Small Computer Monitor

Tutorial

Monitor version 0.5 for the Z80 CPU

Software and Documentation by Stephen C Cousins

Edition 0.5.0

CONTENTS

OVERVIEW 3
PREREQUISITES 4
Decimal numbers *4*
Binary numbers *4*
Hexadecimal numbers *5*
Why not just use decimal numbers? *5*
Bits, Bytes, Nibbles and Words *6*
Avoiding confusion *6*
ASCII characters *8*
GETTING CONNECTED 8
Serial modules *9*
Serial cables *9*
TERMINAL PROGRAMS 10
Tera Term *10*
Putty *10*
HyperTerminal *11*
INSTALLING THE SMALL COMPUTER MONITOR 12
RC2014 original 8k ROM board *12*
RC2014 switchable ROM board *12*
RC2014 pageable ROM board *12*
RC2014 Mini board *13*
SWITCH ON 14
FIRST COMMANDS 15
? or Help *15*
Reset *15*
Memory display *15*
Registers display or edit *16*
Input from port *17*
Output to port *17*
WRITING PROGRAMS 18
High Level Language *18*
Machine Code *19*
Assembly Language *19*
YOUR FIRST PROGRAM 20
Using the assembler *21*
Debugging *22*
Single Step *25*
HELLO WORLD! 27
A better world *31*
REGISTERS AND FLAGS 33

MORE EXAMPLE PROGRAMS....34
Displaying a new line....35
Input from keyboard....36
Flashing an LED....36
Better flashing!....38
Yet more fun with LEDs....42
ASSEMBLERS....45
FAULT FINDING....46
PARTS AND SUPPLIERS....47
RC2014 official modules....47
Chip programmer....47
FTDI cable....47
FTDI 'cable'....47
USB-RS232 cable....47
EEPROM 8k x 8 bit....48
CONTACT INFORMATION....49

Overview

The Small Computer Monitor is a classic machine code monitor enabling debugging of programs and general tinkering with hardware and software. It can also act as a boot ROM, so no other software is required on the target computer system.

This tutorial is designed for people with little, or no experience of machine code monitors and Z80 programming. If you have already had experience of these things the Small Computer Monitor User Guide should be all the documentation you need.

This tutorial assumes you are using an RC2014-Z80 system and that the system is up and running with the supplied BASIC ROM. You should therefore have the RC2014 connected to a terminal or a PC (or similar) running terminal emulation software.

The first thing to do is fit a ROM containing the Small Computer Monitor program, and configure the hardware to boot up from this ROM.

After that you can enter and run machine code programs, test and debug programs, and generally experiment with the hardware.

This tutorial concentrates on the use of the Small Computer Monitor, rather aiming to be a definitive guide to Z80 programming. However, to use the monitor requires some understanding of Z80 programming. Therefore simple programming is explained in this document. This should be enough to get you started.

Once the Small Computer Monitor and the basics of Z80 programming are understood there are plenty of suitable resources available on the internet which can help with further progress.

Prerequisites

Before diving in to using the monitor program there are a few things you need to be familiar with. These are:

- Binary numbers
- Hexadecimal numbers
- ASCII characters

The next few pages give a brief introduction to these topics, but if you are in any doubt please seek further information before continuing.

Decimal numbers

The decimal numbers we use in day to day life are expressed in base ten (or denary). This means we use ten distinct symbols (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9).

To express numbers greater than 9 we use more than one of these digits. Using two digits allows us to represent 100 different numbers (0, 1, 2, … , 97, 98, 99).

The least significant digit is on the right, with the digit to its left representing ten times (base ten remember) the value of the digit to its right. And so on.

Thus decimal number 12 means (1 x 10) + (2 x 1), and the decimal number 123 means (1 x 1000) + (2 x 10) + (3 x 1).

Binary numbers

Binary numbers follow the same pattern as decimal numbers but are in base two. This means we use two distinct symbols (0 and 1).

To express numbers greater than decimal 1 we use more than one of these digits. Using two digits allows us to represent 4 different numbers (0, 1, 2 and 3)

The least significant digit is on the right, with the digit to its left representing two times (base two remember) the value of the digit to its right. And so on.

Thus binary number 10 means decimal (1 x 2) + (0 x 1) = 2, and the binary number 101 means decimal (1 x 4) + (0 x 2) + (1 x 1) = 5.

Hexadecimal numbers

Hexadecimal numbers (or Hex for short) follow the same pattern as decimal numbers but are in base sixteen. This means we use sixteen distinct symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F).

The letters A to F represent decimal numbers 10 to 15 respectively. So A=10, B=11, C=12, D=13, E=14 and F=15.

To express numbers greater than decimal 15 we use more than one of these digits. Using two digits allows us to represent 256 different numbers (0, 1, 2, … , 253, 254, 255).

The least significant digit is on the right, with the digit to its left representing sixteen times (base sixteen remember) the value of the digit to its right. And so on.

Thus hexadecimal number 1F means decimal (1 x 16) + (15 x 1) = 31, and the decimal number 1F2 means (1 x 256) + (15 x 16) + (2 x 1) = 498.

Why not just use decimal numbers?

The reason all this matters is that our normal base ten decimal numbering system does not fit well with digital computers.

Computers are made up of electronics which is digital. Being digital means that each of the smallest components that make up the computer can only be in one of two states, on or off (or one or zero).

Binary notation is thus the most basic way to describe the state of any part, or group of parts, in a computer.

Eight bit microprocessors, such as the Z80, process eight bits at once. To describe any combination of these eight bits requires eight binary digits. This gives the range of binary numbers from 00000000 to 11111111 (or decimal 0 to 255).

Now us mere humans have a bit of trouble working with numbers like 11010111, so we need a more convenient notation. This is where hexadecimal comes in.

A hexadecimal digit has sixteen possible values, which happens to be the same as four binary bits. Thus the following are equivalent:

- Binary 0000 = Hexadecimal 0
- Binary 0001 = Hexadecimal 1
- Binary 0010 = Hexadecimal 2
- Binary 0011 = Hexadecimal 3

- Binary 0100 = Hexadecimal 4
- Binary 0101 = Hexadecimal 5
- Binary 0110 = Hexadecimal 6
- Binary 0111 = Hexadecimal 7
- Binary 1000 = Hexadecimal 8
- Binary 1001 = Hexadecimal 9
- Binary 1010 = Hexadecimal A
- Binary 1011 = Hexadecimal B
- Binary 1100 = Hexadecimal C
- Binary 1101 = Hexadecimal D
- Binary 1110 = Hexadecimal E
- Binary 1111 = Hexadecimal F

By using two hexadecimal digits we can describe all 8 binary digits. This allows us to express the above example (binary 11010111) as hexadecimal D7. It is much easier for us to handle "D7" then "11010111", and thus when programming at the 'binary' level we tend to use hexadecimal notation.

So why not decimal?
Because a single hexadecimal digit can exactly represent four binary digits, and two hexadecimal digits can exactly represent eight binary digits, there is a simple relationship between the binary and hexadecimal. This makes hexadecimal numbers a natural choice when working at the lowest level where we are concerned about what individual digital bits are doing. The same relationship does not apply to decimal notation, where a four binary digits requires either one or two decimal digits depending on the value of the binary digits. This just tends to be messy and inconvenient.

Bits, Bytes, Nibbles and Words

With all these different quantities being used regularly, they had to have names. A Bit is a single digital quantity, which can be described in binary by one digit (0 or 1). A Nibble is a group of four binary digits, which can be represented by one hexadecimal digit (0 to F). A byte is a group of eight binary digits, which can be represented by two hexadecimal digits. A word is a group of sixteen binary digits, which can be represented by four hexadecimal digits.

The Z80 processor handles 8 data bits at once, which is a byte, and uses a 16 bit memory address, which is a word.

Avoiding confusion

You may have noticed above that most times a number was mentioned it was necessary to prefix is with the word decimal, binary or hexadecimal. Without doing

that how do you know if the number “10” is decimal 10, binary 10 (decimal 2) or hexadecimal 10 (decimal 16). In short, you don’t. You can often guess based on context but you can’t be sure.

