JLAB-TN-99-014

Hall A Controls
Hardware Configuration for EPICS

J. Gomez
Jefferson National Accelerator Facility

May 2, 1999

Abstract

We describe the implementation of position independent and local (shareable) device handlers in the Experimental Physics and Industrial Control System (EPICS). The objectives were to make the EPICS device handlers used by Hall A portable and reusable so that developer effort could be minimized. Four years of operations have totally confirmed our expectations.

1 Introduction

The Experimental Physics and Industrial Control System (EPICS)[1] consists of a set of software components and tools that allow to create a distributed control system. The basic components of the system are:

- Operator Interfaces (OPI). These are UNIX based workstations able to run various EPICS tools like MEDM for display and control.
- Input Output Controllers (IOC). These are VME/VXI based crates containing a single board computer with the real-time operating system VxWorks, various I/O modules as well as interfaces to other I/O busses like serial or GPIB.
- Local Area Network (LAN). This is the communication path between the IOCs and the OPIs as well as among the IOCs.

Signal monitoring and control is performed by the IOCs. At the hart of an IOC is a memory resident database describing each of the signals to be monitored and controlled by the IOC. Each database entry (record) corresponds to a signal. When a given record executes, it must access the appropriate module to retrieve/update the signal value. It does so through a device handler routine. But, for the handler to be able to access the hardware, it must know how has the hardware been configured (i.e. assigned addresses and interrupts as well as any interface that it must go through to access the module). Incorporating the hardware configuration information of an IOC into the device handlers is, at present, an unsolved problem for EPICS.[2]

In the case of EPICS 3.12 and previous versions, the suggested procedure is to reserve the required hardware resources (i.e. address space and interrupts) in the EPICS file *module_types.h*. The information

in this file is then compiled into the various device handler binaries (i.e. allocation of hardware resources is static or hard-coded). The procedure is, however, not practical because: (a) the development of device handlers must be coordinated among all EPICS developers, regardless of development site, to avoid conflicts in address space and/or interrupt allocation, (b) the structure of *module_types.h* allows only for very limited configuration information and, (c) older and rarely used devices continue taking precious hardware resources from newer and more commonly used devices. For these reasons, developers are advised in the latest EPICS release (3.13) not to use *module_types.h* when developing new device handlers although no alternative is available or proposed. In reality, *module_types.h* had fallen in disuse long before the release of EPICS 3.13 and the scheme used for handler allocation of hardware resources is very site (and even project) specific.

Regardless of the EPICS version, two issues make EPICS device handlers non-portable and non-reusable:

- Static allocation of hardware resources.
- Monolithic handlers. The device handler code for a particular device, like one of the HP3458A Digital Multimeters of Fig. 1, also contains code to handle all intermediate modules and interfaces required to access the particular device (i.e. the IP488, VIPC610 and VMECHIP2 of Fig.1).

Due to the above two issues, porting of EPICS device handlers is a time consuming and expensive proposition. Developer time is required to change the static hardware resource allocation of a handler every time a hardware conflict is detected. Use of a monolithic device handler in an IOC requires to either replicate all intermediate hardware implicit in the handler or to rewrite the code. Furthermore, static monolithic device handlers are inefficient in terms of IOC memory use, developer effort as well as expandability. Memory use and developer effort because code to handle intermediate interfaces gets repeated in similar handlers. Expandability because a developer either over-allocates hardware resources assuming that there will be more devices in the future or modifies the code to re-allocate resources when the new devices are added. Finally, the developer has to face the dilemma of either imposing the use of homogenous hardware so that device handlers do not need to be modified or manage an EPICS distribution with multiple source copies of a given handler code each corresponding to different IOC hardware configurations. EPICS device handler portability and reuse can be obtained and all the above problems eliminated if the handler code is position independent and local instead of static and monolithic. These concepts are discussed below.

1.1 Position Independent Device Handlers

Position independent device handlers contain no absolute hardware addresses or interrupt vector numbers. Hardware addressing within the handler is relative to a set of device access registers which are loaded with the actual addresses occupied by the device at either IOC initialization time or the first time the handler is requested to execute.

The process of allocating hardware resources should be as automatic (no developer intervention) as possible within the bounds allowed by the hardware. As way of examples, a device handler needing an interrupt vector should contain the means to retrieve the next available interrupt vector, assign it and increment the interrupt vector counter. Other hardware resources, like VME or GPIB addresses, do not lend themselves to automatic configuration. The hardware address of VME and GPIB modules must be assigned before accessing the modules for the first time. In these cases, the device handler could retrieve the information at initialization time from an ASCII based IOC specific configuration file. Such

scheme provides for a flexible IOC configuration; After selecting an IOC to install the device, the ASCII configuration file for that particular IOC would need to be updated only if there are hardware resources that need to be configured manually (i.e. VME addresses). It is easy to check for hardware conflicts because all manually configured hardware resources for the chosen IOC are declared in this ASCII file. The IOC configuration ASCII file does not add complexity to the task of IOC management since already every EPICS IOC requires to load a separate database file(s). Furthermore, if several IOCs have the same hardware configuration, a single ASCII configuration file could be used for all of them.

The main benefits brought by position independent device handlers are:

- Device handler development and use is independent of specific IOC configuration, project or organization. There is no longer any need to impose a homogenous hardware nor to manage multiple copies of the same device handler each with different allocation of hardware resources.
- Device handlers can be made re-entrant. A re-entrant device handler is independent of the number of devices being managed simultaneously while at the same time it minimizes the use of IOC memory. The developer no longer needs to know how many instances of the same device are going to be managed simultaneously.

1.2 Local Device Handlers

A local device handler can only contain code to handle the particular device for which it was developed. Imposing this requirement in our monolithic HP3458A DMM example of Fig. 1 leads to four device handlers; one for each device and interface (HP3458A, IP488, VIPC610 and VMECHIP2). The main benefits are:

- Instead of a monolithic code which can not be re-used unless either the whole hardware configuration is replicated or the handler modified, we are left with a set of brick like modules (like the popular LEGO) which we can re-use to handle an infinite number of device configurations. As way of example, another HP3458A DMM could be added to the left side of Fig. 1, by putting together the IPIC, IP488 and HP3458A device handlers.
- Minimizes developer time (cost) since the developer has now only to concentrate on those devices for which there are no handlers available for re-use.

1.3 Shareable Device Handler Infrastructure

The requirements of position independence and locality that we have imposed on the EPICS device handlers are similar to the requirements that must be met by, in the jargon of computer systems, shareable object code. For that reason, position independent and local device handlers will be referred to as shareable device handlers from now on.

Similar to the case encountered in a computer system when using shareable object code, an infrastructure is required to load and link shareable handlers. This infrastructure consists of a link table and a loader/linker. The link table stores the entry point(s) of the various handlers (their memory location) as well as relations among handlers. The loader/linker function is to bring the various handlers into memory and to fill the link table with the necessary information to access and execute the handlers.

The link table organization is critical since it affects the operation efficiency of both the device handlers as well as the loader/linker. We expect an efficient link table if the chosen link table organization captures

the essential points of IOC hardware representation and device handler execution. These points are discussed in Section 2. Section 3 presents the actual link table software implementation used by the Hall A controls version of EPICS.

2 Link Table Representation

Let us return to the example hardware configuration shown in Fig.1 for a MVME162 single board computer. The IOC hardware architecture can be viewed as a tree:

- The root of the tree represents the CPU chip itself.
- The main trunk represents the CPU bus to main memory.
- The tree branches represent secondary buses which take-off from either the CPU bus (i.e. the VME bus of a single board computer like the MVME162) or from other secondary buses (i.e. a VME based serial interface like the VMIC6016 of Fig. 1).
- The tree leaves represent the end-devices which hang from the various buses (i.e. the DF853 power supply and the HP3458A digital multimeters of Fig. 1).

Traveling in the imaginary world of this tree, from the root (CPU) to the tree leaves (end-devices), one encounters points at which one or more branches take-off from either the main tree trunk or the branch that we are following. These points are tree nodes representing hardware interfaces between buses. An example of a tree node would be the VMIC6016 of Fig. 1. It acts as a tree node between 16 serial buses and the VME bus on which the card sits. Consider now all the branches which meet at a given tree node. The branch with the least number of tree nodes between it and the root will be referred to as the primary branch of the node. All other are secondary branches for that particular node. In the case of the VMIC6016 of Fig. 1, the primary branch is the VME bus and the secondary branches are each of the serial lines. The ports of a tree node device are also classified into primary and secondary according to the type of branch to which they connect. Returning to our VMIC6016 example, the VME interface of this device is the primary port and each of the serial ports are secondary ports.

End-devices (tree leaves) have one port only. The port is classified as a primary port because tree leaves can be viewed as a special case of a tree node with no secondary branches coming off it. The end-devices represent the boundary between the internal IOC architecture and the outside world signals that we want to monitor and/or control and which are represented by the EPICS database records

As discussed above then, the hardware configuration of an IOC can be viewed as a directed hardware tree. The tree is directed because at every junction we have defined a primary branch which is closer to the root than any other branch at the particular junction we are considering. We need now to consider how this idealized tree representation of an IOC hardware is transverse (traveled) when the various device handlers execute. This is discussed next.

Consider, for example, what happens when one of the HP3458A DMM of Fig. 1 is commanded to take a new measurement. The operation begins when an EPICS database record triggers the HP3458A handler to take a measurement. The HP3458A handler determines from the IOC hardware tree representation that this device is not directly connected to the CPU bus but to an IP488 interface. Since only those hardware devices residing on the CPU bus can be directly accessed by their handlers (i.e software routines executing at the CPU/memory level), the HP3458A handler directs all its operation requests to the IP488 handler. The process is repeated by the IP488 and VIPC610 handlers until it reaches the VMECHIP2 handler.

