Writing a Simulator for the SIMH System

Revised 26-Dec-2019 for SIMH V3.10-0

Bob Eager

Copyright Notice

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

Contents

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 may be useful. 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, int32 or t_uint64
t_svalue	simulated signed value, int32 or t_uint64
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 Visual C++, 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:

Name	**Data element**
DEVICE	device definition structure
UNIT	unit definition structure
REG	register definition structure
MTAB	modifier definition structure
CTAB	command definition structure
DEBTAB	debug table entry structure

3. VM Organisation

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

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

Interface	Function
void (***sim_vm_init**)(void)	pointer to once-only initialisation routine for
t_addr(***sim_vm_parse_addr**)(…)	pointer to address parsing routine
void (***sim_vm_fprint_addr**)(…)	pointer to address output routine
char (***sim_vm_read**)(…)	pointer to command input routine
char (***sim_vm_post**)(…)	pointer to command post-processing routine
t_bool (***sim_vm_fprint_stopped**)(…)	pointer to stop message format routine
t_bool (***sim_vm_is_subroutine_call**) (…)	pointer to routine that determines if the current instruction is a subroutine call
CTAB ***sim_vm_cmd**	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 PDPs; pdp8 for the PDP-8; pdp11 for the PDP-11; nova for Nova; hp2100 for the HP 21XX; h316 for the Honeywell 316/516; 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 magnetic 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_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	magnetic tape 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 CPUs 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 VMs 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**:

Interface	**Function**
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	uint8
IBM 1401	7-bit	uin8
PDP-8	12-bit	uint16
PDP-11, Nova	16-bit	uint16
PDP-1	18-bit	uint32
VAX	32-bit	uint32
PDP-10, IBM 7094	36-bit	t_uint64

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 minimise 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-generalise; 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 generalisation.

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 level of 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 level of 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 analyse for the device address.
- Locate the selected device.
- Break down the I/O command into standard fields.
- Call the device processor.

Analysing 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 analysed, 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 the 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 recognised 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 CPUs 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. **DEVICE** structures correspond to devices, **REG** structures to registers, and **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 cancelling 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:

Device	Wait time
PTR (300 char/sec)	3.3 msec
PTP (50 char/sec)	20 msec
CLK (line frequency)	16.6/20 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 VMs 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 and variables 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 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 specified waiting period overrides the currently pending waiting period.
- t_stat **sim_activate_after** (UNIT *uptr, int32 μsec_delay). This routine places the specified unit on the active queue with the specified delay based on the simulator's calibrated clock. The specified delay must be greater than 0 μsecs. If the unit is already active, the active queue is not changed, and no error occurs.
- 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.
- t_bool **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.
- int32 **sim_activate_time** (UNIT *uptr). This routine returns the time the device has remaining in the queue + 1. If it is not pending, 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 32 bits 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 represents the time until the first unit on the event queue that is scheduled to happen. **sim_inst** counts down this value (usually by 1 for each instruction executed). 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 synchronise multiple simulated clocks (to a maximum of 8) to wall time.