To avoid this confusion and to avoid continually having to write “decimal”, “binary” or “hexadecimal” before every number, various notation systems have been used in the computer industry. Unfortunately not everyone uses the same system!

The various systems use either a prefix or postfix for identification of the number base. For example, putting “$” before a number (eg. $10) indicates the number is expressed in hexadecimal.

Some commonly used identifiers are:

Identifier	Base	Example
$	Hexadecimal	$1FC
0x	Hexadecimal	0x1FC
H	Hexadecimal	1FCH
0b	Binary	0b101
%	Binary	%101
B	Binary	101B
+	Decimal	+123
D	Decimal	123D

As the Small Computer Monitor is mainly used to directly manipulate memory, registers and ports, which are digital in nature, the default for all numbers is hexadecimal. Thus hexadecimal numbers don’t usually require special identifiers, but there are some exceptions.

The disassembler shows constants as hexadecimal numbers prefixed by a “$”. The prefix is necessary here for clarity because the processor’s registers B, C, D and E are also valid hexadecimal numbers.

The assembler accepts hexadecimal numbers without any identifiers. To specify a decimal number it must be prefixed with “+”. Hexadecimal numbers prefixed with “$” or “0x” are also accepted.

Within this guide the ‘$’ prefix is normally used where clarity is required.

ASCII characters

Within digital computers all the characters of the alphabet and all other symbols are stored as numbers. The character allocation standard used by the Small Computer Monitor is the American Standard Code for Information Interchange (ASCII).

The tables below show the subset of ASCII used by the Small Computer Monitor.

ASCII	Hex.	Char.	ASCII	Hex.	Char.	ASCII	Hex.	Char.	ASCII	Hex.	Char.
0	00	NUL	16	10		32	20	(spc)	48	30	0
1	01		17	11		33	21	!	49	31	1
2	02		18	12		34	22	“	50	32	2
3	03		19	13		35	23	#	51	33	3
4	04		20	14		36	24	$	52	34	4
5	05		21	15		37	25	%	53	35	5
6	06		22	16		38	26	&	54	36	6
7	07		23	17		39	27	‘	55	37	7
8	08	BS	24	18		40	28	(	56	38	8
9	09		25	19		41	29	)	57	39	9
10	0A	LF	26	1A		42	2A	*	58	3A	:
11	0B		27	1B	ESC	43	2B	ES+C	59	3B	;
12	0C		28	1C		44	2C	,	60	3C	<
13	0D	CR	29	1D		45	2D	-	61	3D	=
14	0E		30	1E		46	2E	.	62	3E	>
15	0F		31	1F		47	2F	/	63	3F	?

ASCII	Hex.	Char.	ASCII	Hex.	Char.	ASCII	Hex.	Char.	ASCII	Hex.	Char.
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

So to refer to the letter ‘A’ when working in the monitor, it is usually necessary to use the decimal value 65 or hexadecimal value 41.

Getting Connected

If you already have your RC2014-Z80 system up and running with the supplied BASIC ROM then you can skip this section.

The RC2014 needs to be connected to a terminal or computer running terminal emulation software. The terminal provides the keyboard input and the displayed output.

The physical connection is a serial cable from the RC2014 to the terminal.

Serial modules

The RC2014 serial port can either be the official Serial I/O module, the official Dual serial module (SIO/2), the RC2014 Mini's serial port or unofficial SIO/2 modules following Grant Searle's register addressing (eg. Dr Scott M Baker's SIO/2 module) . There are other modules available which might also work.

The Serial I/O module and the RC2014 Mini use a MC68B50 or compatible Asynchronous Communications Interface Adapter (ACIA) chip. The Serial I/O module has the option of either a 5 volt FTDI style serial interface or a 9-pin D-type RS232 serial interface, while the Mini only has the 5 volt FTDI style interface.

The Dual serial module uses a Zilog Z80 SIO/2 chip and provides two serial ports. The Small Computer Monitor only requires port A. This module only provides the 5 volt FTDI style serial interface, not the traditional RS232 serial interface.

Serial cables

To use the RS232 style interface you need a null modem lead to connect the 9-pin D-type connector on the RC2014 board to that on the terminal (or computer).

To use the 5 volt FTDI style interface you need an FTDI (or compatible) USB to serial cable. There are a variety of designs available. Note, there have been problems experienced by some RC2014 users with some cables with some values of resistors on the RC2014 boards.

See the section "Parts and Suppliers" for details of known compatible 'cables'.

Terminal programs

Unless you happen to be using a genuine computer terminal you will need to install a terminal emulation program on a PC or similar.

Several freely available terminal emulation programs are detailed below, but in general these are the required settings:

The port number will be determined by the hardware on your computer. Serial ports are normally called “COM#”, where “#” is a number like 3 (eg. COM3). When you plug in a suitable USB to Serial cable a COM port will become available on the computer. If your computer has a legacy RS232 port it will also have a COM port number.

Serial port settings for the RC2014 when running the Small Computer Monitor are normally:

- Baud rate = 115200
- Date bits = 8
- Parity = None
- Stop bits = 1
- Flow control = None
- New line character(s) = Carriage Return
- Local echo = Off
- Backspace key sends = Control+H
- Terminal type = VT100 (not critical as only a basic terminal is required)
- Transmission delay(s) = 0

Terminal software usually has options to add delays to characters sent from the terminal. These can usually be set to zero, but if you experience any loss of characters you can add a small delay.

Tera Term

Unless you already have a preference, I’d suggest using Tera Term.

For Tera Term v4.96, use the Setup menu, Terminal item and Serial ports item to configure as described above. Leave Answerback blank and Auto switch off.

Putty

Putty is a very capable program but has lots of complex options.

For Putty v0.70, use the Configuration dialog box that opens when the program loads.

On the Session page select the Connection Type = Serial.

Select the Category = Serial. Enter the Serial line to connect to = COM# (where # is the number of your serial port), speed = 115200, data bits = 8, stop bits = 1, parity = none, flow control = none.

Use the Terminal page to ensure “Impicit CR in every LF” is cleared, along with “Implicit LF with every CR”. Also local echo and local line editing are Off.

Use the Keyboard page to set Backspace to Control+H.

Return to the Session page, Select Default Settings and click Save. Then click Open.

Once the terminal window opens you can return to the configuration options by clicking on the terminal icon at the top left of the window and selecting Change Settings.

HyperTerminal

For users of older Windows installations, such as XP, you probably already have HyperTerminal installed. For HyperTerminal v5.1, use the File menu, Properties item to configure the settings as detailed above.

Installing the Small Computer Monitor

It is possible to run the Small Computer Monitor by downloading it into RAM, but here it is assumed you are running it from a Read Only Memory (ROM) chip.

If you need to program your own chip you will find the program supplied as an Intel Hex File suitable for opening with most chip programming products.

The programming method is dependent on the programmer you are using and is beyond the scope of this tutorial. See the section "Parts and Suppliers" for details of known compatible programmers.

The Small Computer Monitor can be either in a ROM chip with 8k bytes by 8 bits of memory (often described as 64k bit memory) or in part of a larger chip. Which every chip is used the Monitor program must be located in the Z80's memory map starting at address $0000.

A suitable chip is the AT28C64B-15PU. This is a very convenient chip if you want to reprogram it again later as it is an Electrically Erasable Programmable Read Only Memory (EEPROM). It can be erased and reprogrammed in just a few seconds with a modern low cost programmer like the MiniPro TL866.

The exact fitting instructions depend on the circuit board it is plugged in to. The most common RC2014 boards are detailed below.

RC2014 original 8k ROM board

This ROM board only accepts 8k byte ROM chips and has no links to configure. So plug in the programmed chip and off you go.

RC2014 switchable ROM board

This ROM board can accept a range of chip capacities and has links to select how the ROM is addressed. If you are using an 8k byte chip such as the AT28C64B-15PU the links should be set as follows:

Page selection: A13, A14 and A15 link = Vcc (5 volts)

For larger chips, like the 27C512, links A13 to A15 need to be set to match where in the chip the monitor code resides.

