Small Computer Monitor

User Guide

Monitor version 0.4 for the Z80 CPU

Software and Documentation by Stephen C Cousins

Edition 0.4.0

CONTENTS

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.

The Monitor includes a capable debugging environment with the following features:

- Boot loader to load Intel HEX files from a PC or similar
- Memory display and editing
- Register display and editing
- In-line disassembler
- In-line assembler
- Breakpoint debugging
- Single step debugging (without the need for special hardware)

Primary input/output is via a serial port to a terminal.

Drivers and support for common hardware is included.

The monitor can be modified for different hardware and can be compiled with only the basic features in order to fit in a small ROM.

An executable file version of the Small Computer Monitor can also be compiled, thus allowing it to be run from a storage device rather than from ROM.

The monitor is implemented without using interrupts, thus leaving them free for applications and additional device drivers.

An Application Programming Interface (API) is provided to enable some of these features to be used by other software.

Conventions

Within this guide hexadecimal numbers are prefixed by either '$' or '0x', unless the number is a command parameter. Command parameters default to hexadecimal, so no prefix is required.

Serial port

The default configuration is to use a terminal or terminal emulation software to communicate with the target hardware running the Small Computer Monitor program.

The serial port is set for 115200 bits per second (assuming the clock is 7.3728 MHz), 8 data bits, 1 stop bit, no parity, no flow control. Line termination is Carriage Return ($0D) plus Line Feed ($0A). There should not normally be any need to add delays when sending characters to the target.

RC2014 systems

This document describes the Small Computer Monitor as configured for RC2014 systems.

The RC2014 is an open system and can thus have diverse hardware. The Small Computer Monitor program is designed to work with the official modules, and may not work with third party modules.

The ROM version of the Small Computer Monitor fits in an 8k byte ROM which is mapped into memory from address $0000 to $1FFFF. It requires RAM from $FD00 to $FFFF. The ROM is not paged out of memory during operation.

The Small Computer Monitor for RC2014 does not use Interrupts.

Commands

The Small Computer Monitor is designed to be used with a simple serial terminal or terminal emulation software. Commands are therefore typed in plain text and results are displayed on the terminal.

The Monitor's commands, described below, have the general syntax:

command-name parameter(s)

The command name is often just a single letter.

Parameters are shown, in this document, enclosed by "<" and ">", and further enclosed by "[" and "]" if the parameter is optional. So "E [<start-address>]" describes a command "E" with the start address as an optional parameter.

Command names and any parameters are delimited by a space character. Single letter commands are the exception as they do not require a space between the single letter command and the first parameter.

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

In the examples below, 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" followed by the Return key, and the monitor displays "Breakpoint set".

```
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 an numeric character. The exception to this rule is operands in the assembler.

? 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>
```

The configuration identifier, 'A' in the above example, indicates which build this code is. One source code version can be tailored by conditional assembly for different configurations. Each of these configurations has a unique configuration identifier. Some configuration identifiers refer to the same hardware but with different configurations, such as different memory locations. So a ROM version may have a different identifier to a soft-loading version.

Configuration identifiers currently assigned:

- A = '1' Small Computer Workshop Simulator
- A = '2' Small Computer Development Kit
- A = 'A' RC2014 Standard ROM

API function call

Syntax: API <function number> [<A register value>] [<DE register value>]

The monitor provides an Application Programming Interface (API) to enable other software to use some of its features.

This command enables API functions to be called from the monitor prompt.

The command has three parameters:

- API function number
- Optional value of the A register passed to the function
- Optional value of the DE register pair passed to the function

On completion of the API function the monitor displays the returned value of the register A and the register pair DE.

Full details of the API functions is given later in this guide.

Below are a few examples.

Input a character from the console

To input a character from the current console input device, use API function $01.

```
API 1 {return}
```

Nothing happens until a character is available from the console input device, which is a long winded way of saying nothing happens until you press a key. If the letter “a” is pressed the monitor displays:

```
61 0021
```

The returned values are:

A = ASCII value of character input ($61)
DE = unspecified value

Output a character to the console

To output a character to the current console output device, use API function $02.

To output a pling character (“!”), which has ASCII value $21, enter the command:

```
API 2 21 {return}
!21 0021
```

Note the pling character (“!”) is displayed before the returned register values.

The returned values are:

A = ASCII value of character output ($21)
DE = unspecified value

Digital I/O module LEDs

The API includes a set of functions to manage a simple digital output port, such as the RC2014 digital I/O module’s LEDs.

The first thing to do is to specify which port you wish to control. This is done by calling API function $17 with the port number in the A register. The RC2014 digital I/O output port is usually address $00, thus the API command is:

```
API 17 0 {return}
00 0000
```

The returned values are:

A = current output port data byte ($00)
DE = unspecified value

This functions also clears the output port to zero.

Now to turn on the LED on bit 2:

```
Api 1B 2 {return}
04 0002
```

The returned values are:

A = current output port data byte ($04)
DE = unspecified value

Related functions are:

- $17 Select and initialise output port
- $18 Write to output port
- $19 Read from output port
- $1A Test output port bit
- $1B Set output port bit
- $1C Clear output port bit
- $1D Invert output port bit

There is also a set of API function for handle a simple input port.

Assemble instructions

Syntax: A [<memory address>]

The in-line assembler is invoked by this command.

The memory address parameter is optional. If supplied the assembler begins at the specified address. If not, the last referenced address is used instead.

The address is shown in hexadecimal followed by the hexadecimal byte or bytes which make up the machine code instruction currently at that address. The ASCII characters represented by those bytes is then shown, followed by the instruction mnemonic and operands. Non-printable ASCII characters are shown as a dots.

The user may then enter a new instruction mnemonic and operands which is assembled into machine code and entered into memory at the address shown. The new instruction is then displayed in the format specified above, and the next instruction displayed.

For example:

```
A 8000 {return}
8000: C3 56 10      .V.       JP $1056          >
8003: 00            .         NOP               > ld a,12 {return}
8003: 3E 12         >.        LD A,$12
8005: 00            .         NOP               >
```

The initial instructions shown, such as JP $1056, will be whatever happens to be in memory at the time, so will probably not match the examples shown.

Instead of entering a new instruction, the user can press {return} to move to the next instruction, or {escape} to exit the assembler and return to the monitor prompt. Entering a period character (“.”) followed by {return} also exits the assembler.

The delimiter between the operation and the first operand must be a space, while the delimiter between operands can be a comma or a space.

The assembler supports all the documented Z80 instructions and uses standard Zilog mnemonics. A good reference document is Zilog’s Z80 CPU User Manual, just search for “zilog um0080”.

Instruction operands can be hexadecimal or decimal numbers. The default is hexadecimal, but unlike monitor command parameters they may need clarification. For example, the hexadecimal number BC is also the name of the register pair BC.

To clarify that a hexadecimal number is intended, and not a register name, it is sometimes necessary to ensure the first character is numeric (ie. 0 to 9, not A to F). So prefixing BC with a zero clarifies it is intended to be a hexadecimal number.

Alternatively a hexadecimal number can be prefixed with a "$" or "0x" to ensure it is interpreted correctly. Thus "$BC" and "0xBC" are the hexadecimal number BC.

To identify a number as decimal, prefix with a "+" sign.

For example:

```
LD A,B     = Load register A with register B
LD A,0B    = Load register A with hexadecimal value 0B (decimal 11)
LD A,$B    = Load register A with hexadecimal value 0B (decimal 11)
LD A,0xB   = Load register A with hexadecimal value 0B (decimal 11)
LD A,+11   = Load register A with decimal value 11 (hexadecimal 0B)
```

When entering the address for a relative jump, the address can be either the displacement or the absolute address. An address of $00 to $FF is treated as a displacement, while an address of $0100 to $FFFF is treated as an absolute address. Absolute addresses must be within range of a relative jump or an error message will be shown.

