A Revised I/O Simulation for the HP 21xx/1000

J. David Bryan, 14-Nov-2008; updated 5-Apr-2011

The HP 2100 simulator for the 21xx and 1000 series of machines originally modeled I/O interface communication with the CPU by dispatching I/O instructions to the interfaces for action. A revised model, based on dispatching I/O backplane signals, has been implemented to solve several problems inherent in the original design.

The HP I/O Hardware Structure

The structure of the I/O system is compatible across all HP 21xx/1000-M/E/F systems. The I/O backplane distributes a 16-bit data output path, a 16-bit data input path, and control and timing signals to the interface cards. All I/O card slots are electrically interchangeable, and an interface derives its I/O address (select code) from the slot into which it is installed. Lower-numbered slots have interrupt priority over higher-numbered ones.

An I/O timing cycle is divided into five periods, designated T2 through T6. On the early machines (2114-2116), these form a subset of, and are synchronous with, the machine cycle that occupies T0-T7. On the later microprogrammed machines (2100 and 1000), each microcycle occupies one T-period; the micromachine runs asynchronously with the I/O subsystem and synchronizes whenever an I/O micro-order is executed. Backplane signals are asserted during specific T-periods to control the timing of the interfaces.

The basic device control structure of a typical HP interface consists of a control and a flag flip-flop. A programmed “set control” instruction asserts the STC signal on the I/O backplane to set the control flip-flop on the interface and initiate operation on the I/O device. When an operation completes, the device sets the flag flip-flop, which asserts the FLG signal. The state of the flag can be tested under program control. A programmed “skip if flag set” or “skip if flag clear” instruction asserts either the SFS or SFC signal to test whether the flag flip-flop is set or clear, and the interface responds by asserting the SKF signal if the flag is in the state indicated by the request. This advances the CPU program counter, causing the next instruction to be skipped to indicate that the programmed condition was met.

Because device completion occurs asynchronously with I/O timing, a flag buffer flip-flop is inserted before the flag flip-flop. The flag buffer is set asynchronously by the device, and then the flag is set during the appropriate CPU T-period by the ENF signal. If DMA is employed, the flag also requests a DMA cycle by asserting SRQ. If the device requires a per-operation start signal, then a command flip-flop is added that is set by STC and cleared by device completion, although this flip-flop is not involved in the backplane interface.

The combination of flag buffer, flag, and control enables the generation of the IRQ (interrupt request) signal, presuming that no higher-priority device is interrupting. Priority is established through the chain of PRH and PRL signals passing from higher-priority interfaces to lower-priority interfaces; the chain is broken at the first interface that asserts flag and control. Interrupt acknowledgement asserts IAK, which clears the flag buffer and leaves the flag and control set to continue to hold off lower-priority devices until the interrupt service routine is complete.

Upon application of power to the CPU, the PON signal asserts to indicate that the power supply voltages have stabilized. Pressing the front-panel PRESET button asserts the POPIO signal to reset all interfaces to known states in preparation for running programs. PRESET or a programmed ”clear control” instruction directed to select code 0 asserts the CRS signal. A typical interface might use PON to enable output drivers to the device, POPIO to set the flag buffer and flag, and CRS to clear the control flip-flop. The combination of POPIO and CRS turns off the I/O device and places the interface in an idle state.

An important consideration is that while the foregoing structure is common, it is neither required nor universal. An interface card is free to drive the backplane signals in any manner that meets the I/O timing requirements. Indeed, a few interfaces depart from the standard implementation, either to improve I/O transfer speed, or to meet a particular device control requirement.

The Original SIMH Implementation

At its origin in revision 2.5, the HP simulator embodied the typical interface structure mentioned above. The device reset function simulated power-on and PRESET. The DEVICE structure’s context value pointed to a device information block (DIB) that contained the interface’s select code, the flag buffer, flag, control, and command flip-flops, and a pointer to the interface simulator’s I/O instruction handler. The CPU simulator dispatched I/O instructions to the addressed device, which altered its four flip-flop values accordingly.

The CPU simulator calculated DMA service requests by inspecting the flag values of each interface. Interrupt requests were calculated by ANDing the control, flag, and flag buffer values from each interface, and the priority chain was determined by ANDing the control and flag values. During interrupt acknowledgement, the CPU cleared the flag buffer value of the interrupting interface.

