445   3283BC    3950         LD   (#BC83),A
C448   3EDF      3960         LD   A,#DF
C44A   3277BC    3970         LD   (#BC77),A
C44D   3280BC    3980         LD   (#BC80),A
C450   21B3C4    3990         LD   HL,OPENIN
C453   FD7536    4000         LD   (IY+#36),L
C456   FD7437    4010         LD   (IY+#37),H
C459   2606      4020         LD   H,ROMNUM
C45B   FD7438    4030         LD   (IY+#38),H
C45E   FD743B    4040         LD   (IY+#3B),H
C461   219BC5    4050         LD   HL,INCHAR
C464   FD7539    4060         LD   (IY+#39),L
C467   FD743A    4070         LD   (IY+#3A),H
C46A   FDE5      4080         PUSH IY
C46C   E1        4090         POP  HL
C46D   D5        4100         PUSH DE
C46E   113600    4110         LD   DE,#36
C471   19        4120         ADD  HL,DE
C472   2278BC    4130         LD   (#BC78),HL
C475   110300    4140         LD   DE,#3
```

```
C478  19        4150         ADD  HL,DE
C479  2281BC    4160         LD   (#BC81),HL
C47C  D1        4170         POP  DE
C47D  E1        4180         POP  HL
C47E  F1        4190         POP  AF
C47F  C9        4200         RET
C480  F5        4210 RDCLOS  PUSH AF
C481  FD7E30    4220         LD   A,(IY+#30)
C484  3277BC    4230         LD   (#BC77),A
C487  FD7E31    4240         LD   A,(IY+#31)
C48A  3278BC    4250         LD   (#BC78),A
C48D  FD7E32    4260         LD   A,(IY+#32)
C490  3279BC    4270         LD   (#BC79),A
C493  FD7E33    4280         LD   A,(IY+#33)
C496  3280BC    4290         LD   (#BC80),A
C499  FD7E34    4300         LD   A,(IY+#34)
C49C  3281BC    4310         LD   (#BC81),A
C49F  FD7E35    4320         LD   A,(IY+#35)
C4A2  3282BC    4330         LD   (#BC82),A
C4A5  FD7E3C    4340         LD   A,(IY+#3C)
C4A8  327ABC    4350         LD   (#BC7A),A
C4AB  FD7E3D    4360         LD   A,(IY+#3D)
C4AE  3283BC    4370         LD   (#BC83),A
C4B1  F1        4380         POP  AF
C4B2  C9        4390         RET
C4B3  78        4400 OPENIN  LD   A,B
C4B4  FE00      4410         CP   #0
C4B6  2028      4420         JR   NZ,OP001
C4B8  DD21BFC4  4430         LD   IX,ERR01
C4BC  C3F9C4    4440         JP   BADEND
C4BF  0A0D0A    4450 ERR01   DEFB #0A,#0D,#0A
C4C2  2A2A204D  4460         DEFM "** MUST specify filename **"
C4DD  0A0DFF    4470         DEFB #0A,#0D,#FF
C4E0  FD7354    4480 OP001   LD   (IY+#54),E
C4E3  FD7255    4490         LD   (IY+#55),D
C4E6  1601      4500         LD   D,1
C4E8  CD47C3    4510 OP002   CALL LENTRY
C4EB  FD7E60    4520         LD   A,(IY+#60)
C4EE  FE21      4530         CP   "!"
C4F0  2830      4540         JR   Z,OP005
C4F2  14        4550 OP004   INC  D
C4F3  20F3      4560         JR   NZ,OP002
C4F5  DD2103C5  4570         LD   IX,ERR02
C4F9  CD9FC3    4580 BADEND  CALL OUTPUT
C4FC  3E00      4590         LD   A,#0
C4FE  C600      4600         ADD  A,#0
C500  FE00      4610         CP   #0
C502  C9        4620         RET
C503  0A0A0D    4630 ERR02   DEFB #0A,#0A,#0D
C506  2A2A2046  4640         DEFM "** File not in ROMdisk **"
C51F  0A0DFF    4650         DEFB #0A,#0D,#FF
C522  FDE5      4660 OP005   PUSH IY
C524  D5        4670         PUSH DE
C525  116100    4680         LD   DE,#0061
C528  FD19      4690         ADD  IY,DE
C52A  D1        4700         POP  DE
C52B  CDFAC3    4730         CALL COMPAR
C52E  FDE1      4740         POP  IY
C530  20C0      4750         JR   NZ,OP004
C532  FD7E73    4760         LD   A,(IY+#73)
C535  FE00      4770         CP   #0
C537  2828      4780         JR   Z,OP003
C539  DD213FC5  4790         LD   IX,ERR03
C53D  18BA      4800         JR   BADEND
C53F  0A0D      4810 ERR03   DEFB #0A,#0D
C541  2A2A2066  4820         DEFM "** file is not ASCII BASIC **"
C55E  0A0DFF    4830         DEFB #0A,#0D,#FF
C561  01E2FA    4840 OP003   LD   BC,ROMSEL
C564  ED51      4850         OUT  (C),D
```

```
C566  FD5E54    4860         LD   E,(IY+#54)
C569  FD5655    4870         LD   D,(IY+#55)
C56C  FD7371    4880         LD   (IY+#71),E
C56F  FD7272    4890         LD   (IY+#72),D
C572  FD737A    4900         LD   (IY+#7A),E
C575  FD727B    4910         LD   (IY+#7B),D
C578  D5        4920         PUSH DE
C579  E1        4930         POP  HL
C57A  FD367600  4940         LD   (IY+#76),0
C57E  FD367700  4950         LD   (IY+#77),0
C582  CDDAC5    4960         CALL TWOKAY
C585  FDE5      4970         PUSH IY
C587  E1        4980         POP  HL
C588  116100    4990         LD   DE,#61
C58B  19        5000         ADD  HL,DE
C58C  117001    5010         LD   DE,#170
C58F  FD4E74    5020         LD   C,(IY+#74)
C592  FD4675    5030         LD   B,(IY+#75)
C595  3E16      5040         LD   A,#16
C597  FE00      5050         CP   #0
C599  37        5060         SCF
C59A  C9        5070         RET
C59B  E5        5080  INCHAR PUSH HL
C59C  FD6E7A    5090         LD   L,(IY+#7A)
C59F  FD667B    5100         LD   H,(IY+#7B)
C5A2  7E        5110         LD   A,(HL)
C5A3  23        5120         INC  HL
C5A4  FD757A    5130         LD   (IY+#7A),L
C5A7  FD747B    5140         LD   (IY+#7B),H
C5AA  FD3578    5150         DEC  (IY+#78)
C5AD  201C      5160         JR   NZ,INCH01
C5AF  FD3579    5170         DEC  (IY+#79)
C5B2  2017      5180         JR   NZ,INCH01
C5B4  FD6E71    5190         LD   L,(IY+#71)
C5B7  FD6672    5200         LD   H,(IY+#72)
C5BA  F5        5210         PUSH AF
C5BB  CDDAC5    5220         CALL TWOKAY
C5BE  FD7E71    5230         LD   A,(IY+#71)
C5C1  FD777A    5240         LD   (IY+#7A),A
C5C4  FD7E72    5250         LD   A,(IY+#72)
C5C7  FD777B    5260         LD   (IY+#7B),A
C5CA  F1        5270         POP  AF
C5CB  CB7F      5280  INCH01 BIT  7,A
C5CD  2806      5290         JR   Z,INCH02
C5CF  3E1A      5300         LD   A,#1A
C5D1  FE10      5310         CP   #10
C5D3  1803      5320         JR   INCH03
C5D5  37        5330  INCH02 SCF
C5D6  FEFF      5340         CP   #FF
C5D8  E1        5350  INCH03 POP  HL
C5D9  C9        5360         RET
C5DA  C5        5500  TWOKAY PUSH BC
C5DB  D5        5510         PUSH DE
C5DC  110000    5520         LD   DE,#0000
C5DF  01E0FA    5530  TWLOOP LD   BC,ADDRLO
C5E2  FD7E76    5540         LD   A,(IY+#76)
C5E5  ED79      5550         OUT  (C),A
C5E7  01E1FA    5560         LD   BC,ADDRHI
C5EA  FD7E77    5570         LD   A,(IY+#77)
C5ED  ED79      5580         OUT  (C),A
C5EF  01E8FA    5590         LD   BC,DATAP
C5F2  ED78      5600         IN   A,(C)
C5F4  77        5610         LD   (HL),A
C5F5  FD3476    5620         INC  (IY+#76)
C5F8  2003      5630         JR   NZ,TOOK1
C5FA  FD3477    5640         INC  (IY+#77)
C5FD  23        5650  TOOK1  INC  HL
C5FE  13        5660         INC  DE
C5FF  CB5A      5670         BIT  3,D
```

```
C601  28DC      5680       JR   Z,TWLOOP
C603  FD367800  5690       LD   (IY+#78),00
C607  FD367908  5700       LD   (IY+#79),#08
C60B  D1        5710       POP  DE
C60C  C1        5720       POP  BC
C60D  C9        5730       RET
```

```
Pass 2 errors: 00

ADDRHI FAE1  ADDRLO FAE0  ASC    C148  ASCCHK C13C
BADEND C4F9  BAS    C10B  BIN    C11D  BINCHK C111
COMMOP C227  COMPAR C3FA  CP001  C3FE  CP002  C409
CTRLP  FAE3  DATAP  FAE8  DONE   C3E3  DUNNIT C31B
ERR01  C4BF  ERR02  C503  ERR03  C53F  ERR1   C1C4
ERR2   C231  ERR3   C269  ERR4   C2A7  FORM   C0A1
HEADER C193  HEXAS  C3C4  HIGH   C3D0  INCH01 C5CB
INCH02 C5D5  INCH03 C5D8  INCHAR C59B  INIDUN C073
INIMES C04F  LDEXIT C33E  LENT00 C376  LENT01 C37B
LENT02 C39B  LENT03 C395  LENTRY C347  LOAD01 C211
LOAD02 C256  LOAD03 C28F  LOAD04 C2D2  LOAD05 C2E5
LOAD06 C304  LOAD07 C314  MERR2  C223  NAMET  C0C8
NAMTAB C015  OP001  C4E0  OP002  C4E8  OP003  C561
OP004  C4F2  OP005  C522  OPENIN C4B3  OUTPUT C39F
PRINT1 C3B7  PRINTR C3AC  RDCLOS C480  RDDIR  C075
RDDIR1 C09E  RDDIR4 C185  RDDIR5 C083  RDDIR7 C18C
RDDIR8 C15E  RDINIT C039  RDLOAD C1B6  RDOPEN C40E
ROMNUM 0006  ROMSEL FAE2  SD     C130  SDCHK  C124
SPACE  C3E5  TOOK1  C5FD  TWLOOP C5DF  TWOKAY C5DA
UNKNOW C156  WHAT   C14E  XFER   C3EE

Table used:   990  from  1109
```

Fig. 3.91: ROMDISK software assembly listing.

Conclusion

That concludes the description of the ROMdisk software. Because the ROMdisk uses some CPC firmware features it is fairly complex, but it gives you an extra facility for your CPC machine, which for many readers will probably prove very useful. You can load screen layouts very quickly, BASIC programs load very fast, and with extra software you can make your ROMdisk into a very useful data base for static information about your stamp, record, book, or picture collection with quick access to a potential four megabytes of data, it's up to you! The parts list for the ROMdisk is shown in Fig 3.92 and the power connections to the chips used are listed in Fig 3.93.

Capacitors

```
C1-C4    0.1 mfd   10 Volt disc types.           (RS 124-178)
C5       100 mfd   50 Volt radial                (RS 104-556)
```

Resistors

```
R1-R6    10K Ohms quarter Watt                   (RS 131-378) (6 required)
```

Semiconductors

```
E1       7425  Dual four i/p NOR gate + strobe (RS 304-122)
E2       74LS21 Dual four input NAND gates      (RS 307-553)
E3       74LS20 Dual four input  AND gates      (RS 305-210)
E4       74LS05 Hex o/c output inverters        (RS 309-054)
E5       8255A   PPI chip                       (RS 309-363)
E6       74LS138 3 to 8 line decoder            (RS 307-648)
E7       74LS245 Octal bus transceiver          (RS 308-348)
TR1      BC107 Transistor                       (RS 293-527)
```

Miscellaneous

```
Sockets for all chips
Expansion bus connector                         (RS 468-119)
CONN 1 ROMdisk expansion boards connector       (RS 470-465)
E8-E15: 28 pin DIL IC socket                    (RS 401-970) 8 required
Printed circuit board (See note two below)
```

NOTES:

1) Note that the RS part numbers quoted may be supplied in packs containing more than one part.

2) Complete kits of parts are to be available from Halstead designs LTD.

Fig. 3.92: ROMdisk parts list.

E Number	Device		description	Power connection pins			
	Type	Equivs	Function	+5V	+12V	−12V	OV
1	7425	–	Dual 4I/P NOR +51B	14	–	–	7
2	74LS21	–	Dual 4 I/P AND	14	–	–	7
3	74LS20	–	Dual 4 I/P NAND	14	–	–	7
4	74LS05	–	Hex inverter O/C	14	–	–	7
5	8255A	–	PPI Chip	26	–	–	7
6	74LS138	–	3 to 8 decoder	16	–	–	8
7	74LS245	–	Octal transceiver	20	–	–	10
8							

Fig. 3.93: Voltage connection pins to chips used on ROMdisk board.

The expansion ROMs board

Because Amstrad appreciated at a very early stage that even 64K memory sizes can become restrictive, the design of the CPC computers allows the user to disable the upper ROM (see chapter one) which contains BASIC. The way to do this is via the ROM select register. This write only eight bit register lives at I/O address Hex DF00 and is physically located in the video gate array. The register is loaded with zero to select the on board BASIC ROM. If it is loaded with any value other than zero then external ciruitry should use the ROMDIS signal on the expansion bus to disable the on board ROM when memory reads in the range Hex C000 to FFFF are made. So, in a nutshell, if the ROM select register contains anything other than zero then the upper 16K bytes of the memory map can be depopulated for reads. (Note that writing to addresses in this range always writes into the screen memory).

Having established that BASIC can be forced to vacate its address space, the next step is to add some extra hardware to enable us to place software of our own at these addresses. There is the added bonus that the lower ROM area (containing the CPC operating system firmware), contains software to allow intercommunication between the BASIC ROM and any other ROMs which we may switch in instead of it. Sounds complex doesn't it? In fact, although there are some complex considerations to apply when writing certain catagories of extra ROM software (namely foreground programs to replace BASIC) it is relatively easy to add routines to provide extra keyboard commands. Before we consider the different types of expansion ROM which can be added, let us look at the final hardware project in this book, the expansion ROM board.

Hardware details

Fig 3.94 shows the block diagram of the expansion ROM board. As the ROM select register in the gate array is not accessible externally we have to duplicate it on this board. The internal ROM select register is not affected in any way. The CPC address bus signals are beefed up by buffers becuase we are going to add a number of loads to them, which might degrade the quality of them unacceptably if we did not. The buffered address bus is then connected into the address inputs of all the eight expansion ROM sockets. Once again, to retain compatibility with the EPROM programmer and because they fit into the vacated ROM address space exactly, you should use 27128 EPROMs (or 2764 types if you can fit your extra software into 8K of the space), but unlike the EPROMs on the ROMdisk board the ones you fit to the expansion ROM board must be the faster 250 nanoseconds versions.

The address decoder is actually two decoders. The first of these produces a signal to enable the currently selected ROM when the Z80A reads from an address which is in the range Hex C000 to FFFF. The second produces a strobe signal to the ROM select register (ours, not the internal one!) when it detects a write into an I/O address beginning with Hex DF.

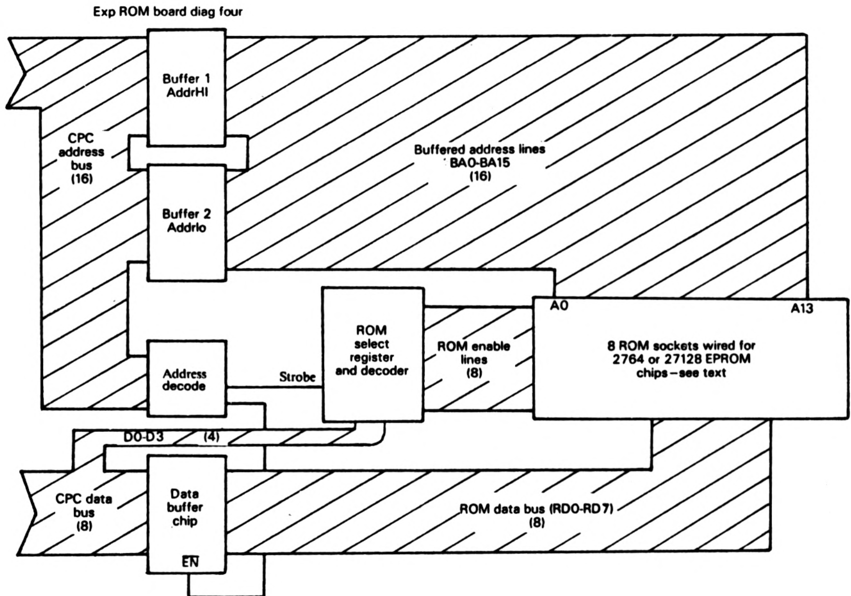


Fig. 3.94: Expansion ROM board – block diagram.

The data buffer chip is fitted for the same reasons as the address buffer chips.

The ROM select register and decoder is a single chip, into which the lower four bits of the address bus are latched when the address decoder strobes it. The chip used also houses a four to sixteen line decoder connected to its internal latch.

The expansion ROM board hardware description

Fig 3.95 shows the address decoder and ROM select register. Gates E1b and E2b form an address decoder which detects when a memory address in range of the upper ROM area is read from. It is conditioned by gate E7 whose output will be high if any other value than zero has been written into the ROM select register. The output from gate E2b when true is used for four purposes. Firstly to enable the data buffer chip (see Fig 3.96). Secondly to pull the expansion bus signal DOUT bar low. (See the description of the DOUT signal in appendix 5). Thirdly to take the expansion bus signal ROMDIS true, thus disabling the CPCs internal ROM when any of the boards on the expansion ROMs board are active.

Still on Figure 3.95, gates E2a and E1a decode the upper four address lines, and when these contain Hex D and the IORQ bar line is low at the same time as the WR bar signal goes low, a strobe signal is pulsed to E3 which is a latch/4 to sixteen line decoder chip. Decoding only the upper four address lines is

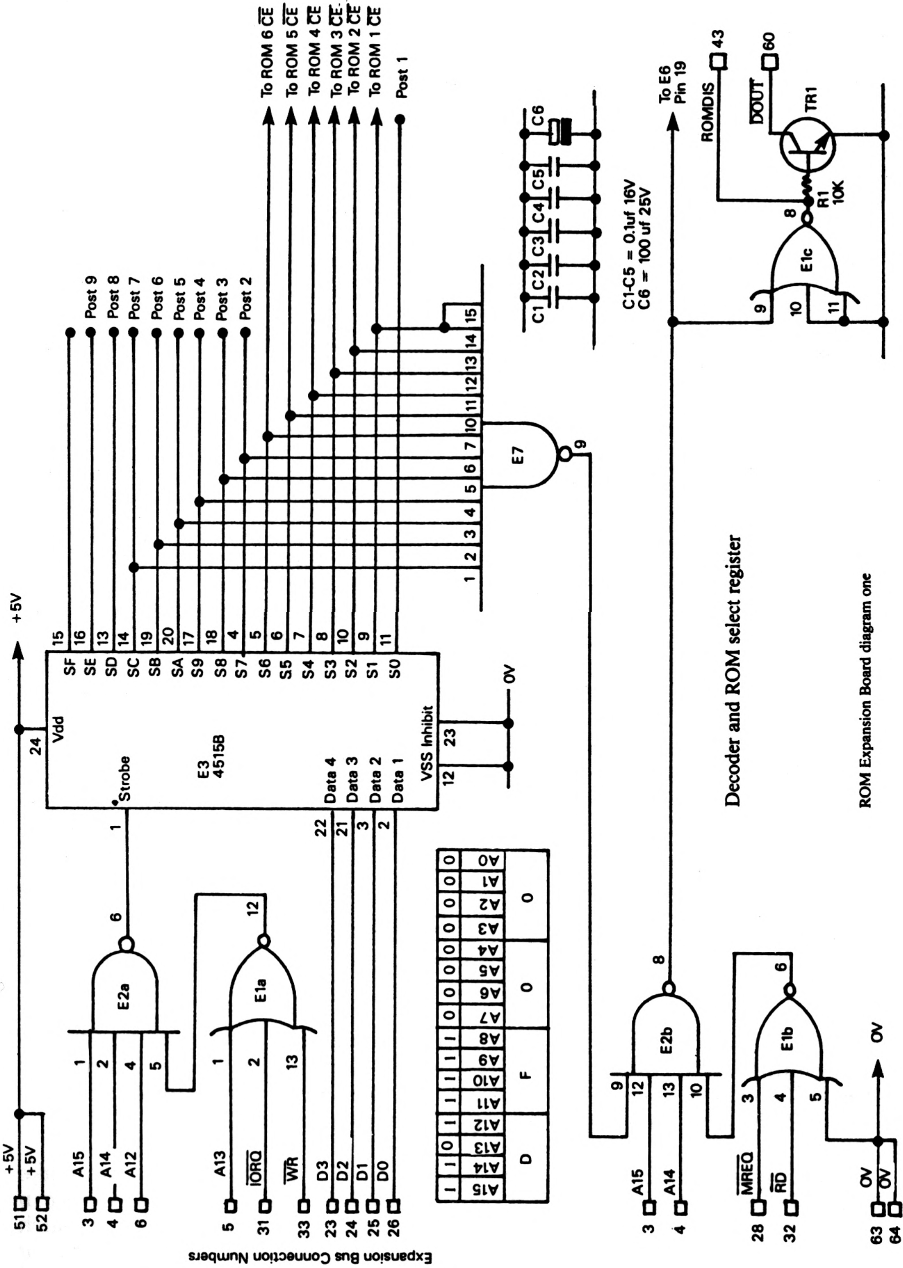


Fig. 3.95: Expansion ROM board address decoder circuit diagram.

permissable because the ROM select register is the only thing whose address in the I/O page begins with D. E3 latches the lower four bits of the CPC data bus when its STROBE input is pulsed low. The S output corresponding to the digit held in the latch of E3 will then go low. The S outputs are connected to the CE bar (Chip enable) inputs of the EPROMs plugged into the expansion ROM sockets. Many of the S outputs from E3 are terminated at solder posts so that you can choose which address you wish to place a ROM at. As we have already seen the BASIC ROM is usually ROM zero (though you can actually use ROM zero, the CPC firmware will always enable your ROM and never BASIC, so if you use ROM zero you may not be able to get into BASIC). ROM position seven is used for the disk firmware ROM, so ROM sockets zero and seven have their CE bar inputs connected to a solder post so that you can solder a jumper wire to one of the rom select decoder outputs. ROM positions one to six are directly connected to the decoder.

Capacitors C1 to C6 decouple and smooth the +5 volt line in the usual way.

Figure 3.96 shows the address buffer chips and the data bus buffer chip. The address buffer chips E4 and E5 are 74LS245 types which take the signals from the CPC expansion bus and pass them through to the address input pins of the ROM sockets. Chip E6 – another 74LS245 – takes the data signals from the ROM sockets data pins and passes them through to the CPC address bus when enabled to do so by the address decoder, as described above These buffer chips all present a very low load to the Z80A, which drives the address and data lines from the CPC, and they preclude the possibility of loading the address and data busses to the point where the logic levels become marginal.

The last diagram is Fig 3.97 which shows how a typical ROM socket is wired. All ROMs have their pins wired in parallel, the only exception being the CE bar input, each one of these is wired to a different S output from the ROM select decoder.

Expansion ROM usage

Having an expansion ROM board fitted to your CPC opens up many possibilities. These will become apparent as we look at each of the types of ROM which the system will recognise.

The most difficult of the types of ROM to do is a foreground ROM. A foreground ROM contains a program which can assume complete control of the CPC, making use of CPC operating system firmware routines from the lower ROM. The on board BASIC ROM is a foreground ROM.

More simple and very useful, is a background ROM containing software like the ROMdisk detailed in the last project description. Background ROMs can be made to provide extra facilities and commands to invoke them, as such they give the capability to extend BASIC, to add control commands for external hardware, and so on. Background ROMs must be located at a ROM select address between one and seven.

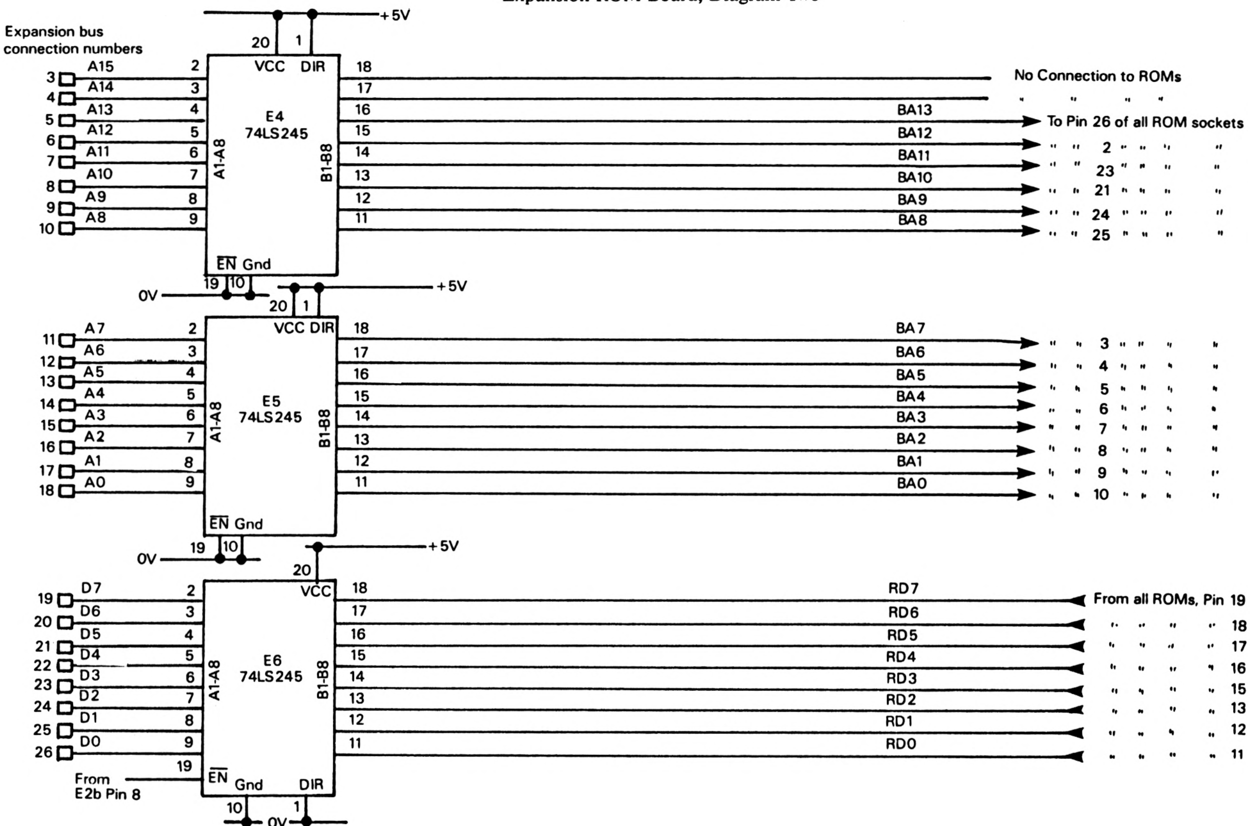


Fig. 3.96: Address and data buffer chips.

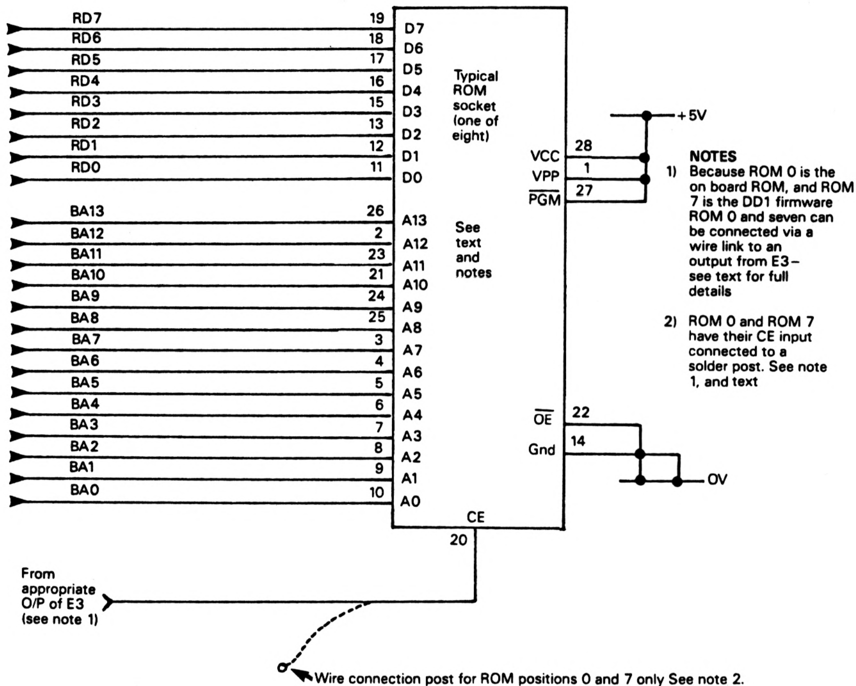


Fig. 3.97: Typical ROM socket wiring on expansion ROM board.

The final type of ROM recognised by the CPC firmware is an extension ROM. These contain programs or data for loading down into RAM for access or running.

There is a very strict layout for the header table of any expansion ROM. This is detailed in SOFT 158 – the firmware manual.

Conclusion

The expansion ROM board will allow you to add permanant software to your CPC, and will also allow you to develop your own software for new hardware which you may wish to construct. Fig 3.98 is the parts list for the expansion ROM board project, fig 3.99 is the power pins list. No problems should be met when building the project, the only possible stumbling block is in selecting which ROM select addresses to use. If you have add ons other than the ones in this book, check the documentation you got with them to see if the have an expansion ROM inside them, if they do then avoid using that ROM select address for any other ROMs which you use on the expansion ROM board. The only two projects which use an expansion ROM in this book are the ROMdisk, and the software for the network project – see chapter six.

Capacitors

```
C1-C5        0.1 mfd    10 Volt disc type.       (Eg RS 124-178)
C6           100 mfd    50 Volt radial           (RS 104-556)
```

Resistor

```
R1           10K Ohm    quarter watt             (RS 131-378)
```

Semiconductors

```
E1       TTL 74LS27 Triple 3 i/p NOR gates        (RS 309-060)
E2       TTL 74LS20 Dual four input NAND gate     (RS 305-210)
E3       CMOS 4515B 4 to 16 line latch/decoder    (RS 309-789)
E4 - E6  TTL 74LS245 Octal bus transceiver.       (RS 308-348) (3 required)
E7       TTL 74LS133 13 i/p NAND gate             (RS 305-030)
TR1      BC107 Transistor                         (RS 293-527)
```

Miscellaneous

```
Sockets for all chips
Expansion bus connector.                          (RS 468-119)
Printed circuit board (see note 2 below)
Solder posts (10 off)                             (RS 434-677)
```

NOTES:

1) Note that the RS part numbers quoted may be supplied in packs containing more than one part.

2) Complete kits of parts are to be available from Halstead designs LTD.

Fig. 3.98: Expansion ROM board parts list.

E Number	Device		description	Power connection pins			
	Type	Equivs	Function	+5V	+12V	−12V	OV
1	74LS27	–	Triple 3I/P NOR	14	–	–	7
2	74LS20	–	Dual 4 I/P NAND	14	–	–	7
3	4515b	–	4 to 6 line latch	24	–	–	12
4	74LS245	–	Octal Transceiver	20	–	–	10
5	74LS245	–	Octal Transceiver	20	–	–	10
6	74LS245	–	Octal Transceiver	20	–	–	10
7							
8							

Fig. 3.99: Power connection pins for chips used on expansion ROM board.

Chapter three conclusion

In this chapter we have looked at ten hardware extension projects which you can add to your CPC to enhance its capabilities, and expand the range of uses to which it can be put. I hope that you will build and enjoy using at least a couple of them. If you get foxed by any of the terms or concepts used in the project descriptions first consult appendix six, if that fails to answer the problem try one of the books listed in the appendix two, and if all else fails your local library may well have something on the shelves of their computer section which will contain the answer. The prime source of Amstrad specific information is of course Amsoft publications, which cover nearly all aspects of the machines in great detail. I cannot undertake to enter into correspondence about the projects, save in exceptional circumstances.

See the beginning of this book for details about kits of parts for the projects and also for how you can buy a complete set of the software on disk which will save you a good deal of tedious typing!

CHAPTER FOUR

Using Amstrad Firmware

Hardware control features in Locomotive BASIC

When you use BASIC on your Amstrad you find many features which are not available on lesser machines. Some manufacturers actively discourage you from attemping any control over the hardware of their computers. Not so with Amstrad. The BASIC features which are applicable to direct hardware control or which provide information about the hardware will be listed and some usage examples will be given for the less obvious ones. The examples are given in alphabetical order:

`BORDER`
This command tells the video gate array (see chapter 1) what colour to set the pixels located at the border of the screen to.

`CAT`
Tells the cassette or disk software to read the tape and produce a catalogue to the screen.

`CLS/CLG`
These two commands cause all the screen memory to be filled with the current background (or `PAPER`) colour

`DI`
This is not a hardware control statement, but is included here to point that out! It should not be confused with the Z80A instruction of the same name. If hardware interrupts were truly disabled then the keyboard would be inoperative, since this is scanned during the interrupt routine which always runs in the background of what the machine actually appears to be doing.

`EOF`
Very useful for file handling. By testing the EOF variable you can tell if the firmware has got any more data from the file which has been opened via an `OPENIN` statement.
EXAMPLE:

```
IF EOF=0 THEN PRINT "NO MORE DATA" ELSE PRINT "MORE DATA TO COME"
```

`FRE`
Shows how much of the memory is free. The construct `FRE("")` forces a garbage collection (that is, a tidy up of memory) by BASIC.

HIMEM
A variable which shows the highest memory address which BASIC will use, thus leaving the rest free for other usages like loading machine code programs into.

INP (Number)
Reads the input port (number).
EXAMPLE:
```
IF (INP(&F500) AND &40)=0 THEN PRINT "PRINTER READY"
```

MEMORY
Sets the HIMEM parameter to any required value. EXAMPLE: `HIMEM 5000`

MODE x
Sets the 6845 VDU controller chip display mode.
EXAMPLE: `MODE 1`
OPENIN/OPENOUT
`OPENIN` reads a buffer full of information from the disk or tape. `OPENOUT` creates a buffer for data destined to be written to tape or disk.

OUT
Provides a way to `POKE` values to I/O addresses, (opposite of INP.)

PEEK/POKE
Allows you to read a byte of memory directly or to write a value into a byte of memory.
EXAMPLE:
```
POKE 40000,6: PRINT "ADDRESS 40000 CONTAINS ";PEEK(40000)
```

POS
This very useful function tells you what column the firmware thinks that the stream in brackets cursor or print head is located. If this is different to the real position then software can take corrective action, see later in this chapter.

SAVE
Causes BASIC to call the firmware routines to write some data to disk or cassette.

VPOS
Similar to `POS` except that this is the vertical position within the screen or printed page where the firmware thinks that the indicated streams cursor or print head is located.

WIDTH
A command to tell the firmware how many columns of print the printer connected to the printer port on stream 8 can print, this will usually be either 80 or 132. (see below)

These then are the BASIC keywords which most affect the hardware of the

Amstrad machines. Now let us look at some points about how the BASIC actually uses the hardware, and how the interaction between hardware and software can cause some problems.

Interactions between BASIC and hardware

Let us begin with a major cause of confusion – the printer. If you type the direct command:

```
PRINT POS(#8)
```

the column number at which the firmware currently thinks the print head on your printer is located will be shown. If this is out of step with the real position, then the firmware will keep inserting line feed and carriage returns into your printout at seemingly odd times. It is probably best to type:

```
WIDTH xx: PRINT #8
```

Where xx is the width in columns of the printer, before you try to print anything at all. This ensures that both software and printer agree. If you try to print something without issuing a `WIDTH` command after just powering the computer on, then the print may be wrongly aligned – even worse, if you are using an 80 column printer the printout may be printed off the paper.

Interrupts are our friends!

As was previously stated in another context earlier in this chapter, the Z80A in the CPC is receiving, acknowledging and actioning a continuous stream of interrupt requests. From BASIC you will be unaware of this in the normal way of things. Think about the keyboard hardware which was covered in chapter one, is the circuitry associated with it in any sense intelligent? No, it doesn't even scan the keys itself, it depends on the continual update of the scan address by the microprocessor. So what is it which initiates this scan update? Another function is the timekeeping, which is accessed via the TIME variable from the BASIC interpreter, what keeps this ticking away when there is no clock chip?

The initiator of all these functions is the interrupt generator. This originates from the video gate array and happens roughly every 1/300th of a second. The interrupt routine which services this interrupt – which is called the time interrupt – can detect whether there is a user attached device which has generated the interrupt, or if it was indeed the internally generated one. During the interrupt routine certain actions occur. These are to do with the system timers, accessed using the AFTER and EVERY keywords in BASIC. The 1/300th second interrupt routine sees if it is time for a time clock update, or time to carry out other feature implementations on the machine. These features include ink flashing, sound generation checks, and keyboard scanning.

So the Z80A is running the interrupt routines in the background all the time the machine is switched on, except for one special case. This special case is the cassette firmware. Becuase all the cassette data recovery timing is done in software (to keep the amount of actual cassette hardware down to a minimum) no interrupts can be allowed whilst the actual data encoding (writing) or decoding (reading) of data is taking place.

Have you ever tried pressing the ESC key during the time that a tape or disk is actually being read? If you have not, try it now. You will find that there is no response, the usual BREAK does not occur. This is because the keyboard is not being scanned, interrupts are disabled. Another good experiment you can try is typing ahead on the keyboard, that is, try typing anything whilst I/O is happening. You will find that much of what you have typed is not remembered by the machine.

Another implication of not having interrupts enabled during cassette I/O is that the clock function is stopped. As stated in the user manual for the CPC, this means that BASIC programs which use the TIME variable as well as using files for data cannot expect the TIME function to be an accurate indication of elapsed time since power on of the machine.

Conclusion

The BASIC supplied with the CPC machines is very good indeed at allowing you to access the hardware, memory, I/O space etc. It is also very good at hiding itself. You cannot PEEK the BASIC ROM, since all references to the area of memory where it resides are redirected to the RAM which occupies the same address space. This means that the BASIC ROM cannot be listed from BASIC.

The next chapter forms an introduction to the concepts and techniques used in a language which allows you, as a programmer, to get a firm grip of both the inbuilt hardware, and of your own add–on units.

CHAPTER 5

Developing Assembler Language Programs

Programming in BASIC makes you very lazy. BASIC is so forgiving and informative, that simple programs can be written at the keyboard, with little or no preparatory work. It is nearly always true that programs written in this way are a good deal less efficient than a carefully flowcharted and coded BASIC program. The point is that if you get it wrong, BASIC will tell you so with an intelligent error message.

However, BASIC has a number of drawbacks, as well as the deplorable encouragement of slothfulness. Perhaps the greatest is its comparative speed to other languages, notably machine code. A machine code program to carry out a specific task will run up to 25 times faster than a BASIC program to do the same thing. Many BASIC versions (though not the Locomotive BASIC in the CPC) limit the amount of direct control which programs can exercise over the hardware they run on. And finally BASIC programs are inherently inefficient. For these reasons, after becoming proficient at a language like BASIC, many programmers actually go into the far more tedious, and less secure process of writing programs in assembler. They find that the attention to detail they must exercise, along with the extensive pre planning, and the more efficient end result, give them far greater satisfaction, once the program is completed.

You already know that the Z80A microprocessor which is the brain of your CPC, does not directly understand statements like PRINT, or READ or GOTO. It does understand numbers, and interprets certain numbers as instructions. You can write these numbers down in any format you please, be it hexadecimal, octal, decimal, or even binary. The Z80A only needs to see the correct code, so long as it can recognise the binary patterns which form its instructions, it is happy.

In chapter two we looked through the Z80A instruction set, and in this chapter we shall be seeing how you can put those instructions to work. Ideally you should have a assembler package like DEVPAC (available from Amsoft). If you ONLY want to copy the programs presented in this book however, a way to use BASIC statements like DATA and POKE to create machine executable programs in memory will be given.

What is an assembler program?

If you owned one of the first ever affordable home computers, like the Science of Cambridge MK14 or the original Acorn machine (circa 1977–1978), you had little choice but to program it in pure machine code. This meant that you got a booklet of instructions, which included a list of the operation codes (opcodes) which the microprocessor understood. The input method was a small keyboard with the Hex digits and some command keys on it. The display was an eight digit calculator style LED array. You entered programs by pressing a sequence of buttons like: "MEM" "F" "E" "F" "0". Which would select memory address Hex FEF0. Then you would press "DEP" "3" "4" to deposit the value 34 into the memory location. There was no BASIC, and no QWERTY keyboard. Programming in this way is very time consuming and boring. Everything had to be worked out by you, opcodes, offset values for jumps, the lot. Furthermore when insertions or deletions to the programs were needed, all the offsets within the program had to recalculated manually!

Because they have QWERTY keyboards and textual displays, later generations of home computers, like the CPC machines, have made it possible to use assembler programs. These will accept a body of text from the keyboard which represent machine code instructions. The powerful bit comes when you issue an assemble command. When that happens the assembler swings into action and produces a machine code program which the microprocessor can execute. During the assembly error messages tell you if you have made any syntactical mistakes, though not generally mistakes of logic, for example a loop from which there is no exit. Ignore error messages given during the assembly at your peril! Remember that when the assembly is completed, and you come to run the machine code program it has created, you can't just press ESC to get out of it. The program must execute properly, or the machine will probably crash. If it does crash you will have to power off and reload everything– assembler and all, from tape or disk. Assemblers also have editing facilities to allow changes to the textual version of the program you are writing to be made. These editing facilities are a little like a specialised form of word processor. DEVPAC also has a debug program, which will be covered later.

Let us try to get all the various terms into their places: the ASSEMBLER takes an ASSEMBLY LANGUAGE PROGRAM and produces a series of MACHINE CODEs which will implement it,(another name which you may hear is OBJECT CODE. This is another name for the machine code program – because producing it is the object of the assembler program.). The printout produced during the actual assembly process is called an ASSEMBLY LISTING.

Non machine-code elements of assembly language

As well as machine code mnemonics, you will also find other things in assembler listings. Most obvious of these are the remarks. These are always preceeded by a semicolon character. Another feature familiar from BASIC is that each line of assembler has a line number. Let us look at an example of a line which loads the "A" register with the contents of the "D" register:

```
100      LD A,D ;  Load A register from D register.
```

Line no. Mnemonic Remarks

With the DEVPAC assembler line numbers can be from 1 to 32767. Mnemonics are any legal Z80A instruction mnemonic. The remarks section is for brief comments on what the line does.

We have already seen that line numbers are used, but they do not serve as locations the way they do in a BASIC program. You cannot execute a `GOTO 100` type of Z80A instruction to get to line 100. Line numbers are primarily used during assembly, or during editing to identify which line has an error in it, or which line you wish to edit. If you want to be able to execute, or reference a point in the program more than once then you have to give that line a label. A label is a name of up to six characters followed by a colon. For example, we could label the line we have just used as "COPYDA":

```
100 COPYDA: LD A,D ; Copy D register to the A register.
```

Label

This would allow us use a "Jump Relative" statement later in the program to come back to this point, by having a line which reads:

