EPICS DEVICE/DRIVER SUPPORT MODULES FOR NETWORK-BASED INTELLIGENT CONTROLLERS

J. Odagiri, J. Chiba, K. Furukawa, N. Kamikubota, T. Katoh, H. Nakagawa
and N. Yamamoto, KEK, Tsukuba, Japan
M. Komiyama, I. Yokoyama, RIKEN, Wako, Japan
H. Song, IHEP, Beijing, China,
Y. Yamamoto, Mitsubishi Electric Co., Kobe, Japan
H. Miyaji, H. Satoh and M. Sugimoto, MCR, Kobe, Japan

Abstract

Modern accelerator control systems adopt a wide variety of intelligent controllers to interface front-end computers with active components of the accelerator. Recently, some of the intelligent controllers come equipped with an Ethernet interface, giving an opportunity to use a network as a kind of field busses.

However, developing device drivers for the use of such controllers is a complex and time-consuming process in most cases. It is essential to have the drivers share as much of their codes as possible while leaving flexibility to adapt to various proprietary communication protocols.

We developed a set of device/driver support modules for the Experimental Physics and Industrial Control System (EPICS) to support several network-based intelligent controllers, such as Program-able Logic Controllers (PLCs) and Device Interface Modules (DIMs).

The software consists of a common driver module, a common device support module, and a device-specific module for each of the devices to be supported. The common driver module encapsulates the details of the programming for communication over the network. The common device support module encapsulates the details of the framework of EPICS for asynchronous I/O transactions.

We decided to implement the common modules in such a manner that the device specific modules can be implemented by using only standard UNIX libraries. As a result, it turned out that the developed device-specific modules were compliant with both EPICS 3.13 on VxWorks and 3.14 on Linux.

INTRODUCTION

It is becoming more common for intelligent device controllers, such as PLCs and DIMs, to have an Ethernet interface. The accelerator control systems of J-PARC at JAERI/KEK [1] and RARF/RIBF at RIKEN [2, 3] will use devices of this kind with EPICS intensively for stability of the TCP/UDP standard protocol widely used in commercial fields and flexibility in a configuration of the devices. A smaller control system of the prototype FFAG accelerator at KEK will also adopt the same scheme [4].

Table 1 summarizes the PLCs and DIMs to be used in the control systems.

Table 1: Devices Supported

Device	Type	Make	Protocol
FA-M3	PLC	Yokogawa	TCP/UDP
MELSEC-Q	PLC	Mitsubishi	TCP/UDP
CVM1/CS1	PLC	Omron	TCP/UDP
EMB-LAN100	DIM	Custom	UDP
N-DIM	DIM	Custom	TCP/UDP
BPMC	DIM	Custom	TCP

All of the devices have one or more communication server, to which an Input Output Controller (IOC) can send a command and obtain a response to read/write the internal memory of the devices. In addition to the command/response-based transaction, some of the devices can send spontaneous messages in order to notify the IOC of an event, or to report the results of measurements that had been requested by the IOC in advance. EPICS device/driver support modules must meet the requirements.

DESIGN OF THE MODULES

This section describes the design and technical details of the device/driver support modules. We focus on the usual command/response-based transaction hereafter through subsection 2.6. The last subsection, 2.7, discusses the support of the device-driven transactions.

Basic Design

The key observation in designing the device/driver support modules is that what to send/receive depends on each of the devices, whereas how to send/receive, as far as the six devices listed in table 1 concerns, does not. The part of the code that handles the latter can be factored out to be common facilities shared by all of the device-specific modules. The architecture of the software is illustrated as the middle block in Fig. 1.

On the other hand, communication protocols at the application layer on the socket interface do not have any standard. It is hard to foresee all of possible future protocols to be supported in this framework. As a result, it is hard to create a common driver module that is generic enough for device-specific modules to have just a simple command/response table.

