1. Overview

SIMH (history simulators) is a set of portable programs, written in C, which simulate various historically interesting computers. This document describes how to design, write, and check out a new simulator for SIMH. It is not an introduction to either the philosophy or external operation of SIMH, and the reader should be familiar with both of those topics before proceeding. Nor is it a guide to the internal design or operation of SIMH, except insofar as those areas interact with simulator design. Instead, this manual presents and explains the form, meaning, and operation of the interfaces between simulators and the SIMH simulator control package. It also offers some suggestions for utilizing the services SIMH offers and explains the constraints that all simulators operating within SIMH will experience.

Some terminology: Each simulator consists of a standard simulator control package (SCP and related libraries), which provides a control framework and utility routines for a simulator; and a unique virtual machine (VM), which implements the simulated processor and selected peripherals. A VM consists of multiple devices, such as the CPU, paper tape reader, disk controller, etc. Each controller consists of a named state space (called registers) and one or more units. Each unit consists of a numbered state space (called a data set). The host computer is the system on which SIMH runs; the target computer is the system being simulated.

SIMH is unabashedly based on the MIMIC simulation system, designed in the late 1960’s by Len Fehskens, Mike McCarthy, and Bob Supnik. This document is based on MIMIC’s published interface specification, “How to Write a Virtual Machine for the MIMIC Simulation System”, by Len Fehskens and Bob Supnik.

2. Data Types

SIMH is written in C. The host system must support (at least) 32-bit data types (64-bit data types for the PDP-10 and other large-word target systems). To cope with the vagaries of C data types, SIMH defines some unambiguous data types for its interfaces:

<table>
<thead>
<tr>
<th>SIMH data type</th>
<th>interpretation in typical 32-bit C</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8, uint8</td>
<td>signed char, unsigned char</td>
</tr>
<tr>
<td>int16, uint16</td>
<td>signed short, unsigned short</td>
</tr>
<tr>
<td>int32, uint32</td>
<td>signed int, unsigned int</td>
</tr>
<tr>
<td>t_int64, t_uint64</td>
<td>long long, _int64 (system specific)</td>
</tr>
<tr>
<td>t_addr</td>
<td>simulated address, uint32 or t_uint64</td>
</tr>
<tr>
<td>t_value</td>
<td>simulated value, uint32 or t_uint64</td>
</tr>
<tr>
<td>t_svalue</td>
<td>simulated signed value, int32 or t_int64</td>
</tr>
<tr>
<td>t_mttrec</td>
<td>mag tape record length, uint32</td>
</tr>
<tr>
<td>t_stat</td>
<td>status code, int</td>
</tr>
<tr>
<td>t_bool</td>
<td>true/false value, int</td>
</tr>
</tbody>
</table>
[The inconsistency in naming t_int64 and t_uint64 is due to Microsoft VC++, which uses int64 as a structure name member in the master Windows definitions file.]

In addition, SIMH defines structures for each of its major data elements:

- **DEVICE**: device definition structure
- **UNIT**: unit definition structure
- **REG**: register definition structure
- **MTAB**: modifier definition structure
- **CTAB**: command definition structure
- **DEBTAB**: debug table entry structure

### 3. VM Organization

A virtual machine (VM) is a collection of devices bound together through their internal logic. Each device is named and corresponds more or less to a hunk of hardware on the real machine; for example:

<table>
<thead>
<tr>
<th>VM device</th>
<th>Real machine hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>central processor and main memory</td>
</tr>
<tr>
<td>PTR</td>
<td>paper tape reader controller and paper tape reader</td>
</tr>
<tr>
<td>TTI</td>
<td>console keyboard</td>
</tr>
<tr>
<td>TTO</td>
<td>console output</td>
</tr>
<tr>
<td>DKP</td>
<td>disk pack controller and drives</td>
</tr>
</tbody>
</table>

There may be more than one device per physical hardware entity, as for the console; but for each user-accessible device there must be at least one. One of these devices will have the pre-eminent responsibility for directing simulated operations. Normally, this is the CPU, but it could be a higher-level entity, such as a bus master.

The VM actually runs as a subroutine of the simulator control package (SCP). It provides a master routine for running simulated programs and other routines and data structures to implement SCP’s command and control functions. The interfaces between a VM and SCP are relatively few:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>char sim_name[]</td>
<td>simulator name string</td>
</tr>
<tr>
<td>REG *sim_pc</td>
<td>pointer to simulated program counter</td>
</tr>
<tr>
<td>int32 sim_emax</td>
<td>maximum number of words in an instruction</td>
</tr>
<tr>
<td>DEVICE *sim_devices[]</td>
<td>table of pointers to simulated devices, NULL terminated</td>
</tr>
<tr>
<td>char *sim_stop_messages[]</td>
<td>table of pointers to error messages</td>
</tr>
<tr>
<td>t_stat sim_load (...)</td>
<td>binary loader subroutine</td>
</tr>
<tr>
<td>t_stat sim_inst (void)</td>
<td>instruction execution subroutine</td>
</tr>
<tr>
<td>t_stat parse_sym (...)</td>
<td>symbolic instruction parse subroutine (optional)</td>
</tr>
<tr>
<td>t_stat fprint_sym (...)</td>
<td>symbolic instruction print subroutine (optional)</td>
</tr>
</tbody>
</table>

In addition, there are six optional interfaces, which can be used for special situations, such as GUI implementations:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>void (*sim_vm_init) (void)</td>
<td>pointer to once-only initialization routine for VM</td>
</tr>
</tbody>
</table>
There is no required organization for VM code. The following convention has been used so far. Let name be the name of the real system (i1401 for the IBM 1401; i1620 for the IBM 1620; pdp1 for the PDP-1; pdp18b for the other 18-bit PDP’s; pdp8 for the PDP-8; pdp11 for the PDP-11; nova for Nova; hp2100 for the HP 21XX; h316 for the Honeywell 315/516; gri for the GRI-909; pdp10 for the PDP-10; vax for the VAX; sds for the SDS-940):

- name.h contains definitions for the particular simulator
- name_sys.c contains all the SCP interfaces except the instruction simulator
- name_cpu.c contains the instruction simulator and CPU data structures
- name_stddev.c contains the peripherals which were standard with the real system.
- name_lp.c contains the line printer.
- name_mt.c contains the mag tape controller and drives, etc.

The SIMH standard definitions are in sim_defs.h. The base components of SIMH are:

<table>
<thead>
<tr>
<th>Source module</th>
<th>header file</th>
<th>module</th>
</tr>
</thead>
<tbody>
<tr>
<td>scp.c</td>
<td>scp.h</td>
<td>control package</td>
</tr>
<tr>
<td>sim_console.c</td>
<td>sim_console.h</td>
<td>terminal I/O library</td>
</tr>
<tr>
<td>sim_fio.c</td>
<td>sim_fio.h</td>
<td>file I/O library</td>
</tr>
<tr>
<td>sim_timer.c</td>
<td>sim_timer.h</td>
<td>timer library</td>
</tr>
<tr>
<td>sim_sock.c</td>
<td>sim_sock.h</td>
<td>socket I/O library</td>
</tr>
<tr>
<td>sim_ether.c</td>
<td>sim_ether.h</td>
<td>Ethernet I/O library</td>
</tr>
<tr>
<td>sim_tmxr.c</td>
<td>sim_tmxr.h</td>
<td>terminal multiplexor simulation library</td>
</tr>
<tr>
<td>sim_tape.c</td>
<td>sim_tape.h</td>
<td>magtape simulation library</td>
</tr>
</tbody>
</table>

### 3.1 CPU Organization

Most CPU’s perform at least the following functions:

- Time keeping
- Instruction fetching
- Address decoding
- Execution of non-I/O instructions
- I/O command processing
- Interrupt processing

Instruction execution is actually the least complicated part of the design; memory and I/O organization should be tackled first.

#### 3.1.1 Time Base

In order to simulate asynchronous events, such as I/O completion, the VM must define and keep a time base. This can be accurate (for example, nanoseconds of execution) or arbitrary (for example, number of instructions executed), but it must be consistently used throughout the VM. All existing VM’s count time in instructions.
The CPU is responsible for counting down the event counter \texttt{sim\_interval} and calling the asynchronous event controller \texttt{sim\_process\_event}. SCP does the record keeping for timing.