```
900 JR,COPYDA ; Jump to point where A is loaded from D
```

Another group of elements in a listing which are not machine code mnemonics, are directives. A directive is a command which is included in the listing to tell the assembler program to carry out certain actions during the actual assembly. These are things like telling it that all values are to be printed out in Hex rather than decimal on the assembly listing, or to reserve some bytes of memory for storing values required by your program. We will encounter many of these directives as we go through the program listings later in this chapter, and they will be explained as they crop up.

The final type of element in a listing which is not a machine code, is a blank line! These can be used to good effect in improving the legibility of programs, by separating the different routines, therefore making it easier to locate specific bits of the program.

So that you can experiment for yourself, the assembler programs which are used for examples in this chapter, are mainly drawn from the drivers and machine code subroutines which are listed as Hex dumps or DATA statements in chapter three. Having assembly listings for these will allow you to understand, adjust, or rewrite them, should you wish to.

Getting going

If you are already successfully using DEVPAC on a CPC 464, or if you have a disk version of DEVPAC then please skip this section.

It is always a good idea to have both the assembler and the debug programs, GENA3 and MONA3 respectively, loaded simultaneously. This allows you to test your program as you develop it. Most of the information required to use the two programs is contained in the manual which comes with them, I, and others, found the instructions for loading both together rather sketchy, so let us just run through the sequence to do this. First load side two of the DEVPAC cassette. Then type: *RUN ""* and press the PLAY button on the datacorder. After about 20 seconds the MONALOADER program will run and ask you LOAD ADDRESS? If you want to load both modules of DEVPAC then answer 30000. Then MONA3 will be loaded, and when this has completed it will run. The first thing it does is print up its front panel display. Now type CTRL/X (CTRL and X keys simultaneously) to get out to BASIC, you may have to press the ENTER key a couple of times to get to the Ready prompt. Now turn over the cassette to side one, rewind and press PLAY. Once again type: RUN "". Then GENALOADER will be loaded, and when it runs it asks you *LOAD ADDRESS?* You should answer with 1000. Then GENA3 will be loaded in, printing up its menu of commands when it is ready to go. Type B and ENTER to exit to BASIC, again a couple of presses of ENTER may be required to get to Ready. Now type CALL 1004. This gets you into GENA3 again, but exit again with the B command. This time you go straight back to BASIC. Finally, type *CALL 30002.* This gets you into MONA3. From now on if you are running MONA3 and want to go into GENA3 you just type CTRL/J, if you are in GENA3 and want to go into MONA3 just type J and ENTER. In this way there is no need to keep going via BASIC.

When using GENA3 you have the option of specifying what address the machine code you generate during an assembly is written into. This is via the *ORG* (Organise) directive. GENA3 will issue the error message *"BAD ORG!"* if obeying the ORG would mean overwriting either the textual representation of the program you are writing (called the textfile), or the GENA3 program itself. Beware of specifying an ORG which would overwrite the memory area where you have loaded MONA3, GENA3 will allow you to do it!

It is not my intention to duplicate the contents of Hisoft's somewhat slim manual on DEVPAC (some more examples would have been nice Hisoft!) so we will now push on with looking at some practical programs, which will be explained step by step.

Writing programs in assembly language

At this point a hallowed publication called SOFT 158 should be brought to your attention. SOFT 158 is a publication available from Amsoft, which gives very comprehensive details about the firmware of the CPC. This, and a Z80 book —of which several are detailed in appendix 2— should be available to you if you intend to do a lot of assembler programming on your CPC. SOFT 158 gives not only details about the I/O addresses allocation, details about how to use scores of subroutines within the lower ROM firmware, but also gives comprehensive details about the I/O. SOFT 158 is expensive, but as a low level CPC programmer you will need it, and it is worth the money.

When you set out to write even a simple assembly language program, it is very important to begin with a clear idea of what you want it to do, and a fairly good idea of how you are going to do it. This involves writing a flowchart. A flowchart is a representation on paper of the actions and decisions which occur in the program, starting with the initialisation stages, and proceeding through all possible paths to the end of the program. The major symbols for flowcharts which are of use when flowcharting computer programs are shown in figure 5.1. Inside each symbol are examples of the kind of thing which you might typically find in a real chart. Fig 5.2 shows a flowchart for the decision process which a reader might go through in order to read this chapter.

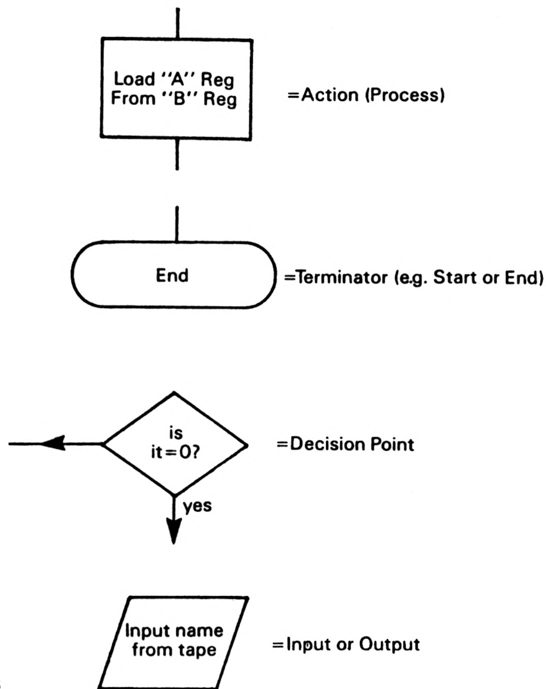
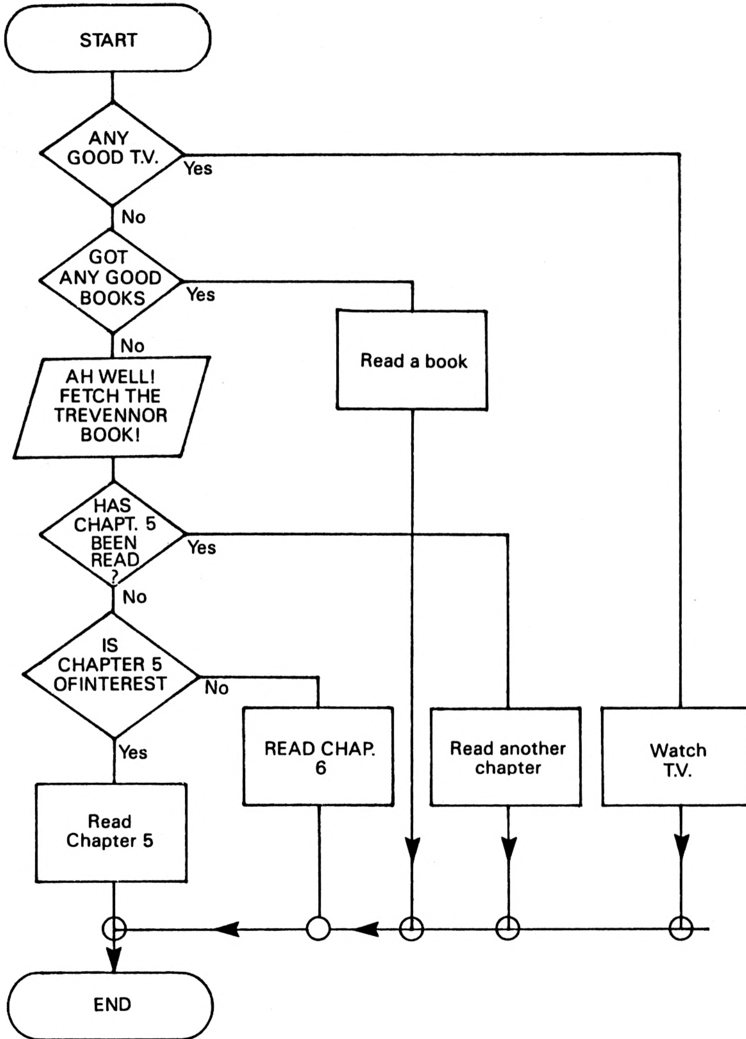


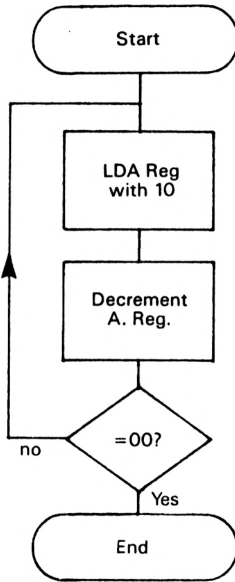
Fig. 5.1 Flowchart Symbols

Fig. 5.2 Reader Decision Flowchart



When you draw flowcharts errors of logic in your approach to the program implementation will be easier to find. Consider the flowchart shown in fig. 5.3. At a glance you can see that the "A" register will never reach zero, because it is always being reloaded with ten. This is a simple example, but when writing more complex programs you will detect and prevent bugs far more easily if you flowchart the whole thing before going anywhere near the keyboard.

Fig. 5.3 A Typical Error–Shown up in a Flowchart



Software for hardware project five

In project five of chapter three we looked at how to use an eight bit input port as a sense register. Let us now develop the software side of that project. We connect eight switches to the port, one to each bit. You could mount the switches in a box on a long flying lead at some distance from the CPC 464. The switches should be snap action, not leaf switches. In the example application of counting how many different species of bird have been spotted, this would allow the user to run a cable from a CPC 464 to a hide. The software we are about to look at can very easily be customised to any spotting application, be it cars, trains, traffic studies, or whatever.

The primary function of the software is to count the number of presses on each switch, and display the counters on screen. We repeatedly scan the port and count how many times each switch is pressed. The counters are all to be capable of counting from 0 to 9999. In the finished version some other features have been provided.

If SHIFT and 3 key (that is, the # symbol) are pressed, then the program ceases scanning the switches, and writes the current counter values into a tape or disk file. The layout of the items within this data file will be detailed later, but its purpose is to allow you to write a program in BASIC to produce

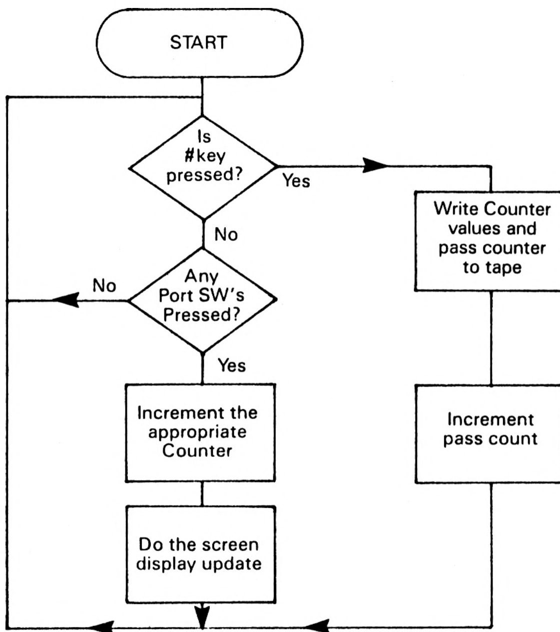
reports of averages, totals over a given period etc. You can exit the program by pressing CTRL and the upward arrow above the COPY key, simultaneously.

Now having established these requirements we can specify that we will need the following program modules:

1) An initialisation section.
2) The switch scanner section.
3) The counter update section.
4) The display update section.
5) The I/O section.

After considering what each of these sections will have to do, we can then go forward to write detailed flowcharts, and then get down to writing the actual program listing. The low detail flowchart is shown in fig 5.4

Figure 5.4 Overall flowchart for data logger program



The initialisation section will have to: initialise PPI #2 to set port B to an input port. There will be a pass counter to count how many samples have been taken, so at the very start the pass counter must be set to zero. The next job is

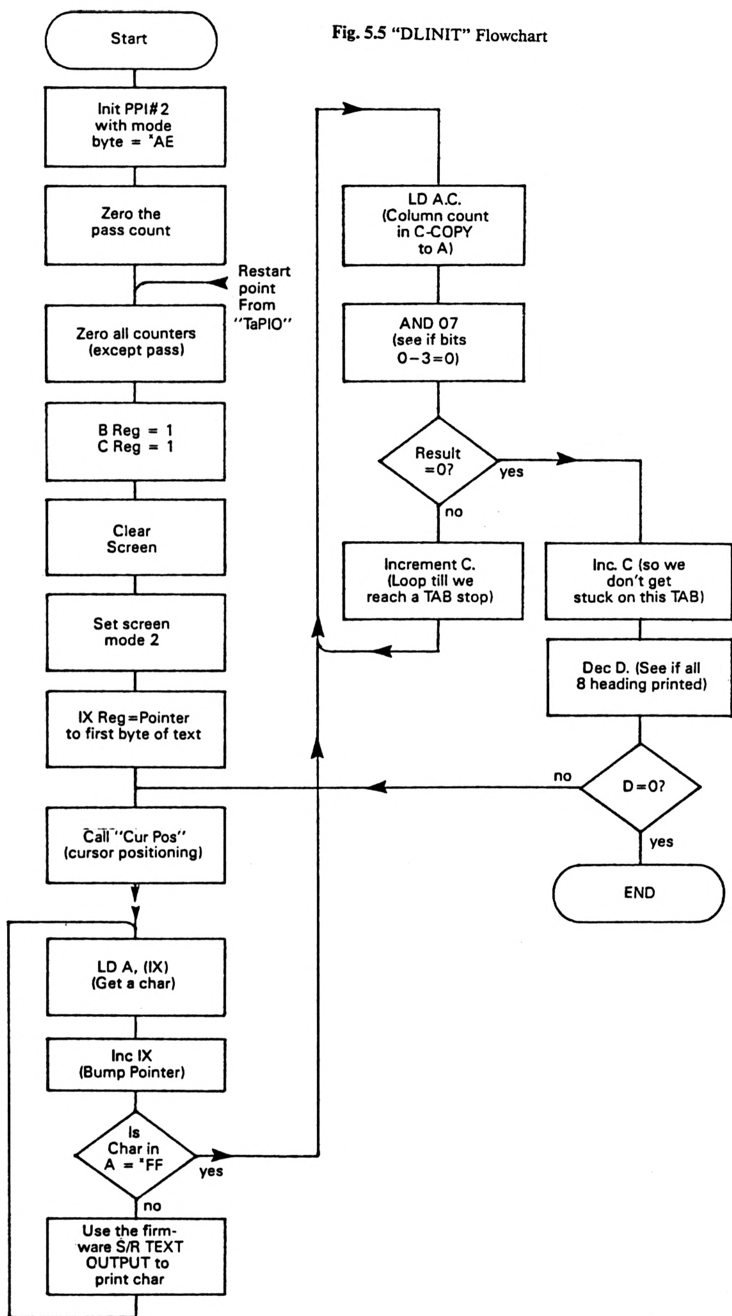


Fig. 5.5 "DLINIT" Flowchart

Fig. 5.6 "SWSCAN" Flowchart

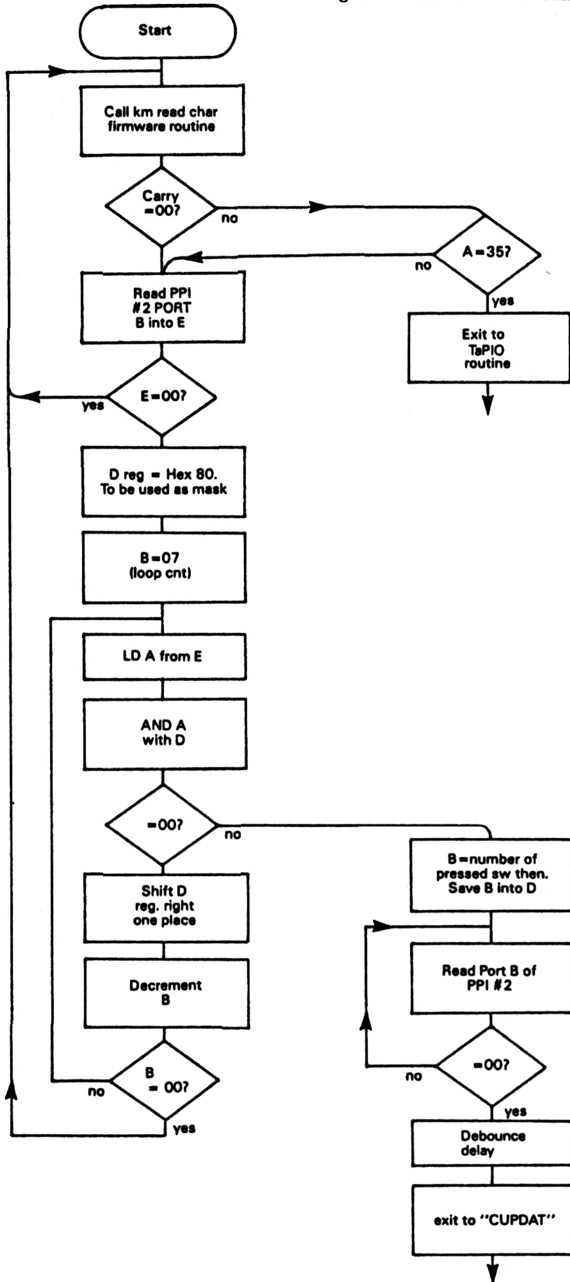


Fig. 5.7 "CUPDAT" Flowchart

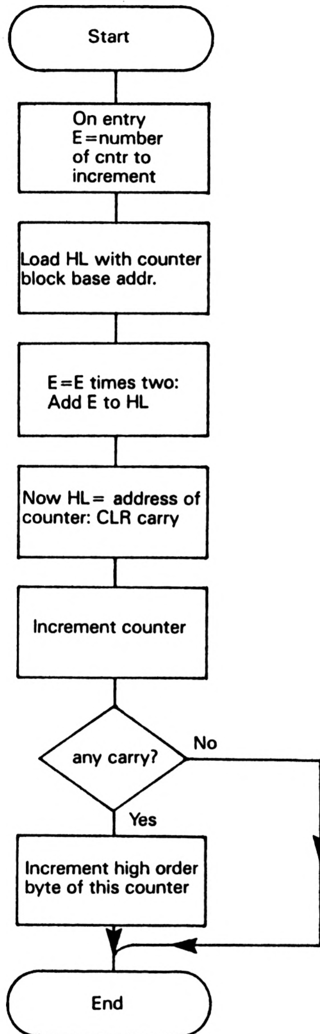
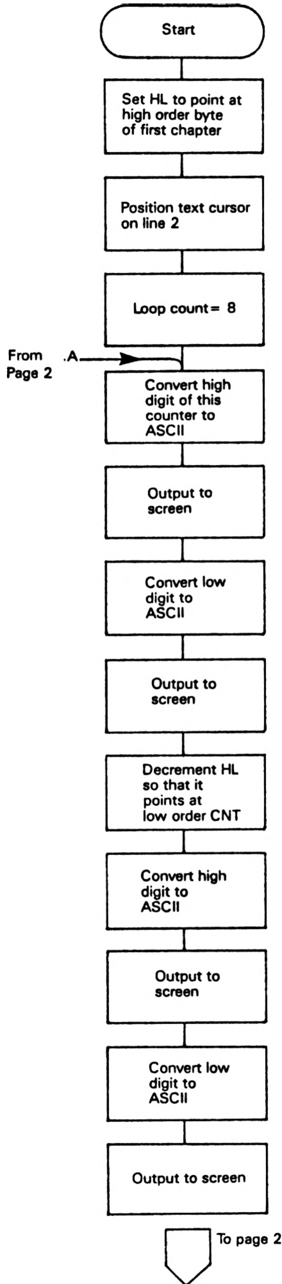


Fig. 5.8 Flowchart for "dispup" Page 1 of 2



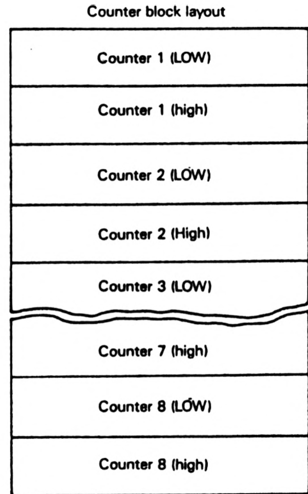
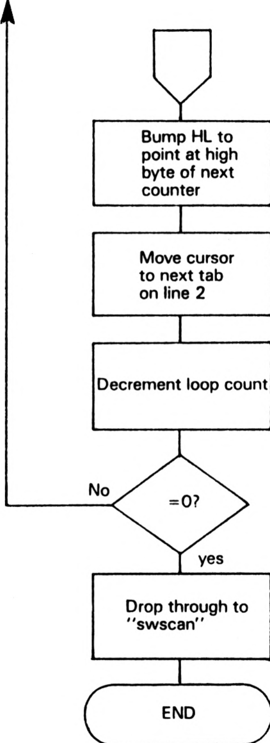


Fig. 5.8 Page 2



to zero all counters (except the pass counter), then create the display screen, using firmware routines to clear the screen, and print a catagory name to go with each counter: Hand over control to the switch scanner section. There must be a re–entry point in the initialisation section to allow everything to be restarted except the pass counter. This is for use after writing everything to a file. Fig 5.5 shows a flowchart for these actions.

The switch scanner section must: Call a firmware subroutine to see if the user has typed either of the actionable key combinations, if # was typed then it must hand over control to the tape I/O section, if CTRL↑has been pressed then it must return – typically to BASIC. If no actionable characters have been typed at the CPC keyboard the program should read the state of port B on PPI #2, then check each bit in turn to see if the switch connected to it was pressed, If a switch is found to be pressed, wait for its release. When the key is released implement a debounce delay. Hand over control to the counter update section, sending it the number of the switch which was pressed. Fig 5.6 is the flowchart representation of these steps.

The counter update section will increment one of the eight counters, the counter which is incremented is the one which is indicated by the value received from the switch scanner. The count is to be kept in decimal – not in HEX. When this is done control is handed over to the display update section. Fig 5.7 shows the flowchart for the counter update section.

The display update section must convert the Binary Coded Decimal (BCD) values of the counters to ASCII codes which can be sent to the CPC firmware for display at their appointed places on the screen. After this has been done control is handed back to the switch scanner. The flowchart for this section is fig 5.8.

The I/O section sets up a call to some firmware routines which will write the values of the 8 counters into a file, obeying the CPC conventions on file opening and closing. After the file has been written the pass counter is incremented: Control now passes to the re–entry point in the initialisation section – ready for another pass. The flowchart for this is shown in figure 5.9.

Developing programs from flowcharts

We have seen the flowcharts for the data logger application, now we get to the final phase of writing our example program, which is to convert the flowcharts into executable programs.

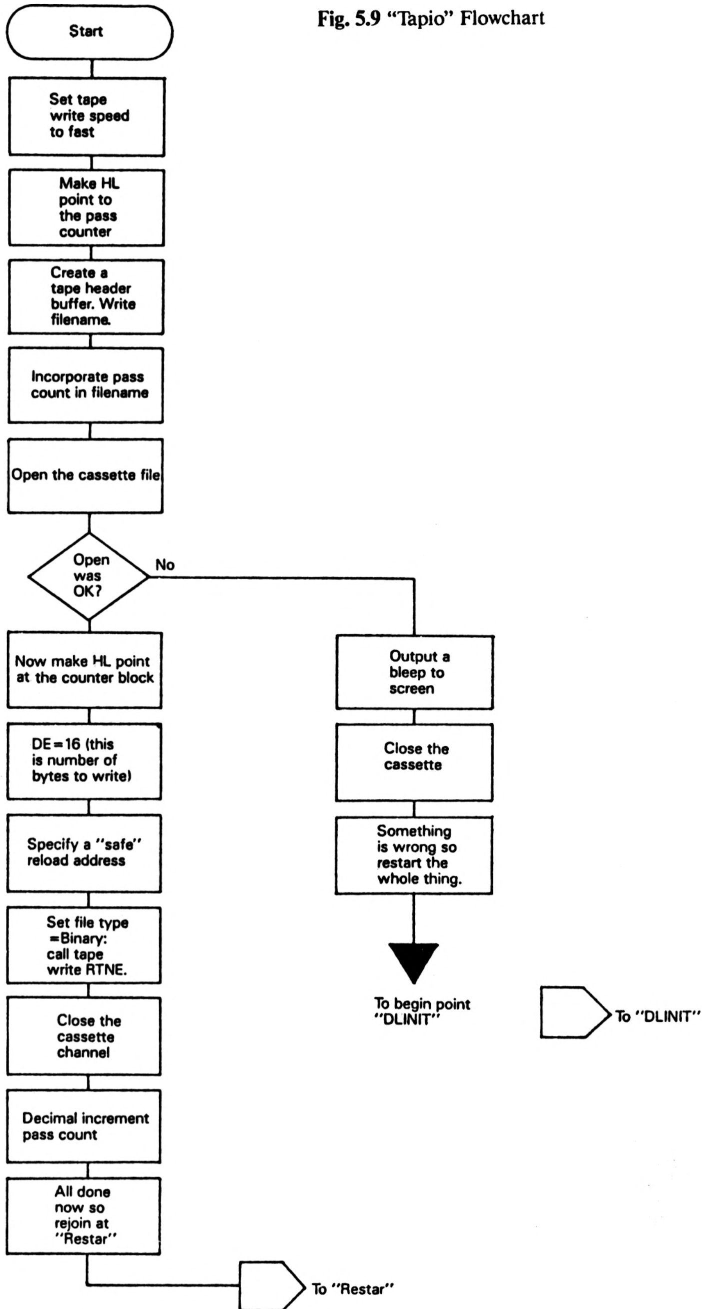
In the following detailed explanations of the module listings, when an instruction has not been referenced before, the number allocated to it in the list of instructions in chapter two will appear beside it in brackets, for example INC IX (Ref 50). Also as the various assembler directives and features are encountered they will be explained.

Figure 5.10 is the listing of an assembly language program to implement the initialisation module which will be called DLINIT (Data Logger INITialisation – geddit?).

At the head of the listing are some remarks which explain the purpose of the program, and its rough layout. Line 210 is the first line which actually does anything. This contains an ORG directive, which tells the assembler to create the machine code starting at address Hex 4E20 (Decimal 20000). If for any reason you wanted to have the program constructed by the assembler to run at any other address (maybe to use in a stand alone unit) you should change this. Lines 220 & 230 perform two functions. Firstly, they represent a definite point to associate with the labels PASS and CBASE. Secondly, they reserve one byte to be the PASS counter, and 16 bytes to be the switch press counters. It might be useful to explain these two directives in some more detail.

The directive DEFB (Define Byte) tells the assembler to place the value after the DEFB (in this case zero) into the next byte which its location counter points to. The assembler's location counter points at the memory address where the

Fig. 5.9 "Tapio" Flowchart



next machine code generated will be placed, in this case, the indicated value is placed there instead of a machine code.

```
 10
 20 ;            ** Data logger **
 30
 40 ;
 50 ;
 60 ;
 70 ;
 80 ;    The program is split into the following modules:
 90 ;
100
110 ;       1) "DLINIT" - Initialisation
120 ;       2) "SWSCAN" - Switch scanner
130 ;       3) "CUPDAT" - Counter update
140 ;       4) "DISPUP" - Display update of counters
150 ;       5) "TAPIO" - Tape I/O section.
160
170 ; ** In addition, there are several embedded
180 ; ** subroutines
190
200
210         ORG  20000
220 PASS:   DEFB 0    ; Define the PASS counter
230 CBASE:  DEFS 16   ; Reserve 16 bytes for the counters
240 DATLOG: ENT  $    ; Here we go!!
250
260 DLINIT: LD   BC,#F8F7
270         LD   A,#AE ; Load the mode byte into PPI #2
280         OUT  (C),A
290         LD   IX,PASS ; IX Points at PASS counter
300         LD   (IX),#0 ; Zero the pass counter
310 RESTAR: LD   D,16   ; Prepare to zero 8 two byte counters
320         LD   IX,CBASE ; Now IX points at counters
330 INIT00: LD   (IX),#0  ; Zero a counter
340         INC  IX ; Bump IX
350         DEC  D
360         JR   NZ,INIT00 ; if any more counters loop again
370         LD   A,#04     ; Now we set screen to mode 2
380         CALL #BB5A     ; Use firmware to out char.
390         LD   A,#2      ; Mode 2 please
400         CALL #BB5A
410         LD   IX,LABTXT ; Now make IX point to Label texts
420         LD   D,8       ; And D to give us 8 loops
430         LD   B,#1 ; Use B as screen line number
440         LD   C,#1 ; Use C as screen column number
450 INIT01: CALL CURPOS    ; Call s/r to position cursor
460 INIT02: LD   A,(IX)    ; Get a byte of text to A
470         INC  IX        ; bump the pointer
480         CP   #FF       ; See if end of entry
490         JR   Z,INIT03 ; Jump if so
500         CALL #BB5A     ; if not then output the character
510         JR   INIT02    ; and loop again
520 INIT03: LD   A,C       ; Load the column count to A
530         AND  #07       ; see if we are at a TAB point
540         JR   Z,INIT04 ; Jump if yes
550         INC  C         ; Else inc C until we are
560         JR   INIT03    ; loop some more
570 INIT04: INC  C ; Move C of the TAB for next time
580         DEC  D ; See if all 8 headings displayed
590         JR   NZ,INIT01   ; jump if not.
600         JP   SWSCAN  ; All done if so. Go in to SWSCAN
610
620
630 CURPOS: LD   A,31    ; ** S/R to position the text cursor
640         CALL #BB5A   ; C= Col number  B=screen line
```

```
650          LD    A,C     ; Send column
660          CALL #BB5A
670          LD    A,B     ; Send line
680          CALL #BB5A
690          RET   ; And now all done so ret.
700
710 ; ** The following are only examples. Up to 7 chars max.
720 LABTXT: DEFM "SPARROW"
730          DEFB #FF
740          DEFM "OWLS"
750          DEFB #FF
760          DEFM "WRENS"
770          DEFB #FF
780          DEFM "CROWS"
790          DEFB #FF
800          DEFM "TITS"
810          DEFB #FF
820          DEFM "CURLEW"
830          DEFB #FF
840          DEFM "ROBINS"
850          DEFB #FF
860          DEFM "FINCH"
870          DEFB #FF
880
```

Fig. 5.10

The directive DEFS (used in the listing beside the CBASE label) tells the assembler to place zero into the indicated number of bytes. In the case of line 230 what we are doing is to reserve 16 bytes of RAM memory to use as counters. These 16 bytes are collectively referred to as the counter block. The first byte in the block is known by the label CBASE (Counter BASE address). With two bytes for each counter the counter block layout, in relation to the location CBASE looks like this:

CBASE = Counter 0 low byte
CBASE + 1 = Counter 0 High byte
CBASE + 2 = Counter 1 low byte
CBASE + 3 = Counter 1 High byte
CBASE + 4 = Counter 2 low byte
CBASE + 5 = Counter 2 High byte
CBASE + 6 = Counter 3 low byte
CBASE + 7 = Counter 3 High byte
CBASE + 8 = Counter 4 low byte
CBASE + 9 = Counter 4 High byte
CBASE + 10= Counter 5 low byte
CBASE + 11= Counter 5 High byte
CBASE + 12= Counter 6 low byte
CBASE + 13= Counter 6 High byte
CBASE + 14= Counter 7 low byte
CBASE + 15= Counter 7 High byte

If we wanted to increment counter five we increment the location CBASE+10, and if that overflows we also increment CBASE+11.

Line 240 contains an ENT $ directive. This informs the assembler that the ENTry point when running the program is here.

Line 260 loads the BC register with the address of the control register of PPI #2 using the LD rp,xxxx instruction (REF 64). Line 270 loads the value required to set up PPI #2 to the configuration we require, that is, with port B as an input. This is done with a LD r,xx instruction (REF 65). Line 280 actually outputs the value to PPI #2, with an OUT (C),A instruction (REF 107).

Line 290 loads the address of the PASS counter into the IX register, using LD IX,xxxx (REF 89). The assembler fills the xxxx field with the address of the PASS label. Next at line 300 the PASS counter is zeroed using LD (IX),#0 (REF 73).

Line 310 is the point at which the program is restarted after the count values have been written to tape. It is also the start of a loop to zero all entries in the counter block. The D register is set up as a loop counter, with 16 loops to be done. Line 320 places the base address of the counter into IX. (one of the development bugs occurred here. This line originally incremented IX but this only works if IX already contains the address of PASS, on any other than the first pass this will not be the case). The actual loop begins at line 330 which is labelled INIT00. At this line we zero the counter which IX currently points at. In line 340 we use INC IX (REF 50) to increment the IX regsiter so that it points to the next counter for next time around the loop. Then at line 350 we use DEC r (REF 26) to decrement the loop count in the D register. Line 360 uses JR cc,xx (REF 62) to see if the loop has been executed 16 times. If not, then line 330 executes again. Once again the assembler fills the xx field of the Jump Relative instruction with the required value to get back to the INIT00 label location.

Now we have reached line 370. The screen mode of the CPC can be set by outputting a code of 4 to the screen software, followed by the mode number (see CPC user manual chapter nine page two). We want to set mode 2 so at line 370 we load the A register with four, and then at line 380 we use CALL xxxx (REF 17) to access a firmware subroutine which outputs the value found in the A register to the CPC screen. At line 390 and line 400 the screen is sent the value of two in the same way, the screen is now set to mode two. At line 410 the IX register is loaded with the address of the label texts. These are the labels which will appear above the counters on the screen to identify what each counter represents. Once again the xxxx field is filled during the assembly with the address of the LABTXT label.

Line 420 sets up the D register as a loop counter again. Line 430 and line 440 set the B register and the C register both to one. These values are parameters to the CURPOS subroutine, which will be encountered a little later at lines 630 to line 690. What CURPOS does is to position the text cursor on the screen at the coordinates indicated by the B and C registers. This can be thought of as the same as a LOCATE command in BASIC (see CPC 464 user manual, chapter eight page 25). Line 450, labelled INIT01 is at the start of a loop which prints up all the counter labels. Line 460 uses LD r,(IX) (REF 71) to fetch a byte from the label text list into the A register. Then line 470 increments the IX pointer to

the text, and line 480 uses CP s (REF 19) to see if what we just fetched was an end of text indicator with a Hex value of FF. Line 490 uses JR cc,xx (REF 61) to jump if it was the end of the text entry. Line 500 is executed if it was not the end of the entry, and it calls the firmware routine to output the character to the screen. Line 510 jumps back to line 460 for another loop. If the end of an entry has been found, then line 520 executes, and the C register, which remember still contains the screen column count, is loaded into the A register. Lines 520 through to 560 use AND s (REF 11) to increment the C register until it contains all zeroes in bits 0–2. When this is so then the column it will position the cursor at, when used in a call to CURPOS, will be a TAB stop. Then at line 570 using INC r (REF 45), the value in C is advanced by one, so that we do not get stuck for next time round! Line 580 and 590 test to see if all the headings have been printed on the screen, and loop back to INIT01 if not.

Line 600 uses JP xxxx (REF 57) to jump directly to the SWSCAN section of the program, since the initialisation is now completed.

Line 630 is the start of the CURPOS subroutine. This is a self contained routine to output codes to the text VDU to make it position the cursor at the column indicated by the value in the C register, and the screen line indicated by the B register contents. It does this by using the control codes method detailed in the user manual. Line 630 loads the first code into the A register, line 640 calls the firmware routine to output that code. Line 650 loads the column number into the A register from the C. Line 660 calls the output routine again. Line 670 loads the line number into A, and line 680 does the screen call. Once this is done then the RET instruction (REF 118) at line 690, causes the return to whichever routine called CURPOS.

LINE 720 is the start of the label texts and it has the label LABTXT. It also has an assembler directive which we have not yet used, this is called DEFM (DEFine Memory). This directive requires some text in quote marks after it. It takes the ASCII codes for the characters within the quote marks and it places them sequentially into the next locations according to its location pointer. Using DEFM you can create text buffers in memory, a text buffer being an area of memory to be copied to the screen or to tape at a certain point in the execution of the program. NOTE: when you assemble the program only the codes for the first four characters are shown, but the rest are placed in memory. The text buffer definition extends throgh to line 870, with DEFB directives being used to insert separators between names.

In figure 5.11 the SWSCAN routine is listed. lines 900 to line 980 contain a brief explanation of the SWSCAN routine. LINE 1000 is labelled SWSCAN, and contains a call to a firmware routine which checks to see if any of the main CPC keyboard keys have been pressed. If any of them have, then the code for that key is passed back to us in the A register, and the C flag is set. If no keys have been pressed then the C flag will be clear. LINE 1010 causes a jump if the C flag is clear, otherwise at LINE 1020 we see if the code passed back in the A register is the one for the # character. If it is then line 1030 jumps off to the cassette I/O routine TAPIO. If the key which was pressed was not # then line 1040 checks to see if it was CTRL/^. LINE 1050 uses RET cc (REF 119) to return to

BASIC if it was. LINE 1060 loads the address of port B into the BC register, and line 1070 uses IN r,(C) (REF 43) to read the state of the external switches. LINE 1080 jumps back to line 1000 if there were no switches pressed.

```
880
890
900 ; **      SWSCAN - switch scanner for the data logger
910 ; **   This routine is activated after the initialisation
920 ; **   and constitutes the kernel of the logger software
930 ; **   It waits until one of the switches connected to
940 ; **   port B of PPI #2 is pressed, then it passes the
950 ; **   Switch number on to the counter update rtne.
960 ; **   If the # key on the CPC 464 Keyboard is pressed
970 ; **   Then the current counter values and the PASS count
980 ; **   are written to tape by TAPIO which this rtne JPs to.
990
1000 SWSCAN: CALL #BB09 ; Call a firmware Rtne to see if any
1010         JR   NC,SCAN01 ; CPC 464 keys pressed. Jump if not
1020         CP   35 ; see if it was #
1030         JP   Z,TAPIO ; Jump if yes - ignore if no
1040         CP   #F8 ; See if CTRL/UPARROW
1050         RET  Z  ; Yes? RET to caller (No stack though)
1060 SCAN01: LD   BC,#F8F5 ; BC points at PPI #2 Port B
1070         IN   E,(C) ; Read port B
1080         JR   Z,SWSCAN ; If no switches pressed then loop
1090         LD   D,#80 ; Load a bit mask into D
1100         LD   B,8   ; set up eight loops
1110 SCAN03: LD   A,E   ; Get port B value into A
1120         AND  D  ; AND A with D register
1130         JR   NZ,SCAN02 ; Jump if tested bit was not zero
1140         SRA  D  ; Shift D right by 1 bit.(advance mask)
1150         DJNZ SCAN03 ; DEC B and jump if not zero
1160         JR   SWSCAN ; Port B not 0 but no sw pressed, Hmm
1170 SCAN02: LD   E,B ; Copy switch number into E to preserve it
1180 ; ** Note the value in E is actually switch number +1
1190 ; ** Becuase the loop counter is used as switch number
1200         LD   BC,#F8F5 ; Now we must wait for release of SW
1210 SCAN04: IN   A,(C)
1220         AND  D ; See if the switch is still pressed
1230         JR   NZ,SCAN04 ; Loop if so
1240         LD   B,#80 ; Debounce delay
1250 SCAN05: DJNZ SCAN05      ; Loop 128 times
1260         DEC  E ; Make E = real counter number (ie 0-7)
1270         JP   CUPDAT ; Now with E=sw number that was pressed
1280 ; do the counter update routine CUPDAT.
1290
1300
```

Fig. 5.11

At this point I should like to suggest a program insert which you could write, to allow a form of remote control over the CPC.

When we get to line 1080 we have got the value which was read from port B in the A register. It would be a simple matter to see if two particular switches on the remote switch box were pressed at the same time, and if this were so perform the same action for the # key being pressed, that is, write the current count values to a file. This would give the great advantage that if you were located some distance away from the CPC, or could not emerge from your bird watching hide! you could start your session by loading the program, and

then place a blank logging cassette in the datacorder, and switching it to RECORD, or for a 664 place a blank disc in the drive. When you are actually logging all you need do is to press two buttons at the same time and the program would save your data, with no need for you to physically return to the computer. (Note that for a CPC 464 you would also have to insert a CALL to CAS NOISY, with A= non zero value to turn off the cassette prompts). Then you wait for about ten seconds and carry on. Inserting this extra feature would be good practice for a fledgling assembly language programmer!

At line 1090 we put a mask value into the D register. A mask is a particular bit pattern which is used to test the state of that same bit in another register, you will see how it is used here as we go along. LINE 1100 sets up eight loops to be done, using B as a loop counter. This is more efficient due to the existence of the `DJNZ xx` command (REF 31). At line 1110 we load the value which was read from port B into the A register. Then at line 1120 we use `AND s` (REF 11) to see if the bit which we have set in the mask is also set in the value we got from port B, line 1130 jumps if so. LINE 1140 uses `SRA r` (REF 141) to shift the contents of register D right by one bit. D contains the mask value, which was initially set to Hex 80 (binary 1000 0000). Then at line 1160 we decrement the B register, and if it has not reached zero then we go to do another loop at `SCAN03`: If it was zero then the impossible has happened! We got into the loop because what we read from port B was not zero, yet having tested all the switches we found none pressed. As my somewhat thoughtful remark indicates for this line, better try again.

An example should make the operation of the mask idea clear. After we have executed the loop which begins at the point labelled `SCAN03`, say twice, then the D register should contain Hex 20 (Binary 0010 0000). Using this method gives us a way to test each bit in another register by simply using the `AND` function with the mask as one operand and the register whose contents we wish to test as the other. As detailed in Appendix six when an eight bit AND function is carried out then only bits which were set in both operands will be set in the result. So if we have 0010 0000 in D and in the A register we have 0001 0000, then the result will be 0000 0000. So we set only one bit in the mask, which is the bit number we are currently testing.

By the time we arrive at line 1170 the B register contains the number of the switch which was pressed, or does it? Look back to line 1100. We set up eight loops, and then at line 1150 we test to see if the loop count has reached zero, and never go through the loop again if it has. This means that the number in the B register when we come out of the loop can never be zero, yet we number the switches from zero to seven. In fact the B register will contain the switch number plus one. At line 1170 therefore we save the slightly inaccurate switch number into the E register.

In the wait loop comprised of lines 1200 to 1230, port B is re read and the mask in D is used to see if the switch is still pressed. We loop around lines 1200–1220 until the switch is released. After it has been released we do another loop starting at line 1240, which is to allow any switch bounce to occur harmlessly.

Finally, at line 1260 we make the contents of the E register into a real switch number by decrementing it. Now SWSCAN is finished, and control passes to the counter update routine CUPDAT, via the JP statment at line 1270. We send CUPDAT the number of the counter we want it to increment in the E register.

CUPDAT starts at line 1300. Figure 5.12 is the listing for this module of the data logger. After some remarks line 1380 loads zero into the H register, and line 1390 uses RL r (REF 122) to rotate a known zero into the carry flag bit of the F register. Then at line 1400 the HL register pair is loaded with the base address of the counter block (CBASE). As we saw in the previous section about SWSCAN this routine gets the number of the switch to be incremented sent to it in the E register. What we now need to do is to convert this into an offset into the counter block, so that once we have added this offset to the contents of the HL register pair, they will point at the address of the counter to be incremented. We do this by shifting the E register to the left by one, which when dealing with small numbers (that is, those which do not use bit 7) is a good way to double any number. This is done to the E register at line 1410. So now that E contains a usable offset we get the lower byte of the HL register into the A register, and at line 1430 we use ADD A,r (REF 7) to add the offset, then we put back the updated value into L at line 1440. Now HL points directly at the low order byte of the counter to be incremented.

```
1300
1310 ; ** This is the counter update routine- CUPDAT.
1320 ; ** On entry the E register contains the counter number
1330 ; ** to be updated. The value supplied is adjusted to make
1340 ; ** it an offset into the counter block.
1350 ; **  The counter is incremented decimally using the DAA
1360 ; ** instruction of the Z80A
1370 ;
1380 CUPDAT: LD   H,0
1390         RL   H ; Make sure carry is clear
1400         LD   HL,CBASE ; HL points at counter block
1410         RL   E; Each CTR=2 Bytes so shift E left
1420         LD   A,L ; (eg 4 becomes 8) Now copy L to A
1430         ADD  A,E ; Now add offset to HL
1440         LD   L,A ; Put back updated L. HL now=addr of ctr.
1450         LD   A,(HL); Get the counter value
1460         INC  A; And Increment it
1470         DAA  ; Decimal adjust it to keep to BCD
1480         LD   (HL),A; put it back into CTAb
1490         JR   NC,ALLDUN; see if there was a carry
1500         CCF  ; If carry was set then clear carry it
1510         INC  HL; Bump HL to next counter location.
1520         LD   A,(HL) ; Get this counter value
1530         INC  A; increment it
1540         DAA  ; decimalise it
1550         LD   (HL),A; put it back
1560 ALLDUN: NOP  ; Now we've done, so drop through to DISPUP
1570
```

Fig. 5.12

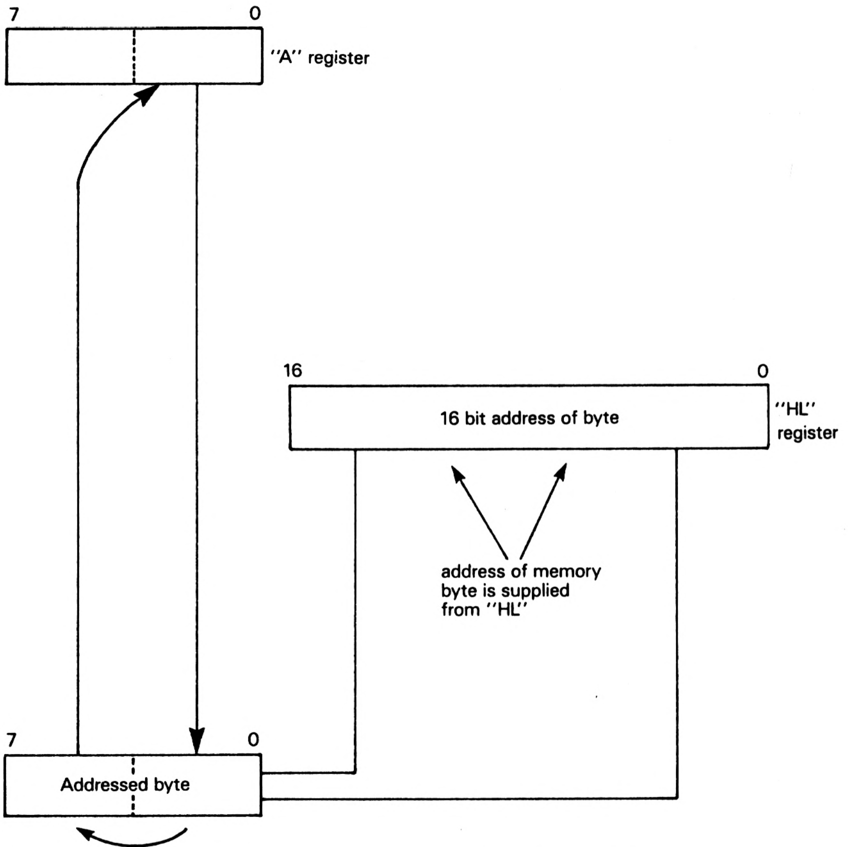
At line 1450 we fetch the counter value into the A register, then line 1460 increments it. When line 1470 has decimally adjusted the new count value using the DAA instruction (REF 25), it is placed back into the counter table block at line 1480. When we get to line 1490 we test to see if the low order byte overflowed (ie went from 99 to 00). If it did not then we jump to line 1560, labelled ALLDUN. If the carry WAS set then we clear it using the CCF (REF 18) instruction which inverts its state. Next line 1510 increments the HL register using INC rp (REF 46). HL now points to the high order byte of the selected counter. Then in lines 1510 to 1550 we do exactly the same as we did to the low byte. At line 1560 we are, so far as CUPDAT is concerned, truly ALLDUN.

Line 1630 is effectively the start of the DISPUP routine, which updates the display on the CPC screen. The listing for this is shown in figure 5.13. At line 1630 we load the HL register pair with the address of the high byte of the first counter. A little thought will reveal the reason for this. When a counter value is to be printed on the screen, we require the highest digits to be printed first, for example, 1500 rather than 0015 for fifteen hundred. At lines 1640 to 1660 we use our old friend CURPOS to position the cursor at line two of the screen, and at column one. With this done we set the B register to eight to begin eight loops at line 1680. This is where things can become a little complex!

