An Object-Oriented Approach to Low-Level Instrumentation Control and Support

J.B.Kowalkowski
Advanced Photon Source
Argonne National Laboratory

ABSTRACT

A common controls requirement is to be able to quickly add support for serial, GPIB, and various data acquisition devices. HiDEOS software provides a means to easily create and maintain low-level device drivers and higher-level control tasks. HiDEOS has an object-oriented task model which hides most operating system details, allowing the user to concentrate on operating the device. HiDEOS imposes a message passing system on the user for interprocess communication, so existing control systems can easily request information and receive results.

INTRODUCTION

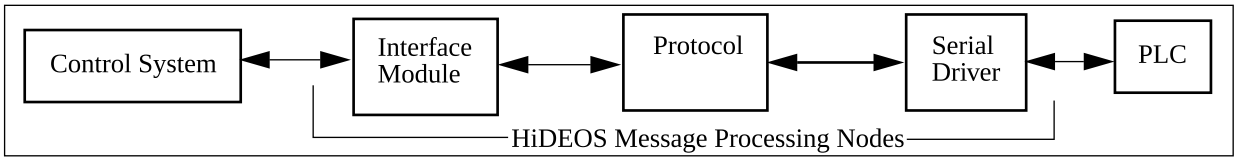
HiDEOS is a software package designed as an addition to the Advanced Photon Source (APS) control system. The initial implementation is on the Motorola MVME162 embedded controller, to operate the Industry Pack (IP) Bus and a set of IP modules. Instruments and sensors attached to the IP modules are also operated by the HiDEOS software package. HiDEOS combines parallel processing and object-oriented techniques to produce an operating system shell using the C++ language. The primary function of the operating system shell, in the context of the APS, is to allow a user to easily create, maintain, and integrate device drivers and algorithms into the control system. Many of the targeted instruments have serial or GPIB type communications that require a dialog or protocol.

HiDEOS utilizes object-oriented techniques to encapsulate operating system and hardware resources. All components of the operating system are implemented as objects. A user's program is actually viewed as a subclass of the system process. Users interact with the operating system by using methods of the parent class. HiDEOS is a message-driven system, a concept usually associated with parallel processing, where processes get scheduled by the presence of work in the form of a message sent to them. Messages can span CPU boundaries, allowing a problem to be distributed. The package includes a preemptive, multi-tasking kernel for use on a board with no operating system. The kernel is currently capable of running the MVME162 without an additional operating system.

A major goal is to view a running system as a collection of independent message processing nodes, with each node being responsible for a particular piece of equipment, running a protocol, or performing an algorithm. Nodes can find other nodes in the system by using a character string name, attach themselves, and send messages back and forth. A node can be resident on any one of a group of CPUs running HiDEOS. The current implementation requires CPUs to be on the same backplane. An application developer or processing node developer does not need to know which CPU a destination process will be running on. Calls to send and receive messages are the same for processes on a remote CPU or the same CPU. This methodology is illustrated with a real-world problem. A control system must interact with a PLC where the only way to communicate is a serial link. The problem can be logically broken into three sections (see Figure 1), each of which require different system knowledge: an interface procedure to communicate results to and get information from the control system, a protocol driver that can have a dialog with the PLC, and a serial link driver that can actually pump data down and retrieve data from the serial link. With HiDEOS, this problem can be viewed as three separate nodes, linked together with a message pipe. Developers of each module can now solve their own problem, and not worry about the mechanism that will transfer data from one node to the next or which CPU will be running the process.

The current implementation of HiDEOS basically consists of six major components. Together these components allow for basic operating system functionality including interprocess communications. The major components are message management, name service, task management, task dispatching, resource management, and utilities. A typical application running under HiDEOS resides within the task management component. The task is an important and fundamental unit of HiDEOS. The task can make use of board-level services such as tick timers and bus controllers using the resource management component or lock out interrupts using one of the utilities. The task can locate other tasks in a group of processors running HiDEOS using the name service component. A task can use the message passing facility to locate the destination task and send or receive understood message structures using the message management component.

