SUPERMON 816 MACHINE LANGUAGE MONITOR

Supermon 816 is a full-featured machine language monitor that is adaptable to most computers that are powered by the Western Design Center's W65C816S 16-bit microprocessor when operated in native mode. There is no support for 65C02 emulation mode operation.

Supermon 816 includes the following:

- ✓ 65C816 instruction assembly and disassembly;
- ✓ Memory compare, copy, dump, edit and fill functions;
- ✓ Microprocessor register dump and edit functions;
- ✓ Program execution functions;
- ✓ Number base conversion;
- ✓ Motorola S-record data loader.

Supermon 816 is a salute to Jim Butterfield (1936-2007), who wrote the SuperMon machine language monitor for the Commodore PET/CBM line of computers in the late 1970s, and subsequently adapted his code to the Commodore 64, naming that version SuperMon 64. Commodore themselves bundled SuperMon 64 with their Macro Assembler Development System (MADS) and a customized version of SuperMon was integrated with the Commodore 128. Fiscal Information Inc., developers of the Lt. Kernal hard drive subsystem for the C-64 and C-128, developed a disk editing tool called LKMON, using the SuperMon core.

In 2009, BCS Technology Limited began work on a custom 65C816-powered machine controller for a client and needed to integrate a machine language monitor into the controller's firmware for debugging and testing. Supermon 816 was the result.

Although Supermon 816 is not an adaptation of Jim Butterfield's work, it was decided to keep the SuperMon name alive, since Supermon 816's general operation and user interface is similar to that of the original SuperMon. Supermon 816 is 100 percent native mode 65C816 code and was developed from a blank canvas.

SYSTEM INTEGRATION

This verison of Supermon 816 will function on any 65C816 system that has a native mode operating environment and a VT-100 compatible console. A secondary (auxiliary) hardware input port is required in order to use the Motorola S-record loader. This section will explain what must be done to successfully integrate Supermon 816 with your hardware and operation system.

Supermon 816 is distributed as source code that is structured to assemble in the Kowalski 65C02 assembler/simulator. Prior to assembly, some editing of the source code must be performed in order to adapt Supermon 816 to the target system. It is assumed that you know how to edit and assemble source code files, and if necessary burn the resulting object code into ROM. Please carefully read this section before attempting to assemble Supermon 816, and be sure to make a backup copy of the source code prior to editing it.

In the following discussion, microprocessor registers are symbolized as follows:

Symbol	Register
.A	accumulator (8 bits)
.C	accumulator (16 bits)
.X	X-index
.Y	Y-index
DB	data bank
PB	program bank
PC	program counter
SP	stack pointer
SR	status register

Additional information on register symbology will be given later on.

Near the top of the source code are symbols that set various values required during assembly. These values must be edited as required to integrate Supermon 816 into your system. Please refer to the following table for details.

Symbol	Description
origin	This is the starting address for assembly. Supermon 816's "cold start" entry point, referred to in the source code as **jmon**, has this address.
vecexit	This is the address to which Supermon 816 will go when the X (exit) command is issued. That is, vecexit is an exit vector that returns control to the operating system. Supermon 816 will do a long jump (JML) to this address, which means that vecexit is interpreted during assembly as a 24 bit address.

vecbrki	This is a vector used by Supermon 816 to "wedge" into your operating system's BRK handler. Supermon 816 will modify this vector so that execution of a BRK instruction is intercepted and the microprocessor registers are captured. Your operating system's BRK service routine front end should jump through this vector after pushing the registers, using the following code:
	
	`ibrk phb ;save DB`
	`phd ;save DP`
	`rep #%00110000 ;16 bit registers`
	`pha ;save accumulator`
	`phx ;save index X`
	`phy ;save index Y`
	`jmp (vecbrki) ;jump through vector`
	
	The code starting at ibrk would be pointed to by the 65C816 hardware vector at $00FFE6.
	
	When a G or J command (described later) is issued to Supermon 816 to execute machine code, the above sequence will be reversed before a jump is made to the code to be executed. Upon exit from Supermon 816, the original address at vecbrki will be restored.
	
