Adding An I/O Device To A SIMH Virtual Machine

Updated 01-Dec-08 for SIMH V3.8-1

This memo provides more detail on adding I/O device simulators to the various virtual machines supported by SIMH.

1. SCP and I/O Device Interactions

1.1 The SCP Interface

The simulator control package (SCP) finds devices through the device list, DEVICE *sim_devices. This list, defined in <simname>_sys.c, must be modified to add the DEVICE data structure(s) of the new device to sim_devices:

```
extern DEVICE new_device;
:
DEVICE *sim_devices[] = {
      &cpu_dev,
      :
      &new_device,
      NULL };
```

The device then defines data structures for UNITs, REGISTERs, and, if required, options.

1.2 I/O Interface Requirements

SCP provides interfaces to attach files to, and detach them from, I/O devices, and to examine and modify the contents of attached files. SCP expects devices to store individual data words right-aligned in container words. The container words should be the next largest power of 2 in width:

Data word	Container word	
1b to 8b	8b	
9b to 16b	16b	
17b to 32b	32b	
33b to 64b	64b	(requires compile flag –DUSE_INT64)

1.3 Save/Restore Interactions

The Save/Restore capability allows simulations to be stopped, saved, resumed, and repeated. For save and restore to work properly, I/O devices must save and restore all state required for operation. This includes control registers, working registers, intermediate buffers, and mode flags.

Save and restore automatically handle the following state items:

- Content of declared registers.
- Content of memory-like structures.
- Device user-specific flags and DEV_DIS.
- Whether each unit is attached to a file and, if so, the file name.
- Whether each unit is active, and, if so, the unit time out.
- Unit U3-U6 words.
- Unit user-specific flags and UNIT_DIS.

There are two methods for handling intermediate buffers. First, the buffer can be made accessible as unit memory. This requires buffer-specific examine and deposit routines. Alternately, the buffer can be declared as an arrayed register.

2. PDP-8

2.1 CPU and I/O Device Structures

Simulated memory is kept in array uint16 M[MAXMEMSIZE]. 12b words are right justified in each array entry; the high order 4b must be zero.

The interrupt structure is implemented in three parallel variables:

- int32 int_req: interrupt requests. The two high order bits are the interrupt enable flag and the interrupts-not-deferred flag
- int32 dev_done: device done flags
- int32 int_enable: device interrupt enable flags

A device without interrupt control keeps its interrupt request, which is also the device done flag, in int_req. A device with interrupt control keeps its interrupt request in dev_done and its interrupt enable flag in int_enable. Pictorially,

```
+----+----+…+----+----+…+----+----+----+
|ion |indf| |irq1|irq2| |irqx|irqy|irqz| irq_req
+----+----+…+----+----+…+----+----+----+

+----+----+…+----+----+…+----+----+----+
| 0  | 0  | | 0  | 0  | |donx|dony|donz| dev_done
+----+----+…+----+----+…+----+----+----+

+----+----+…+----+----+…+----+----+----+
| 0  | 0  | | 0  | 0  | |enbx|enby|enbz| int_enable
+----+----+…+----+----+…+----+----+----+

<- fixed -> <-no enbl-> <- with enable->
```

Logically, the relationship is

```
int_req = (int_req & (OVHD+NOENB)) | (dev_done & dev_enable);
```

Macro INT_UPDATE maintains this relationship after a change to any of the three variables.

Device enable flags are kept in dev_enb. The device enable flag, by convention, is the same bit position as device interrupt flag.