```
1580
1590 ;   ** This is the counter display updater - DISPUP.
1600 ;   ** It prints all eight counters, on screen line 2
1610
1620
1630 DISPUP: LD   HL,CBASE+1 ; HL = ADDR high byte of 1st ctr.
1640         LD   B,2 ; Select screen line 2
1650         LD   C,1 ; and column 1
1660         CALL CURPOS ; Get the cursor positioned
1670         LD   D,8 ; Set up 8 loops for 8 two byte ctrs.
1680 DISP01: LD   A,#30 ; Make A high nibble =3
1690         RLD  ; Move 1st digit into A low. (This is MSD)
1700         CALL #BB5A ; Output the digit
1710         RLD  ; Move in the 2nd digit
1720         CALL #BB5A ; output it
1730         RLD  ; Restor orig. state of (HL)
1740         DEC  HL      ; HL points to low ord byte of counter
1750         RLD  ; Move in 3rd digit
1760         CALL #BB5A ; output it
1770         RLD  ; Move in 4th digit (The LSD)
1780         CALL #BB5A ; print it
1790         RLD  ; Restore (HL) to aswas state.
1800         LD   A;L      ; Now we bump L reg by three so it
1810         ADD  A,#3     ; points at high byte of next counter
1820         LD   L,A      ; Put updated L back
1830         LD   B,4      ; Now six loops of a print curs right
1840         LD   A,9      ; loop, to position curs at next TAB
1850 DISP02: CALL #BB5A
1860         DJNZ DISP02 ; Loop till its been done six times.
1870         DEC  D       ; see if all counters displayed
1880         JR   NZ,DISP01 ; If no then it's go round again time
1890         JP   SWSCAN     ; If so then we go back to scanning
1900
```

Fig. 5.13

Fig. 5.14: Page 2 RLD Instruction Action



When RLD is executed the memory address which HL points at is shifted left by four bits. The lower four bits are loaded with the lower four bits of the "A" register. Finally the original contents of the top four bits of the memory location (HL) are placed in the lower 4 bits of "A" register.

What we are starting to do at line 1680 is to convert each of the four digits contained in two bytes, for each counter, into ASCII ready for output to the screen firmware. The way we do this is with the RLD instruction (REF 129). If you are not already aware of what the RLD instruction does, then look at its description in chapter two, and also at figure 5.13a which diagramatically illustrates its action.

To begin with at line 1680 the A register is loaded with Hex 30, because all ASCII numbers have a high nibble with the value three. (See appendix 4 for an ASCII chart.) Line 1690 turns the A register contents into the ASCII code for the highest digit, and line 1700 calls the output routine. At line 1710 the next digit is moved in to the A register and output at line 1720, then at line 1730 the contents of the byte pointed to by the HL register are restored with yet another RLD. In line 1740 the HL register is decremented so that HL now points at the low order byte of the counter. From line 1750 to 1790 exactly the same sequence is carried out on the low digit. Line 1800 fetches the low order byte of HL into the A register, and then at line 1810 it has three added to it, in order to make it point at the next high counter byte in the table. Then at line 1820 the updated L is put back into the HL register. We have now printed on screen one count value.

Line 1830 loads B with a loop value of four, because we now want to move the cursor at the next TAB stop along line two of the screen. We could have used CURPOS again, but here we will use a different method. The A register is loaded with nine, which when it is output to the screen move the cursor one character position to the right. The B register is set up to make the loop execute four times, because there are eight characters per TAB zone, and we have just used four to print the numbers, therefore we need to advance four positions to the next TAB stop point. We do this is the loop which comprises line 1850, and 1860. After this has been done at line 1870 we decrement D to see if all the counters have been printed, if not then we loop again at line 1680. If not, then we jump directly back to SWSCAN.

Figure 5.15 shows the TAPIO routine. This routine writes a file containing the current counter values, and the current value of the PASS counter. The tape file is always called "DATA LO PASS xx" where xx is the PASS count for the time that the file was written. The actual tape I/O is done by a firmware routine called CAS OUT DIRECT.

For a CPC 664 you should change the filename set up in line 2370 to something like "DLP" to make a legal disk filename. The rest holds good. After the explanations in the remarks the program begins at line 1990 by setting up a call to a firmware routine which decides what speed the tape data is to be written at. (This is equivalent to a SPEED WRITE command from BASIC). This firmware routine requires that upon entering it there should be a value in the HL register which is to be half the length of a zero bit. Also there should be a value in the A register which tells it how much precompensation to apply. In the case of line 1990 and line 2000 we are setting up the same as the BASIC command SPEED WRITE 1, ie 2K. at lines 2020 and 2030 we ensure that the carry flag is clear.

At lines 2370, 2380 and 2390 there are three DEFine directives which represent a small buffer to allow us to build a complete file header for the cassete firmware. This consists of NAMBUF: which consists of the first part of the file name. Then PASCH and PASCL are the two bytes where the ASCII representation of the PASS count will be placed.

```
1900
1910
1920 ; ** This is the TAPIO section of the data logger
1930 ; ** software. It scoops up the values of all the
1940 ; ** counters, and the pass counter, and uses one
1950 ; ** of the firmware routines to write them into a
1960 ; ** tape file. This file will be called -
1970 ; ** "DATA LOG PASS nn" where nn is the pass count
1980
1990 TAPIO:  LD   HL,167 ; First set the cassette spd to fast
2000         LD   A,50
2010         CALL #BC68 ; Call firmware rtne CAS SET SPEED
2020         LD   H,0 ; Ensure carry flag is =0
2030         RL   H ;
2040         LD   HL,PASS ; Set HL to point at the pass counter
2050         LD   A,#30   ; Set upper digit of A to 3
2060         RLD
2070         LD   (PASCH),A ; Put High digit into file name
2080         RLD
2090         LD   (PASCL),A ; And then the low digit
2100         RLD
2110         LD   HL,NAMBUF ; Load the file name address to HL
2120         LD   B,16 ; The length of the filename to B
2130         LD   DE,0 ; Give it a Null buffer- we don't use it
2140         CALL #BC8C ; CALL the CAS OUT OPEN routine
2150         JP   C,OPENOK ; Jump if the open went OK
2160         LD   A,7 ; Else load beep to A, and beep at screen
2170         CALL #BB5A ; to show we are not amused OPEN failed
2180         CALL #BC8F ; Ensure that cas file is closed
2190         JP   DLINIT  ; and restart the program
2200 OPENOK: LD   HL,PASS ; If we opened ok then make HL point
2210         LD   DE,16 ; at ctr block. DE=size of data to write
2220         LD   BC,#FC00 ; Safe value in Addr. field of hdr
2230         LD   A,2 ; Make file type a binary
2240         CALL #BC98 ; Call CAS OUT DIRECT to write to tape
2250         CALL #BC8F     ; Then close the file
2260         LD   IY,PASS   ; Make IY point at the PASS counter
2270         LD   H,0
2280         RL   H ; Make sure carry flag=0
2290         LD   A,(IY) ; Get the pass count into A
2300         INC  A ; Increment pass count
2310         DAA  ; Decimal adjust it
2320         LD   (IY),A ; put it back
2330         JP   RESTAR ; Now restart everything but
2340 ; the PASS counter by jumping to RESTAR.
2350
2360
2370 NAMBUF: DEFM "DATA LOG PASS " ; This will be filename
2380 PASCH:  DEFB 0 ; These are for the ASCII version
2390 PASCL:  DEFS 2 ; of the PASS counter - for filename.
```

Fig. 5.15

Line 2040 gets the address of the PASS counter into HL. Then using exactly the same technique as was used in DISPUP, using the RLD (REF 129) instruction, the PASS counter is converted to two ASCII bytes which are placed on the end of the NAMBUF name buffer. Then at line 2110 HL is made to point at NAMBUF, and then we set B up to sixteen to show the firmware we are about to call that there are that many characters in the filename. We set the buffer address to zero at line 2130. AT line 2140 we call the firmware cassette file open routine CAS_OUT_OPEN. If the file was opened correctly (ie it wasn't already open or the user didn't type escape) then line 2150 jumps to line 2200. If there

were any problems with the open, then we output a bleep (Character value of seven), and then try to mend matters by calling the cassette file close routine, and then we restart the program with a jump to DLINIT.

Now that we have opened the file, we can set up a call to the CAS-OUT-DIRECT routine which writes areas of memory to tape. Line 2200 loads the address of the PASS counter into HL. (don't forget that one byte above PASS is the counter block start.) Then we make DE the size of the data to be written in bytes. Line 2210 sets this to 17. (PASS plus 16 counter bytes). In a header of an Amstrad format tape file there is always a 16 bit address which shows where the file was in memory when it was written. We don't really care about this information in the current instance, so in line 2220 we set it to the screen address, just for fun! Line 2230 sets the file type to binary, and then at line 2240 all this information is fed into the firmware routine, and the tape write takes place. Line 2250 calls the close a cassette file routine. Lines 2260 to 2320 effectively decimal increment the PASS counter. Then at line 2330 the program is restarted by a jump to RESTAR. As previously mentioned the directives at lines 2370 to 2390 merely reserve some storage for the cassette header buffer.

That concludes the description of the software for the data logger device.

Readers who would like to use the software as it stands, but do not have a DEVPAC (you must get an assembler – so sell the cat!) are referred to the next section of this chapter. Finally the assembly listing for the data logger program is shown as figure 5.16.

```
Pass 1 errors: 00

                    10
                    20 ;             ** Data logger **
                    30
                    40 ; This program gives a way to use project Five from the
                    50 ; "Amstrad CPC 464 Hardware book" as a data logger. See
                    60 ; hardware in chapter two. For software details see
                    70 ; accompanying text in Chapter 5.
                    80 ;    The program is split into the following modules:
                    90 ;
                   100
                   110 ;        1) "DLINIT" - Initialisation
                   120 ;        2) "SWSCAN" - Switch scanner
                   130 ;        3) "CUPDAT" - Counter update
                   140 ;        4) "DISPUP" - Display update of counters
                   150 ;        5) "TAPIO" - Tape I/O section.
                   160
                   170 ; ** In addition, there are several embedded
                   180 ; ** subroutines
                   190
                   200
4E20               210        ORG  20000
4E20   00          220 PASS   DEFB 0    ; Define the PASS counter
4E21               230 CBASE  DEFS 16   ; Reserve 16 bytes for the counters
4E31               240 DATLOG ENT  $    ; Here we go!!
                   250
4E31   01F7F8      260 DLINIT LD   BC,#F8F7
4E34   3EAE        270        LD   A,#AE ; Load the mode byte into PPI #2
4E36   ED79        280        OUT  (C),A
```

```
4E38   DD21204E   290          LD    IX,PASS ; IX Points at PASS counter
4E3C   DD360000   300          LD    (IX),#0 ; Zero the pass counter
4E40   1610       310 RESTAR LD    D,16  ; Prepare to zero 8 two byte counters
4E42   DD21214E   320          LD    IX,CBASE ; Now IX points at counters
4E46   DD360000   330 INIT00 LD    (IX),#0   ; Zero a counter
4E4A   DD23       340          INC   IX ; Bump IX
4E4C   15         350          DEC   D
4E4D   20F7       360          JR    NZ,INIT00 ; if any more counters loop again
4E4F   3E04       370          LD    A,#04      ; Now we set screen to mode 2
4E51   CD5ABB     380          CALL  #BB5A      ; Use firmware to out char.
4E54   3E02       390          LD    A,#2       ; Mode 2 please
4E56   CD5ABB     400          CALL  #BB5A
4E59   DD21914E   410          LD    IX,LABTXT ; Now make IX point to Label texts
4E5D   1608       420          LD    D,8        ; And D to give us 8 loops
4E5F   0601       430          LD    B,#1 ; Use B as screen line number
4E61   0E01       440          LD    C,#1 ; Use C as screen column number
4E63   CD834E     450 INIT01 CALL  CURPOS     ; Call s/r to position cursor
4E66   DD7E00     460 INIT02 LD    A,(IX)     ; Get a byte of text to A
4E69   DD23       470          INC   IX         ; bump the pointer
4E6B   FEFF       480          CP    #FF       ; See if end of entry
4E6D   2805       490          JR    Z,INIT03 ; Jump if so
4E6F   CD5ABB     500          CALL  #BB5A     ; if not then output the character
4E72   18F2       510          JR    INIT02    ; and loop again
4E74   79         520 INIT03 LD    A,C       ; Load the column count to A
4E75   E607       530          AND   #07       ; see if we are at a TAB point
4E77   2803       540          JR    Z,INIT04 ; Jump if yes
4E79   0C         550          INC   C         ; Else inc C until we are
4E7A   18F8       560          JR    INIT03    ; loop some more
4E7C   0C         570 INIT04 INC   C ; Move C of the TAB for next time
4E7D   15         580          DEC   D ; See if all 8 headings displayed
4E7E   20E3       590          JR    NZ,INIT01   ; jump if not.
4E80   C3C34E     600          JP    SWSCAN  ; All done if so. Go in to SWSCAN
                  610
                  620
4E83   3E1F       630 CURPOS LD    A,31     ; ** S/R to position the text cursor
4E85   CD5ABB     640          CALL  #BB5A   ; C= Col number  B=screen line
4E88   79         650          LD    A,C     ; Send column
4E89   CD5ABB     660          CALL  #BB5A
4E8C   78         670          LD    A,B     ; Send line
4E8D   CD5ABB     680          CALL  #BB5A
4E90   C9         690          RET   ; And now all done so ret.
                  700
                  710 ; ** The following are only examples. Up to 7 chars max.
4E91   53504152   720 LABTXT DEFM "SPARROW"
4E98   FF         730          DEFB #FF
4E99   4F574C53   740          DEFM "OWLS"
4E9D   FF         750          DEFB #FF
4E9E   5752454E   760          DEFM "WRENS"
4EA3   FF         770          DEFB #FF
4EA4   43524F57   780          DEFM "CROWS"
4EA9   FF         790          DEFB #FF
4EAA   54495453   800          DEFM "TITS"
4EAE   FF         810          DEFB #FF
4EAF   4355524C   820          DEFM "CURLEW"
4EB5   FF         830          DEFB #FF
4EB6   524F4249   840          DEFM "ROBINS"
4EBC   FF         850          DEFB #FF
4EBD   46494E43   860          DEFM "FINCH"
4EC2   FF         870          DEFB #FF
                  880
                  890
                  900 ; **      SWSCAN - switch scanner for the data logger
                  910 ; **  This routine is activated after the initialisation
                  920 ; **  and constitutes the kernel of the logger software
                  930 ; **  It waits until one of the switches connected to
                  940 ; **  port B of PPI #2 is pressed, then it passes the
                  950 ; **  Switch number on to the counter update rtne.
                  960 ; **  If the # key on the CPC 464 Keyboard is pressed
```

```
                970 ; **  Then the current counter values and the PASS count
                980 ; **  are written to tape by TAPIO which this rtne JPs to.
                990
4EC3  CD09BB   1000 SWSCAN CALL #BB09 ; Call a firmware Rtne to see if any
4EC6  3008     1010        JR   NC,SCAN01 ; CPC 464 keys pressed. Jump if not
4EC8  FE23     1020        CP   35 ; see if it was #
4ECA  CA494F   1030        JP   Z,TAPIO ; Jump if yes - ignore if no
4ECD  FEF8     1040        CP   #F8 ; See if CTRL/UPARROW
4ECF  C8       1050        RET  Z  ; Yes? RET to caller (No stack though)
4ED0  01F5F8   1060 SCAN01 LD   BC,#F8F5 ; BC points at PPI #2 Port B
4ED3  ED58     1070        IN   E,(C) ; Read port B
4ED5  28EC     1080        JR   Z,SWSCAN ; If no switches pressed then loop
4ED7  1680     1090        LD   D,#80 ; Load a bit mask into D
4ED9  0608     1100        LD   B,8   ; set up eigtht loops
4EDB  7B       1110 SCAN03 LD   A,E   ; Get port B value into A
4EDC  A2       1120        AND  D  ; AND A with D register
4EDD  2006     1130        JR   NZ,SCAN02 ; Jump if tested bit was not zero
4EDF  CB2A     1140        SRA  D  ; Shift D right by 1 bit.(advance mask)
4EE1  10F8     1150        DJNZ SCAN03 ; DEC B and jump if not zero
4EE3  18DE     1160        JR   SWSCAN ; Port B not 0 but no sw pressed, Hmm
4EE5  58       1170 SCAN02 LD   E,B ; Copy switch number into E to preserve it
               1180 ; ** Note the value in E is actually switch number +1
               1190 ; ** Becuase the loop counter is used as switch number
4EE6  01F5F8   1200        LD   BC,#F8F5 ; Now we must wait for release of SW
4EE9  ED78     1210 SCAN04 IN   A,(C)
4EEB  A2       1220        AND  D ; See if the switch is still pressed
4EEC  20FB     1230        JR   NZ,SCAN04 ; Loop if so
4EEE  0680     1240        LD   B,#80 ; Debounce delay
4EF0  10FE     1250 SCAN05 DJNZ SCAN05      ; Loop 128 times
4EF2  1D       1260        DEC  E ; Make E = real counter number (ie 0-7)
4EF3  C3F64E   1270        JP   CUPDAT ; Now with E=sw number that was pressed
               1280 ; do the counter update routine CUPDAT.
               1290
               1300
               1310 ; ** This is the counter update routine- CUPDAT.
               1320 ; ** On entry the E register contains the counter number
               1330 ; ** to be updated. The value supplied is adjusted to make
               1340 ; ** it an offset into the counter block.
               1350 ; **  The counter is incremented decimally using the DAA
               1360 ; ** instruction of the Z80A
               1370 ;
4EF6  2600     1380 CUPDAT LD   H,0
4EF8  CB14     1390        RL   H ; Make sure carry is clear
4EFA  21214E   1400        LD   HL,CBASE ; HL points at counter block
4EFD  CB13     1410        RL   E; Each CTR=2 Bytes so shift E left
4EFF  7D       1420        LD   A,L ; (eg 4 becomes 8) Now copy L to A
4F00  83       1430        ADD  A,E ; Now add offset to HL
4F01  6F       1440        LD   L,A ; Put back updated L. HL now=addr of ctr.
4F02  7E       1450        LD   A,(HL); Get the counter value
4F03  3C       1460        INC  A; And Increment it
4F04  27       1470        DAA  ; Decimal adjust it to keep to BCD
4F05  77       1480        LD   (HL),A; put it back into CTAb
4F06  3006     1490        JR   NC,ALLDUN; see if there was a carry
4F08  3F       1500        CCF  ; If carry was set then clear carry it
4F09  23       1510        INC  HL; Bump HL to next counter location.
4F0A  7E       1520        LD   A,(HL) ; Get this counter value
4F0B  3C       1530        INC  A; increment it
4F0C  27       1540        DAA  ; decimalise it
4F0D  77       1550        LD   (HL),A; put it back
4F0E  00       1560 ALLDUN NOP  ; Now we've done, so drop through to DISPUP
               1570
               1580
               1590 ;  ** This is the counter display updater - DISPUP.
               1600 ;  ** It prints all eight counters, on screen line 2
               1610
               1620
4F0F  21224E   1630 DISPUP LD   HL,CBASE+1 ; HL = ADDR high byte of 1st ctr.
4F12  0602     1640        LD   B,2 ; Select screen line 2
```

```
4F14   0E01       1650          LD   C,1 ; and column 1
4F16   CD834E     1660          CALL CURPOS ; Get the cursor positioned
4F19   1608       1670          LD   D,8 ; Set up 8 loops for 8 two byte ctrs.
4F1B   3E30       1680 DISP01   LD   A,#30 ; Make A high nibble =3
4F1D   ED6F       1690          RLD  ; Move 1st digit into A low. (This is MSD)
4F1F   CD5ABB     1700          CALL #BB5A ; Output the digit
4F22   ED6F       1710          RLD  ; Move in the 2nd digit
4F24   CD5ABB     1720          CALL #BB5A ; output it
4F27   ED6F       1730          RLD  ; Restor orig. state of (HL)
4F29   2B         1740          DEC  HL      ; HL points to low ord byte of counter
4F2A   ED6F       1750          RLD  ; Move in 3rd digit
4F2C   CD5ABB     1760          CALL #BB5A ; output it
4F2F   ED6F       1770          RLD  ; Move in 4th digit (The LSD)
4F31   CD5ABB     1780          CALL #BB5A ; print it
4F34   ED6F       1790          RLD  ; Restore (HL) to aswas state.
4F36   7D         1800          LD   A,L      ; Now we bump L reg by three so it
4F37   C603       1810          ADD  A,#3     ; points at high byte of next counter
4F39   6F         1820          LD   L,A      ; Put updated L back
4F3A   0604       1830          LD   B,4      ; Now six loops of a print curs right
4F3C   3E09       1840          LD   A,9      ; loop, to position curs at next TAB
4F3E   CD5ABB     1850 DISP02   CALL #BB5A
4F41   10FB       1860          DJNZ DISP02 ; Loop till its been done six times.
4F43   15         1870          DEC  D        ; see if all counters displayed
4F44   20D5       1880          JR   NZ,DISP01 ; If no then its go round again time
4F46   C3C34E     1890          JP   SWSCAN     ; If so then we go back to scanning
                  1900
                  1910
                  1920 ; ** This is the TAPIO section of the data logger
                  1930 ; ** software. It scoops up the values of all the
                  1940 ; ** counters, and the pass counter, and uses one
                  1950 ; ** of the firmware routines to write them into a
                  1960 ; ** tape file. This file will be called -
                  1970 ; ** "DATA LOG PASS nn" where nn is the pass count
                  1980
4F49   21A700     1990 TAPIO    LD   HL,167 ; First set the cassette spd to fast
4F4C   3E32       2000          LD   A,50
4F4E   CD68BC     2010          CALL #BC68 ; Call firmware rtne CAS SET SPEED
4F51   2600       2020          LD   H,0 ; Ensure carry flag is =0
4F53   CB14       2030          RL   H ;
4F55   21204E     2040          LD   HL,PASS ; Set HL to point at the pass counter
4F58   3E30       2050          LD   A,#30    ; Set upper digit of A to 3
4F5A   ED6F       2060          RLD
4F5C   32B14F     2070          LD   (PASCH),A ; Put High digit into file name
4F5F   ED6F       2080          RLD
4F61   32B24F     2090          LD   (PASCL),A ; And then the low digit
4F64   ED6F       2100          RLD
4F66   21A34F     2110          LD   HL,NAMBUF ; Load the file name address to HL
4F69   0610       2120          LD   B,16 ; The length of the filename to B
4F6B   110000     2130          LD   DE,0 ; Give it a Null buffer- we don't use it
4F6E   CD8CBC     2140          CALL #BC8C ; CALL the CAS OUT OPEN routine
4F71   DA7F4F     2150          JP   C,OPENOK ; Jump if the open went OK
4F74   3E07       2160          LD   A,7 ; Else load beep to A, and beep at screen
4F76   CD5ABB     2170          CALL #BB5A ; to show we are not amused OPEN failed
4F79   CD8FBC     2180          CALL #BC8F ; Ensure that cas file is closed
4F7C   C3314E     2190          JP   DLINIT  ; and restart the program
4F7F   21204E     2200 OPENOK   LD   HL,PASS ; If we opened ok then make HL point
4F82   111000     2210          LD   DE,16 ; at ctr block. DE=size of data to write
4F85   0100FC     2220          LD   BC,#FC00 ; Safe value in Addr. field of hdr
4F88   3E02       2230          LD   A,2 ; Make file type a binary
4F8A   CD98BC     2240          CALL #BC98 ; Call CAS OUT DIRECT to write to tape
4F8D   CD8FBC     2250          CALL #BC8F    ; Then close the file
4F90   FD21204E   2260          LD   IY,PASS  ; Make IY point at the PASS counter
4F94   2600       2270          LD   H,0
4F96   CB14       2280          RL   H ; Make sure carry flag=0
4F98   FD7E00     2290          LD   A,(IY) ; Get the pass count into A
4F9B   3C         2300          INC  A ; Increment pass count
4F9C   27         2310          DAA  ; Decimal adjust it
4F9D   FD7700     2320          LD   (IY),A ; put it back
```

```
4FA0  C3404E     2330         JP   RESTAR ; Now restart everything but
                 2340  ; the PASS counter by jumping to RESTAR.
                 2350
                 2360
4FA3  44415441   2370 NAMBUF DEFM "DATA LOG PASS " ; This will be filename
4FB1  00         2380 PASCH  DEFB 0 ; These are for the ASCII version
4FB2             2390 PASCL  DEFS 2 ; of the PASS counter - for filename.

Pass 2 errors: 00

ALLDUN 4F0E   CBASE  4E21   CUPDAT 4EF6   CURPOS 4E83
DATLOG 4E31   DISP01 4F1B   DISP02 4F3E   DISPUP 4F0F
DLINIT 4E31   INIT00 4E46   INIT01 4E63   INIT02 4E66
INIT03 4E74   INIT04 4E7C   LABTXT 4E91   NAMBUF 4FA3
OPENOK 4F7F   PASCH  4FB1   PASCL  4FB2   PASS   4E20
RESTAR 4E40   SCAN01 4ED0   SCAN02 4EE5   SCAN03 4EDB
SCAN04 4EE9   SCAN05 4EF0   SWSCAN 4EC3   TAPIO  4F49

Table used:   371  from  1146
Executes: 20017
```

Fig. 5.16

WOT! No assembler?

I can easily appreciate that there will be many readers who, having bought their CPC, will have exhausted the household authorities willingness to authorise further computer related expenditure. This will naturally prevent them from getting any extra software such as an assembler package. In theory at least, it should be possible to write an assembler in BASIC, but it would be a mammoth task! A more practical approach is to find ways to use published machine code listings, loading them via BASIC. This is easy to do provided you have two items of information about the machine code program. One the list of machine codes. This is almost always listed either as a Hex dump, or in an assembly listing. The second thing you have to know (Especially with code for Z80 which may not be relocatable in memory) is where in memory the program must be loaded to run correctly. With published software you should get this information provided as well.

The BASIC program listed in figure 5.17 should be used to take an area of memory which contains a machine code program, then it will convert it into a BASIC compatible file which will have DATA statements with the machine codes in them. The program will write the resulting listing to tape or disk. Of course this approach is only of use if you already have the machine code program available for loading. This may not be the case. If you are working from a listing only, then you should enter each machine code into DATA statements. (Remember that the format "1000 DATA &4E,&80" is OK – so you don't have to tediously convert each Hex code to decimal). When you have constructed your DATA statements list, you then only need to put a few READ and POKE statements at the head of the program, and then use CALL to run the machine code program you have just placed in memory. This is illustrated in Figure 5.18 which shows how to load the data logger software in exactly this way. Don't forget that you have to inform BASIC that it must not

use the memory you have loaded the machine code into, you do this with the MEMORY statement. I refer from now on to BASIC programs which contain machine code programs in DATA statements as HEXBAS programs.

```
50      '**           OBJ to BAS converter        **

100     '** This program takes an area of memory which contains a
         ** previously loaded machine code program, and converts it
         ** to a BASIC compatible text file. This can then be MERGED
110     '** with other BASIC files.
120      CLS: MODE 2: LOCATE 4,5: INPUT "Name of file to create on tape";
         FIL$: SPEED WRITE 1: IF LEN(FIL$) > 16 THEN FIL$=LEFT$(FIL$,16):
         PRINT "Too long.. name has been truncated to ";FIL$
130      INPUT "Memory address to begin dump from";AD
140      INPUT "How many bytes to convert";BYTES
150      INPUT "First DATA statement number";STAT
160      DIM R(BYTES+15)
170      FOR I= AD TO (AD+BYTES): R(I-(AD-1))=PEEK(I): NEXT:
         OPENOUT FIL$:
180      DUN=0
190      PRINT #9, STAT;" DATA";: FOR I=1 TO 10 : PRINT #9,R(I+DUN);:
         IF I =10 THEN PRINT #9 ELSE PRINT #9,",";
200      NEXT I: STAT=STAT+10: DUN=DUN+10: IF DUN < BYTES THEN GOTO 190
         ELSE CLOSEOUT
32767    END
```

Fig. 5.17: Machine code to DATA statements conversion program

```
500    '**        DATA LOGGER.BAS  ...      V1B Jan 1985
1000   D$="This is the DATA LOGGER software HEXBAS version. See chapter":
       E$="five for more details"

1100   MEMORY 1999 CLS: PRINT D$;E$;STRING$(5,10):
        PRINT "Loading machine code to memory ": FOR I=20000 TO 20404:
        READ R: POKE I,R: PRINT I;CHR$(13);: NEXT: CALL 20017
10000  DATA 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
10010  DATA 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 247 , 248
10020  DATA 62 , 174 , 237 , 121 , 221 , 33 , 32 , 78 , 221 , 54
10030  DATA 0 , 0 , 22 , 16 , 221 , 33 , 33 , 78 , 221 , 54
10040  DATA 0 , 0 , 221 , 35 , 21 , 32 , 247 , 62 , 4 , 205
10050  DATA 90 , 187 , 62 , 2 , 205 , 90 , 187 , 221 , 33 , 145
10060  DATA 78 , 22 , 8 , 6 , 1 , 14 , 1 , 205 , 131 , 78
10070  DATA 221 , 126 , 0 , 221 , 35 , 254 , 255 , 40 , 5 , 205
10080  DATA 90 , 187 , 24 , 242 , 121 , 230 , 7 , 40 , 3 , 12
10090  DATA 24 , 248 , 12 , 21 , 32 , 227 , 195 , 195 , 78 , 62
10100  DATA 31 , 205 , 90 , 187 , 121 , 205 , 90 , 187 , 120 , 205
10110  DATA 90 , 187 , 201 , 83 , 80 , 65 , 82 , 82 , 79 , 87
10120  DATA 255 , 79 , 87 , 76 , 83 , 255 , 87 , 82 , 69 , 78
10130  DATA 83 , 255 , 67 , 82 , 79 , 87 , 83 , 255 , 84 , 73
10140  DATA 84 , 83 , 255 , 67 , 85 , 82 , 76 , 69 , 87 , 255
10150  DATA 82 , 79 , 66 , 73 , 78 , 83 , 255 , 70 , 73 , 78
10160  DATA 67 , 72 , 255 , 205 , 9 , 187 , 48 , 8 , 254 , 35
10170  DATA 202 , 73 , 79 , 254 , 248 , 200 , 1 , 245 , 248 , 237
10180  DATA 88 , 40 , 236 , 22 , 128 , 6 , 8 , 123 , 162 , 32
10190  DATA 6 , 203 , 42 , 16 , 248 , 24 , 222 , 88 , 1 , 245
10200  DATA 248 , 237 , 120 , 162 , 32 , 251 , 6 , 128 , 16 , 254
10210  DATA 29 , 195 , 246 , 78 , 38 , 0 , 203 , 20 , 33 , 33
10220  DATA 78 , 203 , 19 , 125 , 131 , 111 , 126 , 60 , 39 , 119
10230  DATA 48 , 6 , 63 , 35 , 126 , 60 , 39 , 119 , 0 , 33
10240  DATA 34 , 78 , 6 , 2 , 14 , 1 , 205 , 131 , 78 , 22
10250  DATA 8 , 62 , 48 , 237 , 111 , 205 , 90 , 187 , 237 , 111
```

```
10260  DATA 205 , 90 , 187 , 237 , 111 , 43 , 237 , 111 , 205 , 90
10270  DATA 187 , 237 , 111 , 205 , 90 , 187 , 237 , 111 , 125 , 198
10280  DATA 3 , 111 , 6 , 4 , 62 , 9 , 205 , 90 , 187 , 16
10290  DATA 251 , 21 , 32 , 213 , 195 , 195 , 78 , 33 , 167 , 0
10300  DATA 62 , 50 , 205 , 104 , 188 , 38 , 0 , 203 , 20 , 33
10310  DATA 32 ; 78 , 62 , 48 , 237 , 111 , 50 , 177 , 79 , 237
10320  DATA 111 , 50 , 178 , 79 , 237 , 111 , 33 , 163 , 79 , 6
10330  DATA 16 , 17 , 0 , 0 , 205 , 140 , 188 , 218 , 127 , 79
10340  DATA 62 , 7 , 205 , 90 , 187 , 205 , 143 , 188 , 195 , 49
10350  DATA 78 , 33 , 32 , 78 , 17 , 16 , 0 , 1 , 0 , 252
10360  DATA 62 , 2 , 205 , 152 , 188 , 205 , 143 , 188 , 253 , 33
10370  DATA 32 , 78 , 38 , 0 , 203 , 20 , 253 , 126 , 0 , 60
10380  DATA 39 , 253 , 119 , 0 , 195 , 64 , 78 , 68 , 65 , 84
10390  DATA 65 , 32 , 76 , 79 , 71 , 32 , 80 , 65 , 83 , 83
10400  DATA 32 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
```

Fig. 5.18

A useful program written in assembly language

On a CPC 464 there is no way provided (That I can find) to get a hard copy of a tape catalogue. The second assembly language program which we are going to look at (In less detail then the first, since you should now be getting the hang of assembler) is a program to give you this facility. The program has an optional BASIC front end to it, and the assembly and HEXBAS versions will be listed.

In order to understand the operation of the CATS PRINTER program you must first know about how cassette files are stored. Exhaustive information on this subject is given in SOFT 158 (see section 5.4). Every block of data in a cassette file has a header. This is a group of 64 stored bytes which describe the file. These bytes are used as follows:

BYTE 0–15	Filename
BYTE 16	Block number
BYTE 17	If this is not zero then the block which follows the header is the last block of the file.
BYTE 18	File type byte. The bits in this byte are assigned as follows: Bit 0 If set the file is protected. Bit 1–3 File type 000 = Internal BASIC 001 = Binary 010 = Screen image 011 = ASCII file 100– 111 are unallocated types.
BYTES 19–20	Specify the length of the data block
BYTES 21–22	Data location – Where the file was written from.
BYTE 23	If this is not zero this is the first block.
BYTES 24–25	Length of the file in bytes.
BYTES 26–27	Execution address of file if machine code.
BYTES 28–63	Undefined, for you to use.

Now armed with this information we can go ahead and look at the CATS PRINTER program. The BASIC version of this, with integral HEXBAS section, is given in figure 5.19. The BASIC section just asks you for the date, the cassette name, and which side of the cassette you are cataloguing. These items of information are then printed as a header above the files listing.

```
1000 '                   ** CATS PRINTER **
        This program outputs a catalogue of files on a tape to the printer.
        It is the front end of a small machine code program which it
1100 '  POKEs into memory at 25000. Please note the following points;
        unlike the BASIC CAT command the data in the files is not read,
        only the headers are read. The program cannot be co-resident
1200 '  with any other BASIC program, unpredictable results will
        occur if you try. The cassette buffer is set by the machine
        code section, to begin at 40000.
1210    ON BREAK GOSUB 1600
1300    CLS: RESTORE: MEMORY 24999: FOR I=25000 TO 25315: READ P:
        POKE I,P: NEXT I: INPUT "Please type todays date";DOY$
        'put the machine code section in memory. Set date
1305    INPUT "Please type the name of this tape";D$:
        IF D$="" THEN D$="Unknown tape"
1400    INPUT "Which side are you listing the files for";S$:
        IF S$="" THEN S$=" unspecified"
1500    PRINT #8: PRINT #8,,"Catalogue for tape - ";D$;
        ": SIDE ";S$;" ";DOY$: PRINT #8: CALL 25000: PRINT #8
        'Print the tape name,side number and date as a header.
1600    PRINT "Catalogue completed" : PRINT #8,STRING$(25,45):
        'Restart when user has typed escape, after tape has
         been catted.
1610    PRINT : INPUT "Do you want to catalogue any more tapes";R$:
        IF UPPER$(LEFT$(R$,1))="Y" THEN GOTO 1305 ELSE GOTO 32767
10000  DATA 33 , 64 , 156 , 17 , 64 , 0 , 62 , 44 , 63 , 205
10010  DATA 161 , 188 , 56 , 8 , 205 , 153 , 98 , 254 , 0 , 32
10020  DATA 235 , 201 , 17 , 64 , 156 , 38 , 16 , 26 , 254 , 0
10030  DATA 32 , 2 , 62 , 32 , 205 , 207 , 98 , 19 , 37 , 32
10040  DATA 242 , 62 , 5 , 205 , 153 , 98 , 26 , 39 , 103 , 203
10050  DATA 63 , 203 , 63 , 203 , 63 , 203 , 63 , 246 , 48 , 205
10060  DATA 207 , 98 , 124 , 230 , 15 , 246 , 48 , 205 , 207 , 98
10070  DATA 19 , 19 , 62 , 48 , 205 , 153 , 98 , 26 , 230 , 7
10080  DATA 246 , 48 , 205 , 207 , 98 , 62 , 64 , 205 , 153 , 98
10090  DATA 24 , 164 , 0 , 13 , 10 , 10 , 67 , 97 , 116 , 97
10100  DATA 108 , 111 , 103 , 117 , 101 , 32 , 97 , 98 , 111 , 114
10110  DATA 116 , 101 , 100 , 32 , 98 , 121 , 32 , 42 , 42 , 32
10120  DATA 66 , 82 , 69 , 65 , 75 , 32 , 42 , 42 , 255 , 1
10130  DATA 0 , 42 , 42 , 32 , 84 , 97 , 112 , 101 , 32 , 101
10140  DATA 114 , 114 , 111 , 114 , 32 , 45 , 32 , 98 , 105 , 116
10150  DATA 32 , 116 , 111 , 111 , 32 , 108 , 111 , 110 , 103 , 32
10160  DATA 42 , 42 , 255 , 2 , 42 , 42 , 32 , 84 , 97 , 112
10170  DATA 101 , 32 , 101 , 114 , 114 , 111 , 114 , 32 , 45 , 32
10180  DATA 99 , 104 , 101 , 99 , 107 , 115 , 117 , 109 , 32 , 101
10190  DATA 114 , 114 , 111 , 114 , 32 , 42 , 42 , 255 , 5 , 32
10200  DATA 66 , 108 , 111 , 99 , 107 , 32 , 110 , 117 , 109 , 98
10210  DATA 101 , 114 , 32 , 255 , 64 , 32 , 67 , 111 , 114 , 114
10220  DATA 101 , 99 , 116 , 32 , 10 , 13 , 255 , 48 , 32 , 70
10230  DATA 105 , 108 , 101 , 32 , 116 , 121 , 112 , 101 , 61 , 255
10240  DATA 0 , 253 , 229 , 213 , 245 , 253 , 33 , 4 , 98 , 253
10250  DATA 86 , 0 , 253 , 35 , 186 , 40 , 19 , 253 , 35 , 253
10260  DATA 203 , 0 , 126 , 40 , 248 , 253 , 35 , 253 , 86 , 0
10270  DATA 254 , 0 , 40 , 227 , 24 , 229 , 253 , 126 , 0 , 205
10280  DATA 207 , 98 , 253 , 35 , 253 , 203 , 0 , 126 , 40 , 242
10290  DATA 241 , 209 , 253 , 225 , 201 , 197 , 6 , 245 , 237 , 72
10300  DATA 203 , 113 , 32 , 250 , 6 , 239 , 203 , 255 , 237 , 121
10310  DATA 203 , 191 , 237 , 121 , 193 , 201 , 0 , 0 , 0 , 0
32767    END
```

Fig. 5.19

The operation of the BASIC section is self explanatory. The machine code which it POKEs into memory at address 25000 is derived from the assembly language program listed in figure 5.20.

```
 10 ;              CAT PRINTER.
 20 ;  outputs a tape catalogue to the printer.
 40 ; This program calls the CAS READ firmware routine
 50 ; after every call it interrogates the header information
 60 ; loaded from the tape, and prints out the information
 70 ; contained in it as a catalogue.
 80          ORG  25000
 90          ENT  $
100 CATP0:   LD   HL,40000 ; Set cassette buffer at 40000 (D)
110          LD   DE,64 ; read 64 bytes please
120          LD   A,#2C ; We want the header sync character
130          CCF  ; Clear carry
140          CALL #BCA1 ; GO do CASRD s/r.(c set on ret if ok)
150          JR   C,READOK ; If all is well we jump to READOK
160          CALL MESOUT ; If not then call message O/P rtne
170          CP   #0 ; See if message number was zero
180          JR   NZ,CATP0 ; If no try again. Else user typed
190          RET  ; Break do go back to BASIC.
200 READOK:  LD   DE,40000 ; If read was OK then we must now
210 ;             print the filename.
220 ;             DE points to header
230          LD   H,16 ; sixteen chars in filename
240 CATP4:   LD   A,(DE) ; get a byte
250          CP   #0 ; See if its a null
260          JR   NZ,CATP6  ; Jump if not
270          LD   A,#20 ; Make it a space if its a nul
280 CATP6:   CALL PRINT  ; go PRINT this character
290          INC  DE ; Increment pointer
300          DEC  H ; decrement H - loop count
310          JR   NZ,CATP4
320 ; Having output the filename we must O/P the block number
330 ; which IY now points at.
340 SHOBLK:  LD   A,05 ; make A=code for message "Block number""
350          CALL MESOUT ; CALL it
360          LD   A,(DE) ; Get the block number
370          DAA  ; set to BCD
380          LD   H,A ; Copy result to H
390          SRL  A ; Make high digit low digit
400          SRL  A
410          SRL  A
420          SRL  A ; done it!
430          OR   #30 ; make it ASCII
440          CALL PRINT  ; PRINT it
450          LD   A,H ; get saved H
460          AND  #0F ; extract low digit
470          OR   #30 ; make it ASCII
480          CALL PRINT  ; PRINT it
490          INC  DE ; step DE on to point at filetype
500          INC  DE ; which is 2 bytes on
510 ;             Now the filetype must be indicated
520 ;             Do this by changing the upper four bits
530 ;             of filetype byte to be 0011. This converts
540 ;             it to an ASCII number, ready for MESOUT.
550 SHOTYP:  LD   A,#30 ; Load MESNUM for "Filetype=" message
560          CALL MESOUT ; go print it
570          LD   A,(DE) ; get the filetype byte
580          AND  #07 ; zero the top five bits
590          OR   #30 ; convert bottom four to a mesnum
600          CALL PRINT  ; output filtyp as a number
610 HOORAY:  LD   A,#40 ; Get MESNUM for "Correct"CR/LF message.
620          CALL MESOUT ; print it.
630          JR   CATP0 ; and begin again
```

```
640 ;
650 ;
660 ;
670 ;              MESTAB contains message to be printed
680 ; by the subroutine MESOUT. the form is:  message number;
690 ; Message text: HEX FF: MESNUM: until MESNUM=0 whence ends
700 MESTAB: DEFB #00,#0D,#0A,#0A
710         DEFM "Catalogue aborted by ** BREAK **"
720         DEFB #FF,#01,#00
730         DEFM "** Tape error - bit too long **"
740         DEFB #FF,#02
750         DEFM "** Tape error - checksum error **"
760         DEFB #FF,#05
770         DEFM " Block number "
780         DEFB #FF,#40
790         DEFM " Correct "
800         DEFB #0A,#0D
810         DEFB #FF,#30
820         DEFM " File type=",#FF
830         DEFB #FF,#00  ; End of the message table here
840 ;
850 ;
860 ;       MESOUT: outputs the error message whos
870 ;               code it finds in A on entry.
880 ;               The MESTAB table must be added before use
890 ;               The label "OUTDEV:" must be for an output
900 ;               device driver, eg printer, screen, or I/O
910 ; we look it up in the message table MESTAB
920 ; this has the format; MESNUM,TEXT,#FF. a nul byte
930 ; ends the table.
940 ; on entry A=message number to be printed.
950 ; all regs are unmolested on return
960 MESOUT: PUSH IY ; save IY
970         PUSH DE ; save DE entry contents
980         PUSH AF ; save AF
990 MES001: LD   IY,MESTAB ; make IY pointer to MESTAB
1000 MES002: LD   D,(IY) ; Load MESNUM to D
1010         INC  IY ; Bump IY over the MESNUM
1020         CP   D ; see if its the right message
1030         JR   Z,OUTMES ; jump if so.
1040 MES003: INC  IY ; Else step IT through table.
1050         BIT  7,(IY) ; See if bit 7 set
1060         JR   Z,MES003 ; Jump if not
1070         INC  IY ; If do look at byte following
1080         LD   D,(IY)
1090         CP   #0 ; See if its a nul
1100         JR   Z,MES001 ; If so then lookup failed
1110 ; so output the BREAK message and hope for best.
1120         JR   MES002 ; If no see if this mesnum OK
1130 OUTMES: LD   A,(IY) ; Get a character
1140         CALL PRINT  ; PRINT it
1150         INC  IY ; INC table pointer
1160         BIT  7,(IY) ; See if done yet
1170         JR   Z,OUTMES ; If not then jump
1180         POP  AF
1190         POP  DE
1200         POP  IY
1210 GOBACK: RET  ; clean stack and return to caller
1220 ;
1230 ;
1240 ;             PRINT
1250 ;       prints the character in the A reg
1260 ;    NOTE * ROUTINE HANGS IF PRINTER NEVER COMES READY *
1270 PRINT:  PUSH BC ; Preserve BC
1280         LD   B,#F5 ; look at printer ready bit -8255 port B
1290 WAIT:   IN   C,(C) ; get stat
1300         BIT  6,C
1310         JR   NZ,WAIT ; If bit 6=1 printer not ready so loop
1320         LD   B,#EF ; If rdy make BC point at data re
```

```
1330         SET  7,A ; Set strobe bit high on 7 bit char
1340         OUT  (C),A
1350         RES  7,A ; output it, then reset strobe bit
1360         OUT  (C),A ; reoutout with strobe=0
1370         POP  BC ; restore BC register
1380         RET  ; and goodbye
```

Fig. 5.20

The assembler generated part of the program reads all the tape headers it encounters on a playing cassette, and then from the information contained in the headers it prints a one line summary of the information on the printer. Figure 5.21 shows the format of the printout produced by the program.