### 3.1.2 Step Function

SCP implements a stepping function using the \texttt{step} command. \texttt{STEP} counts down a specified number of time units (as described in section 3.1.1) and then stops simulation. The VM can override the \texttt{STEP} command’s counts by calling routine \texttt{sim\_cancel\_step}:

- \texttt{t\_stat sim\_cancel\_step (void) – cancel \texttt{STEP} count down.}

The VM can then inspect variable \texttt{sim\_step} to see if a \texttt{STEP} command is in progress. If \texttt{sim\_step} is non-zero, it represents the number of steps to execute. The VM can count down \texttt{sim\_step} using its own counting method, such as cycles, instructions, or memory references.

### 3.1.3 Memory Organization

The criterion for memory layout is very simple: use the SIMH data type that is as large as (or if necessary, larger than), the word length of the real machine. Note that the criterion is word length, not addressability: the PDP-11 has byte addressable memory, but it is a 16-bit machine, and its memory is defined as uint16 M[]. It may seem tempting to define memory as a union of int8 and int16 data types, but this would make the resulting VM endian-dependent. Instead, the VM should be based on the underlying word size of the real machine, and byte manipulation should be done explicitly. Examples:

<table>
<thead>
<tr>
<th>Simulator</th>
<th>memory size</th>
<th>memory declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1620</td>
<td>5-bit</td>
<td>uint8</td>
</tr>
<tr>
<td>IBM 1401</td>
<td>7-bit</td>
<td>uint8</td>
</tr>
<tr>
<td>PDP-8</td>
<td>12-bit</td>
<td>uint16</td>
</tr>
<tr>
<td>PDP-11, Nova</td>
<td>16-bit</td>
<td>uint16</td>
</tr>
<tr>
<td>PDP-1</td>
<td>18-bit</td>
<td>uint32</td>
</tr>
<tr>
<td>VAX</td>
<td>32-bit</td>
<td>uint32</td>
</tr>
<tr>
<td>PDP-10, IBM 7094</td>
<td>36-bit</td>
<td>t_uint64</td>
</tr>
</tbody>
</table>

### 3.1.4 Interrupt Organization

The design of the VM’s interrupt structure is a complex interaction between efficiency and fidelity to the hardware. If the VM’s interrupt structure is too abstract, interrupt driven software may not run. On the other hand, if it follows the hardware too literally, it may significantly reduce simulation speed. One rule I can offer is to minimize the fetch-phase cost of interrupts, even if this complicates the (much less frequent) evaluation of the interrupt system following an I/O operation or asynchronous event. Another is not to over-generalize; even if the real hardware could support 64 or 256 interrupting devices, the simulators will be running much smaller configurations. I’ll start with a simple interrupt structure and then offer suggestions for generalization.

In the simplest structure, interrupt requests correspond to device flags and are kept in an interrupt request variable, with one flag per bit. The fetch-phase evaluation of interrupts consists of two steps: are interrupts enabled, and is there an interrupt outstanding? If all the interrupt requests are kept as single-bit flags in a variable, the fetch-phase test is very fast:

\[
\text{if (int\_enable && int\_requests) \{ …process interrupt… \}}
\]
Indeed, the interrupt enable flag can be made the highest bit in the interrupt request variable, and the two tests combined:

```c
if (int_requests > INT_ENABLE) { …process interrupt… }
```

Setting or clearing device flags directly sets or clears the appropriate interrupt request flag:

```c
set: int_requests = int_requests | DEVICE_FLAG;
clear: int_requests = int_requests & ~DEVICE_FLAG;
```

At a slightly higher complexity, interrupt requests do not correspond directly to device flags but are based on masking the device flags with an enable (or disable) mask. There are now two parallel variables: device flags and interrupt enable mask. The fetch-phase test is now:

```c
If (int_enable && (dev_flags & int_enables)) { …process interrupt… }
```

As a next step, the VM may keep a summary interrupt request variable, which is updated by any change to a device flag or interrupt enable/disable:

```c
enable: int_requests = device_flags & int_enables;
disable: int_requests = device_flags & ~int_disables;
```

This simplifies the fetch phase test slightly.

At yet higher complexity, the interrupt system may be too complex or too large to evaluate during the fetch-phase. In this case, an interrupt pending flag is created, and it is evaluated by subroutine call whenever a change could occur (start of execution, I/O instruction issued, device time out occurs). This makes fetch-phase evaluation simple and isolates interrupt evaluation to a common subroutine.

If required for interrupt processing, the highest priority interrupting device can be determined by scanning the interrupt request variable from high priority to low until a set bit is found. The bit position can then be back-mapped through a table to determine the address or interrupt vector of the interrupting device.

### 3.1.5 I/O Dispatching

I/O dispatching consists of four steps:

- Identify the I/O command and analyze for the device address.
- Locate the selected device.
- Break down the I/O command into standard fields.
- Call the device processor.

Analyzing an I/O command is usually easy. Most systems have one or more explicit I/O instructions containing an I/O command and a device address. Memory mapped I/O is more complicated; the identification of a reference to I/O space becomes part of memory addressing. This usually requires centralizing memory reads and writes into subroutines, rather than as inline code.

Once an I/O command has been analyzed, the CPU must locate the device subroutine. The simplest way is a large switch statement with hardwired subroutine calls. More modular is to call through a dispatch table, with NULL entries representing non-existent devices; this also simplifies support for modifiable device addresses and configurable devices. Before calling the device
routine, the CPU usually breaks down the I/O command into standard fields. This simplifies writing the peripheral simulator.

### 3.1.6 Instruction Execution

Instruction execution is the responsibility of VM subroutine `sim_instr`. It is called from SCP as a result of a RUN, GO, CONT, or BOOT command. It begins executing instructions at the current PC (`sim_PC` points to its register description block) and continues until halted by an error or an external event.

When called, the CPU needs to account for any state changes that the user made. For example, it may need to re-evaluate whether an interrupt is pending, or restore frequently used state to local register variables for efficiency. The actual instruction fetch and execute cycle is usually structured as a loop controlled by an error variable, e.g.,

```
reason = 0;
do { … } while (reason == 0); or while (reason == 0) { … }
```

Within this loop, the usual order of events is:

- If the event timer `sim_interval` has reached zero, process any timed events. This is done by SCP subroutine `sim_process_event`. Because this is the polling mechanism for user-generated processor halts (^E), errors must be recognized immediately:
  ```
  if (sim_interval <= 0) {
    if (reason = sim_process_event ()) break;  
  }
  ```

- Check for outstanding interrupts and process if required.

- Check for other processor-unique events, such as wait-state outstanding or traps outstanding.

- Check for an instruction breakpoint. SCP has a comprehensive breakpoint facility. It allows a VM to define many different kinds of breakpoints. The VM checks for execution (type E) breakpoints during instruction fetch.

- Fetch the next instruction, increment the PC, optionally decode the address, and dispatch (via a switch statement) for execution.

A few guidelines for implementation:

- In general, code should reflect the hardware being simulated. This is usually simplest and easiest to debug.

- The VM should provide some debugging aids. The existing CPU’s all provide multiple instruction breakpoints, a PC change queue, error stops on invalid instructions or operations, and symbolic examination and modification of memory.

### 3.2 Peripheral Device Organization

The basic elements of a VM are devices, each corresponding roughly to a real chunk of hardware. A device consists of register-based state and one or more units. Thus, a multi-drive disk subsystem is a single device (representing the hardware of the real controller) and one or more units (each representing a single disk drive). Sometimes the device and its unit are the
same entity as, for example, in the case of a paper tape reader. However, a single physical device, such as the console, may be broken up for convenience into separate input and output devices.

In general, units correspond to individual sources of input or output (one tape transport, one A-to-D channel). Units are the basic medium for both device timing and device I/O. Except for the console, all I/O devices are simulated as host-resident files. SCP allows the user to make an explicit association between a host-resident file and a simulated hardware entity.

Both devices and units have state. Devices operate on registers, which contain information about the state of the device, and indirectly, about the state of the units. Units operate on data sets, which may be thought of as individual instances of input or output, such as a disk pack or a punched paper tape. In a typical multi-unit device, all units are the same, and the device performs similar operations on all of them, depending on which one has been selected by the program being simulated.

(Note: SIMH, like MIMIC, restricts registers to devices. Replicated registers, for example, disk drive current state, are handled via register arrays.)

For each structural level, SIMH defines, and the VM must supply, a corresponding data structure. sim_device structures correspond to devices, sim_reg structures to registers, and sim_unit structures to units. These structures are described in detail in section 4.

The primary functions of a peripheral are:

- command decoding and execution
- device timing
- data transmission.

Command decoding is fairly obvious. At least one section of the peripheral code module will be devoted to processing directives issued by the CPU. Typically, the command decoder will be responsible for register and flag manipulation, and for issuing or canceling I/O requests. The former is easy, but the later requires a thorough understanding of device timing.

