Small Computer Monitor

User Guide

Monitor version 1.3 for the Z80/Z180 CPU

Edition 1.3.0, 22 March 2022

CONTENTS

Overview

The Small Computer Monitor (SCM) 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.

SCM 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 SCM can also be compiled, thus allowing it to be run from a storage device rather than from ROM.

The standard releases of SCM are 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.

SCM can be extended by adding extra 'Apps' in the ROM using the SCM's ROM filing system. These extensions are automatically integrated into the Monitor. The ROM filing system works across multiple ROM banks on systems where ROM banks can be selected in software.

The original release of SCM is v1.0 and in all cases it fitted in 8k bytes of ROM space. Later versions have, so far, kept the same basic functionality but the BIOS section has been extended to allow support of more hardware devices.

This document describes SCM as configured for any of the following systems:

- LiNC80 (available in kit form)
- RC2014 (available in kit form)
- Z50Bus (available in kit form)

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 SCM.

The primary serial port is usually set for 115200 bits per second (assuming the clock is 7.3728 MHz), 8 data bits, 1 stop bit, no parity, no flow control (hardware flow control is optional and system dependent). 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.

LiNC80 systems

Currently only v1.0 is available for the LiNC80. This is not really a problem as version 1.3 does not have any significant extra functions other than supporting more hardware devices.

The LiNC80 is an open system and can thus have diverse hardware, although the main board is pretty well equipped on its own. SCM is designed to work with the official modules, and may not work with third party modules.

The standard ROM for the LiNC80 contains SCM, Grant Searle's adaptation of Microsoft BASIC and a CP/M loader for Compact Flash. This all fits in the first ROM bank (16k bytes). In the second ROM bank it has a version of Grant Searle's monitor/loader system for BASIC and CP/M.

The default console device is usually the second serial port (SIO B) and it is usually initialised to 115200 baud, 8 data bits, 1 stop bit, no parity, hardware flow control. The other serial port (SIO A) is usually initialised to 9600 baud, 8 data bits, 1 stop bit, no parity, hardware flow control.

The default console device can be changed by writing the required device number to locations 0x0040 of the ROM.

The default baud rates can be changed by writing the required baud rate codes to locations 0x0041(SIO port A) and 0x0042 (SIO port B) of the ROM. The SIO clock source jumpers must be set to the CTC in order for these baud rates to be used.

SCM for LiNC80 does not use Interrupts.

RC2014 systems

Currently only v1.0 is available for distribution with official RC2014 systems. Version 1.3 is available for third party systems and anyone wishing to upgrade their official systems. Note that some RC2014 systems have a fixed 8k ROM bank size and can not support the larger v1.3 code size. This is not really a problem as version 1.3 does not have any significant extra functions other than supporting more hardware devices.

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

SCM is supplied in a number of different configurations.

Configuration R1 contains just SCM and runs on all standard RC2014 kits (Mini, Classic, Plus, and Pro). It fits in an 8k byte ROM that is mapped into memory from address $0000 to $1FFF. It requires RAM from $FD00 to $FFFF, leaving the rest free for the user. The ROM is not paged out of memory during operation.

Configuration R2 contains SCM and Grant Searle's adaptation of BASIC. It requires ROM from $0000 to $3FFF (16k bytes) and RAM from $4000 to $FFFF (48k bytes). If installed on a paged ROM board, the board must be set for a 16k byte page size. If used with a 64k byte RAM board, RAM must start at 0x4000. Paging is not enabled, as this configuration requires RAM from $4000 to $7FFFF.

Configuration R3 contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. It requires ROM from $0000 to $7FFF (32k bytes) and RAM from $8000 to $FFFF (32k bytes). To run CP/M the ROM must be paged out and RAM must be from $0000 to $FFFF (64k bytes). If installed on a paged ROM board, the board must be set for 32k byte page size. If used with a 64k byte RAM board, the RAM board should have paging enabled to allow CP/M to run using SCM's CP/M loader. The ROM has space for additional 'Apps' and is the only configuration that can be made available with SCM v1.3.

Configuration R4 contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. It requires ROM from $0000 to $3FFF (16k bytes) and RAM from $8000 to $FFFF (32k bytes). To run CP/M the ROM must be paged out and RAM must be from $0000 to $FFFF (64k bytes). If installed on a paged ROM board, the board must be set for 16k byte page size. If used with a 64k byte RAM board, the RAM board should have paging enabled to allow CP/M to run using SCM's CP/M loader.

Configuration S2 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z80 processors with at least 64k of RAM, paged in by port $38 bit 0. It also supports a second bank of 64k RAM selected by port $30 bit 0, and a 9600 baud bit-bang serial port at ports $20 and $28.

Configuration S3 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z80 processors with at least 64k of RAM, paged in by port $38 bit 0. It also supports a second bank of 64k RAM selected by port $38 bit 7.

Configuration S4 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z180 processors but only uses the Z180 as a Z80 replacement. It does not support the Z180's serial ports and memory management unit.

Configuration S5 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z180 processors and supports the Z180's serial ports and memory management unit. The target hardware is a modular RC2014 system with a Z180 and typically 512k ROM and 512k RAM. Eg. SC111 Z180 processor plus SC119 Z180 memory module.

Configuration S6 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z180 processors and supports the Z180's serial ports and memory management unit. The target hardware is SC126 or similar.

The default console device is the first serial port detected and it is usually initialised to 115200 baud, 8 data bits, 1 stop bit, no parity, no flow control.

The default console device can be changed by writing the required device number to locations 0x0040 of the ROM.

SCM for RC2014 does not use Interrupts.

Z50Bus systems

The Z50Bus is an open system and can thus have diverse hardware. SCM is designed to work with the official modules, and may not work with third party modules.

Configuration F1 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z80 processors with at least 64k of RAM, paged in by port $38 bit 0. It also supports

a second bank of 64k RAM selected by port $30 bit 0, and a 9600 baud bit-bang serial port at ports $20 and $28.

Configuration F2 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z80 processors with at least 64k of RAM, paged in by port $38 bit 0. It also supports a second bank of 64k RAM selected by port $30 bit.

Configuration F3 occupies 32k bytes of ROM and contains SCM, Grant Searle's adaptation of BASIC, and a CP/M loader for Compact Flash. This configuration is for Z180 processors and supports the Z180's serial ports and memory management unit.