The VMECHIP2 handler determines from the IOC hardware tree representation that the VMECHIP2 hardware interface is located directly on the CPU bus and, consequently, can be accessed by the CPU. The VMECHI2 handler performs then the requested operation on the VMECHIP2 hardware and the results propagate down through the hardware chain. Two conclusions can be extracted from this example then:

- The device handler link table itself can not be implemented as a tree. A tree has a single entry point (the root) which will force us to scan (transverse) the tree every time a handler needs to execute. Such operations will be extremely inefficient. An array based link table is better suited to our needs since given an index into the array, we could immediately retrieve the necessary handler.
- Every entry of the array based link table must be able to implement a backward transversal node of the IOC hardware tree (i.e. from the leaves to the root).

In our previous example, we assumed that handlers could easily communicate between them. Such communication, however, requires a well defined application programming interface (API). To implement the API we note that, regardless of the internal workings of a given device, interaction with the device is totally determined by its hardware interface (or, in general, bus type). For example, access to any VME device requires an address value and modifier, the word length (i.e. 8, 16 or 32 bits), the type of operation (i.e. read or write) and a data buffer where the data read is to be stored or the data to be written is located. So, several APIs need to be defined, one for each bus type (i.e VME, GPIB and so on). A given device handler will implement the necessary API for each of its hardware interfaces. As an example, the HP3458A handler will implement the GPIB API only while the IP488 handler will implement both Industry Pack and GPIB APIs.

Handler communication not only requires well defined APIs (discussed above) but also a well defined protocol regarding the use of those APIs. This protocol can be deduced from the example process outlined above to take a new measurement with an HP3458A DMM:

- device handlers behave as clients only through their primary port; they make service requests and wait for the responses.
- device handlers behave as servers through each of their secondary ports; they wait for the arrival of a service request and return a response.

Busses not only represent natural boundaries to define the handlers APIs but also the device handler link table indexing. Note that, all devices located in a given bus must have a common API and depend from the same interface (server). The two HP3458A DMM of Fig. 1, for example, depend from the same IP488 interface. This suggests that each link table array entry should correspond to a bus since this will minimize the amount of memory used by the link table as well as the time necessary to access the server handler for backward transversal of the IOC hardware tree.

We have presented in this section the basic ideas guiding and choices made while implementing the device handler link table used by the Hall A controls version of EPICS to manage shareable device handlers. The actual software implementation is presented in the next section.

3 Link Table Implementation

The actual link table implementation can be found in the file *$EPICS/base/include/Hac_Cfg.h* where *$EPICS* represents the EPICS distribution home directory. In this section, we present an overview of the main implementation details.

The first thing needed is to be able to uniquely identify any device within the hardware tree of an IOC. While there are many ways to accomplish this, we choose to use three tags (positive numbers used):

- *BusId*: a unique but otherwise arbitrary number assigned to each tree branch (bus) of an IOC hardware configuration tree.
- *DevId*: a numeric code which allows to recognize the device type within a given bus type. This tag is bus dependent to facilitate detection of improper device configuration information (i.e. a GPIB based device like the HP3458A DMM can not be connected directly to a VME bus).
- *LU*: an instance number used to differentiate identical devices located on the same bus (i.e. same *BusId*).

The device handler link table consist of a global array of pointers to C structures of type *Hac_Bus*:

```
struct Hac_Bus  *Hac_BusPtr[Hac_MxBusId + 1];
```

where the parameter *Hac_MxBusId* represents the largest *BusId* number that can be assigned to any bus (100 at present). Note that the link table is indexed by the *BusId* tag.

The structure *Hac_Bus* is defined as:

```
struct Hac_Bus
  {
  unsigned short   BusType;
  unsigned short   depBusId;
  unsigned short   depDevId;
  unsigned short   depLU;
  unsigned short   depPort;
  union Hac_Vias   Bus;
  };
```

The field *BusType* holds a code a which uniquely identifies the type of bus. The actual codes are defined in the *Hac_Cfg.h* file. At present they are:

```
#define HacBTyp_CPU       0
#define HacBTyp_VME       1
#define HacBTyp_RsNA      2
#define HacBTyp_RsA       3
#define HacBTyp_IPACK     4
#define HacBTyp_GPIB      5
```

A *HacBTyp_CPU* bus represents the CPU chip to main memory connection of an IOC (the main trunk of our idealized hardware tree). There is always one and only one CPU bus per IOC. $BusId = 0$ has been reserved for this bus type. For all other bus types, there could be none, one or multiple instances in a given IOC configuration. A *HacBTyp_RsNA* bus is a serial non-addressable bus (i.e. only one device per bus) like the RS232C standard. A multi-drop (i.e. addressable) RS485 bus represents an example of a *HacBTyp_RsA* bus. *HacBTyp_IPACK* stands for the well known Industry Pack bus introduced by GreenSprings. The *HacBTyp_VME* and *HacBTyp_GPIB* entries represent VME and GPIB busses respectively. The *BusType* field is used, for example, by the handlers of devices which can be setup to

communicate trough RS232C or GPIB to determine which API they must use. Another example are devices which can be setup to communicate through a non-addressable (i.e. RS232C) or addressable (i.e. RS485) serial interface.

The fields *depBusId*, *depDevId*, *depLU* and *depPort* hold the tags which allow to uniquely identify the point of origin of the bus in which we are. The first three fields hold the *BusId*, *DevId* and *LU* of the device from which the bus originates. The field *depPort* holds the *Port* number within the device. These tags are meaningless for a CPU bus since by definition it does not depend from any other bus. These four fields represent the links required for backward transversal of the IOC hardware tree.

Let us consider an example of how these tags are used. Figure 2 shows the IOC hardware configuration of Fig. 1 with the various tags assigned. Consider the link table entry for the serial bus joining the VMIC6016 interface card with the DF853 power supply. A BusId = 2 was assigned to this bus. The link table entry **Hac_BusPtr[2]* would then point to a *Hac_Bus* structure instance with:

```
BusType  =  HacBTyp_RsNA.
depBusId = 5 (the VMIC6016 primary port is attached to the VME bus (BusId = 5))
depDevId = VMIC6016
depLU    = 0
depPort  = 2
```

The last member of the structure *Hac_Bus* is an instance of union *Hac_Vias* which allocates memory space for a *BusType* bus structure:

```
union Hac_Vias
  {
  struct HacBus_CPU    HacCPU;
  struct HacBus_VME    HacVME;
  struct HacBus_RSA    HacRSA;
  struct HacBus_RSNA   HacRSNA;
  struct HacBus_IPACK  HacIPACK;
  struct HacBus_GPIB   HacGPIB;
  };
```

The contents of each bus structure are discussed in detail below.

3.1 Bus Dependent Structures

3.1.1 CPU Bus

The CPU bus is characterized by the structure:

```
struct HacBus_CPU
  {
  unsigned short   IVEC;
  void             *pDevPriv[HacCPU_LstDev + 1][HacCPU_MxLU + 1];
  };
```

The field *IVEC* stores the next available interrupt vector number in the IOC. When a handler needs an interrupt vector, it retrieves the value stored in *IVEC*, allocates the vector and, increments the value of *IVEC* by one. Allocation of interrupt vectors at run time rather than statically (code) eliminates interrupt

collision, handler incompatibility due to interrupt pre-allocation and, it allows multiple instances of the same handler to be spawned as separate tasks. *IVEC* is set to 80 during IOC initialization by the device handler loader/linker.

The pointer array **pDevPriv[][]* is used to store any private structures that may be required by the device handlers (i.e. IPIC and VMECHIP2 in Figure 1). The parameter *HacCPU_MxLU* represents the largest number of instances allowed for any device in a CPU bus (2 at present) The parameter *HacCPU_LstDev* represent the last *DevId* defined for this bus type.

In the following sections, we will introduce various other busses and their corresponding handler APIs. There is no handler API for a CPU bus. Recall from our handler execution example in Section 2 that handlers, being software routines, reside at the CPU/memory level. Contrary to the case of all other bus types, the handler of a device attached to the CPU bus has direct access to the device hardware and, consequently, it does not need to communicate with any other handler to perform the requested operation (i.e. we have reached the root of the tree).

3.1.2 VME Bus

The VME bus is characterized by the structure:

```
struct HacBus_VME
  {
  unsigned char   AM[HacVME_LstDev +1][HacVME_MxLU + 1][HacVME_MxRg + 1];
  unsigned int    MBase[HacVME_LstDev + 1][HacVME_MxLU + 1][HacVME_MxRg + 1];
  unsigned int    MSize[HacVME_LstDev + 1][HacVME_MxLU + 1][HacVME_MxRg + 1];
  unsigned short  (*pBusFtn[HacVME_LstFtn + 2])();
  void            *pDevPriv[HacVME_LstDev + 1][HacVME_MxLU + 1];
  };
```

where parameters *HacVME_MxLU* and *HacVME_LstDev* are similar to those defined previously for other busses. The *HacVME_MxRg* parameter represents the maximum number of register banks (i.e. non-contiguous memory blocks) that a VME device can have (2 at present). Some VME devices use, for example, a small amount of VME A16 address space for control registers and a larger amount of A24(A32) address space for the data itself. For a given DevId, device instance LU and register bank number, arrays *AM[][][]*, *MBase[][][]* and *MSize[][][]* store the address modifier code required to access the memory block, its beginning address and length (in bytes) respectively. The pointer array **pDevPriv[][]* is similar to those defined previously for other busses.