	If your BRK handler's front end doesn't conform to the above you will have to modify Supermon 816 to accommodate the differences. The most likely needed changes will be in the order in which the 65C816's registers are pushed to the stack.

hwstack	This is the address of the top of the 65C816's hardware stack. Supermon 816 will initialize the stack pointer to this address when entered through the cold start vector at jmon. The stack pointer will be undisturbed when entry into Supermon 816 is through jmonbrk, which is the BRK entry point.

Symbol	Description
zeropage	zeropage defines the start of direct page memory used by Supermon 816. Be sure that no conflict occurs with other software, as an overwrite of any of these locations may be fatal to Supermon 816.
getcha	getcha refers to an operating system API (application programming interface) call that returns a datum (byte) from the console in .A. That is, Supermon 816 calls getcha to get typed input. Supermon 816 expects that getcha is a non-blocking subroutine and returns with carry clear to indicate that a datum is in .A, or with carry set to indicate that no datum was available. getcha will be called with a JSR instruction. Supermon 816 also expects .X and .Y to be preserved upon return from getcha. You may have to modify Supermon 816 at all calls to getcha if your "get datum" API works differently than described.
getchb	getchb refers to an operating system API call that returns a datum (byte) from a secondary or auxiliary input port in .A. Supermon 816 calls getchb to get input during an S-record load operation. Supermon 816 expects that getchb is a non-blocking subroutine and returns with carry clear to indicate that a datum is in .A, or with carry set to indicate that no datum was available. getchb will be called with a JSR instruction. Supermon 816 also expects .X and .Y to be preserved upon return from getchb. You may have to modify Supermon 816 at all calls to getchb if your "get datum" API works differently than described.
putcha	putcha refers to an operating system API call that prints one character to the console screen. The character to be printed will be in .A, which will be set to 8-bit width when the API call is made. Supermon 816 expects that putcha will block until the character can be processed. putcha will be called with a JSR instruction. Supermon 816 also expects .X & .Y to be preserved upon return from putcha. You may have to modify Supermon 816 at all calls to putcha if your "put character" routine works differently than described.
chanbctl	chanbctl refers to an API call in BCS Technology Limited's "universal" NXP multichannel UART driver that enables or disables the TIA-232 channel B receiver. If this call is not present in your system's API then it will be necessary to comment out references to it.

Symbol	Description
`stopkey`	Supermon 816 will poll for a "stop key" during display operations, such as code disassembly and memory dumps, so as to halt the display and return control to the Supermon 816 prompt. `stopkey` must be defined with the ASCII value that the "stop key" will emit when typed. The polling is via a call to `getcha` (described above). The default `stopkey` definition of `$03` is for ASCII `<ETX>` or **[Ctrl-C]**. An alternative definition could be `$1B`, which is ASCII `<ESC>` or **[ESC]**.
`ibuffer` `auxbuf`	Supermon 816 will use these locations for workspace in various ways. These buffers may be located anywhere in memory that is convenient, as long as they are in the same bank in which Supermon 816 is running. The buffers are stateless, which means that unless Supermon 816 has control of your system they may be overwritten without consequence. Only `ibuffer` should be edited, unless there is a compelling reason to relocate `auxbuf`. `auxbuf` occupies 33 bytes.

SUPERMON 816 OPERATION

This section will discuss Supermon 816's operation.

Supermon 816 is started by jumping to the address at which it was loaded—the "cold start" entry point, which is defined in the source code as **jmon**. Upon initial startup, the monitor will set up some vectors, display a banner, dump the 65C816's registers and print a dot (**.**), which is the monitor's input prompt.