The default console device is the first serial port detected and it is usually initialised to 115200 baud, 8 data bits, 1 stop bit, no parity, no flow control.

The default console device can be changed by writing the required device number to locations 0x0040 of the ROM.

SCM for Z50Bus 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 a numeric character. The exception to this rule is some operands in the assembler need a prefix.

? 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)
Configuration S6 20220227, Monitor 1.3.0, SCZ180 BIOS 1.3.0

Monitor commands:
A [<address>]  = Assemble          |  D [<address>]   = Disassemble
M [<address>]  = Memory display    |  E [<address>]   = Edit memory
R [<name>]     = Registers/edit    |  F [<name>]      = Flags/edit
B [<address>]  = Breakpoint        |  S [<address>]   = Single step
I <port>       = Input from port   |  O <port> <data> = Output to port
G [<address>]  = Go to program
BAUD <device> <rate>               |  CONSOLE <device>
FILL <start> <end> <byte>          |  API <function> [<A>] [<DE>]
DEVICES, DIR, HELP, RESET
BASIC    Grant Searle's adaptation of Microsoft BASIC
WBASIC   Warm start BASIC (retains BASIC program)
CPM      Load CP/M from Compact Flash (requires prepared CF card)
```

The configuration identifier, ’s6’ 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 options, such as different memory locations. So a ROM version may have a different identifier to a soft-loading version.

The most common major configuration identifiers, the alpha characters), are:

‘F’	Z50Bus compatible systems by Stephen C Cousins
‘L’	LiNC80 Standard ROM
‘R’	RC2014 Standard ROM
‘S’	RC2014 compatible systems by Stephen C Cousins
‘W’	Small Computer Workshop Simulator

Minor configuration identifiers, the numeric character, indicate the build variant.

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 ports

The API includes a set of functions to manage simple digital input and output ports, which are typically connected to switches and 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. In this example the port address used is $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 functions for handling 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 that 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 example 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 or Z180 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 B is also the name of the register B.

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 B with a zero clarifies it is intended to be a hexadecimal number.

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

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
Bad parameter
8014: 00          .      NOP                >
```

Baud rate setting

Syntax: BAUD <device identifier> <baud rate code>

This command enables the baud rate of a serial port to be set.

Not all systems have software selectable baud rates. Some have their baud rates set in hardware with jumpers, while others may have a single fixed rate.

The first parameter is the device identifier. This can be the console device number from 1 to 6, or the letter ‘A’ or ‘B’ for the two channels of a typical Z80 SIO device.

Device description	Identifiers	
Console device 1, typically SIO port A	$1	$A
Console device 2, typically SIO port B	$2	$B
Console device 3	$3	
Console device 4	$4	
Console device 5	$5	
Console device 6	$6	

The second parameter is the baud rate code. This can either be a number from $1 to $C, representing the 12 baud rate options, or it can be the first two digits of the baud rate, such as $96 for 9600 baud.

Baud rate	Rate codes	
230,400	$1	$23
115,200	$2	$11
57,600	$3	$57
38,400	$4	$38
19,200	$5	$19
14,400	$6	$14
9,600	$7	$96
4,800	$8	$48
2,400	$9	$24
1,200	$A	$12
600	$B	$60
300	$C	$30

If either parameter is invalid on the target system an error is reported.

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 "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 that is not there as a breakpoint or step instruction, program execution stops and "Trap" is displayed.

Console

Syntax: CONSOLE <device identifier>

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, with the first two often also having alternative identifiers of "A" and "B".

Device description	Identifiers	
Console device 1, typically SIO port A	$1	$A
Console device 2, typically SIO port B	$2	$B
Console device 3	$3	
Console device 4	$4	
Console device 5	$5	
Console device 6	$6	

The Monitor sets a suitable default console device at reset. However, this can be changed in the ROM by writing the required device number ($01 to $06) to address $0040.

These devices are typically serial ports, but they can be any character based input and output devices.

Device 1 is the first serial port, typically SIO port A.
Device 2 is the second serial port, typically SIO port B.
Device 3 is the third serial port, etc.

The Hardware section of this guide details the supported ports and their console device numbers.

If you have two serial ports with a terminal connected to each, you can swap between them with the commands "Console 1" and "Console 2" (or "Console 3" depending on your hardware configuration).

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 when it started up. The list will vary with different configurations of SCM.

For example:

```
Devices {return}
Supported devices:
  = Z180 ASCI       @ C0  detected
  = Z80 SIO (rc)    @ 80
  = Z80 SIO         @ 80
  = Z80 SIO (rc)    @ 84
  = Z80 SIO         @ 84
  = Z80 CTC         @ 88
  = Z80 CTC         @ 8C
  = ACIA            @ 80
  = ACIA            @ 40
  = CF Card         @ 10
  = Diagnostic LEDs @ 0D  detected
Console devices:
1 = Z180 ASCI       @ C0
2 = Z180 ASCI       @ C0
```

SCM can detect the presence of some hardware devices and includes driver software to support them.

The real reason for this feature is to support a range of serial interfaces and the variety of configurations found on modular and expandable computer systems. By detecting the presence of these devices SCM can configure itself to use whichever devices are available and thus avoid the need for multiple versions of this program.

In addition to using whichever console device is detected for its own input and output, SCM passes on this benefit to any software that makes use to the SCM's API.

When a serial device is detected SCM initialises it as the system's default console device, usually to 115200 bits per second (assuming the clock is 7.3728 MHz), 8 data bits, 1 stop bit, no parity, flow control is system dependent , no interrupts. If additional serial ports are detected they are initialised, but is not used unless the user selects them as the console device. These ports are therefore free to be reconfigured and used as required.

Directory file list

Syntax: DIR
Or syntax: ROM

SCM can include a ROM filing system. This command lists the files found in the ROM.

For example:

```
DIR {return}
Monitor    .EXE
BASIC      .COM
WBASIC     .COM
BASIC      .HLP
CPM        .COM
CPM        .HLP
```

If the system provides multiple banks of ROM, SCM searches them all for files.

The file types are:

- COM Commands run from the Monitor's command line
- DAT Unspecified file contents not used by the Monitor
- EXE Executable programs run from the Monitor's command line
- HLP Help text appended to the Monitor's help display
- TXT Text file not currently used by the Monitor

Command (COM) files are used to extend SCM. These can not normally be used by other installed software as they usually depend on the presence of the SCM. Due to their dependence on SCM they must either be in the same ROM bank or be relocated to RAM before being run.