I/O dispatching is done by explicit case decoding in the IOT instruction flow for CPU IOT's, and dispatch through table dev_tab[64] for devices. Each entry in dev_tab is a pointer to a device IOT processing routine. The calling sequence for the IOT routine is:

```
new_data = iot_routine (IOT instruction, current AC);
```

where

new_data<11:0>	=	new contents of AC
new_data<IOT_V_SKP>	=	1 if skip, 0 if not

new_data<31:IOT_V_REASON> = stop code, if non-zero

2.2 DEVICE Context and Flags

The DEVICE **ctxt** (context) field must point to the device information block (DIB), if one exists. The DEVICE **flags** field must specify whether the device supports the "SET ENABLED/SET DISABLED" commands (DEV_DISABLE). If a device can be disabled, the state of the device flag<DEV_DIS> must be declared as a register for SAVE/RESTORE.

2.3 Adding A New I/O Device

2.3.1 Defining The Device Number and Done/Interrupt Flag

Module pdp8_defs.h must be modified to add the device number definitions and the device interrupt flag definitions. The device number is the lowest device number that the device responds to (e.g, 060 for the RL8A):

```
#define DEV_NEW         0nn                     /* not 0,010,020-027 */
```

If the device has a separate interrupt enable, the interrupt flag must be added above INT_V_DIRECT, and the latter increased accordingly:

```
#define INT_V_TTI4      (INT_V_START+13)        /* clock */
#define INT_V_NEW       (INT_V_START+14)        /* new */
#define INT_V_DIRECT    (INT_V_START+15)        /* direct start */
:
#define INT_NEW         (1 << INT_V_NEW)
```

If the device has only an interrupt/done flag, it must be added between INT_V_DIRECT and INT_V_OVHD, and the latter increased accordingly:

```
#define INT_V_UF        (INT_V_DIRECT+8)        /* user int */
#define INT_V_NEW       (INT_V_DIRECT+9)        /* new */
#define INT_V_OVHD      (INT_V_DIRECT+10)       /* overhead start */
:
#define INT_NEW         (1 << INT_V_NEW)
```

2.3.2 Adding The Device Information Block

The device information block is declared in the device module, as follows:

```
int32 iotrtn1 (int32 instruction, int32 AC);
int32 iotrtn2 (int32 instruction, int32 AC);
:
DIB dev_dib = { DEV_NEW, num_iot_routines, { &iotrtn1, &iotrn2, … } };
```

DEV_NEW is the device number, and num_iot_routines is the number of IOT dispatch routines (allocated contiguously starting at DEV_NEW). If a device number in the range defined by [DEV_NEW, DEV_NEW + num_iot_routines - 1] is not needed, the corresponding dispatch address should be NULL.

3. PDP-11, MicroVAX 3900, VAX-780, and PDP-10

3.1 Memory

For the PDP-11, simulated memory is kept in array uint16 *M, dynamically allocated. For the MicroVAX 3900 and VAX-780, simulated memory is kept in array uint32 *M, dynamically allocated. For the PDP-10, simulated memory is kept in array t_uint64 *M, dynamically allocated. Because the three systems use different memory widths and different I/O mapping schemes, DMA peripherals that are shared among them use interface routines to access memory.

3.2 Interrupt Structure

The interrupt structure is implemented by array int_req, indexed by priority level (except on the PDP-10, where all levels are kept in one word). Each device is assigned a request flag in int_req[device_IPL], according to its priority, with highest priority at the right (low order bit). To facilitate access to int_req across the three systems, each device *dev* defines three variables:

INT_V_*dev* – the bit number of the device's interrupt request flag
INT_*dev* – the mask of the device's interrupt request flag
IPL_*dev* – the index into int_req for the device's priority level (PDP-11, MicroVAX 3900, and VAX-780 only)

Four macros allow simulated devices to access and manipulate interrupt structures independent of the underlying VM:

IVCL (dev) – vector locator for DIB (IPL * 32 + bit number)
IREQ (dev) – resolves to int_req[device_IPL]
CLR_INT (dev) – clears the device's interrupt request flag
SET_INT (dev) – sets the device's interrupt request flag