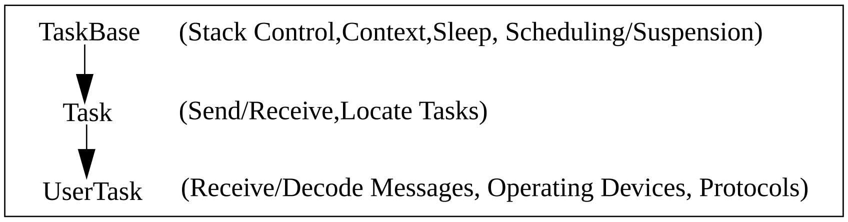
FIGURE 1. Processing Nodes



SYSTEM COMPONENTS

To create a task in HiDEOS, the user must derive a class from the TaskBase class. An actual running task is generated when an instance of the user's derived class is constructed (created). A typical HiDEOS user task has the derivation shown in the class structure diagram of Figure 2. The TaskBase class controls most low-level aspects of a process. A TaskBase instance, to a large degree, is in control of its own destiny. It determines when it should be scheduled to run and when it should be suspended. The TaskBase instance contains the stack and methods to access it. With a setup like this, a dispatcher need only maintain a handle to the TaskBase instance, and manipulate the task through its public interface.

FIGURE 2. Class Structure


TaskBase
(Stack Control,Context,Sleep, Scheduling/Suspension)
Task
(Send/Receive,Locate Tasks)
UserTask
(Receive/Decode Messages, Operating Devices, Protocols)

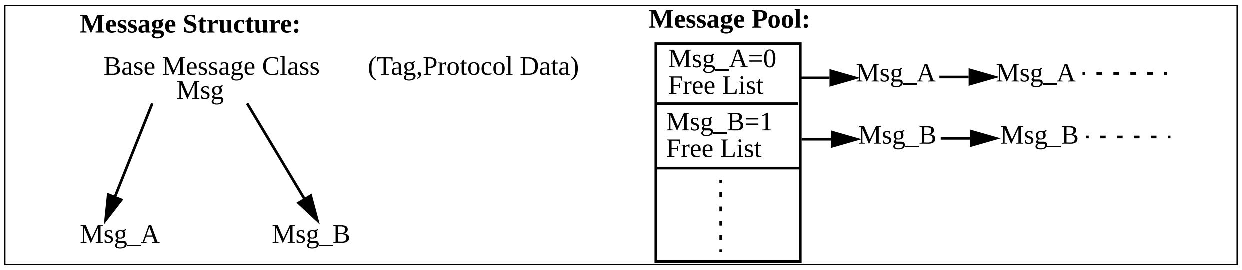
The Task class adds message-passing to TaskBase. The message system will be described later in this document. For this section, the important thing to remember is that the message is a basic unit in HiDEOS that carries information from task to task. The Task class essentially turns the task into an event-driven model. It adds public methods for other task instances to deliver messages to it and the ability for the task instance itself to send and receive messages. In addition, it defines an entry point for user code to be run. When a message is delivered to a task instance, the instance requests that it be scheduled. Some time later it starts running and calls a "Receive Message" function which has been defined by the user. The user is free to run any code in the "Receive Message" function; normally a message will imply a certain action to be carried out. Generally a single task is considered a device driver and is associated with a particular instrument.

Nodes sending and receiving messages from each other require a method to locate each other. The name server component fulfills this requirement. Each CPU in a complete HiDEOS system has one instance of the name server. Each task instance must be given a unique character string name. The name server registers the name with a handle to the task instance. The Task class can be used to automatically register the name of the instance. Any task running in the system can ask the name server for a handle to another task given its name.

Many users of HiDEOS will be interacting with instruments in their tasks. Most instruments hang off of the system bus, so the user must access the instrument by going through the bus controller, which can be thought of as a board-level resource. Board-level resources in HiDEOS are controlled through classes. A resource is any board-level service provided. Examples are the DRAM controller and the bus controller. When HiDEOS starts up, it creates one instance of a specific control class for each of the board-level services available, assuming the service-specific control class has been implemented. The DRAM controller is a good example of a class which operates a service on the motherboard. A user can get a handle to the DRAM instance and ask it for information about the memory, such as "Get Total Available DRAM." The bus controller works in a similar fashion. For the VME bus, the VME bus controller class instance can be asked to enable interrupt levels or open a memory mapping for the backplane.