The register dump will appear as follows when the monitor is started:

```
    PB  PC   NVmxDIZC  .C   .X   .Y   SP   DP  DB
; xx 0000  00000000 0000 0000 0000 xxxx 0000 00
```

The above values are typical. Register heading meanings are as follows:

Heading	Register
PB	8-bit program bank
PC	16-bit program counter
NVmxDIZC	status register flags
.C	16-bit accumulator
.X	16-bit X-index
.Y	16-bit Y-index
SP	16-bit stack pointer
DP	16-bit direct page pointer
DB	8-bit data bank

The register dump is retrieved from a set of "shadow" locations stored on direct page, and excepting the status register, is in hexadecimal. The shadow values are loaded into the 65C816's registers when the monitor is commanded to execute a program. If the program returns control to the monitor when it is finished, the 65C816's registers will be copied to shadow storage and the register dump will reflect what the registers contained at the time the monitor assumed control of the system.

For convenience, the status register is displayed in bitwise fashion, rather than as a hexadecimal number, as often seen in other monitors. The `m` and `x` bits in the status register refer to the accumulator/memory and index register sizes, respectively.

All monitor commands commence with a single character, followed in some cases by whitespace, tab or comma-delimited arguments that are interpreted as addresses or data, depending on context. Recognized commands are:

Command	Function
A	Assemble 65C816 machine code
C	Compare memory ranges
D	Disassemble 65C816 machine code
F	Fill memory range
G	Execute 65C816 code (JMP)
H	Search memory range
J	Execute 65C816 code as subroutine (JSR)
L	Load 65C816 machine code
M	Display memory range
R	Display 65C816 registers
T	Copy memory range
X	Exit Supermon 816
>	Display and edit memory
;	Edit 65C816 registers

Supermon 816 internally processes all numeric input as 32 bit integers regardless of the actual value entered. A number entered without a leading radix symbol is assumed to be hexadecimal. Other bases are supported by preceding the number with an appropriate radix symbol, which will cause an “on-the-fly” conversion to occur:

Symbol	Radix
%	Binary
@	Octal
+	Decimal
$	Hexadecimal

For example, the 65C816's SEP instruction manipulates status register bits. Hence:

```
SEP #%00110000
```

is generally more convenient to enter than:

```
SEP #$30
```

If a radix is used there must be no space between it and the number itself.

Supermon 816 includes a number conversion function as part of its command set. Entering a radix symbol at the input prompt along with a number that is valid for that radix will display the number in all four radices. For example, typing:

```
+12345 [CR]
```

at the prompt will result in the following display:

```
$3039
+12345
@00030071
%11000000111001
```

[CR] represents the return or enter key on your console keyboard. The largest decimal number that may be converted in this fashion is $(2^{32})-1$ or 4,294,967,295.

Many functions accept one or more addresses as command arguments. Hence such arguments may be expressed as eight, 16 or 24 bit values in any radix; most functions that display addresses will display them as 24 bit values. During code assembly, immediate mode instructions, excepting REP and SEP, will accept either eight or 16 bit operands.

Most input is not case-sensitive and extra whitespace between commands and arguments will be ignored. In the event Supermon 816 cannot process your input due to faulty syntax an *ERR diagnostic will be printed and you will be prompted for another command.

The following discussion will cover each Supermon 816 function in detail.

Assemble 65C816 Machine Code

Syntax: **`A <addr> <mnem> [<oper>]`**

This function assembles a 65C816 machine instruction at address `<addr>`, using the mnemonic `<mnem>` and the operand `<oper>`. `<mnem>` must be an instruction mnemonic as described in the assembly language standard in the Western Design Center (WDC) 65C816 data sheet. Alternate mnemonics, such as `DEA` in place of `DEC` (decrement the accumulator) or `BLT` (branch if less-than) in place of `BCC`, are not supported.

Upon successful assembly, the instruction will be disassembled and displayed in place of your typed input, and the assembler will prompt with the address of the next instruction to be assembled. Pressing `[CR]` at the prompt without entering another instruction will discontinue assembly and return you to the command prompt. An assembly error, for example, a branch that is out of range, will cause the monitor to re-prompt with the same assembly address.

The way in which the assembler interprets and assembles some instructions warrants further discussion.

