ZX.ASZMIC
USER MANUAL

C FRAZER JOHNSON

Comprocsys limited

ZX.ASZMIC

ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM

C FRAZER JOHNSON

COMPROCSYS LTD.
29 CAMPDEN ROAD,
SOUTH CROYDON,
CR2 7ER
01-688 4696

LIABILITY NOTICE

Because Comprocsys Ltd and their distributors have no control over the circumstances of use of this product no warranty is given or should be implied as to the suitability of this product for any particular application and no liability can be accepted for any consequential loss, however caused.

Published for Comprocsys Ltd by
Microsource, 1 Branch Rd, Park Street,St Albans.

Copyright 1982 Comprocsys Ltd.

All Rights Reserved

This documentation, and the computer program or code to which it refers are copyright, and reproduction in any form is forbidden without the express written permission of Comprocsys Ltd.

Comprocsys Ltd,
29 Campden Rd,
South Croydon,
Surrey, CR2 7ER.

CONTENTS

Page

5 Chapter 1 Insertion & Overview

7 Chapter 2 First Steps

17 Chapter 3 First Steps Continued

23 Chapter 4 More Debug Operations

31 Chapter 5 Text Operations

39 Chapter 6 The Assembler

53 Chapter 7 Program Execution & Test

59 Chapter 8 Graphics

65 Appendix 1 General Information

71 Appendix 2 The Shift Keys

75 Appendix 3 Debug Commands

85 Appendix 4 System Addresses

87 Appendix 5 Z80 Instruction Set

99 Appendix 6 Application Notes

Chapter 1

INSERTION AND OVERVIEW

Congratulations on your decision to buy ZX.ASZMIC. If you have had previous experience of assembly and debug systems you may wish to proceed immediately to the first 3 appendices, which summarise the ASZMIC features for you. Otherwise the next 3 chapters contain a series of worked examples to help give you a feel for ASZMIC usage. Chapters 5,6 & 7 are a more general discussion of the features; chapter 8 introduces you briefly to the graphics possibilities; and appendices 4 thru 6 contain useful reference information.

What exactly is ASZMIC? It was designed to be an ultra low cost program development station for assembly language. As such it gives you all the facilities you need to write and edit programs, assemble them and then run and debug them in a controlled environment. Since the programs may also be run on an '81 fitted with ASZMIC we tried to document some of the useful routines in the monitor itself to help free you from that curse of assembly language programming....Input/Output...., and we parametrised the display routine to release the graphics possibilities which were latent in the hardware. Unlike some of the other offerings on the market ASZMIC is almost completely independent of Basic, and you should bear in mind that the personality of the computer is completely transformed. ASZMIC tips its hat in the direction of Cambridge and then does its own thing. If you are experienced with microcomputer development systems then much of ASZMIC will be familiar to you; but if you are used only to Basic then ASZMIC will give you a power which you could never achieve when everything you did was filtered thru the Basic interpreter. This is unfortunately a power for both good and evil; since ASZMIC does what you tell it to then a command which is, shall we say less than wise can easily wipe out the system. Fortunately you will not do this very often.

1.1 INSTALLATION

ASZMIC is a direct replacement for the Basic ROM in the ZX81. To insert it in your computer you must first seperate the case by removing the 5 screws from the base of the ZX81; three of which are concealed under the peel-off rubber 'feet'. Then remove the three screws which hold the printed circuit board onto the lid and,taking care not to pull the 'tails' from the touch panel out of their holders, fold back the circuit board and rest it and the lid on a clean surface. If you do pull out the 'tails' they can be pushed back in without too much difficulty.

If you now look at the component side of the board; calling the edge with the edge connector 'top', that closest to the keyboard 'bottom' and the edge with the power, cassette & T.V. connections 'left', then the ROM lies at the bottom of the board. You can look at the diagram on p. 162 of the ZX81 manual to see the ROM identified (and beware, in the picture it is inserted upside down). Although modern technology has made MOS circuits much more tolerant than they used to be you may feel more secure working on a large sheet of aluminium kitchen foil which is connected to a radiator or earth of some sort, or a metal draining board is excellent. Avoid wearing nylon clothing and try to earth yourself before touching any circuit not in its socket on the board. The Basic ROM will probably be in a 28-pin socket, even though it has only 24 pins, so there will be four empty 'holes' in the socket above the ROM. ASZMIC also has 24 pins so when it is inserted it also should be positioned at the bottom of the socket with four empty 'holes' above it. ASZMIC should be inserted so that the 1 on the label lies towards the left. Remove the Basic ROM by gently levering it up from top and bottom with a fine screwdriver until it lies loose in the socket, and then lifting it away. If you are too hasty you may bend the pins. Put it on a little bit of foil. ASZMIC's pins may now need to be bent slightly inwards so that you can press it into the socket, but not too much since the socket generally makes contact on the outside of the pins and you run the risk of a bad contact on one or more pins if you

bend it too much. Push ASZMIC carefully into place. Before putting everything back together again you might like to connect up the power and T.V. lines and check that you have a blinking cursor. If not then you have a bad connection and you should try to remove and replace ASZMIC a couple of times.

If you plan to use both ASZMIC and Basic alternately then you may wish to invest in one of the boards available which will hold both ROMs. Capital Computers Ltd can give you details. When we were developing ASZMIC we cut a hole in the lid of an '81 (from 4 to 6 cms back from the touch keyboard for about 5 cms and if you cut more than a mm into the keyboard you will kill it) and piggy-backed a zero insertion force socket onto the ZX81 ROM socket. This worked surprisingly reliably, but a dual ROM board will give you the ability to preserve regions of memory when you transfer control.

1.2 ZX80 INSTALLATION

The circuit board on the ZX80 is more accessible than on ZX81; you just have to remove plastic pins to release the lid and the ROM lies on the right. ASZMIC should be inserted with the 1 closest to the UHF modulator (metal box). You will not be able to use the G command with a ZX80, and a steady display will not be maintained under Edit operations.

Some ZX80's suffer from excessive load on the A12 address line which can lead to difficulties in reading the key group 6 thru 0. This can be cured by soldering a 4.7K resistor between the left hand (keyboard) side of D7 to the earth plane below IC11 & IC17.

Chapter 2

FIRST STEPS

2.1 POWER UP

After connecting the power to your ZX80/81 you will have a clear screen, except for a funny little character towards the bottom left (End-of-Data) and a blinking cursor on the line just above it. The speed of blink identifies which mode ASZMIC is in..... Fast blink means EDIT, slow blink means DEBUG. With 16k of memory on the system you will be in DEBUG mode but if you are using a "bare" machine then you will be in EDIT mode. Shifting between EDIT & DEBUG modes is achieved by use of the Shift 9 & Shift E keys. Experiment by pressing Shift 9 (DEBUG) and Shift E (EDIT) alternately, and watch what happens to the cursor blink rate. There is only one difference between EDIT and DEBUG modes, but that is an important one. When you hit newline (we shall write newline as /NL/ in future) in DEBUG mode the line you have just finished will be passed to the Command Interpreter. In EDIT mode you just start a new line.

2.2 EDITING

Press Shift E to get yourself in EDIT mode. The cursor will be blinking quickly. Now type A. An "A" appears on the screen & the cursor is advanced to the right. Type A again, but this time leave your finger pressing on the key. After half a second ASZMIC will start writing A's at a rate of 8 per second. Take your finger off the key before the line is full. This illustrates the very useful key repeat feature built into ASZMIC. It works on all keys, including /NL/. Now type /NL/ & then press

B for a second or two, type /NL/ and press C for a second or two, type /NL/ and repeat the process till you have 5 or 6 lines on screen. Notice how the display is scrolled up with each /NL/. Now type a final line of, say, G's but do not press /NL/ at the end. Instead press Shift 5 (CURSOR LEFT) & keep your fingers on the keys. The cursor will move left along the line till it comes to line start. Press Shift 8 (CURSOR RIGHT) continuously & watch the cursor advance along the line to line end. Press Shift 5 (CURSOR LEFT) again until the cursor lies in the middle of the line. Type 12345. The figures are inserted in the middle of the line. What actually happens is that everything under and right of the cursor is shifted right, the character you have typed is placed under the cursor, and then the cursor is shifted one place to the right.

2.2.1 RUBOUTS

Press Shift 0 (RUBOUT). The 5 you have just typed will disappear. This is the first type of rubout in ASZMIC, and is the typing error rubout. The character to the left of the cursor is deleted and everything past it moved left one place. Press Shift 5 twice to position the cursor over the 3. Press Shift Q (EDIT RUBOUT) and watch the 3 disappear. This is the second type of rubout; the editing rubout. The difference is that Shift 0 is more convenient when you are typing in a fresh line, and Shift Q is better when editing a file. Experiment a bit with them if you like. Edit Rubout does not work when there is only one character remaining on a line. You can rubout /NL/'s with Shift 0. Beware that Shift-Q can also delete the "padding" blanks that ASZMIC inserts before a /NL/, which can give problems with assembly & merging.

2.2.2 VERTICAL CURSOR OPERATIONS

Reset your ZX80/81 (turn off & on again). We ask you to do this quite often because if you have only 1K of memory these examples will soon fill up the text area.

for many ASZMIC operations. Now press Shift T (TOP). The cursor and display jump up to the very top. This is useful if you want to search down through large texts, but is most important because the top line has a special significance. It is the Shift Macro line, but more of that later. Now press Shift 9 (DEBUG) and notice how you have jumped down to the bottom line and entered DEBUG mode. Press Shift E (EDIT) and note that you have not only entered EDIT mode (fast blink) but also jumped back to the point where you were when you last pressed Shift 9. This enables you to toggle between EDIT and DEBUG modes without having to search for your place all the time.

Enter DEBUG mode with Shift 9. Type D 0/NL/ .
ASZMIC responds with 0000 F5
Keep your finger on the /NL/ key until ASZMIC has printed 0030 3E and then type . (period) followed by /NL/ followed by E/NL/ to get back to EDIT mode. You have actually done a Dump & Modify of the first 49 monitor locations but we want it just to have some text to play with. Now press Shift 7 (CURSOR UP) and hold it down. Watch the cursor move up the screen until it reaches the top, and then watch the text scroll down until the cursor has moved onto a blank line. Now press Shift 6 (CURSOR DOWN) and watch the cursor move down until it is on the final line just above the End-of-Data character. It will not move down any further. Now press Shift 4 (PAGE UP) a couple of times and see how you flip up through the text. Press Shift 3 (PAGE DOWN) to reverse the process.

2.2.3 DELETING A LINE

Move the cursor up to 0020 7E or thereabouts. Now press Shift 1 (DELETE LINE). The line will vanish. Repeat the process and successive lines will vanish until you are back down above the End-of-Data character. You cannot delete this final line with Shift 1; you have to rub it out character by character. This is to protect the vital End-of-Data character which is used as a "stop"

the dump. Now press Shift 2 (delete file) . All the text from the cursor to the filemark will have been wiped away.

2.2.4 FILES & FILEMARKS

Unlike the BASIC rom, ASZMIC does not construct a special display file for you, but instead uses most of memory as one big text area over which you move the T.V. screen as a sort of window. A lot of operations are controlled by a special character, the £ (sterling) sign, which acts as a delimiter. Since blocks of text seperated by £ signs are treated as files, we call the £ a filemark. It is used by itself to indicate the end of a file, and is needed for print, assembly, cassette save, merge, and file deletion operations to tell ASZMIC that it is time to stop. You indicate the start of a file by a name which is particular to it & does not also occur in the body of the file. It is good practice to have a filemark as the first character of the name. Thus:-

```
£NURSERY.RHYME
1
2
BUCKLE MY SHOE
3
4
KNOCK ON THE DOOR
£
```

is an example of a file, and can be printed, saved, edited etc. by referencing the name £NURSERY.RHYME . A filename can be any combination of alphanumeric characters (0-9 A-Z) and . (period) and is terminated by any other character.

Home the cursor (Shift 9) & type £/NL/ . Then press Shift E to get into EDIT mode & move the cursor till it blinks on the D of your original D 0 line at the top of the dump. Now press Shift-2 (delete file). All the text from the cursor to the filemark will have been wiped away.

2.2.5 RUNNING OUT OF SPACE

Reset the ZX80/81 and hit Shift 9 (DEBUG, as you probably know by now). Type D 0 5000/NL/ . The screen will go blank for several seconds because ASZMIC is generating 625 lines of formatted dump for you. Unfortunately it would take more than a 16k memory pack to contain so much text, even if ASZMIC had not divided up your memory into 3 parts for text and 1 for programs. When the display comes back try typing in a character. Nothing happens. You can move the cursor around (try it) but you cannot insert any fresh text. To use ASZMIC further you must delete some text. Move the cursor up 4 or 5 lines (Shift 7) and then hold Shift 1 continuously depressed. You will delete the last few lines. You can now insert characters (100 or more) until you run out of text space again.

2.2.6 MACROS

We may as well use this mass of text for something. Press Shift T. This takes the cursor up to the top line. Now type E 40 but, since you are in DEBUG mode, do not press /NL/ to execute the command but instead press Shift 9 (DEBUG) to home the cursor on the bottom line. Now press Shift R (Shift macro).
The effect of this is to execute the top line as a DEBUG statement, irrespective of what mode you are in. E 40 is a DEBUG command to enter EDIT mode and position the cursor at the start of the first occurrence of the string "40" above its current position, so you will see the cursor blinking quickly on the "4" of a "40". Press Shift R several times and see how each "40" in the dump is successively searched out. You can have almost any DEBUG command or concatenation of commands on the top line and this is a way of doing a lot of work with a single keystroke. (there is another sort of macro, a Command Macro, built into ASZMIC and we will look at it when we discuss file merging). Now delete the dump by pressing Shift 9, typing £/NL/ (£ is not a valid DEBUG command and is ignored by the Command Interpreter),

pressing Shift T, and then holding down Shift 6 to move the cursor down to the "D" of D 0 5000,and then pressing Shift 2 (delete file). You should have cleared out everything.

2.2.7 MERGING

Reset the ZX80/81. Type Shift E (EDIT). Type the following:-

```
>D 0 1/NL/
D 0 2/NL/
D 0 3/NL/
D 0 4/NL/
D 0 5/NL/
£/NL/
```

Now press Shift G (merge) once. Everything between the > and £ characters has been duplicated. Try it again. Every time you press Shift G a further 5 lines are copied down at the cursor position. Move the cursor up to the D of a D 0 3 and press Shift G. The 5 lines have been inserted before the D 0 3, because that is where the cursor was. Now type Shift 9 (DEBUG).

In EDIT mode the Shift G is a pure merge key. Now that you are in DEBUG mode the effect is rather different, because every time ASZMIC writes a /NL/ to the text area in DEBUG mode it passes control to the Command Interpreter. Press Shift G. Every "D" line has been copied, but since each line is a dump command, it is followed by the appropriate dump. Just by pressing 1 key we have executed 5 lines of DEBUG commands. This is called a Command Macro.

Normally in EDIT mode merging you will want to copy one piece of text into another. You do this by identifying the text to be moved by the > and £ characters, putting the cursor at the point you want the text merged into, pressing Shift G, and then, if you want the original text deleted, positioning the cursor on the > character

and pressing Shift 2 (delete file). Simple, flexible, and, for the poor sweated labourer who wrote ASZMIC, easy to implement. Check Appendix 4 for the symbols SHFTD & SHFTF. If they appear then you have three merge keys. Shift-D is like Shift-G but uses * as merge start identifier, and Shift-F uses < .

2.3 WHAT NEXT?

You have now covered all the Shift keys which control the EDIT functions. If you look at Appendix 2 you will see a formal summary of each Shift. Try setting up some text of your own and working with it. Then look at Appendix 3 for a summary of the DEBUG commands. We are going to look a little more closely at them, although not in alphabetical order, since some of the commands are more complex in their implications than others.

Chapter 3

FIRST STEPS CONTINUED

3 SIMPLER DEBUG COMMANDS

Everything that you can do in EDIT mode you can also do in DEBUG mode. The difference is that the Command Interpreter is called in DEBUG mode when you press /NL/. However if you do any editing in DEBUG mode then you cannot rely on Shift E (EDIT) to take you back to the point where you exited from EDIT mode. It will take you back to the location in memory where the cursor WAS, but you may have edited in something different there (such as a /NL/, in which case the cursor will not appear).

3.1 D for DUMP

Reset your ZX81. Hit Shift 9 to ensure you are in DEBUG mode. Type D :4300 12/NL/ and you will get a display of the form:-

```
4300 00 00 00 00 00 00 00 00
4308 00 00 00 00
```

This is an example of a Dump Range command. It introduces an important new idea, that of a FIELD. The command itself is just a letter D, followed by a field which defines the starting address for the dump and then a field which defines the number of bytes which are to be dumped from successive locations. The : (colon) identifies the 4300 which follows as being a hexadecimal value. Look at the definition of the field types in ASZMIC which you will find in Appendix 1. The use of fields defined there is common throughout ASZMIC

with one exception which we shall discuss in a moment. Fields are seperated by one or more blanks or a comma, but not both. The first field can start immediately after the DEBUG command letter, but an intervening blank often looks better.

There is another form of dump which we call Dump & Modify. Type D :4300/NL/
ASZMIC responds with:-

```
4300 00
```

and the cursor is waiting on the line for input. Depress /NL/ 2 or 3 times. With each press the next location and its contents are printed out thus:-

```
4301 00
4302 00
4303 00
```

How do you get out of this? Press . (period) followed by /NL/ and you are back in ordinary DEBUG mode again. So where does the Modify come in? Type D :4300/NL/ again. Now type 1 4 7 C 12/NL/ followed by . (period) and /NL/. Then do a dump range of the form D :4300 6/NL/ . The response is:-

```
4300 01 04 07 0c 12 00
```

Compare this with the result of the first D :4300 12 that you did. Dump & Modify is the exception to the ASZMIC field rules that we mentioned earlier. Numbers are ASSUMED to be hexadecimal even without the : (colon) prefix. If you want to type in a decimal number use 0+decimalno. For a more dramatic (?) demonstration reset your ZX81,enter DEBUG mode, type 15 or 20 £ signs, then look for the address of DSPBGN in appendix 4 and add 55 to it (e.g. :40B4+55). After terminating your ££'s line with /NL/ type D :40B4+55/NL/ & then 1C/NL/ 1D/NL/ etc. You can see the £ signs being overwritten with the characters corresponding to the codes that you typed in.

3.2 C for COPY

Reset the ZX81 (if you have 16K of memory you need not do this all the time). Enter DEBUG Mode (Shift 9). Type the following commands:-

```
D :4300 20/NL/
D :4300/NL/
1 2 3 4 5 6 7 8/NL/
./NL/
D :4300 20/NL/
C :4300 :4308 8/NL/

D :4300 20/NL/
```

Take a deep breath and look at what you did. First you dumped 20 locations which were all 0 (the machine zeroes memory on reset). Then you did a Dump and Modify of the first 8, and a Dump Range to verify that they had been modified. You then copied 8 bytes from the location :4300 to the location :4308 (the bytes are counted UP from the addresses you give) and finally you dumped the 20 locations again to verify that the copy had taken place.

The format for Copy is C from to count. It has logic in it to ensure that if you specify overlapping memory regions then the data moved to the destination region is not corrupted. The copy goes from top to bottom or vice versa as required to ensure that the source region is not overwritten by the destination region before the source bytes have been copied.

3.3 F for FILL

Reset the ZX81 & enter DEBUG mode. Type

```
F :4300 :4310 :AA/NL/  .
```

Then type D :4300 16/NL/ and note that you have filled the region with AA codes. The format is F from to fillerbyte . The command is most often used to

initialise regions of memory (notably to zero them out, zero is the default fillerbyte if you do not specify a third field)

3.4 E for EDIT

E by itself just shifts back to EDIT mode. If followed by a symbol then ASZMIC searches up for the symbol and positions the cursor at its start. Great for getting to a file when you have a lot of text. You have already used this command in 2.2.6 MACROS so we will not give any examples. Look at the description of the subroutine CMPSTR (used by E & many other ASZMIC commands) to see what constitutes a valid comparison.