### 3.2.1 Device Timing

The principal problem in I/O device simulation is imitating asynchronous operations in a sequential simulation environment. Fortunately, the timing characteristics of most I/O devices do not vary with external circumstances. The distinction between devices whose timing is externally generated (e.g., console keyboard) and those whose timing is externally generated (disk, paper tape reader) is crucial. With an externally timed device, there is no way to know when an in-progress operation will begin or end; with an internally timed device, given the time when an operation starts, the end time can be calculated.

For an internally timed device, the elapsed time between the start and conclusion of an operation is called the wait time. Some typical internally timed devices and their wait times include:

<table>
<thead>
<tr>
<th>Device</th>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTR (300 char/sec)</td>
<td>3.3 msec</td>
</tr>
<tr>
<td>PTP (50 char/sec)</td>
<td>20 msec</td>
</tr>
<tr>
<td>CLK (line frequency)</td>
<td>16.6 msec</td>
</tr>
<tr>
<td>TTO (30 char/sec)</td>
<td>33 msec</td>
</tr>
</tbody>
</table>

Mass storage devices, such as disks and tapes, do not have a fixed response time, but a start-to-finish time can be calculated based on current versus desired position, state of motion, etc.
For an externally timed device, there is no portable mechanism by which a VM can be notified of an external event (for example, a key stroke). Accordingly, all current VM’s poll for keyboard input, thus converting the externally timed keyboard to a pseudo-internally timed device. A more general restriction is that SIMH is single-threaded. Threaded operations must be done by polling using the unit timing mechanism, either with real units or fake units created expressly for polling.

SCP provides the supporting routines for device timing. SCP maintains a list of devices (called active devices) that are in the process of timing out. It also provides routines for querying or manipulating this list (called the active queue). Lastly, it provides a routine for checking for timed-out units and executing a VM-specified action when a time-out occurs.

Device timing is done with the UNIT structure, described in section 4. To set up a timed operation, the peripheral calculates a waiting period for a unit and places that unit on the active queue. The CPU counts down the waiting period. When the waiting period has expired, \texttt{sim\_process\_event} removes the unit from the active queue and calls a device subroutine. A device may also cancel an outstanding timed operation and query the state of the queue. The timing subroutines are:

- \texttt{t\_stat sim\_activate (UNIT *uptr, int32 wait).} This routine places the specified unit on the active queue with the specified waiting period. A waiting period of 0 is legal; negative waits cause an error. If the unit is already active, the active queue is not changed, and no error occurs.

- \texttt{t\_stat sim\_cancel (UNIT *uptr).} This routine removes the specified unit from the active queue. If the unit is not on the queue, no error occurs.

- \texttt{int32 sim\_is\_active (UNIT *uptr).} This routine tests whether a unit is in the active queue. If it is, the routine returns the time (+1) remaining; if it is not, the routine returns 0.

- \texttt{double sim\_gtime (void).} This routine returns the time elapsed since the last RUN or BOOT command.

- \texttt{uint32 sim\_grtime (void).} This routine returns the low-order 32b of the time elapsed since the last RUN or BOOT command.

- \texttt{int32 sim\_qcount (void).} This routine returns the number of entries on the clock queue.

- \texttt{t\_stat sim\_process\_event (void).} This routine removes all timed out units from the active queue and calls the appropriate device subroutine to service the time-out.

- \texttt{int32 sim\_interval.} This variable counts down the first outstanding timed event. If there are no timed events outstanding, SCP counts down a “null interval” of 10,000 time units.

### 3.2.2 Clock Calibration

The timing mechanism described in the previous section is approximate. Devices, such as real-time clocks, which track wall time will be inaccurate. SCP provides routines to synchronize multiple simulated clocks (to a maximum of 8) to wall time.

- \texttt{int32 sim\_rtcn\_init (int32 clock\_interval, int32 clk).} This routine initializes the clock calibration mechanism for simulated clock \textit{clk}. The argument is returned as the result.

- \texttt{int32 sim\_rtcn\_calb (int32 tick\_per\_second, int32 clk).} This routine calibrates simulated clock \textit{clk}. The argument is the number of clock ticks expected per second.
The VM must call `sim_rtcn_init` for each simulated clock in two places: in the prolog of `sim_instr`, before instruction execution starts, and whenever the real-time clock is started. The simulator calls `sim_rtcn_calb` to calculate the actual interval delay when the real-time clock is serviced:

```c
/* clock start */
if (!sim_is_active (&clk_unit)) sim_activate (&clk_unit, sim_rtcn_init (clk_delay, clkno));
etc.

/* clock service */
sim_activate (&clk_unit, sim_rtcb_calb (clk_ticks_per_second, clkno));
```

The real-time clock is usually simulated clock 0; other clocks are used for polling asynchronous multiplexors or intervals timers.

### 3.2.3 Data I/O

For most devices, timing is half the battle (for clocks it is the entire war); the other half is I/O. Except for the console and other terminals, all I/O devices are simulated as files on the host file system in little-endian format. SCP provides facilities for associating files with units (ATTACH command) and for reading and writing data from and to devices in a endian- and size-independent way.

For most devices, the VM designer does not have to be concerned about the formatting of simulated device files. I/O occurs in 1, 2, 4, or 8 byte quantities; SCP automatically chooses the correct data size and corrects for byte ordering. Specific issues:

- Line printers should write data as 7-bit ASCII, with newlines replacing carriage-return/line-feed sequences.

- Disks should be viewed as linear data sets, from sector 0 of surface 0 of cylinder 0 to the last sector on the disk. This allows easy transcription of real disks to files usable by the simulator.

- Magtapes, by convention, use a record based format. Each record consists of a leading 32-bit record length, the record data (padded with a byte of 0 if the record length is odd), and a trailing 32-bit record length. File marks are recorded as one record length of 0.

- Cards have 12 bits of data per column, but the data is most conveniently viewed as (ASCII) characters. Column binary can be implemented using two successive characters per card column.

Data I/O varies between fixed and variable capacity devices, and between buffered and non-buffered devices. A fixed capacity device differs from a variable capacity device in that the file attached to the former has a maximum size, while the file attached to the latter may expand indefinitely. A buffered device differs from a non-buffered device in that the former buffers its data set in host memory, while the latter maintains it as a file. Most variable capacity devices (such as the paper tape reader and punch) are sequential; all buffered devices are fixed capacity.

#### 3.2.3.1 Reading and Writing Data

The ATTACH command creates an association between a host file and an I/O unit. For non-buffered devices, ATTACH stores the file pointer for the host file in the `fileref` field of the UNIT
structure. For buffered devices, ATTACH reads the entire host file into a buffer pointed to by the filebuf field of the UNIT structure. If unit flag UNIT_MUSTBUF is set, the buffer is allocated dynamically; otherwise, it must be statically allocated.

For non-buffered devices, I/O is done with standard C subroutines plus the SCP routines sim_fread and sim_fwrite. sim_fread and sim_fwrite are identical in calling sequence and function to fread and fwrite, respectively, but will correct for endian dependencies. For buffered devices, I/O is done by copying data to or from the allocated buffer. The device code must maintain the number (+1) of the highest address modified in the hwmark field of the UNIT structure. For both the non-buffered and buffered cases, the device must perform all address calculations and positioning operations.

SIMH provides capabilities to access files >2GB (the int32 position limit). If a VM is compiled with flags USE_INT64 and USE_ADDR64 defined, then t_addr is defined as t_uint64 rather than uint32. Routine sim_fseek allows simulated devices to perform random access in large files:

- int sim_fseek (FILE *handle, t_addr position, int where)

sim_fseek is identical to standard C fseek, with two exceptions: where = SEEK_END is not supported, and the position argument can be 64b wide.

The DETACH command breaks the association between a host file and an I/O unit. For buffered devices, DETACH writes the allocated buffer back to the host file.

3.2.3.2 Console I/O

SCP provides three routines for console I/O.

- t_stat sim_poll_char (void). This routine polls for keyboard input. If there is a character, it returns SCPE_KFLAG + the character. If the user typed the interrupt character (^E), it returns SCPE_STOP. If the console is attached to a Telnet connection, and the connection is lost, the routine returns SCPE_LOST. If there is no input, it returns SCPE_OK.

- t_stat sim_putchar (int32 char). This routine types the specified ASCII character to the console. If the console is attached to a Telnet connection, and the connection is lost, the routine returns SCPE_LOST.