Because these calculations were done after each I/O operation, the values were stored in bit vectors during simulation runs for speed. However, to allow I/O device select codes to be reassigned during simulation stops and to allow user alteration of the flip-flop states, the values had to be stored in the DIBs. To accommodate both requirements, the values were copied between the DIBs and the bit vectors each time simulated execution started and stopped.

Problems with the Implementation

This implementation worked well with the initial devices supplied with the HP simulator. As new devices were added, though, minor issues arose, due to interfaces that did not follow the standard design.

For example, the control flip-flops on the 12606B Fixed-Head Disc Memory and 12610B Drum Memory interfaces that were added at version 2.9 are not tied into the interrupt request logic, so setting control, flag, and flag buffer does not generate an interrupt. Because interrupt generation was calculated in the CPU simulator, the interface simulator had to use the command flip-flop as the control flip-flop to avoid generating interrupt requests inappropriately.

At version 3.2, CPU interrupt acknowledgement handling added a special case for the 12581A and 12892B Memory Protect cards, as they clear both flag and flag buffer in response to IAK. Also at 3.2, the DMA service requests were separated from the flags, and a new SRQ flip-flop was added to the DIBs, as the forthcoming 13037D Disc Controller simulator required separate control over these two values. The new disc controller required notification of DMA transfer completion as well, so the EDT (end of data transfer) backplane signal was dispatched to all interfaces as a pseudo-I/O instruction.

At version 3.3, the `-P` option to the `RESET` command was added to allow device reset functions to differentiate between power-up reset and ordinary reset. The 12578A and 12895A DMA simulators had been clearing their control words as part of the reset handler. While this is correct for power-up, it is not correct for PRESET, and this error manifested itself in RTE “slow bootstrap” failures.

At version 3.6, the CRS backplane signal was introduced as another pseudo-I/O instruction. The original implementation had sent a CLC instruction to each interface in response to a CLC 0 execution. Most interfaces respond to CRS and CLC identically by clearing their control flip-flops. However, not all do. In particular, the DMA card clears control in response to CLC but control and command in response to CRS. Clearing command stops an in-progress DMA operation, which the original implementation failed to do.

Impasse

For version 3.8-1, a simulation of the 12936A and 12620A Privileged Interrupt Fences was planned. The PIFs are required to run the 12920A Terminal Multiplexer under the DOS and RTE operating systems. These systems run with the interrupt system off when servicing any device, as they are not reentrant. The 12920A is not buffered and will lose characters if it requests service while the interrupt system is off. The PIF is used in conjunction with special multiplexer drivers to break the priority chain to all lower-priority devices, allowing the interrupt system to remain on. This allows the higher-priority multiplexer to be serviced immediately, even during execution of a lower-priority device interrupt handler.

The 12936A has a unique behavior. Setting either control or flag denies priority. An interrupt occurs when flag and flag buffer are set and control is clear. The flag and flag buffer are cleared with the CLF instruction but set with the OTA/B instruction. This presented a problem because of the implicit assumptions of the roles of control, flag, and flag buffer by the CPU simulator. The only way that those assumptions could be maintained was if the PIF simulator made these translations between its internal flip-flop values and those maintained by the CPU:

$$CONTROL' = CONTROL + FLAG * FLAGBUF$$

$$FLAG' = 1$$

$$FLAGBUF' = \overline{CONTROL} * FLAG$$

The prime values would be presented to the CPU as the device flip-flop values, while the original values would be presented to the user when the device state was examined. While this would coerce the CPU into generating the correct interrupt request and priority chain behavior, interrupt acknowledgement would clear the flag buffer, which would have to be reset to the indicated value for proper operation. Fortunately, the correct value could be restored during processing of the STF instruction that would be sent to the card by the OS interrupt handler. Unfortunately, the visible state presented to the user would be wrong between these two events. Equally unfortunately, user alteration of the visible values would not be reflected in the translated values, because the CPU simulator simply copied the DIB values to the bit vectors when simulated execution began.

The choices, then, were to accept that the state display would be wrong and to disallow user changes to the flip-flop values, to add more special cases to the CPU simulator to accommodate the atypical I/O behavior, or to remodel the I/O simulation to allow interfaces to set the interrupt request and priority chain values directly. Given that the existing implementation embodied assumptions that were not valid across all I/O interfaces, and given that the number of special cases was increasing as the breadth of the HP device simulations increased, implementation of an I/O model closer to the actual hardware was selected.