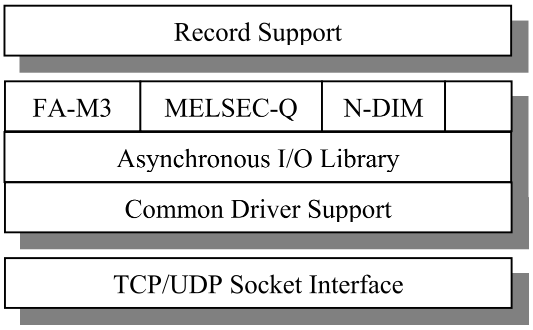

Figure 1: Architecture of software.

We decided to implement the most basic common driver, which handles just sending a command and getting a response, with a method to allow us to implement beyond the basics at a higher layer in the device-specific modules. This design strategy made it possible to keep the code of the common driver simple enough while leaving much flexibility to adapt to various proprietary protocols.

Another design goal was to provide appropriate interfaces with the device-specific modules so that they can be implemented by using only standard UNIX libraries. This should be possible in most cases, since constructing and parsing messages are just a simple operation on a series of bytes of data, regardless of being either ASCI codes or binary ones.

Device-Specific Modules

The essential part of device-specific modules is about constructing commands to be sent to a remote device, and parsing the response messages, in addition to getting address information by parsing the link field of the database records. Every device-specific module must implement the following three functions:

- **Link Field Parser** parses the link field of the run-time database records to get address information of a specified channel upon initialisation of the IOC program.
- **Command Constructor** constructs a command to be sent to a remote device in reference to a specified address, and transfer data into the command message from the record buffer if the operation is writing.
- **Response Parser** parses the contents of the response message upon its arrival, and transfers data included in the message to the record buffer if the operation is reading.

Since device-specific modules interface with record support modules, they include the member functions of Device Support Entry Tables (DSETs). They can be implemented just by wrapping functions supplied by the Asynchronous I/O Library, as explained in the next subsection.

Asynchronous I/O Library Module

The Asynchronous I/O Library module supplies the upper device-specific layer with a set of Application Program Interfaces (APIs) that encapsulates technical details of an asynchronous device support of EPICS. Two main functions, a generic initialisation function and a generic read/write function are provided, which can be wrapped to be member functions of a DSET of a specific record/device type.

The initialisation function invokes *Link Field Parser* to identify a remote device and the address to read/write. If a Message Passing Facility (MPF), which is described in the next subsection, has not yet been created for the remote device, it creates one. The pointers to the three functions mentioned in the previous subsection are passed to the MPF so that they are invoked at appropriate steps of an asynchronous I/O. It also initialises a structure required to call back the record upon its completion stage.

A read/write function is to be invoked twice in an asynchronous I/O, at the initiation and completion stages. At the initialisation stage, it invokes *Command Constructor* to form a message, and puts the I/O request on a queue in an MPF. It then notifies the MPF of the event. At the completion stage, the read/write function has nothing to do, since *Response Parser* transfers the data before the function is invoked.

Common Driver Support Module

The Common Driver Support module creates an MPF, for each communication server running on a remote device. All of the I/O requests heading over to the same remote server are to be lined in a single request queue of the MPF waiting for its turn. An MPF comprises the following threads of processing:

- **Send Task** gets an I/O request from the queue and invokes *Command Constructor* to put the message bytes into an intermediate buffer and to send it to the remote device, then blocks until *Receive Task* gets a response message.
- **Receive Task** waits for the response message to arrive and invokes *Response Parser* to parse the message, and then makes *Send Task* being blocked go to the next round. It finally issues a call-back request to complete the asynchronous I/O.
- **Timeout Handler** cancels an I/O transaction when a watchdog timer has expired with a specified timeout. When this occurs, *Timeout Handler* issues a call-back on behalf of *Receive Task* with an ERROR.

The possible race between *Receive Task* and *Timeout Handler* can be managed by using the difference in priority of the execution. Measures to avoid any misplacement of response messages to an irrelevant record have also been carefully implemented.

Chain Transaction

Response Parser, a function in device-specific modules, can return NOT_DONE as a return value to have *Send Task* in an MPF invoke *Command Constructor* once again. If NOT_DONE is returned, *Receive Task* does not issue a call-back request to complete the I/O. Instead, the task puts another I/O request on the request queue so that next transaction continues.