```
              Catalogue for tape - DEMO: SIDE 1 21-March-1985

CAT PRINTER.EXE   Block number 01 File type=6 Correct
CAT PRINTER.EXE   Block number 02 File type=6 Correct
SPEECH DEMO AUTO  Block number 01 File type=6 Correct
SPEECH DEMO AUTO  Block number 02 File type=6 Correct
SPEECH DEMO MAN   Block number 01 File type=6 Correct
KEY MAT DEMO      Block number 01 File type=1 Correct
KEY MAT DEMO      Block number 02 File type=1 Correct
SERIAL XMIT DEMO  Block number 01 File type=1 Correct
SERIAL XMIT DEMO  Block number 02 File type=1 Correct
MEMORY DUMPER     Block number 01 File type=6 Correct
MINEFIELD !       Block number 01 File type=6 Correct
MINEFIELD !       Block number 02 File type=6 Correct
MINEFIELD !       Block number 03 File type=6 Correct
MINEFIELD !       Block number 04 File type=6 Correct

Catalogue aborted by ** BREAK **------------------------
```

Fig. 5.21: Sample printout from CATS PRINTER program

You could very easily redirect the output of the program to the screen, to a serial port, or any other device (except the cassette) by substituting your own version of the PRINT subroutine. For example if you wanted the output of the program to go to the CPC screen then the PRINT subroutine would be changed to read:

```
1270 PRINT: CALL #BB5A ; Print the char in A
1280        RET        ; and then RETurn to caller.
```

and you can easily patch in your own requirements. The PRINT routine shown makes very hard work of outputting characters to the printer, but dear reader, this is for your benefit! The idea being to show how to drive the printer port directly now that you know how it works. The routine shown could be replaced directly with:

```
1270 PRINT: CALL #BD31 ; Call firmware rtne to print char in A
1280        RET        ; and RETurn
```

This change has not been made to the listed program. As discussed in an earlier chapter the I/O address Hex F5xx (where xx is any value) is the address for the printer port. Bit seven of this has to be pulsed low whilst the code for the character to be printed is held steady on bits 0–6.

Now let us go through the assembler section of CATS PRINTER, routine by routine. At the start `CATP0` sets up the cassette buffer address, and the expected sync character (that is, what the cassette firmware should expect to see as the lead–in to the header). Next it calls a routine at Hex address BCA1, which reads the header into the buffer at decimal memory address 40000. On returning from this routine the carry flag is set if the header was loaded error free. If any error occurred then the `MESOUT` routine is called into action. If the error was that the user typed ESC then a return to BASIC is effected. If all is well then DE is used as a pointer to the loaded header, and first the filename is extracted and printed. Unused bytes in the filename are stored as zeroes (NULs). To preserve a good layout all encountered NULs are printed as spaces. Next the block number is converted to ASCII and printed.

The next thing to be extracted from the loaded header is the filetype. This is printed as a number where:

0 = Internal BASIC
2 = Binary
4 = Screen image
6 = ASCII format file

Finally, the word `"Correct"` is printed to show that the header was loaded error free. It is important to realise the limitations of this program. It only reads headers, the data blocks which they describe are not checked. This is because the program was conceived as a way to get a hard copy of the contents of a tape, not as a data verification utility, the inbuilt `CAT` command will do this already.

The two subroutines `MESOUT` and `PRINT` are mostly explained by the remarks in the listing. The format of the message table `MESTAB`, which contains all the messages which the program may need to issue, is as follows:

```
Message number | Text | #FF | Mesnum | Text | #FF etc
```

The table ends when the byte following an #FF byte is zero. `MESOUT` searches `MESTAB` until it finds the message number matching the one it has been sent in the A register. The PRINT routine has already been discussed, and as previously mentioned it is easy to replace it with a call to another routine.

Author confession – shock horror probe

In the interests of backing my earlier assertion that you should always flowchart an assembler program before writing it, let me come clean. The

CATS PRINTER program works. It is not an elegant program however, and the reason for this is that – you guessed it – I wrote it at the keyboard. This was due to a misguided feeling on my part that it would be finished more quickly. In the event it took me about twice as long. Once the length gets past a certain point, you have to keep printing the program off, then making major corrections and trying again. From now on I shall practise what I preach, Flowcharts rule OK. Figure 5.22 is the CATS PRINTER assembly listing.

```
                  100 ;             CAT PRINTER.
                  110 ;  outputs a tape catalogue to the printer.
                  120 ; This program calls the CAS READ firmware routine
                  130 ; after every call it interrogates the header information
                  140 ; loaded from the tape, and prints out the information
                  150 ; contained in it as a catalogue.
61A8              160           ORG  25000
61A8              170           ENT  $
61A8  21409C      180 CATP0     LD   HL,40000 ; Set cassette buffer at 40000 (D)
61AB  114000      190           LD   DE,64 ; read 64 bytes please
61AE  3E2C        200           LD   A,#2C ; We want the header sync character
61B0  3F          210           CCF  ; Clear carry
61B1  CDA1BC      220           CALL #BCA1 ; GO do CASRD s/r.(c set on ret if ok)
61B4  3808        230           JR   C,READOK ; If all is well we jump to READOK
61B6  CD9962      240           CALL MESOUT ; If not then call message O/P rtne
61B9  FE00        250           CP   #0 ; See if message number was zero
61BB  20EB        260           JR   NZ,CATP0 ; If no try again. Else user typed
61BD  C9          270           RET  ; Break do go back to BASIC.
61BE  11409C      280 READOK    LD   DE,40000 ; If read was OK then we must now
                  290 ;              print the filename.
                  300 ;              DE points to header
61C1  2610        310           LD   H,16 ; sixteen chars in filename
61C3  1A          320 CATP4     LD   A,(DE) ; get a byte
61C4  FE00        330           CP   #0 ; See if its a null
61C6  2002        340           JR   NZ,CATP6  ; Jump if not
61C8  3E20        350           LD   A,#20 ; Make it a space if its a nul
61CA  CDCF62      360 CATP6     CALL PRINT  ; go PRINT this character
61CD  13          370           INC  DE ; Increment pointer
61CE  25          380           DEC  H ; decrement H - loop count
61CF  20F2        390           JR   NZ,CATP4
                  400 ; Having output the filename we must O/P the block number
                  410 ; which IY now points at.
61D1  3E05        420 SHOBLK    LD   A,05 ; make A=code for message "Block number""
61D3  CD9962      430           CALL MESOUT ; CALL it
61D6  1A          440           LD   A,(DE) ; Get the block number
61D7  27          450           DAA  ; set to BCD
61D8  67          460           LD   H,A ; Copy result to H
61D9  CB3F        470           SRL  A ; Make high digit low digit
61DB  CB3F        480           SRL  A
61DD  CB3F        490           SRL  A
61DF  CB3F        500           SRL  A ; done it!
61E1  F630        510           OR   #30 ; make it ASCII
61E3  CDCF62      520           CALL PRINT  ; PRINT it
61E6  7C          530           LD   A,H ; get saved H
61E7  E60F        540           AND  #0F ; extract low digit
61E9  F630        550           OR   #30 ; make it ASCII
61EB  CDCF62      560           CALL PRINT  ; PRINT it
61EE  13          570           INC  DE ; step DE on to point at filetype
61EF  13          580           INC  DE ; which is 2 bytes on
                  590 ;              Now the filetype must be indicated
                  600 ;              Do this by changing the upper four bits
                  610 ;              of filetype byte to be 0011. This converts
                  620 ;              it to an ASCII number, ready for MESOUT.
61F0  3E30        630 SHOTYP    LD   A,#30 ; Load MESNUM for "Filetype=" message
61F2  CD9962      640           CALL MESOUT ; go print it
61F5  1A          650           LD   A,(DE) ; get the filetype byte
```

```
61F6   E607       660        AND  #07 ; Zero the top five bits
61F8   F630       670        OR   #30 ; convert bottom four to a mesnum
61FA   CDCF62     680        CALL PRINT  ; output filtyp as a number
61FD   3E40       690 HOORAY LD   A,#40 ; Get MESNUM for "Correct"CR/LF message.
61FF   CD9962     700        CALL MESOUT ; print it.
6202   18A4       710        JR   CATP0 ; and begin again
                  720 ;
                  730 ;
                  740 ;
                  750 ;             MESTAB contains message to be printed
                  760 ; by the subroutine MESOUT. the form is:  message number;
                  770 ; Message text: HEX FF: MESNUM: until MESNUM=0 whence ends
6204   000D0A0A   780 MESTAB DEFB #00,#0D,#0A,#0A
6208   43617461   790        DEFM "Catalogue aborted by ** BREAK **"
6228   FF0100     800        DEFB #FF,#01,#00
622B   2A2A2054   810        DEFM "** Tape error - bit too long **"
624A   FF02       820        DEFB #FF,#02
624C   2A2A2054   830        DEFM "** Tape error - checksum error **"
626D   FF05       840        DEFB #FF,#05
626F   20426C6F   850        DEFM " Block number "
627D   FF40       860        DEFB #FF,#40
627F   20436F72   870        DEFM " Correct "
6288   0A0D       880        DEFB #0A,#0D
628A   FF30       890        DEFB #FF,#30
628C   2046696C   900        DEFM " File type=",#FF
6297   FF00       910        DEFB #FF,#00  ; End of the message table here
                  920 ;
                  930 ;
                  940 ;         MESOUT: outputs the error message whos
                  950 ;                code it finds in A on entry.
                  960 ;                The MESTAB table must be added before use
                  970 ;                The label "OUTDEV:" must be for an output
                  980 ;                device driver, eg printer, screen, or I/O
                  990 ; we look it up in the message table MESTAB
                 1000 ; this has the format; MESNUM,TEXT,#FF. a nul byte
                 1010 ; ends the table.
                 1020 ; on entry A=message number to be printed.
                 1030 ; all regs are unmolested on return
6299   FDE5      1040 MESOUT PUSH IY ; save IY
629B   D5        1050        PUSH DE ; save DE entry contents
629C   F5        1060        PUSH AF ; save AF
629D   FD210462  1070 MES001 LD   IY,MESTAB ; make IY pointer to MESTAB
62A1   FD5600    1080 MES002 LD   D,(IY) ; Load MESNUM to D
62A4   FD23      1090        INC  IY ; Bump IY over the MESNUM
62A6   BA        1100        CP   D ; see if its the right message
62A7   2813      1110        JR   Z,OUTMES ; jump if so.
62A9   FD23      1120 MES003 INC  IY ; Else step IT through table.
62AB   FDCB007E  1130        BIT  7,(IY) ; See if bit 7 set
62AF   28F8      1140        JR   Z,MES003 ; Jump if not
62B1   FD23      1150        INC  IY ; If do look at byte following
62B3   FD5600    1160        LD   D,(IY)
62B6   FE00      1170        CP   #0 ; See if its a nul
62B8   28E3      1180        JR   Z,MES001 ; If so then lookup failed
                 1190 ; so output the BREAK message and hope for best.
62BA   18E5      1200        JR   MES002 ; If no see if this mesnum OK
62BC   FD7E00    1210 OUTMES LD   A,(IY) ; Get a character
62BF   CDCF62    1220        CALL PRINT  ; PRINT it
62C2   FD23      1230        INC  IY ; INC table pointer
62C4   FDCB007E  1240        BIT  7,(IY) ; See if done yet
62C8   28F2      1250        JR   Z,OUTMES ; If not then jump
62CA   F1        1260        POP  AF
62CB   D1        1270        POP  DE
62CC   FDE1      1280        POP  IY
62CE   C9        1290 GOBACK RET  ; clean stack and return to caller
                 1300 ;
                 1310 ;
                 1320 ;            PRINT
                 1330 ;        prints the character in the A reg
                 1340 ;    NOTE * ROUTINE HANGS IF PRINTER NEVER COMES READY *
```

```
62CF  C5        1350 PRINT  PUSH BC ; Preserve BC
62D0  06F5      1360        LD   B,#F5 ; look at printer ready bit -8255 port B
62D2  ED48      1370 WAIT   IN   C,(C) ; get stat
62D4  CB71      1380        BIT  6,C
62D6  20FA      1390        JR   NZ,WAIT ; If bit 6=1 printer not ready so loop
62D8  06EF      1400        LD   B,#EF ; If rdy make BC point at data re
62DA  CBFF      1410        SET  7,A ; Set strobe bit high on 7 bit char
62DC  ED79      1420        OUT  (C),A
62DE  CBBF      1430        RES  7,A ; output it, then reset strobe bit
62E0  ED79      1440        OUT  (C),A ; reoutout with strobe=0
62E2  C1        1450        POP  BC ; restore BC register
62E3  C9        1460        RET  ; and goodbye

Pass 2 errors: 00

CATP0  61A8  CATP4  61C3  CATP6  61CA  GOBACK 62CE
HOORAY 61FD  MES001 629D  MES002 62A1  MES003 62A9
MESOUT 6299  MESTAB 6204  OUTMES 62BC  PRINT  62CF
READOK 61BE  SHOBLK 61D1  SHOTYP 61F0  WAIT   62D2


Table used:   215  from   685
Executes: 25000
```

Fig. 5.22

Do not use these instructions

There are several instruction types from the Z80A set which it is very unwise to use in the context of the CPC machines. These are mainly to do with the interrupts and the alternate register set.

As we saw in chapter four the interrupt which goes into the Z80A is used to signal that it is time to carry out a number of actions, including scanning the keyboard. If interrupts were to be disabled by an assembly language program using the `DI` (REF 30) instruction and not re–enabled before that program terminated, then no keyboard commands could be typed. Therefore, the recommendation is that unless you have no other option, avoid using the DI instruction.

Another group of instructions which should be avoided in a CPC context are the ones which affect the alternate register set. An example of this is the `EX AF,AF'` (REF 33) instruction. The CPC firmware uses the alternate register set to keep its various interrupt routine counters in, as well as pointers to I/O space etc. The firmware is just as likely to crash if you interfere with the alternate register set. (REF 38– EXX should also never be used).

The comments in the above two paragraphs will not apply if you are developing software to run on an eventual stand alone application. In such a case there will be a separate hardware configuration built, into which EPROMs blown on the EPROM programmer presented in chapter three will be plugged. So long as you realise that the sections of program containing

these instructions may not execute as expected on the CPC, you may decide to use the above instructions in software developed for use outside the CPC machine.

Another point about using the CPC as a test bed for eventual stand alone project software development, is that you have to be extremely careful that you designate I/O addresses for the dedicated project I/O which are compatible with gaps in the I/O address usage in the CPC. If you don't you won't be able to use the CPC to debug your programs on. By careful reference to the I/O address table given in SOFT 158 you will soon realise how the I/O addressing works, and be able to slot in your own hardware to the unused address ranges.

The MONA3 debug program

When you buy DEVPAC you are actually getting two programs. The GENA3 assembler and the MONA3 debug program. MONA3 is very useful since amongst other things it allows you to single step through your program, with a display of the contents of all the CPU registers after every step. The only limitation for single stepping is that if you use the CALL instruction to call firmware routines within your program then you may find that these firmware routines contain instructions like EXX (see previous section). This little problem is easily overcome by the insertion of breakpoints. A breakpoint is inserted using MONA3, and allows you to execute your program through to a certain point in it, after which a return to MONA3s register display will be made. This gives you the power to execute a program to a problem point and see what the registers contain at the trouble spot.

MONA3 also has a disassemble feature. That is to say that if you load some machine code into memory, and then tell MONA3 where it is then MONA3 can produce an assembly language type listing of the machine code. Don't think that it will be easy to break into protected software this way, becuase it is pretty easy for commercial software to be protected against disassembly! You might manage it eventually, but it probably won't be worth the effort.

MONA3 is a very great aid to program development, and has a great many other features too numerous to explain here. You are advised to obtain full details from Amstrad, or from the MONA3 manual.

General points about DEVPAC

I have encountered only a couple of problems with DEVPAC. I pass these on to you so that if you get similar problems you will not think you are doing something wrong when in fact you are not. I have experienced one of these bugs on two separate machines so I don't think that it is due to my hardware.

If during an assembly which is being sent to the printer you type CTRL/C a couple of times the text file (as shown in response to an "L" command) sometimes becomes corrupted.

The DEVPAC assembler seems to assume that you are using continuous stationery in a printer, which is a pain if you are not. Also on the subject of the printer, some printers convert the # (hash) character to a pound sign. This might cause confusion if you are not aware of it.

Conclusions

Assembly language programming is great fun, but also very irritating. You can lose the program you are working on very easily, because the said program crashes the computer, and you have to power it off and on again to regain control. For this reason for anything other than minor alterations it is a good idea to save your program before attempting to execute it.

Once you get used to using it, DEVPAC is a very capable and powerful program development tool, its debug module can give you great help in ironing out those awkward little problems which even the most careful development of programs cannot always prevent.

When you have become experienced at BASIC, try some assembler, you will get far "nearer" to the machine, and if you can use either programming method you will find that many combinational programs can be written which will perform better than just BASIC ones. Also having done a little assembly language programming will stand you in good stead for your future computing activities.

CHAPTER SIX

An advanced project

In chapter three we looked at some hardware add on projects which you could build to increase the usefulness of your CPC computer. In this chapter we shall look at a somewhat more complex application for one of them.

Local network project

The term "network" has been liberally applied in the last few years to encompass everything up to intercontinental computer links with all the frills. The subject matter of this chapter describes CPCNET, which is a low cost, manually operated network.

This chapter describes a way to use the parallel transfer channel hardware described in chapter three (project six) to implement a processor interconnect to allow you to connect two CPC computers to one another, and exchange data or programs between them at high speed.

The CPCNET has been developed for use in places where a number of CPC computers are used, and large programs or large amounts of data need to be exchanged between them. In this scheme you simply connect a suitable cable (detailed later) between them, and use the CPCNET commands to start the transfer. This will be of great value where cassette and disk based machines are in use side by side. With no common media, file transfer can be a headache. Another use is in effecting the transfer of large programs or bodies of data to other makes of machine, when the high speed attainable with parallel transfer greatly reduces the time required when compared with serial transmission methods.

CPCNET has the following design objectives:

1) As low cost as possible.
2) Transfer of text, ASCII versions of programs or binary data to be possible between the two machines.

The CPCNET software will be contained in an EPROM. You will need the expansion ROMs board fitted to your machine (see project ten in chapter three), but you may not necessarily have to have an EPROM programmer, since it is hoped that ready blown EPROMs may be available from the software supplier – see the start of this book.

CPCNET facilities and limitations – in detail

The new commands available after fitting the CPCNET ROM are:

`|NET.OPEN` This patches the CPC jumpblocks so that issuing a LIST #8 command from BASIC will send the ASCII characters comprising the program out over the network rather than to the printer, having first sent a header (see later).

`|NET.CLOSE` This command restores the jumpblock to normal so that a LIST #8 command will again list the file out on the printer.

`|NET.SEND,START,END` This command sends the contents of the memory addresses in the range specified by the start and end addresses to the receiving machine.

`|NET.REC` This routine waits for data reception to begin. Then the program examines the header (see later) and if the incoming data is anything other than an ASCII version of a BASIC program it is loaded into memory starting at the address indicated in the header. Note that this will occur irrespective of the memory limits currently set by BASIC. If the incoming data is found to be an ASCII version of a BASIC program then it is fed into BASIC. (See description of how this is done in the software notes on the ROMdisk project in chapter three – project nine.) After the load has completed any modifications to jumpblocks are undone.

Let us look at the limitations of CPCNET.

Limitations

In a cut price, multi–compromise networking system, limitations are unavoidable. The minimal system presented here relies on the user making the connections manually, by plugging in the two cable ends to the sending and receiving systems. Strictly speaking, this is not networking but inter–processor linkage. However, because you can transfer BASIC files across the link between ant two CPC machines in an office or educational environment simply by plugging the linking cable into the sending and receiving machines, I think that it just qualifies as a manually operated network. If you wanted to develop your own black box network node subsystems, you could use this software and hardware as the basis of a fully blown multi node network.

Transmission headers

The CPCNET transmits the data as a single stream of data over the parallel link. A moments thought will reveal that the receiving machine must br told when the transmission is at an end. We cannot do this by seeing an end of file character, since in binary files the EOF character is a legal machine code, and may thus occur at any point. This creates the need to send a small number of

bytes, before the actual data, which describes what sort of data is about to be sent, and how much of it there is. This is called the header and it consists of 32 bytes. The header is layed out as follows:

Byte 1 : Data type. 0=Binary 1=ASCII BASIC
Byte 2 : Data start address (low byte)
Byte 3 . Data start address (Hi byte)
Byte 4 : Data end address (Low byte)
Byte 5 : Data end address (Hi byte)
Bytes 6–25: 16 byte filename. (see later)
Bytes 26–32: Reserved for future use.

The header bytes are held in a buffer called –reasonably enough– the header buffer.

Detailed description of the software in the CPCNET background ROM

These notes assume that you have a tolerably good knowledge of the contents of the firmware manual SOFT 158. If you do not then you should skip to the next section. In this section we shall look at the illustrations for the CPCNET software which should be blown into an EPROM and plugged into the expansion ROM board.

In several important ways the CPCNET software functions similarly to the ROMdisk software detailed in chapter three. They both modify jump block entries to gain access to BASIC, but CPCNET takes this a stage further and modifies the jumpblocks to allow stream eight in BASIC to be turned into an output stream to the CPCNET hardware. Both CPCNET and ROMdisk software live in background ROMs located on the expansion ROMs board. And, finally, both use a similar library of subroutines.

There are four main modules of the CPCNET software. These will now be described and listed, in the order they should be assembled. After all the modules and hardware connections have been described the assembly listing for the finished ROM will be given, along with a Hexdump.

1) **NETINIT:** This is the balnket name given to the initialisation routines for CPCNET. The listing for this section of the software is Fig. 6.1: The early lines deal with assigning names to addresses for the Parallel transfer channel registers (see later under "CPCNET Hardware"). Then comes the background ROM header table as specified in SOFT 158. And finally the NETGO routine – which is automatically called when the CPC firmware detects that the CPCNET ROM is present during the power up routines.

   NETGO reserves 256 bytes of memory for the exclusive use of CPCNET, the firmware will not use these now. It clears the activity flags. All the

major modules of the CPCNET firmware have assigned to them a flag bit in one of the flag bytes. On entry to a routine it sets its flag bit, and clears it down again when terminating. This was originally to help with trouble shooting the software, since if a routine completed in error it was possible to tell which routine was actually running when the premature ending occurred. The flag bytes will be of great use if you want to modify the software in any way. The final act of NETGO is to output a message to the screen to say that CPCNET has been initialised. This message should appear at every power on of the CPC when the expansion ROMs board is fitted. Having initialised NETGO returns to the firmware with the carry flag set to signal that all is well.

```
 30 ; **               CPCNET    ROM listing            **
 40 ;
 50 ;
 60 ; Manually operated intercomputer linking network for the
 70 ; Amstrad CPC computers. Kernow computers C 1985
 80 ;
 90 ;
100 PTCSND: EQU  #F8F0 ; Define the addresses for the PTC regs.
110 PTCREC: EQU  #F8F1
120 PTCCNT: EQU  #F8F2 ; The control signals for the channels
130 PPICTL: EQU  #F8F3 ; The control reg for the PTC's PPI chip
140 ROMNUM: EQU  5 ; The number of ROM socket we are to live in
150 ;
160 ;
170 ;
180         ORG  #C000
190 ;
200 ;
210         DEFB #1 ; BACKGROUND ROM
220         DEFB #1,#1,#1 ; Mark 1 Vers 1 rev 1
230         DEFW NAMTAB ; Set pointer to name table.
240         JP   NETGO  ; Power on init routine
250         JP   NETOPN
260         JP   NETCLOS
270         JP   NETSND
280         JP   NETREC
290 NAMTAB: DEFM "NET INI" ; Init entry
300         DEFB "T"+#80
310         DEFM "NET.OPE"
320         DEFB "N"+#80
330         DEFM "NET.CLOS"
340         DEFB "E"+#80
350         DEFM "NET.SEN"
360         DEFB "D"+#80
370         DEFM "NET.RE"
380         DEFB "C"+#80
390         DEFB 0 ; The end of name table marker.
400 ;
410 ;
420 ; ** NETGO: This routine initialises the network at power
430 ; ** on. It is limited to setting up the PPI and
440 ; ** creating a work area in RAM which the CPC will leave
450 ; ** alone.
460 NETGO:  PUSH AF
470         PUSH BC
480         DEC  H    ; HL contains upper limit of memory pool
490 ;                grab us 256 bytes for work area
500         LD   BC,PPICTL ; BC points at PPI control
510         LD   A,#AF    ; Mode byte is AF
520         OUT  (C),A    ; write it
530         LD   A,0
```

```
540          LD    (IY+#4E),A
550          LD    (IY+#4F),A ; Zero the flags bytes
560          PUSH  IX
570          LD    IX,INIMES  ; Tell the user its done
580          CALL  OUTPUT     ; by OUTPUTing a message
590          POP   IX
600          POP   BC
610          POP   AF
620          SCF
630          RET   ; Restore regs, set carry and RET.
640 INIMES:  DEFB  #0A,#0D
650          DEFM  "** CPCNET  V1.1 initialised **"
660          DEFB  #0A,#0D,#FF
670 ;
680 ;
690          JR    SNDEND
700 SUCESS:  DEFB  #0A,#0D
710          DEFM  "** Transfer completed **"
720          DEFB  #0A,#0D,#FF
730 SNDEND:  POP   BC
740          POP   IX
750          POP   HL
760          POP   DE
770          POP   AF
780          RET
790 ;
```

Fig. 6.1

2) The next module of the CPCNET software is called **NETOPEN**. This routine opens the Parallel transfer channel to BASIC on stream 8. The listing for NETOPEN is shown in Fig. 6.2. After issuing a `NET.OPEN` command you can send BASIC programs or variables out on the PTC using statements like "`LIST #8`" or "`PRINT #8,D$`". This is achieved by modifying the jumpblock entries which pertain to the printer. The original jumpblock entries are saved in the work area reserved by NETGO and they are replaced with entries which point instead to routines which send characters out on the PTC send channel.

```
10 ; ** NETOPN: This CPCNET routine opens the PTC to **
20 ; ** BASIC on stream 8. This allows text versions **
30 ; ** of CPC BASIC programs to be transferred over **
40 ; ** the CPCNET, using LIST #8  or PRINT #8 type   **
50 ; ** statements                                    **
60 ;
70 ;
80 NETOPN:  PUSH  AF
90          PUSH  HL
100         BIT   0,(IY+#4E) ; See if flags say jump blk already patched
110         JR    NZ,MAKHDR  ; so jump over patching section
120         SET   0,(IY+#4E) ; if we shall patch then flag set
130         SET   0,(IY+#4F) ; Set our active bit in func byte
140         LD    HL,(#BD2B) ; Get first two bytes
150         LD    (IY),L     ; Save first
160         LD    (IY+#1),H  ; Save second
170         LD    HL,(#BD2D) ; Get last two bytes from jmp block
180         LD    (IY+#2),L  ; Save third byte
190         LD    (IY+#3),H  ; And the final one
200         LD    A,#DF      ; Load the restart instruction
210         LD    (#BD2B),A  ; in place of previous contents
220         LD    HL,SEND    ; Now HL points to send routine.
230         LD    (IY+#10),L ; Place first byte
```

```
240            LD    (IY+#11),H ; And second
250            LD    H,ROMNUM   ; Now the ROM number where we live
260            LD    (IY+#12),H ; Place the ROM numb in far address
270            PUSH  IY         ; Again copy IY to HL
280            POP   HL
290            PUSH  DE         ; We need DE for a moment - save it.
300            LD    DE,#10
310            ADD   HL,DE      ; Calculate far address for jmp blk
320            POP   DE         ; Finished with DE so restore it
330            LD    (#BD2C),HL ; And place far address in jmp blk.
340            LD    A,#C9      ; Load a RET instruct into entry after
350            LD    (#BD2E),A  ; PRINT CHAR for the return.
360 MAKHDR:    LD    A,#1         ; Now construct the header buffer
370            LD    (IY+#50),A   ; Set data type to BASIC (01)
380            LD    A,0          ; Buffer address fields get zeroed
390            LD    (IY+#51),A
400            LD    (IY+#52),A
410            LD    (IY+#53),A
420            LD    (IY+#54),A   ; Now we must zero the data fields
430            PUSH  IY           ; Copy IY to HL again
440            POP   HL
450            LD    BC,#55       ; Make HL point to start of name field
460            ADD   HL,BC
470            LD    B,16         ; Sixteen chars to be zeroed
480 OPN009:    LD    (HL),A       ; Zero a byte
490            INC   HL           ; Bump pointer
500            DJNZ  OPN009       ; Loop until done
510            CALL  SNDHDR       ; Call the header send routine
520            LD    IX,OPEND     ; Now we must tell user all about it
530            CALL  OUTPUT
540            JP    OPNEND       ; and end up
550 OPEND:     DEFB  #0A,#0D
560            DEFM  "** CPCNET is now opened for you to send BASIC **"
570            DEFB  #0A,#0D
580            DEFM  "** programs or variables on stream 8. You can **"
590            DEFB  #0A,#0D
600            DEFM  "** use the LIST #8 or PRINT #8 statments to do this.    **"
610            DEFB  #0A,#0D
620            DEFM  "** when you have finished sending, issue a  |NET.CLOSE **"
630            DEFB  #0A,#0D
640            DEFM  "** command to terminate the transmission properly.      **"
650            DEFB  #0A,#0D,#FF
660 OPNEND:    POP   HL
670            POP   AF           ; Then weve done it so restore
680            RES   0,(IY+#4F)  ; reset our active flag
690            RET   ; and go back
```

Fig. 6.2

Because far addresses are used (see SOFT 158) it is essential that the ROMNUM assignment be correct for the ROM socket which you have plugged your CPCNET ROM into. So, for example, if it is plugged in to ROM socket 4 then ROMNUM must be altered from the value shown in figure 6.1, and the altered version blown into a new EPROM. The default ROMNUM should be OK for most systems.

Once the jumpblocks have been tinkered with the other action taken by NETOPEN is to set up a valid header in the header buffer which is located within the CPCNET work area. Note: On entry to all CPCNET routines the CPC firmware has previously set the IY register to the base address of the CPCNET work area. As with the ROMdisk specific bytes in the work area must be referred to as (IY+#n) where n is the number which must be used as

an offset to the entry value of IY to form the desired address.

When it has almost completed, NETOPEN prints up a short message to tell the user how to use the CPCNET send channel via #8. Like the other major routines NETOPEN sets its activity flag whilst it is running, but it also sets bit zero of (IY+#4E) to signal that it has modified the jumpblock entries for the printer. The NETCLOSE routine uses this flag later.

3) The **NETCLOSE** routine comes next. This performs the opposite task to the NETOPEN routine. It examines the flags set by NETOPEN and NETREC (see below) when they modify the jumpblocks. If either flag is set then the appropriate jumpblock entries are restored from the save area for the original contents. If the flags are not found to be set, then there would be no point in restoring the jumpblock contents so a straight return is performed by NETCLOSE. In any case NETCLOSE always sends a Hex FF byte to the PTC send channel, to indicate EOF to BASIC transmissions. Fig. 6.3 is a listing of NETCLOSE.

```
 10 ;
 20 ;
 30 ;
 40 ; ** NETCLO: This CPCNET routine closes the PTC channel
 50 ; ** to BASIC for output and input. It restores the
 60 ; ** jumpblock entries to their original values, so long
 70 ; ** as they have previously been patched. If they haven't
 80 ; ** then no action is taken.
 90 ;
100 NETCLO: PUSH HL
110         PUSH AF
120         SET  1,(IY+#4E) ; Set our active bit in flags 2 byte
130         BIT  0,(IY+#4E) ; See if printer entries changed
140         JR   Z,CLS001     ; jump if not
150         RES  0,(IY+#4E) ; clear flag
160         LD   L,(IY)       ; Get first byte
170         LD   H,(IY+#1)    ; Get second
180         LD   (#BD2B),HL   ; restore them
190         LD   L,(IY+#2)      ; get third byte
200         LD   H,(IY+#3)      ; and the final byte
210         LD   (#BD2D),HL     ; restore them
220 CLS001: BIT  7,(IY+#4F)    ; see if the CAS xx entries have
230         JR   Z,CLSEND       ; been patched, jump if not
240         RES  7,(IY+#4F)    ; clear flag if so - we are going
250         LD   L,(IY+#4)      ; to restore them. Get 1st byte
260         LD   H,(IY+#5)      ; and second
270         LD   (#BC77),HL     ; restore them
280         LD   L,(IY+#6)      ; get third
290         LD   H,(IY+#7)      ; and fourth
300         LD   (#BC79),HL     ; restore them
310         LD   L,(IY+#8)      ; Now the entries for CAS IN CHAR
320         LD   H,(IY+#9)      ; get restored
330         LD   (#BC80),HL
340         LD   L,(IY+#A)
350         LD   H,(IY+#B)
360         LD   (#BC82),HL     ; And then it is done
370 CLSEND: POP  AF
380         POP  HL
390         RES  1,(IY+#4E)    ; reset our active bit
400         RET  ; go back to caller
410 ;
```

Fig. 6.3

4) Next comes **NETSEND**. This routine acts as a binary scoop. That is to say that you specify two memory addresses. The first is the start address and the second is the ending address. NETSEND then sends a copy of the contents of the meory locations within the range specified over the PTC. NETSEND is listed in Fig. 6.4. You must specify a start and end address, an error message is issued if you don't, or if the start address is greater than the end address. These start and end address values are sent in the header (see above) so that the receiving machine knows where to put the received data in its memory. When all the bytes in the range have been sent, the message "** Transfer completed **" is printed on the CPC screen. If for any reason the transfer hangs up, you can usually abort it by typing CTRL/A on the CPC keyboard.

```
 10 ;
 20 ; ** NET SEND: Invoked by a command like - **
 30 ; ** |NET.SEND,START_ADDR,END_ADDR          **
 40 ; ** So long as both start and end addr    **
 50 ; ** are specified they are accepted       **
 60 ;
 70 ;
 80 NETSND: PUSH AF
 90         PUSH DE
100         PUSH HL
110         PUSH IX
120         PUSH BC
130         CP   #2          ; See if two params passed to us
140         JR   Z,SND001    ; Jump if yes
150 SND004: LD   IX,BUMPAR   ; Else output an error messge.
160         CALL OUTPUT      ; Splash it out
170         JP   SNDEND      ; And goodbye
180 BUMPAR: DEFM "** You must specify start and end **"
190         DEFB #0D,#0A
200         DEFM "** addresses for sending. Eg-|NET.SEND,1000,2000 **"
210         DEFB #0A,#0D,#FF
220 SND001: PUSH IY ; Copy IY to HL
230         POP  HL
240         LD   BC,#50
250         ADD  HL,BC       ; Now HL points at header buffer
260         LD   (HL),#00    ; Set datatype as binary
270         INC  HL          ; Step HL to next byte
280         LD   A,(IX+2)    ; Get data start (L)
290         LD   (HL),A      ; Put it buffer
300         LD   A,(IX+3)    ; Get data start (H)
310         INC  HL          ; Bump buffer pointer
320         LD   (HL),A      ; Place data start (H)
330         LD   A,(IX)      ; Get data end (L)
340         INC  HL
350         LD   (HL),A      ; place data end (H)
360         LD   A,(IX+1)    ; Get data end (H)
370         INC  HL          ; bump pointer
380         LD   (HL),A      ; buffer byte
390         INC  HL          ; bump pointer again.
400         LD   B,16        ; Now we put 16 spaces to hdr buffr
410         LD   A,#20
420 SND002: LD   (HL),A
430         INC  HL
440         DJNZ SND002      ; Loop until all 16 done
450         CALL SNDHDR      ; Now get the header sent
460         LD   L,(IY+#51)
470         LD   H,(IY+#52)    ; HL points at data start
480         LD   C,(IY+#53)     ; and BC at data end addr
490         LD   B,(IY+#54)
500         SBC  HL,BC           ; See if end greater than start
510         JP   NC,SND004       ; jump if so.
520         LD   L,(IY+#51)      ; else reload start addr to HL
```

```
530            LD    H,(IY+#52)
540  SND003:  LD    A,(HL)        ; Get a byte from memory
550            CALL  SEND          ; Get the byte sent
560            JR    C,SNDEND      ; If user typed CTRL/A then endup
570            INC   HL            ; Else we bump buff ptr
580            LD    A,L           ; copy L to A
590            CP    (IY+#53)      ; See if at end of buffer yet
600            JR    NZ,SND003     ; Jump if no match in low byte
610            LD    A,H           ; Else get high byte
620            CP    (IY+#54)      ; see if it matches
630            JR    NZ,SND003     ; Jump if not
640  ; ** When we get here the transfer has completed **
650  ; ** so we output a message to say so            **
660  ;
670            LD    IX,SUCESS
680            CALL  OUTPUT
```

Fig. 6.4

5) The last major routine is **NETREC**. This is the main network receptor module. When you issue a NET.REC command the routine begins waiting for reception of a valid header marker byte to be available at the PTC receive channel, using the NETLIB routine RECHDR. If for any reason the header is never received you can exit from RECHDR with CTRL/A when RECHDR has received the header it returns to NETREC. As the listing of NETREC in Fig. 6.5 shows the next thing which happens is that NETREC sets its activity bit in the function byte located in the CPCNET work area. Then it examines the datatype byte of the received header.

```
 10 ;
 20 ; ** NETREC: This CPCNET routine is the main receptor **
 30 ; ** of data from receive channel of the PTC. It      **
 40 ; ** receives the packet header and sees if what is   **
 50 ; ** receiving is a BASIC program. If so then it will **
 60 ; ** create the required  diversions in  jumpblocks   **
 70 ; ** and prompt  user to issue a LOAD command. If the **
 80 ; ** received data type is binary,   NETREC does the  **
 90 ; ** job of storing  incoming data itself. NOTE: when **
100 ; ** receiving binary data no check is made  if the   **
110 ; ** area specified for the buffer is in use or not.  **
120 ;
130 ;
140 NETREC:  PUSH AF
150          PUSH HL
160          PUSH BC
170          PUSH DE ; Now we can use any of prime regs.
180          CALL RECHDR ; First we call header receive s/r
190          JP   C,RECEND ; If CTRL/A was pressed we abort
200          SET  2,(IY+#4F) ; Set our activity bit in func byte
210          LD   A,(IY+#50) ; Get data type
220          CP   #01        ; See if its BASIC
230          JR   Z,BASREC   ; Jump to BASIC receiver if so
240 BINREC:  LD   L,(IY+#52) ; Else HL=buffer start addr
250          LD   L,(IY+#53)
260          LD   E,(IY+#53) ; DE = Last buffer address
270          LD   D,(IY+#54)
280 REC001:  CALL RECEEV     ; Get a character
290          JP   C,RECEND     ; If CTRL/A pressed then abort
300          LD   A,(HL)       ; Else buffer the byte
310          LD   A,E          ; See if low byte of bp
320          CP   L            ; and end value match
330          JR   NZ,REC002    ; Jump if no match in low byte
340          LD   A,D          ; Else compare high bytes
350          CP   H            ; See if high bytes match
```

```
360             JP    Z,RECEND        ; Jump if so
370 REC002:     INC   HL              ; Bump pointer
380             JR    REC001          ; And loop again
390 BASREC:     BIT   0,(IY+#4E)    ; See if we have already patched
400             JR    NZ,REC003       ; jump over patching section if so
410             LD    A,(#BC77)       ; BASIC program loader
420             LD    (IY+#4),A       ; Save first byte of CAS IN OPEN
430             LD    HL,(#BC78)              ; Get the other two bytes
440             LD    (IY+#5),L               ; Save them in wk area
450             LD    (IY+#6),H
460             LD    A,ROMNUM                ; Get ROMnum into A
470             LD    (IY+#15),A              ; Place it as third byte of far addr
480             PUSH  IY
490             POP   HL                      ; Copy IY to HL
500             LD    DE,#13                  ; Make address of far address in HL
510             ADD   HL,DE
520             LD    (#BC78),HL              ; Place in jumpblock
530             LD    A,#DF                   ; And the RST 18
540             LD    (#BC77),A               ; as well.
550             LD    A,(#BC7A)       ; Save the next byte too
560             LD    (IY+#7),A       ; Cos we have to put a RET in it
570             LD    A,#C9           ; Which we duly put there
580             LD    (#BC7A),A
590             LD    DE,BAS001     ; DE points at BAS001 rtne
600             LD    (HL),E      ; Place BAS001 address in far address
610             INC   HL
620             LD    (HL),D
630             LD    A,(#BC80)       ; Now we patch CAS IN CHAR
640             LD    (IY+#8),A       ; Save original bytes
650             LD    HL,(#BC81)
660             LD    (IY+#9),L
670             LD    (IY+#A),H
680             LD    A,(#BC83)       ; And the third byte as well
690             LD    (IY+#B),A
700             LD    A,#C9           ; so we can put a RET there
710             LD    (#BC83),A
720             LD    HL,RECBAS
730             LD    (IY+#16),L      ; Now make the far address
740             LD    (IY+#17),H      ; for the CAS IN CHAR replacement
750             LD    A,ROMNUM        ; and the ROMNUM completes far address
760             LD    (IY+#18),A
770             PUSH  IY
780             POP   HL            ; Copy IY to HL as before
790             LD    BC,#16          ; BC is offset to far address location
800             ADD   HL,BC         ; Now HL points at far address address
810             LD    A,#DF           ; Place RST 18 into jump block
820             LD    (#BC80),A       ; entry for CAS IN CHAR
830             LD    (#BC81),HL      ; and then the far addresses address
840             SET   7,(IY+#4F)    ; Set special bit in func byte which says
850 ; ** the jump block entries for cassette I/O are changed.
860 REC003:     LD    IX,RECMES ; Now we must instruct how to receive
870             CALL  OUTPUT
880             JP    RECEND
890 RECMES:     DEFB  #0A,#0D
900             DEFM  "** The incoming data is to be fed into BASIC. Please **"
910             DEFB  #0A,#0D
920             DEFM  "** type a LOAD  or a MERGE  command to begin        **"
930             DEFB  #0A,#0D
940             DEFM  "** reception of it.                 CPCNET          **"
950             DEFB  #0A,#0D,#FF
960 RECEND:     POP   DE ; Clear up the stack
970             POP   BC
980             POP   HL
990             POP   AF
1000            RES   2,(IY+#4F) ; Clear our activity bit in func
1010            RET   ; And return
1020 ;
```

Fig. 6.5

The datatype found in the header determines which one of two paths the routine now takes. If it indicates that the incoming data is BASIC then a fairly complex sequence is begun, this will be described separately in a moment. If the datatype is binary then a small routine is entered which sets up the end address in the header as its maximum buffer address, and the start address from the header as its buffer start address. Bytes are then loaded as available from the PTC receive channel into the buffer, the pointer to which is incremented after every byte is buffered. Reception terminates when the pointer reaches the same value as the end address.

There is an important implication of reception of binary data in this way. This is that if the incoming start and end address are silly, for example 0000 to FFFF then the CPC could easily crash due to system variables in RAM being overwritten. Protection against this is not really possible without limiting the usefulness of the facility.

If the datatype byte says that the incoming data is BASIC then the user is required to cooperate with the software a little. What actually happens is that NETREC saves the contents of the jumpbox entries for the CAS IN OPEN and CAS IN CHAR routines and replaces them with entries which point to two routines from the NETLIB subroutine library. These are BAS001 and RECBAS. These two routines supply BASIC with the same inputs as the routines they replace. BAS001 duplicates the function of CAS IN OPEN insofar as it fools BASIC into thinking that a cassette or disk file has been opened. As we saw in chapter three the only parameter passed back to BASIC which seems to be checked is the filetype. BAS001 passes a filetype of ASCII BASIC back. RECBAS replaces CAS IN CHAR which passes the contents of the opened file back to BASIC a character at a time, passing specific flag conditions and an EOF (Hex 1B) back when all characters from the file have been supplied.

Once the jumpblock entries have been patched an explanatory message is output to tell the user to type in a `LOAD_` command. BASIC will then start loading incoming characters from the PTC receive channel via the BAS001 and RECBAS routines. The sending machine must end transmission with a byte which has bit seven set. When this happens the reception ends and the BASIC program is now loaded. At any time during the load you can type CTRL/A to abort it. BASIC will be signalled a BREAK condition, just as if you had typed ESC.

6) The **NETLIB** listing is Fig. 6.6. This shows the various subroutines used by the major CPCNET routines. Some of these are familiar from the ROMdisk software, but they all have explanatory notes as to what they do. The SEND routine may need a little further explanation and because it deals with the PTC directly you should also study the hardware notes which follow. SEND is used by any routine wishing to have a byte sent over the CPCNET. SEND waits for the transmit to be ready to accept a byte. If whilst SEND is waiting it detects that you have typed anything at the CPC keyboard then the send operation is aborted and a return to the caller of SEND is made with the carry

flag set to show an abnormal completion. As already stated the remarks for each of the other NETLIB subroutines should sufficiently explain their purpose.