RC2014 pageable ROM board

This is quite a flexible board and has a number of links which need to be set. If you are using an 8k byte chip such as the AT28C64B-15PU the links should be set as follows:

Page size: 8k (links as indicated by the PCB legend)

Page selection: A10, A11 and A12 = no link, A13, A14 and A15 link = 1

For larger chips, like the 27C512:

Page size: 8k (links as indicated by the PCB legend)

Page selection: A10, A11 and A12 = no link, A13, A14 and A15 need to be set to match where in the chip the monitor code resides.

RC2014 Mini board

This ROM board can accept a range of chip capacities and has links to select how the ROM is addressed. If you are using an 8k byte chip such as the AT28C64B-15PU the links should be set as follows:

Page selection: A13, A14 and A15 link = Vcc (5 volts)

For larger chips, like the 27C512:

Page selection: A13, A14 and A15 link = need to be set to match where in the chip the monitor code resides.

Switch on

If all is well, when you switch on your RC2014 system, you should be greeted with something like this on your terminal screen:

```
Small Computer Monitor
*
```

If you do not get a sign on message similar to the above, then consult the “Fault Finding” section which can be found towards the end of this document.

The “*” character displayed on the terminal is the Monitor Prompt. This indicates the monitor is ready to accept a new command. Try typing the question mark character and then pressing the return key.

In the following text, user input is in a Bold Italic font, while the results are shown in a Regular font. Special key presses, such as Escape, are shown enclosed in curly brackets. Thus the example below means the user types “b 5000” on the terminal’s keyboard followed by a press of the Return key, and the monitor displays “Breakpoint set” on the terminal’s screen.

```
b 5000 {return}
Breakpoint set
```

Unless otherwise stated, parameters are hexadecimal numbers, such as FF12. There is no need to prefix them with a hexadecimal identifier or a numeric character. The exception to this rule is operands in the assembler.

Named parameters are shown, in this document, enclosed by “<” and “>”, and further enclosed by “[“ and “]” if the parameter is optional.

Command names and any parameters are delimited by a space character.

Monitor commands are not case sensitive, so can be typed in either upper or lower case, or any combination of upper and lower case.

First Commands

Below are some of the simple monitor commands to try.

? or Help

Syntax: HELP
Or syntax: ?

This displays a list of the monitor commands together with their syntax.

For example:

```
help {return}
Small Computer Monitor by Stephen C Cousins (www.scc.me.uk)
Version 0.4.0 configuration A for Z80 based RC2014 systems

Monitor commands:
A [<address>]   = Assemble instructions
B [<address>]   = Breakpoint set or clear
D [<address>]   = Disassemble instructions
E [<address>]   = Edit memory
F [<name>]     = Flags display or modify
G [<address>]   = Go to program
I <port>       = Input from port
M [<address>]   = Memory display
O <port> <data> = Output to port
R [<name>]     = Registers display or edit
S [<address>]   = Step one instruction
Also: DEVICES, HELP, RESET
API <function> [<A>] [<DE>]
CONSOLE <device number>
FILL <start> <end> <byte>
```

Reset

Syntax: Reset

This command performs a software reset, similar to pressing the reset button.

It can not perform a physical hardware reset on the electronics, but it does run the same software as a hardware reset.

Memory display

Syntax: M [<memory address>]

A block of memory is displayed, with each line showing the memory address in hexadecimal, the contents of sixteen memory locations in hexadecimal, and the contents of those sixteen memory locations in ASCII. Non-printable ASCII characters are shown as dots.

The memory address parameter is optional. If supplied the memory will be displayed starting at the specified address. If not, the memory display starts from the last address referenced.

For example:

```
m 1600 {return}
1600:  7C FF CB CE 21 15 16 CD  93 15 20 08 21 7C FF CB  |...!..... .!|..
1610:  C6 21 1B 16 C9 DD 15 E6  15 F1 15 A9 15 B2 15 BD  .!..............
1620:  15 11 27 16 C3 34 02 5A  38 30 20 62 61 73 65 64  ..'..4.Z80 based
1630:  20 52 43 32 30 31 34 20  73 79 73 74 65 6D 73 05   RC2014 systems.
1640:  00 21 7C FF 11 55 16 CB  46 C4 34 02 11 62 16 CB  .!|..U..F.4..b..
1650:  4E C4 34 02 C9 53 65 72  69 61 6C 20 41 43 49 41  N.4..Serial ACIA
1660:  05 00 53 65 72 69 61 6C  20 53 49 4F 2F 32 05 00  ..Serial SIO/2..
1670:  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ............….
```

Pressing {return} again will display the next block of memory.

Pressing {escape} will exit the memory display mode and return to the monitor prompt.

Alternatively a new command can be entered without first returning to the monitor prompt.

Registers display or edit

Syntax: R [<name of register>]

This command can either display the current processor registers or edit the value of a processor register.

When no parameter is entered the current register values are displayed.

For example:

```
r {return}
PC:0001 AF:0002 BC:0003 DE:0004 HL:0005 IX:0006 IY:0007 Flags:------N-
SP:0011 AF'0012 BC'0013 DE'0014 HL'0015 (S)0016 IR:0017 Flags'---H--N-
```

More on registers later.

Input from port

Syntax: I <port address>

The specified input port address is read and the result displayed in hexadecimal.

For example:

```
I F0 {return}
00
```

If you have the RC2014 digital I/O module you can typically read the switch inputs with the command:

```
I 0 {return}
```

Press one of the buttons on the I/O module and keep it held whilst pressing the Return key at the end of the input command. If you you are holding button '1' then you should see this:

```
I 0 {return}
01
```

Output to port

Syntax: O <port address> <data byte>

The specified data byte is written to the specified output port address.

For example:

```
O F0 55 {return}
```

If you have the RC2014 digital I/O module you can typically write to the LED outputs with the command:

```
O 0 <output byte> {return}
```

For example, this command will turn on the LEDs labelled 0 and 2.

```
O 0 5 {return}
```

The value 5 is written to the LED output latch. In binary the number 5 is 00000101, thus bits 0 and 2 are ON. This results in LED 0 and LED 2 lighting up.

Writing Programs

Programs are made up of instructions. At the lowest level, the Z80 microprocessor only understands instructions, or machine code, which are a series of binary bits. These bits are divided into bytes, with a single instruction being between one and four bytes long. Each byte can be described as a decimal number from 0 to 255, or a hexadecimal number from 00 to FF. So a program is just a list of numbers. Not too friendly really.

So what numbers make up a program?

To answer this question you first need to decide what the program is going to do. This should be something you could write down and read out to anyone so that they can understand what it does.

The next step is to break down the task into simple steps, such as a numbered list of things to do or a flow chart. Again this should be in everyday language you can easily get anyone else to understand.

Each step of the task can then be converted into instructions the computer understands.

High Level Language

You can use high level computer programming languages, where the instructions are quite like normal written languages. This makes it relatively easy to convert your design into instructions the computer understands.

On classic 8-bit computers, high level language usually meant Beginner's All-purpose Symbolic Instruction Code (BASIC). Given the limitations of those early computers BASIC was a good solution. Many people knock classic BASIC but with only a few thousand bytes of memory and perhaps only cassette recorder for storage, it was not easy to provide a more desirable programming language.

Machine Code

This document is a tutorial for the Small Computer Monitor, so we are talking about Machine Code, not high level programming languages. Machine Code is fast and efficient, but not as easy to write as high level language code.

You can use the Small Computer Monitor to write binary Machine Code instructions directly to the computer's memory. This is proper hard-core Machine Code. But there are slightly easier ways to do it.

To avoid working directly in binary we have already established that hexadecimal numbers are easier for people to use, but it is still only for crazy people. Instead the instructions are usual written in Assembly Language.

Assembly Language

Assembly Language is a way of writing Machine Code instructions using a series of Mnemonics and Operands which are much easier to read, write and remember than hexadecimal numbers. These Mnemonics and Operands are converted to Machine Code by an assembler.

The Small Computer Monitor can convert Assembly Language instructions into Machine Code, but only one instruction as a time. To write large programs it is desirable to use a program called an Assembler to compile the Machine Code.

