

# Writing a Simulator for the SIMH System

## Revised 2-Jul-2024 for SIMH V3.12

### COPYRIGHT NOTICE

The following copyright notice applies to the SIMH source, binary, and documentation:

Original code published in 1993-2024, written by Robert M Supnik  
Copyright (c) 1993-2024, Robert M Supnik

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL ROBERT M SUPNIK BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of Robert M Supnik shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from Robert M Supnik.

<b>1. Overview.....</b>	<b>4</b>
<b>2. Data Types .....</b>	<b>4</b>
<b>3. VM Organization.....</b>	<b>5</b>
<b>3.1 CPU Organization .....</b>	<b>6</b>
3.1.1 Time Base .....	6
3.1.2 Step Function .....	6
3.1.3 Memory Organization .....	7
3.1.4 Interrupt Organization .....	7
3.1.5 I/O Dispatching .....	8
3.1.6 Instruction Execution .....	8
<b>3.2 Peripheral Device Organization .....</b>	<b>9</b>
3.2.1 Device Timing .....	10
3.2.2 Clock Calibration .....	11
3.2.3 Idling .....	11
3.2.4 Data I/O .....	12
<b>4. Data Structures.....</b>	<b>13</b>
<b>4.1 sim_device Structure .....</b>	<b>14</b>
4.1.1 Awidth and Aincr .....	15
4.1.2 Device Flags .....	15
4.1.3 Context .....	15
4.1.4 Examine and Deposit Routines .....	16
4.1.5 Reset Routine .....	16
4.1.6 Boot Routine .....	16
4.1.7 Attach and Detach Routines .....	16
4.1.8 Memory Size Change Routine .....	17
4.1.9 Debug Controls .....	17
<b>4.2 sim_unit Structure .....</b>	<b>18</b>
4.2.1 Unit Flags .....	19
4.2.2 Service Routine .....	19
<b>4.3 sim_reg Structure .....</b>	<b>19</b>
4.3.1 Register Flags .....	21
<b>4.4 sim_mtab Structure .....</b>	<b>21</b>
4.4.1 Validation Routine .....	25
4.4.2 Display Routine .....	25
<b>4.5 Other Data Structures .....</b>	<b>26</b>
<b>5. VM Provided Routines .....</b>	<b>26</b>
<b>5.1 Instruction Execution .....</b>	<b>26</b>
<b>5.2 Binary Load and Dump .....</b>	<b>26</b>
<b>5.3 Symbolic Examination and Deposit .....</b>	<b>27</b>
<b>5.4 Optional Interfaces .....</b>	<b>28</b>
5.4.1 Once Only Initialization Routine .....	28
5.4.2 Address Input and Display .....	28
5.4.3 Command Input and Post-Processing .....	28
5.4.4 Command Parsing .....	29

5.4.5	Unit Names .....	29
5.4.6	VM-Specific Commands .....	29
<b>6.</b>	<b><i>Other SCP Facilities.....</i></b>	<b>30</b>
<b>6.1</b>	<b>Terminal Input/Output Formatting Library .....</b>	<b>30</b>
<b>6.2</b>	<b>Terminal Multiplexer Emulation Library .....</b>	<b>31</b>
<b>6.3</b>	<b>Magnetic Tape Emulation Library .....</b>	<b>35</b>
<b>6.4</b>	<b>Breakpoint Support.....</b>	<b>39</b>
<b>6.5</b>	<b>Multiple Device Instance Support.....</b>	<b>40</b>
6.5.1	SCP Actions.....	41
6.5.2	VM Adjustment Routine Actions .....	43

## 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:

SIMH data type	interpretation in typical 32-bit C
int8, uint8	signed char, unsigned char
int16, uint16	signed short, unsigned short
int32, uint32	signed int, unsigned int
t_int64, t_uint64	long long, _int64 (system specific)
t_addr	simulated address, uint32 or t_uint64
t_value	simulated value, uint32 or t_uint64
t_svalue	simulated signed value, int32 or t_int64
t_mtrec	mag tape record length, uint32
t_stat	status code, int
t_bool	true/false value, int

[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:

<b>DEVICE</b>	device definition structure
<b>UNIT</b>	unit definition structure
<b>REG</b>	register definition structure

<b>MTAB</b>	modifier definition structure
<b>CTAB</b>	command definition structure
<b>DEBTAB</b>	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:

VM device	Real machine hardware
CPU	central processor and main memory
PTR	paper tape reader controller and paper tape reader
TTI	console keyboard
TTO	console output
DKP	disk pack controller and drives

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:

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

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

Interface	Function
void ( <b>*sim_vm_init</b> ) (void)	pointer to once-only initialization routine for VM
t_addr ( <b>*sim_vm_parse_addr</b> ) (...)	pointer to address parsing routine
void ( <b>*sim_vm_fprint_addr</b> ) (...)	pointer to address output routine
char ( <b>*sim_vm_read</b> ) (...)	pointer to command input routine
void ( <b>*sim_vm_post</b> ) (...)	pointer to command post-processing routine
CTAB <b>*sim_vm_cmd</b>	pointer to simulator-specific command 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:

Source module	header file	module
scp.c	scp.h	control package
sim_card.c	sim_card.h	card reader/punch library
sim_console.c	sim_console.h	terminal I/O library
sim_ether.c	sim_ether.h	Ethernet I/O library
sim_fio.c	sim_fio.h	file I/O library
sim_shmem.c	sim_shmem.h	shared memory library
sim_sock.c	sim_sock.h	socket I/O library
sim_tape.c	sim_tape.h	magtape simulation library
sim_timer.c	sim_timer.h	timer library
sim_tmxr.c	sim_tmxr.h	terminal multiplexer simulation library

### 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 used consistently throughout the VM. All existing VM's count time in instructions.

The CPU is responsible for counting down the event counter **sim\_interval** and calling the asynchronous event controller **sim\_process\_event**. SCP does the record keeping for timing.

#### 3.1.2 Step Function

SCP implements a stepping function using the step command. 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 STEP command's counts by calling routine **sim\_cancel\_step**:

- `t_stat sim_cancel_step (void)` – cancel STEP count down.

The VM can then inspect variable **sim\_step** to see if a STEP command is in progress. If **sim\_step** is non-zero, it represents the number of steps to execute. The VM can count down **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:

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

### 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:

```
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:

```
if (int_requests > INT_ENABLE) { ...process interrupt... }
```

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

```
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:

```
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:

```
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 internally 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:

PTR (300 char/sec)	3.3 msec
PTP (50 char/sec)	20 msec
CLK (line frequency)	16.6 msec
TTO (30 char/sec)	33 msec

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, **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:

- **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.

- `t_stat sim_activate_abs` (UNIT \*uptr, int32 wait). This routine schedules the specified unit with the specified waiting period, even if the unit is already on the queue.
- `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.
- `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.
- `double sim_gtime` (void). This routine returns the time elapsed since the last RUN or BOOT command.
- `uint32 sim_grtime` (void). This routine returns the low-order 32b of the time elapsed since the last RUN or BOOT command.
- `int32 sim_qcount` (void). This routine returns the number of entries on the clock queue.
- `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.
- `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.

- `int32 sim_rtcn_init` (int32 clock\_interval, int32 clk). This routine initializes the clock calibration mechanism for simulated clock *clk*. The argument is returned as the result.
- `int32 sim_rtcn_calb` (int32 ticksperssecond, int32 clk). This routine calibrates simulated clock *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:

```
/* 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_rtcn_calb (clk_ticks_per_second, clkno);
```

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

### 3.2.3 Idling

If a VM implements a free-running, calibrated clock of 100Hz or less, then the VM can also implement idling. Idling is a way of pausing simulation when no real work is happening, without losing clock calibration. The VM must detect when it is idle; it can then inform the host of this situation by calling **sim\_idle**:

- `t_bool sim_idle (int32 clk, t_bool one_tick)` – attempt to idle the VM until the next scheduled I/O event, using simulated clock *clk* as the time base, and decrement **sim\_interval** by an appropriate number of cycles. If a calibrated timer is not available, or the time until the next event is less than 1ms, decrement **sim\_interval** by 1 if *one\_tick* is TRUE; otherwise, leave **sim\_interval** unchanged.

**sim\_idle** returns TRUE if the VM actually idled, FALSE if it did not.

Because idling and throttling are mutually exclusive, the VM must inform SCP when idling is turned on or off:

- `t_stat sim_set_idle (UNIT *uptr, int32 val, char *cptr, void *desc)` – informs SCP that idling is enabled.
- `t_stat sim_clr_idle (UNIT *uptr, int32 val, char *cptr, void *desc)` – informs SCP that idling is disabled.
- `t_stat sim_show_idle (FILE *st, UNIT *uptr, int32 val, void *desc)` – displays whether idling is enabled or disabled, as seen by SCP.