For example:

```
a 8010 {return}
8010: 00          .      NOP                > jr 30
8010: 18 30       .0     JR $30  (to $8042)
8012: 00          .      NOP                > jr 8010
8012: 18 FC       ..     JR $FC  (to $8010)
8014: 00          .      NOP                > jr 9999
Syntax error
8014: 00          .      NOP                >
```

One little quirk with the assembler syntax is that the "JP (HL)" instruction must be entered, and is displayed, as "JP HL". Similarly for "JP IX" and "JP IY".

Breakpoint set or clear

Syntax: B [<memory address>]

Breakpoints provide an aid to program debugging. When a program is running and reaches a breakpoint, program execution stops and the current state of the processor registers is displayed. Register values can be altered if required, and execution continued.

This command enables the breakpoint to be set or cleared.

To set the breakpoint, enter the command 'B' followed by the address at which the breakpoint should be set. The breakpoint can only be set in random access memory (RAM), not in read only memory (ROM). Only one breakpoint is provided.

For example:

```
b 8000 {return}
Breakpoint set
```

To clear the breakpoint, enter the command 'B' with or without an address. If no address is specified the current breakpoint is cleared. If an address is specified the current breakpoint is cleared and then the new breakpoint is set.

For example:

```
B {return}
Breakpoint cleared
```

To continue execution after a breakpoint, use the with the "Go" command without specifying an address. Just enter ***G {return}***.

Breakpoints (and single stepping) work by replacing the instruction in memory with the instruction RST 28. This causes a call to the Monitor to handle the breakpoint (or step). If a RST 28 instruction is encountered which is not there as a breakpoint or step instruction, program execution stops and "Trap" is displayed.

Console

Syntax: Console <device number>

The Small Computer Monitor supports a number (currently 6) of console style input and output devices. This command allows selection of which one is the current console device and thus provides input and output for the monitor's command line interpreter.

Devices are numbered 1 to 6.

Device 1 is set as the default console device at reset.

Device 1 is the first serial port. Typically this will be the RC2014 serial I/O module (using 68B50 ACIA) or channel A of the RC2014 dual channel SIO/2 module (using Z80 SIO/2).

Device 2 is the second serial port. Typically this will be channel B of the RC2014 dual channel SIO/2 module (using Z80 SIO/2).

Devices 3 and 6 have not yet been allocated.

If you have the dual channel SIO/2 module with a terminal connected to each channel, you can swap between them with the commands "Console 1" and "Console 2". Channel B of the SIO is initialised at reset but is not used by the monitor unless the user selects it as the console device. It is therefore free to be configured and used as required.

There is also an API function to select the console device from within software.

Devices

Syntax: Devices

A list is displayed of hardware devices detected by the Small Computer Monitor when it started up.

For example:

```
Devices {return}
Serial ACIA
```

The Small Computer Monitor can detect the presence of some hardware devices and includes driver software to support them.

The real reason for this feature is to support both types of serial interfaces supplied as official RC2014 modules, namely the classic 6850 ACIA and the newer Z80 SIO/2. By detecting these modules the Small Computer Monitor can configure itself to use whichever module is available and thus avoid the need for multiple versions of this program.

In addition to using whichever serial module is detected for its own input and output, the Small Computer Monitor passes on this benefit to any software which makes use to the Monitor's API.

Currently supported devices:

- Serial ACIA (6850 family)
- Serial SIO/2 (Z80 peripheral)

When the ACIA device is detected the Small Computer Monitor initialises it for 115200 bits per second (assuming the clock is 7.3728 MHz), 8 data bits, 1 stop bit, no parity, no flow control, no interrupts. It is then configured as the console input and output device.

When the SIO/2 device is detected the Small Computer Monitor initialises channel A for 115200 bits per second (assuming the clock is 7.3728 MHz), 8 data bits, 1 stop bit, no parity, no flow control, no interrupts. It is then configured as the console input and output device. Channel B is initialised at reset with the same settings as channel A, but is not used unless the user selects it as the console device. It is therefore free to be configured and used as required.

Disassemble instructions

Syntax: D [<memory address>]

The in-line disassembler is invoked by this command.

The memory address parameter is optional. If supplied the disassembler begins at the specified address. If not, the last referenced address is used instead.

The address is shown in hexadecimal followed by the hexadecimal byte or bytes which make up the machine code instruction at that address. The ASCII characters represented by those bytes is then shown, followed by the instruction mnemonic. Non-printable ASCII characters are shown as dots.

For example:

```
d 1066 {return}
1066: C3 03 FF      ...       JP $FF03
1069: 31 80 FE      1..       LD SP,$FE80
106C: 11 00 00      ...       LD DE,$0000
106F: 21 00 10      !..       LD HL,$1000
1072: 01 69 00      .i.       LD BC,$0069
1075: ED B0         ..        LDIR
```

After the block of instructions is shown, the user can press {return} to display the next block, or {escape} to exit the disassembler and return to the monitor prompt. Alternatively, a new command can be entered without first returning to the monitor prompt.

The disassembler supports all the official documented Z80 instructions and uses standard Zilog mnemonics.

When disassembling a relative jump instruction, both the displacement and the absolute address is shown:

```
d 5010 {return}
5010: 18 30         .0        JR $30  (to $5042)
```

Edit memory

Syntax: E [<memory address>]

The memory editor is presented in a similar way to the assembler, but instead of entering instruction mnemonics you enter hexadecimal or ASCII values.

The memory address parameter is optional. If supplied the editor begins at the specified address. If not, the last referenced address is used instead.

The address is shown in hexadecimal followed by the hexadecimal byte or bytes which make up the machine code instruction at that address. The ASCII characters represented by those bytes is then shown, followed by the instruction mnemonic and operands. Non-printable ASCII characters are shown as a dots.

The user may then enter new memory contents, press {return} to move to the next instruction, or {escape} to exit the editor and return to the monitor prompt. Entering "^" followed by the return key causes the editor to go back one location. Entering a period character (".") followed by {return} also exits the memory editor.

Hexadecimal numbers are entered without the need to clarify as hexadecimal. ASCII characters are entered by preceding with a quote character.

For example:

```
e 8040 {return}
8040: 00          .     NOP             > 3e ff
8042: 00          .     NOP             > "Hellp
8047: 00          .     NOP             > ^
8046: 70          p     LD (HL),B       > "o
8a047: 00          .     NOP            >
```

Fill memory

Syntax: FILL <first address> <last address> <data byte>

Use this command to fill an area of memory with a specified data byte.

It is unlikely the memory will be totally clear, but if it were then a memory display command would look something like this.

```
m 8000 {return}
8000:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8010:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8020:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8030:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8040:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8050:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8060:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8070:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

We could then enter a fill command to write a specified value, in this case $55, to a range of memory locations, in this case $8010 to $801F.

```
fill 8010 801f 55 {return}
```

We should then be able to see the result by issuing another memory display command.

```
m 8000 {return}
8000:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8010:  55 55 55 55 55 55 55 55  55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
8020:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8030:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8040:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8050:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8060:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
8070:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Flags display or modify

Syntax: F [<status>]

The processor's flags are displayed or modified.

If the command is entered without any parameters the processor's registers, including the flags register, is displayed:

```
f {return}
PC:0001 AF:00D7 BC:0003 DE:0004 HL:0005 IX:0006 IY:0007 Flags:SZ-H-PNC
```

The flags register is shown in hexadecimal as part of the AF register pair and also as individual flag bits. If the flag letter is shown, the flag is set, otherwise is it clear.

If a valid parameter is entered the appropriate flag bit is set or cleared.