The array of function pointers *(*pBusFtn[HacVME_LstFtn + 2])()* holds the entry points to the VME API functions. The VME API is given in Appendix A. Each VME API function uses an specific index in the above array and the parameter *HacVME_LstFtn* represents the last index assigned to a VME API function. In the case of the VME API, it consists of a single function (*HacVME_FtnIO()*) with the following index assignment:

```
#define HacVME_FtnIO     0
#define HacVME_LstFtn   HacVME_FtnIO
```

The specific implementation of the VME API function is provided by the device from which the bus originates (i.e. the server device). In the example of Fig. 2, the VMECHIP2 device handler provides the implementation of the VME API function *HacVME_FtnIO()* for the VME bus with $BusId = 5$. Entry points to the API functions of a particular bus type are held in a per bus instance structure (i.e. per

BusId) instead of by bus type (i.e. per *BusType*) because different servers, which could have different internal hardware architecture, are used to manage different instances of the same bus type. For example, in Fig. 2 there are two instances of an Industry Pack bus. The bus instance with $BusId = 7$ is managed by a VIPC610 device while the bus instance with $BusId = 3$ is managed by an IPIC device. Both of them implement the same IPACK API (to be discussed later) but the code implementation of those API routines are different because the internal hardware architecture of the IPIC is different from a VIPC610.

3.2 Serial Bus

Two different structures are used for serial busses. The non-addressable serial bus (RsNA) uses an structure of the form:

```
struct HacBus_RSNA  /* Single-device serial bus (i.e. RS232) */
  {
  SEM_ID          semRSNA;
  unsigned short  (*pBusFtn[HacRS_LstFtn + 1])();
  void            *pDevPriv;
  };
```

where the pointer **pDevPriv* is used to hold any private structure that the single serial device connected to this bus may need. The array of function pointers **pBusFtn[]* holds the entry points to the Serial API functions. The Serial API is given in Appendix B and it consists of three functions with the following index assignment in **pBusFtn[]*:

```
#define HacRS_FtnEOB      0
#define HacRS_FtnRX       1
#define HacRS_FtnTX       2
#define HacRS_LstFtn  HacRS_FtnTX
```

Serial interfaces like the RS232, RS422 and RS485 standards have limited bandwidth (typically less than 10kBytes/sec) and require a two step process to retrieve a value from a given device (send request/receive response). Because of the transaction speed and steps required, there is no guarantee that a transaction that is taken place will not be pre-empted by another transaction (i.e. transaction collision). The semaphore *semRSNA* is provided to avoid transaction collisions. A serial device handler must take *semRSNA* before performing any serial API function calls. After completing the transaction, the handler must then release the semaphore so that another handler or another invocation of the same handler (recall that sharable device handlers are re-entrant) by a different EPICS record can proceed.

The *HacBus_RSA* structure for an addressable serial bus (RsA) represents an extension of the *HacBus_RSNA* to be able to handle several devices in the same bus:

```
struct HacBus_RSA  /* Multi-device serial bus (i.e. RS422, RS485) */
  {
  SEM_ID          semRSA;
  unsigned short  LAdd[HacRS_LstDev +1][HacRS_MxLU + 1];
  unsigned short  (*pBusFtn[HacRS_LstFtn + 1])();
  void            *pDevPriv[HacRS_LstDev + 1][HacRS_MxLU + 1];
  };
```

Most of the structure entries and parameters are similar to those described earlier. The *LAdd[][]* array holds the serial address assigned to the various devices.

3.2.1 Industry Pack Bus

The Industry Pack bus is characterized by the structure:

```
struct HacBus_IPACK
  {
  unsigned char   IPSlot[HacIPACK_LstDev +1][HacIPACK_MxLU + 1];
  unsigned char   IRQ[HacIPACK_LstDev + 1][HacIPACK_MxLU][HacIPACK_MxIRg + 1];
  unsigned int    MBase[HacIPACK_LstDev + 1][HacIPACK_MxLU + 1];
  unsigned int    MSize[HacIPACK_LstDev + 1][HacIPACK_MxLU +1];
  unsigned short  (*pBusFtn[HacIPACK_LstFtn + 1])();
  void            *pDevPriv[HacIPACK_LstDev + 1][HacIPACK_MxLU + 1];
  };
```

where *HacIPACK_LstDev*, *HacIPACK_MxLU* and *hacIPACK_LstFtn* are parameters similar to those defined previously for other bus types. Industry Pack modules can have up to two interrupt vector registers (IR0 and IR1). The parameter *HacIPACK_MxIRg* represents the last interrupt vector register number (1).

The Industry Pack bus, like CAMAC or PCI, is a slot specific bus. The Industry Pack standard defines several address spaces for each Industry Pack module (slot): the ID, IO and MEMORY spaces. The sizes and base addresses (beginning) of the various address spaces is defined by the Industry Pack standard. The sizes are independent of slot but the base addresses are not. The device handler must know in which slot is the device located to be able to calculate the proper base address. The slot information is kept in the array *IPSlot[][]*. The array *IRQ[][][]* holds the interrupt request level (i.e. interrupt priority) at which the device wants to operate.

All Industry Pack modules implement the ID and IO address spaces while only some modules implement the MEMORY space. As previously indicated, the Industry Pack standard defines the MEMORY space base address and maximum range that it can be occupied by a module in a given slot. The standard does not specify however, that a module MEMORY space must begin at the base address specified in the standard (it needs only to be within the limits imposed by the standard) nor how should the unused part of the MEMORY address space be decoded (if at all). It is relatively common practice for manufacturers to begin the MEMORY space allocation of their modules at an address other than the base address specified by the standard (reduces cost of the hardware address decoder). For that reason, the arrays *MBase[][]* and *MSize[][]* are used to hold the beginning address and size of MEMORY space that a particular Industry Pack module may need.

As in the case of other busses, the pointer array **pDevPriv[][]* holds any private structures that may be required by the devices attached to this particular bus instance. The array of function pointers **pBusFtn[]* holds the entry points to the Industry Pack (IPACK) API functions. The IPACK API is given in Appendix C and it consists of two functions with the following index assignment in **pBusFtn[]*:

```
#define HacIPACK_FtnIO   0
#define HacIPACK_FtnIRQ  1
#define HacIPACK_LstFtn   HacIPACK_FtnIRQ
```

3.2.2 GPIB Bus

The GPIB bus is characterized by the structure:

```
struct HacBus_GPIB
  {
  SEM_ID          semGPIB;
  unsigned char   SCA;
  unsigned char   ADD[HacGPIB_LstDev + 1][HacGPIB_MxLU + 1];
  unsigned short  (*pBusFtn[HacGPIB_LstFtn + 1])();
  void            *pDevPriv[HacGPIB_LstDev + 1][HacGPIB_MxLU + 1];
  };
```

The purpose of the semaphore *semGPIB* is similar to the one discussed previously for a serial bus. The field *SCA* holds the GPIB address of the GPIB controller while the array *ADD[][]* holds the GPIB addresses of all non-controller devices attached to this bus instance. The array of function pointers **pBusFtn[]* holds the entry points to the GPIB API functions. The GPIB API is given in Appendix D and it consists of eleven functions with the following index assignment in **pBusFtn[]*:

```
#define HacGPIB_FtnCAC       0
#define HacGPIB_FtnEOB       1
#define HacGPIB_FtnEOI       2
#define HacGPIB_FtnGTS       3
#define HacGPIB_FtnREN       4
#define HacGPIB_FtnRXDATA    5
#define HacGPIB_FtnSIC       6
#define HacGPIB_FtnSTAT      7
#define HacGPIB_FtnTXDATA    8
#define HacGPIB_FtnTXRAW     9
#define HacGPIB_FtnROR      10
#define HacGPIB_LstFtn  HacGPIB_FtnROR
```

4 Device Handler Loader/Linker

The tasks of the device handler loader/linker are to load the required handlers into memory, create the necessary link table nodes and fill those nodes with the appropriate information for handler execution.

There are two general approaches that could be used to load/link the device handlers: delayed and immediate. In the case of delayed load/link, the loader creates at IOC initialization time a link table with all entries marked as uninitialized. When EPICS calls a handler with an uninitialized entry in the link table, the loader is called to load/link the handler and mark its link table entry as initialized. The process is repeated until all required handlers have been loaded and linked. In the case of immediate load/link, all necessary operations are performed at IOC initialization time and the loader/linker can then be removed from memory since it is no longer needed. Mixed approaches can also be implemented. In the case of the Hall A controls version of EPICS, the device handler loader/linker implements the immediate approach by forward transversal of the IOC hardware tree (i.e. from the root to the leaves). The hardware tree configuration is obtained from an IOC specific (or group of IOCs if they have the same hardware configuration) ASCII configuration file. As pointed out earlier, a configuration file is needed because several type of devices (i.e. VME and GPIB) used commonly with EPICS can not be configured automatically. The device handler loader can be found in *$EPICS/base/src/drv/HacCfg.c* where *$EPICS* represents the EPICS distribution home directory.