Executable (EXE) files can be run from SCM's command line, but they do not otherwise make use of SCM's facilities. They are standalone programs that can run without SCM and thus can be started from other installed software. Software in ROM that can be used to "boot" the system, such as SCM itself, are EXEcutable files.

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 that 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 and Z180 instructions and uses standard Zilog mnemonics. Invalid op-code sequences are displayed as “????”.

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 that 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 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             > "Hello
8047: 00          .     NOP             > ^
8046: 70          p     LD (HL),B       > "o
8047: 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 [<name>]

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-
```

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 LiNC80 digital I/O card can typically read the switch inputs with the command:

```
I 30 {return}
```

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

```
I 0 {return}
```

Users of a Z50Bus digital I/O card can typically read the input port with the command:

```
I A0 {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 5 {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 LiNC80 digital I/O card can typically write to the LED outputs with the command:

O 30 5 {return}

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

O 0 5 {return}

Users of a Z50Bus digital I/O card can typically write to the LED outputs with the command:

O A0 5 {return}

In the above examples 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.

SCM 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 to be executed, and the resulting processor registers and flags to be displayed.

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 SCM'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 that 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

SCM includes a means of receiving an Intel Hex File from a console device, typically a 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 SCM.

Below is an example of an Intel Hex File.

```
:1040000021354006090E80EDB3061A3E41F5DB80EE
:10401000CB5728FAF1D3813CCD2A4010F0DB80CB7E
:104020004728FADB81D38118F4C9F5C50E0911645C
:1040300000F7C1F1C91814C403C105681100FFFFDE
:00000001FF
```

When SCM 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.

It is not usually necessary 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

SCM has a simple self-test feature that runs at reset.

To see the output of the test you need a status display port. This is usually a simple output port with 8 LEDs connected. 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.

The status display port for the LiNC80 is the optional digital I/O module, or equivalent, and this must have its output port set to the default address of $30.

The status display port for the RC2014 is the optional digital I/O module, or equivalent, and this must have its output port set to the default address of $00.

The status display port for a Z50Bus system is the optional digital I/O card, or equivalent, and this must have its output port set to the default address of $A0.

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 status display port 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 the few locations essential for SCM to operate, 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 a console device, typically a serial port. 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 the code correctly 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 status display port are present.

Some systems have a single status LED. In this case it will turn on a reset, then flash off and back on if the test is successful. In addition, the LED might flash off and on again a second time to indicate that an optional console device has not been found.

ROM Filing System

SCM supports a simple read only Filing System designed to allow data and program files to be stored in ROM.

This is particularly useful on systems with multiple ROM images in software selectable ROM banks. SCM, or any other compliant software, can scan all available ROMs and locate any program and data files.

To achieve this ROMs need to include a table of contents. Any ROM without this table will not be able to share its files.

The table is stored at the top of each ROM or ROM bank, and contains one or more file references. The first reference is stored in the top 16 bytes, the second reference in the 16 bytes below that, and so on. The end of the table is determined by the absence of the next logical reference.

For a system with multiple ROM banks, each 16k bytes long and starting at address 0x0000, the first file reference in the each bank is at 0x3FF0, the second at 0x3FE0, and so on.

Each file reference has two identifier bytes providing a fairly reliable way of determining if the block of 16 bytes is a file reference. Further checks can be made by testing all eight bytes of the file name for valid characters (ASCII 0x20 to 0x7F).

The file name contains eight ASCII characters. File names shorter than eight characters long have the unused trailing bytes filled with ASCII spaces (0x20). The file name can contain characters A to Z, a to z, 0 to 9, underscore and hyphen. File names must be at least two characters long so as not to conflict with single letter monitor commands.

Each file reference also includes: file type, file attribute flags, file location, and file length.

File reference

Each file in the ROM Filing System requires a file reference as described below.

File reference: (16 bytes)

Offset in reference	Contents
+0x00	File reference identifier byte 0x55
+0x01	File reference identifier byte 0xAA
+0x02	File name, 8 characters, padded with trailing spaces
+0x0A	File type and flags byte
+0x0B	File destination byte (upper byte of address, lower is zero)
+0x0C	File start address, 16-bit, low byte first
+0x0E	File length, 16-bit, low byte first

File type and flags byte:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	BIt 0
Auto	Move	0	0	File type (0 to 15)			

File types:

Type	Extension	Contents
0	DAT	Unspecified data (no action taken by SCM)
1	COM	Command file that requires SCM to run
2	EXE	Executable code that does not use SCM
3	HLP	Help text displayed by SCM
4	TXT	Text (no action taken by SCM
5 - 15	-	Not currently defined

Flag bits:

Bit	Name	Function
4	-	Not currently used, should be zero
5	-	Not currently used, should be zero
6	Move	Code is moved to RAM before being executed
7	Auto	Auto-run flag indicates the program is run at reset

File type and flags byte

The file type and flags byte specifies the file type (0 to 15) and four flags indicating special attributes of the file.

File type 0 - Unspecified data

This file contains data in an unspecified format that SCM takes no action with.

File type 1 - Command file that requires SCM to run

The COM file type is for SCM extensions and applications that make use of the SCM's functionality. An example of a COM files is the version of BASIC included with some configurations of SCM. This BASIC makes use of SCM's console I/O support (eg. Serial ports) and so will not run without SCM.

File type 2 - Executable code that does not use SCM

The EXE file is for programs that run independently of SCM. These programs are totally self contained and can be launched by any other software that is able to scan the ROM Filing System and take the necessary action.

File type 4 - Help text displayed by SCM

This file type contains ASCII text which is is appended to SCM's help.

Flag bit 6 - Move

An executable or command file that has this bit set is moved to RAM before being executed. The file destination byte provides the upper byte of the destination address, while the lower byte is always zero. Moving code to RAM is usually necessary if it is from a paged ROM bank and requires access to code or data in another ROM bank. A COM file in a different paged ROM bank to SCM will need to be moved to RAM in order to run, as COM files make use of functionality in SCM.

Flag bit 7 - Auto

The Auto-run flag indicates that the executable or command file should be run at reset. SCM runs each file it finds with the Auto flag set. These are run just after the SCM initialises its own memory and drivers, and just before the "Small Computer Monitor" start-up message is displayed.