```
 90 ; ----------------------------------------------------------
 91 ;                  NETLIB CPCNET Subroutne library
 92 ; ----------------------------------------------------------
100 ;
110 ;
120 ; ** SNDHDR: This subroutine sends the contents **
130 ; ** of the CPCNET header buffer over the PTC   **
140 ; ** IY points to base of work area.
150 ;
160 SNDHDR: PUSH AF
170         PUSH HL
180         PUSH BC
190         SET  4,(IY+#4F)      ; Set our active bit
200         LD   A,#AA
210         CALL SEND ; Send the marker byte
220         JR   C,HDR001 ; jump out for keypress
230         PUSH IY
240         POP  HL ; Copy IY to HL
250         LD   BC,#50
260         ADD  HL,BC ; Now HL points to the header buffer
270         LD   B,#30  ; B is the loop counter
280 SNDLOP: LD   A,(HL) ; get a byte
290         CALL SEND     ; Send it
300         JR   C,HDR001 ; If user pressed key then exit
310         INC  HL       ; Else bump pointer
320         DJNZ SNDLOP   ; Jump if not done yet
330         LD   A,0     ; Ensure carry clear at end
340         ADC  A,0
350 HDR001: POP  BC
360         POP  HL
370         POP  AF
380         RES  4,(IY+#4F) ; Clear active bit
390         RET  ; Return
400 ;
410 ; ** The output routine outputs messages to the screen **
420 ; ** IX points to the start of message, which ends with**
430 ; ** a byte value of #FF. AF, IX ,HL corrupted.        **
440 OUTPUT: LD   A,(IX) ; Get a byte
450         BIT  7,A    ; See if EOB marker
460         RET  NZ     ; Return if so
470         CALL #BB5A  ; Else output the chracter
480         INC  IX     ; bump buffer pointer
490         JR   OUTPUT ; and do it again
500 HEXAS:  LD   H,A  ; Save original byte in H
510         AND  #0F  ; AND out the low nibble
520         OR   #30  ; Convert to ASCII
530         CP   #3A  ; see if its alpha
540         JP   M,HIGH ; If not then jump over adjustment
550         ADD  A,#7  ; make it HEX
560 HIGH:   LD   L,A   ; Put finished low digit away in L
570         LD   A,H   ; Get out original byte
580         SRL  A     ; Make High nibble low
590         SRL  A
600         SRL  A
610         SRL  A
620         OR   #30
630         CP   #3A   ; See if Alpha
640         JP   M,DONE ; Done if not
650         ADD  A,#7   ; Adjust as before
660 DONE:   LD   H,A    ; Save finished thing into H
670         RET  ; and finish
680 ;
690 ;
```

```
700 ; ** BAS001: This routine replaces the firmware CAS IN OPEN
710 ; ** routine during input of BASIC programs from the CPCNET
720 ; ** It just fools BASIC into thinking that it has opened
730 ; ** a BASIC program stored in ASCII sucessfully
740 ;
750 ;
760 BAS001: LD   A,#16 ; Load ASCII filetype code
770         CP   #0 ; Ensure Z flag is clear
780         SCF  ; and Carry flag is set
790         RET  ; and go back to BASIC
800 ;
810 ;
820 ;
830 ;
840 ; ** RECBAS: This CPCNET routine replaces the firmware
850 ; ** CAS IN CHAR routine, which feeds BASIC programs
860 ; ** fetched from cassette or disk to BASIC a character
870 ; ** at a time. This routine uses the RECEEV s/r and
880 ; ** flags end of file to BASIC if byte received has got
890 ; ** bit seven set.
900 ;
910 ;
920 RECBAS: CALL RECEEV ; Get a character
930         JR   NC,RECB01 ; Jump if CTRL/A not hit by user
940         LD   A,0   ; otherwise
950         ADC  A, #0 ; Set zero and clear carry to tell
960         RET  ; BASIC that BREAK was hit and RET
970 RECB01: BIT  7,A ; See if top bit set
980         JR   Z,RECB02 ; Jump if no
990         LD   A,0
1000        ADC  A,#0
1010        LD   A,#1F ; Ensure carry andZ flag clr and
1020        RET  ; put EOF char into A then ret
1030 RECB02: CP  #FF ; Ensure Z clear
1040        SCF  ; and carry set to tell BASIC its
1050        RET  ; OK then return
1060 ;
1070 ;
1080 ; ** RECEEV: This routine returns characters to
1090 ; ** other software which have been received via the CPCNET
1100 ; ** Parallel transfer channel. It waits for the IBF (input
1110 ; ** buffer full) signal of port B to go high, which indicates
1120 ; ** that a byte is available. While it is waiting it also
1130 ; ** keeps calling a firmware  routine to see if CTRL/A has
1140 ; ** been typed at the keyboard. If it has it returns with
1150 ; ** the carry flag set. When a byte is read it is passed
1160 ; ** back in the A register.
1170 ;
1180 ;
1190 RECEEV: PUSH BC
1200        SET  5,(IY+#4F) ; Set active bit
1210 RECEE2: LD  BC,PTCCNT    ; BC points to control channel
1220        IN   A,(C)        ; Get control status
1230        BIT  1,A          ; See if IBF is true
1240        JR   NZ,RECEE1     ; Jump if so
1250        CALL #BB09        ; else check for CTRL/A's
1260        JR   NC,RECEE2    ; If no key pressed then loop
1270        CP   #1           ; else see if CTRL/A
1280        JR   NZ,RECEE2    ; loop if no
1290        SCF  ; if yes then set carry
1300        JR   RECOUT       ; and go  out
1310 RECEE1: LD  BC,PTCREC    ; BC points at receive channel
1320        IN   A,(C)        ; get the character
1330        CP   A            ; Ensure carry is clear.
1340 RECOUT: POP BC           ; Clean stack
1350        RES  5,(IY+#4F) ; Clear active bit
1360        RET  ; And go back
1370 ;
1380 ;
```

```
1390 ; ** SEND: This CPCNET routine sends the character it **
1400 ; ** gets in the A reg out onto the PTC transmit channel **
1410 ; ** but it first waits for the channel to be ready- as  **
1420 ; ** indicated by a high on bit 7 or port C - PTCCNT.    **
1430 ; ** If the user presses any key while SEND is waiting  **
1440 ; ** then the carry flag is set and a return is done    **
1450 ;
1460 SEND:   PUSH BC
1470         PUSH HL
1480         SET  6,(IY+#4F) ; Set active bit
1490 SEND00: LD   BC,PTCCNT    ; Make BC point to PTC status register
1500         IN   L,(C)       ; Get status into L
1510         BIT  7,L         ; See if send channel is ready
1520         JR   NZ,SEND01   ; Jump if yes
1530         PUSH AF          ; Save the byte for sending
1540         CALL #BB09       ; see if user has typed anything
1550         JR   NC,SEND02   ; Jump if not
1560         POP  AF          ; Restore AF anyway
1570         SCF  ; Set carry
1580         JR   SEND04      ; And end up
1590 SEND02: POP  AF          ; If nowt typed then restore AF
1600         JR   SEND00      ; and loop round again
1610 SEND01: LD   BC,PTCSND   ; Now BC points to send channel
1620         OUT  (C),A       ; and send the byte out
1630         AND  A           ; Ensure carry flag is clear
1640 SEND04: POP  HL          ; POP the stack
1650         POP  BC
1660         RES  6,(IY+#4F)  ; Clear active bit
1670         RET  ; And return
1680 ;
1690 ;
1700 ;
1710 ;
1720 ; ** RECHDR: This routine receives a CPCNET header from the
1730 ; ** receive channel of the PTC. The header must be in the
1740 ; ** correct format, or a warning message is issued.
1750 ;
1760 ;
1770 RECHDR: PUSH BC
1780         PUSH HL
1790         PUSH DE
1800         SET  3,(IY+#4F)     ; Set our function active bit
1810         PUSH IY          ; IY copied to HL
1820         POP  HL
1830         LD   BC,#50
1840         ADD  HL,BC          ; Now HL points to header buffer
1850 RECH01: CALL RECEEV   ; Get a byte
1860         JP   C,RECH04   ; If user wants out then go out
1870         CP   #AA        ; See if it is the start marker
1880         JR   Z,RECH01   ; loop if not
1890         CALL RECEEV   ; if so then reception begins
1900         LD   E,A        ; Validate the datatype (0 or 1 only)
1910         AND  #FE       ; Mask out illegal bits
1920         JR   Z,RECH02   ; If no unused bits set then jump
1930         LD   IX,RECHER ; Else issue a warning message
1940         CALL OUTPUT
1950         JP   RECH04     ; and then abort reception
1960 RECHER: DEFB #0A,#0D
1970         DEFM "** Warning message: RECHDR detects illegal datatype **"
1980         DEFB #0A,#0D
1990         DEFM "** header reception aborted - tell sender           **"
2000         DEFB #0A,#0D,#FF
2010 RECH02: LD   (HL),E ; Put datatype into buffer
2020         INC  HL ; Bump the buffer pointer
2030         LD   B,#1B      ; 20 more bytes to get for header
2040 RECH03: CALL RECEEV   ; Get a byte
2050         JR   C,RECH04   ; Abort if user wants to
2060         LD   (HL),A     ; Buffer it
```

```
2070          INC  HL        ; update pointer
2080          DJNZ RECH03    ; loop till done
2090 RECH04:  POP  DE
2100          POP  HL
2110          POP  BC        ; Restore entry contents
2120          RES  3,(IY+#4F) ; Clear active flag
2130          RET  ; and go back
```

Fig. 6.6

The CPCNET hardware

The CPCNET hardware was detailed in Chapter Three as the Parallel transfer channel (PTC). We shall now look at the precise details of how this is used in the CPCNET application. Fig. 6.7 shows the details of the interconnecting lead required to connect two CPC machines fitted with a PTC to enable them to be used for data transfer via CPCNET.

The PPI chips used in the design have the capability to transact single byte transfers independently of the computer. This capability was described in chapter three. The extent to which the PPI hardware takes care of the transfers allows us to use port C of the PPI as a status register for the PTC. The bits in this status register are used as follows:

Name	TCRDY	ACK	XXX	XXX	XXX	STB	RCRDY	XXX
bit	7	6	5	4	3	2	1	0

0/3/4/5 = Unused bits (designated as XXX)

7 TCRDY = When read as a logic one this means that the transmit channel (PTCSND in the assembly listing) is ready to accept a byte.

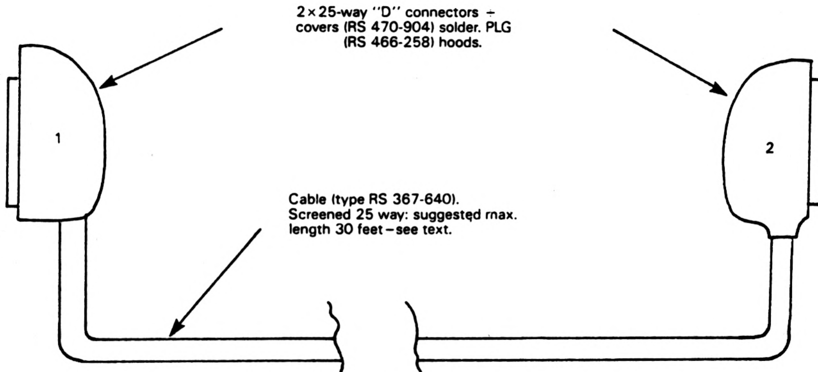
6 ACK = This bit pulses los as the receiving machine (at the other end of the line) accepts the byte just sent to it.

2 STB = Goes low as a byte is strobed into the PTC receive register by the remote machine.

1 RCDY = Receive channel ready. When high, this signal indicates that a byte is available from the PTC receive channel register.

The PTC hardware is very simple to interface to devices other than another PTC. For example my test rig for the PTC involved an interface to a UART. Readers should not find it too difficult to adapt the hardware for connection to other machines.

Fig. 6.7: CPC connecting lead



When using the indicated cable the following connection schedule may be used to make the CPCNET interconnecting cable:

End 1 Pin#	Wire colour/	End 2 pin#
1	Red + screen	1
2	Blue	10
3	Green	11
4	Yellow	12
5	White	13
6	Black	14
7	Brown	15
8	Violet	16
9	Orange	17
10	Pink	2
11	Turquoise	3
12	Grey	4
13	Red/Blue	5
14	Green/Red	6
15	Yellow/Red	7
16	White/Red	8
17	Red/Black	9
18	Red/Brown	21
19	Yellow/Blue	20
20	White/Blue	19
21	Blue/Black	18
25	Orange/Green	25

CPCNET applications

The CPCNET can be adapted to transfer most kinds of data from one machine to another. Its chief benefit lies in the great speed with which transfers can take place. For example to transfer 16K bytes (as in a screen dump) takes slightly less than two seconds. The speed at which transfers take place is somewhat offset by the preparation needed before transfer can commence, but this would not matter where two CPC's fitted with PTC hardware and CPCNET software were left permanently connected.

CPCNET ROM listing

Figure 6.8 is the assembly listing and hexdump of the CPCNET software to be blown into a ROM. The program is pretty well commented so you should not have any problems applying modifications to suit your own needs. The CPCNET work area (to which IY points on entry to any CPCNET routine) is used as follows:

IY+0 to IY+3 =	Used to save the original jumpblock entry for MC PRINT CHAR and the byte following it.
IY+4 to IY+7 =	Used to save the original contents of the CAS IN OPEN jumpblock entry, and the byte following it.
IY+#8 to IY+#B =	Used to save the original jumpblock entry for CAS IN CHAR
IY+#10 to IY+#12 =	Contain the far address of the SEND routine.
IY+#13 to IY+#15 =	Contain far address of BAS001
IY+#16 to IY+#19 =	Contain far address of RECBAS
IY+#4E =	flags2 byte. If bit zero is set this means that the printer jumpblock entries have been changed. If bit one is set it means that NETCLOS is active.
IY+#4F =	Flags1 byte. Bit 0 set means NETOPEN is active Bit 2 set means NETREC is active Bit 3 set means RECHDR is active Bit 4 set means SNDHDR is active Bit 5 set means RECEEV is active Bit 6 set means SEND is active Bit 7 set means that the cassette routines jump block entries have been altered by CPCNET.
IY+#50 to IY+#7F =	header buffer – specifically :
IY+#50 =	Datatype: 0=Binary 1=BASIC
IY+#51 =	Data start address (low byte)
IY+#52 =	Data start address (high byte)
IY+#53 =	Data ends address (low byte)
IY+#54 =	Data ends address (high byte)
IY+#55 to IY+#64 =	Data name (16 chars. Not used in CPCNET V1.1)
IY+#65 to IY+#7F =	unused by CPCNET V1.1

```
                    10 *H+ CPCNET ROM listing
                    20 ;
                    30 ;
                    40 ; **              CPCNET    ROM listing           **
                    50 ;
                    60 ;
                    70 ; Manually operated intercomputer linking network for the
                    80 ; Amstrad CPC computers. Kernow computers C 1985
                    90 ;
                   100 ;
F8F0               110 PTCSND EQU  #F8F0 ; Define the addresses for the PTC regs.
F8F1               120 PTCREC EQU  #F8F1
F8F2               130 PTCCNT EQU  #F8F2 ; The control signals for the channels
F8F3               140 PPICTL EQU  #F8F3 ; The control reg for the PTC's PPI chip
0005               150 ROMNUM EQU  5 ; The number of ROM socket we are to live in
                   160 ;
                   170 ;
                   180 ;
C000               190        ORG  #C000
                   191 *T+ CPCNET.OBJ
                   200 ;
                   210 ;
C000  01           220        DEFB #1 ; BACKGROUND ROM
C001  010101       230        DEFB #1,#1,#1 ; Mark 1 Vers 1 rev 1
C004  15C0         240        DEFW NAMTAB ; Set pointer to name table.
C006  C33EC0       250        JP   NETGO  ; Power on init routine
C009  C382C0       260        JP   NETOPN
C00C  C314C2       270        JP   NETCLOS
C00F  C36BC2       280        JP   NETSND
C012  C35BC3       290        JP   NETREC
C015  4E455420     300 NAMTAB DEFM "NET INI" ; Init entry
C01C  D4           310        DEFB "T"+#80
C01D  4E45542E     320        DEFM "NET.OPE"
C024  CE           330        DEFB "N"+#80
C025  4E45542E     340        DEFM "NET.CLOS"
C02D  C5           350        DEFB "E"+#80
C02E  4E45542E     360        DEFM "NET.SEN"
C035  C4           370        DEFB "D"+#80
C036  4E45542E     380        DEFM "NET.RE"
C03C  C3           390        DEFB "C"+#80
C03D  00           400        DEFB 0 ; The end of name table marker.
                   410 ;
                   420 ;
                   430 ; ** NETGO: This routine initialises the network at power
                   440 ; ** on. It is limited to setting up the PPI and
                   450 ; ** creating a work area in RAM which the CPC will leave
                   460 ; ** alone.
C03E  F5           470 NETGO  PUSH AF
C03F  C5           480        PUSH BC
C040  25           490        DEC  H    ; HL contains upper limit of memory pool
                   500 ;                  grab us 256 bytes for work area
C041  01F3F8       510        LD   BC,PPICTL ; BC points at PPI control
C044  3EAF         520        LD   A,#AF    ; Mode byte is AF
C046  ED79         530        OUT  (C),A    ; write it
C048  3E00         540        LD   A,0
C04A  FD774E       550        LD   (IY+#4E),A
C04D  FD774F       560        LD   (IY+#4F),A ; Zero the flags bytes
C050  DDE5         570        PUSH IX
C052  DD215FC0     580        LD   IX,INIMES ; Tell the user its done
C056  CDF1C4       590        CALL OUTPUT     ; by OUTPUTing a message
C059  DDE1         600        POP  IX
C05B  C1           610        POP  BC
C05C  F1           620        POP  AF
C05D  37           630        SCF
C05E  C9           640        RET  ; Restore regs, set carry and RET.
C05F  0A0D         650 INIMES DEFB #0A,#0D
C061  2A2A2043     660        DEFM "** CPCNET  V1.1 initialised **"
C07F  0A0DFF       670        DEFB #0A,#0D,#FF
```

```
                         680 ;
                         690 ;
                         700 ; ** NETOPN: This CPCNET routine opens the PTC to **
                         710 ; ** BASIC on stream 8. This allows text versions **
                         720 ; ** of CPC BASIC programs to be transferred over **
                         730 ; ** the CPCNET, using LIST #8  or PRINT #8 type  **
                         740 ; ** statements                                   **
                         750 ;
                         760 ;
C082  F5                 770 NETOPN PUSH AF
C083  E5                 780        PUSH HL
C084  FDCB4E46           790        BIT  0,(IY+#4E) ; See if flags say jump blk patched
C088  203E               800        JR   NZ,MAKHDR  ; so jump over patching section
C08A  FDCB4EC6           810        SET  0,(IY+#4E) ; if we shall patch then flag set
C08E  FDCB4FC6           820        SET  0,(IY+#4F) ; Set our active bit in func byte
C092  2A2BBD             830        LD   HL,(#BD2B) ; Get first two bytes
C095  FD7500             840        LD   (IY),L     ; Save first
C098  FD7401             850        LD   (IY+#1),H  ; Save second
C09B  2A2DBD             860        LD   HL,(#BD2D) ; Get last two bytes from jmp block
C09E  FD7502             870        LD   (IY+#2),L  ; Save third byte
C0A1  FD7403             880        LD   (IY+#3),H  ; And the final one
C0A4  3EDF               890        LD   A,#DF      ; Load the restart instruction
C0A6  322BBD             900        LD   (#BD2B),A  ; in place of previous contents
C0A9  2164C5             910        LD   HL,SEND    ; Now HL points to send routine.
C0AC  FD7510             920        LD   (IY+#10),L ; Place first byte
C0AF  FD7411             930        LD   (IY+#11),H ; And second
C0B2  2605               940        LD   H,ROMNUM   ; Now the ROM number where we live
C0B4  FD7412             950        LD   (IY+#12),H ; Place the ROM numb in far address
C0B7  FDE5               960        PUSH IY         ; Again copy IY to HL
C0B9  E1                 970        POP  HL
C0BA  D5                 980        PUSH DE         ; We need DE for a moment - save it,
C0BB  111000             990        LD   DE,#10
C0BE  19                1000        ADD  HL,DE      ; Calculate far address for jmp blk
C0BF  D1                1010        POP  DE         ; Finished with DE so restore it
C0C0  222CBD            1020        LD   (#BD2C),HL ; And place far address in jmp blk,
C0C3  3EC9              1030        LD   A,#C9      ; Load a RET instruct into entry afte
C0C5  322EBD            1040        LD   (#BD2E),A  ; PRINT CHAR for the return.
C0C8  3E01              1050 MAKHDR LD   A,#1         ; Now construct the header buffer
C0CA  FD7750            1060        LD   (IY+#50),A   ; Set data type to BASIC (01)
C0CD  3E00              1070        LD   A,0          ; Buffer address fields get zeroed
C0CF  FD7751            1080        LD   (IY+#51),A
C0D2  FD7752            1090        LD   (IY+#52),A
C0D5  FD7753            1100        LD   (IY+#53),A
C0D8  FD7754            1110        LD   (IY+#54),A   ; Now we must zero the data fields
C0DB  FDE5              1120        PUSH IY           ; Copy IY to HL again
C0DD  E1                1130        POP  HL
C0DE  015500            1140        LD   BC,#55       ; Make HL point to name field
C0E1  09                1150        ADD  HL,BC
C0E2  0610              1160        LD   B,16         ; Sixteen chars to be zeroed
C0E4  77                1170 OPN009 LD   (HL),A       ; Zero a byte
C0E5  23                1180        INC  HL           ; Bump pointer
C0E6  10FC              1190        DJNZ OPN009       ; Loop until done
C0E8  CDC5C4            1200        CALL SNDHDR       ; Call the header send
C0EB  DD21F5C0          1210        LD   IX,OPEND     ; Now we must tell user
C0EF  CDF1C4            1220        CALL OUTPUT
C0F2  C30DC2            1230        JP   OPNEND       ; and end up
C0F5  0A0D              1240 OPEND  DEFB #0A,#0D
C0F7  2A2A2043          1250        DEFM "** CPCNET is now opened **"
C127  0A0D              1260        DEFB #0A,#0D
C129  2A2A2070          1270        DEFM "** programs or variables on stream 8. **"
C159  0A0D              1280        DEFB #0A,#0D
C15B  2A2A2075          1290        DEFM "** use the LIST #8 or PRINT #8 statments
C194  0A0D              1300        DEFB #0A,#0D
C196  2A2A2077          1310        DEFM "** when you have finished sending, |NET.CLOSE
C1CF  0A0D              1320        DEFB #0A,#0D
C1D1  2A2A2063          1330        DEFM "** command to terminate
C20A  0A0DFF            1340        DEFB #0A,#0D,#FF
C20D  E1                1350 OPNEND POP  HL
```

```
C20E  F1        1360         POP  AF          ; Then weve done it so restore
C20F  FDCB4F86  1370         RES  0,(IY+#4F)  ; reset our active flag
C213  C9        1380         RET  ; and go back
                1390 ;
                1400 ;
                1410 ;
                1420 ; ** NETCLO: This CPCNET routine closes the PTC channel
                1430 ; ** to BASIC for output and input. It restores the
                1440 ; ** jumpblock entries to their original values, so long
                1450 ; ** as they have previously been patched. If they haven't
                1460 ;'** then no action is taken.
                1470 ;
C214  E5        1480 NETCLO PUSH HL
C215  F5        1490        PUSH AF
C216  FDCB4ECE  1500        SET  1,(IY+#4E) ; Set our active bit in flags 2 byte
C21A  FDCB4E46  1510        BIT  0,(IY+#4E) ; See if printer entries changed
C21E  2816      1520        JR   Z,CLS001    ; jump if not
C220  FDCB4E86  1530        RES  0,(IY+#4E) ; clear flag
C224  FD6E00    1540        LD   L,(IY)      ; Get first byte
C227  FD6601    1550        LD   H,(IY+#1)   ; Get second
C22A  222BBD    1560        LD   (#BD2B),HL  ; restore them
C22D  FD6E02    1570        LD   L,(IY+#2)     ; get third byte
C230  FD6603    1580        LD   H,(IY+#3)     ; and the final byte
C233  222DBD    1590        LD   (#BD2D),HL    ; restore them
C236  FDCB4F7E  1600 CLS001 BIT  7,(IY+#4F)   ; see if the CAS xx entries have
C23A  2828      1610        JR   Z,CLSEND      ; been patched, jump if not
C23C  FDCB4FBE  1620        RES  7,(IY+#4F)   ; clear flag if so - we are going
C240  FD6E04    1630        LD   L,(IY+#4)     ; to restore them. Get 1st byte
C243  FD6605    1640        LD   H,(IY+#5)     ; and second
C246  2277BC    1650        LD   (#BC77),HL    ; restore them
C249  FD6E06    1660        LD   L,(IY+#6)     ; get third
C24C  FD6607    1670        LD   H,(IY+#7)     ; and fourth
C24F  2279BC    1680        LD   (#BC79),HL    ; restore them
C252  FD6E08    1690        LD   L,(IY+#8)     ; Now the entries for CAS IN CHAR
C255  FD6609    1700        LD   H,(IY+#9)     ; get restored
C258  2280BC    1710        LD   (#BC80),HL
C25B  FD6E0A    1720        LD   L,(IY+#A)
C25E  FD660B    1730        LD   H,(IY+#B)
C261  2282BC    1740        LD   (#BC82),HL    ; And then it is done
C264  F1        1750 CLSEND POP  AF
C265  E1        1760        POP  HL
C266  FDCB4E8E  1770        RES  1,(IY+#4E)   ; reset our active bit
C26A  C9        1780        RET  ; go back to caller
                1790 ;
                1800 ;
                1810 ; ** NET SEND: Invoked by a command like - **
                1820 ; ** |NET.SEND,START_ADDR,END_ADDR        **
                1830 ; ** So long as both start and end addr   **
                1840 ; ** are specified they are accepted      **
                1850 ;
                1860 ;
C26B  F5        1870 NETSND PUSH AF
C26C  D5        1880        PUSH DE
C26D  E5        1890        PUSH HL
C26E  DDE5      1900        PUSH IX
C270  C5        1910        PUSH BC
C271  FE02      1920        CP   #2          ; See if two params passed to us
C273  2866      1930        JR   Z,SND001    ; Jump if yes
C275  DD217FC2  1940 SND004 LD   IX,BUMPAR   ; Else output an error messge.
C279  CDF1C4    1950        CALL OUTPUT      ; Splash it out
C27C  C354C3    1960        JP   SNDEND      ; And goodbye
C27F  2A2A2059  1970 BUMPAR DEFM "** You must specify start and end **"
C2A3  0D0A      1980        DEFB #0D,#0A
C2A5  2A2A2061  1990        DEFM "** addresses for sending. **"
C2D8  0A0DFF    2000        DEFB #0A,#0D,#FF
C2DB  FDE5      2010 SND001 PUSH IY ; Copy IY to HL
C2DD  E1        2020        POP  HL
C2DE  015000    2030        LD   BC,#50
```

```
C2E1  09        2040        ADD  HL,BC      ; Now HL points at header buffer
C2E2  3600      2050        LD   (HL),#00   ; Set datatype as binary
C2E4  23        2060        INC  HL         ; Step HL to next byte
C2E5  DD7E02    2070        LD   A,(IX+2)   ; Get data start (L)
C2E8  77        2080        LD   (HL),A     ; Put it buffer
C2E9  DD7E03    2090        LD   A,(IX+3)   ; Get data start (H)
C2EC  23        2100        INC  HL         ; Bump buffer pointer
C2ED  77        2110        LD   (HL),A     ; Place data start (H)
C2EE  DD7E00    2120        LD   A,(IX)     ; Get data end (L)
C2F1  23        2130        INC  HL
C2F2  77        2140        LD   (HL),A     ; place data end (H)
C2F3  DD7E01    2150        LD   A,(IX+1)   ; Get data end (H)
C2F6  23        2160        INC  HL         ; bump pointer
C2F7  77        2170        LD   (HL),A     ; buffer byte
C2F8  23        2180        INC  HL         ; bump pointer again.
C2F9  0610      2190        LD   B,16       ; Now we put 16 spaces to hdr buffr
C2FB  3E20      2200        LD   A,#20
C2FD  77        2210 SND002 LD   (HL),A
C2FE  23        2220        INC  HL
C2FF  10FC      2230        DJNZ SND002     ; Loop until all 16 done
C301  CDC5C4    2240        CALL SNDHDR     ; Now get the header sent
C304  FD6E51    2250        LD   L,(IY+#51)
C307  FD6652    2260        LD   H,(IY+#52)   ;  HL points at data start
C30A  FD4E53    2270        LD   C,(IY+#53)    ; and BC at data end addr
C30D  FD4654    2280        LD   B,(IY+#54)
C310  ED42      2290        SBC  HL,BC          ; See if end greater than start
C312  D275C2    2300        JP   NC,SND004       ; jump if so.
C315  FD6E51    2310        LD   L,(IY+#51)      ; else reload start addr to HL
C318  FD6652    2320        LD   H,(IY+#52)
C31B  7E        2330 SND003 LD   A,(HL)       ; Get a byte from memory
C31C  CD64C5    2340        CALL SEND         ; Get the byte sent
C31F  3833      2350        JR   C,SNDEND     ; If user typed CTRL/A then endu|
C321  23        2360        INC  HL           ; Else we bump buff ptr
C322  7D        2370        LD   A,L          ; copy L to A
C323  FDBE53    2380        CP   (IY+#53)     ; See if at end of buffer yet
C326  20F3      2390        JR   NZ,SND003    ; Jump if no match in low byte
C328  7C        2400        LD   A,H          ; Else get high byte
C329  FDBE54    2410        CP   (IY+#54)     ; see if it matches
C32C  20ED      2420        JR   NZ,SND003    ; Jump if not
                2430 ; ** When we get here the transfer has completed **
                2440 ; ** so we output a message to say so            **
                2450 ;
C32E  DD2137C3  2460        LD   IX,SUCESS
C332  CDF1C4    2470        CALL OUTPUT
C335  181D      2480        JR   SNDEND
C337  0A0D      2490 SUCESS DEFB #0A,#0D
C339  2A2A2054  2500        DEFM "** Transfer completed **"
C351  0A0DFF    2510        DEFB #0A,#0D,#FF
C354  C1        2520 SNDEND POP  BC
C355  DDE1      2530        POP  IX
C357  E1        2540        POP  HL
C358  D1        2550        POP  DE
C359  F1        2560        POP  AF
C35A  C9        2570        RET
                2580 ;
                2590 ;
                2600 ; **. NETREC: This CPCNET routine is the main receptor **
                2610 ; ** of data from receive channel of the PTC. It      **
                2620 ; ** receives the packet header and sees if what is   **
                2630 ; ** receiving is a BASIC program. If so then it will **
                2640 ; ** create the required  diversions in  jumpblocks   **
                2650 ; ** and prompt  user to issue a LOAD command. If the **
                2660 ; ** received data type is binary,   NETREC does the  **
                2670 ; ** job of storing  incoming data itself. NOTE: when **
                2680 ; ** receiving binary data no check is made  if the    **
                2690 ; ** area specified for the buffer is in use or not.  **
                2700 ;
                2710 ;
```

```
C35B  F5        2720 NETREC PUSH AF
C35C  E5        2730        PUSH HL
C35D  C5        2740        PUSH BC
C35E  D5        2750        PUSH DE ; Now we can use any of prime regs.
C35F  CD8DC5    2760        CALL RECHDR ; First we call header receive s/r
C362  DABCC4    2770        JP   C,RECEND ; If CTRL/A was pressed we abort
C365  FDCB4FD6  2780        SET  2,(IY+#4F) ; Set our activity bit in func byte
C369  FD7E50    2790        LD   A,(IY+#50) ; Get data type
C36C  FE01      2800        CP   #01        ; See if its BASIC
C36E  281F      2810        JR   Z,BASREC   ; Jump to BASIC receiver if so
C370  FD6E52    2820 BINREC LD   L,(IY+#52) ; Else HL=buffer start addr
C373  FD6E53    2830        LD   L,(IY+#53)
C376  FD5E53    2840        LD   E,(IY+#53) ; DE = Last buffer address
C379  FD5654    2850        LD   D,(IY+#54)
C37C  CD3EC5    2860 REC001 CALL RECEEV     ; Get a character
C37F  DABCC4    2870        JP   C,RECEND    ; If CTRL/A pressed then abort
C382  7E        2880        LD   A,(HL)      ; Else buffer the byte
C383  7B        2890        LD   A,E         ; See if low byte of bp
C384  BD        2900        CP   L           ; and end value match
C385  2005      2910        JR   NZ,REC002   ; Jump if no match in low byte
C387  7A        2920        LD   A,D         ; Else compare high bytes
C388  BC        2930        CP   H           ; See if high bytes match
C389  CABCC4    2940        JP   Z,RECEND    ; Jump if so
C38C  23        2950 REC002 INC  HL          ; Bump pointer
C38D  18ED      2960        JR   REC001      ; And loop again
C38F  FDCB4E46  2970 BASREC BIT  0,(IY+#4E) ; See if we have already patched
C393  206F      2980        JR   NZ,REC003   ; jump over patching section if so
C395  3A77BC    2990        LD   A,(#BC77)   ; BASIC program loader
C398  FD7704    3000        LD   (IY+#4),A   ; Save first byte of CAS IN OPEN
C39B  2A78BC    3010        LD   HL,(#BC78)  ; Get the other two bytes
C39E  FD7505    3020        LD   (IY+#5),L   ; Save them in wk area
C3A1  FD7406    3030        LD   (IY+#6),H
C3A4  3E05      3040        LD   A,ROMNUM    ; Get ROMnum into A
C3A6  FD7715    3050        LD   (IY+#15),A  ; Place it as third byte of far addr
C3A9  FDE5      3060        PUSH IY
C3AB  E1        3070        POP  HL          ; Copy IY to HL
C3AC  111300    3080        LD   DE,#13      ; Make address of far address in HL
C3AF  19        3090        ADD  HL,DE
C3B0  2278BC    3100        LD   (#BC78),HL  ; Place in jumpblock
C3B3  3EDF      3110        LD   A,#DF       ; And the RST 18
C3B5  3277BC    3120        LD   (#BC77),A   ; as well.
C3B8  3A7ABC    3130        LD   A,(#BC7A)   ; Save the next byte too
C3BB  FD7707    3140        LD   (IY+#7),A   ; Cos we have to put a RET in it
C3BE  3EC9      3150        LD   A,#C9       ; Which we duly put there
C3C0  327ABC    3160        LD   (#BC7A),A
C3C3  111FC5    3170        LD   DE,BAS001  ; DE points at BAS001 rtne
C3C6  73        3180        LD   (HL),E    ; Place BAS001 address in far address
C3C7  23        3190        INC  HL
C3C8  72        3200        LD   (HL),D
C3C9  3A80BC    3210        LD   A,(#BC80)   ; Now we patch CAS IN CHAR
C3CC  FD7708    3220        LD   (IY+#8),A   ; Save original bytes
C3CF  2A81BC    3230        LD   HL,(#BC81)
C3D2  FD7509    3240        LD   (IY+#9),L
C3D5  FD740A    3250        LD   (IY+#A),H
C3D8  3A83BC    3260        LD   A,(#BC83)   ; And the third byte as well
C3DB  FD770B    3270        LD   (IY+#B),A
C3DE  3EC9      3280        LD   A,#C9       ; so we can put a RET there
C3E0  3283BC    3290        LD   (#BC83),A
C3E3  2125C5    3300        LD   HL,RECBAS
C3E6  FD7516    3310        LD   (IY+#16),L  ; Now make the far address
C3E9  FD7417    3320        LD   (IY+#17),H  ; for the CAS IN CHAR replacement
C3EC  3E05      3330        LD   A,ROMNUM    ; and the ROMNUM completes far address
C3EE  FD7718    3340        LD   (IY+#18),A
C3F1  FDE5      3350        PUSH IY
C3F3  E1        3360        POP  HL        ; Copy IY to HL as before
C3F4  011600    3370        LD   BC,#16      ; BC is offset to far address location
C3F7  09        3380        ADD  HL,BC     ; Now HL points at far address address
C3F8  3EDF      3390        LD   A,#DF       ; Place RST 18 into jump block
```

```
C3FA  3280BC     3400         LD   (#BC80),A    ; entry for CAS IN CHAR
C3FD  2281BC     3410         LD   (#BC81),HL   ;
C400  FDCB4FFE   3420         SET  7,(IY+#4F)   ; Set special bit in func byte
                 3430 ; ** the jump block entries for cassette I/O are changed
C404  DD210EC4   3440 REC003  LD   IX,RECMES ; Now we must instruct how to rece
C408  CDF1C4     3450         CALL OUTPUT
C40B  C3BCC4     3460         JP   RECEND
C40E  0A0D       3470 RECMES  DEFB #0A,#0D
C410  2A2A2054   3480         DEFM "** The incoming data is to be fed into BASI
C447  0A0D       3490         DEFB #0A,#0D
C449  2A2A2074   3500         DEFM "** type a LOAD  or a MERGE  command to begi
C480  0A0D       3510         DEFB #0A,#0D
C482  2A2A2072   3520         DEFM "** reception of it.              CPCNET
C4B9  0A0DFF     3530         DEFB #0A,#0D,#FF
C4BC  D1         3540 RECEND  POP  DE ; Clear up the stack
C4BD  C1         3550         POP  BC
C4BE  E1         3560         POP  HL
C4BF  F1         3570         POP  AF
C4C0  FDCB4F96   3580         RES  2,(IY+#4F) ; Clear our activity bit in func
C4C4  C9         3590         RET  ; And return
                 3600 ;
                 3610 ;
                 3620 ; ** SNDHDR: This subroutine sends the contents **
                 3630 ; ** of the CPCNET header buffer over the PTC   **
                 3640 ; ** IY points to base of work area.
                 3650 ;
C4C5  F5         3660 SNDHDR  PUSH AF
C4C6  E5         3670         PUSH HL
C4C7  C5         3680         PUSH BC
C4C8  FDCB4FE6   3690         SET  4,(IY+#4F)      ; Set our active bit
C4CC  3EAA       3700         LD   A,#AA
C4CE  CD64C5     3710         CALL SEND ; Send the marker byte
C4D1  3816       3720         JR   C,HDR001 ; jump out for keypress
C4D3  FDE5       3730         PUSH IY
C4D5  E1         3740         POP  HL ; Copy IY to HL
C4D6  015000     3750         LD   BC,#50
C4D9  09         3760         ADD  HL,BC ; Now HL points to the header buffer
C4DA  0630       3770         LD   B,#30  ; B is the loop counter
C4DC  7E         3780 SNDLOP  LD   A,(HL) ; get a byte
C4DD  CD64C5     3790         CALL SEND     ; Send it
C4E0  3807       3800         JR   C,HDR001 ; If user pressed key then exit
C4E2  23         3810         INC  HL       ; Else bump pointer
C4E3  10F7       3820         DJNZ SNDLOP   ; Jump if not done yet
C4E5  3E00       3830         LD   A,0      ; Ensure carry clear at end
C4E7  CE00       3840         ADC  A,0
C4E9  C1         3850 HDR001  POP  BC
C4EA  E1         3860         POP  HL
C4EB  F1         3870         POP  AF
C4EC  FDCB4FA6   3880         RES  4,(IY+#4F) ; Clear active bit
C4F0  C9         3890         RET  ; Return
                 3900 ;
                 3910 ; ** The output routine outputs messages to the screen **
                 3920 ; ** IX points to the start of message, which ends with**
                 3930 ; ** a byte value of #FF. AF, IX ,HL corrupted.        **
C4F1  DD7E00     3940 OUTPUT  LD   A,(IX) ; Get a byte
C4F4  CB7F       3950         BIT  7,A    ; See if EOB marker
C4F6  C0         3960         RET  NZ     ; Return if so
C4F7  CD5ABB     3970         CALL #BB5A  ; Else output the chracter
C4FA  DD23       3980         INC  IX     ; bump buffer pointer
C4FC  18F3       3990         JR   OUTPUT ; and do it again
C4FE  67         4000 HEXAS   LD   H,A   ; Save original byte in H
C4FF  E60F       4010         AND  #0F   ; AND out the low nibble
C501  F630       4020         OR   #30   ; Convert to ASCII
C503  FE3A       4030         CP   #3A   ; see if its alpha
C505  FA0AC5     4040         JP   M,HIGH ; If not then jump over adjustment
C508  C607       4050         ADD  A,#7  ; make it HEX
C50A  6F         4060 HIGH    LD   L,A   ; Put finished low digit away in L
C50B  7C         4070         LD   A,H   ; Get out original byte
```

```
C50C  CB3F      4080        SRL  A      ; Make High nibble low
C50E  CB3F      4090        SRL  A
C510  CB3F      4100        SRL  A
C512  CB3F      4110        SRL  A
C514  F630      4120        OR   #30
C516  FE3A      4130        CP   #3A    ; See if Alpha
C518  FA1DC5    4140        JP   M,DONE ; Done if not
C51B  C607      4150        ADD  A,#7   ; Adjust as before
C51D  67        4160 DONE   LD   H,A    ; Save finished thing into H
C51E  C9        4170        RET  ; and finish
                4180 ;
                4190 ;
                4200 ; ** BAS001: This routine replaces the firmware CAS IN OPEN
                4210 ; ** routine during input of BASIC programs from the CPCNET
                4220 ; ** It just fools BASIC into thinking that it has opened
                4230 ; ** a BASIC program stored in ASCII sucessfully
                4240 ;
                4250 ;
C51F  3E16      4260 BAS001 LD   A,#16 ; Load ASCII filetype code
C521  FE00      4270        CP   #0 ; Ensure Z flag is clear
C523  37        4280        SCF  ; and Carry flag is set
C524  C9        4290        RET  ; and go back to BASIC
                4300 ;
                4310 ;
                4320 ;
                4330 ;
                4340 ; ** RECBAS: This CPCNET routine replaces the firmware
                4350 ; ** CAS IN CHAR routine, which feeds BASIC programs
                4360 ; ** fetched from cassette or disk to BASIC a character
                4370 ; ** at a time. This routine uses the RECEEV s/r and
                4380 ; ** flags end of file to BASIC if byte received has got
                4390 ; ** bit seven set.
                4400 ;
                4410 ;
C525  CD3EC5    4420 RECBAS CALL RECEEV ; Get a character
C528  3005      4430        JR   NC,RECB01 ; Jump if CTRL/A not hit by user
C52A  3E00      4440        LD   A,0    ; otherwise
C52C  CE00      4450        ADC  A, #0 ; Set zero and clear carry to tell
C52E  C9        4460        RET  ; BASIC that BREAK was hit and RET
C52F  CB7F      4470 RECB01 BIT  7,A ; See if top bit set
C531  2807      4480        JR   Z,RECB02 ; Jump if no
C533  3E00      4490        LD   A,0
C535  CE00      4500        ADC  A,#0
C537  3E1F      4510        LD   A,#1F ; Ensure carry andZ flag clr and
C539  C9        4520        RET  ; put EOF char into A then ret
C53A  FEFF      4530 RECB02 CP   #FF ; Ensure Z clear
C53C  37        4540        SCF  ; and carry set to tell BASIC its
C53D  C9        4550        RET  ; OK then return
                4560 ;
                4570 ;
                4580 ; ** RECEEV: This routine returns characters to
                4590 ; ** other software which have been received via the CPCNET
                4600 ; ** Parallel transfer channel. It waits for the IBF (input
                4610 ; ** buffer full) signal of port B to go high, which indicates
                4620 ; ** that a byte is available. While it is waiting it also
                4630 ; ** keeps calling a firmware  routine to see if CTRL/A has
                4640 ; ** been typed at the keyboard. If it has it returns with
                4650 ; ** the carry flag set. When a byte is read it is passed
                4660 ; ** back in the A register.
                4670 ;
                4680 ;
C53E  C5        4690 RECEEV PUSH BC
C53F  FDCB4FEE  4700        SET  5,(IY+#4F) ; Set active bit
C543  01F2F8    4710 RECEE2 LD   BC,PTCCNT     ; BC points to control channel
C546  ED78      4720        IN   A,(C)         ; Get control status
C548  CB4F      4730        BIT  1,A           ; See if IBF is true
C54A  200C      4740        JR   NZ,RECEE1     ; Jump if so
C54C  CD09BB    4750        CALL #BB09         ; else check for CTRL/A's
```