- t_stat sim_putchar_s (int32 char). This routine outputs the specified ASCII character to the console. If the console is attached to a Telnet connection, and the connection is lost, the routine returns SCPE_LOST; if the connection is backlogged, the routine returns SCPESTALL.

4. Data Structures

The devices, units, and registers that make up a VM are formally described through a set of data structures which interface the VM to the control portions of SCP. The devices themselves are pointed to by the device list array sim_devices[]. Within a device, both units and registers are allocated contiguously as arrays of structures. In addition, many devices allow the user to set or clear options via a modifications table.

4.1 sim_device Structure
Devices are defined by the `sim_device` structure (typedef `DEVICE`):

```c
struct sim_device {
  char  *name;    /* name */
  struct sim_unit  *units;    /* units */
  struct sim_reg  *registers;    /* registers */
  struct sim_mtab  *modifiers;    /* modifiers */
  int32  numunits;   /* #units */
  uint32  aradix;    /* address radix */
  uint32  awidth;    /* address width */
  uint32  aincr;    /* addr increment */
  uint32  dradix;    /* data radix */
  uint32  dwidth;    /* data width */
  t_stat  (*examine)();   /* examine routine */
  t_stat  (*deposit)();   /* deposit routine */
  t_stat  (*reset)();   /* reset routine */
  t_stat  (*boot)();   /* boot routine */
  t_stat  (*attach)();   /* attach routine */
  t_stat  (*detach)();   /* detach routine */
  void  *ctxt; /* * context */
  uint32  flags;    /* flags */
  uint32  dctrl;    /* debug control flags */
  struct sim_debtab  debflags;    /* debug flag names */
  t_stat  (*msize)();   /* memory size change */
  char  *lname; /* * logical name */
};
```

The fields are the following:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>device name, string of all capital alphanumeric characters.</td>
</tr>
<tr>
<td>units</td>
<td>pointer to array of <code>sim_unit</code> structures, or NULL if none.</td>
</tr>
<tr>
<td>registers</td>
<td>pointer to array of <code>sim_reg</code> structures, or NULL if none.</td>
</tr>
<tr>
<td>modifiers</td>
<td>pointer to array of <code>sim_mtab</code> structures, or NULL if none.</td>
</tr>
<tr>
<td>numunits</td>
<td>number of units in this device.</td>
</tr>
<tr>
<td>aradix</td>
<td>radix for input and display of device addresses, 2 to 16 inclusive.</td>
</tr>
<tr>
<td>awidth</td>
<td>width in bits of a device address, 1 to 31 inclusive.</td>
</tr>
<tr>
<td>aincr</td>
<td>increment between device addresses, normally 1; however, byte addressed devices with 16-bit words specify 2, with 32-bit words 4.</td>
</tr>
<tr>
<td>dradix</td>
<td>radix for input and display of device data, 2 to 16 inclusive.</td>
</tr>
<tr>
<td>dwidth</td>
<td>width in bits of device data, 1 to 32 inclusive.</td>
</tr>
<tr>
<td>examine</td>
<td>address of special device data read routine, or NULL if none is required.</td>
</tr>
<tr>
<td>deposit</td>
<td>address of special device data write routine, or NULL if none is required.</td>
</tr>
<tr>
<td>reset</td>
<td>address of device reset routine, or NULL if none is required.</td>
</tr>
<tr>
<td>boot</td>
<td>address of device bootstrap routine, or NULL if none is required.</td>
</tr>
<tr>
<td>attach</td>
<td>address of special device attach routine, or NULL if none is required.</td>
</tr>
<tr>
<td>detach</td>
<td>address of special device detach routine, or NULL if none is required.</td>
</tr>
<tr>
<td>ctxt</td>
<td>address of VM-specific device context table, or NULL if none is required.</td>
</tr>
<tr>
<td>flags</td>
<td>device flags.</td>
</tr>
<tr>
<td>dctrl</td>
<td>debug control flags.</td>
</tr>
<tr>
<td>debflags</td>
<td>pointer to array of <code>sim_debtab</code> structures, or NULL if none.</td>
</tr>
<tr>
<td>msize</td>
<td>address of memory size change routine, or NULL if none is required.</td>
</tr>
<tr>
<td>lname</td>
<td>pointer to logical name string.</td>
</tr>
</tbody>
</table>

### 4.1.1 Device Flags
The **flags** field contains indicators of current device status. SIMH defines 2 flags:

<table>
<thead>
<tr>
<th>flag name</th>
<th>meaning if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEV_DISABLE</td>
<td>device can be set enabled or disabled</td>
</tr>
<tr>
<td>DEV_DIS</td>
<td>device is currently disabled</td>
</tr>
<tr>
<td>DEV_DYNM</td>
<td>device requires call on <strong>msize</strong> routine to change memory size</td>
</tr>
<tr>
<td>DEV_NET</td>
<td>device attaches to the network rather than a file</td>
</tr>
<tr>
<td>DEV_DEBUG</td>
<td>device supports SET DEBUG command</td>
</tr>
<tr>
<td>DEV_RAW</td>
<td>device supports raw I/O</td>
</tr>
<tr>
<td>DEV_RAWONLY</td>
<td>device supports only raw I/O</td>
</tr>
</tbody>
</table>

Starting at bit position DEV_V_UF, the remaining flags are device-specific. Device flags are automatically saved and restored; the device need not supply a register for these bits.

### 4.1.2 Context

The field contains a pointer to a VM-specific device context table, if required. SIMH never accesses this field. The context field allows VM-specific code to walk VM-specific data structures from the **sim_devices** root pointer.

### 4.1.3 Examine and Deposit Routines

For devices which maintain their data sets as host files, SCP implements the examine and deposit data functions. However, devices which maintain their data sets as private state (for example, the CPU) must supply special examine and deposit routines. The calling sequences are:

```c
  t_stat examine_routine (t_val *eval_array, t_addr addr, UNIT *uptr, int32 switches) –
  Copy sim_emax consecutive addresses for unit uptr, starting at addr, into eval_array.
  The switch variable has bit<n> set if the n’th letter was specified as a switch to the examine command.

  t_stat deposit_routine (t_val value, t_addr addr, UNIT *uptr, int32 switches) – Store the specified value in the specified addr for unit uptr. The switch variable is the same as for the examine routine.
```

### 4.1.4 Reset Routine

The reset routine implements the device reset function for the RESET, RUN, and BOOT commands. Its calling sequence is:

```c
  t_stat reset_routine (DEVICE *dptr) – Reset the specified device to its initial state.
```

A typical reset routine clears all device flags and cancels any outstanding timing operations. Switch –p specifies a reset to power-up state.

### 4.1.5 Boot Routine

If a device responds to a BOOT command, the boot routine implements the bootstrapping function. Its calling sequence is:

```c
  t_stat boot_routine (int32 unit_num, DEVICE *dptr) – Bootstrap unit unit_num on the device dptr.
```
A typical bootstrap routine copies a bootstrap loader into main memory and sets the PC to the starting address of the loader. SCP then starts simulation at the specified address.

4.1.6 Attach and Detach Routines

Normally, the ATTACH and DETACH commands are handled by SCP. However, devices which need to pre- or post-process these commands must supply special attach and detach routines. The calling sequences are:

\[
\begin{align*}
t\text{_stat} & \quad \text{attach\_routine} \quad (\text{UNIT } *\text{uptr}, \text{char } *\text{file}) \quad \text{-- Attach the specified file to the unit uptr.} \\
& \quad \text{Sim\_switches contains the command switch; bit SIM\_SW\_REST indicates that attach is} \\
& \quad \text{being called by the RESTORE command rather than the ATTACH command.} \\
t\text{_stat} & \quad \text{detach\_routine} \quad (\text{UNIT } *\text{uptr}) \quad \text{-- Detach unit uptr.}
\end{align*}
\]

In practice, these routines usually invoke the standard SCP routines, \text{attach\_unit} and \text{detach\_unit}, respectively. For example, here are special attach and detach routines to update line printer error state:

\[
\begin{align*}
t\text{_stat} & \quad \text{lpt\_attach} \quad (\text{UNIT } *\text{uptr}, \text{char } *\text{cptr}) \quad \{ \\
& \quad t\text{_stat }r; \\
& \quad \text{if } \left( (r = \text{attach\_unit} \quad (\text{uptr}, \text{cptr})) \neq \text{SCPE\_OK} \right) \text{return } r; \\
& \quad \text{lpt\_error } = 0; \\
& \quad \text{return } \text{SCPE\_OK}; \\
\} \\
t\text{_stat} & \quad \text{lpt\_detach} \quad (\text{UNIT } *\text{uptr}) \quad \{ \\
& \quad \text{lpt\_error } = 1; \\
& \quad \text{return } \text{detach\_unit} \quad (\text{uptr}); \\
\}
\]

SCP executes a DETACH ALL command as part of simulator exit. Normally, DETACH ALL only calls a unit’s detach routine if the unit’s UNIT\_ATT flag is set. During simulator exit, the detach routine is also called if the unit is not flagged as attachable (UNIT\_ATTABLE is not set). This allows the detach routine of a non-attachable unit to function as a simulator-specific cleanup routine for the unit, device, or entire simulator.