### 3.2.4 Data I/O

For most devices, timing is half the battle (for clocks it is the entire war); the other half is I/O. Some devices are simulated on real hardware (for example, Ethernet controllers). Most 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. Other metadata markers are used for erasures and special cases.
- 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.

### 3.2.4.1 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 **hwmrk** 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.

### 3.2.4.2 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 SCPE\_STALL.

## 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.

Note that a device must always have at least one unit, even if that unit is not needed for simulation purposes. A device must always point to a valid register table, but the register table can consist of just the “end of table” entry.

## 4.1 *sim\_device* Structure

Devices are defined by the **sim\_device** structure (typedef **DEVICE**):

```

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_debtabs *debflags;       /* debug flag names */
    t_stat          (*msize)();         /* memory size change */
    char            *lname;            /* logical name */
    void            *help;              /* help routine (V4 dummy) */
    void            *attach_help;       /* attach help (V4 dummy) */
    void            *help_context;      /* help context (V4 dummy) */
    void            *description;       /* description (V4 dummy) */
    uint32          (*adjust) ();       /* instance adjustment routine */
};

```

The fields are the following:

<b>name</b>	device name, string of all capital alphanumeric characters.
<b>units</b>	pointer to array of <b>sim_unit</b> structures, or NULL if none.
<b>registers</b>	pointer to array of <b>sim_reg</b> structures, or NULL if none.
<b>modifiers</b>	pointer to array of <b>sim_mtab</b> structures, or NULL if none.
<b>numunits</b>	number of units in this device.
<b>aradix</b>	radix for input and display of device addresses, 2 to 16 inclusive.
<b>awidth</b>	width in bits of a device address, 1 to 64 inclusive.
<b>aincr</b>	increment between device addresses, normally 1; however, byte addressed devices with 16-bit words specify 2, with 32-bit words 4.
<b>dradix</b>	radix for input and display of device data, 2 to 16 inclusive.
<b>dwidth</b>	width in bits of device data, 1 to 64 inclusive.
<b>examine</b>	address of special device data read routine, or NULL if none is required.
<b>deposit</b>	address of special device data write routine, or NULL if none is required.
<b>reset</b>	address of device reset routine, or NULL if none is required.
<b>boot</b>	address of device bootstrap routine, or NULL if none is required.

<b>attach</b>	address of special device attach routine, or NULL if none is required.
<b>detach</b>	address of special device detach routine, or NULL if none is required.
<b>ctxt</b>	address of VM-specific device context table, or NULL if none is required.
<b>flags</b>	device flags.
<b>dctrl</b>	debug control flags.
<b>debflags</b>	pointer to array of sim_debtab structures, or NULL if none.
<b>msize</b>	address of memory size change routine, or NULL if none is required.
<b>lname</b>	pointer to logical name string, or NULL if none.
<b>adjust</b>	address of adjustment routine, or NULL if multiple instances are not supported.

The **help**, **attach\_help**, **help\_context**, and **description** fields are for V4 compatibility and are not used.

#### 4.1.1 Awidth and Aincr

The **awidth** field specifies the width of the VM's fundamental computer "word". For example, on the PDP-11, **awidth** is 16b, even though memory is byte-addressable. The **aincr** field specifies how many addressing units comprise the fundamental "word". For example, on the PDP-11, **aincr** is 2 (2 bytes per word).

If **aincr** is greater than 1, SCP assumes that data is naturally aligned on addresses that are multiples of **aincr**. VM's that support arbitrary byte alignment of data (like the VAX) can follow one of two strategies:

- Set **awidth** = 8 and **aincr** = 1 and support only byte access in the examine/deposit routines.
- Set **awidth** and **aincr** to the fundamental sizes and support unaligned data access in the examine/deposit routines.

In a byte-addressable VM, SAVE and RESTORE will require ( $\text{memory\_size\_bytes} / \text{aincr}$ ) iterations to save or restore memory. Thus, it is significantly more efficient to use word-wide rather than byte-wide memory; but requirements for unaligned access can add significantly to the complexity of the examine and deposit routines.

#### 4.1.2 Device Flags

The **flags** field contains indicators of current device status. SIMH defines 7 flags:

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

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.

#### 4.1.3 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.

#### 4.1.4 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:

`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.

#### 4.1.5 Reset Routine

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

`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.

#### 4.1.6 Boot Routine

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

`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.

#### 4.1.7 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:

`t_stat attach_routine (UNIT *uptr, char *file)` – Attach the specified *file* to the unit *uptr*. *Sim\_switches* contains the command switch; bit `SIM_SW_REST` indicates that attach is being called by the RESTORE command rather than the ATTACH command.

`t_stat detach_routine (UNIT *uptr)` – Detach unit *uptr*.

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

```
t_stat lpt_attach (UNIT *uptr, char *cptr) {
    t_stat r;
    if ((r = attach_unit (uptr, cptr)) != SCPE_OK) return r;
    lpt_error = 0;
    return SCPE_OK;
}
```



```

    }

t_stat lpt_detach (UNIT *uptr) {
    lpt_error = 1;
    return detach_unit (uptr);
}

```

If the VM specifies an ATTACH or DETACH routine, SCP bypasses its normal tests before calling the VM routine. Thus, a VM DETACH routine cannot be assured that the unit is actually attached and must test the unit flags if required.

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.

#### 4.1.8 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 **capac** 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:

*t\_stat change\_mem\_size (UNIT \*uptr, int32 val, char \*cptr, void \*desc)* – Change the capacity (memory size) of unit *uptr* to *val*. The *cptr* and *desc* arguments are included for compatibility with the SET command's validation routine calling sequence.

#### 4.1.9 Debug Controls

Devices can support debug printouts. Debug printouts are controlled by the SET CONSOLE {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)(struct sim_unit *up); /* 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 */
    uint32 dynflags;                 /* dynamic flags (V4 dummy) */
    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 */
    void *up7;                       /* V4 dummy */
    void *up8;                       /* V4 dummy */
};
```

The fields are the following:

<b>next</b>	pointer to next unit in active queue, NULL if none.
<b>action</b>	address of unit time-out service routine.
<b>filename</b>	pointer to name of attached file, NULL if none.
<b>fileref</b>	pointer to FILE structure of attached file, NULL if none.
<b>filebuf</b>	buffered devices only; pointer to memory buffer, NULL if none
<b>hwmark</b>	buffered devices only; highest modified address, + 1.
<b>time</b>	increment until time-out beyond previous unit in active queue.
<b>flags</b>	unit flags.
<b>dynflags</b>	for V4 compatibility; unused in this version.
<b>capac</b>	unit capacity, 0 if variable.
<b>pos</b>	sequential devices only; next device address to be read or written.
<b>buf</b>	by convention, the unit buffer, but can be used for other purposes.
<b>wait</b>	by convention, the unit wait time, but can be used for other purposes.
<b>u3</b>	user-defined.
<b>u4</b>	user-defined.
<b>u5</b>	user-defined.
<b>u6</b>	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 15 flags:

flag name	meaning if set
UNIT_ATTABLE	the unit responds to ATTACH and DETACH.
UNIT_RO	the unit is currently read only.
UNIT_FIX	the unit is fixed capacity.
UNIT_SEQ	the unit is sequential.
UNIT_ATT	the unit is currently attached to a file.
UNIT_BINK	the unit measures "K" as 1024, rather than 1000.
UNIT_BUFABLE	the unit buffers its data set in memory.
UNIT_MUSTBUF	the unit allocates its data buffer dynamically.
UNIT_BUF	the unit is currently buffering its data set in memory.
UNIT_ROABLE	the unit can be ATTACHED read only.
UNIT_DISABLE	the unit responds to ENABLE and DISABLE.
UNIT_DIS	the unit is currently disabled.
UNIT_RAW	the unit is attached in RAW mode.
UNIT_TEXT	the unit is in text mode (V4).
UNIT_IDLE	the unit is capable of idling (V4).

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**):

```
struct sim_reg {
    char          *name;          /* name */
    void          *loc;          /* location */
    uint32        radix;         /* radix */
    uint32        width;         /* width */
    uint32        offset;        /* starting bit */
    uint32        depth;         /* save depth */
    size_t        size;          /* access size */
    size_t        stride;        /* array stride */
}
```