Valid parameters include the flag letter (eg. "Z"), the flag letter prefixed with "N" (eg. "NZ") and the 'condition' in conditional instructions (eg. "PO" in "JP PO, <address>"). Thus there are several ways to reference the flags. Due to a conflict between Positive (P) and Parity (P), the parity flag is set with "Pa", not just "P".

The table below shows valid flag names in square brackets and valid condition names in curly brackets and valid .

Flag	*Description*	*Set / Clear*	*Set meaning*	*Clear meaning*	*Bit*
S	Sign	[S] [NS]	Negative {M}	Positive {P}	7
Z	Zero	[Z] [NZ]	Zero {Z}	Not Zero {NZ}	6
H	Half carry	[H] [NH]	Half carry	Not Half carry	4
	or Half borrow		Half borrow	Not Half borrow	4
P	Parity	[Pa] [NP]	Even {PE}	Odd {PO}	2
	or Overflow (V)		Overflow	No Overflow	2
N	Add/subtract	[N] [NN]	Subtract	Add	1
C	Carry	[C] [NC]	Carry {C}	No Carry {NC}	0

For example, to set the zero flag:

```
f z {return}
PC:0001 AF:0042 BC:0003 DE:0004 HL:0005 IX:0006 IY:0007 Flags:-Z----N-
```

Go to program

Syntax: G [<memory address>]

This command allows a machine code program to be executed (run).

Program execution begins at the specified address. If no address is specified execution begins at the address set in the PC variable, as displayed by the Register command. All other processor registers are set to the values stored in the associated variables, again as displayed by the Register command.

If the program being executed is written as a subroutine, whereby it ends with a RET (return) instruction, then at the end of the program control is passed back to the monitor and a monitor prompt is displayed. If the program does not return then control does not pass back to the monitor until the system is reset or a breakpoint is encountered. Of course if the program has a problem and crashes then anything can happen!

In the example below the test program runs until the breakpoint is reached.

```
a 8000 {return}
8000: 00            .       NOP                 > ld a,55 {return}
8000: 3E 55         >.      LD A,$55
8002: 00            .       NOP                 > ret {return}
8002: C9            .       RET

d 8000 {return}
8000: 3E 55         >.      LD A,$55
8002: C9            .       RET


r {return}
PC:0000 AF:0002 BC:0003 DE:0004 HL:1234 IX:0006 IY:0007 Flags:------N-
SP:FE7E AF'0012 BC'0013 DE'0014 HL'0015 (S)0016 IR:0017 Flags'---H--N-

b 8002 {return}
Breakpoint set

g 8000 {return}
Breakpoint
PC:8002 AF:5502 BC:0003 DE:0004 HL:1234 IX:0006 IY:0007 Flags:------N-
8002: C9            >.      RET
```

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
```

The Z80 has a separate address range for input/output (I/O) devices, together with separate processor instructions to access them. This command addresses devices in the I/O space, not the memory space. Generally the I/O space is limited to 256 addresses ($00 to $FF).

Users of the official RC2014 digital I/O card can typically read the switch inputs with the command:

```
I 0 {return}
```

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 1080 {return}
1080:  19 10 F9 3E 00 11 AC 10  CD 83 19 3E 01 11 AE 10  ...>.......>....
1090:  CD 83 19 CD C1 24 CD 90  17 11 B0 10 CD 50 18 CD  .....$.......P..
10A0:  D8 24 31 80 FE CD 3F 11  C3 22 12 C9 ED 4D ED 45  .$1...?.."...M.E
10B0:  05 05 53 6D 61 6C 6C 20  43 6F 6D 70 75 74 65 72  ..Small Computer
10C0:  20 4D 6F 6E 69 74 6F 72  20 62 79 20 53 74 65 70   Monitor by Step
10D0:  68 65 6E 20 43 20 43 6F  75 73 69 6E 73 05 56 65  hen C Cousins.Ve
10E0:  72 73 69 6F 6E 20 30 2E  31 2E 32 20 66 6F 72 20  rsion 0.1.2 for
10F0:  00 50 43 3A 2C 41 46 3A  2C 42 43 3A 2C 44 45 3A  .PC:,AF:,BC:,DE:
```

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.

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}
```

The Z80 has a separate address range for input/output (I/O) devices, together with separate processor instructions to access them. This command addresses devices in the I/O space, not the memory space. Generally the I/O space is limited to 256 addresses ($00 to $FF).

Users of the official RC2014 digital I/O card can typically write to the LED outputs with the command:

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

In the above example 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.

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-
```

The first line shows the most commonly used register:

- PC Program Counter
- AF Accumulator (A) and flags (F)
- BC Register pair BC
- DE Register pair DE
- HL Register pair HL
- IX Index register IX
- IY Index register IY
- Flags register broken down into individual flag bits:
  - S Sign flag
  - Z Zero flag
  - H Half carry flag
  - P Parity/overflow flag
  - N Add/subtract flag
  - C Carry flag

  If the flag letter is shown, the flag is set, otherwise is it clear.

The second line shows the rest of the registers:

- SP Stack Pointer
- AF' Alternative accumulator and flags
- BC' Alternative register pair BC
- DE' Alternative register pair DE
- HL' Alternative register pair HL
- (S) Contents of the stack pointer
- IR Interrupt vector register (I) and memory refresh register (R)
- Flags' is the alternative flags register broken down into bits (as above)

When the optional parameter is entered, the specified register can be edited.

For example:

```
r hl {return}
HL: 0005  1234 {return}
```

In the above example the HL register pair is specified. The current value of HL is displayed (0005) and the user can either enter a new value for HL (1234), or press {escape} to leave the register unchanged. Entering the command “r” now shows the updated registers:

```
r {return}
PC:0001 AF:0002 BC:0003 DE:0004 HL:1234 IX:0006 IY:0007 Flags:------N-
SP:0011 AF'0012 BC'0013 DE'0014 HL'0015 (S)0016 IR:0017 Flags'---H--N-
```

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.

All execution stops, including interrupt routines. The monitor then restarts.

Memory is not cleared, but essential variables are initialised.

The Small Computer Monitor outputs a sign-on message to the console output device (usually a terminal) followed by the monitor prompt character ('*'). For example:

```
Small Computer Monitor
*
```

Step one instruction

Syntax: S [<memory address>]

This command allows single stepping of machine code programs.

Program execution begins at the specified address. If no address is specified execution begins at the address set in the PC variable, as displayed by the Register command. All other processor registers are set to the values stored in the associated variables, again as displayed by the Register command.

Initially the processor registers and flags are displayed.

Pressing the Return key then causes a single instruction is executed, and the resulting processor registers and flags to be.

Pressing {return} again will step another instruction and again display the processor state.

Pressing {escape} will exit single stepping mode and return to the monitor prompt.

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

Below is an example of a simple program being stepped.

```
s 8000 {return}
PC:8000 AF:0040 BC:125C DE:FF86 HL:8000 IX:2034 IY:0007 Flags:-Z------
8000: 3E 02       >.    LD A,$02
PC:8002 AF:0240 BC:125C DE:FF86 HL:8000 IX:2034 IY:0007 Flags:-Z------
8002: CD 10 80     ...   CALL $8010
PC:8010 AF:0240 BC:125C DE:FF86 HL:8000 IX:2034 IY:0007 Flags:-Z------
8010: 3C          <     INC A
PC:8011 AF:0300 BC:125C DE:FF86 HL:8000 IX:2034 IY:0007 Flags:--------
8011: C9          .     RET
PC:8005 AF:0300 BC:125C DE:FF86 HL:8000 IX:2034 IY:0007 Flags:--------
8005: C9          .     RET
```

No special hardware is required for single stepping as it is all handled in software. However, like breakpoints, single stepping can only occur when the code is in RAM. If a call is made to a routine in read only memory, the call is stepped over. Single stepping then continues after the call is complete.