- 65C816 immediate mode (`#`) instructions other than `PEA`, `REP` and `SEP` can operate on eight or 16 bit operands, depending on the condition of the `m` and `x` bits in the status register. Supermon 816's assembler always resolves operands to the least number of bits that is valid for the instruction being assembled. Hence if an immediate mode operand can be resolved to an eight bit quantity, the instruction will be assembled with an eight bit operand. For example:

```
a 002000 LDA #$0030
```

will always be assembled and stored into memory as:

```
>002000 A9 30
```

It is possible to force the assembler to “promote” an eight bit immediate mode operand to 16 bits by preceding the `#` symbol with `!`, a feature referred to as “forced long immediate.” For example, entering:

```
a 002000 LDA !#$30
```

will promote the operand to 16 bits and the instruction will be generated as:

```
>002000 A9 30 00
```

The next assembly address will be $002003 instead of $002002.

It is important to note that the assembler does not "know" whether a 16 bit immediate mode operand is appropriate in the context of the program being assembled. It is your responsibility to keep track of register sizes as you enter instructions.

Attempting to use forced long immediate with the PEA, REP and SEP instructions will fail with a syntax error. PEA is automatically assembled with a 16 bit operand.

- In syntactically-correct symbolic assemblers, the instruction ASL A would mean "left-shift the accumulator," the A operand being a default symbol for the accumulator. The monitor's assembler will interpret such an instruction as ASL $0A. Hence implied accumulator instructions must be entered without an operand. Similarly, the symbolic assembler instruction DEC A (decrement the accumulator) must be entered as DEC without an operand.

- The COP instruction is a two byte instruction, the second byte being referred to as the "signature." Hence the assembler requires that COP be entered with an eight bit operand. The 65C816 data sheet states that signature values from $80 to $FF are "reserved." The assembler does not enforce this distinction and accepts any signature from $00 to $FF.

- Recommended WDC assembler syntax permits assembly of the JMP and JSR instructions with a 24 bit address, for example:

```
a 002000 JMP $A4031F
a 002004 JSR $A40359
```

The WDC syntax also describes the mnemonics JML and JSL as "long" forms of JMP and JSR whose operands are always resolved to a 24 bit address.

For technical reasons, Supermon 816 cannot accept a 24 bit address with the JMP and JSR instructions. If you wish to assemble a long jump instruction you must use JML in place of JMP, and JSL in place of JSR. For example:

```
a 002000 JML $A4031F
a 002004 JSL $A40359
```

JML and JSL operands are always resolved to 24 bits, which means that:

```
a 002000 JML $1
```

will be assembled and stored into memory as:

```
>002000 5C 01 00 00
```

and the next assembly address will be $002004.

As a reminder, if you call a subroutine with JSL you must exit that subroutine with RTL, not RTS.

- The MVN and MVP copy instructions have an irregular syntax. These instructions must entered with two eight bit operands, the first operand representing the bank from which bytes will be copied (source bank), and the second operand representing the bank into which the bytes will be copied (destination bank). For example, the instruction:

```
a 002000 MVN $02 $03
```

will be assembled and stored into memory as:

```
>002000 54 03 02
```

The above instruction will be disassembled as:

```
. 002000  54 03 02     MVN $02,$03
```

and when executed, will cause the 65C816 to copy bytes from bank $02 to bank $03. During instruction entry, the operands may be separated with whitespace or a comma. The official WDC assembler syntax uses a comma.

- The syntax described for the PEA and PEI instructions in the Lichty and Eyes publication *Programming the 65816* is not consistent with the actual behavior of these instructions. PEA is an immediate mode instruction that pushes its operand to the stack as a word (16 bit value). Despite the mnemonic's meaning (**P**ush **E**ffective **A**ddress), the operand is an assembly-time constant that can represent data of any type.

  PEI treats its operand as a contiguous pair of direct page locations from which a word will be loaded and pushed to the stack.

Despite the mnemonic's meaning (**P**ush **E**ffective **I**ndirect), the word pushed to the stack is not obtained via indirection—it is loaded from the direct page address that is the instruction's operand.