3.5 H for HORRIBLE JUMP

The format for this is H field and the result is a jump to the address specified by the field whilst still in the context of the Command Interpreter. You presumably have a program at the address which does something for you and then RET's to ASZMIC. Your routine can analyse further fields on the line using GETFLD, so this is a way of linking in your own personalised commands. If you want an example try looking up the address of INICON in Appendix 4 and then doing an H to that address (remember the :). INICON is the start of the ASZMIC initialisation so the effect should be the same as resetting the ZX81.

3.6 M for MACRO

This is an extension of the Shift G key. Reset the ZX81 and enter DEBUG mode.
Type:-

```
=D 0 8/NL/
£/NL/
+D 16 8/NL/
£/NL/
```

Then type M=/NL/ a couple of times and M+/NL/ a couple of times. The dumps specified are generated. The character after the M specifies the start character for the command macro, unlike Shift G , which always uses > as a start character. You thus have a wide choice of macros.

3.7 O for OLD REGISTERS

In DEBUG mode type O/NL/. You will get a pair of intimidating lines each with six 4-digit hexadecimal numbers on it. These are the registers which are saved in the register context area REGIM. See the definition of O in Appendix 3 to tell you which is which. They do not mean too much until you start executing programs and generating Breaks in them, which causes the registers to be saved in the Image area. It is a very good idea to put an O in the Shift Macro line when you start a session with ASZMIC, since the Shift Macro line is executed for every break condition, and you normally are interested in register contents when a break has occurred.

If you are feeling dynamic try looking up the addres of REGIM in Appendix 4 and then modifying some locations between REGIM and REGIM+24 and noting the effect on the Old Register dump lines.

3.8 P for PRINT

This command will do precisely nothing for you if you do not have a Sinclair printer attached. If you do then reset the ZX81 and enter EDIT mode. Type:-

```
£PRINT.FILE
ANY
SORT
OF
RUBBISH
WILL DO
£
```

and then hit Shift 9 to enter DEBUG mode. Type P £PRINT.FILE/NL/ and the file will be printed. If you want to terminate the printing prematurely (e.g. if you forgot the £ sign which terminates the printing) you just have to hit the Break key.

This concludes the simpler DEBUG commands. In the next chapter we shall be looking at some of the more complex, and interesting, ones.

Chapter 4

MORE DEBUG OPERATIONS

4 SAVING & LOADING

It is now time to look at the ASZMIC cassette interface. This uses the same recording format as the BASIC Rom, so that it is possible, using some tricks described in the chapter on Text Ops, to read in a program saved by BASIC, modify it with ASZMIC e.g. writing REM's full of machine code and then write it back so that it can be loaded by BASIC again. The price you pay for this compatibility is the dreadful slowness of the interface, but of course you are used to that by now.

When ASZMIC writes out a file to cassette it first writes out a title line which identifies the program or file which follows, waits 5 seconds, and then writes out the file or memory region you specified. When you are loading from cassette ASZMIC detects the title line and writes it to screen, using the 5 second pause before data to display it to you. You thus get a catalog of everything on the tape built up for you, which is informative and soothing because you know that your system has not disappeared into one of those black holes which lurk around cassette reading.

4.1 K for KASSETTE (sic)

Reset your ZX81, enter EDIT mode and type in a file, such as:-

```
£LIMERICK
THERE WAS AN OLD MAN OF DUMBREE
WHO TAUGHT LITTLE OWLS TO DRINK TEA
FOR HE SAID TO EAT MICE
IS NOT PROPER OR NICE
THAT AMIABLE MAN OF DUMBREE
£
```

Now enter DEBUG mode, connect your cassette recorder just as you do for BASIC, and type in :-

```
K£S "POETRY" £LIMERICK/NL/
```

The sequence

```
K £ S space "
```

is very important. If you do not write it like that, with only 1 blank between S and ", then ASZMIC will not recognise it as a file title and you will never ever (unless you are very smart) be able to read the file back.

After you hit /NL/ you have 5 seconds to turn on your cassette recorder. It does no harm to turn it on early. The screen then blanks out for half a second and the title line is written out. The display returns for a further 5 seconds and then the file itself is written out. The display comes back again when the save is complete. You can abort a save at any time with the BREAK key. Stop the recorder .

Now type F :4300 :4320 :BB/NL/ to set a region of memory to "BB" codes. Do another save, but this time of a memory region:-

```
K£S "MEMSAVE" :4300 :4320/NL/
```

Cassette operating procedure is as above. Rewind the tape.

4.2 L for LOAD

Reset the ZX81 and enter DEBUG mode. Type:-

```
L "NOTHING"/NL/
```

and start the recorder in playback mode with the same volume setting that you use for BASIC. After a while the screen will light up with:-

```
K£S "POETRY" £LIMERICK
```

for 5 seconds and then blank out again. A little later there comes another 5 second burst but this time the line:-

```
K£S "MEMSAVE" :4300 :4320
```

has been added to the display.Since there is no file called "NOTHING" on the tape you may as well hit BREAK to come back in EDIT mode. Now rewind the tape, enter DEBUG mode and, using the same procedure as above but with:-

```
L "POETRY"/NL/
```

to load the file you first saved. Check that it is O.K.

Reset the ZX81, enter DEBUG mode and rewind the cassette. Check that :4300 to :4320 contains zeroes (D :4300 32/NL/) and load the second file with:-

```
L "MEMSAVE"/NL/
```

Dump :4300 to :4320 to verify that it now contains "BB" bytes.

EASY?

4.5 A for ASSEMBLE

The ASZMIC assembler was developed from a 1K assembler (yes, 1K) written for NASCOM 1 and normally called, with mixed emotion, the "Dirty Dog" assembler.It is a 2 pass assembler with full Zilog mnemonics. If you are not familiar with Zilog's assembly language then it might be a good idea, to put it mildly, if you bought a book on the subject. There are references in Chapter 26 of the ZX81 manual.

Reset the ZX81, enter EDIT mode and type the following sequence (we are going to assume that you know enough to terminate each line with a/NL/):-

```
£FILE
 ORG :4300
START LD HL,0
 LD DE,0
 LD B,10
LOOP INC DE
 ADD HL,DE
LOOPEND DJNZ LOOP
 RST 0
£
```

This is a program. Please always start every program with an ORG directive to tell the assembler where it should be located. It will probably start at (TXTLIM) if you do not, but that default may not necessarily be in your version of ASZMIC. Now we are ready to do an assembly. Enter DEBUG mode and type:-

```
A £FILE 1
```

The "1" is an option to say that you want an assembly listing. Almost instantaneously ASZMIC comes back to you in EDIT mode (It can assemble at up to 300 statements a second) with a listing showing the code generated for each statement, plus the location in memory it has been assembled at. Enter DEBUG mode and dump the program (D :4300 14) to see that it really has

been assembled for you. We call this generated machine executable code "object code".

Now delete all the text in the text area, but do not reset the ZX81 because we do not wish to lose the program.

4.6 J for JUMP

We are now going to execute the program. It generates the sum of the numbers from 1 to 10 in the HL register. To help us see what is happening we are going to use a Shift Macro which is automatically executed whenever a BREAK condition is encountered. (BREAK means breakpoint, single step, RST 0 code or externally generated NMI interrupt). Hit Shift T, type 0 (not 0) without a newline and then Shift 9 to come back in DEBUG mode. Check Appendix 3 for the O command if you have forgotten it. Type:-

```
J :4300
```

You now have a dump of the registers at the end of the program. Look at HL. It should contain :0037, which is the hex equivalent of decimal 55. DE should contain :000A which is 10 of course.

4.7 B for BREAKPOINT

Let us now try to execute the program again, but this time with a breakpoint inserted to stop execution "in midstream". Type B LOOPEND/NL/ (the DJNZ instruction) and then J START /NL/. Now take a look at the registers. Both DE and HL should contain :0001, and B should still contain :0A, since the breakpoint takes effect before the instruction at whose location it was inserted is executed.

Breakpoints work by saving the byte at the breakpoint location and substituting a RST 0 (:C7) code. When you hit a breakpoint in the course of normal execution the "normal" byte is put back there (check with D LOOPEND

1). Now type B /NL/. Since you did not specify an address the breakpoint was reinserted at the previous breakpoint location (check with D LOOPEND 1 again). Type B 0/NL/ to clear the breakpoint.

4.8 G for GO

Now type G/NL/. You have advanced one instruction. Type it a couple more times.This is called single stepping. You can see the Program Counter (PC) changing as well as the registers you are working with. Since we did not specify an address with the G command it used the saved Program Counter address in the register image area (PC1 in REGIM).

Now why not try executing 20 instructions before jumping back to the monitor? Type G START 20/NL/ . The "START" tells the G command where to go and the "20" tells it to execute 20 instructions before it comes back to you. You will see that the Program Counter,which points to the next instruction to be executed, is :4308, HL is :0015, DE is :0006 and B is :04. If you add it all up you will see that this is the sixth time through the loop and we have in fact executed 20 instructions. Then type J /NL/. This shows how J without an address will just continue the program to its end.

We sneaked in a little subtlety without mentioning it to you beforehand. You have used the symbolic names FROM YOUR PROGRAM in DEBUG instructions, and they worked. That sort of thing makes debugging a program very much simpler. By the way, if you try single stepping through a breakpoint then you will single step into the breakpoint logic, which may not be exactly what you had in mind.

4.9 I for IMMEDIATE

We conclude this chapter by looking at the immediate instruction facility in ASZMIC, which to the best of our knowledge is unique. It is a way of giving you the sort of interactive capacity you have in BASIC by allowing you to specify assembler statements which are immediately executed in your program context, and function as an extension of the program itself without the need to recompile.

Clean up the text area if you have a 1K system. Type I LD HL,1/NL/ and look at the HL register in the "0" dump. It contains :0001. Type I EX DE,HL/NL/ and note that the DE and HL registers have been exchanged. You can put any instruction in an Immediate command, and even define labels with the EQU directive, but relative jumps and the other directives (ORG, DEFB, DEFW, DEFM) are meaningless and can crash the system when the I command tries to execute them.

Finally try LD HL,1+2+3+4+5-15/NL/ and look at HL in the register dump. It should be zero, thus demonstrating how you can do simple arithmetic with fields.

We now recommend that you look at the formal definition of the DEBUG commands in Appendix 3 to see what you have been doing all the time. There is one more Debug command, the N command, which is used in conjunction with a special board which holds both Basic & ASZMIC ROMs to switch between ROMs whilst preserving memory. Its use is described in the board documentation.

Chapter 5

TEXT OPERATIONS

This and the next two chapters discuss features of ASZMIC to help 'flesh out' the bare bones of definition in the Appendices and the exemplars of the previous chapters. They should not be taken as comprehensive descriptions of everything which is available.

5.0 INTRODUCTION

The second partition in ASZMIC memory, lying between DSPBGN & (TXTLIM), is used for text preparation and editing. On a 16K system this means you have around 12000 characters available; more if you move up the TXTLIM pointer yourself. ASZMIC treats this as an allowed space in which you can insert characters at will, and your T.V. screen becomes a window which is moved over this enormous area under control of the Editor Shift keys. There is no specific display file as with BASIC, but instead the whole text area is a display file, and the BASIC notions of program, variable and display space are no longer relevant. This is a software utilisation of one of the main strengths of the ZX81, viz. that it allows most of memory to be mapped onto a video display. We force a few conventions of our own onto this space to make life a bit more meaningful for you.

5.1 FILES

The first convention is to introduce a protocol which divides up the text in the text area into identifiable sections. We call these sections files. A file is identified by a symbol whose first character is a £

(sterling) sign left justified on the first line of the file. This symbol, plus its preceding £ sign, becomes the NAME of the file. The end of a file is signalled by a £ sign as the first character of a line. We call the £ signs FILEMARKS in consequence. The ASZMIC character string comparison routine CMPSTR is coded to recognise a comparison of strings beginning with £ only if the destination string is the first on a line. This all sounds a bit complicated but it works out very well in practice, because if you start every file with an unique filename, and terminate it with a filemark, the ASZMIC commands will unerringly pluck out the file you are interested in from the text area and process it for you. The convention is fully integrated into the Debug commands.

This means that you can have as many files as you like, subject only to space restrictions. We should point out that the Debug commands will often work if you have specified an invalid filename for a file (i.e. one not starting with a filemark) but there are no guarantees. Forgetting the terminating filemark can be a nuisance for cassette ops and printing, and a total disaster if you then try to assemble or merge the endless file.

5.2 MERGING & DELETING

We implemented these functions as Edit Shift operations, rather than Debug commands, to give extra flexibility and to tie up with the 'Show it rather than tell it' philosophy of a full-screen Editor. The cursor is used to identify the starting point for merge & delete operations; and a filemark is used to terminate the operation. When a merge key is pressed ASZMIC searches from a little past the shift macro line down to a little beyond the current end of data pointer to find the merge start character (> for Shift-G < for Shift-F * for Shift-D) and then copies the source text down (or up) to the cursor position until it finds a filemark in the source text. Long merges can blank the screen for a moment.

At one extreme this is a conventional file merging operation, at the other it becomes a text macro. You can, for example, copy in a file of subroutines from tape and merge them into a program. You can also, if you are writing a program with a lot of data in it, code something like > DEFB £ high up in the text area and then whenever you hit Shift-G the string DEFB will be incorporated into your program. This can save quite a lot of typing.

When you want to delete the source file/string which you have merged you just position the cursor onto the merge character and press Shift-2 and, Presto, it has been deleted. A filemark will always stop a delete or merge operation. Failure to terminate a merge source string with a filemark will mean that the poor computer will go on trying to merge forever. You cannot escape back to ASZMIC. Deletes just refuse to work if they cannot find a terminating filemark.

5.3 EDITING

You have covered the EDIT functions pretty comprehensively in Chapter 2, so here we will just give a few hints & tips. Remember the use of the E string command as a Shift Macro when you want to search out all occurrences of a name in a file. Hitting /NL/ in the middle of a previously written line will convert it to 2 lines, and Typing Rubout can be used to concatenate lines by deleting a /NL/. Please try not to edit in the top 20 lines of the text area unless you are working with a shift macro. These lines are really padding to preserve a clean display and deleting them could mean that you generate an illegal display. In particular using Type Rubout to delete the /NL/ at the very beginning of display can leave the vital (but invisible) start of data marker exposed with later disastrous consequences. Editing out the blank which ASZMIC ensures is before every /NL/ can also cause you to lose the cursor in some circumstances and can cause cosmetic errors in assembly and merging. Writing more

than a few lines with more than 36 characters in them on any one display page can also cause ASZMIC to position the cursor off the visible page, although this does no harm.

When you want to insert lines in a text you should position the cursor on the line above the desired insertion point, hit Shift-W to get to the end of the line and then hit /NL/ to create a new blank line. Shift-7 followed by Shift-6 will take you back to the beginning of a line. Remember that Shift-Q rubs out under & forwards whilst Shift-0 rubs out backwards.

5.4 PRINTING

The way the printer operates is determined by 2 bits in ASSFLG. This flag is set automatically by the Assembler (it is in fact the options byte) and zeroed at assembly termination (unless you used the Break key to escape from an assembly) so if you want to control the printer you have to set these bits yourself. Bit 1 set to 1 will route almost everything ASZMIC, as opposed to you, writes to printer and Bit 5 will set the printer in fine pitch mode with 64 chars per line.

You can fool the printer into writing double height characters if you want. If you write a line of up to 32 characters and then pad it out with blanks (no /NL/)

to a total of 48 characters, and then type in the 32 characters again in EXACTLY the same order you will get, not a long line with 2 messages, but a single message with double height characters when the line is printed. This can be effective for titling, and if you do it a lot you will learn to use the merge facility as a convenient way of generating the double texts. If you get the number of blanks wrong the effect is, well, interesting but hardly legible. This technique will also work for fine pitch printing but double up to 64 chars of message and 32 padding blanks.

Remember the use of the Break key to get you out of print situations when you have forgotten the terminating filemark. Finally, printing takes place indirectly, that is ASZMIC jumps to the print routine via an address in RAM. If you alter the contents of PRTJMP to the address of your own print routine then ASZMIC will use yours instead of the Sinclair interface built in. Print is entered with the stack containing the start of the line to be printed, and your routine must clear this and exit with a RET with HL pointing at the first character after the /NL/ which terminated the line.

5.5 CASSETTE OPERATIONS

Your cassette recorder should function with ASZMIC at the same volume and tone settings as with Basic. ASZMIC should in fact work rather more reliably since it maintains a tone to the recorder until just before recording starts; so that the automatic gain control on most low cost cassette recorders will not be so obtrusive as it is for Basic. If you want to set up your recorder for a new tape brand then try recording a sequence of blanks (K&S "NULL" :7000 :7FFF); just after switch on when the program area is empty works very well. Try loading it back in with your recorder volume control set to minimum and then watch the effect as you slowly increase the volume. The screen will change from "white & swimmy" to a series of well defined striated bands to a predominantly black screen when the volume is too high. The volume control is best set in the middle of the "band" region.

5.5.1 HEADERS & FILES

Since there are 2 types of "file" we can manipulate with cassette; genuine text files and defined regions

of memory, we have to give each tape file a seperate identification since memory regions do not have file names. When you issue a save command to ASZMIC it writes out the command itself as a header and then

pauses before writing out the file proper. On playback the syntax of the save command represents a virtually unique character sequence which ASZMIC recognises, and causes it to be displayed for a few seconds to identify what is on tape. If the identification in the header and the Load command correspond then ASZMIC checks the rest of the header to find out if it is loading text or memory and reacts accordingly. You always come back from Load in Edit mode. There are a couple of extra features associated with loading a memory region. The first is that the region into which the saved program/data is loaded is offset by the contents of the OFFSET variable, so you can do a relocated load from tape. The second is that if you analysed the contents of LABSTK & LABEND and wrote out a Symbol Table to tape, then if you followed the 'end of region' definition with a space and then the letter L ASZMIC will recognise the load as a Symbol Table and set LABEND accordingly. LABSTK is unchanged so you must have the same size memory and not have relocated the Symbol Table before the save. A small feature but Symbol Tables are sometimes worth saving for library or debug purposes.

The physical seperation of header and file on tape gives you the chance to do some very complex things by juggling and over-recording. This is for the benefit of those hardy souls who are not content till they have abused a system to a maximum, or the more practical who wish to recover a poor recording. The details you have to work out for yourself. Note that if RDCASS is called before the tape has run onto the half-second silence before a file or header it can pick up rubbish (depending on volume setting and screen contents at record time). The silence will, however, synchronise it onto a byte boundary so that the file may be displaced but will not be corrupted.

As ASZMIC will treat everything following the K£S thru to the End of Data marker as the header, the cassette save command should be the final command in the text area when /NL/ is pressed.

5.5.2 CREATING BASIC PROGRAMS

There IS one creative abuse of the cassette recorder which we must mention. For those of you who do not have a board which holds both ASZMIC and BASIC ROMs the cassette recorder is the most accessible form of communication between the two. In the Application notes you will find a listing for a general purpose program which simulates the context of a Basic program with a single REM in it. By placing your own code at the indicated point you can cause it to be incorporated into the REM. A cassette save of the program will then be loadable by Basic, for the example given the identifier "123456789" is used, and you can then add lines of genuine Basic and reference the code in the REM by a USR (16514). Your code should not affect the HL' register. You will need to use the OFFSET facility when assembling the program. On a 16K system ORG :4000 and OFFSET = :3000 works very well, but you must have a 9 byte header since the first 9 bytes of the variables in Basic are not recorded.