Appendix E gives an example of an ASCII file used for IOC hardware configuration. This is actually the hardware configuration file of an IOC in charge of managing the basic infrastructure (i.e. magnet signals, cryogenics and power supplies, vaccumm, collimator and so on) of one of the two High Resolution Spectrometers (HRS) of Hall A. Some of the *Device* names in Appendix E can be recognized from our previous examples in Figs. 1 and 2 while others are either new or slightly different due to space constraints in the figures. The HPE1313A is a VME based, 64 channels, 16 bits resolution, scanning analog-to-digital converter (ADC) manufactured by Hewlett Packard. The VMIC1182 is a VME based, 64 channels, digital input card manufactured by VMIC while the VMIC2210 is a VME based, 64 channels, relay card from the same manufacturer. The VMIC4140 represents a VME based, 32 channels, 12 bits resolution, digital-to-analog converter (DAC) from VMIC while the VMIC4116 is an 8 channels, 16 bits resolution, DAC from the same manufacturer. The DYNAPOWER and DFISIK853 are serial magnet power supplies manufactured by Dynapower and Danfisik respectively. The MP1000 represent serial based LVDT readout controllers manufactured by Lucas Schaevitz.[7] The gsIP488 is an Industry Pack based GPIB controller manufactured by GreenSprings. The LS450_GPIB represent GPIB based gaussmeters from LakeShore[8] while the PT2025_GPIB is a GPIB based NMR manufactured by Metrolab.[9]

The forward (from the root to the leaves) declaration of the IOC hardware can be easily recognized. A *BusId* must have been declared before any device can be attached to it. Since at the beginning there is only one pre-defined bus, the CPU bus with $BusId = 0$, the configuration file starts by declaring the devices attached to this bus and the busses which they give origin to. Consider the first device in the configuration file:

```
1. /*  BusId   Device    LU  NumSecBuses
2.      0    VMECHIP2   0       1
3. /*  DevHdlr               DevHdlr_Path
4.   HacNod_VmeChip2          ./core
5. /* PortNum   BusType   BusId (MUST be UNIQUE)
6.     0        VME              1
7. /* IHdlr  IHdlr_Path  IHdlr_Disp   IHdlr_DatFile
8.   NONE     NONE         KEEP          NONE
```

Lines beginning with /* represent comments and are not processed by the loader/linker. Line 2 declares that one instance of device VMECHIP2, with an assigned $LU = 0$, is attached to $BusId = 0$ (the CPU bus). It also declares that from this device originates one secondary bus (tree branch). The loader/linker treats devices with $NumSecBuses = 0$ different from devices with one or more secondary busses. Devices with $NumSecBuses = 0$ are end-devices. They represent the boundary between the internal IOC hardware architecture and the outside world signals that we want to monitor and/or control with EPICS. Signals and end-devices are handled by EPICS. The first through the database records and the second through the device handlers declared in the EPICS *devSup.ascii* file. The loader/linker does not load any of the end-devices handlers since these will be loaded by EPICS itself. Devices with one or more secondary busses are tree nodes (interfaces among busses) whose existence is not known to EPICS because the end-device handlers loaded by EPICS must be local. Line 4 declares to the loader/linker the name of the file containing the device handler to use for a VMECHIP2 device and its path. Line 6 declares the type of secondary bus (VME), the *Port* in the VMECHIP2 from which this bus originates (0) and the *BusId* assigned to this bus (1). Given the above information, the loader/linker performs the following operations:

- Checks that the given *BusId* has been declared before. For that purpose it checks if the the

link table entry pointer **Hac_BusPtr[BusId]* is *NULL* (not previously declared) or not (already declared). *BusId* = 0 is declared by the loader/linker during the creation of the link table array and, consequently, the check yields a valid entry. The loader/linker then determines the bus type corresponding to the given *BusId*.

- Checks that *DevId* = VMECHIP2 is a valid device for the given bus type.
- Creates an instance of the bus structure *Hac_Bus* and stores its location in the link table entry **Hac_BusPtr[1]*.
- Fills the various structure fields as *BusType* = VME, *depBusId* = 0, *depDevId* = VMECHIP2, *depLU* = 0, *depPort* = 0
- Loads into the IOC memory the specified file. The loaded file contains, besides any internal routine(s) needed by the handler, the VME API implementation routines and a handler initialization routine with the same name than the loaded file.
- The loader/linker then calls the handler initialization routine *HacNod_VmeChip2*.
- The *HacNod_VmeChip2* routine treats the memory space allocated under the field *Bus* of the structure *Hac_Bus* as an instance of a VME bus structure (*HacBus_VME*). It then stores the entry points of the VME API implementation routines in the array of function pointers **pBusFtn[]* of the VME bus structure.

Line 8 allows to call any other initialization routine that it may be needed. *IHdlr* represents the name of the file containing the initialization routine. Like in the case of the device handler, there could be many routines in this file but the calling point (i.e. the main routine) must have the same name than the file. *IHdlr_Path* represents the directory path to retrieve *IHdlr*. If *IHdlr_Disp* = *UNLOAD*, the loader/linker attempts to remove from memory the main initialization routine. Anything else leaves it in memory. *IHdlr_DatFile* represents the path and name of any data file that may be required by *IHdlr*.

Appendix F gives an example of a tree node device handler (i.e. a device with one or more *NumSecBuses*). The example choosen is for a VIPC610, a VME based Industry Pack carrier manufactured by GreenSprings (see Figs. 1 and 2). This example has been choosen because of its simplicity. As discussed above in the loader/linker operations, the VIPC610 device handler file HacNod_gsVipc610 contains two routines (GsVipc610_FtnIO and GsVipc610_FtnIRQ) which implement the IPACK bus API and an initialization routine (HacNod_GsVipc610) which, given a *BusId*, stores the entry point of the two IPACK API functions in the corresponding link table entry. The handler is re-entrant (i.e. capable of handling multiple VIPC610 carriers).

The operations performed by the load/linker for an end-device (*NumSecBuses* = 0) are simpler than for a tree node since there are no new busses to be created. Most of the operations consists of checking the data integrity and storing the information in already existing structures.

5 End-Device Handler Implementation

Appendix G gives the device handler implementation for a VME based HPE1313A ADC (an end-device). The code implementation is straight forward (almost trivial) since in the scheme described in this note, the handler code only has to deal with the internal workings of the ADC itself and not with the various busses and interfaces located between the CPU and the ADC.

Depending on the EPICS record type, different structures are used to pass parameters between the records and their corresponding device handlers. None of the existing structures, however, fitted the scheme described in this note to uniquely identify an end-device in the IOC hardware tree. A new structure (*HacIo*) was created for that purpose and it is used by all the database records with sharable device handlers. In the case of the handler given in Appendix G,

```
pHacIo = (struct HacIo *)&(pai->inp.value);
```

setups a pointer to access the information contained in the structure *HacIo*. That information is then used, through the link table, to gain access to the VME bus structure where the ADC is located:

```
pVME = (struct HacBus_VME *)&(Hac_BusPtr[pHacIo->BusId]->Bus);
```

and, finally, a request is sent to the VME API function HacVME_FtnIO to retrieve the data from the ADC:

```
/** Current value table located in reg bank 1, offset= 0, each entry 4 bytes **/
  status = (*pVME->pBusFtn[HacVME_FtnIO])(pHacIo->BusId,HPE1313A,pHacIo->LU,
     HacOp_RX,pVME->AM[HPE1313A][pHacIo->LU][1],
     pVME->MBase[HPE1313A][pHacIo->LU][1] + (pHacIo->CtlPt)*4,
     1,HacDTyp_Float,&Data);
```

The request will then propagate, with the aid of the link table, through other device handlers until it reaches a handler for a device located at the CPU bus. Is this handler the one that actually accesses the hardware directly and the results of that access propagate down through the hardware tree until they reach the ADC.

6 Conclusion

Shareable device handlers have been implemented in the Hall A controls version of EPICS. The objectives were to make the EPICS device handlers used by Hall A portable and reusable so that developer effort could be minimized. Development of the shareable device handlers started in 1995 and four years of operations have totally confirmed our expectations. In those four years, the Hall A controls systems have gone through innumerable changes (devices added, configuration changes, temporary setups, etc) each taking a small fraction of the time that would otherwise have taken with our limited personnel. We can only hope that the EPICS community would implement the use of shareable device handlers in future releases.

References

[1] EPICS documentation can be found at http://www.aps.anl.gov/asd/controls/epics-/EpicsDocumentation/WWWPages/EpicsDoc.html.

[2] Martin R. Kraimer, Application Developers guide R3.13.0.Beta12, Chapter 10 (June 1998). Document can be found at the location given in Ref. 1.

[3] MVME162 documentation can be found at http://www.mcg.mot.com/WebOS/omf/GSS/MCG-/products/boards/68k-vme.html.

[4] Information concerning the VMIC1182, 2210, 4116, 4140 and 6016 can be found at http://www.vmic.com.

[5] Documentation concerning the IP488 and VIPC610 can be found at http://www.greenspring.com.

[6] Information concerning the HP3458A DMM and HPE1313A ADC can be found at http://www.tmo.hp.com/tmo/TMTop/English/.

[7] Documentation on the MP1000 LVDT readout can be found at http://www.schaevitz.com.

[8] Information concerning the LS450 gaussmeters can be fount at http://www.lakeshore.com.

[9] Documentation on the PT2025 NMR can be obtained at http://www.metrolab.com.

