Writing a Simulator for the SIMH System
Revised 1-Dec-01 for V2.8-0

Overview

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

Some terminology: Each simulator consists of a standard *simulator control package* (SCP), 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.

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	char, unsigned char
int16, uint16	short, unsigned short
int32, uint32	int, unsigned int
t_int64, t_uint64	long long, _int64 (system specific)
t_addr	simulated address, int32
t_value	simulated value, unsigned int32 or int64
t_svalue	simulated signed value, int32 or int64
t_mtrec	mag tape record length, int32
t_stat	status code, int
t_bool	true/false value, int

[The inconsistency in naming t_int64 and t_uint64 is due to VC++, which uses int64 as a structure name member in the master Windows definitions file.]

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

DEVICE	device definition structure
UNIT	unit definition structure
REG	register definition structure
MTAB	modifier definition structure

VM Organization

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

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

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

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

Interface	Function
char **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[]**	table of pointers to simulated devices, NULL terminated
UNIT ***sim_consoles[]**	table of pointers to simulated consoles
char ***sim_stop_messages[]**	table of pointers to error messages
t_stat **sim_load** (…)	binary loader subroutine
t_stat **sim_inst** (void)	instruction execution subroutine
t_stat **parse_sym** (…)	symbolic instruction parse subroutine (optional)
t_stat **fprint_sym** (…)	symbolic instruction print subroutine (optional)

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; pdp1 for the PDP-1; pdp18b for the other 18-bit PDP's; pdp8 for the PDP-8; pdp11 for the PDP-11; nova for Nova; hp2100 for the HP 21XX; id4 for the Interdata 4; pdp10 for the PDP-10):

- *name*.h contains definitions for the particular simulator
- *name*_sys.c contains all the SCP interfaces except the instruction simulator
- *name*_cpu.c contains the instruction simulator and CPU data structures
- *name*_stddev.c contains the peripherals which were standard with the real system.
- *name*_lp.c contains the line printer.
- *name*_mt.c contains the mag tape controller and drives, etc.

The SIMH standard definitions are in sim_defs.h, the simulator control package in scp.c, and the operating-system dependent terminal routines in scp_tty.c. Additional libraries include sim_tmxr.c (header file sim_tmxr.h) for terminal multiplexors, and sim_sock.c (header file sim_sock.h) for network processing.

1.1 CPU Organization

Most CPU's perform at least the following functions:

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

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

1.1.1 Time Base

In order to simulate asynchronous events, such as I/O completion, the VM must define and keep a time base. This can be accurate (for example, nanoseconds of execution) or arbitrary (for example, number of instructions executed), but it must be consistently used throughout the VM. All existing VM's count time in instructions.

The CPU is responsible for counting down the event counter **sim_interval** and calling the asynchronous event controller **sim_process_event**. The record keeping for timing is done by SCP.

1.1.2 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 1401	6-bit	uint8
PDP-8	12-bit	uint16
PDP-11, Nova	16-bit	uint16
PDP-1	18-bit	uint32
PDP-10	36-bit	uint64

1.1.3 Interrupt Organization

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

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

```
if (int_enable && int_requests) { …process interrupt… }
```

Indeed, the interrupt enable flag can be made the highest bit in the interrupt request variable, and the two tests combined:

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

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

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

At a slightly higher complexity, interrupt requests do not correspond directly to device flags but are based on masking the device flags with an enable (or disable) mask. There are now three parallel variables: interrupt requests, device flags, and interrupt enable mask. The fetch-phase test does not change; however, the evaluation of whether an interrupt is pending now requires an extra step:

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

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.

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

1.1.4 I/O Dispatching

I/O dispatching consists of four steps:

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

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

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

1.1.5 Instruction Execution

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

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

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

Within this loop, the usual order of events is:

- If the event timer **sim_interval** has reached zero, process any timed events. This is done by SCP subroutine **sim_process_event**. Because this is the polling mechanism for user-generated processor halts (^E), errors must be recognized immediately:

```
                if (sim_interval <= 0) {
                        if (reason = sim_process_event ()) break;  }
```

- Check for outstanding interrupts and process if required.