3.3 I/O Dispatching

3.3.1 Unibus/Qbus Devices

For Unibus and Qbus devices, I/O dispatching is done by table-driven address decoding in the I/O page read and write routines. Interrupt handling is done by table driven processing of vector and interrupt handling tables. These tables are constructed at run time from device information blocks (DIB's). Each I/O device has a DIB with the following information:

```
{ IO page base address, IO page length, read_routine, write_routine,
  num_vectors, vector_locator, vector, { &iack_rtn1, &iack_rtn2, … } }
```

The calling sequence for an I/O read is:

```
t_stat read_routine (int32 *data, int32 pa, int32 access)
```

The calling sequence for an I/O write is:

```
t_stat write_routine (int32 data, int32 pa, int32 access)
```

For both, the access parameter can have one of the following values:

READ	normal read
READC	console read (PDP-11 only)
WRITE	word write
WRITEC	console word write (PDP-11 only)
WRITEB	byte write

I/O read and I/O word write use word (even) addresses; the low order bit of the address should be ignored. I/O byte write uses byte addresses, and the data byte to be written is right-justified in the calling argument.

If the device has vectors, the vector_locator field specifies the position of the vector in the interrupt tables, using macro IVCL (dev). If the device has static interrupt vectors, they are specified by the DIB vector field and by the DIB num_vectors field. The device is assumed to have vectors at vector, …, vector + ((num_vectors –1) * 4). If the device has dynamic interrupt acknowledge routines, they are specified by the DIB interrupt acknowledge routines. An calling sequence for an interrupt acknowledge routine is:

```
int32 iack_rtn (void)
```

It returns the interrupt vector for the device, or 0 if there is no interrupt (passive release).

3.3.2 Massbus Devices (PDP-11, VAX-780 only)

For Massbus devices, I/O dispatching is done by table-driven address decoding in the Massbus adapter (RH for the PDP11, MBA for the VAX-780). These tables are constructed at run time from device information blocks (DIB's). Each Massbus device has a DIB with the following information:

```
{ Massbus number, 0, mb_read_routine, mb_write_routine,
  0, 0, 0, { &abort_routine } }
```

The calling sequence for a Massbus register read is:

```
t_stat mb_read_routine (int32 *data, int32 offset, int32 drive)
```

The calling sequence for a Massbus register write is:

```
t_stat mb_write_routine (int32 data, int32 offset, int32 drive)
```

For both, offset is the internal register offset of the Massbus register being accessed, and drive is the unit number of the Massbus controller being accessed. These routines can return the following status values:

SCPE_OK	access ok
MBE_NXD	non-existent drive
MBE_NXR	non-existent register
MBE_GOE	error attempting to initiate function

The abort routine is called if the Massbus adapter must stop a data transfer or reset the associated controllers. Its calling sequence is:

```
t_stat mba_abort (void)
```

The abort routine typically invokes the device reset routine to stop all transfers and reset all device controller state.

3.4 DEVICE Context and Flags

For the PDP-11, VAX, and PDP-10, the DEVICE **ctxt** (context) field must point to the device information block (DIB), if one exists. The DEVICE **flags** field must specify whether the device is a Unibus device (DEV_UBUS); a Qbus device with 22b DMA capability, or no DMA capability (DEV_QBUS); or a Qbus device with 18b DMA capability (DEV_Q18); a Massbus device

(DEV_MBUS); or a combination thereof. The DEVICE **flags** field must also specify whether the device supports the "SET ENABLED/SET DISABLED" commands (DEV_DISABLE). Lastly, the DEVICE **flags** field specifies whether the device addresses and vectors are autoconfigured (DEV_FLTA).

Most devices do not care whether the I/O bus is Unibus or Qbus. Those that do can use macro UNIBUS to see if the host bus is Unibus (true) or Qbus (false). On the PDP-11, UNIBUS is derived from the CPU model; on the PDP-10 and VAX-11/780, it is always true; and for the MicroVAX 3900, it is always false.

