COMMON DEVICE ACCESS FOR ACCELERATOR CONTROLS

Reinhard Bacher, Philip Duval and Honggong Wu, DESY MST, Hamburg, Germany

Abstract

An accelerator control system must in general support a variety of hardware devices, and more often than not, a variety of different field bus types. It is very tedious to write control software dealing with the different bus types at the driver level without an overlying systematic interface. The problem of supplying the control system engineer with a common device interface, regardless of field bus, has been tackled with great success by control systems such as EPICS [1], COACK [2], and DOOCS [3], but has as yet received little attention in TINE [4].

We report here on the first efforts to provide a multi-language, multi-platform, generic device interface for the TINE control system. Specifically, the goal is to offer via the TINE client interface a generic view of such disparate busses as USB, CAN, RS232, GPIB, PROFI, VME and SEDAC. The developer must be able to access all of the features of the underlying bus if desired but at the same time be unaware of the details. With this scheme, fully functional and extensible TINE servers can be generated using the TINE server wizard; simple generic servers can offer simple device control via database configuration; and by using the TINE client interface, the server redirection feature allows an engineer to develop a server from a remote location.

INTRODUCTION

The access and control of hardware devices is typically achieved via fundamental 'Get' and 'Set' operations, where a 'Get' is used to acquire data or status information from the bus and a 'Set' is used to change control modes or download data to the hardware. The details behind these simple operations are in general quite varied for disparate bus types. Some bus drivers offer single-channel read and write calls while others utilize duplex channels for read and write. The bus data format can also be different. For instance, RS232 deals with character string data, whereas SEDAC deals with short integers. Hence the interfaces to these 'Get' and 'Set' operations are generally just as varied as the details behind them.

Although commercial hardware bus cards generally come with a driver, it frequently imposes an undue burden on the software developer to master the details of 'yet another' API in order to communicate with his hardware.

Many control systems already address this problem by providing a common device interface which hides all hardware bus specific features. To date, however, this has never been a burning issue with the TINE control system, as independent, decoupled, driver interfaces have traditionally been used for the front-end bus in question.

In particular, as the in-house SEDAC bus dominated the hardware in use by HERA as well as its pre- at home accelerators, most software developers were usually quite with the available SEDAC API. Additional hardware using GPIB, CAN, RS232, or VME was simply dealt with on an individual basis. However, with the advent of the PETRAIII project (along with on-going work with VUV-FEL) the advantages of providing a common device interface to the software developers has become manifest. Namely, SEDAC will no longer be the dominate field bus, and it will be most desirous to be able to seamlessly upgrade the hardware busses in use by the front-end servers without involving major re-writes.

CDI

General philosophy

The basic criteria which the common device interface (CDI) package must fulfill include the following. It must present a device access API to the developer which hides all details of the bus being used. (Indeed, the developer should be completely unaware of which bus the hardware resides on). It must be platform independent. And it must be language neutral as far as possible. That is, CDI calls in C should resemble analogous calls in Java or Visual Basic as much as possible.

The CDI package we make use of has a three-tier architecture. The lowest layer makes use of the direct interface to the bus drivers for a given hardware bus. At this level, the bus-specific 'Get' and 'Set' data operations are mapped into CDI acceptable plugs. The middle tier accepts these plugs, where bus initializers, accessors, terminators, and other configuration routines are provided. The middle tier also maintains a thread engine where all IO is managed. Finally, it offers synchronous and asynchronous device access routines as well as configuration routines to the upper tier. The upper tier maps the raw CDI interface to the TINE control system, where the TINE client-side API calls can then be used to access the local hardware. This is shown schematically below in figure 1.

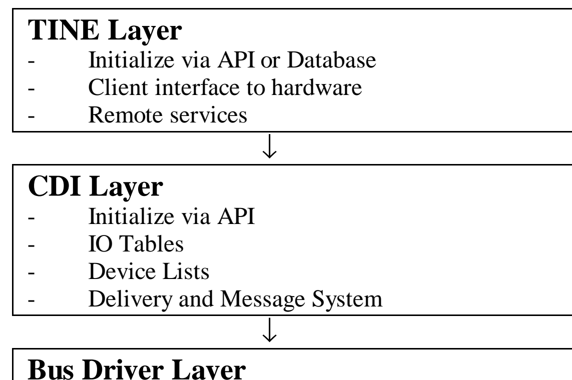

Figure 1: Schematic of CDI architecture.

In general, a TINE device server will have three avenues for accessing its hardware. The designated bus driver is of course available. The raw CDI API (second tier) is likewise available. Finally, the developer can make use of the TINE client API to access the hardware, in which case all data acquisition, be it from another server or the local hardware bus, can be managed with the same API.