A typical user application utilizes the message driven capabilities of HiDEOS to retrieve data from instruments on demand. A set of HiDEOS tasks can send and receive a predefined set of messages to each other. The message management system consists of a basic message class from which all user messages are derived. It also contains a message pool which manages user message buffers efficiently. HiDEOS tasks automatically know how to deal with a basic message, therefore any derived message can also be used in the system. Each message type in a complete system is required to be assigned a unique integer tag. The tag is used to generate and free message buffers, and also to identify the true message type in a running program. This is necessary because the interface to the user program is a function of the form "Receive Message" where message is the basic message. It is the job of the user code to check the tag and cast the basic message into its true type. HiDEOS includes utilities to automatically maintain message tags and the message pool. Tags are enumerated during the build process to match the class name with the word "Type" appended at the end to guarantee uniqueness. The user can easily identify messages in the system by checking the type code in the message against the tag enumerations. In Figure 3 below, two tags will be generated: Msg_A=0 and Msg_B=1. When the user program starts running because a message has arrived, the type can be checked against MSG_AType and Msg_BType so the true message can be recognized and the data extracted.

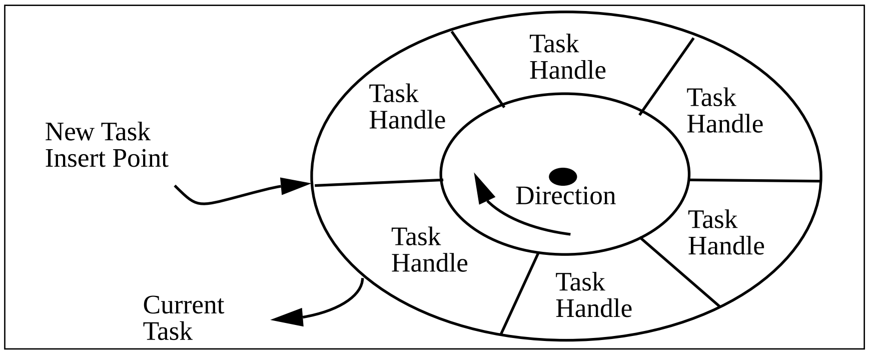
FIGURE 3. Messages



Message buffers must be freed and allocated using the special message pool. There is one instance of the message pool class per CPU running HiDEOS. Message buffers are never released back into the heap. Once allocated from the system heap, they remain in the system and are managed in free lists by the message pool. The message tags are assigned in ascending order, so reusing message buffers from the message pool is just an index into an array of free lists, one list per message type.

HiDEOS contains a dispatcher, shown in Figure 4. As an alternative, the system can easily be set up to use an existing dispatcher which is part of another operating system. An example of where the dispatcher is not used is vxWorks, since it has its own. The dispatcher is extremely simple, it currently does round robin scheduling with only one priority. The dispatcher is implemented as a class. One instance of this class exists for each CPU running HiDEOS. A circular linked list of runnable tasks is maintained by the dispatcher. Each time a time slice is complete, the next task on the linked list is restarted. The TaskBase class is used to add tasks to and remove tasks from the linked list.

FIGURE 4. Dispatcher



MESSAGE PROTOCOL AND DELIVERY SYSTEM

The Task class is an extremely important part of HiDEOS. With all the basic components of HiDEOS introduced, it is now possible to explain the actions carried out by this class for delivering messages. The Task class implements an input queue, and each message delivered to a task in HiDEOS is queued. A task delivers a message to another task by invoking a “Send” method. A task can wait for messages to appear in the input queue by using the “Wait For Any Message” method.

As explained above, the interface from the Task class to the user’s code is through a redefined method in the user’s derived class called “Receive Message.” The “Receive Message” method is always run in the task’s own process space or context, independent of the other tasks running in the system. The user is free to return from this function and should do so when the processing of the current message is complete. Returning from this function automatically implies a “Wait For Any Message” method invocation. At any point in the “Receive Message” function, a call can be made to “Wait For Any Message,” “Wait For Message Of This Type,” or “Wait For Message From A Specific Task.” Invoking any of these can cause the task to suspend itself, removing itself from the dispatching chain until the specified event occurs.

A “Send Message” always executes in the caller’s process space or context. The send actually invokes a public interface task of the intended receiver which will place the message in the receiver’s input queue. The act of doing so can cause the receiving task to schedule itself depending on its state. If the receiving task is already running, then there is no need to schedule. If it is waiting for any message to appear, then the task will schedule itself as part of the message queuing procedure. If the receive task is waiting for a specific message not of the type being queued, then the task will not be scheduled.