Similarly, calls into the monitor code are stepped over. This means calling the Monitor's API appears as just one step and not hundreds.

When a call into read only memory or monitor code occurs, the message: “Stepping over code in ROM or in monitor” is displayed.

Single stepping (and breakpoints) work by replacing the instruction in memory with the instruction RST 28. This causes a call to the monitor to handle the step (or breakpoint). If a RST 28 instruction is encountered which is not there as a breakpoint or step instruction, program execution stops and “Trap” is displayed. Single stepping can not pass a “Trap” instruction.

Hex File Loader

The Small Computer Monitor includes a means of receiving an Intel Hex File from the serial port.

This allows an assembly language program to be created, edited and assembled on a PC (or similar), and then 'sent' to the target computer from a terminal program.

Terminal programs usually have a feature called "Send text file" (or similar). This opens a file on the PC and 'sends' it to the target as if it were typed in on the terminal.

Some terminal programs allow text to be 'Pasted' into the terminal as if typed in. This can be a convenient way of taking a Hex File from another program and sending it to the target system.

Assemblers usually have a means of creating an Intel Hex File, thus it is quick and easy to author a program on a PC and send it to target hardware running the Small Computer Monitor.

Below is an example of an Intel Hex File.

```
:104000002135400609 0E80EDB3061A3E41F5DB80EE
:10401000CB5728FAF1D3813CCD2A4010F0DB80CB7E
:104020004728FADB81D38118F4C9F5C50E0911645C
:1040300000F7C1F1C91814C403C105681100FFFFDE
:00000001FF
```

When the Small Computer Monitor has received the whole file it displays either "Ready" or "File error". If it appears to hang it is probably because it has not received the correct file termination sequence.

There not usually a need to set the terminal to add any delays as it sends the characters, as the monitor can handle hex file loading at a continuous 115200 bits per second when the processor is running at 7.3 MHz.

Self-test

The Small Computer Monitor has a simple self-test feature which runs at reset.

To see the output of the test you need a RC2014 digital I/O module, or equivalent, and this must have its output port set to the default address of zero. Without this module the test may still help when using an oscilloscope for fault finding as the sequence is simple and repeats if no RAM is found.

As long as the power, processor, clock and bus are basically sound the self-test should run. It does not depend on working RAM or a working serial port.

At reset the self-test flashes each LED on the digital I/O board in turn. This takes about half a second to complete. If the LEDs flash in sequence you know the power, processor, bus, ROM and clock are basically working.

After that it does a very limited RAM test. It only tests a few locations, but probably enough to spot a serious fault with the upper 32k of RAM. If it fails the LEDs repeat their sequence and the test repeats, so constantly flashing LEDs indicate a RAM fault.

If the RAM test passes then the LEDs are all turned off and the monitor tries to identify the serial module. If this fails bit 0 LED is turned on to indicate the problem.

A pass through without failure will be indicated by all LEDs off (after the initial cycle) and hopeful a message on the terminal. At this point, with or without a message on the terminal, the other main modules look in good shape.

Other ROM based software, such as 32k MS BASIC, should stand a good chance of running on a system that passes this test.

A pass without a sign on message on the terminal suggests the problem likely relates to the serial interface or terminal.

If the LEDs don't even manage the initial sequence, it likely means the processor is not running code so the fault could be on any of the main modules or the bus, but hopefully it will be a good solid 'digital' fault.

It is worth stripping the system to the minimum at some point when trying the self-test, so that only the processor, clock, ROM and digital I/O are present.

Application Programming Interface (API)

The monitor provides an Application Programming Interface (API) to enable other software to use some of its features.

The following functions are available:

- $00 System reset
- $01 Input character
- $02 Output character
- $03 Input status
- $04 Input line
- $05 Input line default
- $06 Output line
- $07 Output new line
- $08 Get version details
- $09 Claim jump table entry
- $0A Delay in milliseconds
- $0B Output embedded message
- $0C Read jump table entry
- $0D Select console in/out device
- $0E Select console input device
- $0F Select console output device
- $10 Input from specified console device
- $11 Output to specified console device
- $12 Poll idle events
- $13 Configure idle events
- $14 Timer 1 control
- $15 Timer 2 control
- $16 Timer 3 control
- $17 Output port initialise
- $18 Write to output port
- $19 Read from output port
- $1A Test output port bit
- $1B Set output port bit
- $1C Clear output port bit
- $1D Invert output port bit
- $1E Input port initialise
- $1F Read from input port
- $20 Test input port bit

To access these functions from a program, a CALL is made to address $0030¶ with the function number in the C register. Any parameters and results are passed in other registers.

Unless otherwise stated, calling any of these functions may result in registers AF, BC, DE and HL being modified and therefore they may not contain the same value after the function call as they did before the call. Registers IX IY I AF' BC' DE' HL' are not modified (unless otherwise stated).

As a simple example, function zero is used to reset the system:

```
8000: 0E 00      LD C,$00     ; Function 0 = System reset
8002: CD 30 00   CALL $0030   ; Call API
```

The Z80 processor has a special instruction to CALL a few specific addresses in memory. This is called a Restart instruction and has the mnemonic RST.

RST 30 performs the same function as CALL $0030, but it does it with a single byte instruction. It is therefore smaller and faster.

The above code can therefore be improved:

```
8000: 0E 00      LD C,$00     ; Function 0 = System reset
8002: F7         RST 30       ; Call API
```

¶ Address $0030 and RST 30 only apply if the Small Computer Monitor is located at the bottom of memory or the bottom of memory is RAM allowing the Monitor to modify these locations.

If the Small Computer Monitor is not located at the bottom of memory, then the API call address is the start of the Monitor + $0030. Thus if the monitor starts at address $E000, the API call address is $E030.

However, if the monitor can write to lower memory, due to RAM being located there and not ROM, then it is able to write the relevant jump instruction to location $0030. In which case the CALL $0030 or RST 30 instructions can still be used.

API function $00, system reset

Parameters: none
Returns: none

This function 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.

All execution stops, including the user program. The monitor then restarts.

Memory is not cleared, but essential variables are initialised.

Example:

```
8000: 0E 00      LD C,$00     ; Function 0 = System reset
8002: F7         RST 30       ; Call API
```

This function is equivalent to the Monitor's RESET command.

The Small Computer Monitor outputs a sign-on message to the console output device (usually a serial terminal). For example:

```
Small Computer Monitor
```

API function $01, input character

Parameters: none
Returns: A = Character input from console

This function waits for a character from the current console input device, usually a serial terminal.

When a character arrives, it is returned in the A register.

The function does not return until a character arrives.

Example:

```
8000: 0E 01      LD C,$01     ; Function 1 = Input character
8002: F7         RST 30       ; Call API

8003: C9         RET          ; Return to monitor
```

API function $02, output character

Parameters: A = Character to output to console
Returns: A = Character output to console

This function outputs the specified character to the current console output device, usually a serial terminal.

The ASCII value of the character to be output is passed in the A register.

The function does not return until the character has been output.

Example:

```
8000: 3E 21      LD A,'!'      ; ASCII value of character '!'
8002: 0E 02      LD C,$02      ; Function 2 = Output character
8004: F7         RST 30        ; Call API

8005: C9         RET           ; Return to monitor
```

The above example causes a pling character ('!') to be output to the console.