```

uint32      flags;                /* flags */
t_value     maxval;              /* maximum value */
uint32      qptr;                /* current queue pointer */
};

```

The fields are the following:

<b>name</b>	register name, string of all capital alphanumeric characters.
<b>loc</b>	pointer to location of the register value.
<b>radix</b>	radix for input and display of data, 2 to 16 inclusive.
<b>width</b>	display width in bits of data, 1 to 32 (64) inclusive.
<b>offset</b>	bit offset (from right end of data).
<b>depth</b>	size of data array (normally 1).
<b>size</b>	access width in multiples of uint8 (1, 2, 4, or 8).
<b>stride</b>	array/structure stride (normally 0).
<b>flags</b>	flags and formatting information.
<b>maxval</b>	maximum value; if zero, determined from <b>width</b> .
<b>qptr</b>	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.

A register that is 32b or less keeps its data in an 8b, 16, or 32b scalar variable (signed or unsigned). A register that is 33b or more keeps its data in a 64b scalar variable (signed or unsigned).

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

```

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:

```

FLDATA      (name, location, bit_position)

```

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

```

GRDATA      (name, location, radix, width, offset)

```

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

```

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

```

Macro **CRDATA** defines an arrayed register whose data is kept in a two-dimensional C array. It is invoked by:

```

CRDATA      (name, location, radix, width, depth)

```

Macro **SRDATA** defines a register (or array) whose data is a field in a structure. It is invoked by:

```

SRDATA      (name, location, field, radix, width, offset, depth)

```

Finally, macro **XRDATA** defines an arrayed register that requires a non-standard value in the size field. It is invoked by:

**XRDATA** (name, location, radix, width, offset, depth, size, stride)

Variant **SAVEDATA** automatically sets the **XRDATA** register to be hidden and read only, that is, strictly accessible for **SAVE** and **RESTORE**.

For all of these macros, the **flag** field and the **maxval** can be filled in manually, e.g.,

```
REG lpt_reg = {
    { DRDATA      (POS, lpt_unit.pos, 31), PV_LEFT, 9999 }, ... }
```

Finally, macro **URDATA** defines an arrayed register whose data is part of the **UNIT** structure. It is a specific version of the more general **SRDATA** macro. The macro is invoked by:

**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; For example, the following declares an arrayed register of all the **UNIT** position fields in a device with 4 units:

```
{ 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.

flag name	meaning if specified
PV_RZRO	print register right justified with leading zeroes.
PV_RSPC	print register right justified with leading spaces.
PV_LEFT	print register left justified.
REG_RO	register is read only.
REG_HIDDEN	register is hidden (will not appear in EXAMINE STATE).
REG_HRO	register is read only and hidden.
REG_NZ	new register values must be non-zero.
REG_CIRC	register is a circular queue.
REG_VMIO	register is displayed and parsed using VM data routines.
REG_VMAD	register is displayed and parsed using VM address routines.

Two older flags, **REG\_FIT** and **REG\_UNIT**, are still defined but are no longer used.

### 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:

```
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 */
    void        *help;          /* V4 dummy */
}
```

```
uint32      expander;          /* expanded MTAB flags */
};
```

MTAB supports **three** different structure interpretations: regular, extended, and **expanded**. A regular MTAB entry modifies flags in the UNIT flags word; the descriptor entry is not used. The fields are the following:

<b>mask</b>	bit mask for testing the unit <b>flags</b> field
<b>match</b>	value to be stored (SET) or compared (SHOW)
<b>pstring</b>	pointer to character string printed on a match (SHOW), or NULL
<b>mstring</b>	pointer to character string to be matched (SET), or NULL
<b>valid</b>	address of validation routine (SET), or NULL
<b>disp</b>	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:

<b>mask</b>	entry flags
	MTAB_XTD      extended entry
	MTAB_VDV      valid for devices
	MTAB_VUN      valid for units
	MTAB_NMO      valid only in named SHOW
	MTAB_NC       do not convert option value to upper case
	MTAB_SHP      SHOW parameter takes optional value
<b>match</b>	value to be stored (SET)
<b>pstring</b>	pointer to character string printed on a match (SHOW), or NULL
<b>mstring</b>	pointer to character string to be matched (SET), or NULL
<b>valid</b>	address of validation routine (SET), or NULL
<b>disp</b>	address of display routine (SHOW), or NULL
<b>desc</b>	a validation-specific structure, or NULL

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. 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. 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.,

```
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.

An expanded MTAB entry modifies a bit, a set of bits, or a numeric value contained within a contiguous set of bits. The modifying value may be located in the DEVICE or UNIT **flags** field, in a specified field within a DEVICE or UNIT structure, or within a *uint32* global variable (for a device SET/SHOW) or a *uint32* element in a global array (for a unit SET/SHOW) corresponding to the unit number. The fields have the following interpretation:

<b>mask</b>	a bit mask of the bits comprising the value field
<b>match</b>	the value to be stored or compared, or the maximum allowed value
<b>pstring</b>	a pointer to the character string that is printed on a match (SHOW), or NULL
<b>mstring</b>	a pointer to the character string to be matched (SET or SHOW)
<b>valid</b>	the address of a validation routine (SET), or NULL
<b>disp</b>	the address of a display routine (SHOW), or NULL
<b>desc</b>	a pointer to the target location for the value, or NULL
<b>help</b>	a dummy pointer for V4 compatibility; always NULL
<b>expander</b>	the expanded entry flags field, containing:
	MTAB_EDV    valid for devices
	MTAB_EUN    valid for units
	MTAB_EGL    value is in a global variable instead of a DEVICE/UNIT field
	MTAB_EMK    supply the mask field to the validation routine
	MTAB_EBI    value range is biased to zero
	MTAB_EBT    value is converted to a bit positioned from right to left
	MTAB_ELR    bit is positioned from left to right
	MTAB_ERO    target is read-only; valid only for SHOW
	MTAB_ENO    valid only if option is named in the SHOW command
	MTAB_ERX(n) value is set/shown in radix <b>n</b>
	MTAB_EMN(n) minimum allowed value is <b>n</b>
	MTAB_EUR(n) user reserved field is <b>n</b>

An expanded MTAB entry performs one of two operations on the value field, depending on whether a radix is defined with the MTAB\_ERX macro. If the radix is not defined, then the operation is identical to that of a regular MTAB entry, except that the location of the value is not restricted to the UNIT **flags** field. If the radix is defined, then the value to be set is obtained by parsing the <option>=<value> portion of the SET command string in the specified radix, and the value to be shown is displayed in the specified radix.

If the radix is defined, the **match** field is set to the maximum allowed numeric value, which restricts the largest value accepted by the SET command. The minimum allowed value may be optionally set by including an MTAB\_EMN specification; if it is absent, the minimum value is zero. A specified minimum may also be used to bias the stored value by including MTAB\_EBI. This flag causes the minimum to be subtracted from (for SET) or added to (for SHOW) the value actually stored in the target location. This allows, for example, the restricted value range 9-12 to be stored in two bits instead of the four that otherwise would be required.

If MTAB\_EBT is specified, then the numeric value specified in the SET command is converted into a single bit whose position is given by the value and stored in the target location. Bit numbering ascends from right to left; adding the MTAB\_ELR flag changes the bit numbering to ascend from left to right. MTAB\_EBI can be supplied optionally to bias the resulting value before conversion. SHOW displays the numeric position of the first set bit scanning in the direction implied by the presence or absence of MTAB\_ELR.

If MTAB\_ERO is specified, then the target is read-only and only a SHOW command is accepted. An attempted SET command will be rejected with SCPE\_NXPARG. If MTAB\_ENO is specified, then the value is displayed only if the SHOW command explicitly names the **mstring** option.

Normally, the **value** parameter of a validation routine is set to the **match** field. When clearing a bit in a set of flags, the **match** field will be zero. This makes it impossible for the validation routine to determine which bit is being cleared. Specifying the MTAB\_EMK option will supply the **mask** field to the validation routine instead of the **match** field, providing an indication of the flag being cleared.

The user reserved field is four bits wide and may be set to a value that is interpreted by the VM. An example use is to direct the adjustment of the **desc** field when multiple instances of a device are created.

The **expander** field must contain either the MTAB\_EDV flag or the MTAB\_EUN flag, depending on whether a device value or a unit value is to be set or shown. The location of the value is determined by the **desc** field and the MTAB\_EGL flag. If **desc** is NULL, then the location is either the **DEVICE flags** field or the **UNIT flags** field. If **desc** is not NULL, then it points either to a field within the DEVICE or UNIT structure if MTAB\_EGL is omitted, or to a global *uint32* variable if MTAB\_EGL is specified. In the latter case, the location is a scalar if MTAB\_EDV is specified, or an array element that corresponds to the unit number if MTAB\_EUN is specified. If both MTAB\_EDV and MTAB\_EUN are specified in the **expander** field, or if MTAB\_EGL is specified and **desc** is NULL, then the SET or SHOW returns SCPE\_IERR.