- int32 **sim_rtcn_init** (int32 clock_interval, int32 clk). This routine initialises the clock calibration mechanism for simulated clock *clk*. The argument is returned as the result.
- int32 **sim_rtcn_calb** (int32 tickspersecond, 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 prologue 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_rtcb_calb (clk_ticks_per_second, clkno));
```

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

The underlying timer services will automatically run a calibrated clock whenever the simulator doesn't have one registered and running, or when the registered timer is running too fast for accurate clock calibration. This will allow the **sim_activate_after** API to provide proper wall clock relative timing delays.

Some simulated systems use programmatic interval timers to implement clock ticks. If a simulated system or simulated operating system uses a constant interval to provide the system clock ticks, then the clock device is a candidate to be a calibrated timer. If the simulated operating system dynamically changes the programmatic interval more than once, then such a device is not a calibrated timer, but it certainly should use **sim_activate_after** and **sim_activate_time** to implement the programmatic interval delays.

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). This routine attempts to idle the VM until the next scheduled I/O event, using simulated clock *clk* as the time base, and decrements **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, it decrements **sim_interval** by 1 if *one_tick* is TRUE; otherwise, it leaves **sim_interval** unchanged.

sim_idle returns TRUE if the VM actually idled, FALSE if it did not.

In order for idling to be well behaved on the host system, simulated devices which poll for input (console and terminal multiplexors are examples), the polling that these devices perform should be done at the same time as when the simulator will unavoidably be executing instructions. The most common time this happens is when clock tick interrupts are generated. As such, these devices should schedule their polling activities to be aligned with the clock ticks which are happening anyway or some multiple of the clock tick value.

- t_stat **sim_clock_coschedule** (UNIT *uptr, int32 interval). This routine places the specified unit on the active queue behind the default timer at the specified *interval* rounded up to a whole number of timer ticks. An *interval* value 0 is legal; negative intervals cause an error. If the unit is already active, the active queue is not changed, and no error occurs.

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). This routine informs SCP that idling is enabled.
- t_stat **sim_clr_idle** (UNIT *uptr, int32 val, char *cptr, void *desc). This routine informs SCP that idling is disabled.
- t_stat **sim_show_idle** (FILE *st, UNIT *uptr, int32 val, void *desc). This routine 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 an 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.
- Magnetic tapes, 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 (tape) marks are recorded as one record length of 0.
- Cards have 12 bits of data per column, but the data is most conveniently viewed as (ASCII) characters. Column binary can be implemented using two successive characters per card column.

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

3.2.4.1 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 the standard C routines *fread* and *fwrite*, respectively, but will correct for endian dependencies. For buffered devices, I/O is done by copying data to or from the allocated buffer. The device code must maintain the number (+1) of the highest address modified in the **hwmark** field of the UNIT structure. For both the non-buffered and buffered cases, the device must perform ail address calculations and positioning operations.

SIMH provides capabilities to access files >2GB in size (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). This is identical to standard C *fseek*, with two exceptions: 'where = SEEK_END' is not supported, and the position argument can be 64 bits 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_kbd** (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.

3.2.4.3 Simulators for computers without a console port

If a computer being simulated doesn't have a console port, SCP will call **sim_poll_kbd** periodically to detect when a user types ^E (Control E) in the session running the simulator, and they will be returned to the **sim>** prompt.

3.2.4.4 Miscellaneous routines

SCP provides the following additional routines.

- const char ***sim_error_text** (t_stat stat). This returns the text message associated with the status value *stat* (which may originate from SCP or the VM).
- void **sim_printf** (const char * fmt, …). This routine works in a similar way to the standard C *printf*; it outputs the formatted string to standard output, and to the simulator log (if enabled).

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 that does not need registers need not provide a register table; instead the **registers** field is set to NULL.

Device registers serve two purposes:

1. They provide a means of letting the simulator user (more often, the developer) have visibility to examine and potentially change arbitrary state variables within the simulator from the **sim>** prompt rather than having to use a debugger.
2. They provide all of the information in the internal state of a simulated device, so that a SAVE command can capture that state, and a subsequent RESTORE (after exiting and restarting the same simulator) will be able to proceed without any information being missing.

A device unit serves two fundamental purposes in a simulator:

1. It acts as an entity which can generate events which are handled in the simulated instruction stream (via one of the **sim_activate** APIs).
2. It provides a place which holds an open file pointer for simulated devices which have content bound to file contents (via ATTACH commands).

For example: A UNIT can be mapped to real units in a simulated device (e.g. disk drives), or it might serve merely to perform timing related activities, or both of these might be present. The PDP-11 RQ simulation has a combination of both of these; there are four units which map one to one directly to simulated disk drives, and there are two additional units. One is used to time various things, and one is used to provide instruction delays while walking through the MSCP initialization and command processing sequence.

4.1 DEVICE structure

Devices are defined by the **DEVICE** structure:

```
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;            /* address increment */
  uint32             dradix;           /* data radix */
  uint32             dwidth;           /* data width */
```