Your First Program

It is a tradition in the programming world to make your first program one that prints "Hello World!", but here we start with something simpler.

Lets just start by trying to write the character "!" to the terminal screen.

As explained in the previous section, the first thing to do is define the task. Well we did that in the line above.

Next you should break down the task onto easy steps. In a larger program you'd be lucky to even see a step mentioned which is as small as this whole task, but as this is a tutorial I'll go full Rambo on the design and break it right down.

These are the steps required to write a letter to the terminal screen:

1. Store the character's ASCII value in a variable (a processor register)
2. Call the system function which writes a character to the terminal
3. Finish the program

Step 1

The ASCII value for the character "!" is decimal 33 or hexadecimal 21. The processor has a number of special variables called registers. To prepare a character for display the character's ASCII value must be stored in the A register. The assembly language instruction to do this is:

```
LD   A, 21
```

This instruction can be read as "Load the A register with the value hexadecimal 21". Note that the assembler defaults to the operand (21) being in hexadecimal.

Step 2

The Small Computer Monitor provides a set of functions to help programs perform common tasks. This is called an Application Programming Interface (API). To use the API to display a character you need to store the value 2 in register C and then call the API. The assembly language instructions to do this are:

```
LD   C, 2
CALL 30
```

The API subroutine starts at address hexadecimal 30. The API function number to output a character to the terminal is 2.

Step 3

To finish a program you use the Return instruction:

```
RET
```

The whole program written in assembly language is:

```
LD   A, 21
```

```
LD   C,2
CALL 30
RET
```

The assembler converts these assembly language instructions into binary machine code instructions. Expressed in hexadecimal this is:

```
3E 21 0E 02 CD 30 00 C9
```

The conversion process is called assembling. When assembled into memory starting at address hexadecimal 8000, the assembler output listing will usually look something like this:

```
8000: 3E 21      LD   A, 21
8002: 0E 02      LD   C, 2
8004: CD 30 00   CALL  30
8007: C9         RET
```

Using the assembler

To enter this program into memory using the Small Computer Monitor you use the Assemble command (A). To assemble the machine code to address hexadecimal 8000 you type the command:

A 8000 {return}

The monitor then shows the current instruction at address 8000, which looks something like this:

```
8000: 00           .    NOP                >
```

In this case the instruction currently in memory at address 8000 is NOP, but it could be anything at this point. You can then type the instruction you want to assemble there:

```
8000: 00           .    NOP                > LD A,21 {return}
```

The monitor program then displays the instruction you entered in the same list format, followed by the next instruction currently found in memory:

```
8000: 3E 21        >!   LD A,$21
8002: 00           .    NOP                >
```

You can then enter the next instruction: LD C,2

Continue like this until the whole program has been entered.

To exit the assembler press the Escape key.

You can check the whole program is correct by listing the program with the Disassemble command (D):

```
D 8000 {return}
```

The following should then be displayed:

```
8000: 3E 21         >!   LD A,$21
8002: 0E 02         ..   LD C,$02
8004: CD 30 00      .0.  CALL $0030
8007: C9            .    RET
```

Press the Escape key to end the program listing, or the Return key to see the next few instructions.

The characters after the machine code bytes are the ASCII characters for each of those bytes. Thus the first instruction 3E 21, shows ">!".

Now we have the machine code program in memory we can run it. For this we use the Go command (G):

```
G 8000 {return}
!*
```

The output, illustrated above, is the "!" character the program displays followed by the monitor prompt character "*".

Debugging

Hopefully the simple program above just worked without any drama. But what if it didn't?

Sometimes you can just look at a program for ages and not see why it failed. In such cases you need to use the debugging tools built in to the Small Computer Monitor.

There are two tools available to you:

- Breakpoint
- Single step

A breakpoint enables the program to be stopped at a specified address and the state of the processor displayed. Using the above program as an example, try this:

```
B 8004 {return}
Breakpoint set
```

G 8000 {return}

You should now see something like this:

```
Breakpoint
PC:8004 AF:21B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
```

The above indicates that the program has stopped at the breakpoint and shows the processor's registers and flags at that instant.

Note, some of the values will probably be different to those illustrated above. However, the ones used by the example program should be as specified below.

In this case the significant registers are these:

- PC This is the program counter. This tells you where in the program you are. It is actually the address of the next instruction to be executed. In this case it is 8004.
- AF AF is a register pair, in this case containing 21B7. The first two digits of this value are the A register, while the second two are the F register. So A is 21.
- BC BC is a register pair, so C is 02.

Register pairs will be explained more fully later.

This is our program and where it has stopped.

```
8000: 3E 21         >!   LD A,$21

8002: 0E 02         ..   LD C,$02
_____________________________________   We stopped here!

8004: CD 30 00      .0.  CALL $0030

8007: C9            .    RET
```

Just prior to stopping at the breakpoint the program executed the instructions LD A,$21 and LD C,$02. Thus A contains $21 (the ASCII value of "!") and C contains $02 (the API call number).

If you wish to continue the program from that point just enter:

G {return}

The program should then complete as before and a "!" character is displayed.

The Go command without an address specified causes the program to run starting at the address shown in PC of the register display. In this case 8004.

Alternatively, you can alter the register's values before you continue the program with the Go command.

Repeat the above process:

```
B 8004 {return}
Breakpoint set

G 8000 {return}

Breakpoint
PC:8004 AF:21B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
```

To change the character to be printed we need to edit the A register. To do this we enter:

```
R A {return}
A: 21  _
```

The R command followed by the register name result in the contents of the specified register being displayed. In this case the value 21. The input cursor should be at the right of the current value inviting you to enter a new value. Try entering the value 41, which is the ASCII value of the letter "A".

To confirm the register's value has been changed, enter:

```
R {return}
PC:8004 AF:41B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
SP:FE4E AF'B0E6 BC'2176 DE'F789 HL'B354 (S)00B1 IR:801E Flags'SZ---PN-
```

This shows the A register is now 41. The R command shows more registers than are shown at the breakpoint, but these can be ignored for now.

To continue the program with the new value in the A register, enter:

```
G {return}
```

The program should then complete as before, but will display the character "A" instead of the character "!".

If the program were longer it might be worth setting the breakpoint further along in the program before continuing. The program will then continue from the first breakpoint to the new breakpoint.

In more complex programs the ability to stop the program at a specified location and view the processor's registers and flags can be invaluable. Being able to change

register values before continuing is a very useful way of interacting with the program to set up test conditions and try ideas.

Single Step

The other debugging tool is Single Step.

The breakpoint allows the program to be stopped at a specified location, while the single step feature allows the program to execute one instruction at a time. Single stepping is just like setting the breakpoint after each instruction is executed.

To single step the above example, enter:

```
S 8000 {return}
PC:8000 AF:41B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
```

The state of the processor is displayed just before executing the instruction at the specified address. Note, the monitor prompt "*" is not displayed.

To execute the instruction at the address shown, press the Return key. The following should be displayed.

```
8000: 3E 21       >!    LD A,$21
PC:8002 AF:21B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
```

As mentioned previously for breakpoints, some of the register values will probably be different to those illustrated above.

The first line shows the instruction at the current address, the second line shows the processor's state after that instruction is executed. In this case the A register is now 21 and the program counter (PC) is now the address of the next instruction to be executed (8002).

Pressing the Return key then executes the next instruction and so on.

Executing the whole program one step at a time should result in a display something like this:

```
s 8000 {return}
PC:8000 AF:41B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
8000: 3E 21       >!    LD A,$21
PC:8002 AF:21B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
8002: 0E 02       ..    LD C,$02
PC:8004 AF:21B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
8004: CD 30 00    .0.   CALL $0030
Stepping over code in ROM or in monitor
```

```
!PC:8007 AF:2180 BC:1B02 DE:BDBA HL:021B IX:5B5A IY:AEAB Flags:S-------
8007: C9          .     RET
Stepping over code in ROM or in monitor
```

Where the display shows “Stepping over code in ROM or in monitor” at address 8004, the single stepping has skipped all the instructions in the monitor that are executed when outputting a character to the terminal.