```
C54F  30F2      4760         JR   NC,RECEE2    ; If no key pressed then loop
C551  FE01      4770         CP   #1           ; else see if CTRL/A
C553  20EE      4780         JR   NZ,RECEE2    ; loop if no
C555  37        4790         SCF  ; if yes then set carry
C556  1806      4800         JR   RECOUT       ; and go  out
C558  01F1F8    4810 RECEE1  LD   BC,PTCREC    ; BC points at receive channel
C55B  ED78      4820         IN   A,(C)        ; get the character
C55D  BF        4830         CP   A            ; Ensure carry is clear.
C55E  C1        4840 RECOUT  POP  BC           ; Clean stack
C55F  FDCB4FAE  4850         RES  5,(IY+#4F) ; Clear active bit
C563  C9        4860         RET  ; And go back
                4870 ;
                4880 ;
                4890 ; ** SEND: This CPCNET routine sends the character it **
                4900 ; ** gets in the A reg out onto the PTC transmit channel **
                4910 ; ** but it first waits for the channel to be ready- as  **
                4920 ; ** indicated by a high on bit 7 or port C - PTCCNT.    **
                4930 ; ** If the user presses any key while SEND is waiting   **
                4940 ; ** then the carry flag is set and a return is done     **
                4950 ;
C564  C5        4960 SEND    PUSH BC
C565  E5        4970         PUSH HL
C566  FDCB4FF6  4980         SET  6,(IY+#4F) ; Set active bit
C56A  01F2F8    4990 SEND00  LD   BC,PTCCNT    ; Make BC point to PTC status register
C56D  ED68      5000         IN   L,(C)        ; Get status into L
C56F  CB7D      5010         BIT  7,L          ; See if send channel is ready
C571  200D      5020         JR   NZ,SEND01    ; Jump if yes
C573  F5        5030         PUSH AF           ; Save the byte for sending
C574  CD09BB    5040         CALL #BB09        ; see if user has typed anything
C577  3004      5050         JR   NC,SEND02    ; Jump if not
C579  F1        5060         POP  AF           ; Restore AF anyway
C57A  37        5070         SCF  ; Set carry
C57B  1809      5080         JR   SEND04       ; And end up
C57D  F1        5090 SEND02  POP  AF           ; If nowt typed then restore AF
C57E  18EA      5100         JR   SEND00       ; and loop round again
C580  01F0F8    5110 SEND01  LD   BC,PTCSND    ; Now BC points to send channel
C583  ED79      5120         OUT  (C),A        ; and send the byte out
C585  A7        5130         AND  A            ; Ensure carry flag is clear
C586  E1        5140 SEND04  POP  HL           ; POP the stack
C587  C1        5150         POP  BC
C588  FDCB4FB6  5160         RES  6,(IY+#4F)   ; Clear active bit
C58C  C9        5170         RET  ; And return
                5180 ;
                5190 ;
                5200 ;
                5210 ;
                5220 ; ** RECHDR: This routine receives a CPCNET header from the
                5230 ; ** receive channel of the PTC. The header must be in the
                5240 ; ** correct format, or a warning message is issued.
                5250 ;
                5260 ;
C58D  C5        5270 RECHDR  PUSH BC
C58E  E5        5280         PUSH HL
C58F  D5        5290         PUSH DE
C590  FDCB4FDE  5300         SET  3,(IY+#4F)     ; Set our function active bit
C594  FDE5      5310         PUSH IY           ; IY copied to HL
C596  E1        5320         POP  HL
C597  015000    5330         LD   BC,#50
C59A  09        5340         ADD  HL,BC          ; Now HL points to header buffer
C59B  CD3EC5    5350 RECH01  CALL RECEEV   ; Get a byte
C59E  DA37C6    5360         JP   C,RECH04   ; If user wants out then go out
C5A1  FEAA      5370         CP   #AA        ; See if it is the start marker
C5A3  28F6      5380         JR   Z,RECH01   ; loop if not
C5A5  CD3EC5    5390         CALL RECEEV   ; if so then reception begins
C5A8  5F        5400         LD   E,A        ; Validate the datatype (0 or 1 only)
C5A9  E6FE      5410         AND  #FE       ; Mask out illegal bits
C5AB  287D      5420         JR   Z,RECH02   ; If no unused bits set then jump
C5AD  DD21B7C5  5430         LD   IX,RECHER ; Else issue a warning message
```

```
C5B1  CDF1C4    5440          CALL OUTPUT
C5B4  C337C6    5450          JP   RECH04    ; and then abort reception
C5B7  0A0D      5460 RECHER DEFB #0A,#0D
C5B9  2A2A2057  5470          DEFM "** Warning message: **"
C5EF  0A0D      5480          DEFB #0A,#0D
C5F1  2A2A2068  5490          DEFM "** header reception aborted **"
C627  0A0DFF    5500          DEFB #0A,#0D,#FF
C62A  73        5510 RECH02 LD   (HL),E ; Put datatype into buffer
C62B  23        5520          INC  HL ; Bump the buffer pointer
C62C  061B      5530          LD   B,#1B     ; 20 more bytes to get for header
C62E  CD3EC5    5540 RECH03 CALL RECEEV  ; Get a byte
C631  3804      5550          JR   C,RECH04  ; Abort if user wants to
C633  77        5560          LD   (HL),A    ; Buffer it
C634  23        5570          INC  HL       ; update pointer
C635  10F7      5580          DJNZ RECH03  ; loop till done
C637  D1        5590 RECH04 POP  DE
C638  E1        5600          POP  HL
C639  C1        5610          POP  BC        ; Restore entry contents
C63A  FDCB4F9E  5620          RES  3,(IY+#4F) ; Clear active flag
C63E  C9        5630          RET  ; and go back

Pass 2 errors: 00

BAS001 C51F  BASREC C38F  BINREC C370  BUMPAR C27F
CLS001 C236  CLSEND C264  DONE   C51D  HDR001 C4E9
HEXAS  C4FE  HIGH   C50A  INIMES C05F  MAKHDR C0C8
NAMTAB C015  NETCLO C214  NETGO  C03E  NETOPN C082
NETREC C35B  NETSND C26B  OPEND  C0F5  OPN009 C0E4
OPNEND C20D  OUTPUT C4F1  PPICTL F8F3  PTCCNT F8F2
PTCREC F8F1  PTCSND F8F0  REC001 C37C  REC002 C38C
REC003 C404  RECB01 C52F  RECB02 C53A  RECBAS C525
RECEE1 C558  RECEE2 C543  RECEEV C53E  RECEND C4BC
RECH01 C59B  RECH02 C62A  RECH03 C62E  RECH04 C637
RECHDR C58D  RECHER C5B7  RECMES C40E  RECOUT C55E
ROMNUM 0005  SEND   C564  SEND00 C56A  SEND01 C580
SEND02 C57D  SEND04 C586  SND001 C2DB  SND002 C2FD
SND003 C31B  SND004 C275  SNDEND C354  SNDHDR C4C5
SNDLOP C4DC  SUCESS C337

Table used:   758  from  2415
```

Fig. 6.8

The Hexdump of the CPCNET ROM is shown in Fig.6.9

```
    Byte values (hex)

01 01 01 01 15 C0 C3 3E C0 C3 82 C0 C3 14 C2 C3
6B C2 C3 5B C3 4E 45 54 20 49 4E 49 D4 4E 45 54
2E 4F 50 45 CE 4E 45 54 2E 43 4C 4F 53 C5 4E 45
54 2E 53 45 4E C4 4E 45 54 2E 52 45 C3 00 F5 C5
25 01 F3 F8 3E AF ED 79 3E 00 FD 77 4E FD 77 4F
DD E5 DD 21 5F C0 CD F1 C4 DD E1 C1 F1 37 C9 0A
0D 2A 2A 20 43 50 43 4E 45 54 20 20 56 31 2E 31
20 69 6E 69 74 69 61 6C 69 73 65 64 20 2A 2A 0A
0D FF F5 E5 FD CB 4E 46 20 3E FD CB 4E C6 FD CB
4F C6 2A 2B BD FD 75 00 FD 74 01 2A 2D BD FD 75
02 FD 74 03 3E DF 32 2B BD 21 64 C5 FD 75 10 FD
74 11 26 05 FD 74 12 FD E5 E1 D5 11 10 00 19 D1
22 2C BD 3E C9 32 2E BD 3E 01 FD 77 50 3E 00 FD
77 51 FD 77 52 FD 77 53 FD 77 54 FD E5 E1 01 55
00 09 06 10 77 23 10 FC CD C5 C4 DD 21 F5 C0 CD
```

```
F1 C4 C3 0D C2 0A 0D 2A 2A 20 43 50 43 4E 45 54
20 69 73 20 6E 6F 77 20 6F 70 65 6E 65 64 20 66
6F 72 20 79 6F 75 20 74 6F 20 73 65 6E 64 20 42
41 53 49 43 20 2A 2A 0A 0D 2A 2A 20 70 72 6F 67
72 61 6D 73 20 6F 72 20 76 61 72 69 61 62 6C 65
73 20 6F 6E 20 73 74 72 65 61 6D 20 38 2E 20 59
6F 75 20 63 61 6E 20 2A 2A 0A 0D 2A 2A 20 75 73
65 20 74 68 65 20 4C 49 53 54 20 23 38 20 6F 72
20 50 52 49 4E 54 20 23 38 20 73 74 61 74 6D 65
6E 74 73 20 74 6F 20 64 6F 20 74 68 69 73 2E 20
20 20 2A 2A 0A 0D 2A 2A 20 77 68 65 6E 20 79 6F
75 20 68 61 76 65 20 66 69 6E 69 73 68 65 64 20
73 65 6E 64 69 6E 67 2C 20 69 73 73 75 65 20 61
20 20 7C 4E 45 54 2E 43 4C 4F 53 45 20 2A 2A 0A
0D 2A 2A 20 63 6F 6D 6D 61 6E 64 20 74 6F 20 74
65 72 6D 69 6E 61 74 65 20 74 68 65 20 74 72 61
6E 73 6D 69 73 73 69 6F 6E 20 70 72 6F 70 65 72
6C 79 2E 20 20 20 20 20 2A 2A 0A 0D FF E1 F1 FD
CB 4F 86 C9 E5 F5 FD CB 4E CE FD CB 4E 46 28 16
FD CB 4E 86 FD 6E 00 FD 66 01 22 2B BD FD 6E 02
FD 66 03 22 2D BD FD CB 4F 7E 28 28 FD CB 4F BE
FD 6E 04 FD 66 05 22 77 BC FD 6E 06 FD 66 07 22
79 BC FD 6E 08 FD 66 09 22 80 BC FD 6E 0A FD 66
0B 22 82 BC F1 E1 FD CB 4E 8E C9 F5 D5 E5 DD E5
C5 FE 02 28 66 DD 21 7F C2 CD F1 C4 C3 54 C3 2A
2A 20 59 6F 75 20 6D 75 73 74 20 73 70 65 63 69
66 79 20 73 74 61 72 74 20 61 6E 64 20 65 6E 64
20 2A 2A 0D 0A 2A 2A 20 61 64 64 72 65 73 73 65
73 20 66 6F 72 20 73 65 6E 64 69 6E 67 2E 20 45
67 2D 7C 4E 45 54 2E 53 45 4E 44 2C 31 30 30 30
2C 32 30 30 30 20 2A 2A 0A 0D FF FD E5 E1 01 50
00 09 36 00 23 DD 7E 02 77 DD 7E 03 23 77 DD 7E
00 23 77 DD 7E 01 23 77 23 06 10 3E 20 77 23 10
FC CD C5 C4 FD 6E 51 FD 66 52 FD 4E 53 FD 46 54
ED 42 D2 75 C2 FD 6E 51 FD 66 52 7E CD 64 C5 38
33 23 7D FD BE 53 20 F3 7C FD BE 54 20 ED DD 21
37 C3 CD F1 C4 18 1D 0A 0D 2A 2A 20 54 72 61 6E
73 66 65 72 20 63 6F 6D 70 6C 65 74 65 64 20 2A
2A 0A 0D FF C1 DD E1 E1 D1 F1 C9 F5 E5 C5 D5 CD
8D C5 DA BC C4 FD CB 4F D6 FD 7E 50 FE 01 28 1F
FD 6E 52 FD 6E 53 FD 5E 53 FD 56 54 CD 3E C5 DA
BC C4 7E 7B BD 20 05 7A BC CA BC C4 23 18 ED FD
CB 4E 46 20 6F 3A 77 BC FD 77 04 2A 78 BC FD 75
05 FD 74 06 3E 05 FD 77 15 FD E5 E1 11 13 00 19
22 78 BC 3E DF 32 77 BC 3A 7A BC FD 77 07 3E C9
32 7A BC 11 1F C5 73 23 72 3A 80 BC FD 77 08 2A
81 BC FD 75 09 FD 74 0A 3A 83 BC FD 77 0B 3E C9
32 83 BC 21 25 C5 FD 75 16 FD 74 17 3E 05 FD 77
18 FD E5 E1 01 16 00 09 3E DF 32 80 BC 22 81 BC
FD CB 4F FE DD 21 0E C4 CD F1 C4 C3 BC C4 0A 0D
2A 2A 20 54 68 65 20 69 6E 63 6F 6D 69 6E 67 20
64 61 74 61 20 69 73 20 74 6F 20 62 65 20 66 65
64 20 69 6E 74 6F 20 42 41 53 49 43 2E 20 50 6C
65 61 73 65 20 2A 2A 0A 0D 2A 2A 20 74 79 70 65
20 61 20 4C 4F 41 44 20 20 6F 72 20 61 20 4D 45
52 47 45 20 20 63 6F 6D 6D 61 6E 64 20 74 6F 20
62 65 67 69 6E 20 20 20 20 20 20 20 20 20 2A 2A
0A 0D 2A 2A 20 72 65 63 65 70 74 69 6F 6E 20 6F
66 20 69 74 2E 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 43 50 43 4E 45 54 20 20 20 20 20
20 20 20 20 20 20 20 2A 2A 0A 0D FF D1 C1 E1 F1
FD CB 4F 96 C9 F5 E5 C5 FD CB 4F E6 3E AA CD 64
C5 38 16 FD E5 E1 01 50 00 09 06 30 7E CD 64 C5
38 07 23 10 F7 3E 00 CE 00 C1 E1 F1 FD CB 4F A6
C9 DD 7E 00 CB 7F C0 CD 5A BB DD 23 18 F3 67 E6
0F F6 30 FE 3A FA 0A C5 C6 07 6F 7C CB 3F CB 3F
CB 3F CB 3F F6 30 FE 3A FA 1D C5 C6 07 67 C9 3E
```