The following example shows how to include a CPM loader command.

```
; Command code: CPM.COM
; This code is written to run at 0x8000
CPMCode:
#INSERTHEX  ..\Apps\CPM_loader\SCM_CPM_loader_code8000.hex
CPMCodeEnd:


; Help extension: CPM.HLP
CPMHelp:
    .DB  "CPM     Load CP/M from Compact Flash"
    .DB  0x0D,0x0A
    .DB  0
CPMHelpEnd:


    .ORG 0x7FE0     ;File references downwards from 0x7FF0

    .DW  0xAA55     ;Identifier
    .DB  "CPM    "  ;File name ("CPM.HLP")
    .DB  0x03       ;File type 3 = Help
    .DB  0          ;Not used
    .DW  CPMHelp    ;Start address
    .DW  CPMHelpEnd-CPMHelp ;Length

    .DW  0xAA55     ;Identifier
    .DB  "CPM    "  ;File name ("CPM.COM")
    .DB  0x41       ;File type 1 = Monitor command, moved to RAM
    .DB  0x80       ;Run in RAM at 0x8000
    .DW  CPMCode    ;Start address
    .DW  CPMCodeEnd-CPMCode ;Length
```

BASIC

Grant Searle's popular adaptation of classic Microsoft BASIC can run on all the retro computer systems currently supported by SCM.

The BASIC interpreter can either be in the same ROM as SCM, or downloaded to RAM using a terminal program. Some configurations of the SCM ROM include BASIC, others do not. Use SCM's Help command to see if BASIC is included.

There are currently four builds of BASIC. The choice of which build to use depends on the hardware configuration. In each case the BASIC interpreter is the same, it is only the memory map which differs.

Build (file name)	Code (ROM/RAM)	Data (RAM)
SCMon_BASIC_code2000_data4000	0x2000 to 0x3FFF	0x4000 to 0xFBFF
SCMon_BASIC_code2000_data8000	0x2000 to 0x3FFF	0x8000 to 0xFBFF
SCMon_BASIC_code3000_data8000	0x3000 to 0x4FFF	0x8000 to 0xFBFF
SCMon_BASIC_code8000_dataA000	0x8000 to 0x9FFF	0xA000 to 0xFBFF

The first three are most likely to be in ROM along with SCM, while the fourth is likely to be used as a download to RAM. If none of these are optimal, then the Small Computer Workshop program can be used to build a more suitable configuration.

In each case SCM supplies the required hardware support. One of the advantages of running from SCM is that BASIC inherits SCM's range of hardware support, such as different serial interfaces, and also has access to SCM's API.

Starting BASIC

The LiNC80's standard ROM includes the Monitor and BASIC. The following brief guide to using BASIC assumes this is the set up you are using.

To start BASIC from the Monitor, type the command "BASIC" and press the Return key. You should see something like this:

```
BASIC {return}
Memory top?
```

The "Memory top" question allows you to enter the highest location in memory that BASIC will use, thus allowing space to be set aside for,say, machine code routines. If you just press Return, a suitable default is used.

You should then see something like this:

```
Z80 BASIC Ver 4.7b
Copyright (C) 1978 by Microsoft
47742 Bytes free
Ok
```

You can now program in BASIC like it's 1978!

There is plenty of support on the internet for this version of BASIC, so it is not being duplicated here.

To return to the Monitor from BASIC, use BASIC's "monitor" command.

To return to BASIC from the Monitor, without clearing the BASIC program, use the Monitor's "WBASIC" command. That's "W" for Warm start.

Using SCM's API from BASIC

In order to access SCM's API, the following steps are required.

- DOKE the API entry point address to memory
- POKE the API call number to memory
- POKE or DOKE call parameters to memory
- Call the API using the BASIC statement: N=USR(0)
- PEEK or DEEK resulting values from memory

This example shows how to read SCM's version number:

```
LIST {return}
10 DOKE &H4004,&H34: REM Set USR() call address *(see note below)
20 POKE &HFFF4,8: REM API function number 8
30 N=USR(0): REM call Monitor API
40 PRINT "Version ";PEEK(&HFFF7);".";PEEK(&HFFF6)
Ok

RUN {return}
Version  1 . 0
Ok
```

The table below shows the mappings of API registers to BASIC's memory locations, and memory access statements to access then.

Register(s)	Address	Write	Read
A	&HFFF3	POKE &HFFF3,<A>	<A> = PEEK(&HFFF3)
F	&HFFF2	POKE &HFFF2,<F>	<F> = PEEK(&HFFF2)
B	&HFFF5	POKE &HFFF5,<B>	<B> = PEEK(&HFFF5)
C	&HFFF4	POKE &HFFF4,<C>	<C> = PEEK(&HFFF4)
D	&HFFF7	POKE &HFFF7,<D>	<D> = PEEK(&HFFF7)
E	&HFFF6	POKE &HFFF6,<E>	<E> = PEEK(&HFFF6)
H	&HFFF9	POKE &HFFF9,<H>	<H> = PEEK(&HFFF9)
L	&HFFF8	POKE &HFFF8,<L>	<L> = PEEK(&HFFF8)
AF	&HFFF2	DOKE &HFFF2,<AF>	<AF> = DEEK(&HFFF2)
BC	&HFFF4	DOKE &HFFF4,<AF>	<BC> = DEEK(&HFFF4)
DE	&HFFF6	DOKE &HFFF6,<AF>	<DE> = DEEK(&HFFF6)
HL	&HFFF8	DOKE &HFFF8,<AF>	<HL> = DEEK(&HFFF8)
USR() address*	&H4004 &H8004 &HA004	DOKE &H4004,&H34 DOKE &H8004,&H34 DOKE &HA004,&H34	<n> = DEEK(&H4004) <n> = DEEK(&H8004) <n> = DEEK(&HA004)

* The choice of address will depend on the build of BASIC used.
In each case the address is the start of the data area plus 4.

CP/M

CP/M was a very popular operating system in the late 70s and 80s, and was available for many different computers. Early versions of Microsoft's DOS were essentially a copy of CP/M.

Some configurations of the SCM include a CP/M loader. Use SCM's Help command to see if this feature is included.