FIGURE 5. Process/Instance Space

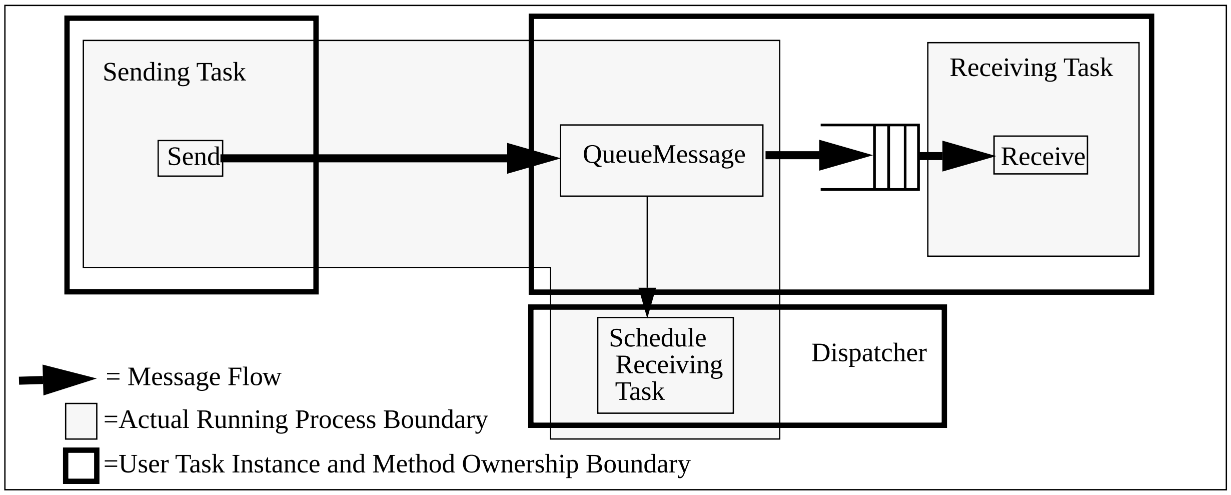
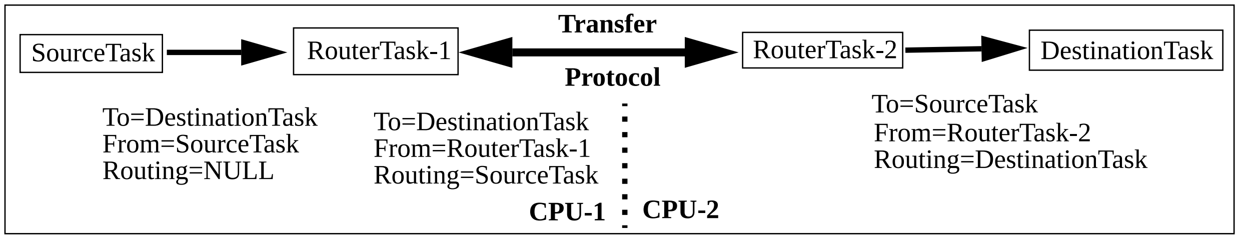


Figure 5 illustrates Task instance bounds and the process or context in which functions get executed. The public interface of a task contains a “QueueMessage” and “ReceiveMessage.” The QueueMessage is always executed in the sender’s context or process space, the Receive is always executed in the receiver’s context.

Messages in HiDEOS use a simple datagram-like protocol. The encapsulation of data within headers is done through subclassing. The Message base class contains the information needed to get messages from one task to another. The class contains the following information: handle to sender’s task, handle to receive’s task, a special routing task handle for remote communications, and a message type code (tag). The data length is not needed because the type code automatically implies the length of the message. In fact, the message pool can be used to discover the length in bytes of any of the messages given a message type code. As mentioned above, messages are always received in the base class form and must be cast into the correct derived type. A message contains enough information in the header for a user program to respond to the actual sender with results. Allowing two tasks on different CPUs to communicate using the same mechanism is more complex. The third handle in the message, the routing handle, allows for a simple way to transfer messages between two processors (CPUs) transparently. A router typically receives a message from a remote task destined for a local task. The first thing it does is put the from information into the routing handle field, and then place its own handle into the from field. The local task will receive the message from the remote task, do the required processing, and respond to the sender. The real sender is actually the handle in the routing field, but the routing task has fooled the receiving task into sending the message to it. The routing task takes the routing handle and places it back into the to field of the message and forwards it to the real destination.

The message router in HiDEOS is implemented as a HiDEOS task that makes use of the routing field and other flags in the Message base class. Figure 6 illustrates the manipulation of the Message fields by the routing tasks, which catch messages coming in from remote systems and massage the to/from fields to make the message appear to have a source on the local CPU. The routing task also takes on the special responsibility of forwarding name server queries to other CPUs running HiDEOS. Each CPU running HiDEOS has one message router running.