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

- Check for an instruction breakpoint. SCP has no breakpoint facility, but it is customary to implement a single instruction breakpoint to help with processor code. All the existing CPU’s use the same mechanism, see the sources for details.

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

A few guidelines for implementation:

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

- The VM should provide some debugging aids. The existing CPU’s all provide an instruction breakpoint, an OLDPC register, and error stops on invalid instructions or operations.

1.2 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 canceling I/O requests. The former is easy, but the later requires a thorough understanding of device timing.

1.2.1 Device Timing

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

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

PTR (300 char/sec)	3.3 msec
PTP (50 char/sec)	20 msec

CLK (line frequency)	16.6 msec
TTO (30 char/sec)	33 msec

Mass storage devices, such as disks and tapes, do not have a fixed response time, but a start-to-finish time can be calculated based on current versus desired position, state of motion, etc.

For an externally timed device, there is no portable mechanism by which a VM can be notified of an external event. Because the only important externally timed device is the console keyboard, all current VM's poll for keyboard input, thus converting the externally timed keyboard to a pseudo-internally timed device.

SCP provides the supporting routines for device timing. SCP maintains a list of devices (called *active devices*) which 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 3. To set up a timed operation, the peripheral calculates a waiting period for a unit and places that unit on the active queue. The CPU counts down the waiting period. When the waiting period has expired, **sim_process_event** removes the unit from the active queue and calls a device subroutine. A device may also cancel an outstanding timed operation and query the state of the queue. The timing subroutines are:

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

- t_stat **sim_cancel** (UNIT *uptr). This routine removes the specified unit from the active queue. If the unit is not on the queue, no error occurs.

- int32 **sim_is_active** (UNIT *uptr). This routine tests whether a unit is in the active queue. If it is, the routine returns the time (+1) remaining; if it is not, the routine returns 0.

- double **sim_gtime** (void). This routine returns the time elapsed since the last RUN or BOOT command.

- uint32 **sim_grtime** (void). This routine returns the low-order 32b of the time elapsed since the last RUN or BOOT command.

- int32 **sim_qcount** (void). This routine returns the number of entries on the clock queue.

- t_stat **sim_process_event** (void). This routine removes all timed out units from the active queue and calls the appropriate device subroutine to service the time-out.

- int32 **sim_interval**. This variable counts down the first outstanding timed event. If there are no timed events outstanding, SCP counts down a “null interval” of 10,000 time units.

1.2.2 Clock Calibration

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

- int32 **sim_rtc_init** (int32 clock_interval). This routine initializes the clock calibration mechanism. The argument is returned as the result.

- int32 **sim_rtc_calb** (int32 tickspersecond). This routine calibrates the real-time clock. The argument is the number of clock ticks expected per second.

The simulator calls **sim_rtc_init** in the prolog of **sim_instr**, before instruction execution starts, and whenever the real-time clock is started. The simulator calls **sim_rtc_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_rtc_init (clk_delay));
etc.

/* clock service */

sim_activate (&clk_unit, sim_rtc_calb (clk_ticks_per_second);
```

1.2.3 Data I/O

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

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

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

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

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

- Cards have 12 bits of data per column, but the data is most conveniently viewed as (ASCII) characters. Existing card reader simulators do not support binary operation.

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.

1.2.3.1 Reading and Writing Data

The ATTACH command creates an association between a host file and an I/O unit. For non-buffered devices, ATTACH stores the file pointer for the host file in the **fileref** field of the UNIT structure. For buffered devices, ATTACH reads the entire host file into an allocated buffer pointed to by the **filebuf** field of the UNIT structure.

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

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.

1.2.3.2 Console I/O

SCP provides two routines for console I/O.

- t_stat **sim_poll_char** (void). This routine polls for keyboard input. If there is a character, it returns SCPE_KFLAG + the character. If the user typed the interrupt character (^E), it returns SCPE_STOP. If there is no input, it returns SCPE_OK.

- t_stat **sim_putchar** (int32 char). This routine types the specified ASCII character on the console. There are no errors.

Data Structures

The devices, units, and registers which 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.

1.3 device Structure

Devices are defined by the **device** structure (typedef **DEVICE**):