4.1.7 Memory Size Change Routine

Most units instantiate any memory array at the maximum size possible. This allows apparent memory size to be changed by varying the capacity field in the unit structure. For some devices (like the VAX CPU), instantiating the maximum memory size would impose a significant resource burden if less memory was actually needed. These devices must provide a routine, the memory size change routine, for RESTORE to use if memory size must be changed:

\[
\begin{align*}
t\text{_stat} & \quad \text{change\_mem\_size} \quad (\text{UNIT } *\text{uptr}, \text{int32 } \text{val}, \text{char } *\text{cptr}, \text{void } *\text{desc}) \quad \text{-- Change the} \\
& \quad \text{capacity (memory size) of unit uptr to val. The cptr and desc arguments are included for} \\
& \quad \text{compatibility with the SET command’s validation routine calling sequence.}
\end{align*}
\]

4.1.8 Debug Controls
Devices can support debug printouts. Debug printouts are controlled by the SET {NO}DEBUG command, which specifies where debug output should be printed; and by the SET <device> {NO}DEBUG command, which enables or disables individual debug printouts.

If a device supports debug printouts, device flag DEV_DEBUG must be set. Field dctrl is used for the debug control flags. If a device supports only a single debug on/off flag, then the debflags field should be set to NULL. If a device supports multiple debug on/off flags, then the correspondence between bit positions in dctrl and debug flag names is specified by table debflags. debflags points to a contiguous array of sim_debtab structures (typedef DEBTAB). Each sim_debtab structure specifies a single debug flag:

```
struct sim_debtab {
    char   name;     /* flag name */
    uint32 mask;    /* control bit */
};
```

The fields are the following:

- **name**: name of the debug flag.
- **mask**: bit mask of the debug flag.

The array is terminated with a NULL entry.

### 4.2 sim_unit Structure

Units are allocated as contiguous array. Each unit is defined with a sim_unit structure (typedef UNIT):

```
struct sim_unit {
    struct sim_unit        *next;    /* next active */
    t_stat     (*action)();    /* action routine */
    char       *filename;      /* open file name */
    FILE       *fileref;       /* file reference */
    void       *filebuf;       /* memory buffer */
    uint32     hwmark;        /* high water mark */
    int32      time;          /* time out */
    uint32     flags;         /* flags */
    t_addr     capac;         /* capacity */
    t_addr     pos;           /* file position */
    int32      buf;           /* buffer */
    int32      wait;          /* wait */
    int32      u3;            /* device specific */
    int32      u4;            /* device specific */
    int32      u5;            /* device specific */
    int32      u6;            /* device specific */
};
```

The fields are the following:

- **next**: pointer to next unit in active queue, NULL if none.
- **action**: address of unit time-out service routine.
- **filename**: pointer to name of attached file, NULL if none.
- **fileref**: pointer to FILE structure of attached file, NULL if none.
- **hwmark**: buffered devices only; highest modified address, + 1.
- **time**: increment until time-out beyond previous unit in active queue.
flags        unit flags.
capac        unit capacity, 0 if variable.
pos          sequential devices only; next device address to be read or written.
buf          by convention, the unit buffer, but can be used for other purposes.
wait         by convention, the unit wait time, but can be used for other purposes.
u3            user-defined.
u4            user-defined.
u5            user-defined.
u6            user-defined.

buf, wait, u3, u4, u5, u6, and parts of flags are all saved and restored by the SAVE and
RESTORE commands and thus can be used for unit state which must be preserved.

Macro UDATA is available to fill in the common fields of a UNIT. It is invoked by

UDATA (action_routine, flags, capacity)

Fields after buf can be filled in manually, e.g,

    UNIT lpt_unit =
    { UDATA (&lpt_svc, UNIT_SEQ+UNIT_ATTABLE, 0), 500 };

defines the line printer as a sequential unit with a wait time of 500.

4.2.1 Unit Flags

The flags field contains indicators of current unit status. SIMH defines 12 flags:

<table>
<thead>
<tr>
<th>flag name</th>
<th>meaning if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT_ATTABLE</td>
<td>the unit responds to ATTACH and DETACH.</td>
</tr>
<tr>
<td>UNIT_RO</td>
<td>the unit is currently read only.</td>
</tr>
<tr>
<td>UNIX_FIX</td>
<td>the unit is fixed capacity.</td>
</tr>
<tr>
<td>UNIT_SEQ</td>
<td>the unit is sequential.</td>
</tr>
<tr>
<td>UNIT_ATT</td>
<td>the unit is currently attached to a file.</td>
</tr>
<tr>
<td>UNIT_BINK</td>
<td>the unit measures “K” as 1024, rather than 1000.</td>
</tr>
<tr>
<td>UNIT_BUFABLE</td>
<td>the unit buffers its data set in memory.</td>
</tr>
<tr>
<td>UNIT_MUSTBUF</td>
<td>the unit allocates its data buffer dynamically.</td>
</tr>
<tr>
<td>UNIT_BUF</td>
<td>the unit is currently buffering its data set in memory.</td>
</tr>
<tr>
<td>UNIT_ROABLE</td>
<td>the unit can be ATTACHED read only.</td>
</tr>
<tr>
<td>UNIT_DISABLE</td>
<td>the unit responds to ENABLE and DISABLE.</td>
</tr>
<tr>
<td>UNIT_DIS</td>
<td>the unit is currently disabled.</td>
</tr>
<tr>
<td>UNIT_RAW</td>
<td>the unit is attached in RAW mode.</td>
</tr>
</tbody>
</table>

Starting at bit position UNIT_V_UF, the remaining flags are unit-specific. Unit-specific flags are
set and cleared with the SET and CLEAR commands, which reference the MTAB array (see
below). Unit-specific flags and UNIT_DIS are automatically saved and restored; the device need
not supply a register for these bits.

4.2.2 Service Routine

This routine is called by sim_process_event when a unit times out. Its calling sequence is:

    t_stat service_routine (UNIT *uptr)
The status returned by the service routine is passed by `sim_process_event` back to the CPU.

### 4.3 sim_reg Structure

Registers are allocated as contiguous array, with a NULL register at the end. Each register is defined with a `sim_reg` structure (typedef `REG`):

```c
struct reg {
    char  *name;    /* name */
    void   *loc;     /* location */
    uint32  radix;    /* radix */
    uint32  width;    /* width */
    uint32  offset;    /* starting bit */
    uint32  depth;    /* save depth */
    uint32  flags;    /* flags */
    uint32  qptr;    /* current queue pointer */
};
```

The fields are the following:

- **name**: device name, string of all capital alphanumeric characters.
- **loc**: pointer to location of the register value.
- **radix**: radix for input and display of data, 2 to 16 inclusive.
- **width**: width in bits of data, 1 to 32 inclusive.
- **offset**: bit offset (from right end of data).
- **depth**: size of data array (normally 1).
- **flags**: flags and formatting information.
- **qptr**: for a circular queue, the entry number for the first entry

The **depth** field is used with “arrayed registers”. Arrayed registers are used to represent structures with multiple data values, such as the locations in a transfer buffer; or structures which are replicated in every unit, such as a drive status register. The **qptr** field is used with “queued registers”. Queued registers are arrays that are organized as circular queues, such as the PC change queue.