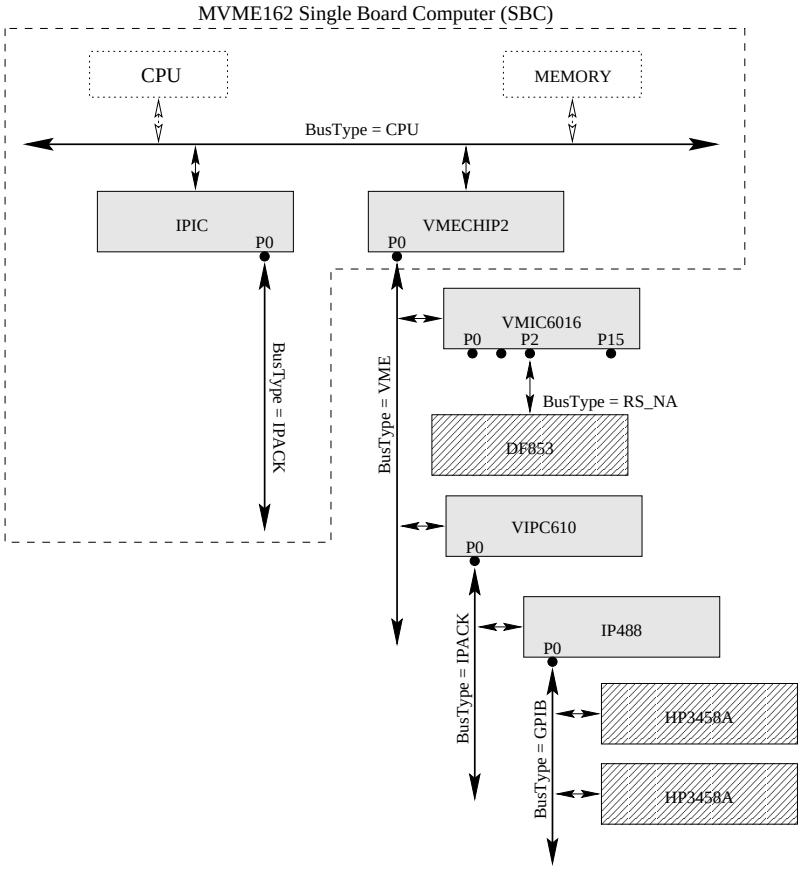

Figure 1: Example of an IOC hardware resource configuration for the Motorola MVME162[3] single board computer (SBC). In this block diagram, IPIC represents the onboard hardware bridge between the CPU to memory bus (CPU bus for short) and the Industry Pack (IPACK) bus. The VMECHIP2 is the onboard bridge between the CPU bus and the VME bus external to the SBC, the VMIC6016 represents a VME based, 16 ports (channels) interface manufactured by VMIC[4] while the DF853 represents a RS232C (non-addressable) serial (RS_NA) based magnet power supply manufactured by Danfisik. The VIPC610 is a VME based Industry Pack module carrier manufactured by GreenSprings,[5] the IP488 is an IPACK based interface to the GPIB bus also manufactured by GreenSprings and, the HP3458A are GPIB based Digital Multimeters from Hewlett Packard.[6]

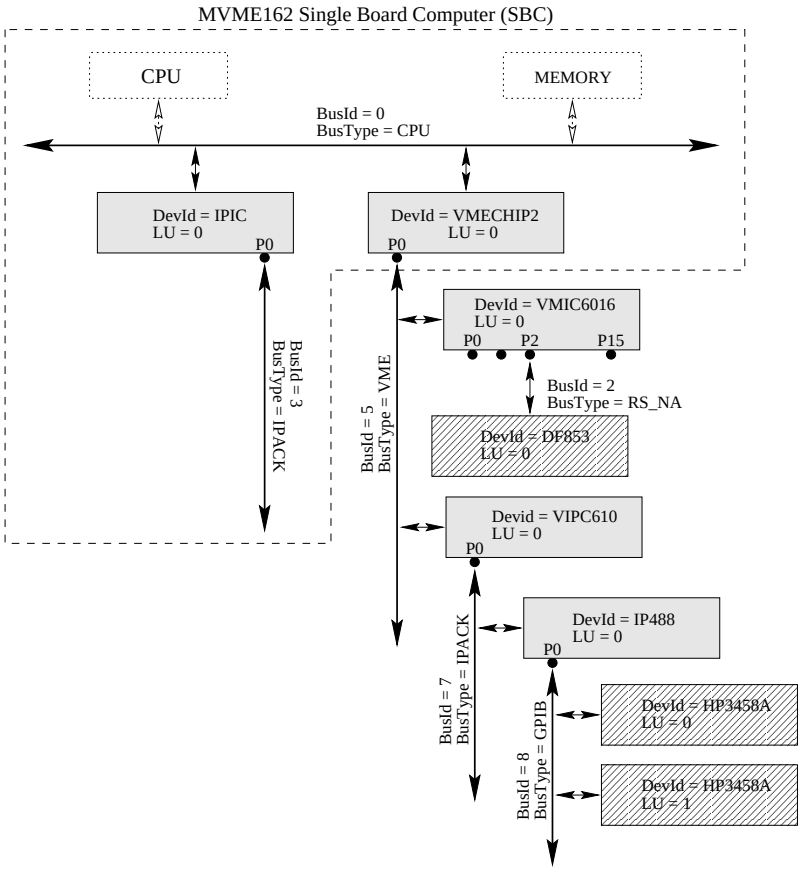

Figure 2: Example of an IOC hardware tree configuration. The *BusId*, *DevId*, LU and *Port* tags discussed in the main text have been assigned to the devices and busses of Fig. 1 to uniquely identify them. The *BusId* tag has being assigned somewhat randomly to emphasize that the only requirement that this tag must satisfy is to be unique within the particular IOC hardware tree. $BusId = 0$ is reserved for the CPU bus. The LU tag must be unique but otherwise arbitrary within a given *BusId* and *DevId*. The HP3458A DMM with $LU = 0$ could, for example, have been assigned other value as long as it is different from 1 which is being used by the second HP3458A DMM. The *Port* tag for each device must be assigned according to the scheme used by the device handler to identify the various hardware ports of the device. Normally, this will be the numbering scheme used by the hardware manufacturer.

A VME Bus API

```
/*====================================================================*/
/* Function: HacVME_FtnIO                                             */
/* Purpose:  Read/write data buffer                                   */
/* Return:   error code                                               */
/*====================================================================*/
unsigned short HacVME_FtnIO
  (
  unsigned short BusId,  /* BusId of client VME device               */
  unsigned short DevId,  /* DevId of client  VME device              */
  unsigned short LU,     /* LU of client VME device                  */
  unsigned char  OP,     /* Requested op: read or write              */
  unsigned char  AM,     /* Address modifier                         */
  unsigned int   ADD,    /* Address within VME space selected        */
  unsigned int   NTRY,   /* Number of buffer entries to read/write   */
  unsigned char  DTYP,   /* Data type                                */
  void           *pBUFF  /* Pointer to buffer                        */
  )
```

B Serial (RsA and RsNA) Bus API

```
/*=================================================================*/
/* Function: HacRS_FtnEOB                                          */
/* Purpose: Query/set End-Of-Block (EOB) code                      */
/* Notes:    EOB is used by HacRS_FtnRX below                      */
/*           On reads, EOB argument returns present value          */
/*           On writes, EOB argument holds new value to be set     */
/*           If EOB = 0x00, HacRS_FtnRX ignores EOB                */
/* Return:   error code                                            */
/*=================================================================*/
unsigned short HacRS_FtnEOB
  (
  unsigned short BusId,  /* BusId of client serial device          */
  unsigned short DevId,  /* DevId of client serial device          */
  unsigned short LU,     /* LU of client serial device             */
  unsigned short OP,     /* Requested op: read or write            */
  unsigned char  EOB
  )



/*=================================================================*/
/* Function: HacRS_FtnRX                                           */
/* Purpose:  Receive data buffer                                   */
/* Notes:    Function reads until EOB code is found (if enabled) or*/
/*           TMOUT expires. TMOUT must be > 0                      */
/* Return:   error code                                            */
/*=================================================================*/
unsigned short HacRS_FtnRX
  (
  unsigned short BusId,  /* BusId of client serial device          */
  unsigned short DevId,  /* DevId of client serial device          */
  unsigned short LU,     /* LU of client serial device             */
  unsigned short TMOUT,  /* System clock ticks to complete op      */
  unsigned int   MXBYTE, /* Buffer length                          */
  void           *pBUFF, /* Ptr to buffer                          */
  unsigned int   *BCNT   /* Ptr to actual number of bytes read     */
  )



/*=================================================================*/
/* Function: HacRS_FtnTX                                           */
/* Purpose:  Send data buffer                                      */
/* Notes:    TMOUT must be > 0                                     */
```

```
/* Return:   error code                                                 */
/*======================================================================*/
unsigned short HacRS_FtnTX
  (
  unsigned short BusId,  /* BusId of client serial device             */
  unsigned short DevId,  /* DevId of client serial device             */
  unsigned short LU,     /* LU of client serial device                */
  unsigned short TMOUT,  /* System clock ticks to complete op         */
  unsigned int   NBYTE,  /* Number of bytes to be written             */
  void           *pBUFF  /* Ptr to buffer                             */
  )
```

C Industry Pack (IPACK) Bus API

```
/*=================================================================*/
/* Function: HacIPACK_FtnIO                                        */
/* Purpose:  Read/write data buffer                                */
/* Return:   error code                                            */
/*=================================================================*/
unsigned short HacIPACK_FtnIO
  (
  unsigned short BusId,  /* BusId of client IP device              */
  unsigned short DevId,  /* DevId of client IP device              */
  unsigned short LU,     /* LU of client IP device                 */
  unsigned char  OP,     /* Requested op: read OR write            */
  unsigned char  SPCE,   /* IP address space: IO, ID or MEM        */
  unsigned int   ADD,    /* Address offset within IP space selected*/
  unsigned int   NTRY,   /* Number of buffer entries to read OR write */
  unsigned char  DTYP,   /* Data type                              */
  void           *pBUFF  /* Pointer to buffer                      */
  )



/*=================================================================*/
/* Function:  HacIPACK_FtnIRQ                                      */
/* Purpose:   Query/set interrupt vector                           */
/* Note:      On reads, IRQ holds contents of IREG                 */
/*            On writes, value in IRQ is written to IREG           */
/*            Interrupts are disabled if content of IREG is 0      */
/* Return:    error code                                           */
/*=================================================================*/
unsigned short HacIPACK_FtnIRQ
  (
  unsigned short BusId,  /* BusId of client IP device              */
  unsigned short DevId,  /* DevId of client IP device              */
  unsigned short LU,     /* LU of client IP device                 */
  unsigned char  OP,     /* Requested op: read OR write            */
  unsigned char  IREG,   /* IP interrupt register: 0 or 1          */
  unsigned char  IRQ     /* Interrupt vector                       */
  )
```