In an effort to be consistent with the way in which `PEA` and `PEI` behave, they are treated as immediate mode and direct page instructions, respectively. Hence `PEA` must be entered as:

```
PEA #<oper>
```

where `<oper>` is anything that can be resolved to 16 bits—`PEA #$01` is acceptable, as the monitor will promote the operand to 16 bits during assembly.

`PEI` must be entered as:

```
PEI <dp>
```

where `<dp>` is an eight bit direct page address.

- All versions of the 65C816 produced to date treat the `WDM` (William D. Mensch) "place-holder" instruction as a two-byte `NOP`. Hence `WDM` must be entered with an eight bit operand.

Usage examples:

```
a 002000 lda #02fd
a 002003 sta $0a0403
```

Compare Memory Ranges

Syntax: `C <addr1> <addr2> <addr3>`

This function compares memory starting at address `<addr1>` and ending at address `<addr2>` inclusive, to memory starting at address `<addr3>`. The range set by `<addr1>` and `<addr2>` may span banks. `<addr1>` must be equal to or lower than `<addr2>` or else an error will occur. The “equal to or lower” test is a 24 bit comparison. `<addr3>` may overlap the range set by `<addr1>` and `<addr2>` without causing an error.

The comparison begins by comparing the byte at `<addr1>` to the byte at `<addr3>`. If they are different, `<addr1>` will be printed to the console as a 24-bit hexadecimal number. Next, `<addr1>+1` will be compared to `<addr3>+1`, `<addr1>+2` to `<addr3>+2`, and so forth. The comparison will stop after `<addr2>` has been checked. The comparison operation can be halted at any time by striking the display “stop key” that has been defined with the `stopkey` symbol in the Supermon 816 source code.

Usage example: `c 002000 002005 003000`

Disassemble 65C816 Machine Code

Syntax: `D [<addr1> [<addr2>]]`

This function disassembles and displays 65C816 machine instructions as mnemonics and operands. If two arguments are entered, the range set by <addr1> and <addr2> may span banks. <addr1> must be equal to or lower than <addr2> or else an error will occur, the "equal to or lower" test being a 24 bit comparison.

When entered with no arguments, disassembly will start at the last known address at which memory was accessed. At initial startup, that address will be $000000. If only <addr1> is specified, disassembly will start at that address and proceed until a maximum of 21 bytes has been disassembled. If both <addr1> and <addr2> are specified, disassembly will start at <addr1> and end after <addr2> has been processed, which may cause the display to scroll. Disassembly can be halted at any time by striking the display "stop key."

The disassembly display is enlarged as compared to an equivalent 6502/65C02 disassembly display in order to account for 24 bit addresses. A typical disassembly might appear as follows:

```
. 002000  BF 9E 12 8F  LDA $8F129E,X
. 002004  DD 00 04     CMP $0400,X
. 002007  F0 6E        BEQ $2077
. 002009  CA           DEX
. 00200A  10 F4        BPL $2000
```

The byte immediately following the disassembly address will be the instruction opcode.

Disassembling immediate mode instructions other than PEA, REP and SEP is somewhat complicated by the fact that they may have 8- or 16-bit operands. Normally, Supermon 816 would not be able to determine the proper operand size, since the way in which the 65C816 processes immediate mode operands is a function of status register bits as the program is running, and is not determined by specific opcodes. For example, the opcode $A9 applies to LDA #$01 and LDA #$0201. The byte sequence $A9 $01 $02 $E8 would normally be disassembled to LDA #$01 followed by COP $E8, even though what may have been assembled was LDA #$0201 followed by INX.

Supermon 816 attempts to compensate by keeping track of the most recent `REP` or `SEP` instruction encountered during disassembly, hence attempting to recreate the assembly sequence that generated the code being disassembled. `REP`/`SEP` state information is initialized to assume 8-bit operands when the disassemble code command is issued with an address. If the next disassemble command is issued with no addresses, the monitor will continue to keep track of `REP` and `SEP` instructions and will continue to correctly distinguish between eight bit and 16 bit immediate mode operands. If a disassemble code command is again issued with an address, `REP`/`SEP` state information will reinitialized and immediate mode instructions will again be assumed to have 8-bit operands until a `REP` instruction is encountered.