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

1. Test to see if the **mstring** entry exists.
2. Test to see that the SET option matches the **mstring**.
3. Test to see if SET is allowed for the entry (not read-only).
4. Test to see if the entry is valid for the type of SET being done (device or unit).
5. Determine the target location for the value.
6. If a radix is defined, parse the value specified after the option name and verify that it is within the range allowed; convert to a positioned bit if requested. If a radix is not defined, then the **match** field (or the **mask** field if MTAB\_EMK is specified) supplies the value.
7. Call the validation routine, if any.
8. If a radix is defined and the entry specifies biasing, subtract the specified minimum from the numeric value or right-shift the positioned bit. Then left-shift the value to align with the **mask** field.
9. Apply the **mask** value to the target location and then merge the value into the location.

For SHOW, an expanded MTAB entry is interpreted as follows:

1. Test to see if the **pstring** and **mstring** entries exist.



2. If an option is specified, test to see that it matches the **mstring** and that a radix is defined, as *named commands* are allowed only for numeric values. If an option is not specified, and the entry requires a named command, return without displaying anything.
3. Test to see if the entry is valid for the type of SHOW being done (device or unit).
4. Determine the source location of the value.
5. Apply the **mask** value to the source value.
6. If a radix is defined, right-shift the value to right-align it and remove any applied bias. If a radix is not defined, test to see that the masked value equals the **match** value.
7. Call the display routine, if any.
8. If a radix is defined, convert from a positioned bit back to a bit number if specified and then print the **pstring** and the value in the specified radix. If a radix is not defined, print the **pstring** by itself.

#### 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:

`t_stat validation_routine` (UNIT \*uptr, int32 value, char \*cptr, void \*desc) – For a regular MTAB entry, test that *uptr*→**flags** can be set to *value*. *cptr* is NULL, and *desc* is set to the value of the **desc** field of the entry.

For an extended MTAB entry, test that the target location can be set to *value*. *uptr* points to the specified unit or to the first unit of the specified device. *cptr* points to the value portion of the parameter string (any characters after the = sign); if *cptr* is NULL, no value was given. *desc* is set to the value of the **desc** field of the entry.

For an expanded MTAB entry, test that the target location can be set to *value*, which will be either the value from the **match** field of the entry, or the **mask** field of the entry if MTAB\_EMK is specified, or the numeric value or positioned bit number parsed from the command line before any specified biasing is applied. If the parse failed, *value* will be -1, and *cptr* will point at the invalid numeric character string. *uptr* will point at the specified UNIT structure or at unit 0 if a device was specified, and *desc* will point at the DEVICE that hosts the unit.

The routine returns SCPE\_OK if validation succeeded or an SCPE error code if validation failed. Failure prevents the value from being set. An expanded MTAB validation routine may also elect to set the value itself, for example if a failed numeric parse occurred with a legitimate alphabetic value. In this case, the routine returns -1; this is treated as success, except that the value will not be set after returning.

#### 4.4.2 Display Routine

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

`t_stat display_routine` (FILE \*st, UNIT \*uptr, int value, void \*desc) – output device- or unit-specific state to stream *st*. For a regular MTAB entry, *value* is the value to be displayed from *uptr*→**flags**, and *desc* is the value of the **desc** field of the entry.

For an extended MTAB entry, *value* is the value to be displayed, and *uptr* points to the specified unit or to the first unit of the specified device. If the entry flags include MTAB\_SHP, *desc* points to the optional value string or is NULL if no value was supplied. If MTAB\_SHP is omitted, *desc* is set to the value of the **desc** field of the entry.

For an expanded MTAB entry, *value* is the value that will be displayed after converting a positioned bit back to a bit number and any specified biasing is removed. *uptr* will point at the specified UNIT structure or at unit 0 if a device was specified, and *desc* will point at the DEVICE that hosts the unit.

When the display routine is called for a regular or extended MTAB entry, SHOW hasn't output anything; the display routine is responsible for all output. SHOW will append a newline after the display routine returns, except for extended MTAB entries with the MTAB\_NMO flag set. The return value is ignored.

For an expanded MTAB entry, the display routine returns SCPE\_OK if the value should be displayed, or an SCPE error code if the display should be inhibited. The error code is ignored except for a named SHOW command, which will display the error code instead. The display routine may also elect to display the value itself. In this case, the routine returns -1; this is treated as success, except that the value will not be displayed after returning.

## 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.

char \***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.

char \***sim\_vm\_release** points to a character string representing a release identifier. It is initialized to NULL. If it is set by the VM, the SHOW VERSION command will print "Release" and the release identifier after the simulator version number.

## 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 addressing units consumed as a **negative** number. If the processing was successful and consumed a single addressing unit, then these routines should return SCPE\_OK. For example, PDP-11 **parse\_sym** would respond as follows to various inputs:

input	return value
XYZGH	SCPE_ARG
MOV R0,R1	-1
MOV #4,R5	-3
MOV 1234,5670	-5

There is an implicit relationship between the **addr** and **val** arguments and the device's **aincr** fields. Each entry in **val** is assumed to represent **aincr** addressing units, starting at **addr**:

val[0]	addr + 0
val[1]	addr + aincr
val[2]	addr + (2 * aincr)
val[3]	addr + (3 * aincr)
:	:

Because **val** is typically filled in and stored by calls on the device's examine and deposit routines, respectively, the examine and deposit routines and **fprint\_sym** and **fparsesym** must agree on the expected width of items in **val**, and on the alignment of **addr**. Further, if **fparsesym** wants to modify a storage unit narrower than **awidth**, it must insert the new data into the appropriate entry in **val** without destroying surrounding fields.

The interpretation of switch values is arbitrary, but the following are used by existing VM's:

switch	interpretation
-a	single character
-c	character string
-m	instruction mnemonic

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.

## 5.4 Optional 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

If the `USE_VM_INIT` symbol is defined during compilation, SCP defines a pointer (**`*sim_vm_init`**)(void). A VM requiring special initialization must fill in this pointer with the address of its special initialization routine:

```
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:

`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*.

SCP defines a pointer `t_bool` (**`*sim_vm_fprint_stopped`**)(FILE \*, t\_stat). This is initialized to NULL. If it is filled in by the VM, SCP will call the specified routine when simulated execution is stopped to print any additional information related to the stop. The calling sequence for the routine is:

`t_bool sim_vm_fprint_stopped (FILE *stream, t_stat reason)` – print additional information for the *reason* for the simulation stop to the output *stream*.

When the routine is entered, SCP will have printed a status message corresponding to the reason for the stop. The routine may then print any additional information desired, depending on the reason. If the routine returns TRUE, SCP will then print the program counter address and the symbolic machine instruction at that address. If the routine returns FALSE, these additional items will be omitted.

### 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:

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:

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 Command Parsing

SCP defines a pointer void \*(**sub\_args**) (char \*, char \*, int32, char \*\*). This is initialized to point at the internal argument substitution routine. If it is changed by the VM, SCP will call the specified routine to substitute arguments in command lines entered at the console or read from a DO command file. The calling sequence for the routine is:

void **sub\_args** (char \*iptr, char \*optr, int32 size, char \*args []) – substitute the arguments contained in the *args* array for the parameter tokens in the input buffer indicated by *iptr* using a temporary buffer indicated by *optr* whose size is *size*.

Parameter tokens are of the form %*n*, where *n* is a number from 0-9. Each token is replaced by the corresponding argument in the *args* array.

SCP defines a pointer int32 (\***sim\_get\_radix**) (const char \*, int32, int32). This is initialized to point at the internal radix parsing routine. If it is changed by the VM, SCP will call the specified routine to parse a radix keyword or command-line switch. The calling sequence for the routine is:

int32 **sim\_get\_radix** (const char \*cptr, int32 switches, int32 default\_radix) – return the radix designated by the keyword pointed to by *cptr* or the command-line switches specified by *switches*; if no radix designator is present, return the *default\_radix*.

On entry, if the *cptr* parameter is not NULL, then it points at the keyword for a SET <device> <radix> command. Otherwise, the *switches* parameter is examined for command-line switches that designate the radix. By providing its own parsing routine, the VM may extend the standard set of radix identifiers.

#### 5.4.5 Unit Names

SCP defines a pointer char \* (\***sim\_vm\_unit\_name**) (const UNIT \*). It is initialized to NULL. If it is filled in by the VM, SCP will call the specified routine to obtain unit names for the SHOW QUEUE command. The calling sequence for the routine is:

char \* **sim\_vm\_unit\_name** (const UNIT \*uptr) – return a pointer to a string designating the name of the unit indicated by *uptr*, or return NULL to indicate that the name should be formed from the corresponding device name and unit number.

The VM may use this routine to provide more descriptive names for specialized, typically hidden units, such as device controllers or multiplexer scanners, that appear in the simulator event queue display.

#### 5.4.6 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**):

```
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:

`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 Input/Output Formatting Library

SIMH provides routines to convert ASCII input characters to the format expected VM, and to convert VM-supplied ASCII characters to C-standard format. The routines are

`int32 sim_tt_inpcvt (int32 c, uint32 mode)` – convert input character *c* according to the *mode* specification and return the converted result (-1 if the character is not valid in the specified mode).