By combining this example with the previous example, we can input a character from the console and then output it to the console, thus enabling us to see what we have typed. The example below repeats this process until the Return key is pressed.

```
8000: 0E 01      LD C,$01      ; Function 1 = Input character
8002: F7         RST 30        ; Call API
; Register A now contains the input character
; which is then output to the console
8003: 0E 02      LD C,$02      ; Function 2 = Output character
8005: F7         RST 30        ; Call API
; Now repeat until the Return key is pressed
8006: FE 0D      CP $0D        ; Is the character a Return ?
8008: 20 F6      JR NZ,$8000   ; No, so repeat
800A: C9         RET           ; Yes, so exit program
```

API function $03, input status

Parameters: none
Returns: NZ flagged in input character available from console

This function checks the status of the current console input device, usually a serial terminal.

When a character is available the function returns with NZ flagged, otherwise Z is flagged.

The function does not wait until a character arrives.

Example:

```
8000: 0E 03      LD C,$03     ; Function 3 = Input status
8002: F7         RST 30       ; Call API

8003: C9         RET          ; Return to monitor
```

A more useful example, below, waits for a character before returning.

```
8000: 0E 03      LD C,$03     ; Function 3 = Input status
8002: F7         RST 30       ; Call API
8003: 28 FB      JR Z,$8000   ; Repeat if Z flagged

8003: C9         RET          ; Return to monitor
```

This function is provided to allow a program to run whilst checking regularly for an input character. Using the input character function would not work well here as it waits for a character, thus blocking further execution of the program until a character is available.

API function $04, input line

Parameters: DE = Start of line in memory
A = Size of line in memory
Returns: DE = Start of line in memory
A = Number of characters in the line

This function inputs a line of characters from the console input device.

The line contains any characters entered from the console input device until a Carriage Return character (ASCII $0D) is entered. The line ends when the Carriage Return is entered but does not include the Carriage Return character.

The line is input to memory starting at DE and is returned as a null (zero) terminated list of characters. Register A sets the maximum size of the line in bytes, which must include one byte for the null terminator.

On return the null (zero) terminated line of characters is stored in memory at the requested location. This location is returned in DE. Register A contains the number of characters in the line, excluding the null terminator.

For example:

```
8000: 11 00 90   LD DE,$9000  ; Start of line = $9000
8003: 3E 51      LD A,$51     ; Size of line = $50 + 1 for terminator
8005: 0E 04      LD C,$04     ; Function 4 = Input line
8007: F7         RST 30       ; Call API

8008: C9         RET          ; Return to monitor
```

After the line “hi” is entered, DE is $9000, A is $02 and the memory contains:

```
9000: 68 69 00
```

API function $05, input line default

Parameters: none
Returns: DE = Start of line
A = Number of characters in line

Input a line of characters from the console input device to the default line buffer.

The functions is similar to function 4, above, except the memory used is the memory reserved for the monitor's own line input. As a result the line will be overwritten when characters are entered at the monitor prompt.

The benefit of using this function is that it does not require its own memory allocation and the call is simpler:

```
8000: 0E 05      LD C,$05     ; Function 5 = Input line default
8002: F7         RST 30       ; Call API

8003: C9         RET          ; Return to monitor
```

API function $06, output line

Parameters: DE = Start of line in memory
Returns: none

This function outputs the specified line of characters to the output device.

The line of characters must be null (zero) terminated. The line is therefore in the same format as the input line obtained with functions 4 and 5.

```
8000: 11 00 90   LD DE,$9000  ; Start of line = $9000
8003: 0E 06      LD C,$06     ; Function 6 = Output line
8005: F7         RST 30       ; Call API

8006: C9         RET          ; Return to monitor

9000: 68 69 00   DB "hi",0    ; Line = "hi"
```

To embed a new line code into the string it is best to avoid specifically using Carriage Return and/or Line Feed as the sequence varies with different hardware. The monitor provides a special character to signify a new line. This is the constant kNewLine which has the value 5. The above example can be modified to include a cursor move to the start of the next lines:

```
9000: 68 69 05   DB "hi",5,0  ; Line = "hi" + new line
9003: 00
```

API function $07, output new line

Parameters: none
Returns: none

This function outputs a new line character sequence to the console output device.

This will cause the console's cursor to move to the start of the next line. The character or characters output will depend on the console device. Typically this will be a Carriage Return and Line Feed (ASCII $0D, $0A).

```
8000: 0E 07      LD C,$07     ; Function 7 = Output new line
8002: F7         RST 30       ; Call API

8003: C9         RET          ; Return to monitor
```

A more complete example, below, outputs a '!', then a new line, then another '!'.

```
8000: 3E 21      LD A,'!'     ; ASCII value of character '!'
8002: 0E 02      LD C,$02     ; Function 2 = Output character
8004: F7         RST 30       ; Call API

8005: 0E 07      LD C,$07     ; Function 7 = Output new line
8007: F7         RST 30       ; Call API

8008: 3E 21      LD A,'!'     ; ASCII value of character '!'
800A: 0E 02      LD C,$02     ; Function 2 = Output character
800C: F7         RST 30       ; Call API

800D: C9         RET          ; Return to monitor
```

API function $08, get version details

Parameters: none
Returns: D, E and B are the monitor source code version
D = Major, E = Minor, B = Revision
C is the configuration identifier
Number characters are development configurations
Letter characters are release configurations
H and L are the target hardware ID
H = Primary ID, L = Options
A is reserved for future use, currently zero

This function returns details of the monitor code version, the configuration identifier character and the target hardware identifier.

The monitor source code version is detailed in registers D, E and B. Register D contains the major version number, E contains the minor version number and B the revision number. For example, if D=1, E=2 and B=3, the version is 1.2.3

The configuration identifier indicates which build this code is. One source code version can be tailored by conditional assembly for different configurations. Each of these configurations has a unique configuration identifier. Some configuration identifiers refer to the same hardware but with different configurations, such as different memory locations. So a ROM version may have a different identifier to a soft-loading version.

Configuration identifiers currently assigned:

A = '1'	Small Computer Workshop Simulator
A = '2'	Small Computer Development Kit
A = 'A'	RC2014 Standard ROM

The target hardware identifier is detailed in registers H and L. Register H is the primary identifier, with register L describing options within the target hardware.

Target hardware identifiers currently assigned:

H = 1	Small Computer Simulator
	L = 0
H = 2	Small Computer Development Kit
	L = 0
H = 3	RC2014-Z80
	L, bit 0 set if serial 6850 ACIA detected
	L, bit 1 set if serial SIO/2 detected
	L, bit 2 to 7 = Not currently used, all cleared to zero

H = 4 Small Computer 101
L, bit 0 set if serial 6850 ACIA detected
L, bit 1 set if serial SIO/2 detected
L, bit 2 to 7 = Not currently used, all cleared to zero

API function $09, claim jump table entry

Parameters: A = Entry number (0 to n)
DE = Address of function code
Returns: none

Some system features, such as console in and out, are redirected through a jump table in RAM. This function enables these jump table entries to be set to jump to any code required.

The following jump table entries are available:

- $00 Non-maskable interrupt handler
- $01 Restart $08, handler (not currently used)
- $02 Restart $10, handler (not currently used)
- $03 Restart $18, handler (not currently used)
- $04 Restart $20, handler (not currently used)
- $05 Restart $28, breakpoint handler
- $06 Restart $30, applications programming interface (API) handler
- $07 Restart $38, mode 1 interrupt handler
- $08 Console input routine
- $09 Console output routine
- $0A Reserved for get console input status
- $0B Reserved for get console output status
- $0C Idle event handler
- $0D Timer 1 event handler
- $0E Timer 2 event handler
- $0F Timer 3 event handler
- $10 Device 1 input character default = serial port channel A
- $11 Device 1 output character default = serial port channel A
- $12 Device 2 input character default = serial port channel B
- $13 Device 2 output character default = serial port channel B
- $14 Device 3 input character
- $15 Device 3 output character
- $16 Device 4 input character
- $17 Device 4 output character
- $18 Device 5 input character
- $19 Device 5 output character
- $1A Device 6 input character
- $1B Device 6 output character

By changing the console input and output routines, any custom hardware can be integrated into the monitor and used instead of the standard hardware.