```
    t_stat            (*examine)();      /* examine routine */
    t_stat            (*deposit)();      /* deposit routine */
    t_stat            (*reset)();        /* reset routine */
    t_stat            (*boot)();         /* boot routine */
    t_stat            (*attach)();       /* attach routine */
    t_stat            (*detach)();       /* detach routine */
    void              *ctxt;             /* Context */
    uint32            flags;             /* flags */
    uint32            dctrl;             /* debug control flags */
    struct sim_debtab debflags;          /* debug flag names */
    t_stat            (*msize)();        /* memory size change */
    char              *lname;            /* logical name */
    void              *help;             /* (4.0 dummy) help routine */
    void              *attach_help;      /* (4.0 dummy) help attach rtn */
    void              *help_context;     /* (4.0 dummy) help context*/
    void              *description;      /* (4.0 dummy) help desc. */
};
```

The fields are the following:

Field	Meaning
name	device name, string of all capital alphanumeric characters
units	pointer to array of **sim_unit** structures, or NULL if none
registers	pointer to array of **sim_reg** structures, or NULL if none
modifiers	pointer to array of **sim_mtab** structures, or NULL if none
numunits	number of units in this device
aradix	radix for input and display of device addresses, 2 to 16 inclusive
awidth	width in bits of a device address, 1 to 64 inclusive
aincr	increment between device addresses, normally 1; however, byte addressed devices with 16-bit words specify 2, with 32-bit words 4
dradix	radix for input and display of device data, 2 to 16 inclusive
dwidth	width in bits of device data, 1 to 64 inclusive
examine	address of special device data read routine, or NULL if none is required
deposit	address of special device data write routine, or NULL if none is required
reset	address of device reset routine, or NULL if none is required
boot	address of device bootstrap routine, or NULL if none is required
attach	address of special device attach routine, or NULL if none is required
detach	address of special device detach routine, or NULL if none is required
ctxt	address of VM-specific device context table, or NULL if none is required
flags	device flags
dctrl	debug control flags
debflags	pointer to array of **sim_debtab** structures, or NULL if none
msize	address of memory size change routine, or NULL if none is required
lname	pointer to logical name string
help	(4.0 dummy) help routine
attach_help	(4.0 dummy) help attach routine
help_context	(4.0 dummy) help context
help_description	(4.0 dummy) help description

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 16 bits, 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**. VMs 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 (memory_size_bytes / **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 the following flags:

Flag name	Meaning if set
DEV_DIS	device is currently disabled
DEV_DISABLE	device can be set enabled or disabled
DEV_DYNM	device requires call on **msize** 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). This should 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). This should 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.The –p switch (available via the global variable **sim_switches**) specifies a reset to power-up state.

The reset routine is a reasonable place to perform one time initialization activities specific to the device, by keeping a static variable indicating that the one time initialization has been performed.

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). This should 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). This should attach the specified *file* to the unit *uptr.* **sim_switches** contains the command switch; bit SIM_SW_REST indicates that *attach_routine* is being called by the RESTORE command rather than the ATTACH command.
- t_stat *detach_routine* (UNIT *uptr). This should 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), instantiation of the maximum memory size could 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). This should 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 {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 the table **debflags**. This 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:

Field	Meaning
name	name of the debug flag
mask	bit mask of the debug flag

The array is terminated with a NULL entry.

The use and definition of debug mask values is up to the particular simulator device. Some simulator support libraries define their own debug mask values that can be used to display various details about the internal activities of the respective library. Library-defined debug masks are defined starting at the high bit of the 32 bit the mask word, so device specific masks should start their mask definitions with the low bits to avoid unexpected debug output if the definitions collide.

Simulator code can produce debug output by calling **sim_debug**, which is declared in header file scp.h:

```
void sim_debug (uint32 dbits, DEVICE *dptr, const char *fmt, ...);
```

The *dbits* is a flag which matches a mask in a **sim_debtab** structure, and *dptr* is the DEVICE which has the corresponding **dctrl** field.

4.2 UNIT structure

Units are allocated as a contiguous array. Each unit is defined with a **UNIT** structure:

```
struct sim_unit {
      struct sim_unit   *next;              /* next active */
      t_stat      (*action)();              /* action routine */
      char        *filename;                /* open file name */
      FILE        *fileref;                 /* file reference */
      void        *filebuf;                 /* memory buffer */
      uint32      hwmark;                   /* high water mark */
      int32       time;                     /* time out */
      uint32      flags;                    /* flags */
      uint32      dynflags;                 /* dynamic flags */
      t_addr      capac;                    /* capacity */
      t_addr      pos;                      /* file position */
      int32       buf;                      /* buffer*/
      int32       wait;                     /* wait */
      int32       u3;                       /* device specific */
      int32       u4;                       /* device specific */
      int32       u5;                       /* device specific */
      int32       u6;                       /* device specific */
      void        *up7;                     /* (4.0 dummy) */
      void        *up8;                     /* (4.0 dummy) */
};
```

The fields are the following:

Field	Meaning
next	pointer to next unit in active queue, NULL if none

action	address of unit time-out service routine
filename	pointer to name of attached file, NULL if none
fileref	pointer to FILE structure of attached file, NULL if none
hwmark	buffered devices only; highest modified address + 1
time	increment until time-out beyond previous unit in active queue
flags	unit flags
capac	unit capacity, 0 if variable
pos	sequential devices only; next device address to be read or written
buf	by convention the unit buffer, but can be used for other purposes
wait	by convention the wait time, but can be used for other purposes
u3	user-defined
u4	user-defined
u5	user-defined
u6	user-defined
up7	(4.0 dummy)
up8	(4.0 dummy)

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 };
```

This 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 these 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 attached in text mode

Units for sequential devices (UNIT_SEQ) must update the unit structure **pos** member to reflect the position in the attached sequential device file as data is read or written to that file. The **pos** value is used to position the attached file whenever simulation execution starts or resumes from the `sim>` prompt.

Starting at bit position UNIT_V_UF up to but not including UNIT_V_RSV, 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. If the user has typed the interrupt character (^E), it returns SCPE_STOP.

4.3 REG structure

Registers are allocated as a contiguous array, with a NULL register at the end. Each register is defined with a **REG** structure:

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

The fields are the following:

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

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

A register that is 32 bits or less keeps its data in a 32 bit scalar variable (signed or unsigned). A register that is 33 bits or more keeps its data in a 64 bit scalar variable (signed or unsigned). There are several exceptions to this rule:

- An arrayed register keeps its data in a C-array whose SIMH data type is as large as (or if necessary, larger than), the width of a register element. For example, an array of 6 bit registers would keep its data in a uint8 (or int8) array; an array of 16 bit registers would keep its data in a uint16 (or int16) array; an array of 24 bit registers would keep its data in a uint32 (or int32) array.
- A register flagged with REG_FIT obeys the sizing rules of an arrayed register, rather than a normal scalar register. This is useful for aliasing registers into memory or into structures.