```
16 FE 00 37 C9 CD 3E C5 30 05 3E 00 CE 00 C9 CB
7F 28 07 3E 00 CE 00 3E 1F C9 FE FF 37 C9 C5 FD
CB 4F EE 01 F2 F8 ED 78 CB 4F 20 0C CD 09 BB 30
F2 FE 01 20 EE 37 18 06 01 F1 F8 ED 78 BF C1 FD
CB 4F AE C9 C5 E5 FD CB 4F F6 01 F2 F8 ED 68 CB
7D 20 0D F5 CD 09 BB 30 04 F1 37 18 09 F1 18 EA
01 F0 F8 ED 79 A7 E1 C1 FD CB 4F B6 C9 C5 E5 D5
FD CB 4F DE FD E5 E1 01 50 00 09 CD 3E C5 DA 37
C6 FE AA 28 F6 CD 3E C5 5F E6 FE 28 7D DD 21 B7
C5 CD F1 C4 C3 37 C6 0A 0D 2A 2A 20 57 61 72 6E
69 6E 67 20 6D 65 73 73 61 67 65 3A 20 52 45 43
48 44 52 20 64 65 74 65 63 74 73 20 69 6C 6C 65
67 61 6C 20 64 61 74 61 74 79 70 65 20 2A 2A 0A
0D 2A 2A 20 68 65 61 64 65 72 20 72 65 63 65 70
74 69 6F 6E 20 61 62 6F 72 74 65 64 20 2D 20 74
65 6C 6C 20 73 65 6E 64 65 72 20 20 20 20 20 20
20 20 20 20 20 2A 2A 0A 0D FF 73 23 06 1B CD 3E
C5 38 04 77 23 10 F7 D1 E1 C1 FD CB 4F 9E C9
```

Fig. 6.9

CPCNET conclusion

CPCNET as presented here has many possibilities for expansion, and I hope that readers will find it easy to expand and customise to local needs. If a second version is ever produced it is hoped to add a few more features to it, like interactive display of incoming data for example. I should be glad to hear from readers who develope any novel uses for CPCNET, though I cannot undertake to reply.

Chapter six conclusions

In the end only CPCNET found its way into this chapter. There was going to be another big project included too, but that proved too complex to finish for even the greatly extended publication deadline! Because the CPC machines are so capable and due to the firmware being so well documented it is an inviting prospect to develop complex application projects for them. I look forward to seeing many more in future publications.

APPENDIX ONE

Circuit symbols and general constructional notes.

Readers with little or no experience of computer hardware, are strongly advised to read appendix six, which is entitled "begin here". This will go some way to preparing you for the material in the rest of the book. Because this is not a book aimed specifically at the beginner, appendix six is adequate as a general introduction, but you would be well advised to fill in the gaps by taking out a regular subscription to one of the many electronics magazines which you will find on sale at newsagents, or at your local electronic components shop.

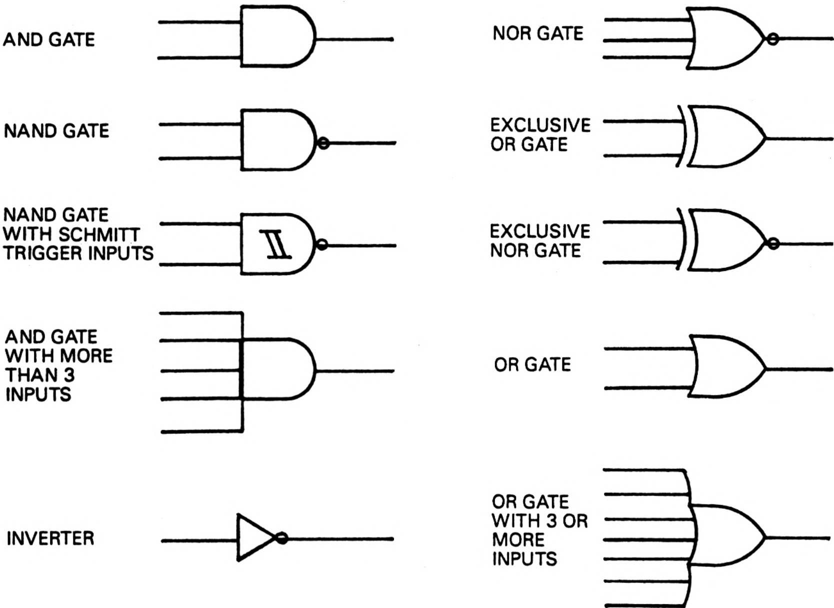
The many circuit diagrams in this book have been drawn using logic element symbols which conform to the American Mil spec. Fig. APl.1 shows these symbols. There are many other symbol sets around, as the reader will find when reading magazines, but the Mil spec ones seem to be most widely used. An additional advantage is that the Amstrad technical documentation also uses these symbols – so compatibility of documentation is maintained.

For each hardware add on project in this book you will find:

1) The circuit diagram or diagrams
2) An exhaustive description of the circuit operation, how it can be used, and any optional extras available.
3) A parts list for the project
4) A list of power connections to each of the chips used
5) Listings of software to drive the project(where appropriate)

The circuit diagrams have all been carefully checked, and are believed to be free of error. The descriptions are designed to give you a complete understanding of how the project works. The parts lists specify all the parts you need to build each project. You will notice that I have given RS Components catalogue numbers in many instances, this is not because I have shares in RS, but because the RS catalogue is to be found in almost every electronic component retailers. If a part is out of stock the retailer can use the description obtained from the RS catalogue to find an equivalent. Please be careful when using equivalent components, especially if you are

Figure APl.1 Logic Symbols used in this book



inexperienced. If you are in any doubt consult someone who can competently advise you. We want to enhance your computer, not blow it up! The list of power connections to the chips used in each project includes a brief functional description of the chip. This should form a ready reference for you when you are reading the circuit diagram for that project. Finally, the software. Where feasible I have tried to use BASIC to drive the projects. This is because I did not want to exclude readers who have not got an assembler package like DEVPAC. There are several projects however, where for reasons of speed or facility, it is not feasible to use BASIC. In these cases machine code programs have been developed. All is not lost for BASIC only readers however, in chapter five you will find a means for converting short machine code programs into BASIC code lists. In all cases the software has been tested as completely as possible, and any bugs fixed. If you find any new bugs please let me know!

The overall scheme for connection to the CPC 464 hinges upon the use of an expansion bus motherboard which has been especially created by Halstead Designs to go with this book. Fig. APl.2 shows how the motherboard fits into the overall scheme. The expansion bus connections are available at the far end of the motherboard, so that you can plug in your disk drive or other peripherals. The expansion bus and various other connectional details are explained in appendix five.

On the subject of power supplies Amstrad informs me that the +5 volt line extended on the expansion bus connector has only 100 milliamps or so of spare capacity. For this reason and the fact that voltages other than +5 volts are required for the RS 232 projects, the SIGMA power supply unit must be added. (Details of the power supply are provided at the start of chapter three)

Appendix three contains reproductions of the manufacturer's data sheets, not only on the major chips used in the CPC 464 but also on the major devices used in the projects presented in this book.

APPENDIX 2

Further reading

The following is a short list of books which are on related subjects to this one. The opinions expressed about them are my own.

1. **Programming the Z80** by Rodnay Zaks. Published 1982 by Sybex.

This book is widely referred to as the definitive work on the Z80 microprocessor. I might take issue with the word Definitive, but it is true that the instruction set description is excellent. The book also contains a great many subroutines and shows techniques on I/O and programming. Throughout the book there are excercises to test your comprehension of the subject matter, and also there are a lot of descriptions of the Z80 support chips. I personallly prefer the Barden book, but due to the excellent instruction set section of the Zaks book, would recommend that you get them both!

2. **The Z80 micrcomputer handbook** by William Barden, jr. Published 1980 by Sams publications.

This is my favourite book on the Z80. It goes deeper into the actual signal timings and general hardware matters than the Zaks book. For example it describes the interface to a disk controller in great detail and it gives a good deal of interfacing information. It is also a fairly slim book, which makes things easier to find. The instruction set is adequately – though not comprehensively – explained. Get this book first, then get the Zaks book.

3. **Build your own Z80 computer** by Steve Ciarcia. Published 1981 by Byte books.

Ciarcia apparently writes for Byte magazine (or did in 1981), and I think that this book reflects his experience in the magazine field. The book takes you step by step through designing a computer with a Z80 as its CPU. There is a description of the Z80 and its instruction set, then descriptions of how to connect memory, a hex keyboard, then make various peripheral devices. The finished machine would be far below the CPC level, but the experience of making and understanding it would give you a level of knowledge which would fit you for any other hardware or system project you could care to name.

On the minus side, the chips used in the book are now rather outmoded (although commercial equipment containing them continues to appear!), but they still function in broadly the same way, and can be bought very cheaply

now. The section on power supply design is good, and very importantly there is the assembly listing of a small monitor program to enable you to use the hex keypad and minimal computer for machine code programming. Copious appendices give general information.

4. **The Amstrad CPC 464 Advanced user guide** by Mark Harrison. Published 1984 by Sigma Technical Press.

This is Sigma's companion volume to the one you now have in your hands, and it covers all the software side of the CPC 464. BASIC, data stuctures, binary coding, memory map for the CPC 464, and much more are to be found in this highly recommended, and invaluable volume.

5. **SOFT 158. The CPC firmware specification.** Published by and available from Amstrad.

If you want to do anything more worthwhile than play games on your CPC you should get this book. It gives you chapter and verse on how the machine runs, and also all its internal "Secrets". There are hundreds of system routines detailed, with entry and exit conditions described for each. GET THIS BOOK!

If some of my passion for hardware has rubbed off on you whilst reading this book, then it is very likely that you will want to keep up to date on all the new devices, with an eye to incorporating them in your own designs. The best way to do this is by getting manufacturers data books. Another way to accumulate information on chips is to subscribe to an electronics magazine, such as Practical Electronics, ETI, Elektor, Hobby electronics, Wireless and Electronics World, Byte magazine and so on. These regularly feature new device descriptions and practical designs for them. Finally, one of the cheapest ways to get information on chips is from mail order retailers catalogues. Look in the hobby press on how to get hold of a Cirkit, RS, or Farnell catalogue – they all contain useful data on chips and the whole range of components applicable to computer designs.

APPENDIX 3

Useful Chip Data

GENERAL INSTRUMENT

SP0256

Narrator™ Speech Processor

- Natural Speech
- Stand Alone Operation with Inexpensive Support Components
- Wide Operating Voltage
- Word, Phrase, or Sentence Library, ROM Expandable
- Expandable to 491K of ROM Directly
- Simple Interface to Most Microcomputers or Microprocessors
- Supports L.P.C. Synthesis: Formant Synthesis: Allophone Synthesis

GENERAL DESCRIPTION

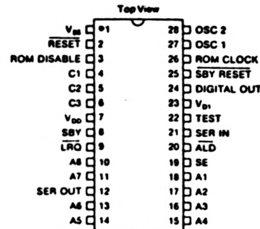
The SP0256 (Speech Processor) is a single chip N-Channel MOS LSI device that is able, using its stored program, to synthesize speech or complex sounds.

The achievable output is equivalent to a flat frequency response ranging from 0 to 5KHz, a dynamic range of 42dB, and a signal to noise ratio of approximately 35dB.

The SP0256 incorporates four basic functions:

- A software programmable digital filter that can be made to model a VOCAL TRACT.
- A 16K ROM which stores both data and instructions (THE PROGRAM).
- A MICROCONTROLLER which controls the data flow from the ROM to the digital filter, the assembly of the "word strings" necessary for linking speech elements together, and the amplitude and pitch information to excite the digital filter.
- A PULSE WIDTH MODULATOR that creates a digital output which is converted to an analog signal when filtered by an external low pass filter.

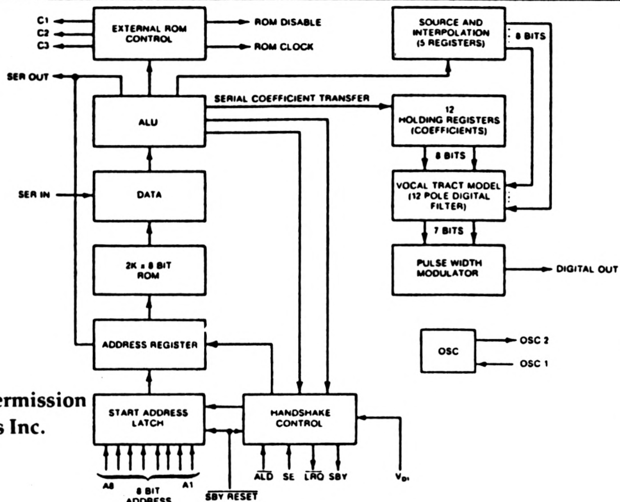
PIN CONFIGURATION
28 LEAD DUAL IN LINE



APPLICATIONS

- Telecommunications
- Appliances
- Computer Peripherals
- Automotive
- Personal Computers
- Toys/Games
- Educational Aids
- Warning Systems
- Security Systems
- Electronic Musical Instruments
- Aids to the Blind
- Narrow Bandwidth
- Communication Systems

SP0256 BLOCK DIAGRAM



Reproduced by kind permission of General Instruments Inc.

Gothic Crellon Limited
380 Bath Road Slough Berks SL1 6JE England
Telephone Burnham (06286) 4300 telex 847571

Sales telephone Burnham (06286) 4434 telex 847571
Birmingham 021 643 6365 telex 338731

MC6845

Advance Information

CRT CONTROLLER (CRTC)

The MC6845 CRT Controller performs the interface to raster scan CRT displays. It is intended for use in processor-based controllers for CRT terminals in stand-alone or cluster configurations.

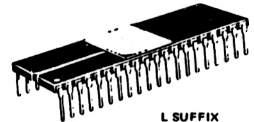
The CRTC is optimized for hardware/software balance in order to achieve integration of all key functions and maintain flexibility. For instance, all keyboard functions, R/W, cursor movements, and editing are under processor control; whereas the CRTC provides video timing and Refresh Memory Addressing.

- Applications include "glass-teletype," smart, programmable, intelligent CRT terminals; video games; information display.
- Alphanumeric, semi-graphic, and full graphic capability.
- Fully programmable via processor data bus. Can generate timing for almost any alphanumeric screen density, e.g. 80 x 24, 72 x 64, 132 x 20, etc.
- Single +5 volt supply. TTL/6800 compatible I/O.
- Hardware scroll (paging or by line or by character)
- Compatible with CPU's and MPU's which provide a means for synchronizing external devices.
- Cursor register and compare circuitry.
- Cursor format and blink are programmable.
- Light pen register.
- Line buffer-less operation. No external DMA required. Refresh Memory is multiplexed between CRTC and MPU.
- Programmable interlace or non-interlace scan.
- 14-bit wide refresh address.

MOS

(N-Channel, Silicon-Gate)

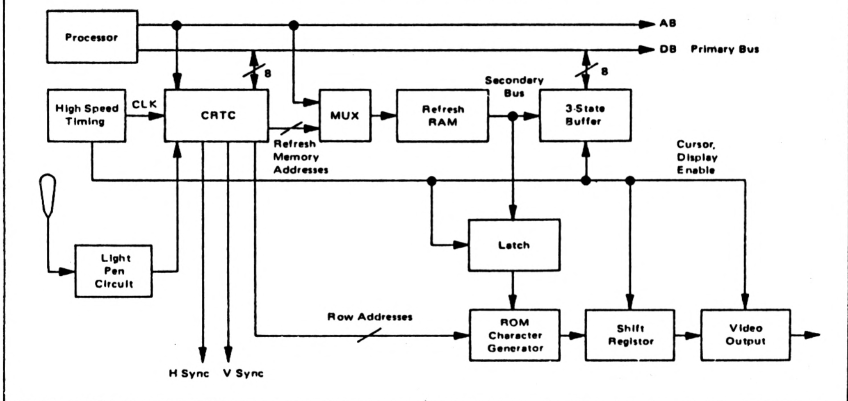
CRT CONTROLLER (CRTC)



L SUFFIX
CERAMIC PACKAGE
CASE 715

NOT SHOWN:
P SUFFIX
PLASTIC PACKAGE
CASE 711

FIGURE 1 – TYPICAL CRT CONTROLLER APPLICATION



Reproduced by kind permission of Motorola Inc.

8255A/8255A-5
PROGRAMMABLE PERIPHERAL INTERFACE

- MCS-85™ Compatible 8255A-5
- 24 Programmable I/O Pins
- Completely TTL Compatible
- Fully Compatible with Intel® Microprocessor Families
- Improved Timing Characteristics
- Direct Bit Set/Reset Capability Easing Control Application Interface
- 40-Pin Dual In-Line Package
- Reduces System Package Count
- Improved DC Driving Capability

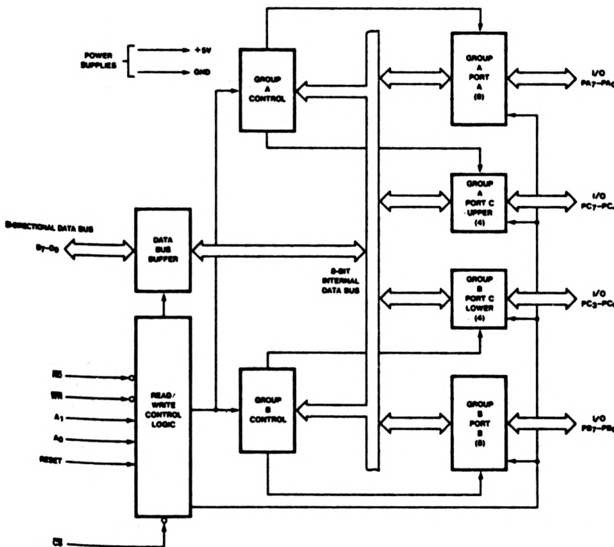


Figure 1. 8255A Block Diagram

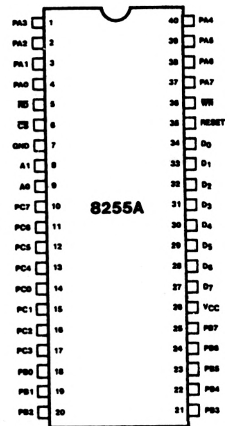


Figure 2. Pin Configuration

Reproduced by kind permission of Intel Corp.

Z8400
Z80® CPU Central Processing Unit

Zilog

Product Specification

September 1983

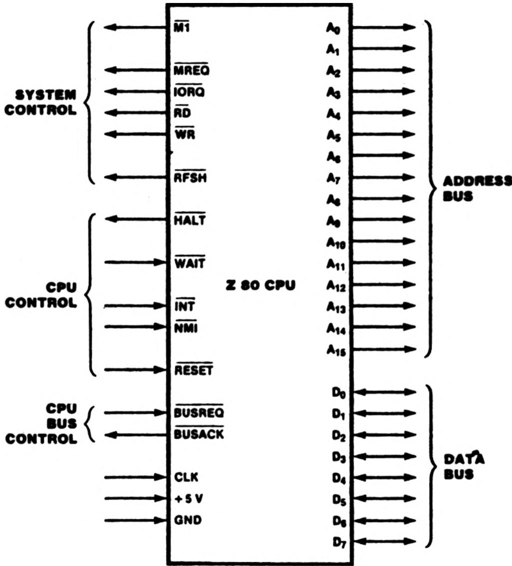


Figure 1. Pin Functions

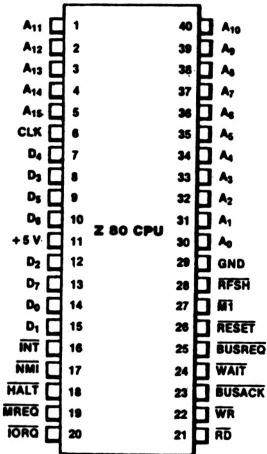


Figure 2. Pin Assignments

Reproduced by kind permission of Zilog Inc.

8251A
PROGRAMMABLE COMMUNICATION INTERFACE

- **Synchronous and Asynchronous Operation**
- **Synchronous 5–8 Bit Characters; Internal or External Character Synchronization; Automatic Sync Insertion**
- **Asynchronous 5–8 Bit Characters; Clock Rate—1, 16 or 64 Times Baud Rate; Break Character Generation; 1, 1½, or 2 Stop Bits; False Start Bit Detection; Automatic Break Detect and Handling**
- **Synchronous Baud Rate—DC to 64K Baud**
- **Asynchronous Baud Rate—DC to 19.2K Baud**
- **Full-Duplex, Double-Buffered Transmitter and Receiver**
- **Error Detection—Parity, Overrun and Framing**
- **Compatible with an Extended Range of Intel Microprocessors**
- **28-Pin DIP Package**
- **All Inputs and Outputs are TTL Compatible**
- **Single +5V Supply**
- **Single TTL Clock**

The Intel® 8251A is the enhanced version of the industry standard, Intel 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART), designed for data communications with Intel's microprocessor families such as MCS-48, 80, 85, and iAPX-86, 88. The 8251A is used as a peripheral device and is programmed by the CPU to operate using virtually any serial data transmission technique presently in use (including IBM "bi-sync"). The USART accepts data characters from the CPU in parallel format and then converts them into a continuous serial data stream for transmission. Simultaneously, it can receive serial data streams and convert them into parallel data characters for the CPU. The USART will signal the CPU whenever it can accept a new character for transmission or whenever it has received a character for the CPU. The CPU can read the complete status of the USART at any time. These include data transmission errors and control signals such as SYNDET, TxEMPTY. The chip is fabricated using N-channel silicon gate technology.

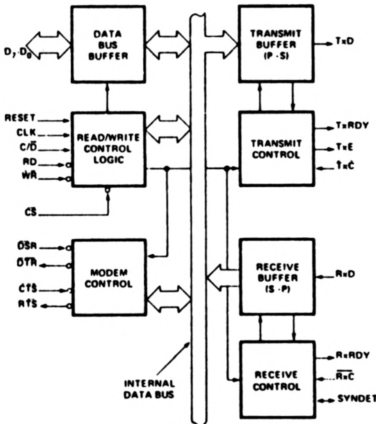


Figure 1. Block Diagram

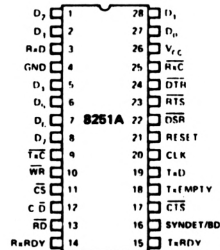


Figure 2. Pin Configuration

©INTEL CORPORATION, 1981

MAY 1981
AFN-01573C

Reproduced by kind permission of Intel Corporation

Gothic Crellon Limited

380 Bath Road Slough Berks SL1 6JE England
Telephone Burnham (06286) 4300 telex 847571

Sales telephone Burnham (06286) 4434 telex 847571
Birmingham 021 643 6365 telex 338731

MC1488

QUAD LINE DRIVER

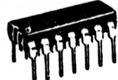
The MC1488 is a monolithic quad line driver designed to interface data terminal equipment with data communications equipment in conformance with the specifications of EIA Standard No. RS-232C.

Features:

- Current Limited Output
  ±10 mA typ
- Power-Off Source Impedance
  300 Ohms min
- Simple Slew Rate Control with External Capacitor
- Flexible Operating Supply Range
- Compatible with All Motorola MDTL and MTTL Logic Families

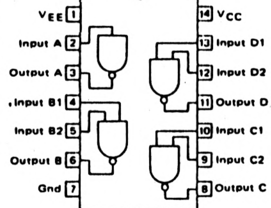
QUAD MDTL LINE DRIVER
RS-232C
SILICON MONOLITHIC INTEGRATED CIRCUIT

L SUFFIX
CERAMIC PACKAGE
CASE 632
TO-116

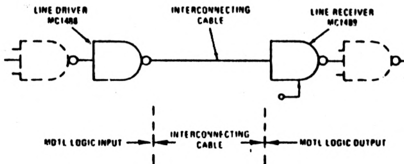


P SUFFIX
PLASTIC PACKAGE
CASE 646

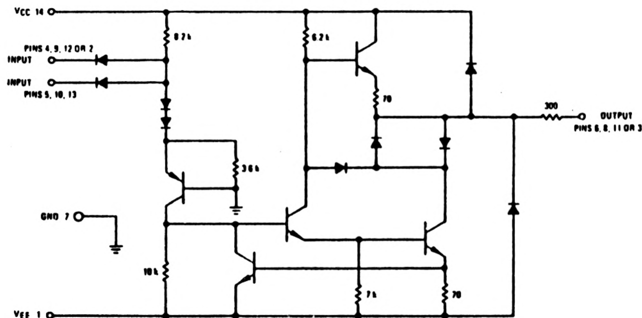
PIN CONNECTIONS



TYPICAL APPLICATION



CIRCUIT SCHEMATIC
(1/4 OF CIRCUIT SHOWN)



MDTL, MHTL, MRTL, MTTL are trademarks of Motorola Inc.

© MOTOROLA INC. 1981

DS9162R2

Reproduced by kind permission of Motorola Semiconductors Inc.

APPENDIX 4

Miscellaneous Data: Pin Connections and ASCII Character Set

Pin connections for RS 232 25 way D plugs

Pin	*Name*	*Function*	*Direction*
1	FG	Frame ground	–
2	TD	Transmitted data	DTE to dce
3	RD	Received data	DCE to DTE
4	RTS	Request to send	DTE to DCE
5	CTS	Clear to send	DCE to DTE
6	DSR	Data set ready	DCE to DTE
7	SG	Signal ground	–
8	DCD	Data carrier detect	DCE to DTE
9	–	Positive DC test volt	DCE to DTE
10	–	Negative DC test	DCE to DTE
11	–	–	–
12	(S)DCD	Secondary DCD	DCE to DTE
13	(S)CTS	Secondary CTS	DCE to DTE
14	(S)TD	Secondary TD	DTE to DCE
15	TC	Transmitter clock	DCE to DTE
16	(S)RD	Secondary RD	DCE to DTE
17	RC	Receiver clock	DCE to DTE
18	RDC	Receiver dibit clk	DTE to DCE
19	(S)RTS	Secondary RTS	DTE to DCE
20	DTR	Data terminal ready	DTE to DCE
21	SQD	Signal quality detect	DCE to DTE
22	RI	Ring indicator	DCE to DTE
23	DRS	Data rate select	DTE to DCE
24	ETX	External transmit clk	DTE to DCE
25	BSY	Busy	DTE to DCE

DTE = Data Terminal Equipment

DCE = Communications Equipment (for example a MODEM)

NOTE

Because of the customisation of this "Standard" over a long period of time readers may find that some terminals or communications equipment does not use some of these pins as indicated. ALWAYS check the manual for the equipment you are interconnecting, before plugging in.

ASCII Character set

LSD:›	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
MSD:	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	‹	=	›	0
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0
5	P	Q	R	S	T	U	V	W	X	Y	Z	«	/	»	ˆ	-
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	«	|	»	~	DEL

Control Codes:

NUL= Fill character : SOH=Start of Header : STX=Start Transmission ETX=End of TeXT : EOT=End of Transmission : ENQ=ENquiry ACK=ACKnowledge : BEL=Bell : BS=Backspace : HT=Horizontal Tab LF=Line Feed : VT Vertical Tab : FF=Form Feed : CR=Carriage Return SO=Shift Out : SI=Shift In : DLE=Data Link Escape : DC1–DC4=Device Control 1–4 : NAK=Negative AcKnowledge : SYN=SYNchronus idle Character : ETB=End of Transmitted Block CAN=Cancel : EM=End of Medium : SUB=SUBstitute : ESC=ESCape US=Unit Separator : SP=SPace : DEL=DELete (also called rubout)

APPENDIX 5

The CPC expansion bus

The extended expansion bus:

The 50 way expansion bus is available at the edge connector marked, "Floppy disc", on the rear of your CPC. The expansion bus has on it all the signals required to add all manner of peripherals to the basic computer. This includes a 5 Volt supply line, ground connections , and the master clock signal which is used to clock the Z80A microprocessor. There are however, several problems, when we need to connect any appreciable number of peripherals to the expansion bus. When there are peripherals present which need power supply voltages like; +12 Volts, or −12 Volts, the CPC expansion bus cannot supply them, – its Power Supply Unit (PSU) only produces +5 Volts. This means that an expansion board like an RS232 serial interface, will have to derive its +12 Volt and – 12 Volt supplies from a source other than the expansion bus. The other problem is that, although two of the 50 lines on the expansion bus are designated as ground connections, only line 27 is used to supply +5 Volts from the computer to external devices. The connection between the CPC and external hardware, will almost certainly be made via a standard ribbon cable, rated at about 1 amp per conductor. When a few external boards are plugged in, this amperage will be exceeded quite easily, resulting in line 27 becoming overloaded, and thus resistive. The net result of this will be that you will get less than 5 Volts delivered to the chips on the external boards. This will result in unreliable operation at best, and total inoperation at worst.

The development of the projects detailed in this book, has led to the design of a 64 way motherboard. This motherboard will be available from the PCB suppliers– see page three. The motherboard will accomodate up to eight plug in expansion boards. The 50 way expansion bus is carried through the motherboard. This will allow your disk drive,or MODEM option, to be plugged into the 50 way edge plug at the far end of the motherboard. A 64 way bus has been selected for several reasons, the main one being to allow the connection on the same bus of the 50 expansion bus signals , and extra voltage rails, using the remaining 14 spare lines. An additional benefit is that it also allows for using several conductors for the supplies and grounds. These extra power supply voltages, and an independent +5 Volt supply, are provided by the SIGMA power supply unit. The details of this are given at the beginning of chapter three. Figures AP5.1, AP5.2, and AP5.3 show the connectional layout of all the units and components involved. These are the CPC, the SIGMA PSU, the motherboard, and the Amstrad DD1 disk drive (If you have one).

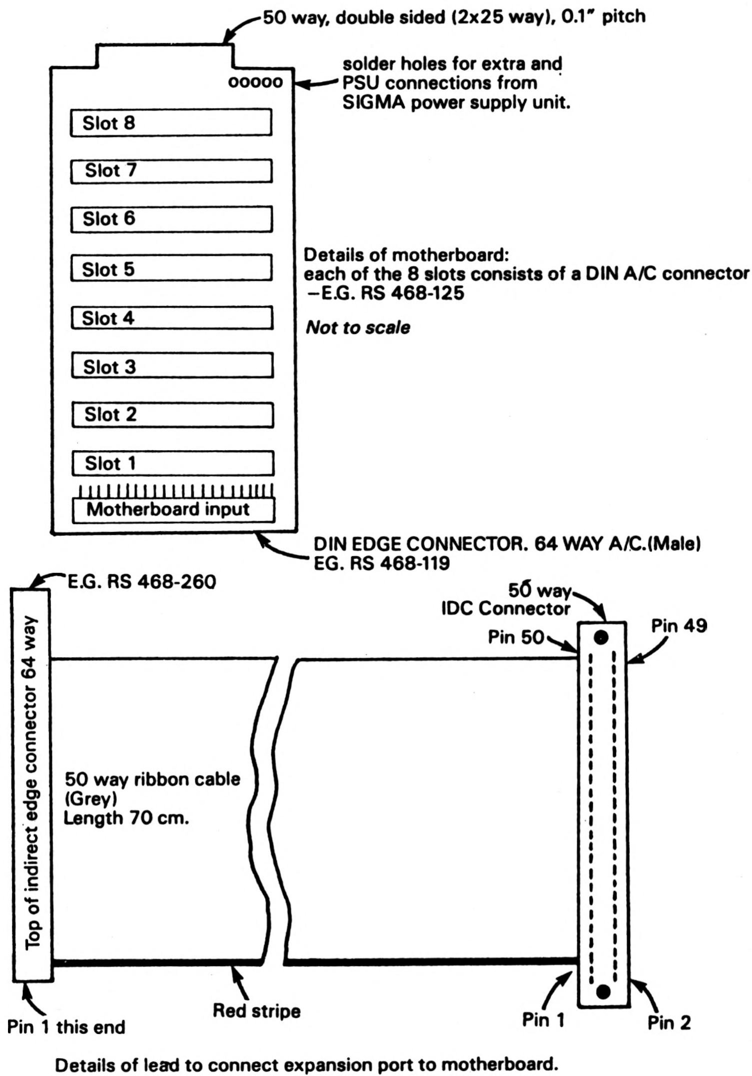
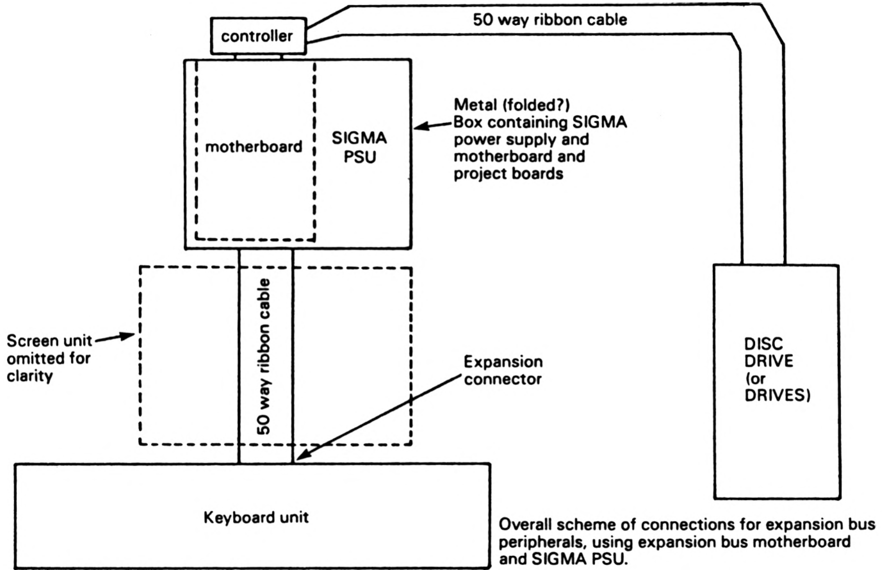


Fig. AP5.1

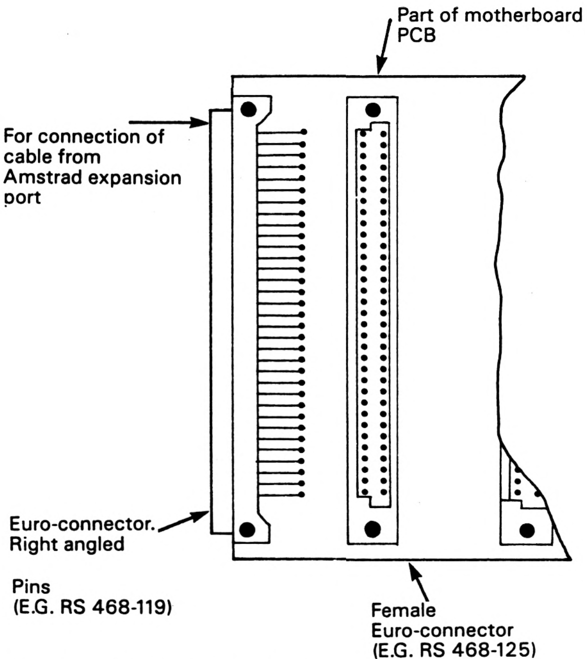
Fig. AP5.2



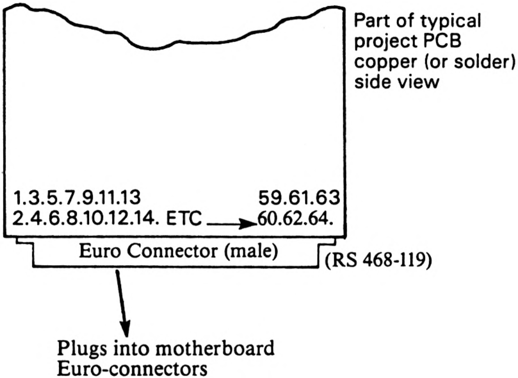
Overall scheme of connections for expansion bus peripherals, using expansion bus motherboard and SIGMA PSU.

Figure AP5.4 shows the pin designations for the expanded expansion bus. As you will see there are two spare lines remaining. These are located so as to give a small measure of protection against accidental short circuits from the + 12 and −12 Volt lines. If these short to anything which normally only works from +5 Volts, then damage usually occurs — so be careful! Pins 61 and 62 carry a voltage rail called +VDC. This is an unregulated, but rectified voltage to be input to voltage regulators located on project boards, if required. Shorting from +VDC will damage any +5 Volts only devices, so be warned!

Figure AP5.3



Top view of motherboard, showing the orientation of a Euro connector socket. (Undrawn sockets are mounted in same way.)



Solderside view of typical project PCB showing how pins on Euro connector are numbered when using this scheme. ***NOTE:*** Differs from number on the plug mouldings.

Fig. AP5.4

Pin	Designation	Pin	Designation	Pin	Designation
1	SOUND	22	D04	43	ROMDIS
2	GROUND	23	D03	44	RAMRD
3	A15	24	D02	45	RAMDIS
4	A14	25	D01	46	CURSOR
5	A13	26	D00	47	LPEN
6	A12	27	+5V	48	EXP
7	A11	28	MREQ	49	GROUND
8	A10	29	M1	50	4MHz clock
9	A09	30	RFSH	51	+5V (SIGMA)
10	A08	31	IORQ	52	+5V (SIGMA)
11	A07	32	RD	53	50Hz TTL (SIGMA)
12	A06	33	WR	54	spare
13	A05	34	HALT	55	+12V (SIGMA)
14	A04	35	INT	56	+12V (SIGMA)
15	A03	36	NMI	57	−12V (SIGMA)
16	A02	37	BUSRQ	58	−12V (SIGMA)
17	A01	38	BUSAK	59	spare
18	A00	39	READY	60	DOUT
19	D07	40	BUS RESET	61	+VDC (SIGMA)
20	D06	41	RESET	62	+VDC (SIGMA)
21	D05	42	ROMEN	63	GROUND
				64	GROUND

Many of the expansion bus signals are taken straight from the Z80A microprocessor. There are however, a number of CPC specific signals and the function of these will now be explained.

BUS RESET—Pin 40: This is an input for connection to an external switch. The switch should be connected between BUS RESET and GROUND (Pin 2,49,63, or 64). When pressed the whole computer is reset, and when the switch is again released the CPC restarts as if it had just been powered on.

RESET—Pin 41: This is an output from the Power On Reset (POR) circuit inside the CPC. It should be connected to the RESET inputs of chips on expansion bus boards, (That is those which have such input pins) to ensure their correct initialisation at power on.

ROMEN—Pin 42: This is an output from the address decode circuitry in the video gate array. It goes low whenever the Address bus contains an address applicable to the Upper ROM. This is in the range HEX C000 To HEX FFFF.

ROMDIS—Pin 43: This is an input which, when taken high, disables the ROM chip inside the CPC. Since the ROM chip is completely disabled, both lower and upper ROM segments will be inaccesible.

RAMRD–Pin 44: When this output goes low, the Video gate array has opened the buffer between the CPC RAM, and the data bus. In other words a low on this output means that the Z80A is reading from RAM. The next signal – RAMDIS, overrides RAMRD.

RAMDIS–Pin 45: This is an input which, when taken low, completely prevents the internal RAM contents from being enabled onto the data bus. This allows the user to implement schemes whereby ROMs, containing custom written software, can be switched into the memory map, to replace the RAM. This is a very useful feature for an application requiring lots of ROM, but only a little RAM space.

CURSOR– Pin 46: This is a connection to the cursor enable pin on the HD6845 VDU controller chip. The function of this is explained in the data sheet reproduced in appendix 3.

LPEN– Pin 47: This is an input connection for a light pen. It is a direct connection to the HD6845 VDU controller chip. See appendix three for full details.

EXP– Pin 48: The function of this input is not clearly defined in any of the Amstrad hardware documentation. It would appear to be a sense bit, perhaps to tell any firmware that needs to know, that there is something plugged into the expansion port. It is connected to bit 5 of port B of the on board P8255A PPI chip. (I/O address for port B is Hex F5xx).

50Hz TTL–Pin 53: This signal is created in the SIGMA power supply unit. It is derived from the very accurate 50 cycles of the mains power supply. It is provided for use in readers own projects.

DOUT– Pin 60: This is one signal you will hopefully never have to use. It is provided so that bus buffers can be fitted to the expansion bus. DOUT will reverse the data flow over the data bus. Only needed if you ever need to fit an intermediate buffer board.

All the signals in the list which are suffixed with the word SIGMA, in brackets, originate in, or are power supply lines from, the SIGMA power supply, – see beginning of chapter three.

Physically the extended expansion bus, consists of the SIGMA power supply, and the expansion bus motherboard. These should be built into a metal case. As shown on the PSU diagram in chapter three, the case must be grounded for safety reasons. Earthing the case can also improve both noise immunity, and reduce interference to nearby Radio or TV receivers. See page iii for supplier details of motherboard and other Printed Circuit Boards.

Expansion bus addresses of projects detailed in this book

Hex I/O address	Used for:	Project name
DF00	ROM select register	Exp ROM bd.
F8E0	Local address (low)	EPROM pgmr
F8E1	Local address (mid)	EPROM pgmr
F8E2	Data read/write register	EPROM pgmr
F8E3	PPI control register	EPROM pgmr
F8E4	EPROM programmer status register.	EPROM pgmr
F8F0	PPI #1 port A	Multiproj bd.
F8F1	PPI #1 port B	Multiproj bd.
F8F2	PPI #1 port C	Multiproj bd.
F8F3	PPI #1 control register	Multiproj bd.
F8F4	PPI #2 port A	Multiproj bd.
F8F5	PPI #2 port B	Multiproj bd.
F8F6	PPI #2 port C	Multiproj bd.
F8F7	PPI #2 control register	Multiproj bd.
F8F8	Speech synth. allophone register	Speech synth.
F8F9	Inflection register	Speech synth.
F9E0	USART 1 data register	RS232 Vers B.
F9E1	USART 1 Control register	RS232 Vers B.
F9E2	USART 2 data register	RS232 Vers B.
F9E3	USART 2 Control register	RS232 Vers B.
F9E4	USART 3 data register	RS232 Vers B.
F9E5	USART 3 Control register	RS232 Vers B.
F9E6	USART 4 data register	RS232 Vers B.
F9E7	USART 4 Control register	RS232 Vers B.
F9E8-F9EB	Baud rate register for chan 0 & 1	RS232 Vers B.
F9EC-F9EF	Baud rate register for chan 2 & 3	RS232 Vers B.
FAE0	LOCAL address (low)	ROMdisk
FAE1	LOCAL address (mid)	ROMdisk
FAE2	LOCAL address (high)	ROMdisk
FAE3	ROMdisk PPI chip control	ROMdisk
FAE8	ROMdisk read data register	ROMdisk

APPENDIX 6

Begin Here

For readers new to computers, or new to computer hardware.

Binary: Address decoding: Logic chips: Circuit diagrams: The physical construction process: Soldering: Pitfalls: General considerations: Power supplies – "Stop! stop!" cried the Mole "This is too much!"

If you have never constructed any circuitry before, your first attempt can be a bewildering and frustrating experience. My own first attempt ended up being taken to a friend, who probed it for a few minutes, and then removed a small piece of solder which had been bridging two chip pins. It then worked perfectly. This tale points up two things. Firstly, when you design and build a project, the likelihood is that it will not work first time. (Don't panic! when you build from published designs, there is a far better chance that a project will work first time.) Secondly, there is nearly always a simple reason why the project refuses to work. Inoperative chips are very rarely the cause.

Let me now try to take some of the confusion out of constructing circuitry, by pointing out the answers to some questions that may occur to you, when you look at the diagrams and explanations for the projects in this book. First some basics, the art of addressing memory and peripherals.

The Z80A microprocessor in the CPC 464/664, is able to access 65536 locations of memory. Think of this as a set of 65536 pigeon holes, such as you might find in a large postal sorting office. Each pigeon hole has a unique identification number attached to it, this number is called its address. This first set of pigeon holes is utilised as memory locations, that is to say they are used to store values, characters, or lists of instructions which form programs. Additionally the Z80A can also access a second set of 65536 pigeon holes. The second set are used for Input/Output. (Often abbreviated to I/O.) We could perhaps say that the second set of pigeon holes have no backs to them, and that characters, or values placed in them can get out of the sorting office, and to the outside world. Similarly, characters or values from the outside world can find their way into the computer via this second set of holes. In theory, the Z80A can read data from, or write data into, any pigeon hole, from either set. In practice it depends on the actual chip where the pigeon hole is physically located. Some pigeon holes are Read Only Memory (ROM), whilst others are read or write memory. (Known as Random Access Memory– commonly abbreviated to RAM).

When the Z80A needs to access a particular location (or pigeon hole) in its memory, it outputs a binary representation of the address of that location onto a thing called the address bus. (binary will be explained shortly). The address bus, is a set of 16 pins on the Z80A chip. These are numbered A0–A15. The 16 signals emanating from these pins are collectively known as the address bus lines. These address bus lines are extended to every part of the computer. Either via the tracks on the CPC printed circuit board (PCB), or via the expansion port connector, to external devices, like the ones detailed in this book, or perhaps your Amstrad disk drive controller. Each bank of memory, or each external device has an address decoder assoctiated with it. The job of the address decoder is to constantly monitor each address output from the Z80A. Each decoder is set up to detect a specific address, or range of addresses. When a decoder detects such an address, it produces a signal to enable the memory chip, or peripheral device with which it is assoctiated. This signal which the address decoder produces is usually called a Chip Select (CS) signal. After the chip select signal has been produced the transfer of data from Z80A to memory or peripheral, or vice versa, takes place over a thing called the data bus. The data bus is a set of eight signal lines over which the transfer of data to and from memory, Z80A, and peripherals takes place. Like the address bus, the data bus is connected to all parts of the computer.

Binary was briefly mentioned above, let us now take a look at the binary numbering system, and why it is so important to computer science. (See also Appendix 2 of your Amstrad user instruction manual.)

We are all conditioned to count in powers of 10. Our money, our weights measuring system, our mathematical addition, subtraction, division, and multiplication, are all done in powers of ten. We have ten numbers in our numbering system – including zero. If I said to you that I had spent 700 man hours in preparing this book, you would know immediately that I meant; no thousands, seven hundreds, no tens, and no units. What if I said I had spent 1010111100 hours on it? You would not believe me, because you would quite reasonably assume that the number above was a decimal value, and that would mean that I had spent over 32 years in preparing this book, Obviously, I would have needed a powerful gift of foresight! But if you look again at the number above, you will see that it contains only two of the ten numbers which we use in our decimal numbering scheme, namely one and zero. As you have now guessed, the number was expressed not in decimal, but in binary.

Let us compare the two numbers 700 and 1010111100. We have already said that 700 decimal, means seven hundreds, no tens, and no units. Let us now look at how the number 1010111100 may be assessed.

In binary there are only two numbers used. These are one and zero. That is why binary is used in computers. It is ideally suited, because Computer signals are either on – voltage present (logic one), or off – voltage absent (logic zero) . The ones and zeros are used to express numerical values, in the same way that ten numbers are used in the decimal system. As with the

decimal system the position in which a digit occurs gives it its weight. (Its weight being its overall contribution to the total.) If we start at the right of a decimal number, that is, the units column, and move right to left across the other columns, we realise that each succesive column has a weight which is ten times the previous one. Thus we move from tens to hundreds (10 x 10 = 100), and from hundreds to thousands, and so on. In the binary numbering system the rightmost column has a weight of one, and as you move across one column to the left, each succesive column has a weight which is double its predecessor. Thus the values of the columns from right to left are; 1,2,4,8,16,32,64,128 and so on. To illustrate this let us write out our 700 example, and next to it, put our binary example – then add up all the individual weights present in each number;

Hundreds	*Tens*	*Units*	*512:*	*256:*	*128:*	*64:*	*32:*	*16:*	*8:*	*4:*	*2:*	*1*
7	0	0	1	0	1	0	1	1	1	1	0	0

7 x 100 = 700
0 x 10 = 000
0 x 1 = 000
———
TOTAL 700
———

0 x 1 = 000
0 x 2 = 000
1 x 4 = 004
1 x 8 = 008
1 x 16 = 016
1 x 32 = 032
0 x 64 = 000
1 x 128 = 128
0 x 256 = 000
1 x 512 = 512
———
TOTAL 700
———

From this example you can easily see how the binary numbering system works. But don't take my word for it, go and ask your Amstrad – Arnold never lies! Get the "Ready" prompt, then type in "PRINT BIN$(700)". This tells the Locomotive BASIC to print up the binary pattern of ones and zeroes, for the decimal number in the brackets. Incidentally BASIC will also do the opposite, that is show the decimal value of a binary number, by typing in "PRINT STR$(&X1010111100)". Try it and see for yourself.

To get back to the Z80A address bus signals, the address bus lines have either voltage on or off conditions on them to inform all devices connected to the address bus, which chip it wishes to access. These signals form the binary pattern for the address. Don't try to look at the signals on the address bus with a meter, they appear and disappear very quickly indeed, each one being only present for perhaps a fraction of a thousandth of one second, before being replaced on the bus by the next address which the Z80 needs to access.

More binary!! Yes, but this time we shall see how to simplify writing binary numbers down. All those great long chains of ones and zeros are horribly tedious to note down, and the scope for mistakes is great. Fortunately, you

very rarely need to convert a number to its ones and zeros. This is due to the widespread usage of the hexadecimal notation system. Hexadecimal is so called because it uses the decimal number set, zero to nine, and also uses the first six letters of the alphabet. The hexadecimal notation gives a way to write down, as a single character, any combination of ones and zeros which can be possible in four binary columns. The complete list is;

Binary:		*HEXADECIMAL:*		*DECIMAL:*
0000	=	0	=	0
0001	=	1	=	1
0010	=	2	=	2
0011	=	3	=	3
0100	=	4	=	4
0101	=	5	=	5
0110	=	6	=	6
0111	=	7	=	7
1000	=	8	=	8
1001	=	9	=	9
1010	=	A	=	10
1011	=	B	=	11
1100	=	C	=	12
1101	=	D	=	13
1110	=	E	=	14
1111	=	F	=	15

When a binary number has more than four columns then we split the ones and zeros into groups of four, from the right. for example, the binary value of 1010 0001 would be written in HEXADECIMAL,(hereinafter shortened to just HEX.) as, HEX A1, whilst binary 0100 1110 would be written down as HEX 4E. Note that the groups of four columns are separated with a space to improve legibility. Note also that when writing a number down in HEX, you should always make it clear that it is a HEX number, rather than a decimal one. You do this by writing either "HEX CE" or "xCE". In the context of Amstrad CPC BASIC the ampersand is also used to signify that the number which follows it is a HEX number. The codes for each HEX character seem daunting, but after a few weeks of working with them they become second nature – you may even find, as I sometimes do, after a lot of HEX working, that you attempt to add up your grocery bill in HEX – not recommended, the cashiers get very confused!

So now we have a way to express those ones and zeros which the Z80A outputs as an address. Remember that the address bus contains the address of the memory or I/O port location (or using the earlier analogy, the pigeon hole), which the Z80A wants to access. The actual transaction of data between the addressed location and the Z80A will take place over the data bus.

Later in this chapter we shall look at an address decoder chip, but before we can do that let us cover some basic facts about chips in general.

The name "chip" is given to any integrated circuit. I expect that you know that a chip contains anything up to half a million transistors , all crammed onto a tiny piece of silicon. Exactly how many transistors there are on any given chip will depend upon how complex the function of the chip is. The actual chip is enclosed in a ceramic package, and the tiny connections at the side of the chip are extended out to the pins on the package. Once again the number of pins on a chip package (hereinafter just called a chip) will depend upon how many external connections it needs to function. Common numbers of pins are as follows;

8 Pins	14 pins	16 Pins
18 Pins	20 Pins	24 Pins
28 Pins	40 Pins	64 Pins

Because the pins are arranged as an even number down each side of the package, the name Dual In Line package has been derived. This is abbreviated to DIL. Thus retailers and manufacturers will talk about a chip being supplied in a 20 pin DIL package. When it actually comes to fitting a chip into a circuit there are two ways to do it. Firstly, you just solder the chip into place, soldering each pin of the chip to the board to make the connection. This is alright if you can solder well and quickly. But if you can't then you can sometimes damage the chip by the application of too much heat for too long a time. Another disadvantage of this approach is that if the chip ever fails, removing it will mean yet more soldering, and the use of a solder sucker or other device. This can easily damage the solder tracks on the circuit board. A slightly more expensive method, which overcomes both the above problems, is to use chip sockets. These are available for all the above DIL package sizes. The socket takes the brunt of the heat, which it is well able to withstand. If the chip ever fails it is a simple unplug and plug in job to fit the new one. The extra cost of sockets is usually justified.

So now you know what a chip is, or do you? I am afraid that you would be laughed out of most component retailers if you were to just walk in and say "I'd like a chip please". It would be rather like walking into a gardening centre and saying "I'd like a flower please".

Chip types are specified as numbers. The numbers are obtained by looking at the parts list of a project in a magazine, or in a book like this one. When you get to designing your own circuits, you will have to get a big thick book, published by one of the chip manufacturers, which details all their products. Some of these books give details of application circuits for the chips too. Let us go in for a little simplification about chip types here. Chips which are less complex than microprocessors, or speech chips, etc, are SSI or MSI chips. (More initials! these stand for Small Scale Integration or Medium Scale Integration). The functions of chips in these classes tend to be things like counters, gates, decoders, and things called shift registers. We will be looking at these various types of chip a little later. SSI chips and MSI chips are available quite cheaply, from about 15 pence upwards to be precise. Of course, the price rises as the complexity of the chip does – up to a point. They

are available in two logic families. This means that you can get chips which fulfil roughly the same functions, but which are fabricated using two totally different manufacturing processes. Let us meet these families.

The first family is called TTL (oh dear, still more initials! These stand for Transistor Transistor Logic). TTL was the first family of logic chips, and became available from about 1970 onwards. Within the TTL family there are many variants. The standard TTL family chips all have numbers which begin with 74 – thus the 74138 chip is a TTL address decoder. Texas Instruments, who originated TTL, have brought out newer sub families of TTL which feature lots of goodies, like faster operation, and lower power consumption. Chief among these sub families is the 74LS range. The LS stands for Low power Schottky. The LS range chips are connectionally and functionally identical to the original 74 range, but in general they consume less power, and can be faster. Thus, a 74LS138 is a plug in replacement for a 74138, or a 74LS32 can replace a 7432. TTL chips can only operate from a +5 Volts DC supply, and they are all fully interconnectable with one another. The LS versions of TTL are probably now more widely used than the second family of logic chips, which we shall now meet.

The second, and slightly newer, family of logic chips is called the CMOS family. (CMOS stands for Complementary Symmetry Metal Oxide Semiconductor). There have been a couple of sub families of CMOS too. The original CMOS chips are called the "A" series. These were initially unpopular because of their extreme vulnerability to static damage. This meant that anybody wearing a wollen jumper who handled an "A" series chip, was very likely to render it inoperative just by the action of holding it. Old time electronics buffs speak with glee of building circuits with early CMOS chips, holding the chips in one hand, and an earthed wire in the other, to drain away body static! Fortunately RCA, who originated CMOS, soon came out with the now widely used "B" series. This second series proved far more robust, and in my experience "B" series chips can be treated in exactly the same way as LS family TTL chips, without any adverse effect. By now you must be wondering what the advantages of this quirky second family are. There are several. The first is phenomenally low power consumption. Many CMOS chips have current consumptions which are measured in micro amps, this makes them ideally suited to battery powered operation. The second most important feature of CMOS chips is that they can be operated at various supply voltages, usually from 5 volts DC to 18 Volts DC. This allows more flexibility in power supply design, and allows direct operation from automotive power supplies etc. Finally there are some logic functions available in CMOS which cannot be found in the TTL family on a single chip.

As to numbering, all CMOS family logic chips begin with 4 and you can tell an "A" series chip from a "B" series one by the suffix after the chip number. Thus a 4040A is an "A" series version of a 12 stage divider chip, and a 4040B is a "B" series version of the same thing. This clarifies a point, like the standard and LS versions of TTL, "B" series CMOS devices are plug in replacements for "A" series chips in 99.99 percent of all logic circuit applications. In the projects

in this book I have not specified that you use "B" series devices, because I advise that you never use "A" series unless a design specifically calls for one. None of the designs in this book call for "A" series. CMOS has not become as popular as TTL in the hobbyist market. This really stems, I think, from the prejudice created by the false start with the "A" series. Pricewise CMOS tends to cost the hobbyist a few pence more per equivalent logic function than TTL.

The families don't feud with one another. (Though there has been the odd verbal skirmish between adherents of the families!). Provided that you use a +5 Volt power supply with CMOS, you can interconnect CMOS and TTL circuitry with complete compatibility, subject to loading restrictions, – but that's another book.

Now let us get to grips with some actual logic functions. We will not be going into this subject very deeply, just enough to give you a feel for it.

Remember that in all these logic function descriptions, when we say a logic high or a logic one, we mean that the voltage at the indicated point is about 4.5 to 5 Volts. When we say logic low or logic zero, we mean that the voltage at the indicated point is less then one volt.

The most simple logic functions are called gates. Because these are so simple you will often find that up to four or five functionally identical, yet independent gates are included on one chip. So what is a gate. A gate is a way of including a simple decison making capacity into a circuit. Your typical gate has a number of input pins, and a single output pin. There are four basic types of gate. These have the following names;

NAND gates AND gates OR gates NOR gates

These rather esoteric sounding names conceal some very simple concepts. Let us take for example an AND gate. As previously stated an AND gate , like any other, has a number of inputs, and an output. For the sake of this example let us look at an AND gate with two inputs. The AND gate has the capability to tell us when both of its inputs have a logic one present upon them. It does this by giving a logic one on its output when input one, AND input two, are high (thus the name AND). If either of the two inputs has a logic low on it then the AND gate ouput will remain stubbornly low. This will hold true even for and AND gate with eight inputs. The output will only be logic one when Input one AND input two, AND input three......AND input eight are all high. From the foregoing it should be easy for you to guess the function of the the OR gate. Yes? well let's just run through it. The output of an OR gate will go to a logic one, when one or more, of its inputs are at a logic one. Thus, for a four input OR gate: if Input one OR input two, OR input three, OR input four go to logic one, then the output will do likewise.

The NAND and NOR functions are rather simple too. The N on the front gives the clue. The input conditions needed to activate the outputs are the same as for the AND and OR gates. The only difference is that instead of being low when the input conditions are not met, and going high when they are (in all inputs high for the AND gate), the outputs of the NAND and NOR gates are normally high, but go low when their input conditions are met. Thus the output of a NAND gate will stay at logic one until all its inputs are taken to logical one, whereupon the NAND output will go low. The output of the NOR gate will go low only when one or more of its inputs are at logical high.

So what use are these gates? Individually they can't really do a lot. But when you interconnect lots of them the result can be very useful. By way of a demonstration look at your CPC! This has lots of gates inside it, albeit that many of them are crowded onto the Z80A microprocessor chip. In broad terms gates are interconnected inside the Z80A to close and open data pathways, based on conditions sampled at their inputs. The Z80A could not exist without gates. At a more humble level you will find that the add on projects in this book make liberal use of individually connected gates. So what does a gate look like? how do you get to grips with a chip containing one? Well it's funny you should ask that, here comes a gate chip right now!

Figure AP6.1 shows the plan view of a TTL chip called a 74LS08. (Remember that the 7408 would look exactly the same, but uses more power). You can see that there are 14 pins on the chip, 7 a side. So we can immediately say that this is a 14 pin DIL package. The pins are numbered. It is a cardinal rule that the pin on any chip which is nearest the little round hole, or sometimes it is a small indent, in the plastic is ALWAYS pin 1. The pins then number down one side and then up the other, as shown for this 74LS08. This convention applies to ALL chips, I have never seen one that did not conform. The next thing to note is the two corner pins, pin 14 and pin 7. Pin 14 is labelled VCC. In plain English this means Positive supply, which for TTL chips as we have seen, is always +5 volts. So this pin takes in the supply voltage to operate all the elements on this chip. Pin 7 is labelled GND. Again in plain English this means the ground or zero volt connection to your +5 Volt power supply. (Haven't got one? look at the start of chapter three.) Now we arrive at the nub of the business! Pins 1 and 2 go into a strange looking symbol, which is actually the circuit symbol for an AND gate. The output of this particular AND gate is connected to pin 3 of this chip. As you can see there are a total of four two input AND gates on this chip. Each one of them will operate entirely independently of the others, the only things they share are the package they live in, and the supply voltage coming into pins 14 and 7. In all respects each AND gate is totally independent of the others. This is not the place to set up experiments for you to try, so please take my word for it that if you were to connect pins 1 and 2 to pin 14, then pin 3 would go to a logic one. Whilst if you connected pin2 or three to pin seven, then pin 3 would be a logic low. The 74LS08 is sold as a quad (because there are four gates) two input, AND gate package. Within both TTL and CMOS there are chips containing AND gates with 3, and four inputs.

Fig. AP6.1: Plan view of 74LS08 chip

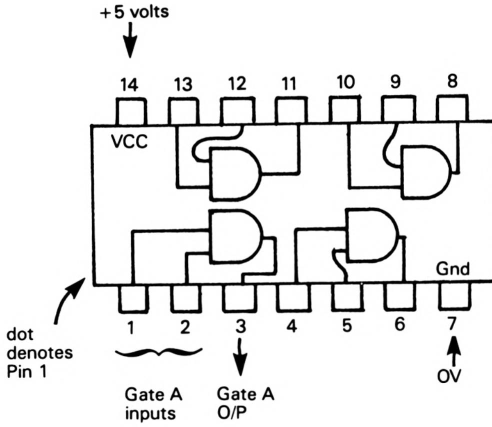
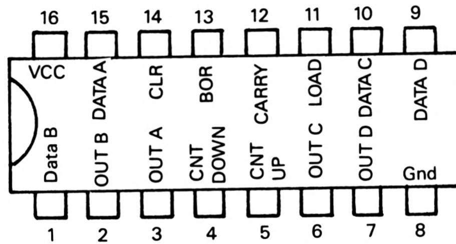


Fig. AP6.2: Plan view of 74LS193 chip



Drawn approx. 2 times real size

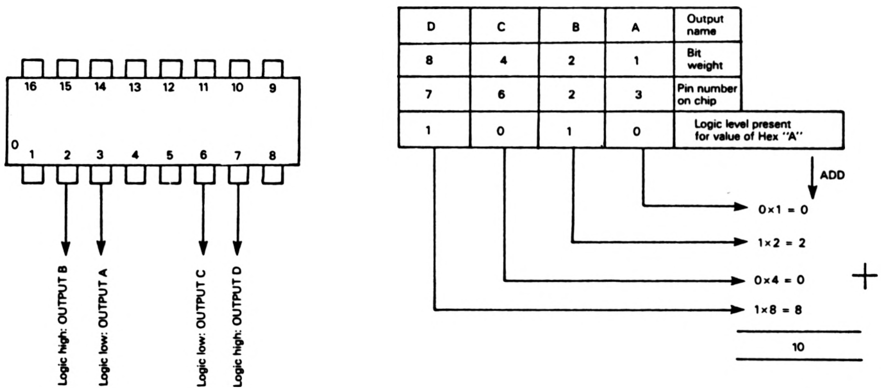
We have just seen the circuit symbol for an AND gate, and you will find a full list of the circuit symbols for the other kinds of gate, in appendix l of this book. Please be aware that there are other symbols used by various parties to portray the same thing, though I think the ones used in this book are probably the most commonly used.

Now that we have looked at gates, lets us go on to look at some slightly more complex logic elements. Counters are another logic element which are crucial to the operation of a computer, and a great many other applications of a logic electronics. Counters can be used for a great many things. Examples are: time delay counting, repeat sequence counting, pulse counting, position sensing counting, digital watches and clocks – the list is endless. Let us take another concrete example.

The 74LS193 is a chip from the TTL family, containing a counter which has the ability to count, in binary, from zero to fifteen.t can count up or down, it can be set to zero, or it can produce a carry signal to another counter (in the same way that when the seconds counter on your digital watch reaches 60, it produces a carry signal to add one to the minutes counter). Fig. AP6.2 shows a plan view of a 74LS193. As before it is in a DIL package, but this time with 16 pins. Also the same as the 74LS08 is the fact that the same corner pins take the power supply voltage – +5 volts, and ground, on pins 16 and 8 respectively. We go back to the beginning of this chapter to revisit binary now. Pins 3,2,6, and 7 are the actual count outputs. That is, on these pins will be the logic one and zero levels which will make up the binary value of the current number in the counter. If, for example, these were as shown in fig. AP6.3, then the count value would be ten (or HEX value A). We can easily set the value of the counter to zero just by applying a logic one to the CLEAR input on pin 14. This should only be temporarily left high though, since the counter will be continuously cleared by it, and can never count until a logic zero is present on the CLEAR input. Next let us see how to make the 74LS193 count. As we said the counter can count upwards, or downwards, that is, zero one two three four..... OR fifteen fourteen thirteen twelve..... and so on. The way in which to make the chip count upwards, is too apply voltage pulses to pin 5, whilst connecting pin 4 to a logic one. The voltage pulses would normally be derived from a thing called a clock circuit. This has little or nothing to do with time of day keeping, but merely generates pulses at fixed intervals. Most computer circuitry makes extensive use of clock pulses from some source or another. In the CPC 464 for example a crystal oscillator provides a clock frequency of 4 million pulses per second. This is used fairly extensively in the projects detailed in this book. So we apply clock pulses to the COUNT UP input of the 74LS193, on pin 5, and each pulse fed into pin 5 increases the count value by one. (In computing, adding one to a count value is called incrementing it.) To make the counter count backwards we hold pin 5 at logic one, and apply the clock pulses to pin 4, the COUNT DOWN input. The count inputs of the 74LS193, and indeed most other TTL and CMOS counter chips, are a special type of input called an edge triggered input. They have to be of this special type due to the sheer speed at which they can operate. To explain, the 74LS193 is capable of counting 25 million input pulses per

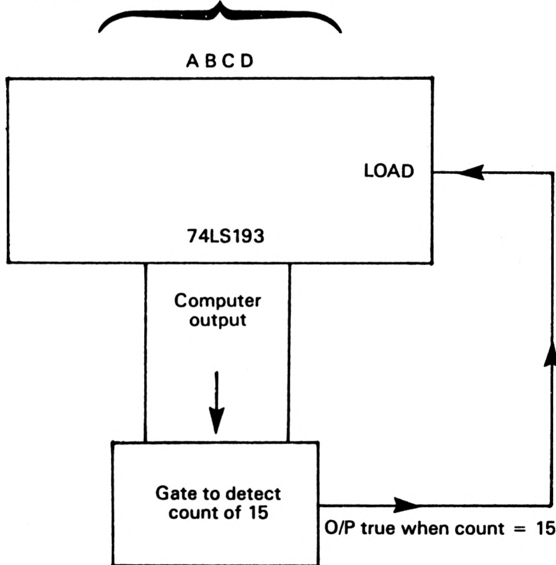
second. (Obviously because it can only count to fifteen the 74LS193 would recycle 156250 times for 25 million input pulses, 16 times 156520 = 25,000,000) In order that no pulses be missed or counted twice, the 74LS193 modifies the counter value when it senses a logic zero changing to a logic one at one of its count inputs. This is called a positive edge triggered count input. A negative edge input (as found on a chip like the 74LS93) is activated by a logic one changing to a logic zero, this is called a negative edge triggered count input. Other types of chip have positive edge triggered inputs.

Fig. AP6.3: Shows logic levels output from 74LS193 counter chip, when count value = Hex "A" (Decimal 10)



The final feature of the 74LS193 we shall look at is its preset capability. Refer to Fig. AP6.2 again, and look at pin 11. This is labelled LOAD. The purpose of the load input is to allow you to place any required value in the counter. The binary pattern of ones and zeros representing the number you wish to preset into the counter are set up on the DATA A, DATA B, DATA C, and DATA D inputs. These are pins 15,10,9, and 1 respectively. To load them into the counter the LOAD pin, which during normal operation must be held at logic high, is pulsed low. I hear many readers asking, Alan, why should you wish to set the counter to preset value? There are many reasons. Consider if you were building an electronic timer which had to count downwards in seconds. You could easily use 74LS193 chips as the basis of such a counter, but consider what happens when counting downwards in seconds. When you get to, let us say, thirty, the next count must be twenty nine. So the lowest digit of the counter must change from zero to nine. Can the 74LS193 do this unassisted? no it cannot, it's count sequence is 3,2,1,0,15,14,13 etc. Therefore we need some extra circuitry to detect when the counter outputs 15 (Binary 1111). Fig. AP6.4. shows in block diagram form how this extra circuitry would be connected. This circuitry will activate the LOAD input of the counter, which would have 9 (binary 1001) set up on it's DATA inputs. There are many other uses for the preset, or load feature on the 74LS193, to give another important example, frequency division.

Fig. AP6.4: Block diagram of count value detect and reload scheme



Briefly, frequency division is another way in which counters can be utilised. The basic scheme is that you feed a square wave signal with a known frequency into a counter, and obtain a sub multiple of that frequency out of it. For example, if you feed a 10,000 Hz frequency into our 74LS193 chip, the output signal at the QD output on pin 7, will be 10000 divided by 16 = 625 Hz. We divide by 16 because we have to include zero as a valid counter state for this application. We could use some extra logic outside to detect a particular count value, the output from this logic would then be connected to the 74LS193 CLEAR input, pin 14. This is called modifying the count length. If the chosen value was 12, then whenever the counter reached 12 the external logic would reset it to zero. Fig. AP6.5 shows block diagram form how we could hook up such external logic. The counter now has a count length of twelve. It never actually reaches twelve, but once again we are including zero. Using this trick you can make a divider which will divide by any number from 1 – to n where n is the actual maximum count which the counter is capable of.

As a final chip to look at, I have chosen the 74LS138. You will probably form the opinion that this is one of my personal favourite chips, when you come to look at the diagrams for the add–on projects in chapter three! The 74LS138 chip is officially known as a 3 to 8 line decoder. Let us see what that means.

Fig. AP6.5: Using a gate to modify the count length of the 74LS193

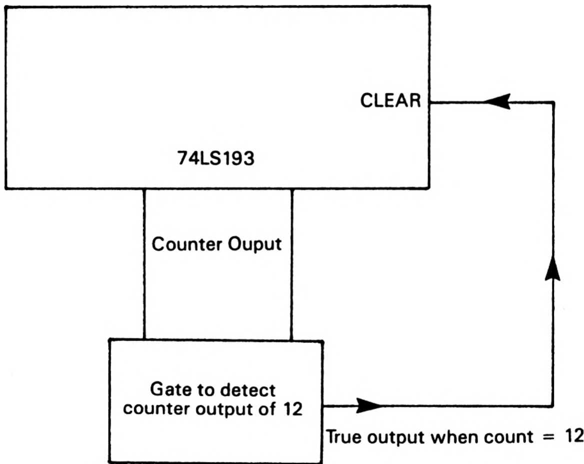
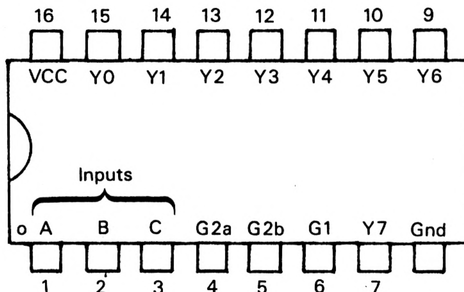


Fig. AP6.6 shows the pinout of the 74LS138 chip. As usual the chip has corner supply pins, that is pin 16 is VCC, and pin 8 is ground. There are 8 outputs, called Y0 – Y7, and a number of inputs, chief among these are the A, B, and C inputs. If we ignore the rest of the inputs for the moment, understanding the chip will probably be easier. Once again good old binary comes into it, the binary number present on the inputs A, B, and C, tell the chip which of the Y outputs to turn on. Since the Y outputs are normally all at logic high, the selected output goes low. So if we input the value 5 (binary 101) to the ABC pins, then the output Y5 on pin 10 of the chip will go low. The purpose of the remaining inputs of the chip, G1, G2a, and G2b, is to enable or disable this basic function. That is to say that if input G1 is low, or if either G2a or G2b is high, then the decode of the ABC inputs will not occur. So to allow the decode to happen, the G1 input must be high at the same time as both G2a and G2b are low.

Fig. AP6.6: Plan view of 74LS193 chip, showing pinout details



The 74LS138 chip is universally used as an address decoder. Think back to the start of this appendix, remember how the Z80A microprocessor outputs a binary number on the address bus to show which location it wants to access? There has to be some circuitry somewhere to continually monitor those address lines and pass a select pulse to the addressed chip. Let us take a non CPC example here of how we could use a 74LS138 chip to decode an address. Pay attention now, because we are going to use a real circuit diagram this time. Let's say that we want to produce a logic low select pulse whenever the address output by the Z80A is hex FFF3. Below we can see the logic state for each of the address lines when the address is hex FFF3:

A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1

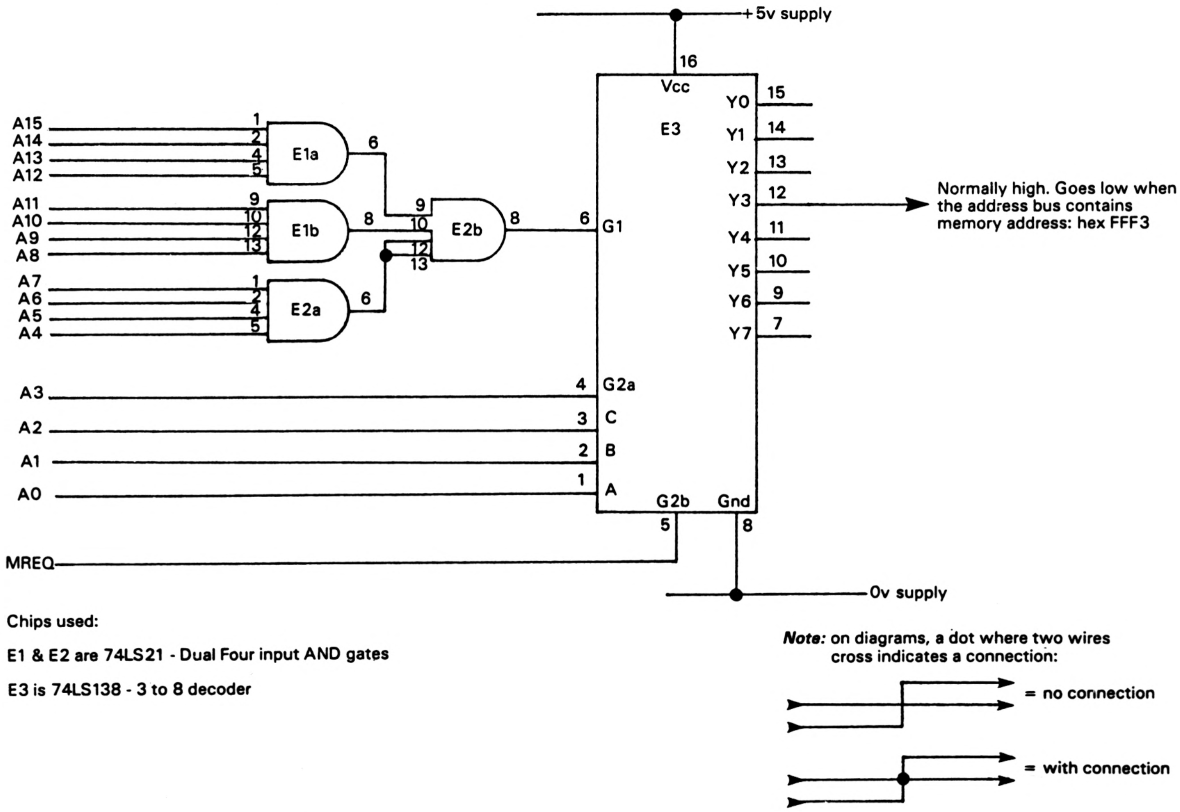
F F F 3

Now we shall use our 74LS138 to form the central element of a circuit to decode this address. The decoder chip will need some help from other gate chips. Looking at the binary pattern above we can see that if we fed address lines 0,1, and 2 into the A,B, and C inputs of the decoder chip, that would decode the binary 011 (decimal 3) part of the address. Address line 3 will be low when the address we require is present, so it makes sense to connect that to the G2a input, which has to be low to enable the decoder. Two dual four input AND gate chips will take care of the rest. There will be four AND gates each with four inputs. As we saw previously the output of each AND gate will go high when its inputs are all high, so we can connect the AND gates as shown in fig. AP6.7. we only need three of the gates to connect to address lines A04–A15. The fourth AND gate, which has its output connected to the G1 input of the decoder, is used to detect when the other three AND gates have detected all highs on their inputs.

The outcome of all this is that the G1 input to the 74LS138 will receive a logic one, thus enabling the decoder when the AND gates detect that address lines A04–A15 are all high. That leaves us with input G2b which we will connect to a signal called MREQ bar. This goes low on Z80 systems when the Z80 is requesting a memory read or write. So to recap, the ABC inputs are enabled to decode address lines A00–A02 only when the AND gates have detected all ones on their inputs, and have thus fed a logic one to decoder input G1, and when A03 is low, and when the MREQ line connected to decoder input G2b is low. When all these conditions are met the Y3 output on pin 12 of the decoder, will go low.

That concludes our brief look at a few real chips. I realise that for many readers the preceding part of this appendix will have been heavy going, but these concepts will be used over and over again, so it is well worth the effort of making them "click". To get on to the more sophisticated aspects of logic electronics, like arithmetic units, parity generators, code convertors etc, readers are advised to buy a book like the Texas instruments TTL data book,

Fig. AP6.7: Decode circuit to decode the address hex FFF3 – see text



or the RCA CMOS data book. These are not cheap (about £10 or so), but will provide you with an endless source of information, not only about the range of chips available, but also about how you can use them. Bear in mind that a great many circuit designs found in electronics magazines are based on the applications examples to be found in such books.

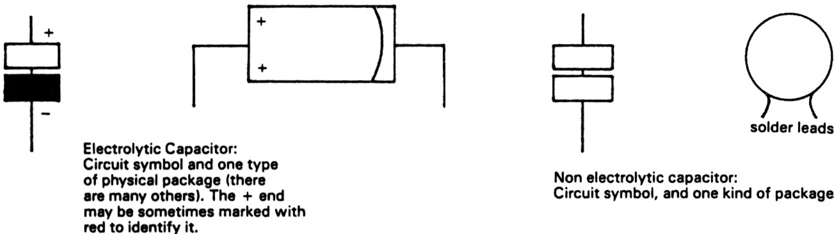
Discrete components

A discrete component is one which is not part of an IC. Capacitors, resistors, diodes, and transistors, are all discrete components. With a little practice you will soon be able to understand logic circuits, and perhaps design your own. You will also need to know how to use the discrete components, so there now follows a brief description of the properties of each of the above mentioned ones.

Capacitors

Capacitors come in many types. The two main types are called electrolytic and non–electrolytic. The circuit symbols for these are shown in fig AP6.8. The capacitance of a capacitor is expressed in micro farads, and, for small ones, submultiples of micro farads called nano farads and pico farads. 1000 pico farads =1 nano farad. 1000 nano farads = 1 micro farad. Fig. AP6.9 shows the usual notations for these ratings.

Fig. AP6.8



Electrolytic Capacitor:
Circuit symbol and one type of physical package (there are many others). The + end may be sometimes marked with red to identify it.

Non electrolytic capacitor:
Circuit symbol, and one kind of package.

Capacitors: Always ensure that the voltage marked on the capacitors you use is greater than the voltage you intend to pass through it.

Fig. AP6.9

Fig. AP6.9 Symbols for capacitance

μF = Microfarad symbol
nf = Nanofarad symbol
pf = Pecofarad symbol

1000 pf = one nano farad 1000 nf = one mic farad

There are two properties of capacitors which are used in computers. The first is that a capacitor can hold an electric charge, not a very large one in the sizes of capacitor we are likely to deal with, but large enough to use as a smoothing influence on an unstable signal. When we use them in this way we connect them as shown in fig. AP6.10. Another use of this is that we can trickle charge a capacitor through a resistor, and use a chip to monitor how high the charge is. This idea forms the basis of many timer circuits, since the capacitor will take roughly the same time to reach a given charge level at every operation. One of the most common uses for capacitors in computers is as decouplers. The action of a TTL chip switching can cause nasty blips on the power supply line, perhaps affecting other chips. This can be minimised by connection of a small 0.1 micro farad capacitor across the power supply lines near one in every four chips. Fig AP6.11 shows how this is done, it gives a little local spare capacity to the +5volt line at the point where it is needed.

Fig. AP6.10: Capacitor connected as smoothing device

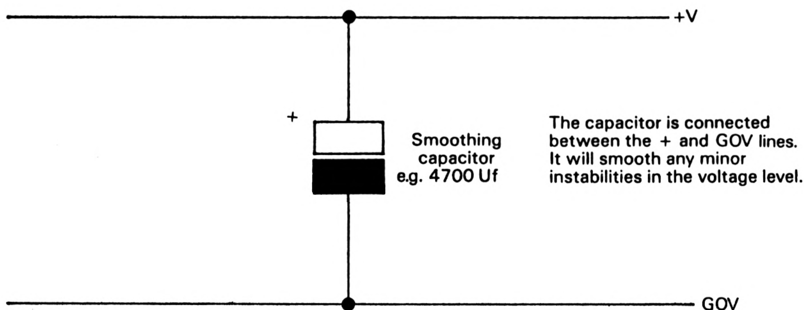
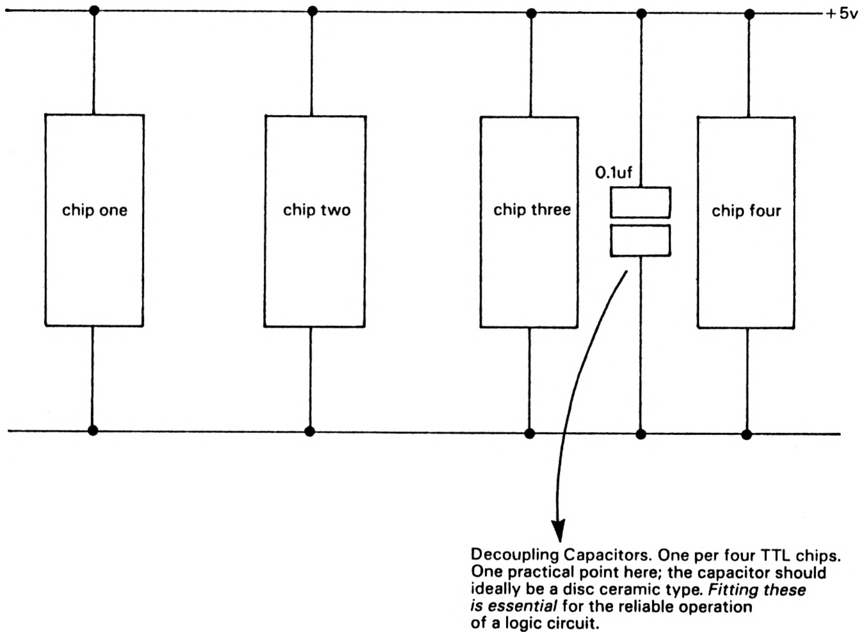
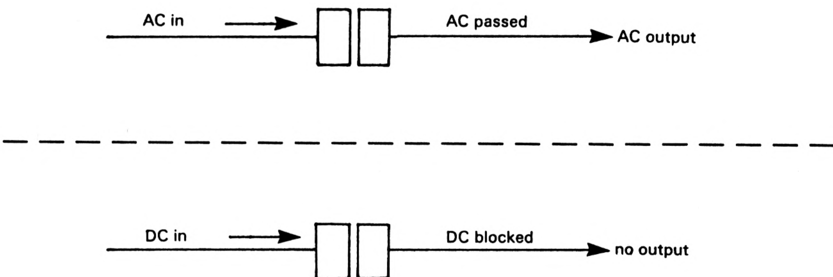


Fig. AP6.11: Decoupling Capacitor



The other way in which we use capacitors is as a DC block. That is to say that when connected as shown in fig. AP6.12 the capacitor will pass any AC signal presented to it, but block any DC signal. This property is widely used in audio circuitry.

Fig. AP6.12: Showing diagrammatically how capacitors block DC



Resistors

The symbols for fixed value, and variable resistors are given in fig AP6.13. It puzzled me when I first started learning about electronics, as to why, having used so much electronics to produce signals, anyone should want to use resistors to resist their flow through the circuit. Fortunately I now know. The purpose of resistors in a computer circuit is mainly to limit current flow to various components, like transistors for example. If we did not use resistors to limit the flow of electricity to certain parts of the computer, the current flowing through them would damage them. Think of this as applied to your TV. turn the volume right up as high as it will go. Can you hear the distortion of the sound? That is because you have removed the resistance of the signal into the audio amplifier, and it is now overloaded. Turn it down quickly before something inside it is damaged. You can't hear them, but similar things can happen to transistors driving lights or relays. When overloaded they get very very hot and eventually cease to function. So resistors are used to prevent overloading. But they can also be used in conjunction with capacitors to form timing circuits, and can be used for pulling logic inputs towards one logic level or another, and for many other functions too. Resistance is measured in Ohms, and each resistor has on it a colour code which indicates how many ohms resistance it has. Fig AP6.14 shows how to use this code. Resistors are available in a number of different wattages. 1/4 watt, 1/2 watt, 1 watt, and 10 watt being the most common. You should always use the wattage specified in the designers parts list for a project.

Fig. AP6.13: Symbols for fixed and variable resistors

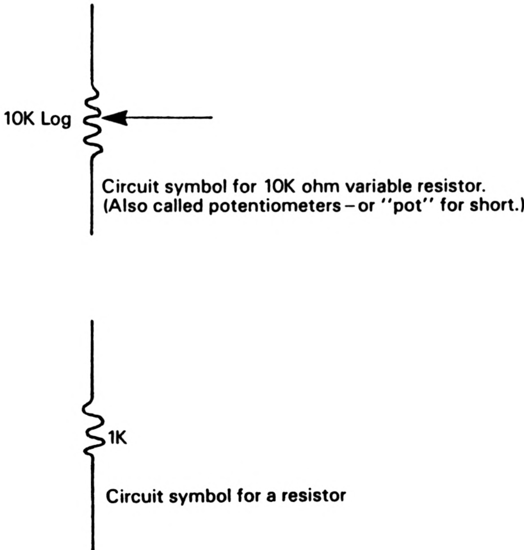
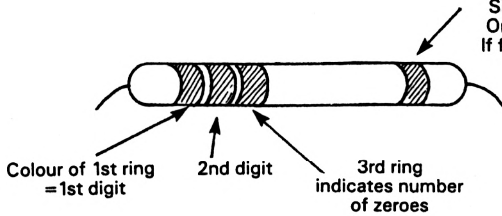


Fig. AP6.14: Resistor Colour Change

separate ring is tolerance specifier
(i.e. how close to value indicated by
the other rings the resistance will be)

Silver=10%: Gold=5%: Yellow=4%
Orange=3%: Red=2%: Brown=1%:
If fourth ring missing assume 20% tolerance



E.g. Red, brown, orange=21000=21,000 ohms.
(Sold as 21K ohms, since in resistor terminology
the letter "K" is used to mean 1000).

Colour	No.
Black =	0
Brown =	1
Red =	2
Orange =	3
Yellow =	4
Green =	5
Blue =	6
Violet =	7
Grey =	8
White =	9

Diodes

A diode is drawn on a circuit diagram as shown in figure AP6.15. The property of a diode is that it will pass electricity in the direction pointed by the arrow, but not the other way. Thus it forms a one way street for electricity. This is very handy when you connect four diodes up to form a bridge rectifier. The AC voltage is fed into two terminals marked AC, and out of two terminals marked + and − will come a DC voltage which when smoothed by a large capacitor, can be used as an input to a voltage regulator. (see chapter three). Elsewhere in the computer, diodes can be used to form OR gates, as shown in fig AP6.16. If the input of any of the diodes is taken high then the end of all the diodes which are joined together, will also be high. Special sorts of diode called zener diodes exist. These diodes, when set into a configuration like the

one shown in fig AP6.17 will provide a reference voltage. The voltage which will be obtained must be specified by you when you buy the zener diode. The one shown in fig AP6.17 is a 3.5 volt zener diode, therefore the reference voltage obtained will be 3.5 volts. Be careful never to overstress the zener diode by making the value of R too small, and the voltage V should always be at least one volt greater than the voltage you wish to obtain as a reference from the zener diode. Another type of diode is the Light Emitting Diode (LED). When current passes through this diode it lights up. LEDs usually need a series resistor, as shown in fig. AP6.18 to limit the flow of current through them.

Fig. AP6.15: Main types of diodes – circuit symbols

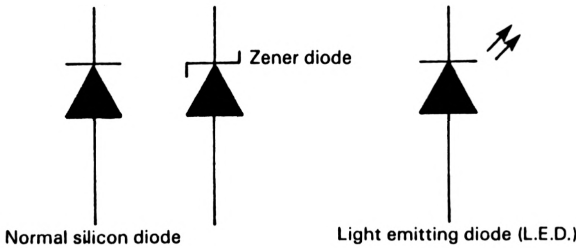
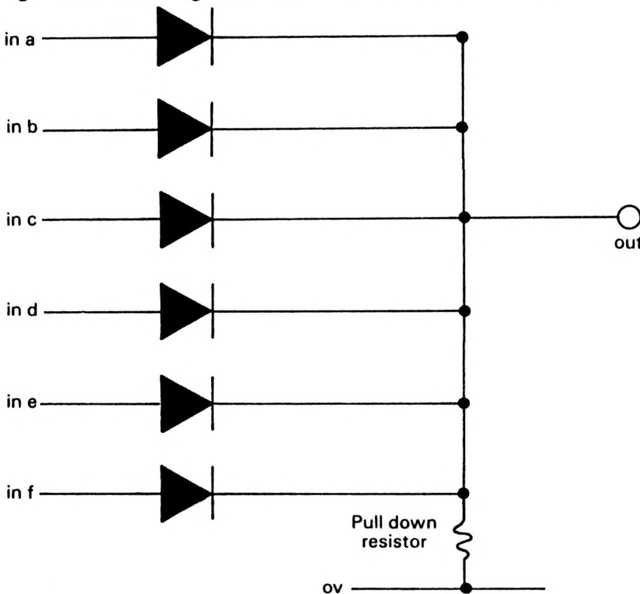


Fig. AP6.16: Showing how to build an OR gate from diodes



Notes: This kind of gate will have a lower performance quality than a chip equivalent to it. Taking any input high will give a high at the output.

Fig. AP6.17: Zener diode circuit

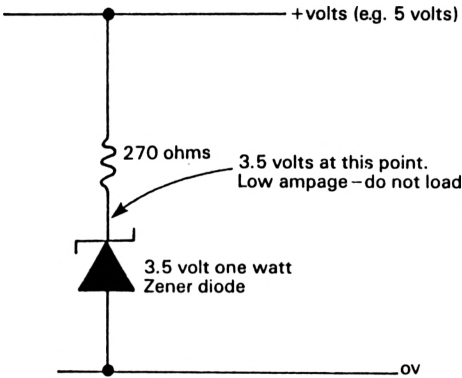
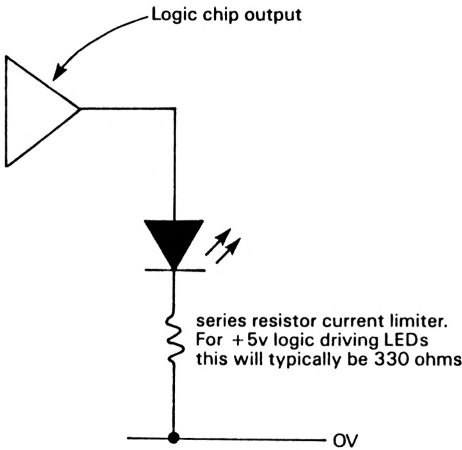


Fig. AP6.18: Typical LED driving



Transistors

These remarkable devices are the stuff of which modern electronics is made. Chips contain lots of tiny integrated transistors, and we also use discrete transistors (that is, one in a package) to interface our relatively small computer logic signals, to power hungry LEDs, relays, or motors and the like. In computers transistors are usually used as switches controlled by logic signals. Refer to figure AP6.19– this shows the symbols for both the basic types of transistor. The NPN transistor is switched on when the base is at a logic one. The PNP transistor is switched on by its base being low. In fact

these descriptions will serve for the purpose of using transistors as logic controlled switches, but please be aware that for other uses far more detail will have to be known about how they operate. When you buy a transistor try to get a data sheet with it. This will only cost a few pence, and will tell you all you need to know about it. One thing you will definitely need to know is which of the leads is base which emitter, and which is the collector. Many leading distributors of electronic components include this pinout information in their catalogues. Discrete transistors can be prone to heat damage when soldering them, so be careful. Fig AP6.20 shows how you can connect an NPN transistor as an inverter. Note the resistor connected to the base, to limit the current flowing.

Fig. AP6.19: Transistor circuit symbols

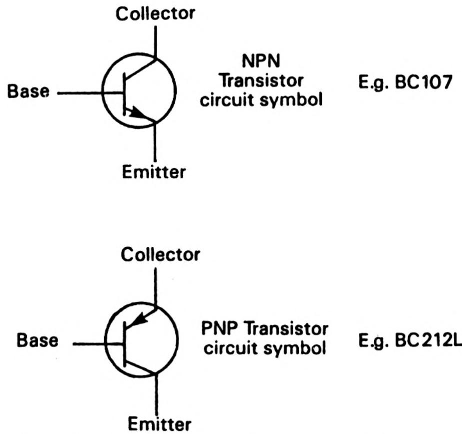
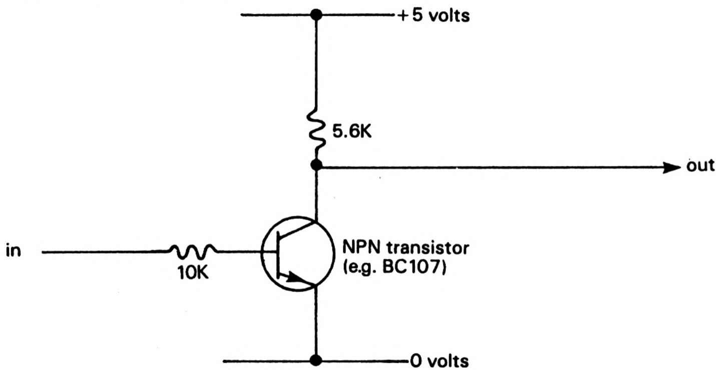


Fig. AP6.20: Using a transistor as an inverter



A high on the input causes the transistor to turn on, this makes the voltage at its collector fall to less than one volt. When the input goes low the transistor is off and the 5.6K resistor pulls the output high. The 10K resistor limits the current flowing into the base of the transistor.

The physical construction process

The methods used to construct the projects detailed in this book are easily described. This is due to the availability of printed circuit boards (PCBs) for them. The PCB designer has created small copper tracks on the PCB to make all the necessary interconnections between the components. All you need to do is solder each of the components into position, then make all the external connections. Having double checked that there are no solder splashes on the board, and that all components have been inserted the correct way round, you've just about finished. If you design some of your own projects you will have to build them on some kind of stripboard. This is a great deal more time consuming than building on a PCB, since all the interconnnecting must be done with tiny wires, and great care has to be taken not to make any unintentional connections. I do not propose to give any further details about building on stripboard, except to advise that for computer projects you always use one of the various microboards which are available.

Soldering

Soldering is easy to do well, unfortunately it is also easy to do badly. The golden rule when soldering is to heat the job, not the solder. Figures AP6.21 and AP6.22 show what is meant by this. You heat the job and use the heat of the job to melt the solder. You DO NOT heat the solder and let the melted solder fall onto the job, this will result in dry joints which whilst they may work for a while, will soon cause trouble. Perfect soldering requires a soldering iron with the correct size bit, and a damp cloth or sponge with which to clean the bit whenever it gets dirty. You cannot make good solder joints with a dirty iron. It should really go without saying that the iron must be good and hot, so the old heat–it–up–on–the–cooker type of iron is definitely OUT.

Fig. AP6.21: Bad soldering

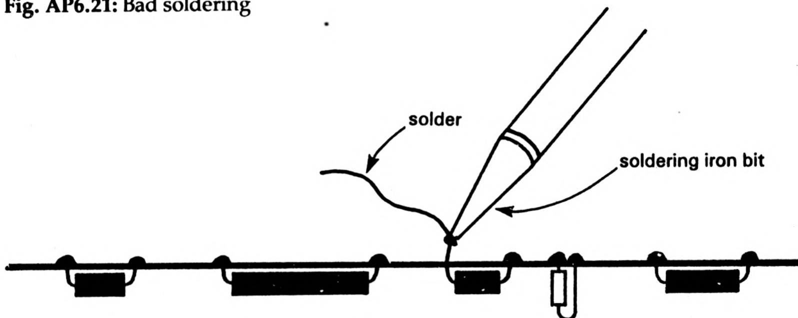
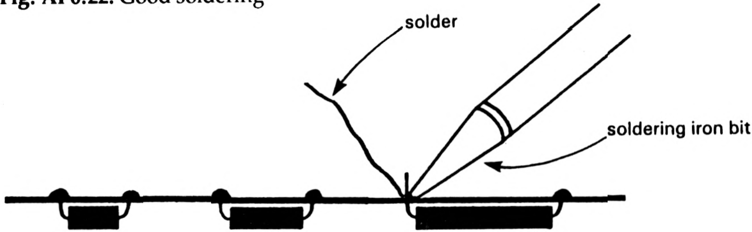


Fig. AP6.22: Good soldering



Pitfalls

This could fill a whole chapter! The major pitfalls in constructing circuitry are really prattfalls, that is to say that with a reasonable amount of care they can be avoided. Here is a list of DO's for avoiding them:

1) DO always use IC sockets, they are cheap and they mean that you can solder the sockets – not the IC, avoiding the possibility of heat damaging the chips. Make sure that you apply the heat of the soldering iron to components for the minimum possible time. Crystals seem especially prone to heat damage.

2) DO make a visual inspection after making each solder joint, to ensure that you have not made a solder bridge to any adjacent connections. Such a bridge could easily prevent the circuit from working properly.

3) DO when designing your own circuits, use components which have a little spare capacity. This applies to extra amperage capacity in power transistors, and also to logic circuitry, try to end up with a couple of spare gates. You may well find that you will need the slack when you have built the circuit, and it doesn't work quite as you had expected!

4) DO always double check for shorts between the +12 volt and −12 volt lines, and any other part of the circuit. A short from +5 volt to some other part of the circuit will probably not be harmful, a short from either of the 12 volt lines will always damage something, and worse still will probably damage your CPC −if connected (see 8 below), so DOUBLE CHECK.

5) DO always include all the components shown in the circuit diagram, unless they are definitely shown as optional. It might seem alright to leave out a decoupling capacitor here, or a diode there, but the designer has put them all there for a reason. If you omit components the long term reliability of the circuit may suffer.

6) DO when building from a kit ensure that you have got all the components. Kit suppliers are sometimes forced to provide equivalent devices to the ones shown in the design. (they usually do this with the permission of the

designer so no compatibility problems should arise.) If this is the case ensure that you know which have been substituted, and that you do not mix all the components up at the insertion stage. It is easy to do, I have done it! Another DO for kit building is to make sure that you read the construction notes carefully. These should come with the kit, or should be with the write up of the project you are building. If there are any wire links to be soldered onto the board, make sure that you insert them correctly.

7) DO always have the ground (or zero volt line from the power supply unit) connected to your circuitry whenever it is switched on. Having +12 volts and −12 volts connected without a ground will effectively place 24 Volts DC across the circuit. It will get hot and be damaged rather quickly, and may well blow back into your CPC.

8) DO always disconnect your circuit from the CPC when you first power it on. This will localise any damage which short circuits might cause. Only when you are sure that all is well connect up the CPC to the circuit.

That about concludes the list of DOs for avoiding pitfalls, the list of DON'Ts is really the opposite of the above list. The only outstanding DON'T is pretty obvious, though people have done it! DON'T connect anything other than the quoted DC voltages to logic circuitry. Connecting it up to any other voltage will leave you with a smouldering heap which used to be a project and a CPC! So please don't do anything silly.

Power supplies

As a practical design for a computer power supply, you are referred to the SIGMA power supply description in chapter three. We shall discuss a few general points here though. Firstly, on the subject of negative and positive voltages. All the voltage levels quoted in a power supply are expressed relative to its ground or zero volts output. The terms zero volt and ground are used to describe the same thing by many people when describing power supplies. The term earth always refers to the earth connection from the mains outlet, which should ALWAYS be bonded, or bolted to the power supply chassis, or case. This will ensure that should the live mains lead ever come into contact with the case, the plug fuse will immediately blow. (If the case is not earthed and this happens then the first person to touch it will be electrocuted.) So the zero volts output of the Power upply Unit (PSU) is the one to which all other outputs are referenced. Sometimes it is very difficult for newcomers to electronics to understand negative voltages. Let us take a non electronic analogy to see if we can clarify this concept.

Imagine a bidirectional blower, of the sort sometimes fitted in offices. Imagine you are standing beside it. It is switched off, so what kind of air pressure do you feel, none do you? We will call this state, no pressure. Imagine now that somebody turns the blower on to suck air out of the office into the street. You can feel a pressure now, it is a negative pressure. Negative

compared to what? compared to the no pressure state which existed previously. Now this same person reverses the blowers action. It is now sucking air in from the street, and blowing it into the office. What do you feel now, a positive pressure? Right. We can understand these concepts, but it is really no different to the negative and positive power supply voltages of the computer PSU. The zero volts is like the no pressure state, the negative volts are like the negative pressure state, and the positive volts are like the positive pressure state. Thinking of the power supply outputs in this way might help you to visualise what exactly the power supply outputs are.

The interconnections from a power supply to the circuits it is supplying are very important. They should be sound, with no opportunities for breaking down. They should be made using wires which are rated at amperages of at least half again as much as the maximum they will ever have to carry. Lastly, they should be as short as possible, this is to avoid noise pickup. PSU lines are very prone to this problem if they are too long.

Serial data transmission

As the final topic in this crash course on hardware subjects, let us take a look at the very important subject of serial data transmission.

The transmission of characters to a VDU (or serial printer which can be interfaced in exactly the same way) involves transmitting the binary code for each character to be displayed from the computer to the VDU. The binary code for each character is allotted via one of the standard code sets in use in the computer industry. The most widely used code is called ASCII. This is an acronym for American Standard Code for Information Interchange. This code set is used on your Amstrad. For example the code for the number zero (0) is decimal 48. (Prove this to yourself by getting the "Ready" prompt from BASIC and typing "PRINT CHR$(48)". This will print the zero character.) The ASCII character set can be found in appendix four of this book.

The ASCII code set uses seven bits, but the characters transmitted to a VDU usually contain eight bits, with the eighth bit either being set to a logic low, or used for other special purposes which we need not concern ourselves with here.

On big computers, which are capable of giving computer facilities to many users simultaneously, the usual method of connecting a VDU (Visual Display Unit) to the computer is via a serial line. Using this method the VDU can be located at any distance from the actual computer, in fact using modem equipment and leased telephone lines, the VDU could be on the other side of the world!

To take a more usual example though, let's imagine a factory which has a computing department on one side of a large building. The managing director of the company decides that she would like a VDU on her desk, so

that she can call up the latest sales figures, payroll details and so on. Unfortunately her office is on the opposite side of the building to the computing department. How do we connect the MD'S VDU to the computer. Do we use standard logic levels and send the signals down an eight wire cable from the computer to her office? We could do, but there are several objections. In a factory there are liable to be a great number of large electrically powered machines, lathes, conveyors, drills etc. These all generate airborne electrical interference – the sort that can ruin a TV picture. If our eight wire cable went near these, the electrical noise which it would be subjected to, might easily change the logic state of one of the bits, and the data would arrive at the VDU corrupted. The other objection, less technical but equally relevant, is the cost of the eight wire cable.

Long ago the idea of transmitting data at TTL levels over very long cables was found to be impractical. This led to the development of a method of transmitting data over just three wires, and at voltage levels which are less likely to be affected by airborne electrical interference. The best known of these standard methods is called RS232.

To transmit data over just three wires from a computer to a VDU, the wires are used as follows:

One wire to be a common ground (or zero volts) connection.
One wire to carry serial data from the computer to the VDU.
One wire to carry serial data from the VDU to the computer.

The other feature of RS232 is that the logic levels travel over the serial link as positive and negative voltages. This level shift is performed by converter chips. These convert standard TTL level signals into positive and negative voltage pulses, ready to travel over the three wire link. To be specific a logic high travels over the link as a negative voltage, and a logic zero travels over the link as a positive voltage. The positive and negative voltages used must be balanced, by which I mean that if you use −12 volts as your negative level, you should use +12 volts as your positive level. The range permitted for the positive voltage is +3 to +15 volts, whilst for the negative voltage −3 to −15 volts is permitted.

Now having established the voltage levels and what we are going to use each of the three wires of the serial link for, we now need to look into the timing of data transmission and reception.

There have been many different chips developed to effect serial transmission of data. The simplest and, until recently, most widely used of these chips, is called a UART. (Wot more initials?? – UART is an abbreviation for Universal Asynchronous Receiver Transmitter.) The UART chip contains all the circuitry required to convert eight bit bytes of data into a serial data stream, and also convert an incoming data stream into eight bit bytes for feeding into a computer. The only extra circuitry required to build a complete serial transmission unit are the level shifting chips mentioned above, and a chip called a BAUD rate generator.

Baud rates and serial transmission timing

As you probably already know, quartz crystals provide very accurate timing pulses when a voltage is passed through them. This has lead to their widespread use as timing elements in digital watches and clocks. They are also used widely in computer circuits which require a signal with a known, accurate frequency. Serial transmission requires such accuracy. The scheme relies on the fact that at each end of the serial link, that is, at the computer end and at the VDU or printer end, an identical frequency is available from a BAUD rate generator chip. (Also sometimes sold as bit rate generators). These chips invariably use a crystal to provide a master frequency which they then divide down to derive one of about sixteen industry standard BAUD rates. (As a rule of thumb think of one BAUD as one pulse per second – or one Hertz.) The UART at each end of the link uses the BAUD rate frequency to determine when the logic level for each of the eight bits of the byte is on the serial line. So long as the exact instant when the transmission of the byte began is known to the UART at both ends, the rest follows. Let us use the CPC to illustrate this. Fig AP6.23 is a listing for a BASIC program which will graphically illustrate a one way serial transmission. The serial transmission concept is adequately shown by this program, but there are two important differences between this and the real thing. Firstly the real thing is two way – there are two interconnecting data lines. Secondly in the real thing some extra bits are transmitted with the eight data bits.

Some of these extra bits are used to allow the receiving device to double check that it has received the data correctly. Another extra bit is the start bit. Do you remember that we said that the UART at both ends of the line must know the exact instant when transmission began? Well the start bit is always sent as a logic 0, whilst an inactive line is always held at logic one (sometimes called marking). After a period of the line being held marking, the receiving UART detects the start of a transmission by the changing of the line to a logic zero. This event starts all the timings upon which the correct transfer depends. The remaining extra bits sent with the data are called stop bits, these are always logic zeroes, and there can be one or two stop bits.

Because the use of RS232 is so widespread on all sizes and makes of computer, it provides a cheap means of exchanging data between machines. A three wire connection between the sending and receiving machine is all that is required. This method of transfer is often used to avoid having to write elaborate software for driving custom designed hardware interfaces.

There are quite a few other points to know about serial data transmission, but the bulk of these are covered in the descriptions of the RS 232 projects in chapter three.