An IOC communicating with a WE7000 [5], although it is not supported in this framework at this moment, expects instantaneous ACK to be returned just after a command to the WE7000 has been sent. In addition, an OK must be sent back to the WE7000 in response to the ACK. Handshake sequences of this kind can be implemented as a state machine in *Command Constructor* and *Response Parser* in the device-specific modules, a state machine that generates a series of chain transaction.

Another application of the chain transaction, which is actually in use, is extending the maximum number of data transferred to/from a device. Most PLCs can transfer up to only some hundreds of words by a single transaction. The device-specific modules of PLCs use a chain of transactions to transfer huge data over the limit by splitting it into multiple pieces, and then transfer them one-by-one.

The chain transaction works for more application-sided sequences of a procedure as well. The “Indirect Write Rule” adopted by the J-PARC control system illustrates how it works. The rule, for security reasons, forces application programs to write a value of data together with its destination address into one of the “communication ports” defined in a memory area of the PLC, instead of writing the data directly at destination address. A ladder program running on the PLC checks the data written in the port and transfers it to the destination address only if the value is acceptable. The rule requires, for one write operation, the following three steps of transactions: 1) Test a busy flag of the port, 2) Write data and its address into the port, and 3) Set the busy flag of the port. The procedure was implemented in the device-specific module as a form of simple state machine to cause the chain of transactions.

Device Driven Transaction

Four devices (FA-M3, EMB-LAN100 [5], N-DIM [2] and BPMC) listed in table 1 have a function of sending event notification messages, or messages that report results of an on-going measurement. The messages may or may not require a response to a remote device. While an IOC should have issued the request of the notification or report in advance, it is the remote device that decides when the messages are dispatched.

To handle this type of device-driven transactions, *Command Constructor* and *Response Parser* in the device-specific modules are replaced with *Response Constructor*, and *Event Parser* respectively. The MPF for device-driven transactions creates a simple server task to get the messages and send back a response message, if any, to the remote device.

PORTING TO R3.14 ON LINUX

While the first version of the device/driver support are developed on R3.13 of EPICS on VxWorks, a recent trend toward Linux and multi-platform compliance of the latest version of R3.14, encouraged us to port the device/driver modules onto it. Using Linux as a platform has the following advantages:

- **Cost-effective**: A single PC can be used for both the target and the development environment, making the initial and running cost much lower.
- **Physically Portable**: The statement above applies to a laptop PC to be used for a test on premises of a site where a controlled device is being developed.
- **Quick start-up**: Getting an IOC program started takes much less time than it takes when the program is booted through a network over to a remote target, making the debugging cycle much faster.

Porting of the common driver was straightforward. We only had to map APIs of VxWorks onto corresponding ones defined in the Operating System Independent (OSI) - libraries of EPICS. As for the device-specific modules, the work was next to nothing.

CONCLUSION

A set of EPICS device/driver support modules of intelligent device controllers was developed as a family in the same framework for communication through Ethernet connections. The adoption of a modular and consolidated design allowed us to implement device-specific modules compliant with both versions of EPICS R3.13 and R3.14 for devices of six deferent types, while supporting a wide variety of features of the devices.

REFERENCES

[1] K. Furukawa et. al., “Implementation of EPICS Device Support for Network-Based Controllers, ICALEPCS 2001, San Jose, Nov. 27-30, 2001.

[2] M. Komiyama, et. al., “Current Status of the Control System for the RIKEN Accelerator Research Facility”, This conference.

[3] T. Tanabe et. al., “Current Status of the Control System Development at RIKEN RI-Beam Factory”, This conference.

[4] Y. Yuasa, to be published in Proc. of the 14th Symposium of Accelerator Science and Technology, Tsukuba, Nov. 11-13, 2003 (in Japanese).

[5] K. Furukawa et. al., “A Network-based Intelligent Controller for J-Parc”, This conference.