`int32 sim_tt_outcvt (int32 c, uint32 mode)` – convert output character *c* according to the *mode* specification and return the converted result (-1 if the character is not valid in the specified mode).

The supported modes are:

TTUF_MODE_8B	8b mode; no conversion
TTUF_MODE_7B	7b mode; the high-order bit is masked off
TTUF_MODE_7P	7b printable mode; the high-order bit is masked off In addition, on output, if the character is not printable, -1 is returned
TTUF_MODE_UC	7b upper case mode; the high-order bit is masked off In addition, lower case is converted to upper case If the character is not printable, -1 is returned

On input, TTUF\_MODE\_UC has an additional modifier, TTUF\_MODE\_KSR, which forces the high order bit to be set rather than cleared.

The set of printable control characters is contained in the global bit-vector variable **sim\_tt\_pchar**. Each bit represents the character corresponding to the bit number (e.g., bit 0 represents NUL, bit 1 represents SOH, etc.). If a bit is set, the corresponding control character is considered printable. It initially contains the following characters: BEL, BS, HT, LF, and CR. The set may be manipulated with these routines:

t\_stat **sim\_set\_pchar** (int32 flag, char \*cptr) – set **sim\_tt\_pchar** to the value pointed to by *cptr*, return SCPE\_2FARG if *cptr* is null or points to a null string, or SCPE\_ARG if the value cannot be converted or does not contain at least CR and LF.

t\_stat **sim\_show\_pchar** (FILE \*st, DEVICE \*dptr, UNIT \*uptr, int32 flag, char \*cptr) – output the **sim\_tt\_pchar** value to the stream *st*.

Note that the DEL character is always considered non-printable and will be suppressed in the UC and 7P modes.

## 6.2 Terminal Multiplexer 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 multiplexers; 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**):

```
struct tmln {
    SOCKET      conn;           /* line conn */
    char         *ipad;         /* IP address */
    uint32       cnms;          /* connect time ms */
    int32        tsta;          /* Telnet state */
    int32        rcve;          /* rcv enable */
    int32        xmte;          /* 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 */
    void         *exptr;        /* extension pointer */
};
```

The fields are the following:

<b>conn</b>	connection socket (0 = disconnected)
<b>ipad</b>	pointer to a string containing the IP address
<b>cnms</b>	connect time in milliseconds
<b>tsta</b>	Telnet state
<b>rcve</b>	receive enable flag (0 = disabled)
<b>xmte</b>	transmit flow control flag (0 = transmit disabled)
<b>dstb</b>	Telnet bin mode disabled
<b>rxbpr</b>	receive buffer remove pointer
<b>rxbpi</b>	receive buffer insert pointer

<b>rxcnt</b>	receive count
<b>txbpr</b>	transmit buffer remove pointer
<b>txbpi</b>	transmit buffer insert pointer
<b>txcnt</b>	transmit count
<b>txlog</b>	pointer to log file descriptor
<b>txlogname</b>	pointer to log file name
<b>rxb</b>	receive buffer
<b>rbr</b>	receive buffer break flags
<b>txb</b>	transmit buffer
<b>exptr</b>	pointer to a VM-defined structure containing extended state, or NULL if not used

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

```
struct tmxr {
    int32      lines;                /* # lines */
    int32      port;                /* listening port */
    SOCKET     master;              /* master socket */
    TMLN       *ldsc;               /* pointer to line descriptors */
    int32      *lnorder;            /* line connection order */
    DEVICE     *dptr;               /* multiplexer device */
    void       *exptr               /* pointer to VM-defined structure */
};
```

The fields are the following:

<b>lines</b>	number of lines (constant)
<b>port</b>	master listening port (specified by ATTACH command)
<b>master</b>	master listening socket (filled in by ATTACH command)
<b>ldsc</b>	array of line descriptors
<b>lnorder</b>	array of line numbers in order of connection sequence, or NULL if user-defined connection order is not required
<b>dptr</b>	pointer to the multiplexer's DEVICE structure, or NULL if the device is to be derived from the UNIT passed to the first attach call
<b>exptr</b>	pointer to a VM-defined structure containing extended state, or NULL if not used.

The number of elements in the **ldsc** and **lnorder** arrays must equal the value of the **lines** field. Set **lnorder** to NULL if the connection order feature is not needed. If the first element of the **lnorder** array is -1, then the default ascending sequential connection order is used. Set **dptr** to NULL if the device should be derived from the unit passed to the **tmxr\_attach** call.

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 the line descriptor state to the idle condition 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) | 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\_msg** (SOCKET sock, const char \*msg) – write the message specified by *msg* to the open socket specified by *sock*.

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  $\geq 0$ , 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  $\geq 0$ , preface the output with the specified line number.

t\_stat **tmxr\_set\_log** (UNIT \*uptr, int32 val, char \*cptr, void \*mp) – for the terminal multiplexer described by the TMXR pointer *mp*, open the log file named by *cptr* for the multiplexer line associated with *uptr* or, if NULL, for the line number specified by *val*. The calling sequence allows the routine to be used as an MTAB processing routine.

t\_stat **tmxr\_set\_nolog** (UNIT \*uptr, int32 val, char \*cptr, void \*mp) – for the terminal multiplexer described by the TMXR pointer *mp*, close the log file for the multiplexer line associated with *uptr* or, if NULL, for the line number specified by *val*. The calling sequence allows the routine to be used as an MTAB processing routine.

t\_stat **tmxr\_show\_log** (FILE \*st, UNIT \*uptr, int32 val, void \*mp) – for the terminal multiplexer described by the TMXR pointer *mp*, print the logging status for the multiplexer line associated

with *uptr* or, if NULL, for the line number specified by *val* on the output stream *st*. The calling sequence allows the routine to be used as an MTAB processing routine.

void **tmxr\_post\_logs** (t\_bool close\_logs) – flush all active terminal multiplexer line log files; if *close\_logs* is TRUE, then also close all log files.

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 multiplexer described by *mp*. The calling sequence allows **tmxr\_dscln** to be used as an MTAB processing routine.

t\_stat **tmxr\_set\_Inorder** (UNIT \*uptr, int32 val, char \*cptr, void \*mp) – set the line connection order array for the terminal multiplexer described by the TMXR pointer *mp*. The string pointed to by *cptr* is parsed for a semicolon-delimited list of ranges. Ranges are of the form:

line	a single numbered line
line1-line2	ascending sequence from <b>line1</b> to <b>line2</b>
line1/length	ascending sequence from <b>line1</b> to <b>line1+length-1</b>
ALL	ascending sequence of all lines defined by the multiplexer not otherwise specified

Sequential connections will be made to multiplexer lines in the order specified. Lines defined by the multiplexer but not specified in the list will not be eligible for connection. The keyword ALL may be appended to a connection list to add the remaining lines to the list in ascending order. Absent a call to this routine, the default connection order is equivalent to a list containing just ALL, i.e., all defined lines in ascending order.

The line order array must provide an int32 element for each line. The calling sequence allows **tmxr\_set\_Inorder** to be used as an MTAB processing routine.

t\_stat **tmxr\_show\_Inorder** (FILE \*st, UNIT \*uptr, int32 val, void \*mp) – for the terminal multiplexer described by the TMXR pointer *mp*, output the line connection order to stream *st*. The order is rendered as a semicolon-delimited list of ranges. The calling sequence allows **tmxr\_show\_Inorder** to be used as an MTAB processing routine.

t\_stat **tmxr\_show\_summ** (FILE \*st, UNIT \*uptr, int32 val, void \*desc) – outputs the summary status of the multiplexer (TMXR \*) *desc* to stream *st*.

t\_stat **tmxr\_show\_cstat** (FILE \*st, UNIT \*uptr, int32 val, void \*desc) – outputs either the connections (*val* = 1) or the statistics (*val* = 0) of the multiplexer (TMXR \*) *desc* to stream *st*. Also checks for multiplexer not attached, or all lines disconnected.

t\_stat **tmxr\_show\_lines** (FILE \*st, UNIT \*uptr, int32 val, void \*desc) – outputs the number of lines in the terminal multiplexer (TMXR \*) *I* to stream *I*.