Usage example: `d 0C2000 0C2020`

Fill Memory Range

Syntax: `F <addr1> <addr2> <fill>`

This function writes the eight bit `<fill>` value into all addresses beginning at `<addr1>` and ending with `<addr2>` inclusive. `<addr1>` must be in the same bank and equal to or lower than `<addr2>` or else an error will occur.

Caution must be exercised with this command, as inadvertently overwriting system areas may trigger undefined hardware behavior or cause a crash.

Usage example: `f 0e2000 0e2fff ea`

The above example will write a `NOP` instruction into RAM starting at address `$0E2000`, with the final `NOP` being written to `$0E2FFF`.

Execute Code

Syntax: `G [<addr>]`

This function loads the 65C816's registers with the values displayed by the most recent register dump and then starts execution of a program. If no argument is given, execution will commence at the address displayed in the register dump. Otherwise, execution will commence at address `<addr>`. Assuming that the `BRK` instruction is properly intercepted by the system (see above discussion in the system integration section), execution of `BRK` will return control to Supermon 816, at which time `*BRK` will be printed on the console screen, a register dump will occur and the input prompt will appear.

Usage example: `g 002000`

Search Memory Range

Syntax: `H <addr1> <addr2> <seq>`

This function searches (**H**unts through) memory for the byte sequence `<seq>`, starting at address `<addr1>` and ending at address `<addr2>` inclusive. The range set by `<addr1>` and `<addr2>` may span banks. `<addr1>` must be equal to or lower than `<addr2>` or else an error will occur, the "equal to or lower" test being a 24 bit comparison.

`<seq>` may be entered as one or more whitespace or comma-delimited byte values, or as a character string. If the latter is desired, the string must be preceded with a single quote character (`'`), which will not be included in `<seq>`. See the below examples for the correct syntax. A character string search is case-sensitive.

During the search, each address at which `<seq>` is found will be printed to the console screen as a 24-bit hexadecimal number. The search operation can be halted at any time by striking the display "stop key." Search speed will be affected by the size of `<seq>`, which may a maximum of 32 bytes, as well by the selected memory range.

Usage examples:

```
h 02E000 02E800 A9 04 00    (byte pattern search)
h 0B2000 0B2fff 'testing    (character string search)
```

Execute Subroutine

Syntax: `J [<addr>]`

This function loads the 65C816's registers with the values displayed by a register dump and then starts execution of a program. If no argument is given, execution will commence at the address displayed in the most recent register dump. Otherwise, execution will commence at the 24-bit address `<addr>`. The execution address will be treated as the entry point of a subroutine, which means an internal monitor return address will be pushed to the stack prior to execution of the target code.

Execution of an `RTS` instruction will return control to Supermon 816 if the hardware stack remains "in balance," at which time `*RTS` will be printed before dumping the registers. In this case, the stack pointer value in the dump will be what it was prior to executing the called subroutine, unless the subroutine modified the stack and loaded `SP` with a new value to reflect the changes.

CAUTION: The called subroutine must terminate with `RTS`, not `RTL`. The monitor does not `JSL` to the subroutine. If it is necessary to call a subroutine in a bank other than the one in which the monitor is executing it will be necessary to specify a full 24 bit address or change the `PB` and `PC` register values before execution.

Usage example: `j 04e015`

The above example will call a subroutine at `$E015` in bank `$04` and assuming the subroutine ends with `RTS` and does not modify the stack, control will return to Supermon 816.

Load 65C816 Machine Code

Syntax: `L [<bank> [<offset>]]`

This function is the means by which data may be transferred into POC from a foreign source.

Data transfer into POC is accomplished through the transmission of Motorola hex data records, also known as S-records, from the data source. The S-record loader processes S1, S5 and S9 records, and accepts but ignores other S-record types. There may be multiple S1 records, but only one each of an S5 and S9 record. Transmission of an S5 record after all S1 records have been sent is an optional step, but is recommended as an additional error check. The final record in the data stream must be an S9. General information about the Motorola S-record format is readily available from a variety of sources and will not be discussed here.