When the program terminates with the RET instruction at address 8007, the ““Stepping over code in ROM or in monitor” message is displayed again as return passes back to the monitor program. Single stepping then ends and the monitor prompt is shown.

When stepping through a program you can press the Escape key to exit single stepping, or just start typing a new command. You don’t have to Escape first.

You can combine breakpoint and single stepping, for example:

```
b 8004 {return}
Breakpoint set

g 8000 {return}
Breakpoint
PC:8004 AF:2180 BC:1B02 DE:BDBA HL:021B IX:5B5A IY:AEAB Flags:S-------

s {return}
PC:8004 AF:2180 BC:1B02 DE:BDBA HL:021B IX:5B5A IY:AEAB Flags:S-------
8004: CD 30 00     .0.   CALL $0030
Stepping over code in ROM or in monitor
!PC:8007 AF:2180 BC:1B02 DE:BDBA HL:021B IX:5B5A IY:AEAB Flags:S-------
8007: C9          .     RET
Stepping over code in ROM or in monitor
```

When debugging a large program stepping from the beginning can be very tedious, so it is better to set a breakpoint at the start of the suspect code and the single step from there.

Hello World!

Now for the classic “Hello World!” Program.

We have already shown how to display a single character on the terminal.

```
8000: 3E 21         >!   LD A,$21
8002: 0E 02         ..   LD C,$02
8004: CD 30 00      .0.  CALL $0030
8007: C9            .    RET
```

To display “Hello World!” we could just repeat this:

```
LD   A,$48          ;"H"
LD   C,$02
CALL $0030

LD   A,$65          ;"e"
LD   C,$02
CALL $0030

LD   A,$6C          ;"l"
LD   C,$02
CALL $0030

---

RET
```

This would get the job done but it is very inefficient. A better way would be to have the characters of the message stored in memory and have a small program that scans the message one character at a time displaying each as it goes.

To write characters to memory you can use the Edit command (E) as follows.

```
e 8100 {return}
8100: 74          t     LD (HL),H           > "Hello world!
810C: 3F          ?     CCF                 > 0
810D: D1          .     POP DE              >
```

Press the Escape key to exit editing memory.

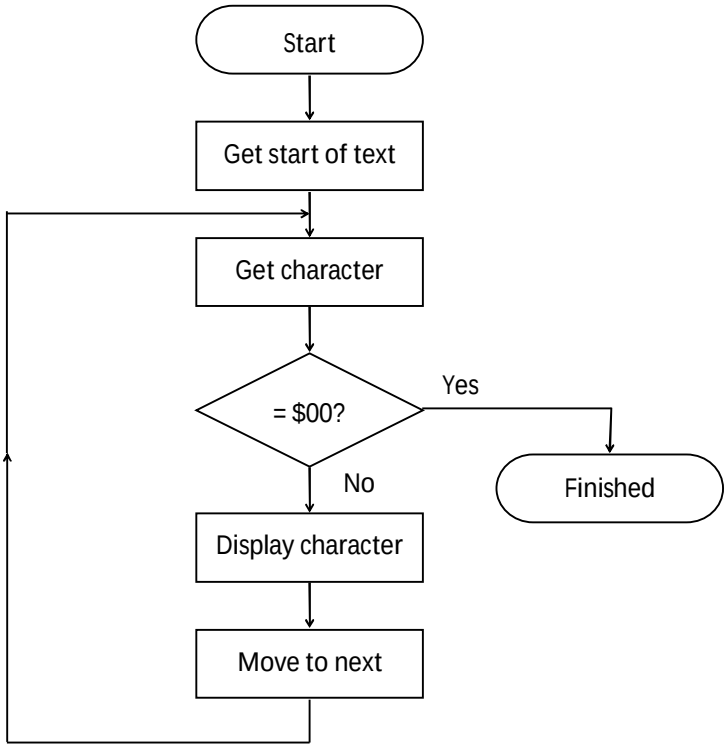
The edit command shows the current contents of memory and invites you to enter new contents. It accepts hexadecimal numbers, such as F3, or a string of ASCII characters prefixed with a Quote character. Notice that you don’t put a Quote character at the end of the characters you enter. When entering hexadecimal numbers you can enter more than one, just put a space between them.

In the example above the zero is entered as an end marker.

You can check the characters are correctly entered with the Memory command (M).

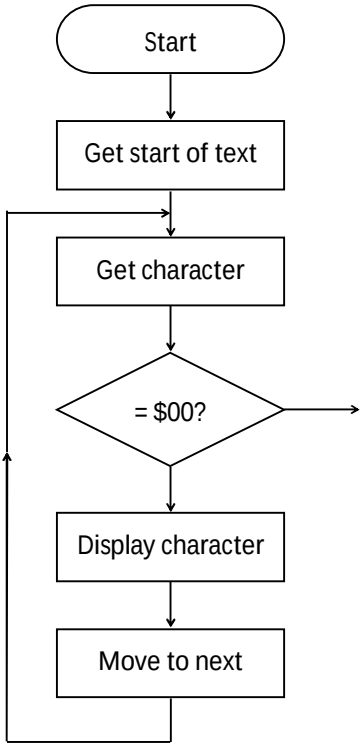
```
m 8100 {return}
8100:  48 65 6C 6C 6F 20 77 6F  72 6C 64 21 00 D1 94 E3  Hello world!....
8110:  9A 82 B7 15 BA 13 B8 01  96 53 42 57 2E D7 9C BD  .........SBW....
8120:  E4 FD D7 A9 D7 F2 8B 77  DD 89 9D 7F B7 9C 7E 24  .......w......~$
8130:  22 83 7B 12 12 92 64 AB  42 02 10 A8 80 B0 02 A6  ".{...d.B.......
8140:  61 6A D8 42 44 B8 45 26  D8 AF B6 BF E9 78 58 6B  aj.BD.E&.....xXk
8150:  2A F7 85 D7 50 34 27 9B  6F 11 49 A3 85 DB 94 90  *...P4'.o.I.....
8160:  DD 5D C6 4B 34 63 83 7B  99 EF ED 90 B1 E7 7D C2  .].K4c.{......}.
8170:  72 12 D3 DC 72 19 56 29  CA 86 7D 32 9C 8C 03 4E  r...r.V)..}2...N
```

We now have the characters in memory, so we need a simple way to read each character in turn and display it, stopping when the zero marker is reached. This is how it will work:



Converting the flow chart to assembler instructions gives us this:

```
8000: 21 00 81   !..          LD HL,$8100

8003: 7E         ~            LD A,(HL)

8004: FE 00      ..           CP $00
8006: C8         .            RET Z

8007: 0E 02      ..           LD C,$02
8009: E5         .            PUSH HL
800A: CD 30 00   .0.          CALL $0030
800D: E1         .            POP HL

800E: 23         #            INC HL

800F: C3 03 80   ...          JP $8003
```



Considering each step of the program in turn, here is a detailed explanation.

LD HL, $8100
The string of characters we want to display is in memory at 8100. The instruction "LD HL, $8100" sets the HL register pair to the value 8100, the location of the string of characters. Thus HL points to the start of the string of characters.

LD A, (HL)
The contents of memory pointed to by HL is referenced with "(HL)". Therefore "LD A,(HL)" takes the contents of memory pointed to by HL and stores it in register A. So A now contains a character from the string.

CP $00
This instruction compares the value in the A register with the value specified, in this case $00. So the instruction asks the question: is A = 0? If the value in A is equal to the specified value the processor's Zero flag (Z) is set. In other words, the difference

between the two is Zero. If they are not equal the processor's Zero flag is cleared. This is referred to as Not Zero or NZ. This instruction checks to see if we have reached the string terminator character which has the ASCII value $00.

RET Z
This is a variation on the Return (RET) instruction. Return means return from subroutine. A program is written as a subroutine, so return from subroutine ends the program. This version of the instruction is RET Z, which means Return if the Zero flag is set. Thus if the previous instruction finds that the character is $00 and thus sets the Zero flag, this instruction will end the subroutine and thus end the program.