The CP/M loader is just a loader. It is not CP/M. The loader looks for a Compact Flash card and attempts to load CP/M from that, in much the same way MS-DOS is loaded from disk. You therefore need a Compact Flash card with CP/M already installed on it.

Installing CP/M on a Compact Flash card is beyond the scope of this guide. Suppliers of retro computer kits provide documentation and a suitable version of CP/M. For homebrew systems a good source of information is Grant's Searle's website at www.searle.wales/

Assuming the CP/M loader is included in your configuration of SCM, and you have a Compact Flash with a suitable version of CP/M already installed on it, plus a Compact Flash interface on your retro system, then the command "CPM" should start CP/M.

```
Small Computer Monitor - LiNC80
*cpm

Z80 CP/M BIOS 1.0 by G. Searle 2007-13

CP/M 2.2 Copyright 1979 (c) by Digital Research

A>
```

Because CP/M is a complete operating system, it does not use SCM at all once loaded. CP/M has its own device drivers and thus you need a version of CP/M to match your hardware. If SCM works on your hardware, but CP/M does not, it is most likely to be due to an incompatible version of CP/M or a faulty installation of CP/M.

There is plenty of information on the internet about CP/M, so it is not being duplicated here.

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
- $21 Set baud rate
- $22 Execute command line
- $23 Get pointer to command line

- $24 Skip delimiter in command line
- $25 Skip non-delimiter in command line
- $26 Get hexadecimal parameter from command line
- $27 Get current console I/O device numbers
- $28 Get top of free memory
- $29 Set top of free memory
- $2A Read from banked RAM (see note)
- $2B Write to banked RAM (see note)

Note: All of the above API's have been present in all releases of SCM, except for $2A and $2B, which are not supported by all configurations of SCM v1.0.

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: 3E 00      LD A,$00     ; 0 = Cold start, 1 = Warm start
8002: 0E 00      LD C,$00     ; Function 0 = System reset
8004: 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: 3E 00      LD A,$00     ; 0 = Cold start, 1 = Warm start
8002: 0E 00      LD C,$00     ; Function 0 = System reset
8004: F7         RST 30       ; Call API
```

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

If SCM is not located at the bottom of memory, then the API call address is the start of SCM + $0030. Thus if SCM 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: A = Reset type
0 = Cold start monitor
1 = Warm start monitor
Returns: none

This function performs a system reset, either a cold start or a warm start.

A cold start is 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.

A warm start returns to the monitor prompt, but does not re-initialise hardware and workspace.

A cold start terminates interrupts and stops the user program. The monitor then restarts. Memory is not cleared, but essential variables are initialised.

The following example is equivalent to SCM's RESET command.

```
8000: 3E 00      LD A,$00     ; Reset type = Cold start
8002: 0E 00      LD C,$00     ; Function 0 = System reset
8004: F7         RST 30       ; Call API
```

SCM 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 if an input character is available from the 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 = Maximum 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 line:

```
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 A are the monitor source code version
D = Major, E = Minor, A = Revision
B, C are the configuration identifier
B = Major configuration identifier (ASCII character)
Letters are official release configurations
Numbers are development, user or custom
C = Minor configuration identifier (ASCII character)
Numbers 1 to 9 are official release configurations
Number 0 is a user or custom configuration
H and L are the target hardware ID
H = Primary ID, L = Options

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

The monitor source code version is detailed in registers D, E and A. Register D contains the major version number, E contains the minor version number, and A the revision number. For example, if D=$01, E=$02 and A=$03, the version is 1.2.3

The configuration identifier characters indicate 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.

The most common major configuration identifiers currently assigned are:

F = Z50Bus compatible systems
L = LiNC80 SBC1
R = Official RC2014 systems
S = Systems designed by Stephen C Cousins
W = SCWorkhop simulator

All other upper case letters are reserved for official configurations of SCM.

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
- H = 5 LiNC80 SBC1
- H = 6 Tom's SBC
- H = 7 Bill Shen's Z280RC
- H = 9 Bill Shen's Z80PG
- H = 10 Steve Markowski's ZORAk
- H = 11 John Squire's Z80 Playground
- H = 21 Z50Bus system with Z80 CPU and bit-bang serial port
- H = 22 Z50Bus system with Z80 CPU but no bit-bang serial port
- H = 23 Z50Bus system with Z180 CPU
- H = 32 RC2014 compatible with Z80 CPU but no bit-bang serial
- H = 33 RC2014 compatible with Z80 CPU and bit-bang serial
- H = 34 RC2014 compatible with Z180 CPU in Z80 replacement mode
- H = 35 RC2014 compatible with Z180 CPU in native mode
- H = 126 SC126, Z180 SBC/motherboard
- H = 130 SC130, Z180 SBC/motherboard
- H = 131 SC131, pocket-sized Z180 system

Target hardware options indicate which devices have been detected:

- L, bit 0 ACIA #1 at primary ACIA address
- L, bit 1 SIO #1 at primary SIO address
- L, bit 2 ACIA #2 at secondary ACIA address
- L, bit 3 Bit-bang serial at 0x20 + 0x28
- L, bit 4 CTC #1 at primary CTC address
- L, bit 5 SIO #2 at secondary SIO address
- L, bit 6 CTC #2 at secondary CTC address
- L, bit 7 Other console interface (eg. Z180's serial port)

This method of identifying hardware options has run out of bits for the growing range of supported hardware!

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, console character output
- $02 Restart $10, console character input
- $03 Restart $18, console input status
- $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-maskable interrupts. This is because code for these interrupts must begin at fixed locations in memory that 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, which are used for console input and output, and Restart $20, which is 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 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.

This function assumes a processor clock speed is 7.3728 MHz. If the speed is different from this then the delay times will be proportionally shorter or longer.

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 SCM to the console output device.