FIGURE 6. Message Routing



To summarize, each instance of the task class in HiDEOS is a separate process or thread. The task class has a public interface with two important functions in it: “Receive Message” and “Queue Message.” Other tasks in the system can invoke “Queue Message.” Doing so can cause the task in which the “Queue Message” function was invoked to be scheduled to run. The “Receive Message” function is actually user code that is called for each message in the task’s input queue. The “Receive Message” function is always invoked within the task instance that owns the input queue in which a message arrived. Messages always appear to be coming from a task on the same CPU, so a user can always reply to the sender of a message.

CONTROL SYSTEM INTERFACE

In order to quickly and easily integrate HiDEOS-based devices into an existing control system, an interface class is available for sending and receiving messages using general user-defined functions. The interface allows for non-HiDEOS programs to interact with HiDEOS programs and to be addressed with a handle just like HiDEOS tasks. It is important to be able to hook HiDEOS into an existing control system, using the existing control system constructs. The interface class allows this to be done. The interface class supplies methods for finding HiDEOS tasks by name and sending messages to them. There is a “Receive Message” call that can be made. This call blocks until a message has been delivered to the interface instance. The interface class provides a way for applications to be event driven. Upon construction of an interface instance, a user event function can be registered that will be invoked each time a messages appears which is destined for this instance. It is up to the user function to do something with the message; usually it will be queued and a second process will be informed that there is work to do. The interface class does not provide any queuing, so it is important that the user event function keep all messages that come in.

A HiDEOS process can receive a message from the interface class instance. The message appears as a message from another HiDEOS task; there is really no distinction between a message from the interface classes and from other tasks. An external process communicating with HiDEOS tasks using the interface class must still retrieve and free message buffers using the HiDEOS message pool component.

One important attribute of using this interface is the ability to create one control system interface for a class of instruments such as ADCs. The interface can specify a message format and protocol which it uses to get information from ADCs. HiDEOS ADC drivers can be created that conform to that protocol. One piece of interface code is capable of talking to many different ADCs, locating them by a character string name.

INITIAL IMPLEMENTATION

HiDEOS has been implemented as an embedded system using the C++ language and the above-outlined concepts. The C++ language was chosen because it is straightforward to understand the generated machine code. It has a simple memory allocation scheme similar to C, which can be used in a very efficient manner by letting HiDEOS manage blocks of commonly used memory. C++ generally produces code in a similar fashion to C, allowing high performance applications to be developed. Complete embedded executables can be produced which do not require any additional run-time libraries. Also, the compiler is available free from GNU.

Most components of HiDEOS were straightforward to implement using simple class hierarchies in C++. However, using C++ with its tight typecasting and lack of true dynamic binding posed several problems with the message passing portion. The C++ class instance creator "new" does not allow the user to dynamically (as the program is running) request a specific type of class instance to be constructed. In other words, the program cannot determine that a "LongMessage" is required and ask the "new" operator to create one. The only argument to the "new" operator is a hard-coded class. This is a problem in HiDEOS because messages require special buffer management so as to not fragment the memory by constantly allocating and freeing messages from the heap. The message pool handler discussed above is responsible for maintaining the message buffers. It is not possible in C++ to create a general "Get Message Buffer Of This Type" function that takes an argument of a message tag. HiDEOS gets around this problem by generating a table of classes and a tag for each class (the type tag). In addition, a case statement is generated (in C++). The case statement has one entry for each integer tag and code which knows how to create a class instance for the given tag (perform the "new" operation).

One outcome of the C++ implementation is a single downloadable HiDEOS image. A completely self-contained HiDEOS system is built for a particular application to run on a given CPU. This is good for embedded applications. What this means is that the downloadable executable will start running as soon as the CPU boots and must have information in it as to what services must be provided or what tasks must be running. The next section discusses the procedure for doing this.

CONSTRUCTING PROCESSES OR TASKS

Developing an application under HiDEOS is a three-step process: decide on a message and message interface that can be used to communicate with the instrument, develop a device driver for the instrument, and create or add instructions to an existing start-up procedure describing how to generate and name the new task.