The Revised I/O Implementation

Whereas the old model was based on dispatching I/O instructions, the new model is based on dispatching I/O backplane signals. This allows the interface to take whatever action it wants in response. Instead of examining the control, flag, and flag buffer flip-flop values, the CPU monitors these signals from the interface simulators:

- PRL — priority low
- IRQ — interrupt request
- SRQ — service request
- SKF — skip on flag

PRL indicates that interrupt requests by lower-priority devices may be granted. IRQ is set when an interface wants to interrupt the CPU. SRQ is set to initiate a DMA cycle. SKF indicates that the programmed flag test is true and that the next instruction should be skipped.

The interface simulators monitor reception of these signals and dispatch for action accordingly:

- CLC — clear the control flip-flop
- STC — set the control flip-flop
- CLF — clear the flag flip-flop
- STF — set the flag flip-flop
- SFC — skip if the flag is clear
- SFS — skip if the flag is set
- IOI — I/O data input
- IOO — I/O data output
- ENF — enable flag
- EDT — end of data transfer
- SIR — set interrupt request
- IAK — interrupt acknowledge
- CRS — control reset
- POPIO — power-on preset to I/O
- PON — power on normal

The first eight signals are generated as a result of I/O instructions: the LIA/B and MIA/B instructions generate IOI, the OTA/B instructions generate IOO, and the remaining signals are generated by their namesake instructions. ENF sets the flag buffer and flag flip-flops. EDT occurs at the end of a DMA transfer. SIR asks the interface to calculate and set its IRQ, PRL, and SRQ values. The CPU sends IAK to acknowledge an interrupt. CRS, POPIO, and PON have been discussed previously.

In addition to allowing more flexibility in interface design, the new implementation has a few other advantages:

- a more consistent structure (only signals are handled, rather than a mixture of signals and I/O instructions)
- elimination of special cases in the CPU simulator (each interface simulator determines its own responses)
- elimination of the flip-flop values from the DIB and of copying values between the DIB and the bit vectors (the CPU no longer examines flip-flop values, and the bit vectors are set at simulated execution start by sending SIR to all devices; no action is needed at execution stop)

- unified handling of flip-flop values (values exist in one place—as local variables in the individual device simulators—rather than in the DIB and in the bit vectors, reducing coding error potential)
- simplification of CPU interrupt determination (only the IRQ and PRL vectors need to be examined)
- simpler handling of power-on and preset conditions (the device reset function simply dispatches PON and/or POPIO and CRS to the signal handler; no duplication of the initialization code)

For efficiency, the simulator does not implement signal generation exactly as in the hardware. In hardware, ENF and SIR are periodic, PON is asserted continuously, and most signals are common to all interfaces and are qualified at the interface by the select code. Under simulation, signals are sent only when actions are to be taken and then only to the specific target device. For instance, PON is dispatched only once during power-on reset, rather than being included in every I/O cycle. SIR is dispatched only when flip-flops affecting the PRL, IRQ, or SRQ signals are changed, rather than after every instruction. ENF is sent only when the device indicates that the flag buffer and flag are to be set, whereas in hardware, ENF samples the flag buffer value at every T2 and sets the flag accordingly.

A Problem with the Revised Implementation

The initial revised I/O implementation modeled the parallel hardware backplane as a sequence of individual signal dispatches, with each signal assigned an enumeration value. For programmed I/O instructions, the CPU simulator sent single signals (e.g., STC) or a signal pair (e.g., STC + CLF) to the target device's signal handler. The CLF enumeration value was chosen so that the handler could separate the two signals from the sum.

While the CPU asserts at most two signals concurrently, DMA may assert up to five. A normal DMA I/O cycle consists of an IOI or IOO signal to transfer the data, a CLF signal to clear the device flag to complete the prior I/O request, and an optional STC signal to set the command flip-flop to begin the next request. In addition to these three signals, the last DMA cycle adds an EDT signal to indicate the end of the DMA transfer and an optional CLC signal to idle the device. These signals are asserted in specific T-periods, as follows:

	Input		Output	
Signal	**Normal Cycle**	**Last Cycle**	**Normal Cycle**	**Last Cycle**
IOI	T2-T3	T2-T3		
IOO			T3-T4	T3-T4
STC *	T3		T3	T3
CLC *		T3-T4		T3-T4
CLF	T3		T3	T3
EDT		T4		T4
* if enabled by DMA Control Word 1				

The initial implementation simulated a DMA cycle by dispatching the required signals sequentially. For example, a normal output cycle might send IOO and then STC + CLF, and a final output cycle might send IOO, then STC + CLF, then CLC, and then EDT.