```
100 '        ** Program to illustrate serial transmission concepts **

200      MODE 2: LOCATE 1,5: PRINT "Type any alphanumeric character ";:
         WHILE TR$="": TR$=INKEY$: WEND: IF ASC(TR$) > 32 AND ASC(TR$) <
         127 THEN 300 ELSE PRINT CHR$(7): TR$="": GOTO 200
300      PRINT TR$: BIN2$=BIN$(ASC(TR$)): IF LEN(BIN2$) < 8
         THEN BIN2$=STRING$(8-LEN(BIN2$),48)+BIN2$
400      FOR I=1 TO 8: BINARY$=BINARY$+MID$(BIN2$,I,1)+CHR$(32): NEXT I
410      PRINT: PRINT "Slow or fast demonstration <slow>": WHILE R$="":
         R$=INKEY$: WEND: IF UPPER$(R$) <>"F" THEN SPD=50 ELSE SPD=10
420      ON BREAK GOSUB 32767
500      MODE 1: PRINT "Serial data transmission demonstration"
600      MOVE 0,317: GOSUB 900: MOVE 325,15: GOSUB 900: INK 1,26: INK 2,9:
         MOVE 500,200: DRAWR 40,0,1: MOVER -40,-30: DRAWR 40,0,2 '** DRAW
         the transmiting and receiving registers, set up inks, and then **
610      LOCATE 21,13: PRINT "Logic zero=": LOCATE 21,15:
         PRINT "Logic one =" 'print an explanation on screen for
         each state of the interconnecting line.
700      LOCATE 2,5: PRINT CHR$(34);TR$;CHR$(34);"=ASCII HEX "
         ;HEX$(ASC(TR$)): LOCATE 2,3: PRINT BINARY$
750      IK%=1: GOSUB 21000 'Draw the serial link with ink 1.
760      TOSEND =8: EVERY SPD,0 GOSUB 22000
770      GOTO 770 '** wait until EVERY takes effect **

900 ' ** Subroutine to draw a box repesenting a serial transmission
        register. The bottom left hand side of this register will
        be at the current graphics cursor position. **
1000    DRAWR 275,0,1: DRAWR 0,55,1: DRAWR -275,0,1: DRAWR 0,-55,1:
        DRAWR 0,23,1: DRAWR 275,0,1: RETURN

21000    '**  Subroutine to draw the interconnecting link in the
              colour ink indicated by the variable IK%     **
21010   MOVE 323,50: DRAWR -23,0,IK%: DRAWR 0,310,IK%:
        DRAWR -23,0,IK%: RETURN

22000   '**  Subroutine to advance the serial transmission illustrated
             on the screen by one bit. **
22010   TOSEND=TOSEND-1: LOCATE 2,3: PRINT SPACE$(16-(TOSEND*2));
        LEFT$(BINARY$,TOSEND*2):
        IK%=1+VAL(LEFT$(RIGHT$(BINARY$,16-(TOSEND*2)),2)):GOSUB 21000:
22020   FOR I=5 TO 21: LOCATE 20,I-1: PRINT " ": LOCATE 20,I:
        PRINT MID$(BINARY$,(TOSEND*2)+1,1): FOR P=1 TO SPD: NEXT:
        NEXT: LOCATE 20,21: PRINT CHR$(32)
22050   LOCATE 22,22: PRINT RIGHT$(BINARY$,16-(TOSEND*2)):
        IF TOSEND <> 0 THEN RETURN
22060   LOCATE 22,24: PRINT CHR$(34);TR$;CHR$(34);"=ASCII HEX ";
        HEX$(ASC(TR$)): LOCATE 2,5: PRINT STRING$(16,32): D=REMAIN(0):
        LOCATE 1,12: PRINT"Press any key": WHILE INKEY$="": WEND: RUN

32767   MODE 2: END
```

Fig AP6.23: Serial transmission visualisation program

Conclusion

I am painfully aware that this appendix may well have failed. I cannot teach you all about logic electronics, let alone the other subjects, in so small a space. What I hope you will have learned by reading this chapter are the first concepts. I also hope that you will have been persuaded that hardware is a subject worth delving into. It is, for a number of reasons. The software side of computers is densely overpopulated, whilst the hardware side tends to be

rather neglected. Oh, there are plenty of people around who know about it, but not many of them publish books or sell products for the home computer market. If software is your first love, you'll be able to write better software as a result of gaining a good working knowledge of hardware. And, finally, you can save yourself lots of money by building and designing your own hardware add—ons.

I hope you will go further!

APPENDIX 7

The printer port

The printer port connections to the Amstrad CPC are somewhat difficult when you come to connect a standard ribbon cable connector to it. Reference to the connectional diagram in the appropriate appendix of the computers user manual, shows that instead of the normal arrangement of pin numbers, an unusual numbering arrangement has been used. The net effect of this is that the ribbon cable which you attach to the printer port should have every other conductor connected, instead of the more usual arrangement – as used on the expansion port connector – where the conductors number sequentially. Fig.AP7.1 shows the printer port arrangement.

The printer port is a seven bit port. Bit eight is always set to a logical zero in the data received by the printer.

Another problem regarding the connection of a printer to the Amstrad, and indeed all other computers, is that of escape sequences.

An escape sequence is a feature of many modern printers. Escape sequences provide a way for a program running in a computer, to control certain special print characteristics, or mechanical actions inside a printer. The whole idea hinges on the use of the ASCII escape character. Printers which feature escape sequences regard the escape character as the start of a special command. As most printers now feature microprocessor control, it is a simple matter for each character arriving for printing to be checked to see if it is the Escape character– ASCII 27. If it is escape then the printer microprocessor interprets the next one or two characters received, as parameters for the escape sequence. These parameters will cause some predefined action to take place. Typical ones are ;

> Print all characters received from now on, with an underline.
>
> Print all characters received from now on in darker ink (that is, more boldly).
>
> On dot matrix printers, there may be an escape sequence to specify that all subsequently received characters be printed at double the normal height, or double the normal width.

There are many more features provided on printers in general. Obviously for every special feature which can be turned on, there is always an escape sequence to turn it off again.

The only real snag with escape sequences, is that there is no proper industry standard for them. (The nearest thing is the ANSI Standard – American National Standards Institute.) This means that you must carefully check the printer manual which you get when you buy a printer. Make sure that if the printer is capable of providing special print features, you get the information needed to use them. Bear in mind that a few printers do not have any special features. Many word processor packages allow you to tailor the escape sequences for your particular printer. The AMSWORD package written especially to run on your CPC, allows you to define up to 40 escape sequences. This is more than enough for all but the most sophisticated graphics printers, which would anyway cost thousands of pounds. The golden rule when using escape sequences is to read the manual for your printer, the escape sequences are quite likely to be different from any other make.

One final word on escape sequences. There are now a great many second hand VDUs (Visual Display Units) on the market, at bargain prices. If you buy one of these, try to ensure that you get a copy of the manual for them. Most VDU units manufactured after about 1977 feature at least a few escape sequences. If you are going to connect a bargain basement VDU to the serial interfaces described in chapter three of this book, you could use the effects set up by the use of escape sequences for games, or business graphics. VDUs like the DEC VT52 or the Systime 5753 have many such features. They can't equal the graphics capabilities of your CPC screen, but they could be useful in some applications.

Example of an escape sequence:

To print with underscore on an Epson FX80 printer, from BASIC you should use the sequence;

```
"PRINT #8,CHR$(27);CHR$(27);CHR$(45);CHR$(49)"
```

This will turn on the underline feature. To turn it off again use the sequence;

```
"PRINT #8,CHR$(27);CHR$(45);CHR$(48);CHR$(32)"
```

INDEX

27128 chip . . . 131
6845 chip . . . 12
74LS08. . . . 294
74LS138 . . . 297
74LS193 . . . 294, 295
74LS273 . . . 13
74LS629 . . . 64
8251A . . . 275
8255 chip . . . 72
8255 PPI . . . 1
8255A . . . 273

A/D converter . . . 59
Accumulator . . . 23
Address lines . . . 288
AFTER . . . 193
Allophone . . . 65
Analogue to digital . . . 59
AND . . . 292
ASCII . . . 56, 72, 277, 312
Assembler . . . 195

BASIC . . . 158, 195
BASIC, hardware control with . . . 191
Baud rate . . . 54, 113, 314
BIN$ . . . 6
Binary . . . 287
Blowing (EPROM) . . . 140
BORDER . . . 191
Buffer . . . 48

Capacitors . . . 301
ÇAT . . . 191
Catalogue printer . . . 229
Circuit symbols . . . 266
CLG . . . 191
Clock . . . 11, 52
CLS . . . 191
CMOS . . . 291
Colour codes, resistor . . . 305
Condition codes . . . 27
CPCNET . . . 238
Crystal . . . 60
CTRL . . . 207
Cursor . . . 12

D/A. . . . 59
Data buffer . . . 117

Data logging . . . 89
Data sheets . . . 271
DATA statements (machine code) . . . 225
Datacorder . . . 14
Debouncing . . . 72
Debugging (aids for) . . . 198
DEVPAC . . . 195, 198
DEVPAC, problems with . . . 236
DI . . . 191
Digital to analogue . . . 59
DIL . . . 1, 290
Diode . . . 305
Directive . . . 197
Disk unit . . . 15
Dynamic RAM . . . 8

EOF . . . 191
EPROM . . . 128
EPROM, blowing . . . 140
ESC . . . 194
Escape character . . . 317
Escape sequences . . . 317
EVERY . . . 193
Expansion bus . . . 279
Expansion ROM board . . . 183
External keyboard . . . 71

Firmware . . . 191
Flowchart . . . 199
FRE . . . 191

Gate . . . 292
Gate array . . . 11
GENA3 . . . 198

Hexadecimal . . . 289
HEXBAS listings . . . 225
HIMEM . . . 192
Hisoft . . . 198

Inflection . . . 67
INP . . . 192
Instruction set . . . 24, 26
Interrupt . . . 91, 235
Interrupt vector . . . 24
Interrupts . . . 193

Key bounce . . . 72
Key matrix port . . . 77

Keyboard matrix 7
Keyboard, external 71

Label 197
Latch chip 13
LED 55
Light emitting diode 55
Light pen input 11
Line driver 276
Local network 238
Locomotive BASIC 191
Logic gate 292
Logic symbols 267

Machine code 196
Masking 215
MC1488 276
MC6845 272
MEMORY 192
Memory map 13
Minefield (game) 82
Mnemonic 197
MODE 192
MODEM 112, 279
MONA3 . . . . 198, 236
Multichannel VDU 112

NAND 292
Network project 238
NOR 292

Object code 196
OR 292
OUT 192

Parallel data 89
Parity 113, 50
PCI 101, 275
PEEK 192
Pin connections 277
POKE 192, 195
Port, printer 317
Ports 1
POS 192
Power supply 43, 311
PPI 1, 58, 273
Printer buffer 48
Printer port 317
PSG 3, 8

RAM 8, 286
Registers 3, 24
Regulator 45
Resistors 304
ROM 7, 286
ROM board, expansion 183
ROM disk 145
RS232 . . . . . 46, 99
RS Components 266
RS232, pin connections 277

SAVE 192
Sense register 86
Serial data 47, 99, 312
Signal timings 20
SOFT 158 199
Sound generator 3, 8
Speech synthesiser 58
Split speed working 100
SPO 256 . . . . . 60, 271
Stack 23
Stack pointer 23
Static RAM 8
Statistics, gathering of 87
Stop bit 52
Strobe 91

TIME 193
Timings (Z80A) 20
Transistors 307
Truth table 3
TTL 291

UART 314, 55
USART 101

VDU chip 11
VDU, multichannel 112
VPOS 192

WIDTH 192

XOFF 47
XON 47

Z80 programming 22
Z80A 1, 16, 274
Zener diode 305

Taking the next step ....

The novelty of using a microcomputer just for games and screen-based programs soon wears off. How nice it would be, you think to yourself, to use a micro to do something *useful.*

But, the most useful things involve using a micro to "listen in" to the outside world and, as a result, to take suitable actions. That's when you need to add extra hardware to your 464, 664 or 6128 *and* you have to know how to control it all

So, this book starts with an easy to learn but *in depth* approach to understanding how your Amstrad computer works and the Z80 processor inside it. You'll quickly learn where the important pieces of hardware are located - the RAM, ROM, sound chip, VDU controller and the other special Amstrad chips. With this knowledge, you'll be in full control of all the "on-board" hardware, and ready to move on to the next part of the book.

This is where the fun starts. You'll use your knowledge of the Amstrad plus some inexpensive components in exciting projects that include:

- An external keyboard.
- A 4-Channel RS232 Interface.
- An expansion ROM board.
- A speech synthesiser.

Advanced projects for the more adventurous include.

- A complete EPROM programmer.
- A *local area network* for your office, factory or school.

Worried about your electronics knowledge? There's no need - not only are all the necessary concepts introduced, but readers can obtain ready made *printed circuit boards and complete kits of parts* exclusively from HALSTEAD DESIGNS (See page iii).

About the Author

Alan Trevennor is a Field Service Engineer with Systime Computers PLC. He has already written a successful Sigma book "Operating Systems - a user friendly guide".

About Us

Sigma Press are still growing - right up from hobbyist to professional applications. For a catalogue, or to tell us about the book you've always wanted to write, contact Sigma at:

5 Alton Road
Wilmslow
Cheshire SK9 5DY

GB £ NET +008.95

ISBN 1-85058-018-9

Understanding and Expanding your Amstrad CPC-464/664/6128

Alan Trevennor

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.
[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.
[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.