The following messages are available:

System messages

- $00 Message = None (null)
- $01 Message = “Small Computer Monitor ”
- $02 Message = “Devices detected:”
- $03 Message = <About this version of the monitor>
- $04 Message = <List of hardware devices detected>
- $05 Message = “Ready”
- $06 Message = “File error”

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”

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

The following example displays information about SCM.

```
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 1.0.0 configuration R1 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

SCM 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 that 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 usually set as the default console device at reset. However, this can be changed in the ROM by writing the required device number ($01 to $06) to address $0040.

Device 1 is usually the first serial port.

The Hardware section of this guide details the supported ports and their console device numbers.

If you have two serial ports with a terminal connected to each, you can swap between them with this function (or with the Console command).

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 succeeds NZ flagged and A != 0
If character output fails: 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

SCM supports a system of timer events that 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 SCM 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. Small computers often don't 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.

Fortunately the LiNC80 does have a hardware timer (a Z80 CTC) so event times are usually pretty accurate. However, standard RC2014 systems do not.

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 that runs for more than a few milliseconds. Even doing this the timer events will be very erratic on systems without a hardware timer.

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 = Timer events enabled
Returns: none

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

Timer events are explain above (API function $12).

Following a processor reset the event mode is zero, so SCM 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

SCM 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 on the current digital output port:

```
8000: 3E 01      LD A,1        ;Output 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 repeatedly toggle the output bit, 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 Z50Bus, LiNC80, or 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 Z50Bus, LiNC80, or RC2014 digital I/O module's input 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)

API function $21, set baud rate

Parameters: A = Device identifier ($1 to $6, or $A or $B)
E = Baud rate code
Returns: A = 0 if either parameter is invalid on the target system

Not all systems have software selectable baud rates. The LiNC80 does have software selectable baud rates, while standard RC2014 systems do not.

Parameter A is the device identifier. This can be the console device number from $1 to $6, or $A or $B for the two channels of a typical Z80 SIO device.

Device description	Identifiers	
Console device 1, typically SIO port A	$1	$A
Console device 2, typically SIO port B	$2	$B
Console device 3	$3	
Console device 4	$4	
Console device 5	$5	
Console device 6	$6	

Parameter E is the baud rate code. This can either be a number from $1 to $C, representing the 12 baud rate options, or it can be the first two digits of the baud rate, such as $96 for 9600 baud.

Baud rate	Rate codes	
230,400	$1	$23
115,200	$2	$11
57,600	$3	$57
38,400	$4	$38
19,200	$5	$19
14,400	$6	$14
9,600	$7	$96
4,800	$8	$48
2,400	$9	$24
1,200	$A	$12
600	$B	$60
300	$C	$30

API function $22, execute command line

Parameters: DE = Pointer to command line string
Returns: A = 0 and Z flagged if command recognised and executed

The specified command line is executed.

Register pair DE points to a null terminated string, which the monitor attempts to execute. Any monitor command can be specified.

The following example executes a memory fill command.

Write a null terminated monitor command string to memory:

```
9000:  46 49 4C 4C 20 38 30 30  30 20 38 46 46 46 20 46  FILL 8000 8FFF F
9010:  46 00 55 55 55 55 55 55  55 55 55 55 55 55 55 55  F.UUUUUUUUUUUUU
```

Enter this code:

```
9100: 11 00 90     ...   LD DE,$9000
9103: 0E 22        ."    LD C,$22
9105: F7           .     RST 30
9106: C9           .     RET
```

Memory before executing the code:

```
m 8100 {return}
8100:  55 55 55 55 55 55 55 55  55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
8110:  55 55 55 55 55 55 55 55  55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
8120:  55 55 55 55 55 55 55 55  55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
8130:  55 55 55 55 55 55 55 55  55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
```

Execute the code:

```
G 9100 {return}
```

Memory after executing the code:

```
m 8100
8100:  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
8110:  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
8120:  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
8130:  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
```

API function $23, get pointer to command line

Parameters: none
Returns: DE = Pointer to current location in command line string

API function $24, skip delimiter in command line

Parameters: none
Returns: DE = Pointer to current location in command line string

API function $25, skip non-delimeters in command line

Parameters: none
Returns: DE = Pointer to current location in command line string

API function $26, get hexadecimal parameter from command line

Parameters: A = Error reporting flags
Bit 0 set reports bad hexadecimal parameters
Bit 1 set reports missing parameter
DE = Location of parameter in input line
Returns: If a valid hexadecimal parameter if found:
A = $00 and Z flagged and C flagged
DE = Pointer to current location in command line string
HL = Value of hexadecimal parameter
If an invalid hexadecimal parameter is found:
A = $FF and NZ flagged and NC flagged
DE = Pointer to current location in command line string
If no parameter is found:
A = $00 and Z flagged and NC flagged
DE = Pointer to current location in command line string

The group of functions is designed to allow a Monitor extension App to read its parameters from the command line that launched it.

Monitor extension Apps are included in the ROM and integrate themselves into the Monitor. The BASIC interpreter is an example of a Monitor extension App.

Providing the App is in the same ROM bank as the Monitor, or is relocated to RAM before running, it can use these APIs to read command line parameters. Such Apps are listed as ".COM" programs, where "COM" stands for Monitor COMmands.

API function $27, get current console device numbers

Parameters: none
Returns: D = Current console output device (1 to 6)
E = Current console input device (1 to 6)

The currently selected console input and output device numbers are returned.

The input and output device will usually be the same, but they can be different.

Typically this function returns D = 1 and E = 1, indicating the current console device for both input and output is device 1 (usually the first serial port).

API function $28, get top of free memory

Parameters: none
Returns: DE = Highest location not in use by SCM

API function $29, set top of free memory

Parameters: DE = Highest location not in use by SCM
Returns: none

These functions ($28 and $29) enable the top of free memory to be determined and also to be changed.

The top of free memory is the highest location in RAM that is not being used by SCM. Typically this will be $FBFF.

Any software that requires some private memory can lower this location with the “set top of free memory” API, thus claiming some space. A device driver may wish to do this.

API function $2A, read from banked RAM

Parameters: DE = Address in banked RAM
Returns: A = Byte read from RAM

API function $2B, write to banked RAM

Parameters: A = Byte to be written to RAM
DE = Address in banked RAM
Returns: A = Byte written to RAM

Functions $2A and $2B provide a means of reading and writing data in banked RAM.

Some systems have more than 64k bytes of RAM and this can usually be accessed by swapping banks of RAM in and out of the processor's 64k byte address space.