The data stream is transmitted to your 65C816 system's auxiliary input port, whose "get datum" API call (symbolized as `getchb`) is defined as described above in the system integration section. As each S1 record arrives, it will be translated to binary, error-checked and if no error is detected, written into memory. The load operation will be completed when an S9 record has been received and processed.

Each S1 record includes a load address, which is a 16 bit field that indicates where in memory the first data byte of the record will be stored. As each data byte in the record is stored the monitor will increment a working load address. By default, storage will occur in the program bank (`PB`) that was displayed in the most recent 65C816 register dump.

If the optional eight bit bank parameter `<bank>` is entered, storage will be directed to that bank. If the optional eight bit page offset parameter `<offset>` is also entered, it will be used along with `<bank>` to perform a relocating load to any page boundary within the 65C816's address space. During a relocating load, `<offset>` will be added to the most significant byte (MSB) of the working load address, with any carry into bit 16 being discarded.

On completion of a successful load, the non-zero load address specified in the S9 record will be copied to the `PC` shadow register and will appear in a subsequent register dump. If an alternate bank was used with the load command, that bank will be copied to the `PB` shadow register. If a page offset was also entered, it will be added to the S9 load address and the new address will be written to the `PC` shadow register. Hence entering the `G` or `J` command without an argument following a successful load will cause execution to start at the effective load address.

The load procedure is as follows:

1. Verify that you have working connection between your system and the data source. If the connection is via TIA-232 it is strongly recommended that hardware handshaking be used to pace data flow. Software handshaking is unreliable at speeds in excess of 9600 bits per second.

2. At the data source, assemble your code and save it into a "flat" (text) file in Motorola S-record format. Each S-record must be delimited by an ASCII <LF> (linefeed, $0A) character or a <CRLF> sequence (carriage return, $0D, followed $0A). There must be at least one S1 record and only one S9 record. An S5 record is optional but recommended. *The final record must be an S9.*

3. Type **L** at Supermon 816's prompt, including bank and page offset arguments if a relocating load is desired. When Supermon 816 is ready to load data, **Ready:** will be printed on the console, the cursor will appear and a loop will be entered awaiting input from the data source. You can abort the process at any time by striking the display "stop key."

   Note: Aborting while records are being loaded and processed will result in an incomplete load. The PB and PC shadow registers will not be updated if the load is aborted.

4. At the data source, perform whatever steps are required to output your S-record file to the port that is in communication with your 65C816 system. As S-records are received, Supermon 816 will print a dot on the console screen for each successfully processed record.

5. Upon successful processing of the S9 record, the starting and ending addresses for the load will be printed, the PB and PC shadow registers will be updated as necessary, and control will return to Supermon 816's input prompt.

6. In the event of an error, the load will abort and Supermon 816 will print a diagnostic. In most cases, an error will be due to a mismatch between the checksum embedded in the most recently received S-record and the checksum calculated during the load. This sort of error is often the result of transmission glitches on the data link, but could also be due to a computation error at the originating end involving the generation of the record's checksum.

   Another possible source of error would be an improperly formatted or corrupted object code file. At least one S1 record is required, and only one S5 and S9 record can be present. The total record count in the S5 record must agree with the number of S1 records that were received and processed.

Caution must be exercised with this command, as inadvertently overwriting system areas may trigger undefined hardware behavior or cause a crash.

Usage example: `l 4 3e`

The above command will perform a relocating load to bank `$04`, adding `$3E00` to the address of each loaded S-record, discarding any carry into bit 16 of the address.

Display Memory Range

Syntax: `M [<addr1> [<addr2>]]`

This function dumps the contents of a range of memory into a human-readable format consisting of hexadecimal byte values and ASCII equivalents. If both arguments are entered, `<addr1>` must be equal to or lower than `<addr2>` or else an error will occur, the "equal to or lower" test being a 24 bit comparison. The range set by `<addr1>` and `<addr2>` may span banks.