A problem arose, however, when the forthcoming 12821A Disc Interface simulator was being written. This card takes certain actions when IOO and CLF or EDT are asserted concurrently. Because the DMA signals were dispatched sequentially, detection would fail.

A review of the existing device simulators showed that two other cards also acted upon multiple signals:

Interface	Device	Condition	Action
12566B	LPS	STC + CLC	Flag does not set in diagnostic mode
12821A	DI	CLC + CLF	Master reset
12821A	DI	IOO + CLF	Inhibit setting of end-of-transfer flag
12821A	DI	IOO + EDT	Sets last-byte-out flag
12875A	IPL	IOO + EDT	Delay DMA completion interrupt for TSB

The problem of sequential signal dispatching had been worked around in the LPS and IPL simulators, but there was no easy solution to the DI issues, and it was clear that a better I/O implementation was needed.

An Improved I/O Implementation

To address these issues, and to accommodate future I/O card simulators more generally, a fully parallel I/O signal dispatch was implemented. Each I/O cycle, whether originated by the CPU or DMA, now results in a single call on the device's signal handler. The signal parameter passed to the handler was replaced by a set of signals, and the DMA cycle simulator was rewritten to supply all of the signals required for a given I/O cycle concurrently. The signal handler still processes the signals sequentially, but a device simulator can now detect whether signals are issued together.

The signal handler processes a "concurrent" set of signals sequentially in ascending enumeration value order. The order of execution generally follows the order of T-period assertion. One complication is that the assigned T-periods for certain signals differ between CPU I/O and DMA I/O cycles:

Signal	CPU I/O Cycle	DMA I/O Cycle
IOI	T4-T5	T2-T3
STC	T4	T3
CLC	T4	T3-T4
CLF	T4	T3

The period shift allows sufficient time for SRQ assertion to steal consecutive I/O cycles from the CPU. This is not germane to simulation, so a single signal processing order is used for both CPU and DMA cycles.

Multiple-Device Signal Handlers

SIMH device simulators are usually written to handle a single instance of an I/O card or card set. There is no inherent provision for multiple copies of a given card, although cards are generally configurable for the maximum number of connected devices allowed by the hardware. For example, a disc controller card simulator may allow connection of up to eight disc drives, but a second controller card simulator with its complement of drives is not supported, except by duplicating the simulation code and assigning different device and function names.

In general, there is a one-to-one correspondence between a card and a DEVICE structure. A good example is the common two-card disc interface, such as the 13210A Disc Controller supporting one to four 7900 disc drives. The DP device simulator defines separate DEVICEs (DPD and DPC) for the data and command cards, as well as separate I/O signal handlers. This is optimal, as the cards are of different designs and respond differently to I/O signals.

In some cases, though, a peripheral may use two cards with identical or nearly identical behaviors. In this case, duplicating the I/O signal handler functions is unnecessary, more difficult to maintain, and may result in significant code bloat if the card operation is complex.

Current examples of this are the DMA devices and the 12875A Interprocessor Link device. DMA consists of two channels that are identical except for priority. The IPL device consists of two identical 12566A Microcircuit Interface cards—one used as an input device, and the other used as an output device. In each case, a common signal handler (or handlers, in the case of DMA, which has primary and secondary select codes for each channel) is employed, and the cards' state variables, such as control and flag flip-flops, are kept in arrays indexed by a zero-based card number. For DMA, the channel number is derived from the last bit of the select code addressed (2 or 6 $\rightarrow$ 0, 3 or 7 $\rightarrow$ 1). For the IPL, the card number is derived from an explicit select code comparison (input select code $\rightarrow$ 0, output select code $\rightarrow$ 1). In addition, the IPL signal handler needs the device and unit pointers associated with the card to access the flip-flop and buffer variables. Similarly, a common unit service routine is used for both cards, which needs the card number and the debug flags (via the device pointer) for the indicated card.