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, bit_position)

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)

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

REG lpt_reg = { { DRDATA (POS, lpt_unit .pos, 31), PV_LFT }, ... }

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

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 filed. The flags can be any of the normal register flags; REG_UNIT will be OR'd in automatically. For example, the following declares an arrayed register of all the **UNIT** position fields in a device with 4 units:

```
{ 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 set**
PV_RZRO	print register right justified with leading zeros
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_UNIT	register resides in the **UNIT** structure
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
REG_FIT	register container uses arrayed rather than scalar size rules

The PV flags are mutually exclusive. PV_RZRO is the default if no formatting flag is specified.

Starting at bit position REG_V_UF, the remaining flags are user-defined. These flags are passed to the VM-defined **fprint_sym** and **parse_sym** routines in the upper bits of the *addr* parameter; they are merged with the lower 16 bits containing the register radix value.

If a user-defined flag or the REG_VMIO flag is specified in a register's *flag* field, the EXAMINE and DEPOSIT commands will call **fprint_sym** and **parse_sym** instead of the standard print and parse routines. The user-defined flags passed in the *addr* parameter may be used to identify the register or determine how it is to be handled.

If REG_UNIT is clear, the register data is located at the address specified by the *loc* pointer. If REG_UNIT is set, the register name is used to refer to a field in a UNIT structure, and *loc* points to that field in the UNIT structure for unit 0. The examine and deposit commands will adjust that address by the unit number times the size of the UNIT structure to determine the actual data address.

4.4 MTAB structure

Device-specific SHOW and SET commands are processed using the modifications array, which is allocated as a 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;          /* (4.0 dummy) */
  };
```

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

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

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

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

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

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

Extended MTAB entries have a different interpretation:

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

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

1. Test to see if the **mstring** entry exists.
2. Test to see if the SET parameter matches the **mstring**.

3. Test to see if the entry is valid for the type of SET being done (SET device or SET unit).
4. If a validation routine exists, call it and return its status. The validation routine is responsible for storing the result.
5. If **desc** is NULL, exit.
6. 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.

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). This should test that *uptr*.**flags** can be set to *value*. *cptr* points to the value portion of the parameter string (any characters after the = sign); if *cptr* is NULL, no value was given. *desc* points to the REG or int32 used to store the parameter.

4.4.2 Display routine

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

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

When the display routine is called for a regular MTAB entry, SHOW hasn't output anything. SHOW will append a newline after the display routine returns, except for entries with the MTAB_NMO flag set.

4.5 Other data structures

- char **sim_name**[] is a character array containing the VM name.
- char **sim_prog_name**[] is a character array containing a system-dependent idea of the name of the program executable, and can be used in error messages, etc.
- 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, but less than SCPE_BASE, then **sim_stop_message[n]** is printed by SCP.

5. VM provided routines and hooks

5.1 Instruction execution

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

- t_stat **sim_instr** (void). This should 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 *fnarn, t_bool flag). If *flag*=0, data should be loaded from binary file *fptr.* If *flag* =1, data should be dumped 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 the specified file before calling **sim_load**, and close it on return.

sim_load may optionally load or dump data in different formats based on flag options specified in the **sim_switches** variable. If or how this is done, and what any switches mean, are completely up to the simulator's implementation in the **sim_load** function.

5.3 Symbolic examination and deposit

If the VM provides symbolic examination and deposit of data, it must provide two routines, **fprlnt_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, this routine should 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, character string *cptr* should be parsed 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 the 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** field. Each entry in *val* is assumed to represent **aincr** addressing units, starting at *addr*:

Entry	Address
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 **fparse_sym** must agree on the expected width of items in *val*, and on the alignment of *addr*. Further, if **fparse_sym** 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 number of words in the val array is given by the global variable **sim_emax**.

The interpretation of switch values is arbitrary, but the following are used by existing VMs in theor **fprint_sym** implementations:

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.

fprint_sym is called to print the instruction at the program counter value for the simulation stop message, for registers containing user-defined or REG_VMIO flags in their flag fields and memory values printed by the EXAMINE command, and for printing the values printed by the EVAL command. These cases are differentiated by the presence of special flags in the *switch* parameter. For a simulation stop, the "M" switch and the SIM_SW_STOP switch are passed. For examining registers, the SIM_SW_REG switch is passed. In addition, the user-defined flags and register radix are passed in the *addr* parameter. Register radix is taken from the radix specified in the register definition, or overridden by –d, -o, or –x switches in the command. For examining memory and the EVAL command, no special switch flags are passed.

parse_sym is called to parse memory, register, and the logical and relational search specifier values for the DEPOSIT command and the symbolic expression for the EVAL command. As with **fprint_sym**, these cases are differentiated by the presence of special flags in the *switch* parameter. For registers, the SIM_SW_REG switch is passed. For all other cases, no special switch flags are passed.