If the monitor program is in ROM and that ROM is mapped to the very bottom of memory, it is not possible to directly change the interrupt code for interrupt mode 1 and non-markable interrupts. This is because code for these interrupts must begin at fixed locations in memory which happen to be very near the bottom of memory, just where the ROM is! By redirecting these interrupts through the jump table it is possible to configure your own interrupt handler. The same redirection feature is provided to Restart instructions $08, $10, $18 and $20, which are currently free to be used as required. Also Restart instructions $28 (breakpoint handler) and $30 (API handler) are redirected through this table.

Console devices 1 to 6 are also redirected through the table enabling each to be individually replaced as required.

In the example below, the console input routine is replaced by a new routine at location $9000.

```
8000: 3E 08      LD A,$08     ; Console input jump table entry
8002: 11 00 80   LD DE,$9000  ; Console output routine at $9000
8005: 0E 09      LD C,$09     ; Function 9 = Claim jump
8007: F7         RST 30       ; Call API

8008: C9         RET          ; Return to monitor
```

API function $0A, delay

Parameters: DE = Number of milliseconds delay
Returns: none

This function simply waits for the specified number of milliseconds to pass before returning.

The delay is created by a simple software loop and during this time the processor is not looking for key presses or any other activity. Therefore, unless the system is using interrupts to capture such actions, they will be missed.

The delay time is dependent on the clock speed of the processor. If a non-standard clock speed is used the delay time will be some multiple or fraction of DE milliseconds.

For the RC2014-Z80 system the standard clock speed is 7.3728 MHz.

In this example there is a delay of 2 seconds before the monitor prompt is shown.

```
8000: 11 D0 07   LD DE,$07D0  ; Delay time = 2000 ms
8003: 0E 09      LD C,$0A     ; Function $0A = Delay
8005: F7         RST 30       ; Call API

8006: C9         RET          ; Return to monitor
```

When entering this code with the assembler, the first instruction is “LD DE,+2000”. The plus sign indicates a decimal number. Once entered this is shown in hexadecimal as $07D0.

API function $0B, output embedded message

Parameters: A = Message number (0 to n)
Returns: none

This function outputs messages embedded in the Small Computer Monitor to the console output device.

The following messages are available:

System messages

- $00 Message = None (null)
- $01 Message = “Small Computer Monitor ”
- $02 Message = “Devices:”
- $03 Message = <About this version of the monitor>
- $04 Message = <List of hardware devices detected>

Monitor messages

- $20 Message = “Bad command”
- $21 Message = “Bad parameter”
- $22 Message = “Syntax error”
- $23 Message = “Breakpoint set”
- $24 Message = “Breakpoint cleared”
- $25 Message = “Unable to set breakpoint here”
- $26 Message = Monitor commands Help text
- $27 Message = “Feature not included”
- $28 Message = “Ready”

All the message end with a new line character sequence, except for message $00 and $01. Message zero outputs nothing.

The following example displays information about the Small Computer Monitor.

```
8000: 3E 03      LD A,3        ; Message number 3
8003: 0E 0B      LD C,$0B      ; Function $0B = Message
8004: F7         RST 30        ; Call API

8005: C9         RET           ; Return to monitor
```

Running this example should give a display something like this.

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

API function $0C, read jump table entry

Parameters: A = Entry number (0 to n)
Returns: DE = Address of function code

Some system features, such as console in and out, are redirected through a jump table in RAM. This function enables these jump table entry addresses to be read.

See function $09 (claim jump table entry) for list of jump table functions.

API function $0D, select console input/output device

API function $0E, select console input device

API function $0F, select console output device

Parameters: A = Device number (1 to 6)
Returns: none

The Small Computer Monitor supports a number (currently 6) of console style input and output devices. These functions allow selection of which one is the current console input device and which one is the current console output device.

The current device is the one which provides input and/or output for the monitor's command line interpreter. It is also the one which, by default, is the input and output for the user's programs.

The first function selects the device which provides both console input and console output, while the other two functions allow selection of just input or just output.

Devices are numbered 1 to 6.

Device 1 is set as the default console device at reset.

Device 1 is the first serial port. Typically this will be the RC2014 serial I/O module (using 68B50 ACIA) or channel A of the RC2014 dual channel SIO/2 module (using Z80 SIO/2).

Device 2 is the second serial port. Typically this will be channel B of the RC2014 dual channel SIO/2 module (using Z80 SIO/2).

Devices 3 to 6 have not yet been allocated.

If you have the dual channel SIO/2 module with a terminal connected to each channel, you can swap between them with this function (or with the Console command). Channel B of the SIO is initialised at reset but is not used by the monitor unless the user selects it as the console device. It is therefore free to be configured and used as required.

API function $10, input a character from the specified device

Parameters: E = Device number (1 to 6)
Returns: A = Character input from the specified device (if there is one)
NZ is flagged if a character has been input

A character is input from the specified console input device.

If no character is available the function returns with the Z flag set.

API function $11, output a character to the specified device

Parameters: A = Character to be output
E = Device number (1 to 6)
Returns: If character output succeeded: NZ flagged and A != 0
If character output failed: Z flagged and A = Character to output

The character specified in register A is output to the console device number in register E.

If the device is not able to accept the character the function returns with the Z flag set. This can occur, for example, when a serial output device is busy.

In the example below the pling character ("!") is output to device 4. If the device is busy the request is repeated until in succeeds. If the specified device, in this case 4, does not exist the API function will always return the failed status and this subroutine will never complete.

```
8100: 3E 21      LD A,'!'      ;Character to output

8102: 1E 04      LD E,4        ;Output device number
8104: 0E 11      LD C,$11      ;API $11 = output to device
8106: F7         RST 30        ;Call API
8107: 28 F9      JR Z,$8102    ;Repeat if busy
8109: C9         RET
```

API function $12, poll idle events

Parameters: none
Returns: none

The Small Computer Monitor supports a system of timer events which are generated when the processor is otherwise idle.

When the system is doing such things as waiting for console input it can be set to repeatedly poll the idle event handler. This looks for timer events and, when appropriate, calls the user functions configured to handle them.

By doing this the Small Computer Monitor makes it easy to have simple background task running. These timer events are really handy for activities like flashing lights.

There is a slight problem however. Typical small computers, such as the standard RC2014 kits, do not usually include a hardware timer. As a result the time period of the events generated is not accurate. It is reasonable when simply waiting for user input, but becomes very inaccurate under less predictable conditions.

This function allows events to be polled when the processor would otherwise be busy for extended periods and thus not checking for events. To keep background events running this call should be made regularly by any program which runs for more than a few milliseconds. Even doing this the timer events will be very erratic.

Fortunately not all activities that need regular processing require accurate timing. A thermostatically controlled heating system, for example, would work just fine with this kind of crude timing.

The following example shows the code required to poll background events.

```
8000: 0E 0B      LD C,$12      ; Function $12 = Poll idle events
8002: F7         RST 30        ; Call API
```

Don't forget when making any API calls the processor's registers can be changed by that call, so it may be necessary to store important registers before the call and restore them after.

API function $13, configure idle events

Parameters: A = Event mode:
0 = Off (no events are detected)
1 = Software generated timer events
Returns: none

This function enables selection of the event mode. Mode zero is Off (no events are detected). Mode one is software generated timer events.

Software generated timer events are explain above (API function $12).

Following a processor reset the event mode is zero, the Small Computer Monitor does not look for events.