D GPIB Bus API

```
/*====================================================================*/
/* Function: HacGPIB_FtnCAC                                           */
/* Purpose:  Take control of GPIB bus (i.e. assert ATN line).         */
/* Return:   error code                                               */
/*====================================================================*/
unsigned short HacGPIB_FtnCAC
  (
  unsigned short BusId,  /* BusId of client GPIB device               */
  unsigned short DevId,  /* DevId of client GPIB device               */
  unsigned short LU,     /* LU of client GPIB device                  */
  unsigned char  ASYNC   /* ASYNC = 0: take control synchronously     */
  )                      /* ASYNC = 1: take control asynchronously    */



/*====================================================================*/
/* Function: HacGPIB_FtnEOB                                           */
/* Purpose:  Query/set End-Of-Block (EOB)                             */
/* Notes:    On read, EOB holds present value                         */
/*           On write, EOB holds new value to be set                  */
/*           EOB = 0x00 disables use of EOB                           */
/* Return:   error code                                               */
/*====================================================================*/
unsigned short HacGPIB_FtnEOB
  (
  unsigned short BusId,  /* BusId of client GPIB device               */
  unsigned short DevId,  /* DevId of client GPIB device               */
  unsigned short LU,     /* LU of client GPIB device                  */
  unsigned char  OP,     /* Requested op: read or write               */
  unsigned char  EOB
  )



/*====================================================================*/
/* Function: HacGPIB_FtnROR                                           */
/* Purpose:  Query/set Return-On-Ready flag                           */
/* Notes:    If ROR = TRUE, the server HacGPIB_FtnTXDATA routine      */
/*           delays return (and any further GPIB bus activity) until  */
/*           the device releases the Not-Ready-For-Data (NRFD) line.  */
/*           Most GPIB instruments assert the NRFD line to indicate   */
/*           that they are not ready to receive new commands but bus  */
/*           activity like GTS and ATN can continue. There are a few  */
/*           devices which, however, can not tolerate such activity.  */
```

```
/*           On read, ROR holds present value of flag                  */
/*           On write, ROR holds value to be set                       */
/*           ROR is either TRUE or FALSE                               */
/* Return:   error code                                                */
/*====================================================================*/
unsigned short HacGPIB_FtnROR
  (
  unsigned short BusId,  /* BusId of client GPIB device                */
  unsigned short DevId,  /* DevId of client GPIB device                */
  unsigned short LU,     /* LU of client GPIB device                   */
  unsigned char  OP,     /* Requested op: read or write                */
  unsigned char  ROR
  )



/*====================================================================*/
/* Function: HacGPIB_FtnEOI                                           */
/* Purpose:  Query/set use of End-Of-Input (EOI) by HacGPIB_FtnTXDATA */
/* Notes:    On read, EOI holds present value                         */
/*           On write, EOI holds new value to be set                  */
/*           HacGPIB_FtnTXDATA asserts the EOI line if EOI = TRUE     */
/* Return:   error code                                               */
/*====================================================================*/
unsigned short HacGPIB_FtnEOI
  (
  unsigned short BusId,  /* BusId of client GPIB device                */
  unsigned short DevId,  /* DevId of client GPIB device                */
  unsigned short LU,     /* LU of client GPIB device                   */
  unsigned char  OP,     /* Requested op: read or write                */
  unsigned char  EOI
  )



/*====================================================================*/
/* Function: HacGPIB_FtnGTS                                           */
/* Purpose:  Go-To-Standby (GTS) (i.e. release ATN line)              */
/* Return:   error code                                               */
/*====================================================================*/
unsigned short HacGPIB_FtnGTS
  (
  unsigned short BusId,  /* BusId of client GPIB device                */
  unsigned short DevId,  /* DevId of client GPIB device                */
  unsigned short LU      /* LU of client GPIB device                   */
  )
```

```
/*===================================================================*/
/* Function: HacGPIB_FtnREN                                          */
/* Purpose:  Set/release Remote Enable line (REN)                    */
/* Notes:    REN = FALSE releases Remote Enable line                 */
/*           REN = TRUE set Remote Enable line                       */
/* Return:   error code                                              */
/*===================================================================*/
unsigned short HacGPIB_FtnREN
  (
  unsigned short BusId,  /* BusId of client GPIB device              */
  unsigned short DevId,  /* DevId of client GPIB device              */
  unsigned short LU,     /* LU of client GPIB device                 */
  unsigned char  REN
  )



/*===================================================================*/
/* Function: HacGPIB_FtnRXDATA                                       */
/* Purpose:  Read data buffer                                        */
/* Notes:    Function reads until EOB is found (if enabled), EOI is  */
/*           detected or TMOUT expires. This function sets the GPIB  */
/*           controller as listener, the client device as talker,    */
/*           reads the data and clears all listeners and talkers     */
/*           before returning.                                       */
/* Return:   error code                                              */
/*===================================================================*/
unsigned short HacGPIB_FtnRXDATA
  (
  unsigned short BusId,  /* BusId of client GPIB device              */
  unsigned short DevId,  /* DevId of client GPIB device              */
  unsigned short LU,     /* LU of client GPIB device                 */
  unsigned short TMOUT,  /* System clock ticks. Must be > 0          */
  unsigned int   MXBYTE, /* Length of buffer                         */
  void           *pBUFF  /* Buffer                                   */
  unsigned int   *BCNT   /* Number of bytes received                 */
  )



/*===================================================================*/
/* Function: HacGPIB_FtnSIC                                          */
/* Purpose:  Send Interface Clear (SIC)                              */
```

```
/* Return:   error code                                                */
/*====================================================================*/
unsigned short HacGPIB_FtnSIC
  (
  unsigned short BusId,  /* BusId of client GPIB device               */
  unsigned short DevId,  /* DevId of client GPIB device               */
  unsigned short LU      /* LU of client GPIB device                  */
  )



/*====================================================================*/
/* Function: HacGPIB_FtnSTAT                                          */
/* Purpose:  Query present status of GPIB interface                   */
/* Return:   error code                                               */
/*====================================================================*/
unsigned short HacGPIB_FtnSTAT
  (
  unsigned short BusId,  /* BusId of client GPIB device               */
  unsigned short DevId,  /* DevId of client GPIB device               */
  unsigned short LU      /* LU of client GPIB device                  */
  unsigned short STAT    /* status                                    */
  )



/*====================================================================*/
/* Function: HacGPIB_FtnTXDATA                                        */
/* Purpose:  Send a data buffer                                       */
/* Notes:    If the EOI flag is set (see HacGPIB_FtnEOI) then the EOI */
/*           line will be asserted when the last character in pBuff   */
/*           is sent. Note that any EOB character that the listening  */
/*           might require must have been inserted in pBuff by the    */
/*           client routines. This routine set the client GPIB device */
/*           as listener, the GPIB server as talker, sends the message*/
/*           and clears all talkers and listeners in the bus.         */
/* Return:   error code                                               */
/*====================================================================*/
unsigned short HacGPIB_FtnTXDATA
  (
  unsigned short BusId,  /* BusId of client GPIB device               */
  unsigned short DevId,  /* DevId of client GPIB device               */
  unsigned short LU,     /* LU of client GPIB device                  */
  unsigned short TMOUT,  /* System clock ticks. Must be > 0           */
  unsigned int   NBYTE,  /* Number of bytes to be written             */
  void           *pBUFF  /* Pointer to buffer                         */
```

```
  )


/*===================================================================*/
/* Function: HacGPIB_FtnTXRAW                                        */
/* Purpose:  Raw send                                                */
/* Notes:    This routine is similar to HacGPIB_FtnTXDATA except that */
/*           no talker or listener is setup. It may be used to send    */
/*           commands (after setting the ATN line with HacGPIB_FtnCAC)*/
/*           but not data.                                           */
/* Return:   error code                                              */
/*===================================================================*/
unsigned short HacGPIB_FtnTXRAW
  (
  unsigned short BusId,  /* BusId of client GPIB device              */
  unsigned short DevId,  /* DevId of client GPIB device              */
  unsigned short LU,     /* LU of client GPIB device                 */
  unsigned short TMOUT,  /* System clock ticks. Must be > 0          */
  unsigned int   NBYTE,  /* Number of bytes to be written            */
  void           *pBUFF  /* Pointer to buffer                        */
  )
```