```
struct device {
        char            *name;                  /* name */
        struct unit     *units;                 /* units */
        struct reg      *registers;             /* registers */
        struct mtab     *modifiers;             /* modifiers */
        int             numunits;               /* #units */
        int             aradix;                 /* address radix */
        int             awidth;                 /* address width */
        int             aincr;                  /* addr increment */
        int             dradix;                 /* data radix */
        int             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 */
};
```

The fields are the following:

name device name, string of all capital alphanumeric characters.
units pointer to array of **unit** structures, or NULL if none.
registers pointer to array of **reg** structures, or NULL if none.
modifiers pointer to array of **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 31 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 32 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.

1.3.1 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 (typically just the CPU) must supply special examine and deposit routines. The calling sequences are:

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

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

1.3.2 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.

1.3.3 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_number) – Bootstrap the specified unit.

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.

1.3.4 Attach and Detach Routines

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

t_stat *attach_routine* (UNIT *uptr, char *file) – Attach the specified *file* to the unit *uptr*.

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

In practice, these routines always 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);
}
```

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_ATTABLE flag is set. During simulator exit, the detach routine is called unconditionally. 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.

1.4 unit Structure

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

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

The fields are the following:

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

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

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

```
UDATA           (action_routine, flags, capacity)
```

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

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

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

1.4.1 Unit Flags

The **flags** field contains indicators of current unit status. SIMH defines 11 flags:

flag name	meaning if set
UNIT_DISABLE	the unit responds to ENABLE and DISABLE.
UNIT_DIS	the unit is currently disabled.
UNIT_ATTABLE	the unit responds to ATTACH and DETACH.
UNIT_ATT	the unit is currently attached to a file.
UNIT_BUFABLE	the unit can buffer its data set in memory.
UNIT_MUSTBUF	the unit must buffer its data set in memory.
UNIT_BUF	the unit is currently buffering its data set in memory.
UNIT_ROABLE	the unit can be ATTACHed read only.

UNIT_RO	the unit is currently read only.
UNIT_SEQ	the unit is sequential.
UNIX_FIX	the unit is fixed capacity.
UNIT_BINK	the unit measures “K” as 1024, rather than 1000.

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

1.4.2 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.

1.5 reg Structure

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

```
struct reg {
        char        *name;                  /* name */
        void        *loc;                   /* location */
        int         radix;                  /* radix */
        int         width;                  /* width */
        int         offset;                 /* starting bit */
        int         depth;                  /* save depth */
        int32       flags;                  /* flags */
};
```

The fields are the following:

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

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.

Macros **ORDATA**, **DRDATA**, and **HRDATA** define right-justified octal, decimal, and hexidecimal 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 flags can be any of the normal register flags; REG_UNIT will be OR'd in automatically. For example, the following declares an arrayed registers of all the **UNIT** position fields in a device with 4 units:

```
        { URDATA        (POS, dev_unit[0].pos, 8, 31, 0, 4, 0) }
```

1.5.1 Register Flags

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

flag name	meaning if specified
PV_RZRO	print register right justified with leading zeroes.
PV_RSPC	print register right justified with leading spaces.
PV_LEFT	print register left justified.
REG_RO	register is read only.
REG_HIDDEN	register is hidden (will not appear in EXAMINE STATE).
REG_HRO	register is read only and hidden.
REG_NZ	new register values must be non-zero.
REG_UNIT	register resides in the **UNIT** structure.