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 turned off by default. Use of them requires intimate knowledge of how SCP functions internally, and is not recommended to the novice VM writer.

5.4.1 Once-only initialisation routine

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

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

The special initialization routine can perform any actions required by the VM. If the other optional interfaces are to be used, the initialization routine can fill in the appropriate pointers; however, this can just as easily be done in the CPU reset routine (since that is called during SCP initialization, as well as when a RESET command is issued later on).

5.4.2 Address input and display

SCP defines a pointer t_addr (***sim_vm_parse_addr**)(DEVICE *, char *, char *). This is initialised 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). This should 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 initialised 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). This should output address *addr* to *stream* in the format required by the device pointed to by *dptr*.

5.4.3 Radix handling

SCP defines a pointer int32 (***sim_get_radix**) (const char *cptr, int32 switches, int32 default_radix). This is initialised to a default routine for processing radix-relevant switches. If it is filled in by the VM, SCP will use the specified routine to process switches.

The routine supplied by the VM should normally return 0 if *cptr* is not NULL, as this implies a SET command. Otherwise, it should return the radix selected by the switches, or *default_radix* if none are set.

5.4.4 Command input and post-processing

SCP defines a pointer char *(**sim_vm_read**)(char *, int32 *, FILE *). This is initialised 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_read** (char *buf, int32 *max, FILE *stream). This should 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 initialised 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 **sim_vm_post** routine is:

- void **sim_vm_post** (t_bool from_scp). If called from SCP, the argument *from_scp* is TRUE; otherwise, it is FALSE.

5.4.5 Simulator Stop Message Formatting

SCP defines a pointer, t_bool (***sim_vm_fprint_stopped**)(FILE *, t_stat). It is initialised to NULL. If it is filled in by the VM, SCP will call this routine when a simulator stop occurs. The calling sequence for the **sim_vm_fprint_stopped** routine is:

- t_bool **sim_vm_fprint_stopped** (FILE *stream, t_stat reason). This should write a simulator stop message to *stream* for the reason specified, and return TRUE if SCP should append the program counter value, or FALSE if SCP should not

When the instruction loop is exited, SCP regains control and prints a simulator stop message. By default, the message is printed with this format:

<reason>, <program counter label>: <address> (<instruction mnemonic>)

For example:

SCPE_STOP prints "Simulation stopped, P: 24713 (LOAD 1)"
SCPE_STEP prints "Step expired, P: 24713 (LOAD 1)"

For VM stops, this routine is called after the reason has been printed and before the comma, program counter label, address, and instruction mnemonic are printed. Depending on the reason for the stop, the routine may insert additional information, and it may request omission of the PC value by returning FALSE instead of TRUE. For example, a VM may define these stops and their associated formats:

STOP_SYSHALT	prints "System halt 3, P: 24713 (LOAD 1)"
STOP_HALT	prints "Programmed halt, CIR: 030365 (HALT 5), P: 24713 (LOAD 1)"
STOP_CDUMP	prints "Cold dump complete, CIR: 000020"

For these examples, the VM's **sim_vm_fprint_stopped** routine prints " 3" and returns TRUE for STOP_SYSHALT, prints ", CIR: 030365 (HALT 5)" and returns TRUE for STOP_HALT, prints ", CIR: 000020" and returns FALSE for STOP_CDUMP, and prints nothing and returns TRUE for all other VM stops.

5.4.6 VM-specific commands

SCP defines a pointer CTAB ***sim_vm_cmd**. This is initialised to NULL. If filled in by the VM, SCP interprets it as a pointer to a supplementary 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 **CTAB** structure:

```
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). This should process the input string *buf* based on the optional argument *arg.*

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

When looking for a matching command, SCP scans the command table from first to last entry, looking for a command name that begins with the command supplied by the user. The first one found is considered the matching command. If no match is found, the SCP standard command table is scanned next, using the same "first match" rule. You may need to adjust command names for VM-specific commands to avoid conflicting with commonly used standard commands. For example, if a VM defined the single VM-specific command "NORMAL_START", SCP would accept "N" as an abbreviation for this command. This might confuse users who expect "N" to be an abbreviation of the "NEXT" command. The "first match is used" rule is useful when a VM needs to redefine a standard SCP command with a different syntax. For example, the VAX simulators do this in several different ways to redefine the BOOT command.

6. Other SCP facilities

6.1 Terminal input/output formatting library

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

- int32 **sim_tt inpcvt** (int32 c, uint32 mode). This should convert input character *c* according to the *mode* specification, and return the converted result (or -1 if the character is not valid in the specified mode).
- int32 **sim_tt_outcvt** (int32 c, uint32 mode). This should convert output character *c* according to the *mode* specification, and return the converted result (or -1 if the character is not valid in the specified mode).

The supported modes are:

Mode	**Meaning**
TTUF_MODE_8B	8 bit mode; no conversion
TTUF_MODE_7B	7 bit mode; the high order bit is masked off
TTUF_MODE_7P	7 bit 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	7 bit 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). This will set **sim_tt_pchar** to the value pointed to by *cptr*. It returns 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. The string argument must be in the default radix of the current simulator.
- t_stat **sim_show_pchar** (FILE *st, DEVICE *dptr, UNIT *uptr, int32 flag, char *cptr). This will output the **sim_tt_pchar** value to the stream *st*. The **sim_tt_pchar** value will be displayed in the default radix of the current simulator, and character mnemonics for each set bit will also be displayed,

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:

```
struct tmln {
        SOCKET      conn;               /* line conn */
        uint32      ipad;               /* IP address */
        uint32      cnms;               /* connect time ms */
        int32       tsta;               /* telnet state */
        int32       rcve;               /* rcv enable */
        int32       xmte;               /* xmt enable */
        int32       dstb;               /* disable Telnet 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:

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

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

```
struct tmxr {
        int32       lines;              /* # of lines */
        int32       port;               /* listening port */
        SOCKET      master;             /* master socket */
        TMLN        *ldsc;              /* pointer to line descriptors */
        int32       *lnorder;           /* line order */
        DEVICE      *dptr;              /* multiplexer device*/
  };
```

The fields are the following:

Field	Usage
lines	number of lines (constant)
port	master listening port (specified by ATTACH command)
master	master listening socket (filled in by ATTACH command)
ldsc	array of line descriptors
lnorder	array of line numbers in order of connection sequence, or NULL if user-defined connection order is not required
dptr	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

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). This will poll for a new connection to the terminals described by *mp*. If there is a new connection, the routine resets all the line descriptor state (including receive enable) and returns the line number (index to line descriptor) for the new connection. If there isn't a new connection, the routine returns -1.
- void **tmxr_reset_ln** (TMLN *lp). This will reset the line described by *lp*. The connection is closed and all line descriptor state is reset.
- int32 **tmxr_getc_ln** (TMLN *lp). This will return the next available character from the line described by *lp*. If no character is available, the return variable is 0. If a character is available, the return variable is:

  ```
  (1 « TMXR_V_VALID) | character
  ```

  If a BREAK occurred on the line, SCPE_BREAK will be ORed into the return variable. If no character is available, the return value is 0.
- void **tmxr_poll_rx** (TMXR *mp). This will poll for input available on the terminals described by *mp*.
- void **tmxr_rqln** (TMLN *lp). This will return the number of characters in the receive queue of the line described by *lp* which are ready to be read now.
- t_stat **tmxr_putc_ln** (TMLN *lp, int32 chr).This will output the 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). This will poll for output complete on the terminals described by *mp*.
- void **tmxr_tqln (**TMLN *lp). This will return the number of characters in the transmit queue of the line described by *lp*.
- int32 **tmxr_send_buffered_data** (TMLN *lp). This will flush any buffered data for the line described by *lp*. It returns the number of bytes sent.
- t_stat **tmxr_attach** (TMXR *mp, UNIT *uptr, char *cptr). This will 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). This will 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). This will detach all connections for the terminals described by *mp* and unit *uptr*.
- t_stat **tmxr_close_master** (TMXR *mp). This will 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). This is a stub examine routine, needed because the extra terminals are marked as attached; it always returns an error.
- t_stat **tmxr_dep** (t_value val, t_addr addr, UNIT *uptr, int32 sw). This is a stub deposit routine, needed because the extra terminals are marked as detached; it always returns an error.
- void **tmxr_msg** (SOCKET sock, const char *msg). This will output the character string *msg* to socket *sock*.