E IOC Hardware Configuration Example

```
/* HacCfg.dat file

/*=========================================================================
/* Define devices located on the CPU bus (BusId = 0)
/*=========================================================================
/*  BusId   Device    LU  NumSecBuses
      0    VMECHIP2   0       1
/*  DevHdlr               DevHdlr_Path
   HacNod_VmeChip2          ./core
/* PortNum   BusType   BusId (MUST be UNIQUE)
     0        VME              1
/* IHdlr  IHdlr_Path  IHdlr_Disp   IHdlr_DatFile
   NONE     NONE         KEEP          NONE



/*  BusId   Device    LU  NumSecBuses
      0      IPIC     0       1
/*  DevHdlr           DevHdlr_Path
   HacNod_IpIc          ./core
/* PortNum   BusType   BusId (MUST be UNIQUE)
     0        IPACK            50
/* IHdlr  IHdlr_Path  IHdlr_Disp   IHdlr_DatFile
   NONE     NONE         KEEP          NONE



/*=========================================================================
/*  Define VME devices located at BusId = 1
/*=========================================================================
/*  BusId    Device    LU   NumRegBanks   NumSecBuses
      1     HPE1313A   0        2             0
/*  RegBank  AddMod   MemBase     MemSize
      0      0x10    0xC000       0x003F
      1      0x20    0x00000000   0x0003FFFF
/*  IHdlr           IHdlr_Path    IHdlr_Disp       IHdlr_DataFile
 HacInit_hpe1313a     ./core        UNLOAD     ./config/HPE1313A:0.dat



/*  BusId    Device    LU   NumRegBanks   NumSecBuses
      1     HPE1313A   1        2             0
/*  RegBank  AddMod   MemBase     MemSize
      0      0x10    0xC040       0x003F
```

```
      1      0x20    0x00080000   0x0003FFFF
/*  IHdlr           IHdlr_Path    IHdlr_Disp       IHdlr_DataFile
 HacInit_hpe1313a     ./core        UNLOAD     ./config/HPE1313A:1.dat



/*  BusId    Device    LU   NumRegBanks   NumSecBuses
      1     VMIC1182   0        1             0
/*  RegBank  AddMod    MemBase       MemSize
      0      0x0020    0x00040000   0x00003FFF
/*  IHdlr        IHdlr_Path    IHdlr_Disp       IHdlr_DataFile
  HacInit_VME     ./core          KEEP      ./config/VMIC1182:0.dat



/*  BusId    Device    LU   NumRegBanks   NumSecBuses
      1     VMIC1182   1        1             0
/*  RegBank  AddMod    MemBase       MemSize
      0      0x0020    0x00044000   0x00003FFF
/*  IHdlr        IHdlr_Path    IHdlr_Disp       IHdlr_DataFile
  HacInit_VME     ./core          KEEP      ./config/VMIC1182:1.dat



/*  BusId    Device    LU   NumRegBanks   NumSecBuses
      1     VMIC2210   0        1             0
/*  RegBank  AddMod    MemBase      MemSize
      0      0x0010    0x0000       0x001F
/*  IHdlr        IHdlr_Path    IHdlr_Disp       IHdlr_DataFile
 HacInit_VME      ./core          KEEP      ./config/VMIC2210:0.dat



/*  BusId    Device    LU   NumRegBanks   NumSecBuses
      1     VMIC4140   0        1             0
/*  RegBank  AddMod    MemBase      MemSize
      0      0x0010    0x0300       0x007F
/*  IHdlr        IHdlr_Path    IHdlr_Disp       IHdlr_DataFile
 HacInit_VME      ./core          KEEP      ./config/VMIC4140:0.dat



/*  BusId    Device    LU   NumRegBanks   NumSecBuses
      1     VMIC4116   0        1             0
/*  RegBank  AddMod    MemBase      MemSize
      0      0x0010    0x0400       0x0012
```

```
/*  IHdlr        IHdlr_Path    IHdlr_Disp       IHdlr_DataFile
 HacInit_VME      ./core          KEEP      ./config/VMIC4116:0.dat



/*  BusId    Device     LU    NumRegBanks    NumSecBuses
      1      VMIC6016    1       2               4
/*  RegBank  AddMod   MemBase     MemSize
      0      0x0010  0x0200      0x00FF
      1      0x0020  0x000C0000  0x0003FFFF
/*     DevHdlr         DevHdlr_Path
   HacNod_Vmic6016       ./core
/* PortNum    BusType    BusId (Must be UNIQUE)
      0         RS_NA            18
      1         RS_NA            19
/*    2         RS_NA            20
/*    3         RS_NA            21
/*    4         RS_NA            22
/*    5         RS_NA            23
/*    6         RS_NA            25
/*    7         RS_NA            24
/*    8         RS_NA            26
/*    9         RS_NA            27
/*   10         RS_NA            28
     11         RS_NA            29
     12         RS_NA            30
/*   13         RS_NA            31
/*   14         RS_NA            32
/*   15         RS_NA            33
/*   IHdlr            IHdlr_Path    IHdlr_Disp        IHdlr_DataFile
 HacInit_Vmic6016      ./core         UNLOAD      ./config/VMIC6016:1.dat



/*=========================================================================
/*    Define RS_NA devices located at BusId = 18
/*=========================================================================
/*   BusId    Device      LU     NumSecBuses
      18    DYNAPOWER     0          0
/*   IHdlr    IHdlr_Path    IHdlr_Disp     IHdlr_DataFile
     NONE        NONE         UNLOAD            NONE



/*=========================================================================
/*    Define RS_NA devices located at BusId = 19
```

```
/*=========================================================================
/*    BusId    Device      LU     NumSecBuses
       19     DFISIK853     0          0
/*   IHdlr    IHdlr_Path    IHdlr_Disp     IHdlr_DataFile
     NONE        NONE         UNLOAD            NONE



/*=========================================================================
/*    Define RS_NA devices located at BusId = 29
/*=========================================================================
/*   BusId    Device      LU     NumSecBuses
      29      MP1000      0          0
/*   IHdlr    IHdlr_Path    IHdlr_Disp     IHdlr_DataFile
     NONE        NONE         UNLOAD            NONE



/*=========================================================================
/*    Define RS_NA devices located at BusId = 30
/*=========================================================================
/*   BusId    Device      LU     NumSecBuses
      30      MP1000      0          0
/*   IHdlr    IHdlr_Path    IHdlr_Disp     IHdlr_DataFile
     NONE        NONE         UNLOAD            NONE



/*=========================================================================
/*    Define IPACK devices located at BusId = 50
/*=========================================================================
/*  BusId   Device      LU  NumSecBuses
      50    gsIP488     0       1
/*  IP-slot   MemBase    MemSize (0x00 = none)     IRQ0     IRQ1
      0       0x00       0x00                      0x03     0x00
/*  DevHdlr           DevHdlr_Path
   HacNod_GsIp488         ./core
/* PortNum   BusType   BusId (MUST be UNIQUE)
     0        GPIB            51
/*    IHdlr           IHdlr_Path  IHdlr_Disp   IHdlr_DatFile
   HacInit_GsIp488     ./core       UNLOAD    ./config/gsIP488:0.dat



/*  BusId   Device      LU  NumSecBuses
      50    gsIP488     1       1
```

```
/*  IP-slot   MemBase    MemSize (0x00 = none)     IRQ0     IRQ1
      1       0x00       0x00                      0x03     0x00
/*    DevHdlr           DevHdlr_Path
   HacNod_GsIp488         ./core
/* PortNum   BusType   BusId (MUST be UNIQUE)
     0        GPIB            52
/*    IHdlr           IHdlr_Path  IHdlr_Disp   IHdlr_DatFile
   HacInit_GsIp488     ./core       UNLOAD    ./config/gsIP488:1.dat



/*=========================================================================
/*    Define GPIB devices located at BusId = 51
/*=========================================================================
/*   BusId    Device        LU      GPIBAdd    NumSecBuses
       51     HP3458A        0          1           0
/*      IHdlr              IHdlr_Path      IHdlr_Disp     IHdlr_DataFile
  HacInit_hp3458a_HRS        ./core           UNLOAD           NONE



/*   BusId    Device        LU      GPIBAdd    NumSecBuses
       51     HP3458A        1          2           0
/*      IHdlr              IHdlr_Path      IHdlr_Disp     IHdlr_DataFile
  HacInit_hp3458a_HRS        ./core           UNLOAD           NONE



/*   BusId    Device        LU      GPIBAdd    NumSecBuses
       51     HP3458A        2          3           0
/*      IHdlr              IHdlr_Path      IHdlr_Disp     IHdlr_DataFile
  HacInit_hp3458a_HRS        ./core           UNLOAD           NONE



/*   BusId    Device        LU       GPIBAdd   NumSecBuses
       51    LS450_GPIB      0          4           0
/*   IHdlr   IHdlr_Path      IHdlr_Disp     IHdlr_DataFile
     NONE     ./core           UNLOAD          NONE



/*   BusId    Device        LU       GPIBAdd   NumSecBuses
       51    LS450_GPIB      1          5           0
/*   IHdlr   IHdlr_Path      IHdlr_Disp     IHdlr_DataFile
     NONE     ./core           UNLOAD          NONE
```

```
/*   BusId    Device        LU       GPIBAdd   NumSecBuses
       51    LS450_GPIB      2          6           0
/*   IHdlr   IHdlr_Path      IHdlr_Disp     IHdlr_DataFile
     NONE     ./core            UNLOAD          NONE



/*   BusId    Device        LU       GPIBAdd   NumSecBuses
       51    LS450_GPIB      3          7           0
/*   IHdlr   IHdlr_Path      IHdlr_Disp     IHdlr_DataFile
     NONE     ./core            UNLOAD          NONE



/*=============================================================================
/*    Define GPIB devices located at BusId = 51
/*=============================================================================
/*   BusId    Device          LU     GPIBAdd     NumSecBuses
       52     PT2025_GPIB      0        10             0
/*  IHdlr            IHdlr_Path      IHdlr_Disp       IHdlr_DataFile
 HacInit_pt2025       ./core           UNLOAD      ./config/HacH_PT2025.dat
```