If you want to go back in the other direction (it takes all sorts.....) then you must start by initiating a save of a memory region large enough to hold the Basic program. You can abort it with Break as soon as the memory starts to be written out; all you want is the header. Then back the tape so that you are just before the data which began to be recorded. If you then perform your save of a basic program it can be loaded back by ASZMIC. You will have to abort the load manually when the Basic program is complete. Alternatively you can write a very small program which uses RDCASS to get bytes from the tape and store them. This latter approach is probably quicker and more controllable.

In the beginning ASZMIC had a super fast cassette interface with parity checking and lots of goodies, plus an offset ASCII character set and an RS232 style printer interface. We changed it all to give you Sinclair compatibility on characters, printer & cassette so we hope that someone does make use of this, otherwise it was all wasted.

Chapter 6

THE ASSEMBLER

Before discussing the Assembler section of ASZMIC we should look at what an assembler does for us. Unlike BASIC, assembler statements are closely related to the actual operations of the Z80 cpu. The Z80 has a well defined set of logical, arithmetic and data movement operations which it can perform.....operations which are relatively easy to understand even if it takes a little time and practice to use them to perform 'useful' work.Unfortunately the instructions which direct these operations when they are read in from memory are more meaningful to the digital decoding logic in the Z80 than they are to flesh and blood.

6.1 MNEMONICS

The first thing an assembler can do for us is to allow us to write down the instructions for the Z80 in a form which is more expressive for us, and then translate them to the Z80 codes; for example CCF as an abbreviation for Complement Carry Flag is easier to remember than hexadecimal 3F. This can be extended further by allowing us express the Z80 instructions in terms of operations on operands meaningful in terms of the Z80 internal architecture(which is relatively comprehensible) and having the assembler decode the fields down into valid instructions. The assembler thus enables us to write Z80 instructions in terms of our conceptual picture of the Z80.

6.2 SYMBOLS

The other thing that the assembler can do for us is to allow us to use symbols instead of the binary codes that the Z80 enjoys to represent data and addresses. Data items can be written in the decimal or character formats which we are used too. The Z80 works with defined addresses in memory. Very often we do not know as we code a program where its constituent routines and entry points lie. We can give them symbolic names, called LABELS, and let the assembler bear the burden of assigning precise addresses to them.

Some of the symbols the assembler uses are self-defining, things like numeric constants, and the rest tend to be labels which can either be explicitly defined with a directive to the assembler, or can appear as the first field of a statement in which case they are assigned the address at which the code generated by that statement will lie in memory. This address is called the location of the code, the assembler keeps track of it by an internal variable called the location counter, and there is a special symbol, the $ sign, which the assembler interprets every time it encounters it as the value of the location counter at the point of encounter. Thus the statements:-

```
LABLAB JP LABLAB
       JP $
```

have the same effect when assembled.

6.3 HOW DOES IT DO IT?

When the assembler reads through your program for the first time, and it encounters a label used as an argument in a statement, then if the label has been previously defined to the assembler the address or value associated with it can be incorporated in the

instruction, but if the label is something like a jump address further down in the program then the poor old assembler is stuck for a value. The solution to this problem is that the assembler always assembles your program twice (makes what the refined refer to as a second pass over over the program). The first pass pretends to generate code, but really only generates a table (The Symbol Table) which at its conclusion contains a defined value for every label. The second pass can then reference the Symbol Table to substitute valid values for symbols and generate executable (object) code, fancy listings and error messages. Whilst we are talking about the Symbol Table let us slip in a few definitions. A label which is encountered as an argument before it is defined is called a forward reference, a label which is defined more than once in a program is called a duplicate definition, and a label which is referenced but never defined within the program is called an unsatisfied reference unless you have found some way to define it into the Symbol Table (e.g. Immediate statements or use of force pass 2 option) externally in which case it is called, surprisingly enough, an external reference.

6.4 OPTIONS

When you use the A command to tell the Command Interpreter to invoke the Assembler you specify a file which is to be assembled, and also an OPTIONS field, which tells the Assembler how it is to work. Some of the options are pretty obvious. You can elect to generate an assembly listing...very important when debugging your program. You can elect to have the listing and any error messages directed to printer, with 64 characters per line if you wish. You can suppress the generation of object code.... useful in the first stages of checking a program and in other circumstances too.

Then you can select some less obvious options. They are concerned with the linking together of seperate programs into a functioning whole. Normally when you initiate an assembly you zero out the Symbol Table, but

you can elect to preserve it if you wish. Why? To allow your program to use external references defined in it. You can also elect that the assembler will proceed immediately to pass 2, which from our discussion earlier should be disastrous. Why? Because when coupled with the Symbol Table preservation option this allows us to assemble together many seperate programs with cross-references into a functioning whole.

If you look in Appendix 3 at the A command it summarises the options available and the numbers which represent them. The options field is just the sum of the numbers for the options you want . So if a listing on printer, normal pitch, but with no object code is your desire the options field becomes 64+2+1 or, if your mental arithmetic is good and your patience weak, you can write 67 directly.

6.5 OFFSETS

There is a variable called OFFSET, whose location you will find in Appendix 4, which has a special significance for both assembly and loading from cassette. The value of this variable is normally zero, and it is used as a relocation offset by the Assembler.

The way it works is this:- Normally when you code a program you expect it to be assembled at the location you have specified with an ORG staement so that it will execute at that location. We say that in that case the EXECUTION LOCATION COUNTER and the LOAD LOCATION COUNTER are synchronous. But what happens if you are writing a program which will not be executed after assembly, but will instead be written out to an EPROM programmer to generate an EPROM which is supposed to execute starting at location zero. You have to write an ORG 0 statement at the start of the program so that the program is internally consistent, but if you just left it at that the assembler would try to generate the object code on top of the ASZMIC ROM. This would do no harm but your object code would be lost. You need some way to tell ASZMIC that it must place the object code somewhere else.

The OFFSET variable is the technique you use to achieve this. When ASZMIC is writing out object code it adds the contents of the OFFSET variable to the execution address to generate a load address at which each object byte is to be situated. If, before an assembly, you set OFFSET to :7000 then code which is supposed to execute starting at location 0 will in fact be loaded into memory starting at address :7000.

Remember to reset OFFSET to 0 when you are finished. ASZMIC will not do it automatically for you and you will have a fine old time wondering where your next assembly has vanished to.

6.6 ASSEMBLING

You invoke the assembler by typing in A followed by the name of the file you wish to assemble and, unless you want to take the default of object code, no listing, error messages on screen, and new symbol table, an options field to control the assembly.

You must obviously have first written the program you wish to assemble, using the Editor or loading in a previously written file from cassette. Make sure that the file containing the program obeys ASZMIC conventions i.e. that it starts with a filename whose first character is a filemark, and that it is terminated by a filemark as the first character of the line after the final program line. Within the file the program lines, assembly statements, are written according to the conventions used by ZILOG and MOSTEK, the manufacturers of Z80 chips (the only exceptions are the EQU directive and the format :1000 instead of 1000H for hexadecimal numbers). These conventions are, in brief, that the first character of the statement should be a blank unless you wish to specify a label at that point, and that any label be terninated by at least one blank. There then follows an op-code or directive which must be terminated by at least one blank. If the op-code or directive needs it there then comes one or two arguments. If there are two arguments

they must be seperated by a comma without any preceding or following blanks. The final field (argument, op-code or directive) that you write may be terminated by a /NL/, a blank or a ; (semicolon). If a semicolon is found then any characters after it on the line are trated as comments.

If the first character of a statement is a semicolon then the whole statement is treated as a comment. If a label starting in column 1 of the statement is terminated by an = (equals) sign then the assembler assumes you are using an EQU directive to define the label, and searches for an argument after the = sign (you can have intervening blanks if you want) which is evaluated and assigned to the label.

Please note that the Assembler exits by using an RST 0 instruction instead of a RET, so that you can use the O command to see the end of the region for object code in the IX and IY registers.

6.6.1 DIRECTIVES

The ASZMIC assembler has two genuine directives, ORG and =, which do not generate executable code, and three pseudo-directives, DEFB, DEFW, and DEFM which do not generate EXECUTABLE code but produce object code which is a representation of the data values expressed in the argument after the directive itself. We touched on = in the last paragraph; so we will look at ORG now. ORG causes the following argument to be evaluated and transferred to the execution location counter (the IX register). You can use it at program start to define where the program expects to be executing (and where object code will be located if you have left OFFSET at 0) and in the middle of a program to generate space. A cunning use of ORG in mid-program is a statement of the type:-

```
ORG $+20
```

which will reserve 20 bytes of empty space in mid-program for use as a buffer or whatever. ZILOG use a DEFS directive for this purpose which we did not implement in ASZMIC because it was redundant. You cannot use forward references in ORG or = directives (there are ways round this but please phone your friendly local university computer department to find out about them rather than us).

DEFB will quite simply generate one byte of object code with the value, modulo 256, of the argument in it. DEFW will generate 2 bytes but with the least significant byte first in the way the Z80 expects so that:-

```
DEFW :1234
```

will produce what looks like :3412 in memory.

DEFM searches for a " (quote) sign and then transforms every character after it to a byte in memory according to the Sinclair values assigned to each character, until it finds another " which tells it to stop. This means that you cannot have " signs in a DEFM argument. Do not try coding two together to get round this, it just does not work.

6.6.2 OP-CODES and ARGUMENTS

Op-codes are the mnemonic representation of Z80 operations. You can read about them in the Z80 assembly manual of your choice, and remind yourself about what they are and what they do by browsing thru Appendix 5. Arguments are expressed in the standard ASZMIC form which you have been using already. They are defined for you in the "FIELDS" paragraph of Appendix 1.

The only argument you will encounter which you do not use generally throughout ASZMIC is the $ (dollar) symbol, which represents the value of the execution location counter at the start of the statement in which the $ occurs. This has been a short paragraph for a large topic because this is the information which you will hopefully have gleaned from your book on Z80 Assembly Programming.

6.6.3 COMMENTS

Comments are always said to be a VERY GOOD THING in a program of any sort, and are particularly desirable in assembler programs because the code tends to obscure function by detail (the Devil's Data Dictionary claims that by the time the average professional programmer has reached the twelfth line of a program he has completely forgotten what the first six did). We follow ZILOG conventions by allowing you to terminate a statement with a semicolon and then fill up the rest of the line with commentary. Unfortunately the screen can only have 36 characters on a line (32 for the printer) so that by the time you have allowed 16 positions for location and code in an assembly listing you are liable to lose long arguments from the program statement, never mind comments tagged onto the end. They are still useful in the source code (sorry, source code is the name for the programs you input to the Assembler). Option Bit 5 will remove the truncation effect, and also set the printer in fine pitch mode to give you up to 64 characters per line. The ZX81 does not have all that much memory to play with, and has a diabolically slow cassette interface, so for large programs the hardware offers little encouragement for in-line documentation. One recommends even so that you try to describe what each section of the program is doing as you code it. You will forget sooner than you believe possible what was going on otherwise.

6.7 ERRORS

In a tight little 4K ROM there was a limit to the amount of error analysis which could be done by the Assembler. The authors also had some problems deciding what sort of errors should be checked for, since they never make mistakes (ouch). After clandestine observation of their less gifted colleagues they came to the conclusion that euthanasia was the best error checker. Joking apart, once you have gained some experience with assembly programming (remember as you struggle and curse at the beginning that you will be a

beginner for a few weeks and an expert for the rest of your life) the mistakes you make fall into 3 categories. You muddle up label names by misspelling, coding the same subroutine twice or forgetting to include it at all; you insert so much code between a relative jump and its destination that the jump goes out of range; and you make mistakes with Op-Codes,normally by coding them in column 1. These are the three classes of error that ASZMIC will identify for you.

Errors are signalled to you by the setting of a non-blank column 1. If you are not generating an assembly listing the errored lines will be displayed for you in any case.

The nature of the character in column 1 could tell you a lot about the error, but the rules are so complicated that it is easier to set about it logically instead. First look to see if any object code appears on the error line. If it does not then you have an Op-Code error. Possible causes are starting the Op-Code in column 1, terminating it with a comma instead of a blank, or just misspelling it (because of its unusual design the Assembler can only detect seven-eighths of the possible misspellings). Then look to see if the you have a relative jump in the statement. If so the error is almost certainly a jump out of range. Finally you are left with duplicate definitions or unsatisfied references for labels, so look at the label(s) in the arguments and find out if it has been defined, or defined more than once.

In addition, if you code a number with an invalid character for the base of the number, ASZMIC will signal an error.

6.8 SEQUENCE

We recommend the following sequence when assembling a program:-

1) Edit the program.

2) Save it on tape immediately.

3) Assemble it with option 64.

4) If any errors were indicated identify them and go to 1).

5) Assemble it with option 65 (67 if you have a printer).

6) Check the listing to see if this is what you really wanted. If not go to 1).

7) Assemble with no options field.

8) Debug as described in chapter 7. If errors detected go to 1).

9) Save the object code, and possibly Symbol Table on tape (Remember the L field).

6.9 LISTINGS

The hallmark of the assembly language programmer is a battered and well annotated assembly listing. It is the fundamental aid to getting your programs in working order. If you do not have a printer life will be that much harder for you. ASZMIC tries to help you by permitting the use of program labels in Debug statements, allowing you to generate listings on screen, into which you can edit filenames and filemark delimiters and then save and restore them using cassette, and the ASZMIC Editor is good enough to let you use the screen listing as a notepad so that development is still possible, but it can never replace a true listing (hard copy).

The assembly listing consists of four distinct fields. The first is the single byte error flag which is placed

in column 1. The second is the four hexadecimal digit location at which the code generated from each statement will lie (i.e. the value of the Execution Location Counter at the start of each statement); the third is a hexadecimal representation of the one to four bytes of code which each statement can generate (note that DEFM can generate up to 5 bytes before truncation); and the fourth is as much of the original statement as the Assembler can fit into a 32 character line.

The listing, or error messages if a listing is not selected by the OPTIONS field, will be routed to printer instead of to screen if bit 1 of the OPTIONS byte is set.

6.10 LIBRARIES

If you are just beginning with assembly programming you can ignore this section and the next one. They describe slightly exotic functions which you probably will not want to use for a while. A library is a piece of object code which contains frequently used routines accessable to many different and independent programs. If you write a program which needs such a routine you could always load in the source code from cassette (or microdrive when they become available), merge it into your program and then assemble the whole, but after a while you tend to find this a cumbersome and unecessary procedure. How much more pleasant to load in your collection of useful object code routines once (things like input and output routines, drivers for special peripherals, multiplication and division routines and the like) and have them available for all comers when required. Such a collection of routines is called a library (no prizes for guessing why) and is characterised by the fact that the routines within it are internally self sufficient, and do not need to reference any external program. A library contains definitions, but no external references.

How would you use such a set of routines with ASZMIC? To begin with you would assemble the library modules (module = constituent program) and save both the object code (the library itself), AND THE SYMBOL TABLE WHICH ITS ASSEMBLY HAD GENERATED. When you needed to assemble a program which referenced (used) some routine in the library you would first load in the Symbol Table as described earlier, and then perform an assembly with bit 2 ("4") of the OPTIONS field set. This would mean that your program would start off with a Symbol Table which defined the addresses of all the names in the library programs, and references made to them would not be flagged as errors. Of course when you came to execute the program you would have to be sure that you had remembered to load the library into memory otherwise the result would be disaster.

6.11 CROSS REFERENCING

The next stage in your development as a jejeune assembler programmer is to write programmes so large that you do not have enough memory to assemble them as a single entity, or to co-operate with friends or colleagues to write seperate sections of the same programme. In this case you are faced with a situation rather different from that we met with libraries, because each seperately coded and assembled section contains not only definitions which other sections can use, but references to the other sections as well. how do we handle a situation like this? it is probably simplest to postulate an example with three seperately coded but mutually (sic) dependent programs; PROGA, PROGB and PROGC. PROGA is coded and as its final statement line has ENDA=$. PROGB starts with an ORG ENDA and terminates with an ENDB=$. PROGC starts with and ORG ENDB statement.

If you then assemble PROGA with an OPTIONS byte which includes bit 6 (no object code) and then assemble both

PROGB and PROGC with an OPTIONS byte which includes a setting of both bits 6 and 2 (see Appendix 3) then you will have built up a symbol table which includes label definitions from all three programs. You then assemble all three programs again, but this time using an OPTIONS field which includes both bits 7 and 2, and you will have at the conclusion a single piece of object code which is internally self consistent, always assuming that you have not made any errors.

What you did was to suppress object code the first time you assembled the three programs in sequence so that at the end you were left with a Symbol Table which contained all the symbol definitions for all three programs, plus a bunch of error messages which, providing they refer only to unsatisfied cross references, are irrelevant. Because PROGB and PROGC were assembled selecting the symbol table preservation option the definitions ENDA and ENDB were available in the Symbol Table at the time the ORG statements were assembled so that the programs automatically lie together in memory. The second time we assembled each program we used a preserve Symbol Table option so that all the definitions were available but, since the purpose of the first pass of any assembly is just to build up the Symbol Table and we had done that already, we also used a bit 7 (Force Pass 2) option to prevent the creation of a lot of duplicate definitions.

6.12 VALETE

That pretty much covers all we wanted to say about the Assembler. The next chapter tells you how to debug your assembled object code. It is a very good idea to save your object code on cassette before you try to execute it for the first time, since mistakes can wipe out the system, and if you want to use program labels in your DEBUG commands you might also save the Symbol Table. Remember that ASZMIC comes back in EDIT mode after an assembly and, please, always start each program with an ORG directive.

Chapter 7

PROGRAM EXECUTION & TEST

When you have written and assembled a program you naturally want to execute it. ASZMIC gives you two major commands for program execution and one for single stepping through a program (we are discounting possibilities such as referencing the program in an immediate statement using CALL or JP). We shall start by looking at the DEBUG statements available for program testing.

7.1 THE J COMMAND

This is your primary command for program test. It can be specified with an address field, in which case the first instruction executed is at the address specified, or by itself, in which case the address stored in PC1 when last a Break condition occurred will be used as a transfer address. The J command is characterised by the fact that all the registers are loaded up from the register image area (REGIM) before execution, and that the I register is set to 1 to facilitate non-maskable interrupt (NMI) handling and Break conditions. If your program does not perform register initialisation then you can initialise the register image area using I or D commands.

This is all straightforward, but if you J to your program and it contains a fault, then your chances of getting back to ASZMIC to find out what happened are slim unless you have an NMI interrupt button fitted to break you out of unforeseen loops. Your first line of defence against the self-immolating program is the breakpoint.

7.2 BREAKPOINT

A breakpoint is an RST 0 code. It can be inserted via the B command, in which case ASZMIC will handle the restoration of the original byte and manipulate the saved Program pointer value so that you can continue with the instruction when you wish, or you can insert it yourself by the D command or by placing it in the original code in which case you must perform any skipping or replacement needed yourself. The conventional way to use a breakpoint is to divide your program into logical sections, and then use the B command to place a breakpoint at the end of the first section. When you execute the program and come to the breakpoint you can check using the DEBUG D and O commands that everything has gone as you wish, and then move the breakpoint to the end of the next section and use the J command to execute the second section. The process is repeated until you identify a section which fails. Very often the mere identification of the part of the program which is failing will concentrate your mind most wondrously and enable you to isolate and correct the mistake. Remember that the RST 0 code is removed, and the original byte replaced, when the breakpoint is encountered; so if you want to break again at that point use the 3 commands G B J to get you thru the breakpoint, restore it, and restart execution. If you have a breakpoint active and load in another program on top of the one in which you had the breakpoint then subsequent moving of the Breakpoint will restore the byte from the old program into the new. Remember that you cannot single step through a breakpoint without single stepping into the breakpoint handler.

7.3 THE G COMMAND

When you have isolated the section of a program which is failing you can then examine its functioning in detail using the G command. This is the ASZMIC single step feature. It has the effect of the J command but with an automatic Break condition occurring after each

instruction. Break conditions always cause context saving in the REGIM area. If you want to step thru a number of instructions before coming back to ASZMIC you can specify a step count as a second field in the G command. In that case you have to specify the first field, the address to GO to, explicitly even if it is the stored Program Counter value.