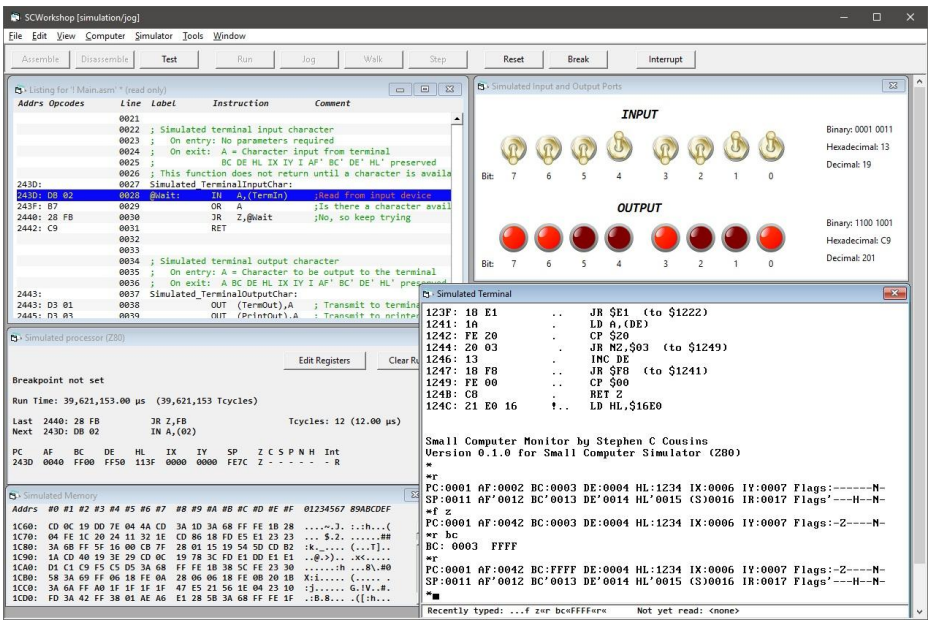
These functions are limited to a total of 64k bytes of RAM that can be swapped into the processor's address space. Typically, this is mapped as a single 64k bank, with no common memory between it and the main 64k RAM. With this simple scheme these functions only allow access to the top 32k of the bank RAM. More sophisticated schemes could, for example, have four 16k byte banks.

Source Code and Assembly

The source code for the Small Computer Monitor is freely available. Assembling it requires the Small Computer Workshop program, which is provided with the Monitor source code.

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.



The Small Computer Workshop program is available for download, but be warned it is not finished, not everything works and it is rather buggy. Having said that it is still the easiest way to modify the Small Computer Monitor.

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 that 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.

Actually, to be totally honest the decision was largely due to my desire to write something from scratch to my own design. So I reinvented another wheel.

The assembler has a 'C' style pre-processor that 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 "$" or "0x".

Customising the Small Computer Monitor

To fully customise the Small Computer Monitor it is necessary to modify the source code and assemble a new version. This is explained on the Small Computer Central website.

However, it is possible to make small changes to the hex file before programming the ROM, or to the image before running in RAM.

Location $0040 contains the default console device number. This can be any device from $01 to $06. To select device 3 as the default console device, change this location from its typical value of $01 to $03. The DEVICES command lists the available devices. See commands CONSOLE and DEVICES for more details.

Location $0041 contains the baud rate code for console device 1, which is typically port A of a Z80 SIO. The valid baud rate codes are listed in the monitor commands section for the Baud command, and in the API section for the Set baud rate API. Not all hardware supports software selectable baud rates.

Location $0042 to $0046 contain the baud rate code for console devices 2 to 6.

Memory Map

SCM'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 that have fixed uses and can not be moved.

The standard build places SCM code starting at $0000 and currently extends to about $2500. Workspace and stack is at the very top of memory from about $FC00 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 SCM ROM or be in RAM.

These Z80 specific fixed locations are:

$0000	Restart 00, reset address
$0008	Restart 08
$0010	Restart 10
$0018	Restart 18
$0020	Restart 20
$0028	Restart 28
$0030	Restart 30
$0038	Restart 38, interrupt mode 1 address
$0066	Non-maskable interrupt address

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

With SCM code starting at 0x0000, these locations have fixed uses:

$0000	Restart 00, SCM cold start	(CP/M 2 warm boot)
$0003	Warm start SCM	(CP/M 2, IOBYTE, drive/user)
$0005	Reserved for FDOS	(CP/M 2, FDOS entry point)
$0008	Restart 08, console character output	
$000C	Cold start SCM	
$0010	Restart 10, console character input	
$0018	Restart 18, console input status	
$000C	Null terminated string "SCM"	[added at SCM v1.3]
$0020	Restart 20, not currently used	
$0028	Restart 28, use for debugging breakpoint	
$002C	SCM info, SCM identifier, major	[added at SC v1.3]
$002D	SCM info, SCM identifier, minor	[added at SC v1.3]

$0030	Restart 30, API entry point	
$0038	Restart 38, interrupt mode 1 handler	
$0040	SCM customisation: Initial console device ($01 to $06)	
$0041 - 6	SCM customisation: Initial baud rate for console devices 1 to 6	
$0047	SCM customisation, initial status input port	
$0048	SCM customisation, initial status output port	
$0049	Not currently used, to $004C	
$004D	SCM info, top of ROM filing system (hi byte)	
$004E	SCM info, start of SCM code (hi byte)	
$004F	SCM info, end of SCM code (hi byte)	
$0050 - B	BIOS specific configuration data/options	
$005C	Not currently used, to $0065	(CP/M 2, default FCB, to $007F)
$0066	Non-maskable interrupt handler	(CP/M 2 default FCB)
$0080	Start of available code space	(CP/M 2, DMA buffer to $00FF)

Target Hardware

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

SCM 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 SCM 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.

SCM makes use of one optional hardware feature; a simple input/output port, with the output assumed to have 8 LEDs connected. The output port is used for the self-test status display. The input and output ports are used as the default ports for the API's port functions.

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

Currently, SCM's full feature set occupies about 9k bytes of code space and requires about 1000 bytes of RAM (including stack space). The code size can be significantly reduced by excluding features like the in-line assembler. SCM v1.0 is still the best code base for anyone wanting full feature set in 8k bytes. Later versions take up more space as they have a flexible BIOS enabling better device support. Additional functionality is also anticipated in the future, which will take up more space.

Below are details of some of the hardware types supported by SCM. There are now too many to go into detail about all of them. Futher information is available online.

Hardware type 1: Small Computer Workshop 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

Type 3 hardware covers official RC2014 systems and compatibles using the Z80 processor. Official RC2014 modules are available from “RFC2795 Ltd” 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, Pro, ZED) meet these requirements, SCM is compatible with all of them. All these kits provide a means of running SCM from ROM, whilst some also provide configurations suitable to run SCM in RAM.