The forthcoming 12821A Disc Interface card is used to interface several device classes to the HP 1000. This card supports the HP 7906/20/25 ICD (Amigo) disc drives, the CS/80 family of disc drives, and the HP 7974/78 Amigo reel-to-reel tape drives. Under simulation, each class consists of a device and several units. Because the card is complex, a common signal handler for all three interfaces is desirable to avoid duplication of code.

To accommodate this, a slightly revised peripheral model is needed. The current one-to-one mapping of cards to DEVICEs to signal handlers is extended to provide a many-to-one map of cards to a common signal handler by including an explicit card number in the DIB. The one-to-one map of cards to DEVICEs remains.

I/O Device Simulator Details

The new I/O implementation requires changes in the CPU simulator and in each device simulator.

The CPU maintains the PRL, IRQ, and SRQ values for all devices in global bit vectors. Each vector requires a two-element array of unsigned 32-bit integers:

```
uint32 dev_prl [2] = { ~0, ~0 };
uint32 dev_irq [2] = {  0,  0 };
uint32 dev_srq [2] = {  0,  0 };
```

Element 0 holds the bits for devices with select codes 0-31 (0-37 octal), and element 1 holds the bits for devices with select codes 32-63 (40-77 octal). Within each element, the LSB corresponds to the lowest-numbered device. The initial values indicate that all devices are granting priority to lower-priority devices, and no device is requesting an interrupt or DMA service,

A device requests an interrupt by setting its bit in the IRQ vector and clearing its bit in the PRL vector. The lowest-numbered (highest-priority) request for which an unbroken priority chain exists (that is, all bits below its location are set) is granted. An IAK signal is sent to the device, which must clear its IRQ bit. This removes the interrupt source but maintains a hold-off of lower-priority requests.

When the CPU is called via *sim_instr* to begin executing instructions, it first resets each vector to its initial value. It then sends an SIR signal to every enabled device. Each device calculates the values of its IRQ, PRL, and SRQ responses and sets them into the vectors.

A simulator for a given device defines a DIB structure and places a pointer to it in the *ctxt* field of the associated DEVICE structure. The DIB contains a pointer to the I/O signal handler, the device select code, and a card index associated with the device. For signal handlers that serve only one device, the card index value is set to 0. If several devices are served, the card index values of the corresponding DIBs are set to 0, 1, 2, etc.

A simulator for device *dev* must declare the handler as:

```
IOHANDLER dev_io;
```

...and then define it as:

```
uint32 dev_io (DIB *dibptr, IOCYCLE signal_set, uint32 stat_data)
```

The CPU simulator calls the signal handler and passes a pointer to the device's DIB, a set of I/O signals, and a combined status and data value. The handler returns a combined status and data value representing the result of the operation.

The state variables that represent the standard control, flag, and flag buffer flip-flops used by the signal handler should be declared in a structure:

```
struct {
   FLIP_FLOP control;
   FLIP_FLOP flag;
   FLIP_FLOP flagbuf;
   } dev;
```

...and assigned values using the enumeration constants CLEAR and SET. If a signal handler is to serve multiple cards, an array of structures should be used:

```
struct {
   FLIP_FLOP control;
   FLIP_FLOP flag;
   FLIP_FLOP flagbuf;
   } dev [2];
```

...and any additional per-card state variables should be declared in the structure array as well. Additional device flip-flops, e.g., *command* or *srq*, may be declared either as scalars or as structure members, as desired.

The following signal macros are provided in *hp2100_defs.h* to aid implementation:

```
setSKF(B)   — set SKF to boolean value B
setPRL(S,B) — set PRL for select code S to boolean value B
setIRQ(S,B) — set IRQ for select code S to boolean value B
setSRQ(S,B) — set SRQ for select code S to boolean value B

setstdSKF(N) — set SKF from fields in structure N
setstdPRL(N) — set PRL from fields in structure N
setstdIRQ(N) — set IRQ from fields in structure N
setstdSRQ(N) — set SRQ from fields in structure N

PRL(S) — return boolean PRL state for select code S
IRQ(S) — return boolean IRQ state for select code S
SRQ(S) — return boolean SRQ state for select code S
```

The *setstdNNN* macros use the standard logic to set the indicated signal values. That is:

$$SKF = SFS * FLAG + SFC * \overline{FLAG}$$

$$PRL = \overline{CONTROL * FLAG}$$

$$IRQ = CONTROL * FLAG * FLAGBUF$$

$$SRQ = FLAG$$

For example:

```
setstdSRQ (dev);
```

...sets the SRQ vector bit for the select code given by *dibptr→select_code* to the value of variable *dev.flag*. The macros assume that the indicated structure contains fields named *control*, *flag*, and *flagbuf*.