3.5 Memory Access Routines

3.5.1 Unibus/Qbus Devices

Unibus/Qbus DMA devices access memory through four interface routines:

```
int32 Map_ReadB (t_addr ba, int32 bc, uint8 *buf);
int32 Map_ReadW (t_addr ba, int32 bc, uint16 *buf);
int32 Map_WriteB (t_addr ba, int32 bc, uint8 *buf);
int32 Map_WriteW (t_addr ba, int32 bc, uint16 *buf);
```

The arguments to these routines are:

```
ba          starting memory address
bc          byte count
*buf        pointer to device buffer
```

Note that the PDP-10 can only share a small number of PDP-11 peripherals, because of its dependence on 18b transfers on the Unibus; and that all non-Massbus peripherals are on Unibus 3.

The routines return the number of bytes not transferred: 0 indicates a successful transfer. Transfer failures can occur if the mapped address uses an invalid mapping register or maps to non-existent memory.

3.5.2 Massbus Devices

Massbus devices access memory through three interface routines, for read, write, and write check respectively:

```
int32 mba_rdbufW (uint32 mbus, int32 bc, uint16 *buf);
int32 mba_wrbufW (uint32 mbus, int32 bc, uint16 *buf);
int32 mba_chbufW (uint32 mbus, int32 bc, uint16 *buf);
```

The arguments to these routines are:

```
mbus        Massbus adapter number
bc          byte count
*buf        pointer to device buffer
```

The routines the number of bytes successfully transferred. Transfer failures can occur if a mapped address uses an invalid mapping register, maps to non-existent memory, or on a write-check, if a miscompare occurs.

3.6 Adding A New I/O Device

3.6.1 Defining The I/O Page Region

I/O page regions are defined by a base address and a byte length. The base address is defined as an offset against the I/O page base address (IOPAGEBASE). These definitions are kept in pdp11_defs.h (vaxmod_defs.h, pdp10_defs.h). For example, if a new IPL 4 device has I/O addresses 17777700-17777707:

```
#define IOBA_NEWIPL4    (IOPAGEBASE + 017700)   /* base addr */
#define IOLN_NEWIPL4    010                     /* length = 8 bytes */
```

Note that the offsets are always the low order 13b of the I/O address, because the I/O page is only 8KB long.

3.6.2 Defining The Device Parameters

If the device can interrupt, pdp11_defs.h (vaxmod_defs.h, vax780_moddefs.h, pdp10_defs.h) must be modified to add the device interrupt flag(s) and priority level. The device flag(s) should be inserted using a spare bit (or bits) at the appropriate priority level. On the PDP-11, the PIRQ interrupt flags (PIR) must always be the last (lowest priority) device in the level.

/* IPL 4 devices */

```
#define INT_V_LPT       4
#define INT_V_NEW       5                       /* new IPL 4 dev */
#define INT_V_PIR4      6                       /* used to be 4 */
:
#define INT_NEW         (1u << INT_V_NEW)
:
#define IPL_NEW         4
```

The device vector(s) must also be defined:

```
#define VEC_NEW         0360
```

3.6.3 Adding The Device Information Block

The device information block is declared in the device module, as follows:

```
t_stat new_rd (int32 *data, int32 addr, int32 access);
t_stat new_wr (int32 data, int32 addr, int32 access);
int32 new_iack1 (void);
int32 new_iack2 (void);
:
DIB new_dib = { IOBA_NEW, IOLN_NEW, &new_rd, &new_wr,
      num_vectors, IVLC (NEW), VEC_NEW, { &new_iack1, &new_iack2, … };
```

3.6.4 Adding The Device To Autoconfiguration (PDP-11, VAX, VAX-780 only)

If the device needs to be autoconfigured, and it is not presently included in the autoconfiguration table, it must be added to table **auto_tab** in pdp11_io_lib.c. Entries are in rank order. The fields for each entry are:

char ***dnam**[4]	list of controller names for this device type, maximum 4
int32 **numc**;	number of controllers per device name (used by terminal multiplexers)
int32 **numv**;	number of vectors per controller

uint32 **amod**	address modulus
uint32 **vmod**	vector modulus
uint32 **fix[4]**	fixed CSR addresses, maximum 4; 0 = end of list
uint32 fixv[4]	fixed vectors, maximum 4; 0 = end of list

4 Nova

4.5 CPU and I/O Device Structures

Simulated memory is kept in array uint16 M[MAXMEMSIZE].