LD C,$02
This stores the value 2 in the C register. This is in preparation for calling the Application Programming Interface (API) with function number 2.

PUSH HL
Previously when we have called the API we were not worried what the register values were when the API function finished, but here we are. We need to remember the value of HL as we still need it to find the next character in the string. We therefore have to remember the value of HL in case the API function changes it. The instruction "PUSH HL" puts a copy of HL on the Stack. The stack is like a pile of paper. If you put a new piece of paper, with a number written on it, on the stack, it will stay there. When you need the number again you can get it off the stack. You can have lots of pieces of paper on the stack but you can only put them on the top and take them off the top, so you can't change the order.

CALL $0030
This calls the API. The register value C contains the function number, in this case 2. Function 2 outputs the ASCII character in A to the terminal display.

By way of further explanation, the CALL instruction works by PUSHing the current value of the program counter on the stack and then loading the program counter with the specified address. At the end of the subroutine, the RET instruction POPs a value off the stack and writes it to the program counter register. This causes the program to return to the instruction immediately following the CALL.

POP HL
To retrieve a value from the stack you use the instruction "POP". This POPs a value off the stack and stores it in the specified register pair. In this case we restore HL to the address of the character in the character string. To use the pile of paper analogy you take the top sheet of paper off the stack.

INC HL

This increments the value of the specified register pair. Increment simply means to add one. So HL = HL + 1. As HL points to a character in the character string, it now points to the next character in the string.

JP $8003
This causes the program to jump to the specified address. It actually stores the specified value in the register PC. The program therefore loops back to consider the next character in the string.

And that's it. The Hello World! Program done.

A better world

Well actually it is not quite all over as there are some improvements to make and some lessons to learn.

The string display program above is so useful it is worth organising as a subroutine you can use in future programs.

Here is the re-usable string display routine:

```
8200: 7E         ~            LD A,(HL)
8202: FE 00      ..           CP $00
8203: C8         .            RET Z
8004: 0E 02      ..           LD C,$02
8006: E5         .            PUSH HL
8007: CD 30 00   .0.          CALL $0030
800A: E1         .            POP HL
800B: 23         #            INC HL
800C: C3 00 82   !..          JP $8200
```

Now to display the string you write:

```
8000: 21 00 81   !..          LD HL,$8100
8003: CD 00 82   ...          CALL $8200
8006: C9         .            RET
```

But there are further improvements too.

You can replace "CP $00" with "OR A". The instruction "OR A" will also set the Zero flag if A is $00, but is smaller and faster.

The CALL $0030 instruction can also be replaced. The Z80 processor has a few special CALL instructions which are smaller and faster. These are Restart instructions, where RST 30 is equivalent to CALL 0030.

This tutorial does not aim to be a complete guide to programming the Z80. There are plenty of other materials on the internet to consult, so no further explanation of the above improvements is being given here.

There is one improvement however which is specific to the Small Computer Monitor, and thus needs a mention. The subroutine to display a string of characters is so useful and so often used that the Monitor provides this function as one of its API calls. So that re-usable subroutine above is not actually needed in your tool kit of useful code. Instead to display the Hello World! String, just write this:

```
8000: 11 00 81   ...          LD DE,$8100
8003: 0E 06      ..           LD C,$06
8005: F7         .            RST 30
8006: C9         .            RET
```

Some would say it is good practice to turn this operating system specific code into a subroutine:

```
8300: 0E 06      ..           LD C,$06
8302: F7         .            RST 30
8303: C9         .            RET
```

And call it with:

```
8000: 11 00 81   ...          LD DE,$8100
8003: CD 00 83   ...          CALL $8300
8306: C9         .            RET
```

This way the bulk of your code is independent of the operating system it is running on. Again there are plenty of resources available concerning programming techniques.

Registers and Flags

We have already seen how to examine the processor's register values.

```
R {return}
PC:8004 AF:41B7 BC:6902 DE:BDBA HL:BCDE IX:5B5A IY:AEAB Flags:S--H-PNC
SP:FE4E AF'B0E6 BC'2176 DE'F789 HL'B354 (S)00B1 IR:801E Flags'SZ---PN-
```

The first line of registers shows:

- Program Counter (PC)
- Register pairs (AF, BC, DE, HL)
  - The A register is the processor's accumulator
  - The F register holds the processor's status flags
- Index registers (IX, IY)
- Flags shown as named bits (S, Z, H, P, N, C)

The second line of registers shows:

- Stack Pointer (SP)
- Alternative register set (AF', BC', DE', HL')
- The contents of the stack pointer (S)
- Interrupt vector and refresh register pair (IR)
- Alternative register flags shown as named bits (S, Z, H, P, N, C)

There are plenty of resources on line to explain these registers and flags.

We have seen how to modified a single register value.

```
R A {return}
A: 41  _
```

We can modify the value of a register pair in the same way.

```
R HL {return}
A: BCDE  _
```

The Flags register can also be modified using the same method.

```
r f {return}
F: 01  _
```

While this method of modifying the Flags register works, it is not very convenient. Each flag is a single bit within the 8-bit register, so manipulating flag bits by entering a hexadecimal byte requires quite a bit of thought. The Small Computer Monitor therefore has a separate command to modify individual flag bits.

Typing the Flag command (F) on its own displays the registers, including the flag bits.

```
f {return}
PC:7000 AF:6597 BC:1B02 DE:BDBA HL:1234 IX:5B5A IY:AEAB Flags:S--H-PNC
```

Typing the Flag command (F) followed by a flag name will set that flag bit and display the updated value.

```
f z {return}
PC:7000 AF:65D7 BC:1B02 DE:BDBA HL:1234 IX:5B5A IY:AEAB Flags:SZ-H-PNC
```

To clear a flag, type the Flag command (F) followed by the NOT version of the flag name:

```
f nz {return}
PC:7000 AF:6597 BC:1B02 DE:BDBA HL:1234 IX:5B5A IY:AEAB Flags:S--H-PNC
```

The Small Computer Monitor's User Guide gives details of all the flag and condition names that can be used with this command.

More Example Programs

This section contains some example programs designed to should how various features of the Small Computer Monitor can be used and also to show additional Z80 programming techniques.

You may want to consult the Small Computer Monitor User Guide which contains reference information related to some of these examples.

These examples are written as source code only and do not include the assembled machine code hexadecimal bytes shown in previous examples.

It is good practice when writing source code to add comments to explain the code. Comments start with a semi-colon and extend to the end of the current line. Comments can be on their own line or at the end of an instruction line. When using a full blown assembler these comments form part of the source code and help document how it works.

When using the Small Computer Monitor's in-line assembler you do not enter comments.

Displaying a new line

Earlier examples showing how to display characters on a terminal screen have avoided outputting an end of line, and thus starting any further display on a new line.

This code shows how to output a character, then a new line, then another character.

```
; Example showing output on a new line
                LD   A,31     ;ASCII character "1"
                LD   C,2      ;API call 2 = Output character
                RST  30       ;API call outputs "1"
                LD   C,7      ;API call 7 = Output new line
                RST  30       ;API call outputs new line
                LD   A,32     ;ASCII character "2"
                LD   C,2      ;API call 2 = Output character
                RST  30       ;API call outputs "2"
                RET
```

The output should look like this.

```
1
2*
```

You could output a new line by outputting a carriage return character ($0D) and possibly a line feed character ($0A) as well. The word "possibly" is the issue. Some devices use carriage for a new line, others use carriage return plus line feed, and yet others use just line feed. By using the API call to output a new line, the exact character(s) required do not have to be coded into your program. Thus when the program is moved to a system with a different new line scheme it will still run correctly.

It is not always convenient to call the API each time a new line is required. For example, you may wish to have a string of characters (like the Hello World example) which includes a new line. The Small Computer Monitor allows this by expanding the character $05 to whatever new line sequence the monitor is configured for.