TMLN \***tmxr\_find\_Idsc** (UNIT \*uptr, int32 val, TMXR \*mp) – for the terminal multiplexer described by *mp*, return a pointer to the multiplexer line descriptor associated with *uptr* or, if NULL, for the line number specified by *val*. If neither parameter describes a valid multiplexer line, the routine returns NULL.

int32 **tmxr\_send\_buffered\_data** (TMLN \*lp) – send the buffered data for the multiplexer line described by *lp* and return the number of bytes remaining in the buffer after transmission.

void **tmxr\_init\_line** (TMLN \*lp) – reset the state of the multiplexer line described by *lp* to that of an idle line with no buffered data pending.

void **tmxr\_report\_connection** (TMXR \*mp, TMLN \*lp, int32 line) – send a notification message reporting the connection of line number *line* to the multiplexer line specified by line descriptor pointer *lp* that is associated with the terminal multiplexer described by *mp*.

void **tmxr\_disconnect\_line** (TMLN \*lp) – send a disconnection notification message to the multiplexer line specified by line descriptor pointer *lp* and then close the connection and reset the descriptor state.

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

The library also provides the following optional interfaces. These are implemented as global pointers to internal routines that may be overridden by a VM to provide its own replacements.

int32 (**\*tmxr\_read**) (TMLN \*lp, int32 length) – read up to *length* characters into the character buffer associated with line descriptor pointer *lp* and return the actual number of characters read.

int32 (**\*tmxr\_write**) (TMLN \*lp, int32 length) – write up to *length* characters from the character buffer associated with line descriptor pointer *lp* and return the actual number of characters written.

void (**\*tmxr\_show**) (TMLN \*lp, FILE \*stream) – print the connection status of the line associated with line descriptor pointer *lp* to the output *stream*.

void (**\*tmxr\_close**) (TMLN \*lp) – close the multiplexer line connection associated with line descriptor pointer *lp*.

t\_bool (**\*tmxr\_is\_extended**) (TMLN \*lp) – return TRUE if the line described by the line descriptor pointer *lp* has been extended by the VM and FALSE if it has not.

A VM may use these routines and the **exptr** field of the TMLN structure to extend multiplexer services to communications protocols other than Telnet.

### 6.3 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:

MTUF_WLK	unit is write locked
MT_F_STD	unit is using SIMH Standard format
MT_F_EXT	unit is using SIMH Extended format
MT_F_E11	unit is using E11 format
MT_F_TPC	unit is using TPC format
MT_F_P7B	unit is using P7B format

The unit format flags specify the initial tape format used by the unit. A simulator may allow the user to change the format if desired. If none of the format flags are specified, SIMH standard format is used.

These tape format codes are defined:

MTUF_F_STD	SIMH Standard format
MTUF_F_EXT	SIMH Extended format
MTUF_F_E11	E11 format
MTUF_F_TPC	TPC format
MTUF_F_P7B	P7B format

If magtape simulators need to define private unit flags, those flags must 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*. This routine will parse and set the tape format if the `-F` switch is included on the command line. Simulators must call this routine, rather than the standard `attach_unit` routine, to ensure proper tape unit initialization.

**t\_stat sim\_tape\_detach** (UNIT \*uptr) – Detach tape unit *uptr* from its current file. Simulators must call this routine, rather than the standard `detach_unit` routine, to ensure proper tape unit cleanup.

**t\_stat sim\_tape\_set\_fmt** (UNIT \*uptr, int32 val, char \*cptr, void \*desc) – Set the tape format for unit *uptr* to the format code specified by *val* or to the format name specified by string *cptr*. If *desc* is NULL, then the string pointed to by *cptr* will be parsed for a name corresponding to the desired tape format. If the initial format was SIMH Extended when the first call to this routine was made, then a subsequent call specifying the name "SIMH" will change back to SIMH Extended format. If the initial format was not SIMH Extended, then a subsequent call specifying "SIMH" will return to SIMH Standard format. This ensures that a simulator will not receive extended-format records if it is not prepared to handle them. If *desc* is not NULL, then it must point at a uint32 value that will receive the current format code, *val* must contain one of the MTUF\_F\_\* format codes above that specifies the new tape format, and *cptr* is ignored.

**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\_set\_capac** (UNIT \*uptr, int32 val, char \*cptr, void \*desc) – Set the tape capacity for unit *uptr* to the capacity, in MB, specified by string *cptr*.

**t\_stat sim\_tape\_show\_capac** (FILE \*st, UNIT \*uptr, int32 val, void \*desc) – Write the capacity for unit *uptr* to the file specified by descriptor *st*.

**t\_stat sim\_tape\_set\_dens** (UNIT \*uptr, int32 val, char \*cptr, void \*desc) – Set the tape density in bits per inch for unit *uptr* to the value specified by the density code *val* or the density represented by the character string *cptr*. If *desc* is NULL, then *val* must be set to one of the MT\_DENS\_\* constants in `sim_tape.h` other than MT\_DENS\_NONE; the supplied value is used as the tape density, and the character string is ignored. Otherwise, *desc* must point at an int32 value containing a set of allowed densities constructed as a bitwise OR of the appropriate MT\_\*\_VALID constants defined in `sim_tape.h`. In this case, the string pointed to by *cptr* will be parsed for a decimal value corresponding to the desired density in bits per inch and validated against the set of allowed values.

**t\_stat sim\_tape\_show\_dens** (FILE \*st, UNIT \*uptr, int32 val, void \*desc) – Write the tape density for unit *uptr* to the file specified by descriptor *st*.

**t\_stat sim\_tape\_rdrecf** (UNIT \*uptr, uint8 \*buf, t\_mtrInt \*tbc, t\_mtrInt 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\_rdreocr** (UNIT \*uptr, uint8 \*buf, t\_mtrInt \*tbc, t\_mtrInt 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\_wrrpcf** (UNIT \*uptr, uint8 buf, t\_mtrInt tbc) – Write buffer *buf* of size *tbc* as the next record on unit *uptr*.

**t\_stat sim\_tape\_sprecf** (UNIT \*uptr, t\_mtrInt \*tbc) – Space unit *uptr* forward one record. The size of the record is returned in *tbc*.

**t\_stat sim\_tape\_spreocr** (UNIT \*uptr, t\_mtrInt \*tbc) – Space unit *uptr* reverse one record. The size of the record is returned in *tbc*.

**t\_stat sim\_tape\_wrmrk** (UNIT \*uptr, t\_mtrInt mk) – Write private marker *mk* on unit *uptr*. If the unit is not set for SIMH Extended format, MTSE\_FMT status is returned. If *mk* is not in the private marker class, MTSE\_RESERVED status is returned.

**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\_wrgap** (UNIT \*uptr, uint32 gaplen) – Write an erase gap on unit *uptr* of *gaplen* tenths of an inch in length. The tape density must have been set by a previous **sim\_tape\_set\_dens** call; if it has not, then no action is taken, and MTSE\_IOERR is returned.

**t\_stat sim\_tape\_erase** (UNIT \*uptr, t\_mtrInt tbc) – Write an erase gap on unit *uptr* of *tbc* bytes in length.

**t\_stat sim\_tape\_errecf** (UNIT \*uptr, t\_mtrInt tbc) – Write an erase gap in the forward direction on unit *uptr* of a length corresponding to a record containing the number of bytes specified by *tbc*. This function is used to erase a record in place. After erasure, the tape will be positioned at the end of the gap. If a length of 0 is specified, then a tape mark at the current tape position will be erased. If the tape is not positioned at a tape mark, the routine fails with MTSE\_INVRL, and the tape position is unchanged.

**t\_stat sim\_tape\_errecr** (UNIT \*uptr, t\_mtrInt tbc) – Write an erase gap in the reverse direction on unit *uptr* of a length corresponding to a record containing the number of bytes specified by *tbc*. This function is used to erase a record in place. After erasure, the tape will be positioned at the start of the gap. If a length of 0 is specified, then a tape mark preceding the current tape position will be erased. If the tape is not positioned after a tape mark, the routine fails with MTSE\_INVRL, and the tape position is unchanged.

**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) – Return TRUE if unit *uptr* has exceed the capacity specified of the specified unit (kept in uptr->capac).