To enable software generated timer events:

```
8000: 3E 01      LD A,1       ; Event mode 1
8002: 0E 0C      LD C,$13     ; Function $13 = Configure events
8004: F7         RST 30       ; Call API
```

To disable software generated timer events:

```
8000: 3E 00      LD A,0       ; Event mode 0
8002: 0E 0C      LD C,$13     ; Function $13 = Configure events
8004: F7         RST 30       ; Call API
```

WARNING: With software generated events turned on the system is not as responsive to console input as it is with events turned off. This is not normally a problem as console input tends to mean slow typing on a keyboard. However, it may be necessary to set delays in a terminal program when sending a HEX file to the monitor. Hex files are normally sent flat out and it is assumed the monitor will keep up, which may not be the case when events are turned on.

API function $14, timer 1 event set up

API function $15, timer 2 event set up

API function $16, timer 3 event set up

Parameters: A = Event time period
DE = Address of timer event handler
Returns: none

The Small Computer Monitor supports three event timers. This group of functions enables these timers to be set up.

Timer 1 has a resolution of 1 millisecond and can be set to a period of 1 to 255 milliseconds (0.001 to 0.255 seconds).

Timer 2 has a resolution of 10 milliseconds and can be set to a period of 10 to 2550 milliseconds (0.01 to 2.55 seconds).

Timer 3 has a resolution of 100 milliseconds and can be set to a period of 100 to 25500 milliseconds (0.1 to 25.5 seconds).

When the specified timer event occurs the subroutine at address DE is called. This subroutine should only be a short piece of code that does not keep the processor busy for long. It must also preserve all the processor's registers, with the exception of AF and HL which are preserved automatically.

The timer events are only detected when event mode 1 is set by API call $13.

An explanation of the timer accuracy issue is given above (API function $12).

The following code inverts the state of bit 1 LED on the RC2014 digital I/O module:

```
8000: 3E 01      LD A,1        ;LED bit number
8002: 0E 1D      LD C,$1D      ;API $1D = invert output bit
8004: F7         RST 30        ;Call API
8005: C9         RET
```

To call this code every 0.5 seconds and thus flash the LED, issue these API calls:

API 16 5 8000 {return}
API 13 1 {return}

To first API call sets Timer 3 to run the code at $8000 every 5 x 100 milliseconds. The second API call enables the timers. To stop the timers enter:

API 13 0 {return}

API function $17, output port initialise

Parameters: A = Output port address
Returns: A = Output port data byte (which will be zero)

This function sets the output port address to be used for subsequent output port functions. The output port is cleared (all outputs low/off) by this call.

This set of functions allows a simple 8-bit output port, such as the RC2014 digital I/O module's LED port, to be manipulated.

API function $18, write to output port

Parameters: A = Output data byte
Returns: A = Output port data byte

API function $19, read from output port

Parameters: none
Returns: A = Output port data byte

API function $1A, test output port bit

Parameters: A = Bit number 0 to 7
Returns: A = 0 and Z flagged if bit is low (off)
A != 0 and NZ flagged if bit is high (on)

API function $1B, set output port bit

Parameters: A = Bit number 0 to 7
Returns: A = Output port data byte

API function $1C, clear output port bit

Parameters: A = Bit number 0 to 7
Returns: A = Output port data byte

API function $1D, invert output port bit

Parameters: A = Bit number 0 to 7
Returns: A = Output port data byte

API function $1E, input port initialise

Parameters: A = Input port address
Returns: A = Input port data byte

This function sets the input port address to be used for subsequent input port functions.

This set of functions allows a simple 8-bit input port, such as the RC2014 digital I/O module's push button port, to be accessed.

API function $1F, read from input port

Parameters: none
Returns: A = Input port data byte

API function $20, test input port bit

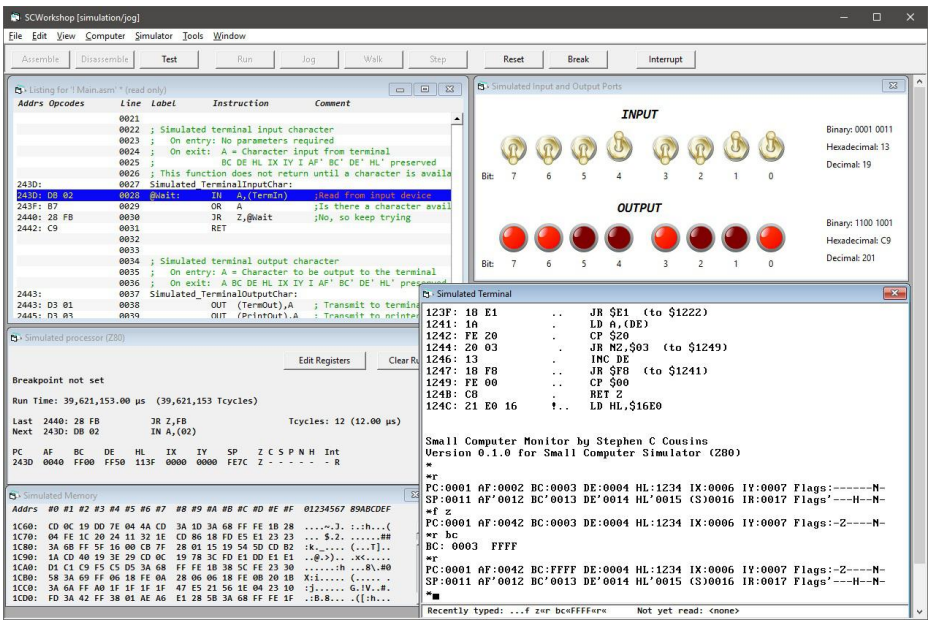
Parameters: A = Bit number 0 to 7
Returns: A = 0 and Z flagged if bit is low (off)
A != 0 and NZ flagged if bit is high (on)

Source Code and Assembly

The source code for the Small Computer Monitor is freely available. Assembling it however may not be so easy.

Small Computer Workshop

The source code has been written specifically within my own Small Computer Workshop integrated development environment (IDE), which includes its own editor, assembler and simulator, as illustrated below.



When writing the development environment a study was made of currently available assemblers. Unfortunately what the study revealed was that there is not one common standard, meaning source code written for one assembler would not usually assemble using a different assembler.

Initially it was hoped that an assembler could be designed which could cope with all variations of syntax and conventions. This however was deemed unrealistic due to the extra code complexity it created and the problem caused by some mutually exclusive features.

The final decision was to write the system in the way that provided the most desirable features without creating unnecessary complications. The resulting

assembler is, ironically, not totally compatible with any other and thus exasperates the very compatibility problem originally identified. However, it is also not hampered by any restrictions such compatibility would have imposed.

The assembler has a 'C' style pre-processor which includes some special statements relating to the development environment, such as #TARGET. It also supports a range of IF.. ELSE.. END and INCLUDE statements to allow conditional assembly, which is how it supports different builds of functionality options and target hardware options, from the same source code.

Assembler directives are prefixed with a period ('.'), such as ".ORG". Assembler directives include support for memory sections using ".CODE" and ".DATA".

Local labels are supported. These begin with the '@' character. These labels are local to the most recent global label, allowing local labels to be reused in other functions. There are quite a few local versions of the label "@Loop" in the source code!

Hexadecimal values are prefixed with "$".

Memory Map

The Small Computer Monitor's code can be assembled to almost anywhere in memory and so can its workspace (in RAM). However, there are some locations in the bottom of memory which have fixed uses and can not be moved.

The standard build places the monitor code starting at $0000 and currently extends to about 7 kbytes. Workspace and stack is at the very top of memory from about $FD00 to $FFFF.