- void **tmxr_linemsg (**TMLN *lp, char *msg). This will output the character string *msg* to line *lp.*
- void **tmxr_fconns** (FILE *st, TMLN *lp, int32 ln). This will output a connection status to stream *st* for the line described by *lp*. If *ln* is >= 0, the output will be prefaced by the specified line number.
- void **tmxr_fstats** (FILE *st, TMLN *lp, int32 ln). This will output connection statistics to stream *st* for the line described by *lp*. If *ln* is >= 0, the output will be prefaced by the specified line number.
- tstat **tmxr_set_log** (UNIT *uptr, int32 val, char *cptr, void *mp). This enables logging of a line of the multiplexer described by *mp* to the filename pointed to by *cptr*. If *uptr* is NULL, then *va*l indicates the line number; otherwise, the unit number within the associated device implies the line number. This function may be used as an MTAB validation routine.
- tstat **tmxr_set_nolog** (UNIT *uptr, int32 val, const char *cptr, const void *mp). This disables logging of a line of the multiplexer described by *mp* to the filename pointed to by *cptr*. If *uptr* is NULL, then *val* indicates the line number; otherwise, the unit number within the associated device implies the line number. This function may be used as an MTAB validation routine.
- tstat **tmxr_show_log** (FILE *st, UNIT *uptr, int32 val, const void *mp). This outputs the logging status of a line of the multiplexer described by *mp* to stream *st*. If *uptr* is NULL, then *val* indicates the line number; otherwise, the unit number within the associated device implies the line number. This function may be used as an MTAB display routine.
- t_stat **tmxr_dscln** (UNIT *uptr, int32 val, char *cptr, void *mp). This will parse the string pointed to by *cptr* for a decimal line number. If the line number is valid, it will 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_lnorder** (UNIT *uptr, int32 val, char *cptr, void *desc). This will set the line connection order array associated with the TMXR structure pointed to by *desc*. The string pointed to by *cptr* is parsed for a semicolon-delimited list of ranges. The line order array must provide an int32 element for each line. The calling sequence allows **tmxr_set_lnorder** to be used as an MTAB processing routine. Ranges are of the form:

Range	**Description**
line1-line2	ascending sequence from line1 to line2
line1/length	ascending sequence from line1 to line1+length-1
ALL	ascending sequence of all lines defined by the multiplexer

- t_stat **tmxr_show_lnorder** (FILE *st, UNIT *uptr, int32 val, void *desc). This will output the line connection order associated with the TMXR structure pointed to by *desc*, to stream *st*. The order is rendered as a semicolon-delimited list of ranges. The calling sequence allows **tmxr_show_lnorder** to be used as an MTAB processing routine.
- t_stat **tmxr_show_summ** (FILE *st, UNIT *uptr, int32 val, void *desc). This 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). This outputs either the connections (val = 1) or the statistics (val = 0) of the multiplexer (TMXR *) *desc* to stream *st*.. It also checks for multiplexer not attached, or all lines disconnected.
- t_stat **tmxr_show_lines** (FILE *st, UNIT *uptr, int32 val, void *desc). This outputs the number of lines in the terminal multiplexer (TMXR *) to stream *st*.

The OS-dependent socket routines should not need to be accessed by the terminal simulators. The routine **sim_sock** is for internal use by the TMXR library only and should be not be used directly by any simulator.

6.2.1 Terminal multiplexer user hooks

SCP defines four routines for socket access to a line. They are initialised to default routines for read, write, show and close functionality. They can be overridden by VM-specific versions if necessary.

- int32 **tmxr_read** (TMLN *lp, int32 length). This reads up to *length* bytes into the buffer associated with line *lp*.The actual number of bytes read is returned. If no bytes are available, 0 is returned. If an error occurred while reading, **-**1 is returned.
- int32 **tmxr_write** (TMLN *lp, int32 length). This writes up to *length* bytes from the buffer associated with line *lp*. The actual number of bytes written is returned. If an error occurred

while writing, -1 is returned.
- void **tmxr_show** (TMLN *lp, FILE *stream). This writes the description of line *lp* to the file indicated by *stream*.
- void **tmxr_close** (TMLN *lp). This closes the line indicated by *lp*.

The default routines can be found in sim_tmxr.c. They are named the same as the hook routines, but with the names prefixed by **tmxr_local** instead of just **tmxr**.

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:

Flag	Meaning
MTUF_WLK	unit is write locked

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

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

- t_stat **sim_tape_attach** (UNIT *uptr, char *cptr). This attaches tape unit *uptr* to file *cptr*. Tape simulators should call this routine, rather than the standard **attach_unit** routine, to allow for future expansion of format support.
- t_stat **sim_tape_detach** (UNIT *uptr). This detaches tape unit *uptr* from its current file.
- t_stat **sim_tape_set_fmt** (UNIT *uptr, int32 val, char *cptr, void *desc). This sets the tape format for unit *uptr* to the format specified by string *cptr*.
- t_stat **sim_tape_show_fmt** (FILE *st, UNIT *uptr, int32 val, void *desc). This writes 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). This sets 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). This writes the capacity for unit *uptr* to the file specified by descriptor *st*.
- t_stat **sim_tape_set_dens** (UNIT *uptr, int32 val, const char *cptr, void *desc). This sets the tape density for unit *uptr* to the density, in bits per inch, specified by string *cptr*. Only specific densities are supported; *desc* must point at an int32 value consisting of one or more MT_*_VALID constants logically ORed together that specifies the densities allowed. Alternately, *desc* may be set to NULL and *val* may specify one of the MT_DENS_* constants to set the density directly; in this case, *cptr* is ignored.
- t_stat **sim_tape_show_dens** (FILE *st, UNIT *uptr, int32 val, const void *desc). This writes the density for unit *uptr* to the file specified by descriptor *st*.
- t_stat **sim_tape_rdrecf** (UNIT *uptr, uint8 *buf, t_mtrlnt *tbc, t_mtrlnt max). This forward reads the next record on unit *uptr* into buffer *buf* of size *max*. It returns the actual record size in *tbc*.
- t_stat **sim_tape_rdrecr** (UNIT *uptr, uint8 *buf, t_mtrlnt *tbc, t_mtrlnt max). This reverse reads the next record on unit *uptr* into buffer *buf* of size *max*. It returns the actual record size in *tbc*. Note that the record is returned in forward order; that is, byte 0 of the record is stored in *buf*[0], and so on.
- t_stat **sim_tape_wrrecf** (UNIT *uptr, uint8 buf, t_mtrlnt tbc). This writes buffer *buf* of size *tbc* as the next record on unit *uptr*.
- t_stat **sim_tape_errecf** (UNIT *uptr, t_mtrlnt tbc). Starting at the current tape position, this writes an erase gap in the forward direction on unit *uptr* for a length corresponding to a record containing the number of bytes specified by *tbc*. If *tbc* is 0, then the tape mark at the current position is erased. If the tape is not positioned at a record of the specified length or at a tape mark, the routine returns MTSE_INVRL.

- t_stat **sim_tape_errecr** (UNIT *uptr, t_mtrlnt tbc). Starting at the current tape position, this writes an erase gap in the reverse direction on unit *uptr* for a length corresponding to a record containing the number of bytes specified by *tbc*. If *tbc* is 0, then the tape mark preceding the current position is erased. If the tape is not positioned at the end of a record of the specified length or at a tape mark, the routine returns MTSE_INVRL.
- t_stat **sim_tape sprecf** (UNIT *uptr, t_mtrlnt *tbc).This spaces unit *uptr* forward one record. The size of the record is returned in *tbc*.
- t_stat **sim_tape_sprecr** (UNIT *uptr, t_mtrlnt *tbc). This spaces unit *uptr* back (reverse) one record. The size of the record is returned in *tbc*.
- t_stat **sim_tape_wrtmk** (UNIT *uptr). This writes a tape mark on unit *uptr*.
- t_stat **sim_tape_wreom** (UNIT *uptr). This writes 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). This writes an erase gap on unit *uptr* of *gaplen* tenths of an inch in length at a tape density specified by a preceding call of routine **sim_ tape_set_dens**.
- t_stat **sim_tape_rewind** (UNIT *uptr). This rewinds unit *uptr.* This operation succeeds whether or not the unit is attached to a file.
- t_stat **sim_tape_reset** (UNIT *uptr). This resets unit *uptr.* This routine should be called when a tape unit is reset.
- t_bool **sim_tape_bot** (UNIT *uptr). This returns TRUE if unit *uptr* is at beginning-of-tape.
- t_bool **sim_tape_wrp** (UNIT *uptr). This returns TRUE if unit *uptr* is write-protected.
- t_bool **sim_tape_eot** (UNIT *uptr). This returns TRUE if unit *uptr* has exceeded the capacity indicated for the specified unit (kept in *uptr*->capac).

The library supports reading and writing erase gaps in standard (SIMH) tape format image files. Before writing a gap with **sim_tape_wrgap**, the tape unit density must be set by calling **sim_tape_set_dens**; failure to do so will result in an error. For reading, if the tape density has been set, then the length is monitored when skipping over erase gaps. If the gap length reaches 25 feet (the maximum allowed by the ANSI/ECMA standards), motion is terminated and "tape 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. If the density has not been set, then a gap of any length is skipped, and tape runaway status is never returned; in effect, any erase gaps present in the tape image file will be transparent to the calling simulator.

The library supports writing erase gaps over existing data records and writing records over existing gaps. If the end of a gap overlays part of a data record, the record will be truncated, but the tape image will remain valid.

An attempt to write an erase gap in an unsupported tape format results in no action and no error. This allows a device simulator that supports writing erase gaps to call **sim_tape_wrgap** without concern for the tape format currently selected by the user.

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

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

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 }, ...
};
```

6.4 Breakpoint support

SCP provides underlying mechanisms to track multiple breakpoints of different types. Most VMs 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 throuqh Z, are supported.

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

- **sim_brk_types**. This should be initialised by the VM (usually in the CPU reset routine) to a mask of all supported breakpoints.
- **sim_brk_dflt**. This should be initialised by the VM to the mask for the default breakpoint type.
- **sim_brk_summ**. This is 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:

- uint32 **sim_brk_test** (t_addr addr, int32 typ). This routine tests to see if a breakpoint of type *typ* is set for location *addr*. It returns 0 if a breakpoints is not set, and a bit mask of all breakpoints that match *typ* if any breakpoints are set.

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).