F HacNod_GsVipc610 Device Handler File

```
/*****************************************************************************
 *  HacNod_GsVipc610.c
 *
 *  Hall A Controls - hardware tree node driver:
 *    Device model : Green Spring VIPC610
 *    Device type  : VME slave 4-slot IPACK carrier.
 *    Requirements : Hall A Controls (HAC) hardware configuration architecture.
 *
 *  J. Gomez - Jefferson Lab - January.1997
 *****************************************************************************/
#include <Hac_Cfg.h>
#define GsVipc610_FtnCnt  2

/*****************************************************************************
 * HacNod_GsVipc610
 * Called by HacCfg.c during configuration of the HAC hardware tree.
 * Links IPACK server functions provided by the Green Spring VIPC610 to the IPACK bus
 * instance given by IPACKBusId.
 *****************************************************************************/
int HacNod_GsVipc610(int IPACKBusId)
  {
  int                   i,j;
  unsigned char         Fnd;
  SYM_TYPE              pType;
  unsigned short        (*pHdl)();
  struct HacBus_IPACK   *pIPACK;

/* Generic name of IPACK server functions supported by the Green Spring VIPC610  */
/* Names MUST match those defined in Hac_Cfg.h       */
  char FtnSrch[GsVipc610_FtnCnt][30] =
    {
    "HacIPACK_FtnIO", "HacIPACK_FtnIRQ"
    };

/* Private name of IPACK server functions supported by the Green Spring VIPC610 */
  char GsVipc610Ftn[GsVipc610_FtnCnt][30]=
    {
    "_GsVipc610_FtnIO", "_GsVipc610_FtnIRQ"
    };

  pIPACK = (struct HacBus_IPACK *)&(Hac_BusPtr[IPACKBusId]->Bus);
  for(i=0;i<GsVipc610_FtnCnt;i++)
    {
    Fnd = FALSE;
```

```
    for(j=0; j<=HacIPACK_LstFtn; j++)
      {
      if(strcmp(HacIPACK_FtnName[j],FtnSrch[i]) == 0)
        {
        Fnd = TRUE;
        if(symFindByName(sysSymTbl,GsVipc610Ftn[i], (void *)&pHdl, &pType) != OK)
          {
          logMsg("HacNod_GsVipc610 @ IPACK BusId %hu: Ftn %s not in sysSymTbl.\n",
             IPACKBusId,GsVipc610Ftn[i],0,0,0,0);
          return(-1);
          }
        else
          {
          pIPACK->pBusFtn[j] = pHdl;
          }
        }
      }
    if(!Fnd)
      {
      logMsg("HacNod_GsVipc610 @ IPACK BusId %hu: No such generic IPACK Ftn %s available.\n",
        IPACKBusId,FtnSrch[i],0,0,0,0);
      return(-1);
      }
    }
  return(0);
  };



/*********************************************************
 * GsVipc610_FtnIO
 *********************************************************/
unsigned short GsVipc610_FtnIO
  (
  unsigned short  BusId,
  unsigned short  DevId,
  unsigned short  LU,
  unsigned char   OP,
  unsigned char   SPCE,
  unsigned int    ADD,
  unsigned int    NTRY,
  unsigned char   DTYP,
  void            *pBUFF
  )
  {
  unsigned int         AbsAdd = 0, IPSlot;
```

```
  unsigned short       MyBusId, MyLU, ERR = 0;
  struct HacBus_IPACK  *pIPACK;
  struct HacBus_VME    *pVME;


  pIPACK = (struct HacBus_IPACK *)&(Hac_BusPtr[BusId]->Bus);
  IPSlot = pIPACK->IPSlot[DevId][LU];
  if(IPSlot > 3)
    {
/* Green Spring VIPC610 can handle up to 4 Industry Packs */
    logMsg("GsVipc610_FtnIO: IPSlot %u > 3.\n",IPSlot,0,0,0,0,0);
    return(HacErr_ARGOP);
    }

/* Determine BusId and LU of this Green Spring VIPC610.  */
  MyBusId = Hac_BusPtr[BusId]->depBusId;
  MyLU = Hac_BusPtr[BusId]->depLU;
  pVME = (struct HacBus_VME *)&(Hac_BusPtr[MyBusId]->Bus);

/* IPACK space being requested */
  switch(SPCE)
    {
    case(HacIPACK_IO):
      AbsAdd = pVME->MBase[gsVIPC610][MyLU][0] + 0x0100 * IPSlot + ADD;
      break;

    case(HacIPACK_ID):
      AbsAdd = pVME->MBase[gsVIPC610][MyLU][0] + 0x0100 * IPSlot + 0x0080 + ADD;
      break;

    case(HacIPACK_MEM):
      logMsg("GsVipc610_FtnIO @ BusId %hu, LU %hu: memory operations not implemented.\n",
         MyBusId,MyLU,0,0,0,0);
      return(HacErr_ARGOP);
      break;

    default:
      logMsg("GsVipc610_FtnIO @ BusId %hu, LU %hu: unknown SPCE code %x.\n",
         SPCE,0,0,0,0,0);
      return(HacErr_ARGOP);
      break;
    };

/* Check data type before requesting op */
  switch(DTYP)
    {
```

```
    case(HacDTyp_UShort):
    case(HacDTyp_UChar):
    case(HacDTyp_UInt):
    case(HacDTyp_Float):
      ERR = (*pVME->pBusFtn[HacVME_FtnIO])(MyBusId,gsVIPC610,MyLU,OP,
        pVME->AM[gsVIPC610][MyLU][0],AbsAdd,NTRY,DTYP,pBUFF);
      if(ERR != 0) logMsg("GsVipc610_FtnIO @ BusId %hu, LU %hu: err %x from VME server.\n",
                          MyBusId,MyLU,ERR,0,0,0);
      break;

    default:
      logMsg("GsVipc610_FtnIO: unknown DTYP code %x.\n",DTYP,0,0,0,0,0);
      return(HacErr_ARGOP);
      break;
    }

  return(ERR);
  };



/*********************************************************************
 * GsVipc610_FtnIRQ
 * The Green Spring VIPC610 sets the IRQ with jumpers.
 * No way to read back value from hardware. Return contents IRQ array
 * in structure HacBus_IPACK
 *********************************************************************/
unsigned int GsVipc610_FtnIRQ
  (
  unsigned short BusId,
  unsigned short DevId,
  unsigned short LU,
  unsigned char  OP,
  unsigned char  IREG,
  unsigned char  IRQ
  )
  {
  unsigned int         IPSlot;
  unsigned short       MyBusId,MyLU;
  struct HacBus_IPACK  *pIPACK;

  pIPACK = (struct HacBus_IPACK *)&(Hac_BusPtr[BusId]->Bus);
  IPSlot = pIPACK->IPSlot[DevId][LU];
  MyBusId = Hac_BusPtr[BusId]->depBusId;
  MyLU = Hac_BusPtr[BusId]->depLU;
  if(IPSlot > 3)
```

```
    {
 /* VIPC610 can handle up to 4 Industry Packs */
    logMsg("GsVipc610_FtnIRQ @ BusId %hu, LU %hu: IPSlot %u > 3.\n",
       MyBusId,MyLU,IPSlot,0,0,0);
    return(HacErr_ARGOP);
    }


/* Perform operation */
  switch(OP)
    {
    case(HacOp_RX):
      if(IREG == 0)
        {
        IRQ = pIPACK->IRQ[DevId][LU][0];
        }
      else
        {
        IRQ = pIPACK->IRQ[DevId][LU][1];
        }
      break;

    case(HacOp_TX):
      if(IRQ & 0x07)sysIntEnable(IRQ & 0x07);
      break;

    default:
      logMsg("GsVipc610_FtnIRQ @ BusId %hu, LU %hu: unknown OP code %x.\n",
         MyBusId,MyLU,OP,0,0,0);
      return(HacErr_ARGOP);
      break;
    }
  return(0);
  };
```

G HPE1313A Device Handler

```
/*****************************************************************************
 *  HacAi_hpe1313a.c
 *
 *  EPICS device handler:
 *     Record type  : analog input.
 *     Device model : HPE1313A from Hewlett-Packard (HP).
 *     Device type  : 32/64 channels high speed scanning ADC.
 *     Requirements : Hall A Controls (HAC) hardware configuration architecture.
 *
 *  J. Gomez - CEBAF - 20.June.1995
 *
 *  Modified to accomodate changes in the Hall A Controls (HAC) hardware tree
 *  architecture.
 *  J. Gomez - Jefferson Lab - November.1996
 *****************************************************************************/

#include <vxWorks.h>
#include <types.h>
#include <stdioLib.h>
#include <string.h>

#include <alarm.h>
#include <cvtTable.h>
#include <dbDefs.h>
#include <dbAccess.h>
#include        <recSup.h>
#include <devSup.h>
#include <link.h>
#include <dbScan.h>
#include <aiRecord.h>
#include    <Hac_Cfg.h>

static long init_record();
static long read_ai();

struct {
long number;
DEVSUPFUN report;
DEVSUPFUN init;
DEVSUPFUN init_record;
        DEVSUPFUN       get_ioint_info;
DEVSUPFUN read_ai;
DEVSUPFUN special_linconv;
} HacAi_hpe1313a={
```

```
6,
NULL,
NULL,
init_record,
NULL,
read_ai,
NULL};

static long init_record(struct aiRecord  *pai)
  {
  struct HacIo *pHacIo;

  switch (pai->inp.type)
    {
    case(HAC_IO):
      pai->udf = FALSE;
      break;

    default:
      recGblRecordError(S_db_badField,(void *)pai,
        "HacAi_hpe1313a(init_record) Illegal INP field");
      return(S_db_badField);
    }
  return(0);
  };


/***************************************************
 *  read_ai
 ***************************************************/
static long read_ai(struct aiRecord  *pai)
  {
  float             Data = 0.0;
  unsigned short    status;
  struct HacIo      *pHacIo;
  struct HacBus_VME *pVME;


  pHacIo = (struct HacIo *)&(pai->inp.value);
  pVME = (struct HacBus_VME *)&(Hac_BusPtr[pHacIo->BusId]->Bus);

/** Current value table located in reg bank 1, offset= 0, each entry 4 bytes **/
  status = (*pVME->pBusFtn[HacVME_FtnIO])(pHacIo->BusId,HPE1313A,pHacIo->LU,
     HacOp_RX,pVME->AM[HPE1313A][pHacIo->LU][1],
     pVME->MBase[HPE1313A][pHacIo->LU][1] + (pHacIo->CtlPt)*4,
     1,HacDTyp_Float,&Data);
```

```
  if(status != 0)
    {
    logMsg("HacAi_hpe1313a(read_ai) @ BusId %d, LU %d, Chann %d: status %x\n",
      pHacIo->BusId,pHacIo->LU,pHacIo->CtlPt,status,0,0);
    }

/** Assumes Auto-range, +/- 16 volts inp. max **/
  pai->val = pai->egul + (Data + 16.0)*(pai->eguf - pai->egul)/32.0;
  return(2);       /* do not convert - conversion taken care above */
  }
```