The use of a step count with the G command can be extremely useful. Remember that although you do not return to ASZMIC until the step count is exhausted there is a break with context save and restoration after every instruction so that execution is very much slower than it would be normally. Since the single step feature works by using the ZX81 NMI coupled timer to simulate a breakpoint, the breakpoint and single step breaks are handled by the same code, and this gives you the facility to simulate a ROM breakpoint. If you have a home cooked Eprom with routines starting at :2000, and you want a breakpoint at :2122, then you use a B :2122-1 command and execute the Eprom routine with a G :2000 32767 command. ASZMIC will then single step through the Eprom code until it has executed the instruction before that at :2122, and then it will think that it has encountered a breakpoint and stop the single stepping.

The G command uses the ZX81 NMI interrupt timer in a special way, and is thus not fully compatible with routines which drive the display since they require an I register with 14 in it and an enabled maskable interrupt. A little ingenuity with breakpoints and single stepping over non critical sections will probably see you through problems like that.

7.4 THE O COMMAND

This command, which displays the saved registers from the REGIM context save area, really comes into its own when you are program testing. The registers are displayed as 2 lines in the order:-

```
PC   HL   HL'  BC'  DE'  AF'
AF   BC   DE   IX   IY   SP
```

The ' (prime) registers are the Z80 alternate registers. PC is the Program Counter which tells you which instruction you are next due to execute. SP is the Stack Pointer and the rest are the standard register pairs. We are sorry that we could not print up headers above the dumped registers, but it takes as much ROM space as a printer interface to do that and it burns up the text area twice as quickly, so we felt that it was better to ask you to refer to the documentation.

The O command is usefully placed as a Shift Macro when you are debugging a program (i.e. hit Shift T and type O on the top line). Every Break condition, be it breakpoint or single step, then gives you an automatic register dump. If you have important variables in your program which you also want to dump out at each breakpoint you can concatenate commands on the Shift Macro line, using ;/ (semicolon slash) as a delimiter between each command and starting the command immediately after the /. This facility is really meant for dump commands but most Debug commands WILL work there; but do not use an I command since it will then chase its own tail for eternity (purists call this a reiterative loop).

7.5 THE I COMMAND

We are told that some CP/M debugs have the equivalent of the I command in them, which proves that it is such a good idea that many people will discover it independently, not that we ripped off the notion from CP/M. The I command enables you to write a line of assembler which is then immediately assembled and executed for you. Before execution the registers are loaded up using an internal form of the J command, and after the assembled instruction is executed a breakpoint is forced to save the context again, so that the I commands execute in the context of your program. Providing you still have the original assembly Symbol Table intact you can specify labels from your program in I commands, just as in other Debug commands. The only snag is that an I command will change the stored

PC value, so remember to J or G to a specific address if you have used I before them or you will start executing the ASZMIC stack. Use of I is a good way to rectify some unimportant omissions in a program and continue with mainstream testing.

The I command is normally used for priming registers, but any executable command can be given. If you do not leave a space between the I and the start of the assembler statement then you will be presumed to have started the statement with a label which you are declaring via the = directive. The only snag is that the label declaration logic in ASZMIC uses /NL/ as a delimiter so that the I is taken to be part of the label. You can declare any label you like so long as it starts with I. Henry Ford would have approved.

7.6 THE D, F & C COMMANDS

These commands are fairly straightforward in their operation, and we really only mention them to preserve an illusion of completeness. Dump, and Modify, are the kernel commands of any debug system. Fill is valuable for initialising buffer or workspace areas to an initial value. C, whilst having its primary use in the relocation of code assembled or loaded using OFFSET, can often be useful to reinitialise complex data arrays from a 'spare' copy in memory when you want to return to an earlier stage in a program for re-testing without the trouble of reloading the program. F and C are excellent ways of wiping out the system, so check for typing errors before hitting /NL/.

7.7 VALETE

Debugging a program, even more then writing one, is a black art in which luck, logic and intuition are inextricably blended. Logic and intuition you must supply for yourself, but we wish you all the luck in the world.

Chapter 8

GRAPHICS

ASZMIC offers high resolution graphics facilities. The display driver in the Basic ROM uses a fixed number of rasters (lines) for every character which it displays. This severely limits the graphics possibilities available with Basic unless you install special hardware. ASZMIC has a programmable driver which allows you to set the parameters which control display yourself, and has also substituted for the Sinclair graphics characters a special set which are independent of the hardware offsetting based on raster count. The effect is that from a machine language program you can plot on a 255 x 144 matrix, maintain a continuous display (ZX81 only), and still have time to do computation for movement effects. We suspect that although ASZMIC was really designed as a development station for assembly language programs, and to a lesser extent as a teaching tool, many people may use it just because it is the cheapest way to give yourself a convincing 'Dungeons & Dragons' scenario on the ZX computers.

8.1 THE ASYNCHRONOUS DRIVER

In the Application notes this program is identified as the KERNEL routine. Its function is to create a blank display file, display it 50 times a second, handle syncs & keyboard read, and hand over control to the user program when it is not busy. From a user point of view all his program has to do is manipulate the display file contents; the KERNEL takes care of all the rest. If you are using a ZX80 you cannot use the KERNEL but must instead build up your display, send it out using OFRM1, and handle timing and frame sync yourself. If you have a ZX81 and use KERNEL then your program can act as if the mechanism of display was invisible to it.

Do not take KERNEL as sacrosanct; we wrote it quickly (as did we all the graphics examples) just to show thw possibilities. You can probably do better yourself You might start by tuning NNN and IDLE (rasters at bottom of frame and sync pulse length tuning respectively) so that they suit your television.

8.2 PLOT

The PLOT routine in the Application notes is fed by a subroutine CALL with the B register containing the X coordinate and the C register containing the Y coordinate relative to screen bottom left. It computes the byte in the display that contains the point, decodes what is in the byte and inserts the new point, and then encodes the byte back again. UNPLOT does the same but deletes the relevant point instead. It does not check for point out of range; that you can put in yourself. It uses a shift and subtract algorithm to compute line address, and a little bit of fiddling with the high order bit in the desired byte to convert to and from the four points (pixels) which each byte can contain.

8.3 LINE

There is another subroutine given which will use PLOT to generate a line beteen two specified points. If the line is between XY & X'Y' then D=Y, E=X, B=Y' & C=X' when the subroutine is called. It uses a successive incrementing algorithm to calculate the points needed with the increment maximised to avoid redundent plotting. It works in theory, and in practice too, but other algorithms can give lines which are subjectively more pleasing; particularly when the line subtends only a small angle to one of the axes. A 'dotted line' algorithm in particular can often look much neater since one tends to join up the dots mentally with a much higher resolution than the screen can achieve; but we hesitated to provide any 'psychological' subroutines. They seemed to be straying too far from our path of self-imposed utilitarianism. ULINE deletes a line between the two specified points.

8.4 UPROG's

This is the unlovely name we give to the application programs which create requests for points and lines which PLOT & LINE generate, and KERNEL displays. We offer two examples. STRUCTURES will generate diamond patterns using PLOT which can look like futuristic space cages. MOIRE will just write tightly packed lines to give a watered silk (who wears watered silk nowadays?) effect. We suggest that you get down and write your own UPROG to get the feel of the graphics. One useful technique that you can use is to achieve translation of images on the screen by use of LDIR instructions. A move of less than 37 bytes will translate on the X axis, and a multiple of 36 will give Y translation. This is a much easier way of moving composite images than painstakingly deleting and re-drawing them. You CAN also mix text and graphics by starting the text on a raster ('Y') which is a multiple of 8 from screen top and writing it 8 times on sucessive lines (Y's). Remember that if you put a byte with bit 6 set to 1 in the display then the Sinclair hardware will try to execute it as an instruction, with results varying from minimal to a full blooded system crash.

8.6 OTHER RESOLUTIONS

You can alter the vertical resolution of each pixel by altering PIXSIZE in the KERNEL, and the total number of vertical pixels by changing RASTERS, and the number of blank lines at screen top by changing TOPS. If the product of PIXSIZE and RASTERS is summed with TOPS and NNN, and the result is around 300, then you will still probably have a synchronised display. Cheap portable T.V's seem most tolerant of liberties taken with display timing, and costly colour sets the least. We have to be a bit general about this since there are over ten billion display type alternatives. Choose one by muttering to yourself

"TOPS changes blank space at screen top"
"NNN controls my compute time per frame"
"RASTERS controls the number of active lines in the display"
"PIXSIZE sets the vertical size of each pixel"

and looking at your application. We do not frankly see anyone wanting to use a PIXSIZE larger than 2. Remember to change CLEAR and DISPEND if you change the display file size or position.

8.7 SPECIAL OPERATIONS

When ASZMIC initialises at startup, it tests location :1000 at the end of initialisation to see if there is a JP instruction there, and if so does a CALL :1000. This means that you can add your own ROM onto a system with ASZMIC and cause it to integrate itself into ASZMIC. You have seen already how DADDR can be used to intercept command handling, PRTJMP to link in your own printer routine and INTJMP to take over break conditions after context save (that is what KERNEL does). There is one final reflection that we have not previously mentioned. KEYJMP normally contains the address KEYRET. If you put the address of your own handler there it can take over key interpretation (BC contains the undecoded result from KEYBRD) and then either jump back to KEYRET or RET to LIX if you have done all the work yourself.

The address SAVMEM can be used as a jump point if you ever want to reset ASZMIC whilst still retaining data in high memory, or to initialise it for a >16K memory. The HL register is loaded up with the address which is to be taken as the top of memory and then a JP made to SAVMEM. This roughly corresponds to a Basic "NEW" with RAMTOP altered.

When you are debugging programs it can be quite useful to have an external non-maskable interrupt button. The J command supports this by loading up the I register with 1 so that a single NMI pulse gives the effect of a breakpoint. You should, of course, use a Schmitt

debounced monostable to provide this but we got pretty good mileage out of a 300 pF capacitor wielded on ZX80, where in the absence of a G command you really need NMI. Failure to debounce correctly may cause the screen to assume an "heiroglyphic" appearence but the first key pressed cures this. On ZX80 bad debouncing can also fool ASZMIC into thinking that it is living in a ZX81; easily cured by providing a further 270 NMI pulses so perhaps you had better build a proper circuit instead.

8.8 THE END

We would like to apologise for a slight flippancy which seems to have manifested itself in places in this documentation. We have felt increasingly frustrated by our inability to do any more than sketch out the bare bones of ASZMIC usage unless we issue a 3 volume set in a year or two; and this manifests itself as a seeming lack of seriousness. There is just so much which you can do with ASZMIC. We hope that you find it stimulating and useful. In the future there will probably be quite a lot of supporting hardware and software developed for it, and we are scheduled to work on a microdrive version, so with luck it will prove a long term extension to the capabilities of your ZX80/81.

C FRAZER JOHNSON
Nykoeping Sept. 1982

Appendix 1

GENERAL INFORMATION

INITIALISATION

After turning on a ZX80/81 equipped with ZX.ASZMIC, memory will have been divided into 3 partitions. The first consists of system variables, buffers, stack and register image area (REGIM). The second, which lies between DSPBGN and (TXTLIM), is the text area. The third is the program and data area which lies between (TXTLIM) and the top of memory. This area is sized to one quarter of the available memory on the system. The pointers LABEND and LABSTK both point to the top of memory. These pointers define the Symbol Table, and assembler operations will cause (LABEND) to be the address of the current bottom of the Symbol Table. (LABSTK) is always the top of available memory. The IY register is loaded with :4000, the I register with 14, and the interrupt mode set to 1.

If any key is held depressed for more than half a second it will repeat at a rate of around 8 per second until released.

The screen contains 34 lines of up to 36 characters each. The ZX81 hardware automatically generates a new line for longer lines but since the EDIT display logic works by counting /NL/ characters this is a facility which should be used sparingly.

EDIT & DEBUG Modes

ASZMIC has 2 modes: EDIT mode, identified by a fast blinking cursor, and DEBUG mode, identified by a slower blink rate. The difference between them is that in DEBUG mode the typing of a /NL/ (newline) causes the line just terminated to be passed to the Command Interpreter. If the first letter of this line lies in the range A-P then some action will be taken, since the line is then assumed to be a command. It is otherwise ignored.

Keystrokes which do not pass control to the Command Interpreter use a vertical synchronisation clock which does its best to hold the screen steady whilst the keystroke is processed. The Command Interpreter disables the clock for the duration of its operations. The overall effect is that scrolling and cursor operations generate a slight flicker but still maintain a readable display. This feature is not available on ZX80.

FIELDS

The Assembler and Command Interpreter both use a subroutine called GETFLD, which is a general purpose field interpreter. The fields which are valid as arguments are:-

a) A decimal number containing the characters 0-9 (1 to 5 digits).

b) A hexadecimal number identified by a : (colon)

prefix and containing the characters 0-9 A-F (1 to 4 digits).

c) A $ sign, meaning the current contents of the IX register (used as a location counter by the Assembler).

d) One or two characters enclosed in quotation marks (").

(Items a) thru d) are self defining fields)

e) A Symbol. This is a character string consisting of 3 or more characters in the range A-Z 0-9 . (period) and starting with a character in the range A-Z. A symbol is only meaningful when it has been defined by appearance in the label field of an assembled statement (via A or I commands). If a symbol is preceded by a ? (question mark) WHEN REFERENCED AS AN ARGUMENT then the rules for minimum number of characters and alphabetic start character are relaxed. The ? is purely to identify the following string as a symbol,and is not a part of the symbol itself.

f) Any combination of items a) thru f) seperated by + or - characters. + causes subfield addition, - causes subfield subtraction. Parentheses are not used. (A left parenthesis will cause BFLAG to be set non zero).

Fields are terminated by a blank or comma , but not both.

Examples:-

```
"A*"
12345
:12345
FUDGE
ACCOUNTS.PAYABLE
$
12345+"A*"+:12345-FUDGE+$-?HL+ACCOUNTS.PAYABLE
```

FILES

A file is a portion of the text area which is identified by a filename at its start and a filemark (£) as the first character of its terminating line. The filename should have a filemark (£) as its first character and can contain the characters A-Z 0-9 . (period).

Example

```
NOTHING
RUBBISH
£FILE1
THIS IS AN
EXAMPLE OF
A FILE
£
MORE RUBBISH
```

The file £FILE1 is defined as the 3 lines

```
THIS IS AN
EXAMPLE OF
A FILE
```

and may be used by commands such as A (not recommended in this case) P, K & E. The user can define as many files as he wishes, providing each filename is unique.

PARTITIONS

The user can create more file or program space for himself by manipulating the TXTLIM pointer which defines the boundary between text and program areas. Use Dump & Modify or Immediate statements. The symbol table may be relocated by altering LABEND (low memory bottom of Symbol Table) & LABSTK (high memory table top) pointers, and moving the content of memory between them if the table was not empty.

If a non standard memory unit is attached to the ZX80/81 then the ways in which memory can be divided up will depend on the decoding which the unit provides. Bear in mind that execution of a program over the 32K boundary can activate the ZX80/81 display logic hardware, which also relies on duplicate mapping of the 16-32K & 48-64K regions for its effect.

USE OF ASZMIC ROUTINES

Internal subroutines exist in ASZMIC to aid the user in handling I/O and various encoding and decoding functions. These must normally be used from an ASZMIC context (System variables internally consistent, IY=:4000, I=14, IM 1, 22 bytes of stack available). Routines include:-

GETFLD	decode a field
PUTDE	encode a hex number
WRITA	encode a single hex byte
WSTRNG	write contents of print buffer to screen
NRM2	write a character to screen
OUTFRM	write out a frame to screen
KEYBRD	do basic keyboard decode
KEYINT	translate decode to a character
RDCASS	read a character from cassette
WRCASS	write a character to cassette
PRNT	write a line to printer

& many others. See the application notes for definitions, calling sequences and examples.

ASSEMBLER

The Assembler is a small, very fast subprogram within ASZMIC which will accept all standard ZILOG mnemonics for assembler statements. The DEFM, DEFW, DEFB & ORG directives are supported. The EQU directive is supported in a nonstandard form. Instead of "LABEL EQU value" use "LABEL=value" without imbedded blanks. The DEFS value directive is not supported: use ORG $+value instead, which has the same effect of reserving (value) bytes of free space. There are no conditional compilation, listing control or macro directives (hence the SET directive is not implemented).

A comments field may be appended after every assembler statement if preceded by a ; (semicolon). A ; in column 1 makes the whole line commentary.

Always start each file to be assembled with an ORG directive. ASZMIC will probably default you at (TXTLIM) but this is not a design feature and may be withdrawn.

ORG & EQU directives may not use forward references.

Assembler options (see A command in appendix 3) can be used to control object code generation and routing of listings.

Errors tested for include label errors (undefined and doubly defined), Op-Code & relative jump range errors. An error is indicated by a non blank column 1 and error lines are listed even if no list option has been specified.

Use of the Symbol Table preservation option, combined with the "force pass 2" flag, enables many seperate programs with cross references to be assembled seperately to generate a single piece of internally consistent object code. The OFFSET variable can be used to generate an offset between load and execution counters which enables code to be loaded at one location for subsequent movement to and execution at another.

Appendix 2

THE SHIFT KEYS

These are the keys which control the editing and macro functions of ZX.ASZMIC.

Shift-0 TYPING RUBOUT

The cursor is moved 1 position to the left, the character under it is deleted and all subsequent characters to the right are shifted left 1 place. Positioning the cursor to the start of a line and then using Shift-0 will delete the preceding /NL/ to concatenate the two lines.

Shift-9 HOME TO DEBUG MODE

Set ASZMIC in DEBUG mode. Remember the current cursor position for use by the Shift-E keyin. Move the cursor to the bottom line of the bottom page and move the display file pointers so that this line appears on screen. The DEBUG mode flag automatically causes the cursor to assume a slow blink.

Shift-8 CURSOR RIGHT

Move the cursor 1 character to the right but never onto a /NL/ character.

Shift-7 CURSOR UP

Position the cursor at the left of the line above its current position. If necessary scroll down the display so that the cursor remains on screen. Do not move cursor onto or past an End-of-Data character.

Shift-6 CURSOR DOWN

Like CURSOR UP, but cursor moves down & scrolling is upwards.

Shift-5 CURSOR LEFT

Move cursor 1 character to left, but never onto a /NL/ character.

Shift-4 PAGE FLIP UP

Move the display start up 27 lines. Position the cursor at the middle line of the screen. Do not move display onto or over an End-of-Data character.

Shift-3 PAGE FLIP DOWN

As for PAGE FLIP UP, but the display is moved down 27 lines. If you move to the final page then the cursor is positioned at the bottom line.

Shift-2 DELETE FILE

All text from the current cursor position right to the first filemark (£) detected is deleted. No action if a terminating filemark not detected. If the deletion moves the current cursor position onto the final display page then the cursor is homed onto the bottom line.

Shift-l DELETE LINE

Delete the line which currently contains the cursor. Position cursor at start of next line. Do nothing if the cursor is currently on the bottom line of the final page.

Shift-T GO TO DISPLAY TOP

Move the cursor to the Shift-Macro definition line at the top of the display. Display page is altered accordingly.

Shift-R SHIFT MACRO EXECUTION

Independent of the current ASZMIC mode (EDIT/DEBUG) pass the contents of the Shift-Macro line to the Command Interpreter for execution. Cursor and display page are unchanged unless as a result of the executed DEBUG commands. If the line is empty no action results.

Shift-E EDIT RETURN

Change the ASZMIC mode to EDIT. Position the cursor at the location it had when HOME (shift 9) was last pressed. Alter display page if required to keep cursor on screen. Effect unpredictable if editing has taken place whilst in DEBUG mode. Fast blinking cursor identifies EDIT mode.