Macros `ORDATA`, `DRDATA`, and `HRDATA` define right-justified octal, decimal, and hexadecimal registers, respectively. They are invoked by:

```c
xRDATA (name, location, width)
```

Macro `FLDATA` defines a one-bit binary flag at an arbitrary offset in a 32-bit word. It is invoked by:

```c
FLDATA (name, location, bit_position)
```

Macro `GRDATA` defines a register with arbitrary location and radix. It is invoked by:

```c
GRDATA (name, location, radix, width, bit_position)
```

Macro `BRDATA` defines an arrayed register whose data is kept in a standard C array. It is invoked by:

```c
BRDATA (name, location, radix, width, depth)
```

For all of these macros, the **flag** field can be filled in manually, e.g.,
Finally, macro `URDATA` defines an arrayed register whose data is part of the `UNIT` structure. This macro must be used with great care. If the fields are set up wrong, or the data is actually kept somewhere else, storing through this register declaration can trample over memory. The macro is invoked by:

```c
URDATA (name, location, radix, width, offset, depth, flags)
```

The location should be an offset in the `UNIT` structure for unit 0. The width should be 32 for an int32 or uint32 field, and `T_ADDR_W` for a `t_addr` field. The flags can be any of the normal register flags; `REG_UNIT` will be OR’d in automatically. For example, the following declares an arrayed register of all the `UNIT` position fields in a device with 4 units:

```c
{ URDATA (POS, dev_unit[0].pos, 8, T_ADDR_W, 0, 4, 0) }
```

### 4.3.1 Register Flags

The `flags` field contains indicators that control register examination and deposit.

<table>
<thead>
<tr>
<th>flag name</th>
<th>meaning if specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>PV_RZRO</td>
<td>print register right justified with leading zeroes.</td>
</tr>
<tr>
<td>PV_RSPC</td>
<td>print register right justified with leading spaces.</td>
</tr>
<tr>
<td>PV_LEFT</td>
<td>print register left justified.</td>
</tr>
<tr>
<td>REG_RO</td>
<td>register is read only.</td>
</tr>
<tr>
<td>REG_HIDDEN</td>
<td>register is hidden (will not appear in EXAMINE STATE).</td>
</tr>
<tr>
<td>REG_HRO</td>
<td>register is read only and hidden.</td>
</tr>
<tr>
<td>REG_NZ</td>
<td>new register values must be non-zero.</td>
</tr>
<tr>
<td>REG_UNIT</td>
<td>register resides in the <code>UNIT</code> structure.</td>
</tr>
<tr>
<td>REG_CIRC</td>
<td>register is a circular queue.</td>
</tr>
<tr>
<td>REG_VMIO</td>
<td>register is displayed and parsed using VM data routines.</td>
</tr>
<tr>
<td>REG_VMAD</td>
<td>register is displayed and parsed using VM address routines.</td>
</tr>
</tbody>
</table>

### 4.4 sim_mtab Structure

Device-specific SHOW and SET commands are processed using the modifications array, which is allocated as contiguous array, with a NULL at the end. Each possible modification is defined with a `sim_mtab` structure (synonym `MTAB`), which has the following fields:

```c
struct sim_mtab {
    uint32 mask;    /* mask */
    uint32 match;   /* match */
    char *pstring;  /* print string */
    char *mstring;  /* match string */
    t_stat (*valid)();  /* validation routine */
    t_stat (*disp)();  /* display routine */
    void *desc;      /* location descriptor */
};
```

`MTAB` supports two different structure interpretations: regular and extended. A regular `MTAB` entry modifies flags in the `UNIT` flags word; the descriptor entry is not used. The fields are the following:
mask  bit mask for testing the unit.flags field
match  value to be stored (SET) or compared (SHOW)
pstring  pointer to character string printed on a match (SHOW), or NULL
mstring  pointer to character string to be matched (SET), or NULL
valid  address of validation routine (SET), or NULL
disp  address of display routine (SHOW), or NULL

For SET, a regular MTAB entry is interpreted as follows:

1. Test to see if the mstring entry exists.
2. Test to see if the SET parameter matches the mstring.
3. Call the validation routine, if any.
4. Apply the mask value to the UNIT flags word and then or in the match value.

For SHOW, a regular MTAB entry is interpreted as follows:

1. Test to see if the pstring entry exists.
2. Test to see if the UNIT flags word, masked with the mask value, equals the match value.
3. If a display routine exists, call it, otherwise
4. Print the pstring.

Extended MTAB entries have a different interpretation:

mask  entry flags
MTAB_XTD  extended entry
MTAB_VDV  valid for devices
MTAB_VUN  valid for units
MTAB_VAL  takes a value
MTAB_NMO  valid only in named SHOW
MTAB_NC  do not convert option value to upper case
MTAB_SHP  SHOW parameter takes optional value

match  value to be stored (SET)
pstring  pointer to character string printed on a match (SHOW), or NULL
mstring  pointer to character string to be matched (SET), or NULL
valid  address of validation routine (SET), or NULL
disp  address of display routine (SHOW), or NULL
desc  pointer to a REG structure (MTAB_VAL set) or a validation-specific structure (MTAB_VAL clear)

For SET, an extended MTAB entry is interpreted as follows:

1. Test to see if the mstring entry exists.
2. Test to see if the SET parameter matches the mstring.
3. Test to see if the entry is valid for the type of SET being done (SET device or SET unit).
4. If a validation routine exists, call it and return its status. The validation routine is responsible for storing the result.
5. If desc is NULL, exit.
6. If MTAB_VAL is set, parse the SET option for "option=n", and store the value n in the register described by desc.
7. Otherwise, store the match value in the int32 pointed to by desc.

For SHOW, an extended MTAB entry is interpreted as follows:
1. Test to see if the `pstring` entry exists.
2. Test to see if the entry is valid for the type of SHOW being done (device or unit).
3. If a display routine exists, call it, otherwise,
4. If MTAB_VAL is set, print "=n", where the value, radix, and width are taken from the register described by `desc`, otherwise,
5. Print the `pstring`.

SHOW [dev|unit] <modifier>{=<value>} is a special case. Only two kinds of modifiers can be displayed individually: an extended MTAB entry that takes a value; and any MTAB entry with both a display routine and a `pstring`. Recall that if a display routine exists, SHOW does not use the `pstring` entry. For displaying a named modifier, `pstring` is used as the string match. This allows implementation of complex display routines that are only invoked by name, e.g.,

```c
MTAB cpu_tab[] = {
    { mask, value, "normal", "NORMAL", NULL, NULL, NULL },
    { MTAB_XTD|MTAB_VDV|MTAB_NMO, 0, "SPECIAL",
      NULL, NULL, NULL, &spec_disp },
    { 0 } }
```

A SHOW CPU command will display only the modifier named NORMAL; but SHOW CPU SPECIAL will invoke the special display routine.

### 4.4.1 Validation Routine

The validation routine can be used to validate input during SET processing. It can make other state changes required by the modification or initiate additional dialogs needed by the modifier. Its calling sequence is:

```c
t_stat validation_routine (UNIT *uptr, int32 value, char *cptr, void *desc) – test that uptr.flags can be set to value. cptr points to the value portion of the parameter string (any characters after the = sign); if cptr is NULL, no value was given. desc points to the REG or int32 used to store the parameter.
```

### 4.4.2 Display Routine

The display routine is called during SHOW processing to display device- or unit-specific state. Its calling sequence is:

```c
t_stat display_routine (FILE *st, UNIT *uptr, void *desc) – output device- or unit-specific state for uptr to stream st. If the modifier is regular MTAB entry, or an extended entry without MTAB_SHP set, desc points to the structure in the MTAB entry. If the modifier is an extended MTAB entry with MTAB_SHP set, desc points to the optional value string or is NULL if no value was supplied.
```

When the display routine is called for a regular MTAB entry, SHOW has output the `pstring` argument but has not appended a newline. When it is called for an extended MTAB entry, SHOW hasn’t output anything. SHOW will append a newline after the display routine returns, except for entries with the MTAB_NMO flag set.

### 4.5 Other Data Structures

char `sim_name[]` is a character array containing the VM name.
int32 sim_emax contains the maximum number of words needed to hold the largest instruction or
data item in the VM. Examine and deposit will process up to sim_emax words.

DEVICE *sim_devices[] is an array of pointers to all the devices in the VM. It is terminated by a
NULL. By convention, the CPU is always the first device in the array.

REG *sim_PC points to the reg structure for the program counter. By convention, the PC is
always the first register in the CPU’s register array.

cchar *sim_stop_messages[] is an array of pointers to character strings, corresponding to error
status returns greater than zero. If sim_instr returns status code n > 0, then
sim_stop_message[n] is printed by SCP.

5. VM Provided Routines

5.1 Instruction Execution

Instruction execution is performed by routine sim_instr. Its calling sequence is:

   t_stat sim_instr (void) – execute from current PC until error or halt.

5.2 Binary Load and Dump

If the VM responds to the LOAD (or DUMP) command, the load routine (dump routine) is
implemented by routine sim_load. Its calling sequence is:

   t_stat sim_load (FILE *fptr, char *buf, char *fnam, t_bool flag) - If flag = 0, load data from
binary file fptr. If flag = 1, dump data to binary file fptr. For either command, buf contains
any VM-specific arguments, and fnam contains the file name.