Currently, SCM version 1.0 is the distribution version for official RC2014 systems. This is because v1.0 fits in the 8k ROM space that even the lowest specification systems provide. The standard RC2014 build is designed to run from ROM starting at address $0000. It is an 8 kbyte ROM image that supports the serial I/O modules listed below.

Serial modules supported:

- Official RC2014 Serial I/O Module (68B50 or 63B50 ACIA) or compatible
  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
  If present this is assigned as console device 1
- Official RC2014 Dual Serial Module (SIO/2) or compatible
  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
  If present this is assigned as console device 1 (port A) and 2 (port B)
- Dual Serial Module (SIO following Grant Searle’s register order)
  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
  If present this is assigned as console device 1 (port A) and 2 (port B)
- Official RC2014 Serial I/O Module, with modified address (68B50 or 63B50 ACIA) or compatible
  Base I/O address = $40, 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

SCM 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 the same base I/O address, at a time.

The optional status display port is at I/O address $00, which is typically the RC2014 digital I/O module. The output port of the digital I/O module has 8 LEDs connected, which light when the output is set to '1' (a logic high). This module is also used by default by the API port functions.

If SCM is unable to identify a suitable serial module at reset it turns On bit 0 LED of the status display port (digital I/O module) to indicate the problem.

Grant Searle's original SIO design has different register order 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.

Hardware type 5: LiNC80

Type 5 hardware covers LiNC80 systems using the Z80 processor. Information about the LiNC80 and its accessories, and links to the store page where kits can be purchased can be found at http://linc.no/go/linc80.

The optional status display port is at I/O address $30, which is typically the LiNC80 digital I/O module. The output port of the digital I/O module has 8 LEDs connected, which light when the output is set to '1' (a logic high). This module is also used by default by the API port functions.

If SCM is unable to identify the serial interface at reset it turns On bit 0 LED of the status display port (digital I/O module) to indicate the problem.

Hardware type 6: Tom's SBC

Type 6 hardware covers Tom Szolyga's Single Board Computer version C. This is a very small computer designed to run CP/M.

Tom's SBC has a Z80 processor, 64k RAM, 32 or 64k ROM, two serial ports via a Z80 SIO, and a Compact Flash card connected via an 8-bit IDE port provides storage. It does not have a general purpose expansion bus and does not have a status display port.

For more information about Tom's SBC visit https://easyeda.com/peabody1929.

Hardware types 21 to 23: Z50Bus

Z50Bus systems currently support Z80 or Z180 processor. Full details online.

Hardware types 32 to 35: Designs by Stephen C Cousins

These are mostly RC2014 compatible design. Full details online.

There are other supported hardware types. Too many to describe in detail here. For further details visit the Small Computer Central website.

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:

B0001 Fixed: 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 Fixed: 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 Fixed: 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 and Z180 instructions.

To do list

T0001 Fix the bugs!

T0002 Consider addressing any 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
2018-03-18	v0.5.0	Program and documentation released. Main changes: Fixed bug B0001 trap too large relative addresses Fixed quirk Q0002 assembler handling of JP (rr) Fixed limitation L0002 sensible error message

		Reduced assembler's need for hex prefixes Added support for second ACIA at address $40 Default console device set at $0040 in the ROM
2018-04-16	v0.6.0	Program and documentation released. Main changes: Added complete support for LiNC80 SBC1 Added console I/O support via RST $08, $10, $18 Added support for software selectable baud rates Added ROM filing system Added option BASIC and CP/M loader
2018-04-20	v0.6.2	Added support for Tom Szolyga's SBC version C
2018-04-28	v1.0.0	Program and documentation released. Program code very nearly the same as v0.6.2 Documentation has small changes only
2018-04-23	v1.0.0	Released SCM User Guide edition 1.1.0
2019-06-02	v1.0.1	Released SCM S5 for Z180 CPU. Core functionality unchanged.
2019-07-15	v1.1.0	Released SCM S6 for SC126 SBC/motherboard. Core functionality unchanged but BIOS enhanced.
2020-06-16	v1.2.0	Interim development release with new BIOS framework. DEVICES command now provides more information.
2022-02-15	v1.2.1	Added support for SC516.
2022-02-27	v1.3.0	Released with new BIOS framework and Z180 assembly and disassembly support (contributed by Ivan De La Fuente .
2022-03-07	v1.3.0	Released v1.3.0 source code with SCW v0.3.4
2022-03	v1.3.0	Released SCM User Guide edition 1.3.0

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
- Scripting language to write small programs
- Support for more hardware
- Interrupt driven I/O (option)

As this list has not changed in a couple of years it seems unlikely I'll achieve them all.

I think it is worth noting that SCM v1.0 is still the right version to use for some systems, particularly those with on 8k of ROM space. It is also the best code base for anyone wanting to create a custom version of SCM with a small code size.

Credits

The Small Computer Monitor started off as all my own work, written from scratch - just because it interested me. Following its first public release I have received feedback, encouragement, and help from many people in the retro computer community. Some have even gone as far as to port SCM to their own designs.

I'd like to thank all those who have taken the trouble to provide feedback and especially those who have contributed to the project. I would like to acknowledge everyone by name and describe their contribution but that is not realistic. However, I do think Jon Langseth (from LiNC, developer of the LiNC80 SBC1 and the Z50Bus) needs a mention here as he provided a great deal of feedback, help and support as SCM approached its first full release. To everyone else, I thank you.

Contact Information

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

Stephen C Cousins, United Kingdom.

LiNC80

Issues related to the LiNC80 can be posted on the google group “LiNC80”.
Information about the LiNC80 and its accessories, and links to the store page where kits can be purchased can be found at http://linc.no/go/linc80

RC2014

Issues related to the RC2014 can be posted on the google group “RC2014-Z80”.
Information about the RC2014 can be found at www.rc2014.co.uk
Kits are available from www.tindie.com

Designs by Stephen C Cousins

Information and support can be found at www.scc.me.uk (Small Computer Central).

Tom’s SBC

Issues relating to Tom’s SBC can be posted at
https://groups.google.com/forum/#!topic/rc2014-z80/IFkzQh3bHhY
Information can be found at https://easyeda.com/peabody1929

General Community Support

Google group: retro-comp