The main purpose of a HiDEOS message-processing node is to operate a device on demand. Deciding on a message format is very important; it must convey as much pertinent information as possible in one transaction. Several general purpose messages are predefined by the system. Using these messages, if they match, is usually good practice because there are probably a set of other applications that want to communicate with the new task and know how to send and receive the general-purpose messages. Defining new messages usually implies that clients wanting the new services provided by the task must be modified to understand the new messages. An example of a general-purpose message is a "StringMessage." The message passes a generic string of bytes to the receiving task. This message is useful for most serial link instruments. A second is the "LongMessage," which transfers a simple long integer value along with status information. Figure 7 shows an extremely simple example of a user message definition written using C++ syntax. During the HiDEOS build process, the message will be discovered, an integer tag will be assigned, and code will be generated for the message pool to create it, given the tag. An enumerated name "UserMessageType" will also be generated which will be equivalent to the integer tag, and the enumeration will be placed into a global header file of all message tags.

FIGURE 7. UserMessage Definition

```
class UserMessage : public Message
{
public:
   long value;
}
```

The second step in developing an application is to write the user code or driver. An stated earlier, all user programs must be derived from the Task class, contain a constructor to initialize data or a device with which it will be communicating, and have defined a "Receive Message" function to be invoked automatically by the system to process messages. Figure 8 is a simple example of the definition of a HiDEOS task that will use the above-defined UserMessage. The purpose of this example task is to read a register in the address space and return the value back to the

FIGURE 8. UserTask Example

```
class UserTask:public Task
{
public:
   UserTask(char* name);
   void Receive(Message* msg);
private:
    long total_trans;
}
```

requester. The constructor for the UserTask just passes the name to the base class Task, and zeros the total transaction counter:

```
UserTask::UserTask(char* name):Task(name) { total_trans=0; }
```

The “Receive Message” function decodes the message type, casts it to the real derived type, sets the value field from a read of the hardware register, sends the message back to the source, and adds to the total transaction counter. As can be seen, the code fragment only recognizes the message type “UserMsgType”; other messages are passed to the base class Task for processing.

```
UserTask::Receive(Message* msg)
{
   switch(msg->type)
   {
   case UserMsgType: // real type
      UserMessage* m=(UserMsg*)msg;
      m->value=(0xfffffc00); //read register
      Send(msg->from,msg);
      total_transactions++;
      break;
   default: Task::Receive(msg);
   }
}
```

HiDEOS requires a set of instructions to set up processing nodes that are called upon during system initialization. The instructions are contained in a function and are actually a set of user-written C++ statements that can be viewed as a set of start-up rules. These start-up rules specify all the instruments and devices that will be controlled. When HiDEOS starts running, it calls the special user function to create a task instance for each service or device that will be available. Each task instance sits idle after initializing, waiting for incoming messages requesting data transfers from or to the device it owns. The start-up function has the responsibility of creating task instances for devices and giving them names that will be registered in the system. An additional requirement is to hook or link tasks that will be working together, such as a high-level protocol and a serial link task. A typical start-up function contains a series of calls to create and name task instances and a set of statements that connect tasks together is required. The current implementation requires a C++ system initialize hook function. In the future, a parsed rule file will be used to configure a HiDEOS system. This initialization function actually gets called before the dispatcher is started, so processing of messages has not yet begun. Figure 9 shows an example of a function that starts up the task with a name. Other tasks in the system can locate this task by using the name “my-task”.

FIGURE 9. User Initialization Function

```
InitializationFunction()
{
   UserTask* ut=new UserTask(“my_task”);
}
```

CONCLUSION

This paper is designed to be an overview of the underlying concepts of HiDEOS and the methodology used to develop HiDEOS applications. There are many capabilities and details in the real implementation not covered here. For a more complete discussion, including a user’s guide, see [1]. Another paper describing the integration of HiD-

EOS into the EPICS control system at the Advanced Photon Source [2] includes a list of supported hardware and several applications using the system, along with extensions required to operate a real HiDEOS system.

ACKNOWLEDGEMENTS

The HiDEOS analysis and design review assistance from John Winans of the Advanced Photon Source at Argonne National Laboratory helped tremendously in producing a solid, coherent product. This work was supported in part by the U.S. Department of Energy, Office of Basic Energy Sciences, under Contract No. W-31-109-ENG-38.

REFERENCES

[1] J.B. Kowalkowski; “Home Of HiDEOS,” World Wide Web URL: http://www.aps.anl.gov/asd/controls/hideos/intro.html.

[2] J.B. Kowalkowski; “A Cost-Effective Way to Operate Instrumentation Using the Motorola MVME162 Industry Pack Bus and HiDEOS,” these proceedings.