If LOAD or DUMP is not implemented, sim_load should simply return SCPE_ARG. The LOAD
and DUMP commands open and close the specified file for sim_load.

5.3 Symbolic Examination and Deposit

If the VM provides symbolic examination and deposit of data, it must provide two routines,
fprint_sym for output and parse_sym for input. Their calling sequences are:

   t_stat fprint_sym (FILE *ofile, t_addr addr, t_value *val, UNIT *uptr, int32 switch) –
Based on the switch variable, symbolically output to stream ofile the data in array val at
the specified addr in unit uptr.

   t_stat parse_sym (char *cptr, t_addr addr, UNIT *uptr, t_value *val, int32 switch) – Based
on the switch variable, parse character string cptr for a symbolic value val at the specified
addr in unit uptr.

If symbolic processing is not implemented, or the output value or input string cannot be parsed,
these routines should return SCPE_ARG. If the processing was successful and consumed more
than a single word, then these routines should return extra number of words (not bytes)
consumed as a negative number. If the processing was successful and consumed a single
word, then these routines should return SCPE_OK. For example, PDP-11 parse_sym would
respond as follows to various inputs:
The interpretation of switch values is arbitrary, but the following are used by existing VM’s:

<table>
<thead>
<tr>
<th>switch</th>
<th>interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>single character</td>
</tr>
<tr>
<td>-c</td>
<td>character string</td>
</tr>
<tr>
<td>-m</td>
<td>instruction mnemonic</td>
</tr>
</tbody>
</table>

In addition, on input, a leading ' (apostrophe) is interpreted to mean a single character, and a leading " (double quote) is interpreted to mean a character string.

### 5.4Optional Interfaces

For greater flexibility, SCP provides some optional interfaces that can be used to extend its command input, command processing, and command post-processing capabilities. These interfaces are strictly optional and are off by default. Using them requires intimate knowledge of how SCP functions internally and is not recommended to the novice VM writer.

#### 5.4.1 Once Only Initialization Routine

SCP defines a pointer (*sim_vm_init)(void). This is a “weak global”; if no other module defines this value, it will default to NULL. A VM requiring special initialization should fill in this pointer with the address of its special initialization routine:

```c
void sim_special_init (void);
void (*sim_vm_init)(void) = &sim_special_init;
```

The special initialization routine can perform any actions required by the VM. If the other optional interfaces are to be used, the initialization routine can fill in the appropriate pointers; however, this can just as easily be done in the CPU reset routine.

#### 5.4.2 Address Input and Display

SCP defines a pointer t_addr *(sim_vm_parse_addr)(DEVICE *, char *, char **). This is initialized to NULL. If it is filled in by the VM, SCP will use the specified routine to parse addresses in place of its standard numerical input routine. The calling sequence for the sim_vm_parse_addr routine is:

```c
t_addr sim_vm_parse_addr (DEVICE *dptr, char *cptr, char **optr) – parse the string pointed to by cptr as an address for the device pointed to by dptr. optr points to the first character not successfully parsed. If cptr == optr, parsing failed.
```

SCP defines a pointer void *(sim_vm_fprint_addr)(FILE *, DEVICE *, t_addr). This is initialized to NULL. If it is filled in by the VM, SCP will use the specified routine to print addresses in place of its standard numerical output routine. The calling sequence for the sim_vm_fprint_addr routine is:
t_addr sim_vm_fprint_addr (FILE *stream, DEVICE *dptr, t_addr addr) – output address addr to stream in the format required by the device pointed to by dptr.

5.4.3 Command Input and Post-Processing

SCP defines a pointer char* (sim_vm_read)(char *, int32 *, FILE *). This is initialized to NULL. If it is filled in by the VM, SCP will use the specified routine to obtain command input in place of its standard routine, read_line. The calling sequence for the sim_vm_read routine is:

```c
char sim_vm_input (char *buf, int32 *max, FILE *stream) – read the next command line from stream and store it in buf, up to a maximum of max characters
```

The routine is expected to strip off leading whitespace characters and to return NULL on end of file.

SCP defines a pointer void *(sim_vm_post)(t_bool from_scp). This is initialized to NULL. If filled in by the VM, SCP will call the specified routine at the end of every command. This allows the VM to update any local state, such as a GUI console display. The calling sequence for the vm_post routine is:

```c
void sim_vm_postupdate (t_bool from_scp) – if called from SCP, the argument from_scp is TRUE; otherwise, it is FALSE.
```

5.4.4 VM-Specific Commands

SCP defines a pointer CTAB *sim_vm_cmd. This is initialized to NULL. If filled in by the VM, SCP interprets it as a pointer to SCP command table. This command table is checked before user input is looked up in the standard command table.

A command table is allocated as a contiguous array. Each entry is defined with a sim_ctab structure (typedef CTAB):

```c
struct sim_ctab {
    char  *name;    /* name */
    t_stat (*action)();   /* action routine */
    int32  arg;    /* argument */
    char  *help;    /* help string */
};
```

If the first word of a command line matches ctab.name, then the action routine is called with the following arguments:

```c
t_stat action_routine (int32 arg, char *buf) – process input string buf based on optional argument arg
```

The string passed to the action routine starts at the first non-blank character past the command name.

6. Other SCP Facilities

6.1 Terminal Multiplexor Emulation Library

SIMH supports the use of multiple terminals. All terminals except the console are accessed via Telnet. SIMH provides two supporting libraries for implementing multiple terminals: sim_tmxr.c
(and its header file, sim_tmxr.h), which provide OS-independent support routines for terminal
multiplexors; and sim_sock.c (and its header file, sim_sock.h), which provide OS-dependent
socket routines. Sim_sock.c is implemented under Windows, VMS, UNIX, and MacOS.

Two basic data structures define the multiple terminals. Individual lines are defined by an array of
tmln structures (typedef TMLN):

```c
struct tmln {
    SOCKET conn;    /* line conn */
    uint32 ipad;    /* IP address */
    uint32 cnms;    /* connect time ms */
    int32 tsta;    /* Telnet state */
    int32 rcve;    /* rcv enable */
    int32 xmt;    /* xmt enable */
    int32 dstb;    /* disable Tlnt bin */
    int32 rxbpr;    /* rcv buf remove */
    int32 rxbpi;    /* rcv buf insert */
    int32 rxcnt;    /* rcv count */
    int32 txbpr;    /* xmt buf remove */
    int32 txbpi;    /* xmt buf insert */
    int32 txcnt;    /* xmt count */
    FILE *txlog;    /* xmt log file */
    char *txlogname;  /* xmt log file name */
    char rxb[TMXR_MAXBUF];  /* rcv buffer */
    char rbr[TMXR_MAXBUF];  /* rcv break */
    char txb[TMXR_MAXBUF];  /* xmt buffer */
};
```

The fields are the following:

- **conn**: connection socket (0 = disconnected)
- **tsta**: Telnet state
- **rcve**: receive enable flag (0 = disabled)
- **xmte**: transmit flow control flag (0 = transmit disabled)
- **dstb**: Telnet bin mode disabled
- **rxbpr**: receive buffer remove pointer
- **rxbpi**: receive buffer insert pointer
- **rxcnt**: receive count
- **txbpr**: transmit buffer remove pointer
- **txbpi**: transmit buffer insert pointer
- **txcnt**: transmit count
- **txlog**: pointer to log file descriptor
- **txlogname**: pointer to log file name
- **rxb**: receive buffer
- **rbr**: receive buffer break flags
- **txb**: transmit buffer

The overall set of extra terminals is defined by the tmxr structure (typedef TMXR):

```c
struct tmxr {
    int32 lines;    /* # lines */
    int32 port;    /* listening port */
    SOCKET master;    /* master socket */
    TMLN *ldsc;    /* pointer to line descriptors */
};
```
The fields are the following:

- **lines**: number of lines (constant)
- **port**: master listening port (specified by ATTACH command)
- **master**: master listening socket (filled in by ATTACH command)
- **ldsc**: array of line descriptors

Library sim_tmxr.c provides the following routines to support Telnet-based terminals:

- **int32 tmxr_poll_conn (TMXR *mp)** – poll for a new connection to the terminals described by `mp`. If there is a new connection, the routine resets all the line descriptor state (including receive enable) and returns the line number (index to line descriptor) for the new connection. If there isn’t a new connection, the routine returns -1.

- **void tmxr_reset_ln (TMLN *lp)** – reset the line described by `lp`. The connection is closed and all line descriptor state is reset.