Input from keyboard

This example waits for the user to press a key on the terminal's keyboard, then waits one second before displaying the character on the terminal's screen.

```
        LD   C,1      ;API call 1 = Input character
        RST  30       ;API call inputs a character
        PUSH AF       ;Remember the key character
        LD   C,+10    ;API call 10 = Delay by DE milliseconds
        LD   DE,+1000 ;Delay in milliseconds
        RST  30       ;API call delays for 1 seconds
        POP  AF       ;Restore the key character
        LD   C,2      ;API call 2 = Output character
        RST  30       ;API call outputs character
        RET
```

Flashing an LED

If you have the RC2014 digital I/O module you can write a simple program to flash an LED.

```
8000   Loop:  LD   A,1      ;LED bit zero ON value
              OUT (0),A     ;Turn LED on
              LD   C,+10    ;API call 10 = Delay by DE milliseconds
              LD   DE,+500  ;Delay in milliseconds
              RST  30       ;API call delays for 0.5 seconds
              LD   A,0      ;LED bit zero OFF value
              OUT (0),A     ;Turn LED off
              LD   C,+10    ;API call 10 = Delay by DE milliseconds
              LD   DE,+500  ;Delay in milliseconds
              RST  30       ;API call delays for 0.5 seconds
              JP   Loop     ;Repeat forever
```

Suggested program location (8000) shown above.

The above example assumes the module is configured for output port address zero.

When this program runs, the LED flashes, and flashes, and flashes. It does not end until the processor resets. It is actually quite difficult to write a program that can do something as apparently simple as flashing a light whilst also doing something else.

If you are using a fully featured assembler, "Loop" is a label that can be used to reference Loop's memory address. In this case the instruction "JP Loop" is assembled as a jump to the memory location containing the instruction "LD A,1".

When using the Small Computer Monitor's in line assembler only one line is considered at a time and labels on other lines are not supported. Therefore it is necessary to enter the address of the jump, as earlier examples have shown. You will soon be wanting a fully featured assembler!

Better flashing!

The “Flashing an LED” example can be improved. Whenever you see blocks of code being repeated it is worth looking to see if the code can be written to avoid the duplication.

If we take the common block of code out and turn it into an “Led” subroutine we get this:

```
8000   Loop:      LD    A,1     ;LED bit zero ON value
                  CALL Led      ;Write to LEDs and delay
                  LD    A,0     ;LED bit zero OFF value
                  CALL Led      ;Write to LEDs and delay
                  JP    Loop    ;Repeat forever

8100   Led:       OUT (0),A     ;Turn LED(s) on/off
                  LD   C,+10    ;API call 10 = Delay by DE milliseconds
                  LD   DE,+500  ;Delay in milliseconds
                  RST  30       ;API call delays for 0.5 seconds
                  RET           ;Return from this subroutine
```

Suggested program locations (8000 and 8100) are shown above.

Now the first Loop can be expanded to include more light changes, such as:

```
Loop:             LD    A,1     ;LED bit 0 ON value
                  CALL Led      ;Write to LEDs and delay
                  LD    A,2     ;LED bit 1 ON value
                  CALL Led      ;Write to LEDs and delay
                  LD    A,3     ;LED bit 2 ON value
                  CALL Led      ;Write to LEDs and delay
                  LD    A,4     ;LED bit 3 ON value
                  CALL Led      ;Write to LEDs and delay
                  JP    Loop    ;Repeat forever
```

Now the loop above looks like it can be improved too. Introducing tables...

To further refine this code and to make it scale up better to a longer light show, we will use a data table. Data tables are very powerful devices.

In the above example we repeat the sequence:

```
LD A,x
CALL Led
```

For each light change.

A better solution, especially for longer sequences, would be to have a list (or table) of the LED output values and have a small routine which reads each item in the list and outputs it to the LEDs. To change the light sequence you then only need to edit the table.

A table for the above example would be:

```
DB 1,2,3,4
```

In this case the assembler directive DB stands for Define Byte(s) and results in the number values listed being written into memory as a series of bytes.

The following program has been entered into the on line assembler at www.Asm80.com. See www.scc.me.uk for a brief Asm80.com tutorial.

```
                .ORG    $8000

; Load the B register with the length of the list (14 decimal)
; Load HL with the address of the start of the list (TABLE)
START:          LD      B,14
                LD      HL,TABLE

; Loop back to here for each LED output value
LOOP:

; Load the A register with the contents of memory addressed by HL
; This being one of the LED output values
                LD      A,(HL)

; Call the subroutine which outputs the value to the LED port
; and then waits for a while before returning
; As the LED subroutine might change the value of the registers
; B and HL, which we are using to count and point to the table, we
; must first store their values on the stack (PUSH instructions) and
; after we must restore their values from the stack (POP instructions)
                PUSH    BC
                PUSH    HL
                CALL    LED
                POP     HL
                POP     BC

; Increment HL to point to the next value in the table
                INC     HL

; Repeat until we get to the end of the table (when B gets to zero)
; DJNZ is a clever instruction which decrements the B register
; and if it is not then zero it jumps to the specified location
                DJNZ    LOOP

; In order to repeat the sequence indefinitely we go back to the
; start of the program
                JR      START

; Alternatively we could just return to the monitor
                RET

; This subroutine first outputs the value in the A register to
; the LED port, then waits a while before returning
; Warning: API calls (RST $30) may change the value of some
; registers so it may be necessary to store their values before
```

```
; the call and restore them after
LED:            OUT    ($00),A
                LD     C,10
                LD     DE,500
                RST    $30
                RET

; This table is a list of LED output values
TABLE:          DB     1,2,4,8,16,32,64,128
                DB     64,32,16,8,4,2
```

Often tables have an end marker so that it is not necessary to know the length of the table in advance. The table can simply be read until the end marker value is found. In this case however no byte value can be reserved as an end marker because any LED pattern might be required for the light show.

A table with a NULL ($00) as an end marker might look like this:

```
                DB     64,32,16,8,4,2
                DB     0      ; End marker
```

The on line assembler produces a HEX file to send to the Small Computer Monitor as documented in the Asm80.com tutorial. It also produces a list file as shown below. In this case the comments have been excluded.

```
8000                            .ORG $8000
8000  06 0E      START:        LD   B,14
8002  21 1C 80                  LD   HL,TABLE
8005  7E          LOOP:         LD   A,(HL)
8006  C5                        PUSH  BC
8007  E5                        PUSH  HL
8008  CD 13 80                  CALL  LED
800B  E1                        POP   HL
800C  C1                        POP   BC
800D  23                        INC   HL
800E  10 F5                     DJNZ  LOOP
8010  18 EE                     JR    START
8012  C9                        RET
8013  D3 00       LED:         OUT  ($00),A
8015  0E 0A                     LD    C,10
8017  11 F4 01                  LD    DE,500
801A  F7                        RST   $30
801B  C9                        RET
801C  01 02 04 08 10 20 40 80 40 20 10 08 04 02
                  TABLE:        DB   1,2,4,8,16,32,64,128
                                DB   64,32,16,8,4,2
```

In this example the data table is a simple list of numbers. Think of this as a spreadsheet with only one column. By using more than one column much more complex tables can be created. By carefully encoding values and functions into combinations of bits and bytes, quite complex problems can be reduced to small

programs which gain most of their apparent sophistication from the data table. The assembler and disassembler built into the Small Computer Monitor use this technique. Trying to write code to directly handle each Z80 instruction would result in a huge unwieldy program.

Yet more fun with LEDs

How can we do something as apparently simple as flashing a light or running a light show whilst also doing something else?

There are various techniques to achieve this.

You could use interrupts if your Small Computer has a hardware timer. Sadly the standard RC2014 kits do not have these as standard.

A hardware timer would enable the system to be interrupted at regular time intervals. Whatever software is running when the hardware interrupt occurs is suspended and your designated interrupt handler routine is run. The interrupt handler should be a short subroutine which, in this case, changes the LED state(s) before returning. Upon return the previous activity is resumed. This would normally happen so quickly the user would not notice any delays.

Without a hardware timer a similar subroutine could be used to sequence the LEDs, but it would have to be called regular by polling. That is to say it is necessary for the software running on the target system to call the LED subroutine regularly. This is a right nuisance and can, at best, only result in the LED subroutine being called at approximately the required interval.