The interrupt structure is implemented in three parallel variables:

- int32 int_req: interrupt requests. The two high order bits are the interrupt enable flag and the interrupts-not-deferred flag
- int32 dev_done: device done flags
- int32 dev_disable: device interrupt disable flags

Pictorially,

```
+----+----+…+----+----+…+----+----+----+
|ion |indf| |irqa|irqb| |irqx|irqy|irqz| irq_req
+----+----+…+----+----+…+----+----+----+

+----+----+…+----+----+…+----+----+----+
| 0  | 0  | |dona|donb| |donx|dony|donz| dev_done
+----+----+…+----+----+…+----+----+----+

+----+----+…+----+----+…+----+----+----+
| 0  | 0  | |disa|disb| |disx|disy|disz| dev_disable
+----+----+…+----+----+…+----+----+----+

<- fixed -> <------- I/O devices ------>
```

Logically, the relationship is

```
int_req = (int_req & ~INT_DEV) | (dev_done & ~dev_disable);
```

Device enable flags are kept in iot_enb. The device enable flag, by convention, is the same bit position as device interrupt flag.

I/O dispatching is indirectly through dispatch table dev_table, which has one entry for each possible I/O device. Each entry is a structure of the form:

```
int32       mask;                   /* interrupt/done mask bit */
int32       pi;                     /* PI out mask bit */
t_stat      (*iot_routine)();       /* addr of I/O routine */
```

The I/O routine is called by

new_data = iot_routine (IOT pulse, IOT subopcode, AC value);

where

new_data<15:0>	=	new contents of AC, if DIA/DIB/DIC

new_data<IOT_V_SKP>	=	1 if skip, 0 if not
new_data<31:IOT_V_REASON>	=	stop code, if non-zero

4.6 DEVICE Context and Flags

The DEVICE **ctxt** (context) field must point to the device information block (DIB), if one exists. The DEVICE **flags** field must specify whether the device supports the "SET ENABLED/SET DISABLED" commands (DEV_DISABLE). If a device can be disabled, the state of the device flag<DEV_DIS> must be declared as a register for SAVE/RESTORE.

4.7 Memory Mapping

On mapped Nova's and on Eclipse's, DMA transfers use a memory map to translate 15b virtual addresses to physical addresses. The mapping function is called by:

```
int32 MapAddr(int32 map, int32 addr)
```

with the following arguments:

```
map         map number, usually 0
addr        virtual address
```

The routine returns the physical address to be used for the transfer.

4.8 Adding A New I/O Device

4.8.1 Defining The Device Number And The Done/Interrupt Flag

Module nova_defs.h must be modified to add the device number definitions and the device interrupt flag definitions.

```
#define DEV_NEW         0nn                     /* can't be 00, 01 */
```

Device flags are kept as a bit vector. If priority is unimportant, the device flag can be defined as one of the currently unused bits:

```
#define INT_V_NEW       1                       /* new */
:
#define INT_NEW         (1 << INT_V_NEW)
```

If the device requires a specific priority with respect to existing devices, it must be assigned the appropriate flag bit, and the other device flag bits moved up or down.

The device's PI mask bit must also be defined:

```
#define PI_NEW          000200
```

4.8.2 Adding The Device Information Block

The device information block is declared in the device module, as follows:

```
int32 iot (int32 pulse, int32 code, int32 AC);
:
DIB new_dib = { DEV_NEW, INT_new, PI_new, &iot };
```