- **int32 tmxr_getc_ln (TMLN *lp)** – return the next available character from the line described by `lp`. If a character is available, the return variable is:

  \[(1 << TMXR_V_VALID) | \text{character}\]

  If no character is available, the return variable is 0.

- **void tmxr_poll_rx (TMXR *mp)** – poll for input available on the terminals described by `mp`.

- **void tmxr_rqln (TMLN *lp)** – return the number of characters in the receive queue of the line described by `lp`.

- **t_stat tmxr_putc_ln (TMLN *lp, int32 chr)** – output character `chr` to the line described by `lp`. Possible errors are SCPE_LOST (connection lost) and SCPE_STALL (connection backlogged).

- **void tmxr_poll_tx (TMXR *mp)** – poll for output complete on the terminals described by `mp`.

- **void tmxr_tqln (TMLN *lp)** – return the number of characters in the transmit queue of the line described by `lp`.

- **t_stat tmxr_attach (TMXR *mp, UNIT *uptr, char *cptr)** – attach the port contained in character string `cptr` to the terminals described by `mp` and unit `uptr`.

- **t_stat tmxr_open_master (TMXR *mp, char *cptr)** – associate the port contained in character string `cptr` to the terminals described by `mp`. This routine is a subset of `tmxr_attach`.

- **t_stat tmxr_detach (TMXR *mp, UNIT *uptr)** – detach all connections for the terminals described by `mp` and unit `uptr`.

- **t_stat tmxr_close_master (TMXR *mp)** – close the master port for the terminals described by `mp`. This routine is a subset of `tmxr_detach`.

- **t_stat tmxr_ex (t_value *vptr, t_addr addr, UNIT *uptr, int32 sw)** – stub examine routine, needed because the extra terminals are marked as attached; always returns an error.
t_stat tmxr_dep (t_value val, t_addr addr, UNIT *uptr, int32 sw) – stub deposit routine, needed because the extra terminals are marked as detached; always returns an error.

void tmxr_linemsg (TMLN *lp, char *msg) – output character string msg to line lp.

void tmxr_fconns (FILE *st, TMLN *lp, int32 ln) – output connection status to stream st for the line described by lp. If ln is >= 0, preface the output with the specified line number.

void tmxr_fstats (FILE *st, TMLN *lp, int32 ln) – output connection statistics to stream st for the line described by lp. If ln is >= 0, preface the output with the specified line number.

t_stat tmxr_dscln (UNIT *uptr, int32 val, char *cptr, void *mp) – parse the string pointed to by cptr for a decimal line number. If the line number is valid, disconnect the specified line in the terminal multiplexor described by mp. The calling sequence allows tmxr_dscln to be used as an MTAB processing routine.

The OS-dependent socket routines should not need to be accessed by the terminal simulators.

6.2 Magnetic Tape Emulation Library

SIMH supports the use of emulated magnetic tapes. Magnetic tapes are emulated as disk files containing both data records and metadata markers; the format is fully described in the paper “SIMH Magtape Representation and Handling”. SIMH provides a supporting library, sim_tape.c (and its header file, sim_tape.h), that abstracts handling of magnetic tapes. This allows support for multiple tape formats, without change to magnetic device simulators.

The magtape library does not require any special data structures. However, it does define some additional unit flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTUF_WLK</td>
<td>unit is write locked</td>
</tr>
</tbody>
</table>

If magtape simulators need to define private unit flags, those flags should begin at bit number MTUF_V_UF instead of UNIT_V_UF. The magtape library maintains the current magtape position in the pos field of the UNIT structure.

Library sim_tape.c provides the following routines to support emulated magnetic tapes:

- **t_stat sim_tape_attach** (UNIT *uptr, char *cptr) – Attach tape unit uptr to file cptr. Tape Simulators should call this routine, rather than the standard attach_unit routine, to allow for future expansion of format support.

- **t_stat sim_tape_detach** (UNIT *uptr) – Detach tape unit uptr from its current file.

- **t_stat sim_tape_set_fmt** (UNIT *uptr, int32 val, char *cptr, void *desc) – Set the tape format for unit uptr to the format specified by string cptr.

- **t_stat sim_tape_show_fmt** (FILE *st, UNIT *uptr, int32 val, void *desc) – Write the tape format for unit uptr to the file specified by descriptor st.

- **t_stat sim_tape_rdrecf** (UNIT *uptr, uint8 *buf, t_mtrlnt *tbc, t_mtrlnt max) – Forward read the next record on unit uptr into buffer buf of size max. Return the actual record size in tbc.
t_stat sim_tape_rdrecr (UNIT *uptr, uint8 *buf, t_mtrlnt *tbc, t_mtrlnt max) – Reverse read the next record on unit uptr into buffer buf of size max. Return the actual record size in tbc. Note that the record is returned in forward order, that is, byte 0 of the record is stored in buf[0], and so on.

t_stat sim_tape_wrrecf (UNIT *uptr, uint8 *buf, t_mtrlnt tbc) – Write buffer uptr of size tbc as the next record on unit uptr.

t_stat sim_tape_sprecf (UNIT *uptr, t_mtrlnt *tbc) – Space unit uptr forward one record. The size of the record is returned in tbc.

t_stat sim_tape_sprecr (UNIT *uptr, t_mtrlnt *tbc) – Space unit uptr reverse one record. The size of the record is returned in tbc.

t_stat sim_tape_wrtmk (UNIT *uptr) – Write a tape mark on unit uptr.

t_stat sim_tape_wreom (UNIT *uptr) – Write an end-of-medium marker on unit uptr (this effectively erases the rest of the tape).

t_stat sim_tape_rewind (UNIT *uptr) – Rewind unit uptr. This operation succeeds whether or not the unit is attached to a file.

t_stat sim_tape_reset (UNIT *uptr) – Reset unit uptr. This routine should be called when a tape unit is reset.

t_bool sim_tape_bot (UNIT *uptr) – Return TRUE if unit uptr is at beginning-of-tape.

t_bool sim_tape_wrp (UNIT *uptr) – Return TRUE if unit uptr is write-protected.

t_bool sim_tape_eot (UNIT *uptr, t_addr cap) – Return TRUE if unit uptr has exceed the capacity specified by cap.

Sim_tape_attach, sim_tape_detach, sim_tape_set_fmt, and sim_tape_show_fmt return standard SCP status codes; the other magtape library routines return private codes for success and failure. The currently defined magtape status codes are:

```c
MTSE_OK  operation successful
MTSE_UNATT  unit is not attached to a file
MTSE_FMT  unit specifies an unsupported tape file format
MTSE_IOERR  host operating system I/O error during operation
MTSE_INVRL  invalid record length (exceeds maximum allowed)
MTSE_RECE  record header contains error flag
MTSE_TMK  tape mark encountered
MTSE_BOT  beginning of tape encountered during reverse operation
MTSE_EOM  end of medium encountered
MTSE_WRP  write protected unit during write operation
```

Sim_tape_set_fmt and sim_tape_show_fmt should be referenced by an entry in the tape device’s modifier list, as follows:

```c
MTAB tape_mod[] = {
    { MTAB_XTD|MTAB_VDV, 0, “FORMAT”, “FORMAT”,
        &sim_tape_set_fmt, &sim_tape_show_fmt, NULL }, ... 
};
```
6.3 Breakpoint Support

SCP provides underlying mechanisms to track multiple breakpoints of different types. Most VM's implement at least instruction execution breakpoints (type E); but a VM might also allow for break on read (type R), write (type W), and so on. Up to 26 different breakpoint types, identified by the letters A through Z, are supported.

The VM interface to the breakpoint package consists of three variables and one subroutine:

- `sim_brk_types` – initialized by the VM (usually in the CPU reset routine) to a mask of all supported breakpoints.
- `sim_brk_dflt` – initialized by the VM to the mask for the default breakpoint type.
- `sim_brk_summ` – maintained by SCP, providing a bit mask summary of whether any breakpoints of a particular type have been defined.

If the VM only implements one type of breakpoint, then `sim_brk_summ` is non-zero if any breakpoints are set.

To test whether a breakpoint of particular type is set for an address, the VM calls

```
t_bool sim_brk_test (t_addr addr, int32 typ) – test to see if a breakpoint of type `typ` is set for location `addr`
```

Because `sim_brk_test` can be a lengthy procedure, it is usually prefaced with a test of `sim_brk_summ`:

```
if (sim_brk_summ && sim_brk_test (PC, SWMASK ('E'))) {
    <execution break> }
```