Shift-W RIGHT JUSTIFY CURSOR

Move the cursor to the rightmost position of the current line.(note:- There is no corresponding left justification key. Use CURSOR UP followed by CURSOR DOWN instead)

Shift-Q EDIT RUBOUT

Like TYPING RUBOUT, but the cursor is not shifted left and it is the character at the current cursor position which is deleted. (It is not possible to delete the last character on a line with a edit rubout i.e. one which lies between 2 /NL/ characters)

Shift-G MERGE

Search down from beginning of display file to find a merge character (>). Copy all text after it up to but not including a filemark (£) into the text position identified by the current cursor position. Effect disastrous if merge and filemark characters missing. In DEBUG mode if a /NL/ is copied then the line it terminates is passed to the command interpreter, thus making the copied text into a Command Macro. In DEBUG mode Shift-G has the same effect as a M> command (see M in Appendix 3).
UNSUPPORTED FEATURE - The Shift-D & Shift-F keys may be implemented on your system. they are like Shift-G but with start character * and < respectively.

****

Input of a normal character causes everything under & to the right of the cursor to be shifted right one place and the character input is placed under the cursor. The cursor is then moved one position to the right. The EOD pointer is incremented.

Input of a /NL/ character will cause a trailing blank to be appended to the line if the previous character was non-blank. The cursor is advanced past the /NL/ to the start of the next line. The display page is moved down one line (i.e. text is scrolled up a line). The EOD pointer is incremented as required.

If input of a character would cause the EOD pointer to be incremented past the (TXTLIM) text area upper partition then the character is ignored.

Appendix 3

DEBUG COMMANDS

Whenever a /NL/ character is written to the text area by ASZMIC when in DEBUG mode the line just terminated is passed to the Command Interpreter, which identifies the command by the first letter on the line and calls the appropriate handler subroutine. A first character not in the range A-P is ignored.
Several DEBUG commands may be concatenated on a single line by using the seperator character sequence ;/ (semicolon slash) and commencing the next command immediately after the slash e.g.

```
D 0 3;/D 5 3;/D :7000 10
```

The contents of the Shift Macro line are also passed to the Command Interpreter when a Shift R key is typed, and this line is also executed whenever a BREAK condition is encountered (Breakpoint, RST 0, Single Step, External NMI).

A ..... ASSEMBLE

A £filename options

The named file is identified and assembled down to its terminating filemark (£) under the control of the option field. If no option field is specified the default is a 2 pass assembly with object code (executable machine code) generation but no assembly listing and no preservation of a previous symbol table.

The option field is converted to an 8-bit byte whose bits when set represent the following options :-

BIT 7..(128).. force second pass
BIT 6..(64).. do not generate object code
BIT 5 (32).. Fine pitch mode on printer. No truncation of listing lines.
BIT 2..(4).. keep and add onto a previous Symbol Table
BIT 1..(2).. direct listing output to printer
BIT 0..(1).. generate assembly listing

Thus an option field to generate a listing on printer without object code would be :43 (decimal 67) or 64+2+1 (you can write it like that).

Assembler lines start with either a ; in column 1, in which case the line is treated as a comment; another non-blank character, in which case the character is assumed to be the first character of a symbol to be defined, or a blank. There then follows an Op-Code or assembly directive delimited by a blank, plus up to 2 argument fields seperated by a comma. A comment field can terminate the line if it is preceded by a ; (semicolon). The file is terminated by a line with a filemark (£) in column 1.

Example:-

```
£EXAMPLE
 ORG :7000
NRM2=:492 ;CHECK VALUE FOR YOUR SYSTEM IN APP. 4
START LD HL,$+120 ;ACTUALLY THE ADDRESS OF TABLE

 LD B,TABLEND-TABLE
LOOP LD A,(HL)
 PUSH HL
 PUSH BC
 CALL NRM2
 POP BC
 POP HL
 INC HL
 DJNZ LOOP
; AND NOW EXIT TO ASZMIC
 RST 0
;
 ORG :7000+120
TABLE DEFM "TEST"
TABLEND=$
£
A £EXAMPLE 1
```

NOTE:- If the variable OFFSET is non-zero its value will be used to relocate the object code produced, although the code itself will be generated to execute at its ORG'd location. OFFSET wraps-around, i.e. ORG :8000 & OFFSET :C000 will locate the object code at :4000.

B ..... BREAKPOINT

```
B address
B
```

If an address is specified then the current breakpoint is removed (saved byte substituted back at the current breakpoint address) and the address specified becomes

the new breakpoint address. The byte at that address is saved and a RST 0 (:C7) code substituted. When the breakpoint is encountered in the course of program execution the saved byte is automatically restored ready for recommencement of execution. If the B command is given without an address then a RST 0 code is placed at the current breakpoint address.

C ..... COPY

C from to bytecount

An intelligent copy operation of the specified number of bytes from the first address specified to the second. If the source and destination ranges overlap the copy will proceed so as not to corrupt the data in the destination range.

D ..... DUMP

D address bytecount
D address

In the first case a formatted dump of the specified number of bytes starting at the address specified is produced. There are 8 hexadecimal bytes to a line preceeded by a hexadecimal address. Long dumps may cause screen blanking for a few seconds whilst the dump is generated. Break key aborts a dump.

The second example is of a Dump & Modify mode. The contents of the address specified is displayed and ASZMIC waits for input. Successive bytes are placed in memory starting at the prompt address. Input is terminated by a . (period) after the prompt. This is the only case where hexadecimal fields do not have to be preceded by a : (colon). The colon is assumed, and decimal values must be input as 0+decimal field.
NOTE:- If bit 1 of ASSFLG has been set to route dump output to the printer then the user must precede input on each line by at least 8 blanks.

E .... EDIT

```
E
E symbol
```

If no argument is specified ASZMIC merely sets itself in EDIT mode (fast cursor blink). In addition, if a symbol is specified, the string is searched for upwards from the end of the file, and the cursor placed at its start. The display page is altered if needed. No action is taken if the symbol is not found. (Any character other than £ . 0-9 A-Z will terminate the symbol comparison operation. £ is only allowed as the first character)

F ..... FILL

```
F from to fillerbyte
```

The specified range is filled with the filler byte.

G ..... GO *** ZX81 ONLY ***

```
G
G address
G address stepcount
```

This is ASZMIC's single step feature. If no arguments are specified then the context is restored from the register image area, and execution of the instruction at the saved Program Counter address (PC1) takes place. A single step break is generated at the end of the instruction, the new context is saved in the register image area, (if the INTJMP variable has been modified to contain an address other than INTRET a jump to (INTJMP) occurs at this point) and the Shift Macro line is executed before returning control to ASZMIC.
If an address is specified then it overwrites the saved Program Counter address (PC1) & becomes the address of the instruction to be executed.

If a stepcount is specified then the operation proceeds as above, but after context save and before Macro execution the saved step count is decremented &, if still positive, then context is restored & the next instruction executed. The rate of execution is typically a hundredth of normal, so a large stepcount may take some time to work thru. Maximum stepcount is :7FFF (decimal 32767). The G command will not work on ZX80.

NOTE:- The single step feature can be used to simulate a ROM breakpoint. Set the breakpoint for the desired address-1 & use G address 32767. Break handling logic will then think it has reached a breakpoint, & terminate single stepping, when the stop address is reached.

H ..... HORRIBLE JUMP

H address

This is not really so nasty. It is just a straightforward jump to the specified address in the context of the ASZMIC Command Interpreter. HL register points to the command line after the address, & the routine jumped to can do processing in the ASZMIC context, terminating with a simple RET. An easy way to link in your own commands.

I ..... IMMEDIATE

I assembler line

An unusual feature which enables immediate assembly and execution of assembler statements. The assembler line follows immediately after the I (i.e. if a blank follows the I then no label has been specified). The line is assembled into object code in the low stack, followed by a break code, and then executed immediately using an internal form of the J command. It thus operates in the saved program context of the REGIM area. After execution the new context is saved just as for a normal BREAK.

Labels should only be defined via the "Label=value" statement form in immediate statements, & the directives ORG, DEFM, DEFB, DEFW & the JR and DJNZ instructions should be avoided.

J ..... JUMP

```
J
J address
```

This is just like the G command, except that there is no single step break in execution, which continues under the program logic unless the B command has been used to insert a breakpoint somewhere in the logic flow. The effect of a breakpoint, or an externally generated NMI interrupt, is similar to the single step interrupt.

K ..... CASSETTE SAVE

```
K£S "i.d." £filename
K£S "i.d." from to
K£S "i.d." from to L
```

ASZMIC uses the same recording protocol as the standard ZX80/81, but the way in which it is used is rather different. ASZMIC can save either files or regions of memory (which presumably contain programs or data). When you type a K command there is a 5 second wait to allow you to turn on your recorder, then the command line itself is written out to tape to identify the file, a further 5 second pause ensues, and the file or memory region is written out.

The command must start with the 5 character sequence K FILEMARK S SPACE QUOTE as shown above since this is used to identify a title line to the cassette load routine. The string enclosed in quotes identifies the file for the load routine, just like a standard ZX80/81. The filename or memory region to be saved is then indicated. If the memory range is followed by "space L" then the load command will alter LABEND to contain the "from" value when the region is loaded.

L ..... LOAD

L "i.d."

In response to this command the cassette input is scanned continuously, and if a valid title line is found it is written to screen and the display activated for 5 seconds, thus generating a running catalogue of the tape contents. If the "i.d."s of the Load Command and the saved title line match then the title line is analysed to determine if a file or memory load is required, and the following file is loaded. In the case of memory load the program/data is loaded in the region specified on the title line unless the variable OFFSET has been set to a non zero value, in which case this value is used as a relocation offset for the region. If the region range in the title line is followed by an L then the region will be presumed to be a Symbol Table, and LABEND will be modified to contain the "from" value. This presumes that save & load took place on the same size systems.
Both K & L commands can be aborted by the Break key, which simulates a BREAK condition.

M ..... MACRO

M character

This is very like Shift G, but you can specify the start identifier yourself instead of using > as a default. Terminator is £ as usual. Do not specify M by itself, always give a character after it and remember it is the first occurrence of the character(except for the Shift macro line) which defines macro start.

N ..... NEW

Loads up BC with (TXTLIM). Sets HL to :3CA (Basic NEW command implementation). Jumps to (TXTLIM). Used in conjunction with dual Basic / ASZMIC board.

O ..... OLD REGISTERS

O

The register image area is dumped as 2 lines of 6 four hexadecimal digit numbers,
The registers appear in the order:-

```
PC   HL   HL'  BC'  DE'  AF'
AF   BC   DE   IX   IY   SP
```

where PC is the Program Counter, SP the Stack Pointer, and the prime suffix (') indicates one of the alternate registers. We recommend that the user place an O command in the Shift Macro line as a useful default, since the registers will then be displayed whenever a BREAK condition occurs.

P ..... PRINT

P £filename

The named file is written out to printer until a terminating file mark is encountered as the first character of a line. The operation may be aborted at any time by pressing the Break key.
NOTE:- If PRTJMP variable is set to a value other than PRTRET then the address in it will be used as the address of the Print Line routine. This enables the implementation of users own print routines.

Hex Addr	Name	Size	Function
4000	MFLAG	1	Flags: bit 7 is edit/debug flag; rest used as NMI counter
4001	ARG1	2	Used by GET2 to decode debug command arguments
4003	ARG2	2	
4005	ASSFLG	1	Assembler options byte described in manual
4006	BOPSAV	1	Breakpoint saved byte
4007	BADDR	1	Breakpoint address
4009	SSCNT	2	Single step counter
400B	OFFSET	2	Offset value used by load & assemble
400D	TEMP	2	Assembler storage
400F	STMEND	2	Latest keyboard decode (see KEYBRD routine)
4011	USAMOD	4	Unused
4015	PRTJMP	2	Address of printer routine
4017	KEYJMP	2	Addres of STMEND analysis routine
4019	INTJMP	2	Address of routine to handle breaks after context save
401B	DADDR	2	Address of debug command interpreter
401D	ECPOSN	2	Cursor address when shift 9 last pressed
401F	CURSOR	2	Cursor address
4021	EOD	2	Address of current end of text
4023	DFILE	2	Address of :76 before first byte of current display
4025	TXTLIM	2	Address of partition between text & program areas
4027	LABEND	2	Bottom of symbol table address
4029	LABSTK	2	Top of symbol table pointer;usually top of memory
402B	PRBUFF	65	Printer buffer
406C	TEMP2	2	Temporary storage; mostly for assembler
406E	LSTEXP	1	No of frames to delay for keyboard debounce
406F	REPEAT	1	No of frames to delay for key repeat
4070	FRAMES	2	Frame count; used for keyin simulation after a no of frames
4072	ELEM1	2	Result of GETFLD analysis
4074 - 4079		6	GETFLD working variables
407A	STKLOW	32	ASZMIC stack area
409A	REGIM	24	Context save area pc1 hl1 hl2 bc2 de2 af2 af1 bc1 del ix1 iy1 sp1
40B4	DSPBGN		Start of text area

E07

app4

ACOM2	089C	ACOMM	08C3	ACOMM	08CB	AF1	40A6	AF2	40A4	ARG1	4001
ARG2	4003	ASEX	086C	ASSFLG	0085	ASSMBL	086F	AUDUMP	0221	AXX	0545
BADDR	4007	BC1	40A8	BC2	40A0	BCOMM	054A	BFLAG	4075	BFLGST	0C2D
BIGMEM	01A5	BLONE	055C	BOPSAV	4006	BRKCHK	030B	BRKOD	00C7	CALLAS	0822
CCC	0BC2	CCOMM	055F	CDLP1	047F	CHRTAB	0D91	CINCRT	046E	CKINV	0C9E
CLDIR	0577	CLNLP	0944	CLNUP	0940	CLPRLD	0BE7	CMDSUB	0293	CMPSTR	02A1
CMPSTX	02AE	CMRTX1	0510	CMSM1	02C1	COMINC	0C22	COMINT	0505	COMMND	052B
COMRTX	050F	COMXTB	051B	CONLIN	0023	CRCHAR	0076	CREN1	0459	CRGEN1	0455
CRHNDL	0C23	CSTR1	02B1	CTAB	0D03	CTAB2	0D4A	CUDRET	048B	CUBRTX	0485
CURSOR	401F	DADDR	401B	DBTLIM	0584	DCLP1	0880	DCOMM	057A	DCOMX	057F
DCONT	0894	DE1	40AA	DE2	40A2	DECLOK	08A1	DEFMNT	0933	DELAY5	00DA
DFILE	4023	DFLIP	032C	DIGCON	0325	DLLP1	00D4	DLLP5	00D2	DLN1	03DE
DLOOP1	058E	DLOOP2	0585	DLY05	00D0	DMD2	05ED	DMDLP1	05DE	DMODIN	05CB
DMOUT	05B0	DMPMOD	05B3	DMPREG	0732	DOLLAR	0C1D	DONCHR	0C6D	DONE	0CB8
DOTTIM	0803	DPG1	03FC	DSPBGN	40B4	DSPSET	019C	DSPTCH	0946	ECOMM	05F0
ECPOSN	401D	EDCODE	0080	EDEXIT	05F7	EDIN	0AC9	EDINX	0AC8	EDLP1	0774
EDLP2	0775	ELEM1	4072	EOD	4021	EODCHR	0005	EODOK	04B6	EX102	09E0
EX103	0A10	EX104	09F5	EX105	0A46	EX107	0A80	EX1213	09BA	EX1NN	09AB
EX1NTA	09BD	EX2DBR	0A12	EX2IND	0A29	EX2NAF	09F8	EX2NAG	0A06	EX2NB	09E2
EX3IR	0A3A	EX3IRX	0A41	EXCF1	09A1	EXCF2	09CF	EXCF3	0A30	EXTNAC	09CB
EXTNHL	0A26	FCOMM	0602	FILCHR	000C	FLDFND	4076	FND2	0C82	FNDLBL	0C71
FNDLCR	0028	FNDRCR	0030	FRAMES	4070	FRMSND	00E3	FRMSNX	00E6	G1COMN	096D
G2IN2	0B87	G2SBCN	0B75	G2SUB	0008	G34TAB	0AA1	G3G4	0A82	G42ER	0A4C
G4IN2	0A68	G4IN3	0A75	G4IN4	0A78	G4REG	0A56	G6INX	0ACD	G6PA	0AC6
G8NIXY	0AF0	G92	0AFF	GAHERE	0B17	GANOBR	0B22	GARNDX	0B11	GAUNC	0B1A
GB01	0B39	GBCOND	0B36	GCENTX	0228	GCHERE	0B47	GCINX2	0977	GCNTSB	024E
GCOM2	0972	GCOMM	0615	GCUNC	0B5C	GDLOOP	0B62	GELOOP	0B68	GET2	0059
GETCHR	0224	GETFLD	0010	GFCHAR	0C60	GFDCLN	0BE0	GFDHEX	0C19	GFDLBL	0C88
GFDLBX	0C85	GFDNUM	0C3F	GFDREG	0CBC	GFDTNM	0C37	GHXVAL	0BDD	GROUP0	095E
GROUP1	095E	GROUP2	097B	GROUP3	0A48	GROUP4	0A4C	GROUP5	0AAB	GROUP6	0AB8
GROUP7	0ACF	GROUP8	0AE2	GROUP9	0AF2	GROUPA	0B02	GROUPB	0B2F	GROUPC	0B3B
GROUPD	0B60	GROUPE	0B67	GROUPF	0B70	HASH	08C6	HASHDN	08DB	HCOMM	0653
HFLIP	0070	HL1	409C	HL2	409E	HLRIN	0CCF	HLRTST	0CD6	HMCHK	03F7
HOME2	0432	HOME3	0446	HOMLP1	0448	HRTIN1	094F	HRTLP	0952	HSHDN2	08FD
HSHERR	0912	HSHLP1	08CA	IAENT	061A	ICOMM	0656	IGNBLK	0020	IMTAB	0AB5
INHERE	0C9E	INICON	017C	INIT	017A	INIT2	0187	INIT3	0181	INTJMP	4019
INTRET	01FA	IX1	40AC	IXIY	4074	IY1	40AE	JCOMM	0668	JGENT	0617
JMP2	063E	JMPTYP	064E	JPTAB	0BCD	KBD1	00F6	KBD2	00F8	KCOMM	066C
KEYADD	027D	KEYBRD	0143	KEYINT	0275	KEYINX	0273	KEYJMP	4017	KEYRET	0252
KYRDLP	014D	LABEND	4027	LABSTK	4029	LBINX	08AB	LCOMM	06B8	LDF2	071A
LDLP1	06C4	LDLP2	06D3	LDLP3	0704	LDLPB	08A3	LDN2	06BD	LDN6	06BC
LINEND	00C1	LIX	022B	LIX2	0234	LIXIMM	0239	LIXSUB	00AD	LODFIL	0717
LOOPGF	0BEF	LSTEXP	406E	LX	4000	LY	4015	LZ	406E	MCOMM	0725
MFLAG	4000	MIDWAY	0C6F	MKDEF	0016	MRGCHR	0017	MRGIN	0350	MRGLP1	0359
MSKINT	004A	MSMTCH	02CE	NCOMM	0729	NEGEND	0CAF	NEGFLG	4077	NEGNOT	0CAE
NGENT	0CB3	NGHNDL	0C32	NLBLF	0C44	NLONP	0023	NOBRK1	0623	NOBXRK	061E
NOGCNT	061F	NOLIST	0858	NORMSV	0691	NOTBRK	0213	NOTHL	0CE9	NOTIXY	096B
NRM2	0490	NRM4	04A4	NRMCHR	048F	NULIN2	04E4	NULINE	04D7	NXTPLN	0817
NXTSTP	0879	OCOMM	0732	OFFSET	400B	OFRM1	016B	OFRM2	016F	OUTFRM	0167
PARSE	0825	PC1	409A	PCHAR	07F7	PCOMM	0751	PDOT	0808	PDOTX	07F8
PRBUFF	402B	PRCLR	02FF	PREADY	07D8	PRIGET	0923	PRNTER	07CC	PROFF	081D
PRTJMP	4015	PRTRET	07D1	PSLP	0755	PTCON	00CB	PUTA	0B9C	PUTB	0B9B
PUTDE	0317	PUTDEF	0314	PUTENT	0BAD	PUTNN	0B8D	PUTNNX	0B91	PUTOUT	0BBE
QCOMM	0761	QUOTE	000B	RASCON	07E1	RASPRS	0043	RASTER	07D4	RCOMM	0761
RCOMP	0D98	RCOMPX	0DAF	RDC1	07A1	RDC2	07A3	RDC3	07B1	RDC4	07B3
RDC5	07C4	RDCASS	07A1	RDCX	07AF	REG	4078	REGCOD	4079	REGIM	409A
REPEAT	406F	RESTOR	0628	RETNR	0C23	RLB	0B95	RLOAD	0CE4	RUBDNO	037E
RUBOK	0376	SAVMEM	018B	SAVSTR	001B	SFILE	067F	SHFT0	0424	SHFT1	03BC
SHFT8	0454	SHFT9	042B	SHFTD	0337	SHFTE	0396	SHFTF	033B	SHFTG	033F
SHFTQ	036A	SHFTR	03AF	SHFTT	03B5	SHFTW	0392	SHIFTS	0337	SHOME	03AC
SINGLE	01D5	SINGLX	01F0	SNLP1	0699	SNLP2	06A9	SP1	40B0	SRECUR	067A
SSCNT	4009	START	0000	STENT	039D	STKLOW	407A	STMEND	400F	STRBOK	0772
STRCDB	0762	STRSCH	0761	TABLE	0082	TEMP	400D	TEMP2	406C	TIMING	0066
TXTLIM	4025	UPDG1	0416	UPDG2	041F	USAMOD	4011	V	0040	VRSION	0E07
WCNORM	08F7	WCRPT	08E0	WCTAB	0DC6	WCTABX	0DE3	WRC1	0782	WRC2	078A
WRCASS	077F	WREG	073B	WREGL	0741	WRITA	031C	WSTR1	02F0	WSTRG2	02E0
WSTRLP	02DA	WSTRNG	02D4	X1	0010	X2	0020	X3	0030	ZXJPND	0505
ZXJPTB	04F3										

Appendix 4

SYSTEM ADDRESSES

To ensure that the ASZMIC ROM corresponds to the version described here, check the two bytes at MKDEF against the declared variable "VRSION" in the list. They should correspond; i.e. VRSION E04 has MKDEF 04 0E.