When entered with no arguments, Supermon 816 will start the memory dump at the last known address at which memory was accessed. At initial startup, that address will be `$000000`. If both addresses are omitted or only `<addr1>` is specified, a total of one page (256 bytes) of data will be dumped. If both `<addr1>` and `<addr2>` are specified, the dump will start at `<addr1>` and end at `<addr2>` inclusive, which may cause the display to scroll. The dump can be halted at any time by striking the display "stop key."

The display will consist of one or more formatted lines, such as the following example:

```
>002000 42 43 53 20 54 65 63 68 6E 6F 67 79 00 00:BCS Technology..
```

Sixteen bytes will be dumped per line, and bytes in the range `$20-$7E` inclusive will also be displayed as ASCII. Bytes outside of that range will be displayed as a dot (`.`). If supported by the console hardware, the ASCII portion of the display will be in reverse video, as depicted in the above example.

Usage example: `m 002000 00207f`

Display 65C816 Registers

Syntax: `R`

This function dumps the 65C816's registers as known to Supermon 816. An error will occur if any arguments are entered. A typical dump following the execution of a program might be as follows:

```
   PB  PC   NVmxDIZC  .C   .X   .Y   SP   DP  DB
; 01 C207  00110000 1B73 00A0 0001 CDFF 0000 02
```

The dumped data are retrieved from Supermon 816's shadow registers, which are updated when a running program is interrupted by `BRK`, or when a subroutine executed via the **`J`** function returns control to Supermon 816 via `RTS`. See the **`;`** register change function for the procedure used to change register values.

Copy Memory Range

Syntax: `T <addr1> <addr2> <addr3>`

This function copies (**T**ransfers) memory starting at address `<addr1>` and ending at address `<addr2>` inclusive, to memory starting at address `<addr3>`. `<addr1>` must be equal to or lower than `<addr2>` and in the same bank as `<addr2>` or else an error will occur. `<addr3>` may be in any bank and if in the same bank from which copying is to take place, may overlap the range set by `<addr1>` and `<addr2>` without causing an error, as long as `<addr3>` is not the same as `<addr1>`.

Caution must be exercised with this command, as inadvertently overwriting system areas may trigger undefined hardware behavior or cause a crash.

Usage example: `t 002000 0020ff 043000`

The above example will copy the memory range `$002000-$0020FF` to `$043000`.

Edit Memory

Syntax: `> <addr> [<data>]`

This function may be used to edit memory. If entered with a valid address only it will function as a one-line memory dump. Otherwise, memory starting at address `<addr>` will be overwritten with the data in `<data>`. `<data>` may be entered as one or more whitespace or comma-delimited byte values, up to 32, or as a character string of no more than 32 characters. If the latter is desired, the string must be preceded with a single quote character (`'`), which will not be included in `<data>`. See the below examples for the correct syntax.

Caution must be exercised with this command, as inadvertently overwriting system areas may trigger undefined hardware behavior or cause a crash.

Usage examples:

```
> 002000 00 EA 00 EA      (enters a byte pattern)
> 042000 'testing 123     (enters a character string)
```

Modify 65C816 Registers

Syntax: `; [<PB> [<PC> [<SR> ...]]]`

This function is used to change one or more of the shadow values that are loaded into the 65C816's registers when **G** or **J** is used to execute a program. If no arguments are entered the effect is the same as the **R** (dump registers) function. If you wish to change the register values, enter new values in the same order and of the correct size for the corresponding register. You need only enter data up to the last register to be changed. Upon entering the new values another register dump will occur displaying the new values.

Usage example: `;4 2000 %00110000`

The above will set `PB` to `$04`, `PC` to `$2000` and `SR` to `$30`. No other registers will be changed. When a **G** or **J** command is entered, program execution will begin at `$042000`, with all status register bits except `m` and `x` cleared.

Exit Supermon 816

Syntax: **`x`**

This function will terminate execution of Supermon 816 and return control to the local operation system via the vector `vecexit`.