If the signal handler serves multiple cards, then the structure name should be an array reference:

```
setstdSRQ (dev [card]);
```

...where *card* is the current card number (identified by *dibptr→card_index*). This would set the SRQ vector bit to the value of *dev [card].flag*.

If the standard logic is not applicable, a simulator may use the *setNNN* macros to set the indicated signals explicitly.

The IOHANDLER function dispatches the signal set as follows:

```
IOSIGNAL signal;
IOCYCLE  working_set = IOADDSIR (signal_set);

while (working_set) {
   signal = IONEXT (working_set);

   switch (signal) {
      case ioCLF:
         ...
         break;

      case ioSTF:
         ...
         break;

      [ additional signal handlers... ]

      default:
         ...
         break;
      )

   working_set = working_set & ~signal;
   }

return stat_data;
```

The IOADDSIR macro adds an *ioSIR* signal to the signal set if it contains any signal that potentially affects interrupt or DMA requests. If such a signal is present, SIR processing is added to update the PRL, IRQ, and SRQ signals. The IONEXT macro isolates the next signal in the execution order to be processed.

After that signal is processed, it is removed from the working set, and signal dispatching continues until the set is exhausted. The original signal set remains available to enable detection of concurrent signal assertions, if needed.

For example, executing an STC 10B,C instruction calls the handler with a *signal_set* value of *ioSTC* | *ioCLF*. The IOADDSIR macro adds an *ioSIR* signal to the working set, as both STC and CLF affect interrupt requests. The IONEXT macro extracts in turn the *ioSTC*, *ioCLF*, and *ioSIR* signals. Finally, a combined status and data value is returned to the caller.

A simulator for an interface card with the standard flip-flop logic and I/O buffers employs the following common signal handlers within the `switch` statement. For the flag logic:

```
case ioCLF:
   dev.flag = dev.flagbuf = CLEAR;
   break;

case ioSTF:
case ioENF:
   dev.flag = dev.flagbuf = SET;
   break;

case ioSFC:
   setstdSKF (dev);
   break;

case ioSFS:
   setstdSKF (dev);
   break;
```

Interface responses to the STF and ENF signals are usually identical, and therefore the *ioSTF* handler may simply fall into the *ioENF* handler. If the actions are different, however, then *ioSTF* must be given its own handler.

The CPU implements the SKF signal as the data value returned from the signal handler when it is called to process the SFS and SFC signals. The *setstdSKF* macro sets the *stat_data* value to *ioSKF* if the flag condition is true or to *ioNONE* if the flag condition is false. Although the standard responses to *ioSFC* and *ioSFS* are the same, separating the cases improves the optimization of the SKF value calculation.

For input and output:

```
case ioIOI:
   stat_data = IORETURN (status, dev_ibuf);
   break;

case ioIOO:
   dev_obuf = IODATA (stat_data);
   break;
```

The following macros in *hp2100_defs.h* assist in data handling:

```
IORETURN(E,D) — form return value from status E and data D
IODATA(C)     — extract data value from combined value C
IOSTATUS(C)   — extract status value from combined value C
```

For input, the handler returns a combined status and data value that is formed by the IORETURN macro. For output, the handler is called with a status of *SCPE_OK* combined with the value to be written. The IODATA macro isolates the data value.

For the control logic:

```
case ioPON:
   ...
   break;

case ioPOPIO:
   dev.flag = dev.flagbuf = SET;
   break;

case ioCRS:
case ioCLC:
   dev.control = CLEAR;
   break;

case ioSTC:
   dev.control = SET;
   break;
```

Application of main power generates the hardware signals PON, POPIO, and CRS. Most cards do not process PON, so the corresponding handler is usually omitted. POPIO and CRS are also generated for PRESET; CLC 0 generates CRS only. If the CRS action is the same as the CLC action, the *ioCRS* handler may simply fall into the *ioCLC* handler, as in the example above. Otherwise, *ioCRS* must be given its own handler.