As the ability to easily run simple background activities, like flashing LEDs, is so desirable the Small Computer Monitor provides a polling solution. It does this the best it can without unduly burdening the user and the user's programs. It is a very compromised solution but is still rather handy to have. The mechanism adopted relies on the fact that most of the time a computer is simply waiting for the user to press a key. Waiting for a key press is quite a predictable process and allows reasonably accurate time keeping when that is the only thing going on. Accuracy becomes far less predictable under other conditions.

Running simple background activities, such as flashing LEDs, is an extremely important mechanism and leads ultimately to modern event driven operating systems.

Eight bit computers are relatively limited and usually work by your program taking control of the system (other than hardware interrupts) and usually being structured in a loop or series of loops, which wait for some input and then do something with it before repeating the process. In event driven systems the user's program is more like a slave to the operating system and only gets a look in when the operating system requests that the user's program responds to some event.

As described above, with a hardware timer interrupt you need to write a subroutine to sequence the LEDs by one step each time the subroutine is called. The same is true for the polled timer. In this case however we call the subroutine an event handler.

As an example we will convert the previous table based LED light show into an event driven program. This requires the program to remember where in the sequence it had reached last time it was called.

In the previous implementation the Small Computer was not running any other code so the processor's registers contained the current position in the sequence. Now it is necessary to store these values in variables so that the register values can be restored the next time the subroutine is called.

The program below was created using www.Asm80.com. It uses the Small Computer Monitor's timer events to sequence the light show.

```
                .ORG    $8000

; Start the program by setting its initial state
START:

; Turn on the Small Computer Monitor's Idle events
                LD      A,1
                LD      C,$13
                RST     $30

; Set timer 2 to call the subroutine "POLL" every 50 x 10 milliseconds
                LD      A,50
                LD      DE,POLL
                LD      C,$15
                RST     $30

; Set up to initialise or repeat the light show sequence
REPEAT:         LD      A,14
                LD      (COUNT),A
                LD      HL,TABLE
                LD      (POINT),HL
                RET

; This subroutine is called regularly by the Small Computer Monitor
; It uses the variables POINT and COUNT to keep track of where
; in the light show it has got to
POLL:

; First restore the pointer to the data table,
; then read the value from the table and output it to the LEDs,
; then increment the pointer to the next table value and store it
                LD      HL,(POINT)
                LD      A,(HL)
                OUT     ($00),A
                INC     HL
```

```
                LD      (POINT),HL

; Decrement the table counter
; When it reaches zero, go set up to repeat the sequence
                LD      A,(COUNT)
                DEC     A
                JR      Z,REPEAT
                LD      (COUNT),A

                RET

; Table of LED output values
TABLE:          DB      1,2,4,8,16,32,64,128,64,32,16,8,4,2

; These are the variables used to remember where in the table
; we are
COUNT:          DS      1
POINT:          DS      2
```

Start the program as usual with the command:

G 8000 {return}

Note that while this is running the Small Computer Monitor is still active so you can type commands and run programs.

While the above light show is running, try speeding it up by altering the timer interval to 2 x 10 milliseconds with the monitor's API command:

API 15 2 8019 {return}

Warning: this will only work if the program is exactly as shown above, as this API command sets the event handler to address $8019.

API $15 sets event timer to the specified multiple of 10 milliseconds and the event handler to the specified address. See the Small Computer Monitor user guide for more details of API functions.

Assemblers

The Small Computer Monitor has an in-line assembler built in, which is really handy, but it is not well suited to writing large programs.

There are two basic approaches to developing large machine code programs:

- Install a full blown editor and assembler on your target small computer
- Use an editor and assembler on a PC (or similar)

The first approach is all most people had back in the day. Often this would have been a suit of development software running under CP/M. This option is still available today. The software and documentation can be found on line, but you will need a high end small computer to run it.

The second option is very attractive today as we have easy access to relatively powerful modern computers. Software to edit source code and to assemble it into Z80 machine code is available on line. There are also simulators to enable most software to be tested on a modern computer before installing it on to the target small computer.

Take a look at “www.Asm80.com” which provides an on line editor and assembler suitable for use with the Small Computer Monitor. A tutorial to help you get started writing programs with Asm80.com for use with the Small Computer Monitor can be found at www.scc.me.uk.

Once you have written and assembled your program on a modern computer you can download it to your target small computer to run and test on real hardware. Assemblers can usually produce an Intel Hex file containing the machine code program. This file can be downloaded to the target small computer using the same terminal program you have already been using. Just look for an option like “Send Text File”.

The Small Computer monitor recognises an Intel Hex file as it arrives and decodes it, writing the machine code to memory as it does so. Once completely downloaded you can run the program with the Go command. You then have all the features of the Small Computer Monitor to help test and debug the program.

These assemblers are not specific to the Small Computer Monitor, so are not documented in detail here.

Fault Finding

If you do not see the monitor's sign on message on the terminal when you switch the system on, then here are some things to try:

Press the RC2014 reset button.

If your RC2014 was not previously tested with the supplied BASIC ROM, then if possible check it does work with the BASIC ROM. If that is not possible then you'll need to go through all the usual fault finding processes: check the power supply, check all links, check no chips have bent legs and thus not making contact with their socket, carefully inspect all soldering, check all the chips are inserted the right way round, check all the components are in the right place. Check your serial connection looks right and that the terminal is correctly set. Then cry!

If your RC2014 was known to be working with the supplied BASIC ROM, then verify the Small Computer Monitor ROM contains the correct code and check the links related to addressing the ROM (especially if the chip has a different capacity to the one containing BASIC). Other than that you would appear to have an odd problem as the Monitor ROM should, in theory, work if the RC2014 standard BASIC ROM works.

It should be noted that there are a number of different serial modules available for the RC2014 and they are not all compatible. Currently the Small Computer Monitor only works with official RC2014 serial modules or modules totally compatible with these, and also SIO/2 modules following Grant Searle's register addressing.

Parts and Suppliers

The following is a list of parts and suppliers used during development of the Small Computer Monitor.

RC2014 official modules

Information at www.rc2014.co.uk
Parts purchased through Tindie:
https://www.tindie.com/stores/Semachthemonkey/?ref=offsite_badges&utm_source=sellers_Semachthemonkey&utm_medium=badges&utm_campaign=badge_medium

Chip programmer

WINGONEER TL866CS Universal USB MiniPro EEPROM FLASH BIOS Programmer AVR GAL PIC SPI
Amazon ASIN: B071H5XGR7
https://www.amazon.co.uk/gp/product/B071H5XGR7/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1

FTDI cable

TTL-232R-5V - USB to Serial Converter Cable, 5V, 6Way, 1.8m
Farnell order code: 2419945
http://uk.farnell.com/ftdi/ttl-232r-5v/usb-to-serial-converter-cable/dp/2419945?ost=2419945&iscrfnonsku=false&ddkey=http%3Aen-GB%2FElement14_United_Kingdom%2Fsearch

FTDI 'cable'

HALJIA FT232RL FTDI USB to TTL Serial Converter Adapter Module Mini USB 3.3V 5.5V Board for Arduino
Amazon ASIN: B06XDH2VK9
https://www.amazon.co.uk/gp/product/B06XDH2VK9/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1

USB-RS232 cable

UGREEN 20210 USB Serial Cable, USB to RS232 DB9 9 pin Converter Cable
Amazon ASIN: B00QUZY4UG
https://www.amazon.co.uk/gp/product/B00QUZY4UG/ref=oh_aui_search_detailpage?ie=UTF8&psc=1
Note, you still need a null modem lead between this and the RC2014.

EEPROM 8k x 8 bit

Microchip Technology AT28C64B-15PU Parallel EEPROM Memory, 64kbit, 150ns, 4.5 → 5.5 V PDIP 28-Pin
RS part number: 127-6572
http://uk.rs-online.com/web/p/eeprom-memory-chips/1276572/

Contact Information

If you wish to contact me regarding the Small Computer Monitor please use the contact page at www.scc.me.uk.

Issues related to the RC2014 can be posted on the RC2014-Z80 google group.

Stephen C Cousins, Chelmsford, Essex, United Kingdom.