The fixed locations at the bottom of memory are determined by the Z80's special features, namely hardware reset, restart instructions and interrupt service. These addresses can fall within the monitor ROM or be in RAM.
These locations are:

$0000	Restart 00, reset monitor (or CP/M 2 warm boot)
$0003	Reserved for CP/M 2, IOBYTE
$0004	Reserved for CP/M 2, drive and user numbers
$0005	Reserved for CP/M 2, FDOS entry point
$0008	Restart 08, not currently used
$0010	Restart 10, not currently used
$0018	Restart 18, not currently used
$0020	Restart 20, not currently used
$0028	Restart 28, use for debugging breakpoint
$0030	Restart 30, API entry point
$0038	Restart 38, interrupt mode 1 handler
$005C	Reserved for CP/M 2, default FCB (to $007F)
$0066	NMI interrupt handler (or CP/M 2 default FCB)
$0080	Reserved for CP/M 2, DMA buffer (to $00FF)

If the monitor code is assembled to the bottom of memory, these special locations are included in the monitor code and are thus set up as required. If the monitor is assembled elsewhere in memory these locations must be in RAM and are written to during initialisation to establish the required contents.

Target Hardware

The Small Computer Monitor can be built for a range of Z80 based computer hardware and simulators.

The Monitor code can be run from ROM or RAM, and can be assembled to any desired location. However, to fully utilise features such as its Application Programming Interface (API) and Breakpoints it must either be running at the bottom of memory (from location $0000 upwards) or the bottom of memory must be in RAM to allow the monitor to write Jump instructions to key locations in the range $0000 to $0068. Very simple Z80 hardware is likely to meet this requirement by having ROM at location $0000. More sophisticated hardware will likely have ROM at location $0000 but allow it to be paged out and replaced by RAM once the system is up and running.

The only other hardware requirement is an interface to a terminal or a computer running terminal software. This interface will normally be a simple serial port. The terminal only needs to be a simple Teletype style device.

The hardware does not need to support interrupts as the Monitor does not currently use them. Thus interrupts are currently free for user programs and additional device drivers.

Currently the Monitor's full feature set occupies just over 6k bytes of code space and requires about 700 bytes of RAM (including stack space). The code size can be significantly reduced by excluding features like the in-line assembler.

Hardware type 1: Small Computer Simulator (Z80)

This 'hardware' is a software simulation only. It forms part of my in-house Small Computer Workshop's integrated development environment (IDE). This is used to write and test hardware independent software.

Hardware type 2: Small Computer Development Kit (Z80)

The kit is in-house development hardware only.

Hardware type 3: RC2014 (Z80)

Type 3 hardware covers RC2014 systems. Official RC2014 modules are available from "Semachthemonkey" via Tindie:
https://www.tindie.com/stores/Semachthemonkey/

Z80 based RC2014 systems are supported provided they meet the ROM/RAM requirements above and use one of the serial I/O modules listed below as the primary connection to a terminal or terminal emulation software.

As all the currently available complete RC2014 kits (Mini, Classic, Plus and Pro) meet these requirements the Small Computer Monitor is compatible with all of them. All these kits provide a means of running the Small Computer Monitor from ROM, whilst some also provide configurations suitable to run the Small Computer Monitor in RAM.

The standard RC2014 build is designed to run from ROM starting at address $0000. It is an 8 kbyte ROM image which supports the serial I/O modules listed below.

Serial modules supported:

- Official RC2014 Serial I/O Module (68B50 or 63B50 ACIA)
  Base I/O address = $80, ACIA RS signal = A0
  Set for 115200 baud (when using 7.3728 MHz clock), 8 data bits, 1 stop bit, no flow control, no parity
- Official RC2014 Dual Serial Module (SIO/2)
  Base I/O address $80, SIO A/B signal = A1, SIO C/D signal = /A0
  Set for 115200 baud (when using 7.3728 MHz clock), 8 data bits, 1 stop bit, no flow control, no parity
- Dual Serial Module (SIO/2 following Grant Searle's design)
  Base I/O address $80, SIO A/B signal = A0, SIO C/D signal = A1
  Set for 115200 baud (when using 7.3728 MHz clock), 8 data bits, 1 stop bit, no flow control, no parity

The Small Computer Monitor auto detects these modules so one version of the Monitor code supports all of the above modules, although you can only have one of these in the system at a time.

If the Small Computer Monitor is unable to identify a suitable serial module at reset it turns On bit 0 LED of the digital I/O module to indicate the problem.

Grant Searle's original SIO design has different register selection to the official RC2014 module. Any module following Grant's design should work. This includes the popular SIO module by Dr. Scott M. Baker (tested).

For information about RC2014 systems visit RC2014.co.uk

Hardware type 4: SC101

Type 4 hardware is my own minimal homebrew Z80 system. This has 128k bytes of ROM (Flash), 128k bytes of RAM, a simple memory paging system, a bit-bang (software) serial port and expansion via RC2014 bus sockets.

Bugs, Quirks, Limitations and To do list

This section details all those embarrassing little issues that make software development such fun and mean software it is never really finished.

Bugs

The outstanding bugs are currently:

B0000 Assembling a JR instruction with an out of range address is not detected.

Quirks

Q0001 Fixed: The assembler's Restart instruction parameter must be a two digit hexadecimal number and must not be prefixed with "$" or "0x". Thus the instruction "RST 08" is valid, while "RST 8" and "RST $08" are not.

Q0002 The assembler and disassembler's handling of instructions JP (HL), JP (IX) and JP (IY) does not follow standard Z80 mnemonics. These are actually input and output as JP HL, JP IX and JP IY.

Limitations

L0001 Fixed: The Intel hex file loader does not currently check the integrity of the data by testing the file's checksum values.

L0002 The assembler does not trap all cases where parameter values are too large and those it does trap are only indicated by the message "Syntax error".

L1003 The assembler and disassembler only handles the official Zilog documented Z80 instructions.

To do list

T0001 Fix the bugs!

T0002 Consider addressing the Quirks and Limitations

T0003 Expose more functionality via the API

History

This section documents the release history of the Small Computer Monitor.

Date	*Version*	*Description*
2017-11-18	v0.2.1	Preview release, needs testing.
2017-12-12	v0.3.0	Program and documentation released. Main changes: Added commands FILL and CONSOLE Added support for SC101 computer Changed/added text messages Added simple self-test Added support for SIO/2 channel B Added support for up to four console devices Added support for original SIO/2 addressing (untested) Added extra API functions Added extra operand formats to assembler Added decimal format (+n) to memory editor Single letter commands no longer need a space Added checksum test to Hex file download Many changes and improvements behind the scenes
2018-01-01	v0.4.0	Program and documentation released. Main changes: Added monitor command API to call API functions Fixed quirk Q0001 (assembler handling of RST instruction) Extended console devices from 4 to 6 Added feature to select console in and out separately Add “.” {return} to abort assembly and memory edit Added background event handling and supporting APIs Added APIs for I/o port functions Changed console output to return if device is busy Other changes and improvements behind the scenes Support for original SIO/2 addressing now tested

Future Plans

It is my intention to continue development of the Small Computer Monitor; to fix bugs, add additional hardware support and increase functionality.

Possible future additions:

- Conditional breakpoints
- Logging of events just prior to breakpoint
- Simple filing system
- Command to launch operating systems such as CP/M
- Command to launch other ROM based software such as BASIC
- Scripting language to write small programs
- Support for more hardware eg. Z80 CTC
- Interrupt driven I/O (option)

Contact Information

If you wish to contact me regarding the Small Computer Monitor please use the contact page at www.scc.me.uk (or smallcomputercentral.wordpress.com).

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

Stephen C Cousins, Chelmsford, Essex, United Kingdom.