For the interrupt logic:

```
case ioSIR:
   setstdPRL (dev);
   setstdIRQ (dev);
   setstdSRQ (dev);
   break;

case ioIAK:
   dev.flagbuf = CLEAR;
   break;
```

The CPU sends the IAK signal when the device's interrupt request is granted.

Finally, there should be a default handler for all signals that are not used by the simulator (e.g., PON or EDT):

```
default:
   break;
```

The return value from the signal handler contains two parts: a status code and a data value. The returned status normally is *SCPE_OK*, and this status is supplied in the *stat_data* parameter when the handler is called. The parameter value may be returned unmodified by the handler if desired. If an error status return is desired, the IORETURN macro should be used to form the value. Returning a status code other than *SCPE_OK* will cause a simulation stop.

The returned data value is significant only for the IOI signal, where the value will be stored in a register or memory, and for the SFS and SFC signals, where the value is the signal asserted in response (*ioSKF* or *ioNONE*). For all other signals, the returned data value is ignored.

The device reset function is called to simulate a power-on condition or a front-panel PRESET operation. The two states are differentiated by the ”P” value of the *sim_switches* global variable: a power-on reset is invoked with the `RESET -P` command, whereas PRESET is invoked with `RESET`.

If the interface responds to PON, the device reset function is implemented as follows:

```
t_stat dev_reset (DEVICE *dptr)
{
if (sim_switches & SWMASK ('P')) {     /* PON reset? */
   IOPOWERON (&dev_dib);
   ...   /* power-on simulator-specific init */
   }
else                                   /* PRESET device */
   IOPRESET (&dev_dib);

...      /* general simulator-specific init */

return SCPE_OK;
}
```

A `RESET -P` invocation uses the IOPOWERON macro in *hp2100_defs.h* to send *ioPON*, *ioPOPIO*, and *ioCRS* directly to the signal handler. This may be followed by any power-on actions specific to simulation. A `RESET` invocation uses the IOPRESET macro to send *ioPOPIO* and *ioCRS* directly to the handler. Finally, any required simulator-specific operations, e.g., canceling in-progress simulation events, are performed.

If the interface ignores PON, then the `if` statement above becomes:

```
if (sim_switches & SWMASK ('P')) {     /* power-on init? */
   ...   /* power-on simulator specific init */
   }
IOPRESET (&dev_dib);                   /* PRESET device */
```

If no actions specific to power-on reset are needed, then the entire `if` statement may be omitted.

The PON and POPIO signals cannot be generated programmatically, so the division of the initialization effort between the signal handler and the device reset function is arbitrary. However, the convention embodied in the current devices is to place actions implemented in the hardware (e.g., clearing flip-flops) in the signal handler and actions present only in simulation (e.g., canceling simulation events) in the reset function. The CPU simulator's signal handler and reset function (*cpuio* and *cpu_reset*, respectively) provide an illustration of this separation.

A device simulator generally sets the flag buffer and flag in response to operation completion, typically in the device's unit service routine. This may be done by:

```
dev_io (&dev_dib, ioENF, 0);           /* send ENF signal */
```

If special handling is required, e.g., because SRQ is separated from FLG on the interface, then ENF may be used to set the flag as above, and SRQ may be set explicitly:

```
dev.srq = SET;                         /* set SRQ flip-flop */
dev_io (&dev_dib, ioSIR, 0);           /* send SIR signal */
```

SIR must be dispatched if any of the flip-flops that affect the generation of interrupts or DMA requests are changed. For a standard interface where SRQ follows FLG, these would be the control, flag buffer, or flag flip-flops. If SRQ is independent of FLG, then SIR is required after changing the SRQ flip-flop as well.

One additional macro is provided in *hp2100_defs.h*:

```
IOERROR(B,E) — if boolean value B is true, return status E
```

This is employed in the unit service routine to return an error status (unit not attached, unit offline, etc.) when the device simulator is set to stop execution on an I/O error.

Summary

The original I/O simulation structure was based on devices handling I/O instructions from the execution stream. This was a good match to the typical HP interface card, as embodied in the set of devices provided with the release of the HP simulator. However, as more complex and higher-performance interfaces were added, special cases had to be included to allow for atypical behavior. Eventually, reimplementation based on a model of I/O backplane signals became attractive to alleviate restrictions of the original design. This new model also removed the special cases and allowed for easier future expansion of the simulated device repertoire.