ACOM2	0891	CREN1	045B	EDEXIT	05F7	G4IN2	0A69
ACOMIN	08C4	CRGEN1	0457	EDIN	0ACA	G4IN3	0A76
ACOMM	052D	CRHNDL	0C24	EDINX	0AC9	G4IN4	0A79
AF1	40A6	CSTR1	02B3	EDLP1	0775	G4REG	0A57
AF2	40A4	CTAB	0D04	EDLP2	0776	G6INX	0ACE
ARG1	4001	CTAB2	0D4B	ELEM1	4072	G6PA	0AC7
ARG2	4003	CUDRET	048D	EOD	4021	G8NIXY	0AF1
ASEXIT	086D	CUDRTX	0487	EODCHR	0005	G92	0B00
ASSFLG	4005	CURSOR	401F	EODOK	04B8	GAHERE	0B18
ASSMBL	0870	DADDR	401B	EX102	09E1	GANOBR	0B23
AUDUMP	0223	DBTLIM	0584	EX103	0A11	GARNDX	0B12
AXX	0547	DCLP1	0881	EX104	09F6	GAUNC	0B1B
BADDR	4007	DCOMM	057A	EX105	0A47	GB01	0B3A
BC1	40A8	DCOMX	057F	EX107	0A81	GBCOND	0B37
BC2	40A0	DCONT	0895	EX1213	09BB	GCENTX	022A
BCOMM	054C	DE1	40AA	EX1NN	09AC	GCHERE	0B48
BFLAG	4075	DE2	40A2	EX1NTA	09BE	GCINX2	0978
BFLGST	0C2E	DECLOK	08A2	EX2DBR	0A13	GCNTSB	0250
BIGMEM	01A7	DEFMNT	0934	EX2IND	0A2A	GCOM2	0973
BLONE	055E	DELAY5	00DC	EX2NAF	09F9	GCOMM	0615
BOPSAV	4006	DFILE	4023	EX2NAG	0A07	GCUNC	0B5D
BRKCHK	030D	DFLIP	032E	EX2NB	09E3	GDLOOP	0B63
BRKOD	00C7	DIGCON	0327	EX3IR	0A3B	GELOOP	0B69
CALLAS	0823	DLLP1	00D6	EX3IRX	0A42	GET2	0059
CCC	0BC3	DLLP5	00D4	EXCF1	09A2	GETCHR	0226
CCOMM	0561	DLN1	03E0	EXCF2	09D0	GETFLD	0010
CDLP1	0481	DLOOP1	058E	EXCF3	0A31	GFCHAR	0C61
CHRTAB	0D92	DLOOP2	0585	EXTNAC	09CC	GFDCLN	0BE1
CINCRT	0470	DLY05	00D2	EXTNHL	0A27	GFDHEX	0C1A
CKINV	0C9F	DMD2	05ED	FCOMM	0602	GFDLBL	0C89
CLDIR	0577	DMDLP1	05DE	FILCHR	000C	GFDLBX	0C86
CLNLP	0945	DMODIN	05CB	FLDFND	4076	GFDNUM	0C40
CLNUP	0941	DMOUT	05B0	FND2	0C83	GFDREG	0CBD
CLPRLD	0BE8	DMPMOD	05B3	FNDLBL	0C72	GFDTNM	0C38
CMDSUB	0295	DMPREG	0733	FNDLCR	0028	GHXVAL	0BDE
CMPSTR	02A3	DOLLAR	0C1E	FNDRCR	0030	GROUP0	095F
CMPSTX	02B0	DONCHR	0C6E	FRAMES	4070	GROUP1	095F
CMRTX1	0512	DONE	0CB9	FRMSND	00E5	GROUP2	097C
CMSM1	02C3	DOTTIM	0804	FRMSNX	00E8	GROUP3	0A49
COMINC	0C23	DPG1	03FE	G1COMN	096E	GROUP4	0A4D
COMINT	0507	DSPBGN	40B4	G2IN2	0B88	GROUP5	0AAC
COMMND	052D	DSPSET	019E	G2SBCN	0B76	GROUP6	0AB9
COMRTX	0511	DSPTCH	0947	G2SUB	0008	GROUP7	0AD0
COMXTB	051D	ECOMM	05F0	G34TAB	0AA2	GROUP8	0AE3
CONLIN	0023	ECPOSN	401D	G3G4	0A83	GROUP9	0AF3
CRCHAR	0076	EDCODE	0080	G42ER	0A4D	GROUPA	0B03

GROUPB	0B30	LDLP2	06D2	PDOT	0809	SHFT9	042D
GROUPC	0B3C	LDLP3	0703	PDOTX	07F9	SHFTD	0339
GROUPD	0B61	LDLPB	08A4	PRBUFF	402B	SHFTE	0398
GROUPE	0B68	LDN2	06BC	PRCLR	0301	SHFTF	033D
GROUPF	0B71	LDN6	06BB	PREADY	07D9	SHFTG	0341
HASH	08C7	LINEND	00C1	PRIGET	0924	SHFTQ	036C
HASHDN	08DC	LIX	022D	PRNTER	07CD	SHFTR	03B1
HCOMM	0652	LIX2	0236	PROFF	081E	SHFTT	03B7
HFLIP	0070	LIXIMM	023B	PRTJMP	4015	SHFTW	0394
HL1	409C	LIXSUB	00AD	PRTRET	07D2	SHIFTS	0339
HL2	409E	LODFIL	0716	PSLP	0756	SHOME	03AE
HLRIN	0CD0	LOOPGF	0BF0	PTCON	00CB	SINGLE	01D7
HLRTST	0CD7	LSTEXP	406E	PUTA	0B9D	SINGLX	01F2
HMCHK	03F9	LX	4000	PUTB	0B9C	SNLP1	0698
HOME2	0434	LY	4015	PUTDE	0319	SNLP2	06A8
HOME3	0448	LZ	406E	PUTDEF	0316	SP1	40B0
HOMLP1	044A	MCOMM	0724	PUTENT	0BAE	SRECUR	0679
HRTIN1	0950	MFLAG	4000	PUTNN	0B8E	SSCNT	4009
HRTLP	0953	MIDWAY	0C70	PUTNNX	0B92	START	0000
HSHDN2	08FE	MKDEF	0016	PUTOUT	0BBF	STENT	039F
HSHERR	0913	MRGCHR	0017	QCOMM	0762	STKLOW	407A
HSHLP1	08CB	MRGIN	0352	QUOTE	000B	STMEND	400F
IAENT	061A	MRGLP1	035B	RASCON	07E2	STRBOK	0773
ICOMM	0655	MSKINT	004A	RASPRS	0043	STRCDB	0763
IGNBLK	0020	MSMTCH	02D0	RASTER	07D5	STRSCH	0762
IMTAB	0AB6	NCOMM	0728	RCOMM	0762	TABLE	0082
INHERE	0C9F	NEGEND	0CB0	RCOMP	0D99	TEMP	400D
INICON	017E	NEGFLG	4077	RCOMPX	0DB0	TEMP2	406C
INIT	017C	NEGNOT	0CAF	RDC1	07A2	TIMING	0066
INIT2	0189	NGENT	0CB4	RDC2	07A4	TXTLIM	4025
INIT3	0183	NGHNDL	0C33	RDC3	07B2	UPDG1	0418
INTJMP	4019	NLBLF	0C45	RDC4	07B4	UPDG2	0421
INTRET	01FC	NLONP	0023	RDC5	07C5	USAMOD	4011
IX1	40AC	NOBRK1	0623	RDCASS	07A2	V	0040
IXIY	4074	NOBXRK	061E	RDCX	07B0	VRSION	0E04
IY1	40AE	NOGCNT	061F	REG	4078	WCNORM	08F8
JCOMM	0667	NOLIST	0859	REGCOD	4079	WCRPT	08E1
JGENT	0617	NORMSV	0690	REGIM	409A	WCTAB	0DC7
JMP2	063D	NOTBRK	0215	REPEAT	406F	WCTABX	0DE4
JMPTYP	064D	NOTHL	0CEA	RESTOR	0628	WRC1	0783
JPTAB	0BCE	NOTIXY	096C	RETNR	0C24	WRC2	078B
KBD1	00F8	NRM2	0492	RLB	0B96	WRCASS	0780
KBD2	00FA	NRM4	04A6	RLOAD	0CE5	WREG	073C
KCOMM	066B	NRMCHR	0491	RUBDNO	0380	WREGL	0742
KEYADD	027F	NULIN2	04E6	RUBOK	0378	WRITA	031E
KEYBRD	0145	NULINE	04D9	SAVMEM	018D	WSTR1	02F2
KEYINT	0277	NXTPLN	0818	SAVSTR	001B	WSTRG2	02E2
KEYINX	0275	NXTSTP	087A	SFILE	067E	WSTRLP	02DC
KEYJMP	4017	OCOMM	0733	SHFT0	0426	WSTRNG	02D6
KEYRET	0254	OFFSET	400B	SHFT1	03BE	X1	0010
KYRDLP	014F	OFRM1	016D	SHFT2	03E2		
LABEND	4027	OFRM2	0171	SHFT3	0408		
LABSTK	4029	OUTFRM	0169	SHFT4	0412		
LBINX	08AC	PARSE	0826	SHFT5	0423		
LCOMM	06B7	PC1	409A	SHFT6	0473		
LDF2	0719	PCHAR	07F8	SHFT7	045F		
LDLP1	06C3	PCOMM	0752	SHFT8	0456		

Appendix 5

Z80 INSTRUCTIONS

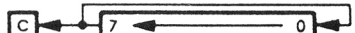
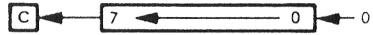
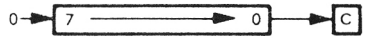
TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
I/O	IN	A,port	2							[A]←[port] Input to Accumulator from directly addressed I/O port. Address Bus: A0-A7: port A8-A15: [A]
	IN	reg,(C)	2		X	X	P	X	0	[reg]←[[C]] Input to register from I/O port addressed by the contents of C.* If second byte is 70_{16}, only the flags will be affected.
	INIR		2		1	?	?	?	1	Repeat until [B]=0: [[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a block of data from I/O port addressed by contents of C to memory location addressed by contents of HL, going from low addresses to high. Contents of B serve as a count of bytes remaining to be transferred.*
	INDR		2		1	?	?	?	1	Repeat until [B]=0: [[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a block of data from I/O port addressed by contents of C to memory location addressed by contents of HL, going from high addresses to low. Contents of B serve as a count of bytes remaining to be transferred.*
	INI		2		X	?	?	?	1	[[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a byte of data from I/O port addressed by contents of C to memory location addressed by contents of HL. Decrement byte count and increment destination address.*
	IND		2		X	?	?	?	1	[[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a byte of data from I/O port addressed by contents of C to memory location addressed by contents of HL. Decrement both byte count and destination address.*
	OUT	port,A	2							[port]←[A] Output from Accumulator to directly addressed I/O port. Address Bus: A0-A7: port A8-A15: [A]
	OUT	(C),reg	2							[[C]]←[reg] Output from register to I/O port addressed by the contents of C.*
	OTIR		2		1	?	?	?	1	Repeat until [B]=0: [[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a block of data from memory location addressed by contents of HL to I/O port addressed by contents of C, going from low memory to high. Contents of B serve as a count of bytes remaining to be transferred.*

*Address Bus A0-A7: [C]
A8-A15: [B]

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
I/O (Continued)	OTDR		2		1	?	?	?	1	Repeat until [B]=0: [[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a block of data from memory location addressed by contents of HL to I/O port addressed by contents of C, going from high memory to low. Contents of B serve as a count of bytes remaining to be transferred.*
	OUTi		2		X	?	?	?	1	[[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a byte of data from memory location addressed by contents of HL to I/O port addressed by contents of C. Decrement byte count and increment source address.*
	OUTD		2		X	?	?	?	1	[[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a byte of data from memory location addressed by contents of HL to I/O port addressed by contents of C. Decrement both byte count and source address.*
PRIMARY MEMORY REFERENCE	LD	A,(addr)	3							[A]←[addr] Load Accumulator from directly addressed memory location
	LD	HL,(addr)	3							[H]←[addr + 1], [L]←[addr] Load HL from directly addressed memory
	LD	rp,(addr) xy,(addr)	4							[rp(HI)]←[addr + 1], [rp(LO)]←[addr] or [xy(HI)]←[addr + 1], [xy(LO)]←[addr] Load register pair or Index register from directly addressed memory.
	LD	(addr),A	3							[addr]←[A] Store Accumulator contents in directly addressed memory location.
	LD	(addr),HL	3							[addr + 1]←[H], [addr]←[L] Store contents of HL to directly addressed memory location
	LD	(addr),rp (addr),xy	4							[addr + 1]←[rp(HI)], [addr]←[rp(LO)] or [addr + 1]←[xy(HI)], [addr]←[xy(LO)] Store contents of register pair or Index register to directly addressed memory.
	LD	A,(BC) A,(DE)	1							[A]←[[BC]] or [A]←[[DE]] Load Accumulator from memory location addressed by the contents of the specified register pair
	LD	reg,(HL)	1							[reg]←[[HL]] Load register from memory location addressed by contents of HL.
	LD	(BC),A (DE),A	1							[[BC]]←[A] or [[DE]]←[A] Store Accumulator to memory location addressed by the contents of the specified register pair.
	LD	(HL),reg	1							[[HL]]←[reg] Store register contents to memory location addressed by the contents of HL.
	LD	reg,(xy + disp)	3							[reg]←[[xy] + disp] Load register from memory location using base relative addressing.
	LD	(xy + disp),reg	3							[[xy] + disp]←[reg] Store register to memory location addressed relative to contents of Index register.

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
BLOCK TRANSFER AND SEARCH	LDIR		2				0	0	0	Repeat until [BC]=0: [[DE]]←[[HL]] [DE]←[DE] + 1 [HL]←[HL] + 1 [BC]←[BC] - 1 Transfer a block of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE, going from low addresses to high. Contents of BC serve as a count of bytes to be transferred
	LDDR		2				0	0	0	Repeat until [BC]=0: [[DE]]←[[HL]] [DE]←[DE] - 1 [HL]←[HL] - 1 [BC]←[BC] - 1 Transfer a block of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE, going from high addresses to low. Contents of BC serve as a count of bytes to be transferred
	LDI		2				X	0	0	[[DE]]←[[HL]] [DE]←[DE] + 1 [HL]←[HL] + 1 [BC]←[BC] - 1 Transfer one byte of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE. Increment source and destination addresses and decrement byte count
	LDD		2				X	0	0	[[DE]]←[[HL]] [DE]←[DE] - 1 [HL]←[HL] - 1 [BC]←[BC] - 1 Transfer one byte of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE. Decrement source and destination addresses and byte count.
	CPIR		2		X	X	X	X	1	Repeat until [A]=[[HL]] or [BC]=0 [A] - [[HL]] (only flags are affected) [HL]←[HL] + 1 [BC]←[BC] - 1 Compare contents of Accumulator with those of memory block addressed by contents of HL going from low addresses to high. Stop when a match is found or when the byte count becomes zero.
	CPDR		2		X	X	X	X	1	Repeat until [A]=[[HL]] or [BC]=0 [A] - [[HL]] (only flags are affected) [HL]←[HL] - 1 [BC]←[BC] - 1 Compare contents of Accumulator with those of memory block addressed by contents of HL going from high addresses to low. Stop when a match is found or when the byte count becomes zero.