Erase gaps are currently supported only in SIMH tape image format files. If tape erase functions are called for a unit that is not set to a supported format, the calls will succeed, but no action will be taken. This allows a device simulator that supports writing erase gaps to call the routines without first verifying the tape format currently selected by the user.

If the tape format is set to SIMH Extended, then the **sim\_tape\_wrrrecf** routine allows private data records to be written. The *tbc* parameter specifies a data record class in the upper four bits and a data record length in the lower 28 bits. If the specified class is a reserved class or the private marker class, MTSE\_RESERVED is returned. If a Class 0 (good data) record is specified with a zero length, MTSE\_INVRL is returned.

For SIMH Standard and E11 formats, the class must be either Class 0 to indicate a good data record or Class 8 to indicate a bad data record. If an unsupported class is specified, MTSE\_FMT is returned. If a Class 0 record is specified with a zero length, then the routine returns MTSE\_OK with nothing done.

If the tape format is set to SIMH Extended, then before calling the **sim\_tape\_rdrecf**, **sim\_tape\_rdrecr**, **sim\_tape\_sprecf**, or **sim\_tape\_sprecr** routine, the variable addressed by the *tbc* parameter must be set to a bitmap of the object classes to return. Each of the classes is represented by its corresponding bit, i.e., bit 0 represents Class 0, bit 1 for Class 1, etc. The routine will return only objects from the selected classes. Unselected class objects will be ignored by skipping over them until the first selected class object is seen. This allows a simulator to declare those classes it understands (e.g., standard classes 0 and 8, plus private classes 2 and 7) and those classes it wishes to ignore.

Markers and data records in the SIMH-reserved classes are read and interpreted by the tape library, regardless of whether or not the corresponding class bits are set. The bits only affect whether the objects are returned to the caller. Setting the bitmap to zero (no classes selected) will cause the routine to return only when it encounters a tape mark, end-of-medium, or the physical end of file—an action equivalent to that of a "space file forward" or "space file reverse" routine.

On return, the variable addressed by the *tbc* parameter contains either the marker class in the upper four bits and the marker-specific value in the lower 28 bits, or a data record class in the upper four bits and a data record length in the lower 28 bits. The MTR\_CL macro may be used to extract the class, and the MTR\_RL macro may be used to extract the record length.

If a "bad data" record is selected and successfully read, MTSE\_RECE will be returned in addition to the class and length in the *tbc* parameter variable. Successful reads of the "good data" record class and all private data and marker classes return MTSE\_OK.

If SIMH-reserved classes are selected and read, MTSE\_RESERVED is returned if the object is not recognized by the tape library; otherwise, MTSE\_OK is returned for data records, or the MTSE value appropriate for the standard marker is returned. In all cases, the *tbc* parameter variable will contain the class and length word. If a routine returns MTSE\_EOM, the *tbc* parameter variable will contain MTR\_EOM if an EOM marker was read, or some other value if the physical EOF was encountered while reading.

Reads of marker class objects do not use the *buf* and *max* parameters.

**Sim\_tape\_attach**, **sim\_tape\_detach**, **sim\_tape\_set\_fmt**, **sim\_tape\_show\_fmt**, **sim\_tape\_set\_capac**, **sim\_tape\_show\_capac**, **sim\_tape\_set\_dens**, and **sim\_tape\_show\_dens** return standard SCP status codes; the other magtape library routines return private codes for success and failure. The currently defined magtape status codes are:

MTSE_OK	operation successful
---------	----------------------

MTSE_UNATT	unit is not attached to a file
MTSE_FMT	unit specifies an unsupported tape file format, or an operation is not supported with the currently selected format
MTSE_IOERR	host operating system I/O error during operation
MTSE_INVRL	invalid record length (the record is larger than the supplied buffer, or the record is incomplete)
MTSE_RECE	successful read of a bad data record
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
MTSE_RUNAWAY	a read did not encounter a data record or tape mark within 25 feet
MTSE_RESERVED	a reserved class was read or specified (SIMH Extended format only)

If the tape density has been set by a previous **sim\_tape\_set\_dens** call, then the length is monitored when skipping over erase gaps during reads. If the length reaches 25 feet, motion is terminated, and MTSE\_RUNAWAY status is returned. Runaway status is also returned if an end-of-medium marker or the physical end of file is encountered while spacing over a gap; however, MTSE\_EOM is returned if the tape is positioned at the EOM or EOF when the read is initiated. If the density has not been set, then a gap of any length is skipped, and MTSE\_RUNAWAY status is never returned.

**sim\_tape\_set\_fmt**, **sim\_tape\_show\_fmt**, **sim\_tape\_set\_capac**, and **sim\_tape\_show\_capac** should be referenced by an entry in the tape device's modifier list, as follows:

```
MTAB tape_mod[] = {
    { MTAB_XTD|MTAB_VDV, 0, "FORMAT", "FORMAT",
      &sim_tape_set_fmt, &sim_tape_show_fmt, NULL },
    { MTAB_XTD|MTAB_VUN, 0, "CAPACITY", "CAPACITY",
      &sim_tape_set_capac, &sim_tape_show_capac, NULL }, ...
};
```

If tape density support is desired, an additional modifier entry should be included, as follows:

```
{ MTAB_XTD|MTAB_VUN, 0, "DENSITY", "DENSITY",
  &sim_tape_set_dens, &sim_tape_show_dens, NULL },
```

## 6.4 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

`uint32l sim_brk_test (t_addr addr, int32 typ)` – test to see if a breakpoint of type *typ* is set for location *addr*; returns 0 if no, and a bit mask of all breakpoints that match *typ* if yes