1.6 mtab Structure

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

```
struct mtab {
        int32       mask;                   /* mask */
        int32       match;                  /* match */
        char        *pstring;               /* print string */
        char        *mstring;               /* match string */
        t_stat      (*valid)();             /* validation routine */
        t_stat      (*disp)();              /* display routine */
        void        *desc;                  /* location descriptor */
};
```

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

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

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

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

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

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

Extended MTAB entries have a different interpretation:

mask entry flags
MTAB_XTD extended entry
MTAB_VDV valid for devices
MTAB_VUN valid for units
MTAB_VAL takes a value
MTAB_NMO valid only in named SHOW
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
an int32 (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.
5. If **desc** is NULL, exit; validation routine presumably stored result.
6. If MTAB_VAL is set, parse the SET option for “option=n”, and store the value n in the register described by **desc**.
7. Otherwise, store the **match** value in the int32 pointed to by **desc**.

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

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

SHOW {dev|unit} <modifier> 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.

1.6.1 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) – 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.

1.6.2 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, void *desc) – output device- or unit-specific state for *uptr* to stream *st*. *desc* points to the **REG** or int32 used to store the parameter.

When the display routine is called for a regular MTAB entry, SHOW has output the **pstring** argument but has not appended a newline. When it is called for an extended MTAB entry, SHOW hasn't output anything. SHOW will append a newline after the display routine returns.

1.7 Other Data Structures

char **sim_name[]** is a character array containing the VM name.

int32 **sim_emax** contains the maximum number of words needed to hold the largest instruction or data item in the VM. Examine and deposit will process up to **sim_emax** words.

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

UNIT ***sim_consoles[]** is an array of pointers to the units of simulated consoles, alternating input and output. (If a console has only an input unit, the output slot should also point to the input unit.) This structure is only used for multi-console support. If the VM has only one console, the pointer should be NULL.

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

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

VM Provided Routines

1.8 Instruction Execution

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

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

1.9 Binary Load and Dump

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

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

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

1.10 Symbolic Examination and Deposit

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

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

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

If symbolic processing is not implemented, or the output value or input string cannot be parsed, these routines should return SCPE_ARG. If the processing was successful and consumed more than a single word, then these routines should return extra number of words (not bytes) consumed as a **negative** number. If the processing was successful and consumed a single word, then these routines should return SCPE_OK. For example, PDP-11 **parse_sym** would respond as follows to various inputs:

input	return value
XYZGH	SCPE_ARG
MOV R0,R1	SCPE_OK
MOV #4,R5	-1
MOV 1234,5670	-2

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

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

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

1.11 Multi-Terminal Support (Telnet)

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

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

```
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 Tlnt bin */
        int32         rxbpr;                      /* rcv buf remove */
        int32         rxbpi;                      /* rcv buf insert */
        int32         rxcnt;                      /* rcv count */
        int32         txbpr;                      /* xmt buf remove */
        int32         txbpi;                      /* xmt buf insert */
        int32         txcnt;                      /* xmt count */
        uint8         rxb[TMXR_MAXBUF];           /* rcv buffer */
        uint8         txb[TMXR_MAXBUF];           /* xmt buffer */
        };
```

The fields are the following:

conn	connection socket (0 = disconnected)
tsta	Telnet state
rcve	receive enable flag (0 = disabled)
xmte	transmit flow control flag (0 = transmit disabled)
dstb	Telnet bin mode disabled
rxbpr	receive buffer remove pointer
rxbpi	receive buffer insert pointer
rxcnt	receive count
txbpr	transmit buffer remove pointer
txbpi	transmit buffer insert pointer
txcnt	transmit count
rxb	receive buffer
txb	transmit buffer

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

```
struct tmxr {
        int32         lines;                      /* # lines */
        SOCKET        master;                     /* master socket */
        TMLN          *ldsc[TMXR_MAXLIN];         /* line descriptors */
        };
```

The fields are the following:

lines	number of lines (constant)
master	master listening socket (specified by ATTACH command)
ldsc	array of line descriptors

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

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

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

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

(1 << TMXR_V_VALID) **|** character

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

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

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

void **tmxr_putc_ln** (TMLN *lp, int32 chr) – output character *chr* to the line described by *lp*.

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

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

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

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

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

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

void **tmxr_msg** (SOCKET sock, char *msg) – output character string *msg* to socket *sock*.

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