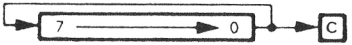
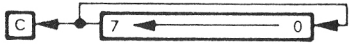
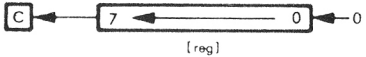
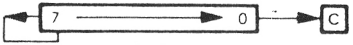
TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
BLOCK TRANSFER AND SEARCH (Continued)	CPI		2		X	X	X	X	1	[A] - [[HL]] (only flags are affected) [HL]←[HL] + 1 [BC]←[BC] - 1 Compare contents of Accumulator with those of memory location addressed by contents of HL Increment address and decrement byte count.
	CPD		2		X	X	X	X	1	[A] - [[HL]] (only flags are affected) [HL]←[HL] - 1 [BC]←[BC] - 1 Compare contents of Accumulator with those of memory location addressed by contents of HL Decrement address and byte count.
SECONDARY MEMORY REFERENCE	ADD	(HL)	1	X	X	X	O	X	0	[A]←[A] + [[HL]] or [A]←[A] + [[xy] + disp] Add to Accumulator using implied addressing or base relative addressing
		(xy + disp)	3							
	ADC	(HL)	1	X	X	X	O	X	0	[A]←[A] + [[HL]] + C or [A]←[A] + [[xy] + disp] + C Add with Carry using implied addressing or base relative addressing
		(xy + disp)	3							
	SUB	(HL)	1	X	X	X	O	X	1	[A]←[A] - [[HL]] or [A]←[A] - [[xy] + disp] Subtract from Accumulator using implied addressing or base relative addressing
		(xy + disp)	3							
	SBC	(HL)	1	X	X	X	O	X	1	[A]←[A] - [[HL] - C or [A]←[A] - [[xy] + disp] - C Subtract with Carry using implied addressing or base relative addressing
		(xy + disp)	3							
	AND	(HL)	1	0	X	X	P	1	0	[A]←[A] ∧ [[HL]] or [A]←[A] ∧ [[xy] + disp] AND with Accumulator using implied addressing or base relative addressing
		(xy + disp)	3							
	OR	(HL)	1	0	X	X	P	1	0	[A]←[A] ∨ [[HL]] or [A]←[A] ∨ [[xy] + disp] OR with Accumulator using implied addressing or base relative addressing
		(xy + disp)	3							
	XOR	(HL)	1	0	X	X	P	1	0	[A]←[A] ⊻ [[HL]] or [A]←[A] ⊻ [[xy] + disp] Exclusive-OR with Accumulator using implied addressing or base relative addressing
		(xy + disp)	3							
	CP	(HL)	1	X	X	X	O	X	1	[A] - [[HL]] or [A] - [[xy] + disp] Compare with Accumulator using implied addressing or base relative addressing. Only the flags are affected.
		(xy + disp)	3							
	INC									
		(HL)	1		X	X	O	X	0	[[HL]]←[[HL]] + 1 or [[xy] + disp]←[[xy] + disp] + 1 Increment using implied addressing or base relative addressing
		(xy + disp)	3							
	DEC	(HL)	1		X	X	O	X	1	[[HL]]←[[HL]] - 1 or [[xy] + disp]←[[xy] + disp] - 1 Decrement using implied addressing or base relative addressing
		(xy + disp)	3							

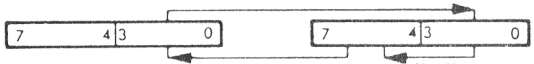
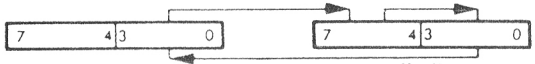
TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
MEMORY SHIFT AND ROTATE	RLC	(HL) (xy + disp)	2 4	X	X	X	P	0	0	 [[HL]] or [[xy] + disp] Rotate contents of memory location (implied or base relative addressing) left with branch Carry
	RL	(HL) (xy + disp)	2 4	X	X	X	P	0	0	 [[HL]] or [[xy] + disp] Rotate contents of memory location left through Carry.
	RRC	(HL) (xy + disp)	2 4	X	X	X	P	0	0	 [[HL]] or [[xy] + disp] Rotate contents of memory location right with branch Carry.
	RR	(HL) (xy + disp)	2 4	X	X	X	P	0	0	 [[HL]] or [[xy] + disp] Rotate contents of memory location right through Carry.
	SLA	(HL) (xy + disp)	2 4	X	X	X	P	0	0	 [[HL]] or [[xy] + disp] Shift contents of memory location left and clear LSB (Arithmetic Shift).
	SRA	(HL) (xy + disp)	2 4	X	X	X	P	0	0	 [[HL]] or [[xy] + disp] Shift contents of memory location right and preserve MSB (Arithmetic Shift).
	SRL	(HL) (xy + disp)	2 4	X	X	X	P	0	0	 [[HL]] or [[xy] + disp] Shift contents of memory location right and clear MSB (Logical Shift).
IMMEDIATE	LD	reg,data	2							[reg] ← data Load immediate into register.
	LD	rp,data16 xy,data16	3 4							[rp] ← data16 or [xy] ← data16 Load 16 bits of immediate data into register pair or Index register.
	LD	(HL),data (xy + disp),data	2 4							[[HL]] ← data or [[xy] + disp] ← data Load immediate into memory location using implied or base relative addressing.

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
JUMP	JP	label	3							[PC]←label Jump to instruction at address represented by label.
	JR	disp	2							[PC]←[PC] + 2 + disp Jump relative to present contents of Program Counter.
	JP	(HL) (xy)	1 2							[PC]←[HL] or [PC]←[xy] Jump to address contained in HL or Index register.
SUBROUTINE CALL AND RETURN	CALL	label	3							[[SP]-1]←[PC(HI)] [[SP]-2]←[PC(LO)] [SP]←[SP]-2 [PC]←label Jump to subroutine starting at address represented by label.
	CALL	cond,label	3							Jump to subroutine if condition is satisfied; otherwise, continue in sequence.
	RET		1							[PC(LO)]←[[SP]] [PC(HI)]←[[SP] + 1] [SP]←[SP] + 2 Return from subroutine
	RET	cond	1							Return from subroutine if condition is satisfied; otherwise, continue in sequence
IMMEDIATE OPERATE	ADD	data	2	X	X	X	O	X	0	[A]←[A] + data Add immediate to Accumulator.
	ADC	data	2	X	X	X	O	X	0	[A]←[A] + data + C Add immediate with Carry
	SUB	data	2	X	X	X	O	X	1	[A]←[A] - data Subtract immediate from Accumulator
	SBC	data	2	X	X	X	O	X	1	[A]←[A] - data - C Subtract immediate with Carry
	AND	data	2	0	X	X	P	1	0	[A]←[A] ∧ data AND immediate with Accumulator
	OR	data	2	0	X	X	P	1	0	[A]←[A] ∨ data OR immediate with Accumulator.
	XOR	data	2	0	X	X	P	1	0	[A]←[A]⊻data Exclusive-OR immediate with Accumulator.
	CP	data	2	X	X	X	O	X	1	[A] - data Compare immediate data with Accumulator contents; only the flags are affected

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
JUMP ON CONDITION	JP	cond,label	3							If cond. then [PC]←label Jump to instruction at address represented by label if the condition is true
	JR	C,disp	2							If C=1, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Carry flag is set
	JR	NC,disp	2							If C=0, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Carry flag is reset
	JR	Z,disp	2							If Z=1, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Zero flag is set.
	JR	NZ,disp	2							If Z=0, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Zero flag is reset
	DJNZ	disp	2							[B]←[B] - 1 If [B] ≠ 0, then [PC]←[PC] + 2 + disp Decrement contents of B and Jump relative to contents of Program Counter if result is not 0.
REGISTER-REGISTER MOVE	LD	dst,src	1							[dst]←[src] Move contents of source register to destination register. Register designations src and dst may each be A, B, C, D, E, H or L
	LD	A,IV	2		X	X	I	0	0	[A]←[IV] Move contents of Interrupt Vector register to Accumulator
	LD	A,R	2		X	X	I	0	0	[A]←[R] Move contents of Refresh register to Accumulator
	LD	IV,A	2							[IV]←[A] Load Interrupt Vector register from Accumulator
	LD	R,A	2							[R]←[A] Load Refresh register from Accumulator
	LD	SP,HL	1							[SP]←[HL] Move contents of HL to Stack Pointer
	LD	SP,xy	2							[SP]←[xy] Move contents of Index register to Stack Pointer
	EX	DE,HL	1							[DE]←→[HL] Exchange contents of DE and HL
	EX	AF,AF'	1							[AF]←→[AF'] Exchange program status and alternate program status
	EXX		1							$\begin{pmatrix}[BC]\\[DE]\\[HL]\end{pmatrix} \longleftrightarrow \begin{pmatrix}[BC']\\[DE']\\[HL']\end{pmatrix}$ Exchange register pairs and alternate register pairs

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
REGISTER-REGISTER OPERATE	ADD	reg	1	X	X	X	O	X	0	[A]←[A] + [reg] Add contents of register to Accumulator.
	ADC	reg	1	X	X	X	O	X	0	[A]←[A] + [reg] + C Add contents of register and Carry to Accumulator.
	SUB	reg	1	X	X	X	O	X	1	[A]←[A] - [reg] Subtract contents of register from Accumulator.
	SBC	reg	1	X	X	X	O	X	1	[A]←[A] - [reg] - C Subtract contents of register and Carry from Accumulator
	AND	reg	1	0	X	X	P	1	0	[A]←[A] ∧ [reg] AND contents of register with contents of Accumulator
	OR	reg	1	0	X	X	P	1	0	[A]←[A] ∨ [reg] OR contents of register with contents of Accumulator
	XOR	reg	1	0	X	X	P	1	0	[A]←[A] ⊻ [reg] Exclusive-OR contents of register with contents of Accumulator
	CP	reg	1	X	X	X	V	X	1	[A] - [reg] Compare contents of register with contents of Accumulator. Only the flags are affected.
	ADD	HL,rp	1	X				?	0	[HL]←[HL] + [rp] 16-bit add register pair contents to contents of HL
	ADC	HL,rp	2	X	X	X	O	?	0	[HL]←[HL] + [rp] + C 16-bit add with Carry register pair contents to contents of HL
	SBC	HL,rp	2	X	X	X	O	?	1	[HL]←[HL] - [rp] - C 16-bit subtract with Carry register pair contents from contents of HL
	ADD	IX,pp	2	X				?	0	[IX]←[IX] + [pp] 16-bit add register pair contents to contents of Index register IX (pp=BC, DE, IX, SP)
	ADD	IY,rr	2	X				?	0	[IY]←[IY] + [rr] 16-bit add register pair contents to contents of Index register IY (rr=BC, DE, IY, SP).
REGISTER OPERATE	DAA		1	X	X	X	P	X		Decimal adjust Accumulator, assuming that Accumulator contents are the sum or difference of BCD operands.
	CPL		1					1	1	[A]←[$\overline{A}$] Complement Accumulator (ones complement)
	NEG		2	X	X	X	O	X	1	[A]←[$\overline{A}$] + 1 Negate Accumulator (twos complement)
	INC	reg	1		X	X	O	X	0	[reg]←[reg] + 1 Increment register contents.
	INC	rp xy	1							[rp]←[rp] + 1 or [xy]←[xy] + 1 Increment contents of register pair or Index register
	DEC	reg	2		X	X	O	X	1	[reg]←[reg] - 1 Decrement register contents.
	DEC	rp xy	1 2							[rp]←[rp] - 1 or [xy]←[xy] - 1 Decrement contents of register pair or Index register

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
REGISTER SHIFT AND ROTATE	RLCA		1	X				0	0	 [A] Rotate Accumulator left with branch Carry.
	RLA		1	X				0	0	 [A] Rotate Accumulator left through Carry.
	RRCA		1	X				0	0	 [A] Rotate Accumulator right with branch Carry.
	RRA		1	X				0	0	 [A] Rotate Accumulator right through Carry.
	RLC	reg	2	X	X	X	P	0	0	 [reg] Rotate contents of register left with branch Carry
	RL	reg	2	X	X	X	P	0	0	 [reg] Rotate contents of register left through Carry.
	RRC	reg	2	X	X	X	P	0	0	 [reg] Rotate contents of register right with branch Carry.
	RR	reg	2	X	X	X	P	0	0	 [reg] Rotate contents of register right through Carry.
	SLA	reg	2	X	X	X	P	0	0	 [reg] Shift contents of register left and clear LSB (Arithmetic Shift).
	SRA	reg	2	X	X	X	P	0	0	 [reg] Shift contents of register right and preserve MSB (Arithmetic Shift)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
REGISTER SHIFT AND ROTATE (Continued)	SRL		2	X	X	X	P	0	0	 [reg] Shift contents of register right and clear MSB (Logical Shift).
	RLD		2		X	X	P	0	0	 [A] [[HL]] Rotate one BCD digit left between the Accumulator and memory location (implied addressing). Contents of the upper half of the Accumulator are not affected.
	RRD	reg	2		X	X	P	0	0	 [A] [[HL]] Rotate one BCD digit right between the Accumulator and memory location (implied addressing). Contents of the upper half of the Accumulator are not affected.
BIT MANIPULATION	BIT	b,reg	2		X	?	?	1	0	Z ← $\overline{reg(b)}$ Zero flag contains complement of the selected register bit.
	BIT	b,(HL) b,(xy + disp)	2 4		X	?	?	1	0	Z ← $\overline{[[HL]](b)}$ or Z ← $\overline{[[xy] + disp](b)}$ Zero flag contains complement of selected bit of the memory location (implied addressing or base relative addressing).
	SET	b,reg	2							reg(b) ← 1 Set indicated register bit.
	SET	b,(HL) b,(xy + disp)	2 4							[[HL]](b) ← 1 or [[xy] + disp](b) ← 1 Set indicated bit of memory location (implied addressing or base relative addressing).
	RES	b,reg	2							reg(b) ← 0 Reset indicated register bit.
	RES	b,(HL) b,(xy + disp)	2 4							[[HL]](b) ← 0 or [[xy] + disp](b) ← 0 Reset indicated bit in memory location (implied addressing or base relative addressing).
STACK	PUSH	pr xy	1 2							[[SP]-1] ← [pr(HI)] [[SP]-2] ← [pr(LO)] [SP] ← [SP]-2 Put contents of register pair or Index register on top of Stack and decrement Stack Pointer.
	POP	pr xy	1 2							[pr(LO)] ← [[SP]] [pr(HI)] ← [[SP] + 1] [SP] ← [SP] + 2 Put contents of top of Stack in register pair or Index register and increment Stack Pointer.
	EX	(SP),HL (SP),xy	1 2							[H] ← [[SP] + 1] [L] ← [[SP]] Exchange contents of HL or Index register and top of Stack.

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUS						OPERATION PERFORMED
				C	Z	S	P/O	A_C	N	
INTERRUPT	DI		1							Disable interrupts
	EI		1							Enable interrupts.
	RST	n	1							[[SP]-1]←[PC(HI)] [[SP]-2]←[PC(LO)] [SP]←[SP]-2 [PC]←$(8 \cdot n)_{16}$ Restart at designated location.
	RETI		2							Return from interrupt.
	RETN		2							Return from nonmaskable interrupt.
	IM	0 1 2	2							Set interrupt mode 0, 1, or 2
STATUS	SCF		1	1				0	0	C ← 1 Set Carry flag.
	CCF		1	X				?	0	C ← $\overline{C}$ Complement Carry flag.
	NOP		1							No operation — volatile memories are refreshed.
	HALT		1							CPU halts, executes NOPs to refresh volatile memories.

Appendix 6

APPLICATION NOTES

Appendix 4 contains a variable VRSION which defines the model number of ASZMIC which you are using. This is also stored as DEFW at MKDEF in the ROM so you can check that the addresses given in Appendix 4 are correct for the ROM you are using.

NAME ... BRKCHK
FUNCTION ... Check break key; simulate BREAK condition if pressed
CALLING SEQUENCE ... CALL BRKCHK
EFFECT...Also sets sync pulse level low
USES ... Abort routines

****

NAME ... CMPSTR
FUNCTION ... Compare two strings
CALLING SEQUENCE ... CALL CMPSTR (HL) points to found string-1,(DE) to base string
EFFECT...end comparison on encountering a /NL/ or any character < . (if the found string is not preceded by a character less than . (period) then the comparison is declared invalid. If the first character of the found (& base) strings is a filemark then a /NL/ must precede the £ in the found string for the comparison to be accepted).
If strings are the same then carry reset, HL points at the beginning of the found string, DE points at the base string delimiter. If the strings are not identical carry is set and HL,DE unchanged.
REGISTERS USED ... A,HL,DE
USES ... String identification

****

NAME ... COMMANDS
FUNCTION ... £
CALLING SEQUENCE ... All commands have form *COMM,
where * is DEBUG letter.
HL points at first non-blank char after DEBUG letter.
They exit by RET (to LIX)
EFFECT ... Performs actions appropriate to the Command.
USES ... £

****

NAME ... DFLIP
FUNCTION ... Set address jumped to for Command
Interpretation
CALLING SEQUENCE ... CALL DFLIP HL contains new
Command Int. address
EFFECT ... Loads HL into DADDR, takes old value and
returns with it on stack. Uses HL.
USES ... User handling of newline character i.e. ASZMIC
EDITOR used as an input routine for user programs.

****

NAME ... DELAY5
FUNCTION ... Display for 5 seconds
CALLING SEQUENCE ... CALL DELAY5
EFFECT ... Send out a display for 5 seconds and return
to caller. FRAMES is loaded with 250 & FRMSND invoked.
See FRMSND for details of return. There is another
delay called DLY05 which sets sync low and then loops
for half a second before returning (uses A & DE).
USES ... £

****

NAME ... EDLP1
FUNCTION ... Entry point for STRSCH which expects DE to
point to base string, HL to point to high end of search
region and BC to be search region size in bytes+1.
CALLING SEQUENCE ... CALL EDLP1 registers as above
EFFECT ... See STRSCH
USES ... Table search

****

NAME ... FNDLCR
FUNCTION ... find first newline char to left
CALLING SEQUENCE ... RST 40 HL points to text
EFFECT ... Positions HL to the left of the first newline found left (above) its initial position. Uses A & HL.
USES ... Syntax analysis, text manipulation.

****

NAME ... FNDRCR
FUNCTION ... Find first newline char to right
CALLING SEQUENCE ... RST 48 HL points to current text position
EFFECT ... Positions HL to the right of the first newline char found right (below) its initial position. Uses A & HL.
USES ... See FNDLCR

****

NAME ... FRMSND
FUNCTION ... Transmit a display file to screen until FRAMES (if positive) becomes zero or a key is pressed.
CALLING SEQUENCE ... CALL FRMSND DFILE must contain address of a valid display file.
EFFECT ... Sends display file to screen; generates sync pulses; reads keyboard and performs debounce; blinks cursor; returns with carry set if key pressed, reset if FRAMES timeout. KEYBRD value lies in BC and STMEND, not HL.
USES ... Display

****

NAME ... GETFLD
FUNCTION ... Analyse a field down to a 16 bit value
CALLING SEQUENCE ... RST 16 HL points at or before field start
EFFECT ... HL advanced to field terminator; field converted to 2 bytes in DE and ELEM1. Zero flag set on return if no field found. Any argument recognisable to ASZMIC may be in the field. Uses all registers except BC,IX,IY,I.
USES ... Myriad

****

NAME ... GET2
FUNCTION ... Analyse up to 2 fields
CALLING SEQUENCE ... CALL GET2 HL points at or before start of fields.
EFFECT ... Uses GETFLD. Loads ARG1 and BC with first field value, ARG2 and DE with second field value. Zero flag set on return if less than 2 fields found before ; or /NL/ terminator. HL points after last field processed. I, IX, IY unaffected.
USES ... Syntax analysis

****

NAME ... IGNBLK
FUNCTION ... Advance HL register to point at a non-blank character
CALLING SEQUENCE ... RST 32 HL points at character string
EFFECT ... HL advanced until a non-zero byte pointed to. A contains (HL).
USES ... Syntax analysis

****

NAME ... KEYBRD
FUNCTION ... Scan keyboard matrix
CALLING SEQUENCE ... CALL KEYBRD
EFFECT ... Read the keyboard matrix into H (D5-D1 + shift as D0), plus 8 address line bits in L. Both H & L = :FF if no key pressed. Uses A, BC, DE, HL. HL also stored in STMEND. See ZX81 construction leaflet for key-address/data line connection.
USES ... Read keyboard; Initiate vertical sync pulse.

****

NAME ... KEYINT
FUNCTION ... Decode a key stroke
CALLING SEQUENCE ... CALL KEYINT BC contains HL pattern obtained from KEYBRD
EFFECT ... Both B & C must contain at least one zero bit if routine is to return. Carry set on return if unallowed multiple key depressions. Otherwise HL is absolute address of byte containing and A is offset from TABLE start (HL=TABLE-1+(A)). Note that the shifted keys A-G, Q-T, 1-0 do not return a valid character.
USES ... Keyboard read interpretation

****

NAME ... LIX
FUNCTION ... Return point for all handlers in ASZMIC
CALLING SEQUENCE ... £
EFFECT ... £
USES ... £

****

NAME ... MSKINT
FUNCTION ... Set up some aspects of ASZMIC context
CALLING SEQUENCE ... CALL MSKINT
EFFECT ... Loads I=14,IY=:4000, resets bit 0 of MFLAG, sets interrupt mode 1. Uses A register.
USES ... Programs entered by J command may need this if they use ASZMIC routines.