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> }
```

To accommodate more complex breakpoint schemes, SCP implements a concept of breakpoint spaces. Each breakpoint space is an orthogonal collection of breakpoints that are tracked independently. For example, in a symmetric multiprocessing simulation, breakpoint spaces could be assigned to each CPU to distinguish E (execution) breakpoints for different processors. SCP supports up to 64 breakpoint spaces; the space is specified by bits <31:26> of the *typ* argument to **sim\_brk\_test**. By default, there is only one breakpoint space (space 0).

## 6.5 Multiple Device Instance Support

SCP supports the creation and deletion of multiple instances of VM devices. A VM can designate specific devices as *clonable* and allow the user to create a limited number of independent copies of the device. The number of current instances of a device can be adjusted upward or downward as desired by the user, within constraints imposed by the device simulator.

In support of multiple instances, SCP offers the following services:

- Processing of a global command, `SET device COUNT=number`, that allows the user to specify the number of instances of a clonable device.
- Allocation of the new **DEVICE** structures, each a copy of the device specified by the user, and insertion of pointers to the structures in the **sim\_devices** array.
- Automatic generation of names for the new devices, with a choice of naming algorithms.
- Allocation and copying of the unit and register structure arrays.
- Invocation of a VM-supplied, device-specific routine that is responsible for adjusting the copies to ensure that each references its own local state.

For a device to support multiple instances, the VM must:

- Declare the **sim\_devices** array with additional, initially **NULL** elements that will be used to hold pointers to new device instances.
- Declare an adjustment function and place a pointer to the function in the **adjust** field of the **DEVICE** structure.

Each device instance must be able to reference its own copy of the variables that comprise the device state. Typically, the device adjustment routine will allocate and initialize a state structure for each device instance. Each of the device's functions that accesses device state must either be passed a pointer to the state structure or must be able to derive the pointer from other function parameters, such as channel addresses or device or unit pointers.



### 6.5.1 SCP Actions

When the user enters a SET COUNT command to change the number of instances of a specified device, SCP performs the following actions:

1. If the **adjust** field of the DEVICE structure is NULL, or the **flags** field includes the DEV\_CLONE flag, then SCP reports "Command not allowed" and returns to the command prompt.
2. If the instance count is missing from the command string, then SCP reports "Missing value" and returns to the command prompt.
3. The device's adjustment function is called with a NULL pointer parameter. The function returns a formatted 32-bit value (described below) that indicates the size of the **sim\_devices** array, the number of device instances allowed, and an optional identifier used to determine the naming algorithm.
4. If parsing of the instance count fails, or the supplied value is larger than the instance limit, SCP reports "Invalid argument" and returns to the command prompt.
5. The current **sim\_devices** array is scanned to determine the current number of instances of the specified device and the location in the array where new clones will be added or existing clones will be removed.
6. If the user-specified instance count is equal to the current instance count, no further action is required, and SCP returns to the command prompt.
7. If the new instance count is greater than the current count, then the difference indicates the number of new clones to add.
  - 7.1. If insufficient space remains in the **sim\_devices** array to accommodate all of the pointers to the new devices, SCP reports "Address space exceeded" and returns to the command prompt.
  - 7.2. The set of currently occupied **sim\_devices** elements is expanded by inserting the required number of NULL entries immediately after the current set of clones of the specified device, if there are any, or after the original device.
  - 7.3. For each new instance to be added, the following actions are performed.
    - A DEVICE structure is allocated and initialized from the original device, and a pointer to the structure is added to the **sim\_devices** array. Each new DEVICE has its **ctxt** and **lname** fields set to NULL, and the DEV\_CLONE flag is added to the **flags** field.
    - A new character array that is ten characters longer than the original device name is allocated and initialized from the name, and the pointer is placed in the DEVICE **name** field. The name is then modified using one of two algorithms.

If the naming identifier returned by the first call to the device's adjustment routine is zero, then the last character of the name is incremented by the clone number (e.g., "LPR" becomes "LPS" for the first clone, "LPT" for the second, etc.).

If the naming identifier is non-zero, then it is appended to the name after incrementing by the clone number (e.g., "LPR" with identifier "A" becomes "LPRB" for the first clone, "LPRC" for the second, etc., and "TTY" with identifier "0" becomes "TTY1" for the first clone, etc.). In addition, a logical name for the original device is created with the naming identifier appended to keep the naming of all instances uniform (e.g., the logical name "LPRA" is created for

original device "LPR"). The VM adjustment routine is free to alter the name if neither algorithm is suitable.

- New UNIT and REG structure arrays are allocated and initialized from the corresponding arrays in the original device, and pointers are placed in the DEVICE **units** and **registers** fields. Each unit is sanitized by clearing the UNIT\_RO, UNIT\_ATT, and UNIT\_BUF flags from the **flags** field and the UNIT\_PIPE flag from the **dynflags** field. Also, the **next**, **filename**, **fileref**, and **filebuf** fields are set to NULL, and the **hwmrk**, **time**, and **pos** fields are set to zero.

If any of the memory allocations fail, all of the newly allocated items are freed, the **sim\_devices** array is returned to its pre-expansion state, and SCP reports "Memory exhausted" and returns to the command prompt.

- 7.4. The device's adjustment routine is called and passed a pointer to the first new entry in **sim\_devices** and the count of new instances. The routine performs whatever device-specific adjustments to the new instances are required, such as allocating new device context structures, and returns a *t\_stat* status code to indicate success or failure of the adjustment. If the routine returns a status code other than SCPE\_OK, all of the items newly allocated by SCP are freed, the **sim\_devices** array is returned to its pre-expansion state, and SCP reports the error message associated with the status code and returns to the command prompt.
- 7.5. If the original device has a reset routine, it is called with the P (power-on) switch set for each new device instance to perform whatever power-on initialization may be required. If the routine returns a status code other than SCPE\_OK, SCP reports the error message associated with the status code and returns to the command prompt. Note that in this case, the new device instances are not removed, as it may be possible for the user to correct the error and then issue a RESET -P command to the new devices.
- 7.6. Once all of the new devices are reset, SCP returns to the command prompt
8. If the new instance count is less than the current count, then the difference indicates the number of new clones to delete. A new instance count of one will remove all existing clones, leaving only the original device.
  - 8.1. The device's adjustment routine is called and passed a pointer to the first entry in **sim\_devices** to remove and the count of removed instances. The routine performs whatever device-specific adjustments are needed before the affected devices are removed, such as freeing previously allocated device context structures. If the device removal can proceed, the routine returns SCPE\_OK. However, if the routine determines that deallocation cannot occur, e.g., because a cloned device is in the middle of a critical operation, it returns an SCPE error code. In this case, SCP will report the error and not deallocate the devices.
  - 8.2. For each instance to be removed, the following actions are performed.
    - All queued units are removed from the event queue, and all attached units are detached.
    - The memory allocations referenced by pointers in the DEVICE **name**, **units**, **registers**, and **lname** fields are freed.
    - The allocated DEVICE referenced by the pointer in **sim\_devices** is freed.
  - 8.3. The set of currently occupied **sim\_devices** elements is reduced by removing the required number of clone entries.

- 8.4. If all clones have been removed, and a logical name for the original device was created to keep the naming of all instance names uniform, then the logical name is removed to return the device to its original name. This will not occur if the user had subsequently changed the logical name from its original assignment.
- 8.5. SCP returns to the command prompt, reporting any error status returned from the device adjustment call.

## 6.5.2 VM Adjustment Routine Actions

Each device that supports multiple instances must supply an adjustment routine and store a pointer to the routine in the DEVICE **adjust** field. The routine's calling sequence is:

`uint32 adjust_routine (DEVICE **dvptr, int32 delta)` – If *dvptr* is NULL, return a formatted 32-bit value described below. Otherwise, *dvptr* points to the first of *delta* elements within the **sim\_devices** array that are being added (if *delta* is positive) or will be deleted (if *delta* is negative). Adjust the device(s) as needed and return a *t\_stat* status code indicating the success or failure of the adjustment.

When the adjustment routine is called with a NULL pointer parameter, SCP expects the 32-bit return value to be formatted as follows:

31	30	--	25	24	23	22	--	17	16	15	14	--	2	1	0
naming identifier					instance limit					sim_devices size					

The fields have the following meanings:

<i>sim_devices size</i>	The total number of elements defined in the <b>sim_devices</b> array, including the NULL element at the end. The difference between this number and the number of initialized elements gives the number of new device instances that can be added by the user. A VM whose <b>sim_devices</b> array has all elements initialized (except the NULL at the end) does not support device cloning.
instance limit	The maximum number of instances allowed for the specified device. The count includes the original instance. Some devices may be constrained by an external factor, such as the number of available I/O slots, assigned channel numbers, or device addresses. Setting this limit ensures that the user cannot request more device instances than can be accommodated. Temporarily setting the limit to zero will prevent the user from changing the current number of instances.
naming identifier	A character used to produce unique names for new device instances. If the ID is zero, new instance names will be formed by copying the original device name and incrementing the last character in the name by the device instance number. If the ID is non-zero, it will be appended to the original device name and incremented by the device instance number. In the latter case, the character should be alphanumeric, although no check for this is made.

When the adjustment routine is called with a non-NULL pointer parameter, it must perform whatever actions are necessary to prepare the indicated device instances for use or destruction. Preparation for use involves, at a minimum, ensuring that all of the newly generated device names (including the logical name for the original device, if applicable) are unique. SCP does not check this before calling the

adjustment routine because it does not know how the VM would want to resolve duplicates, i.e., by reporting the duplicate and rejecting the change, or by assigning a different name to the conflicting device.

Additional preparation for new instances by the adjustment routine may involve:

- Replacing the auto-generated device names with names produced by a different algorithm. SCP provides name string allocations that are ten characters larger than the original name for this purpose. If this is not sufficient, the VM may free the SCP allocation and substitute its own.
- Undoing or changing the logical name assignment for the original device that is done when the clone ID is non-zero.
- Allocating new MTAB structure arrays. SCP defaults all device instances to use the original modifier table. This is appropriate if the **desc** fields of the MTAB elements are all NULL or point at global variables. Any fields that reference local state variables must be relocated to point at the per-device state.
- Relocating REG **loc** fields to point at per-device state variables.
- Allocating new device context structures.
- Allocating new state variables structures.
- Changing the device or unit flags or the device-specific unit fields of the new devices. The new instances will contain copies of these items as they are at the time the command is entered. So, e.g., if a device has been disabled and is then cloned, each of the clones will be disabled as well. It may be preferable to start all new instances in some defined state rather than in the current state of the original device.

When the *delta* parameter is negative, the adjustment routine must free any memory allocations that were performed when the new instances were added. The routine may instead return an error status code to prevent SCP from deallocating the cloned device structures. This may be needed if the adjustment routine determines that an affected device is in the middle of a critical operation.

Typical actions performed by the adjustment routine in response to a deallocation call are:

- Freeing the device context structures.
- Freeing the state variables structures.
- Freeing the modifier structure arrays if they were allocated when the devices were added.
- Cleaning up after units in the event queue (SCP will cancel all scheduled events, but the VM may need to perform some special action, such as a cache flush, depending on the event service that would have been performed).
- Detaching any files that need special handling (if a device has a detach routine, SCP will call it, but a normal call to that routine might defer the detach until some checkpoint is reached, such as the end of a printed line).

The adjustment routine should be structured to execute the deallocation portion in response to an allocation failure when new devices are being created. This is to ensure that all partial allocations made before the allocation failure have been freed before the routine returns to SCP.