****

NAME ... OUTFRM
FUNCTION ... Transmit a single frame to screen
CALLING SEQUENCE ... CALL OUTFRM valid DFILE content required. B should contain the number of lines to be written and C should contain the number of blank rasters at screen top.
EFFECT ... Display a frame
USES ... Display

****

NAME ... OFRM1
FUNCTION ... Special purpose OUTFRM
CALLING SEQUENCE ... CALL OFRM1 like OUTFRM but in addition D register must be loaded with no of rasters per line.
EFFECT ... £
USES ... Graphics

****

NAME ... OFRM2
FUNCTION ... Specialised display
CALLING SEQUENCE ... CALL OFRM like OFRM2 but in addition user is responsible for the OUT instruction to clear the vertical sync pulse and to load A with the number of M1's before the first horizontal sync pulse is required.
EFFECT ... £
USES ... Sophisticated graphics

****

NAME ... PRCLR
FUNCTION ... Clear printer buffer to zeroes
CALLING SEQUENCE ... CALL PRCLR
EFFECT ... Blanks out PRBUFF. Uses HL,BC,DE.
USES ... User abuse of PRBUFF

****

NAME ... PRNTER
FUNCTION ... Print a line on Sinclair printer
CALLING SEQUENCE ... CALL PRNTER HL points at start of line to be written. Line is written out to printer until a /NL/ is encountered. On return HL points to char after the terminating /NL/. Uses AF, BC,DE,HL.
EFFECT ... £
USES ... Printing

****

NAME ... PUTDE
FUNCTION ... Hexadecimal encode
CALLING SEQUENCE ... CALL PUTDE HL points to output region, DE contains the number to be encoded into the region.
EFFECT ... Uses WRITA to encode the contents of the DE register as four hexadecimal digits starting at (HL). HL is incremented past the last digit. PUTDEF is an entry point which presets HL to PRBUFF+1. Uses HL, A.
USES ... Output routines

****

NAME ... RDCASS
FUNCTION ... Read a byte from cassette recorder
CALLING SEQUENCE ... CALL RDCASS
EFFECT ... Returns with a byte read from cassette in the A register. Can be aborted by Break key. Uses A,HL,BC,DE. You have about 800 microseconds to process the byte before RDCASS must be called again to catch the next byte.
USES ... Specialised tape analysis

****

NAME ... SHIFTS
FUNCTION ... £
CALLING SEQUENCE ... All Shift commands are called by the sequence CALL SHFT* where * is the shifted character and must have HL=(CURSOR) on entry. They perform action approprate to the shift and then return to caller.
EFFECT ... £
USES ... £

****

NAME ... START
FUNCTION ... Breakpoint or , if I=0, Restart
CALLING SEQUENCE ... RST 0
EFFECT ... Restart initialises ASZMIC. Breakpoint causes saving of context and return to ASZMIC
USES ... Monitor return

****

NAME ... STRSCH
FUNCTION ... String search
CALLING SEQUENCE ... CALL STRSCH HL points at first char of base string
EFFECT ... If (HL)<DSPBGN+40 then HL first loaded from CURSOR (Shift Macro use). Text area from (HL) to DSPBGN+50 searched to find a destination string which matches the base string. Uses CMPSTR so carry set if not found or set if found and HL & DE as for CMPSTR except that if not found HL points at DSPGN+39. Uses AF,BC,DE,HL. EDLPl is an entry point which expects DE to point to base string, HL to point to high end of search region and BC to be no of bytes to be searched+1.
USES ... Syntax analysis

****

NAME ... WRCASS
FUNCTION ... Write a byte to cassette recorder
CALLING SEQUENCE ... CALL WRCASS HL points to byte to be written out.
EFFECT ... Byte written out using Sinclair BASIC standard. On return HL has been incremented and A contains byte just written. Uses AF,BC,DE,HL.
USES ... Special cassette operations

****

NAME ... WRITA
FUNCTION ... Encode a byte
CALLING SEQUENCE ... CALL WRITA A contains number, HL points to encode region
EFFECT ... Number encoded as 2 hex digits. HL incremented past second
USES ... output

****

NAME ... WSTRNG
FUNCTION ... Write out PRBUFF to screen or printer
CALLING SEQUENCE ... Call WSTRNG PRBUFF contains at least 1 non blank char.
EFFECT ... Presets B to length of PRBUFF. Fills PRBUFF backwards with /NL/'s until a non-blank char found. Drops thru to WSTRG2. Uses A,B,HL
USES ... Output

****

NAME ... WSTRG2
FUNCTION ... See WSTRNG
CALLING SEQUENCE ... CALL WSTRG2 B preset to max no chars to write
EFFECT ... If bit 1 of ASSFLG is set then PRNTER invoked followed by PRCLR and return. Presets DE to PRBUFF. Drops thru to WSTR1.
USES ... £

****

NAME ... WSTR1
FUNCTION ... Write string to screen
CALLING SEQUENCE ... CALL WSTR1 DE points at string start. B preset to max no of chars to be written.
EFFECT ... Writes out from (DE) to screen, increments until B count exhausted or a /NL/ written. Drops thru to PRCLR. Uses all registers except IX,IY,I.
USES ... Output

****

EXAMPLES

Invoke by H command. Use = directives to declare the addresses of the undefined symbols as defined in Appendix 4. Use an ORG directive appropriate to your system memory size.

1 WRITE A CHAR TO SCREEN

```
START LD A,"X"
 CALL NRM2
 RET
```

2 WRITE OUT A STRING TO SCREEN

```
START LD DE,STRING
 LD B,7
 CALL WSTR1
 RET
STRING DEFM "ABCDEFG"
```

3 ADD 2 NUMBERS AND DISPLAY RESULT. CALL BY H START NO1 NO2

```
START RST 16 ;GETFLD
 PUSH DE
 RST 16 ;GET 2ND NO
 POP HL
 ADD HL,DE
 EX DE,HL
 CALL PUTDEF
 LD (HL),:76
 CALL WSTRNG
 RET
```

4 PALINDROME. INVOKE WITH H START. IMPORTANT BECAUSE IT SHOWS YOU HOW TO LINK IN YOUR OWN PROGRAM AS A COMMAND INTERPRETER, AND HOW TO RESTORE ASZMIC AT END. AFTER THE H START YOU TYPE IN THE LINE TO BE INVERTED.

```
START LD HL,HANDLE; ADDRESS OF USER C.I.
 CALL DFLIP; SWAP CI ADDRESSES
 JP LIX ;JUMP TO ASZMIC  CONTEXT STILL ON STACK
;
HANDLE RST 48; FNDRCR
 RST 40; FNDLCR
 SET 7,(IY) ; SET EDIT MODE
LOOP LD A,(HL)
 PUSH HL
 PUSH AF
 CALL NRM2
 POP AF
 POP HL; CHAR WRITTEN
 DEC HL
 CP :76 ;DID WE WRITE A /NL/?
 JR NZ,LOOP; LOOP IF NOT
 POP AF ;CLEAR LIX RETURN ON STACK
 POP HL ;ASZMIC CI ADDRESS STORED BY DFLIP
 LD (DADDR),HL; RESTORE ASZMIC CI ADDRESS
 RES 7,(IY) ;ONLY IF YOU WANT TO RETURN IN DEBUG
MODE
 RET
```

5 PRINT NUMBERS FROM 0 TO 6. USE H START

```
START XOR A
 LD HL,PRBUFF
LOOP LD B,A; SAVE A
 CALL WRITA
 LD A,B; RESTORE
 INC HL; SPACE
 INC A
 CP 7
 JR NZ,LOOP
 CALL WSTRNG
 RET
```

6 PRINT A LINE

```
START LD HL,TEXT
 CALL PRNTER
 RET
TEXT DEFM "SAMPLE TEXT"
 DEFB :76; /NL/
```

7 CREATE A BASIC PROGRAM WITH A SINGLE REM STATEMENT WHICH CONTAINS MACHINE CODE WHICH YOU CAN WRITE YOURSELF (SEE THE FINAL SECTION IN CHAPTER 5)

```
ORG :4000
DEFM "12345678"
DEFB "9"+128 ;FINAL TITLE CHAR INVERTED A LA ZX81
DEFB 0
DEFW 1
DEFW DFILE
DEFW DFILE+1
DEFW VARS
DEFW 0
DEFW VARS+1
ORG $+9
DEFW MEMBOT
DEFW 512
DEFW 0
DEFW :FDBF
DEFW :37FF
DEFW DFILE
ORG $+9
DEFW -1
DEFW 0
DEFW :218C
DEFW :4018
ORG $+32
DEFB :76
MEMBOT ORG $+32
 DEFW 256
 DEFW DFILE-PROG
PROG DEFB :EA ;REM
```

```
      ;
      ;
      ;  PUT HERE THE CODE WHICH YOU WANT TO BE INCLUDED
      IN THE REM
      ;
      ;
      DFILE=$+1
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
       DEFW :7676
      VARS DEFB 128

8  GRAPHICS PROGRAMS . .. ... SEE DESCRIPTION IN
CHAPTER 8
ENTER BY H :7000

      ;DEFINITIONS FOR E04 ASZMIC.....CHECK FOR YOUR
      VRSION
       ORG :7000
      INTJMP=:4019
      DFILE=:4023
      PCONE=:409A
      TXTLIM=:4025
      FRAMES=:4070
      OFRM1=:16D  :16B
      MFLAG=:4000
      RESTOR=:628
      KEYBRD=:145  :143
      STMEND=:400F
      SAVMEM=:18D  :18B
      NNN=46
      TOPS=1
      PIXSIZE=1
      IDLE=30
```

```
RASTERS=255
DISPEND=:6501
DSTART=:4100
;
; KERNEL ROUTINE
;
KERNEL LD HL,ONEP
 LD (INTJMP),HL ;INTERCEPT BREAK PROCESSING BY OWN
HANDLER
 CALL CLEAR ;SETUP DISPLAY
 LD HL,UPROG
 LD (PCONE),HL ;PRIME CONTEXT SAVE TO RETURN TO
UPROG
;
ONEP CALL KEYBRD ;READ KEYBOARD (RAW)
 LD HL,(FRAMES)
 INC HL
 LD (FRAMES),HL ;BUMP FRAME COUNT
 LD B,IDLE
 DJNZ $ ;YOU JUST KEEP ME HANGIN' ON
 LD D,PIXSIZE
 LD B,RASTERS
 LD C,TOPS
 CALL OFRM1 ;WRITE OUT A PART FRAME
 LD A,1
 LD (MFLAG),A ;LET NMI INTERRUPT HANDLER KNOW IT
MUST BREAK ON 0 COUNT
 LD C,NNN+NNN+1 ;SETUP C FOR G COMMAND
 EXX
 JP RESTOR ;GO INTO MIDDLE OF G HANDLING
;
;SETUP DISPLAY FILE,,,,COULD BE IMPROVED
;
CLEAR LD HL,DSTART
 LD DE,DSTART+2
 LD BC,:2400
 LD (HL),:76 ;/NL/
 LD (DFILE),HL
 INC HL
 LD (HL),0
 LDIR
 RET
```

```
;
; PLOT & UNPLOT SUBROUTINES.....Y->B     X->C
;
PLOT LD D,15
 JR $+4
UNPLOT LD D,0
 LD A,3 ;MASK FOR PIXEL NO IN BYTE
 AND C
 SRL C
 SRL C
 INC A ;C IS NOW X BYTE, NOT PIXEL
 LD E,16 ;BIT MASK
LOOP1 SRL E
 DEC A
 JR NZ,LOOP1 ;LOOP UNTIL 0-3 CONVERTED TO 8,4,2,1
IN E
 CP D ;UNPLOT
 JR NZ,ENT2 ;J IF NOT
 LD A,E
 CPL
 LD D,A ;MASK OUT BYTE
ENT2 PUSH DE
 LD HL,DISPEND ;PREPARE TO COMPUTE Y ADDRESS
 LD DE,:1200
 LD A,B
 LD B,8
LOOP2 RLCA
 JR NC,NOSUBT
 OR A
 SBC HL,DE ;SUBTRACT ONLY IF BINARY POWER PRESENT
IN Y VALUE
NOSUBT SRL D
 RR E ;SHIFT SUBTRACTOR
 DJNZ LOOP2
 ADD HL,BC ;ADD ON X TO GET TARGET BYTE
; NOW PROCESS THE BYTE TO PUT IN THE PIXEL
 POP DE
 LD A,(HL)
 BIT 7,A ;GET ROUND INVERSE TRICKERY
 JR Z,$+3
 CPL ;CONVERT 8,2,1,0 TO 3,2,1,0
 AND 15
 ;
```

```
 OR E
 AND D ;PIXEL NOW IN
; REENCODE
 CP 8
 JR C,NOTINV
;DO NOT NEED INVERSION
 CPL
 AND :87
NOTINV LD (HL),A
;
;FOLLOWING CODE JUST CAUSES SUBROUTINE TO LOOP
AROUND UNTIL
;A FRAME HAS BEEN SENT. IT IS NOT ESSENTIAL BUT
SLOWS PLOTTING
;DOWN A BIT
;
NLINE LD DE,FRAMES
 LD A,(DE)
 LD B,A
LOOPX LD A,(DE)
 CP B
 JR Z,LOOPX
;END OF DELAY CODE
 RET
;
;LINE OR UNLINE FROM XY TO X'Y'
;
; X->E  Y->D  X'->C  Y'->D
;
LINE LD A,15
 JR $+3
UNLINE XOR A
 LD (DORDEL),A ;MEMORY TO DETERMINE PLOT OR UNPLOT
 LD HL,XMID
;INIT CELLS
 XOR A
 LD (HL),A
 INC HL
 LD (HL),E
 INC HL
 LD (HL),A
 INC HL
 LD (HL),D
 INC HL
```

```
 LD (HL),C
 INC HL
 LD (HL),B
;
; NOW CALCULAT THE INCREMENTS
;
 LD L,C
 LD D,A
 LD H,A
 SBC HL,DE
 PUSH HL ;X'-X
 LD L,B
 LD H,A
 LD A,(YMID+1)
 LD E,A
 SBC HL,DE ;HL NOW Y'-Y
 POP DE ;LEFT JUSTIFY INCREMENTS TILL ONE >=
128/256
JUSTIFY LD C,L
 LD A,H
 RLCA
 RL L
 ADC A,0
 EX AF,AF'
 LD B,E
 LD A,D
 RLCA
 RL E
 ADC A,0
 JR NZ,JSTDUN
 EX AF,AF'
 JR Z,JUSTIFY
JSTDUN LD E,B
 LD B,H
;
; DE IS XINC BC IS YINC
;
LLOOP LD HL,(XMID)
 ADD HL,DE ;INCREMENT
 LD (XMID),HL
 LD HL,(YMID)
 ADD HL,BC
 LD (YMID),HL
 EXX ;PRESERVE INCREMENTS
LPLOT LD A,(XMID+1)
```

```
 LD C,A ;X
 LD A,(YMID+1)
 LD B,A ;Y
 LD A,(DORDEL)
 LD D,A
 CALL UNPLOT+2  ;PLOT OR UNPLOT THE NEW POINT
 LD HL,XPRIM
 LD A,(XMID+1)
 CP (HL)
 INC HL
 EX AF,AF'
 LD A,(YMID+1)
 CP (HL)
 EXX
 JR NZ,NYLIM ;HAVE NOT REACHED Y LIMIT
 LD BC,0 ;ZERO YINC IF AT Y LIMIT
NYLIM EX AF,AF'
 JR NZ,NXLIM
 LD DE, 0 ;ZERO XINC IF A X LIMIT
NXLIM LD A,D ;USE XINC + YINC BOTH 0 AS END OF
LINE TEST
 OR E
 OR B
 OR C
 JR NZ,LLOOP
 RET
;
;DATA REGION
;
XMID DEFW 0 ;ORDER IS IMPORTANT
YMID DEFW 0
XPRIM DEFB 0
YPRIM DEFB 0
DORDEL DEFB 0
;
```

```
********

 MOIRE

********


UPROG LD SP,:7F00
MAINLOOP LD DE,:4078
 LD BC,:101
MLOOP PUSH DE
 PUSH BC
 CALL LINE
 POP BC
 POP DE
 INC D
 INC D
 INC B
 INC B
 INC B
 INC B
 LD A,250
 CP B
 JR C,OUTCOD
 LD A,(STMEND)
 AND 1
 JR NZ,MLOOP
;ZXCV PRESSED HERE
 LD HL,:8000 ;FIRST LOC OF UNAVAILABLE MEMORY
 JP SAVMEM
OUTCOD CALL CLEAR
 JP MAINLOOP
```

```
************

 STRUCTURES

************


UPROG LD SP,:7F00
 JR OUTCOD
MAINLOOP LD IX,XVAR
 CALL PROSUB
 LD IX,YVAR
 CALL PROSUB
 LD B,(IX+VAR)
 LD A,(XVAR)
 LD C,A
 CALL PLOT
 LD A,(STMEND)
 AND 1
 JR Z,OUTCOD
 JR MAINLOOP
;
OUTCOD LD HL,DUMMY
 LD DE,XVAR
 LD BC,12
 LDIR ;INIT VARIABLE REGION
 CALL CLEAR
 LD HL,(FRAMES)
 LD A,(HL)
 AND 15
 INC A
 LD (XVAR+1),A
 INC HL
 LD A,(HL)
 AND 7
 INC A
 LD (YVAR+1),A
 JR MAINLOOP
```

```
;
PROSUB LD A,(IX+VAR)
 ADD A,(IX+DIR)
 LD (IX+VAR),A
 CP (IX+MAX)
 JR NC,OUTCOD
 CP (IX+LOW)
 JR NC,POSCHEK
 LD A,(IX+INC)
 CPL
 ADD A,(IX+LOW)
 LD (IX+LOW),A
 CALL DIDLIM
POSCHEK LD A,(IX+VAR)
 CP (IX+HIGH)
 RET C
 LD A,(IX+INC)
 ADD A,(IX+HIGH)
 LD (IX+HIGH),A
;
DIDLIM LD A,(IX+DIR)
 NEG
 LD (IX+DIR),A
 LD A,(IX+INC)
;
 DEC A
 JR NZ,$+3
 INC A
;
 LD (IX+INC),A
 RET
;
;DATA
;
VAR=0
INC=1
DIR=2
LOW=3
HIGH=4
```

```
MAX=5
;
XVAR ORG $+6
YVAR ORG $+6
DUMMY DEFB 75
 DEFB 8
 DEFB -1
 DEFB 70
 DEFB 90
 DEFB 135
 DEFB 110
 DEFB 12
 DEFB 1
 DEFB 90
 DEFB 130
 DEFB 240
 DEFB 0
```

E07 ASZMIC TUNED FOR USE IN UNITED STATES OF AMERICA

This version of ASZMIC has been produced with assembly parameters for use in the USA on 60 Hz 525 line domestic televisions. We regret that this means that the number of lines displayed on screen has had to be reduced to 28. If you wish to use the graphics then the parameters in the kernel routine will need some modification to give a 60 Hz frame rate. The product of PIXSIZE & RASTERS summed with TOPS & NNN will need to be about 244. In the example given on pp 111-112 if rasters is decreased to 205; 1800 decimal is subtracted from DISPEND (i.e. 50 lines of 36 bytes) and from the value loaded into BC in the CLEAR; then everything should synchronize nicely. If you use the UPROG's given then you may need to trim some of the end values to keep points on screen; or better still put limit checking in PLOT at the very beginning.

You may find discrepancies in the manual, which was written for European 625 line ASZMIC, & even in the functions of ASZMIC although , frankly, we doubt it; but we normally thrash our software for several months before releasing it and the mention in 'BYTE' forced us to turn on the USA assembly parameters sooner than we had intended in order to meet the very positive response from enthusiasts in America.

We took the opportunity to clear up all outstanding "bugs" (both of them) and in consequence some portions of code do not have the addresses given in Appendix 4 (p.85). Please use the system address list below instead. COMPROCSYS will be producing more exciting add-ons for ZX81 in the near future.