Device Access Interface to bus driver

For interfacing a bus driver with the CDI a set of standard interface calls must be registered in the lowest layer. These are basically the ‘Get’, ‘Set’, ‘Open’ and ‘Close’ operations which define the bus behavior. For duplex bus drivers a callback function is used for receiving data. The bus specific parameters needed are passed directly from the middle layer of the CDI. To support a new bus interface card one essentially needs to provide the mapping of the bus driver calls onto the CDI plug. In this layer, data are passed directly back and forth without any manipulation.

Common Device Interface - CDI

The CDI API includes the following calls:

- cdiOpen() initializes a bus line with bus specific parameters. In the case of RS232, for example, the basic communication parameters, such as baud rate, data word size and so on need to be provided, whereas for CAN, the data transmission speed, CAN message protocol used, etc. need to be provided.
- cdiClose() terminates the communication for the bus line.
- cdiRegisterDevice() registers a connected device.
- cdiRequest() transmits data to or from the bus for a given set of devices. Besides information describing the input and output data sets this routine accepts data masks and comparison patterns which together determine under what circumstances a callback event is fired, if the call is asynchronous. A call to CdiRequest() can also optionally specify any additional bus parameters which might be relevant for the ensuing device access. Furthermore, the nature of the bus access is specified in the call. This can be simple ‘READ’ and ‘WRITE’ operations as well as ‘pseudo-atomic’ ‘READ-WRITE’ or ‘WRITE-READ’ operations.
- cdiRegisterRedirectedDevice() registers a device connected to a remote server.

The CDI API can be used to access the hardware, in which case initialization calls such as cdiOpen() and cdiRegisterDevice() must supply the hardware bus and hardware address. After a device name has been bound to a device, calls to cdiRequest() no longer need to pass bus specific information. Such registration API calls can be avoided by supplying a CDI device database description (see below).

TINE Interface to CDI

The CDI calls are easily mapped to the TINE client API calls. Thus the synchronous and asynchronous data acquisition calls ExecLink() and AttachLink() [4] can be used in the same way when accessing the local hardware as when accessing information from a remote server.

Furthermore, automatic format conversion can take place at this level. If the developer wants an array of 10 float values from the hardware, the TINE layer will see that he gets 10 float values, regardless if the bus is delivering character strings or integers. Of course the string parsing instructions must be known at this level!

In addition, a TINE Front End Controller (FEC) which uses CDI will automatically offer remote access to its hardware and to its hardware configuration by running a CDI server as a separate TINE equipment module. As a consequence it is a simple matter to use the TINE server redirection mechanism to redirect ‘local’ hardware access to a remote server, a feature which is very useful when commissioning and developing a front end server. This is depicted below in figure 2.

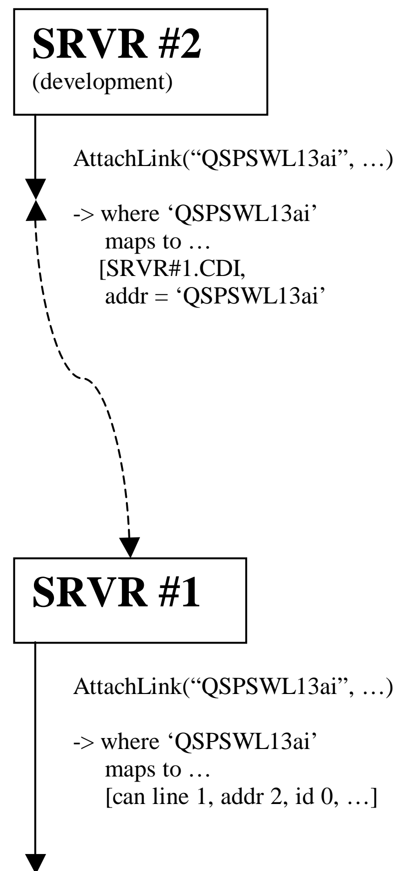

Figure 2: Schematic of data flow for hardware redirection from server #2 to server #1.

In the above figure, we see that the local CDI database (see section 2.6 below) for Server #2 instructs any references to a device named "QSPSWL13ai" to be forwarded to Server #1. Whereas the local database for Server #1 instructs the same reference to be forwarded to the CDI layer, which will know that the device in question is located on a CAN bus (line 1), with address 2, and so on.

In this manner, the server code and algorithms can be developed on a remote server and moved when ready to the production server, which might even have another operating system.

CDI Diagnostics

The CDI layer also maintains a number of statistics counters relating to bus errors, timeouts, total number of messages, bus load, etc. These counters can be read out at any time either locally via a set of CDI API calls or remotely by accessing the TINE CDI server module.

Database Configuration of CDI

All CDI initialization and configuration is available via the API calls alluded to in section 2.3 above. In many cases however, it is desirable to hide all details concerning the hardware bus and device addressing from the server code. To this end one can make use of the basic CDI configuration database, which contains all address information and bus parameters for the CDI controllable devices. If this database is present it will be scanned by CDI at boot time. In such a case there is no need to make any CDI calls to 'Open' or 'Close' a line, or to register device names. The server code need only make the data acquisition calls ExecLink() and/or AttachLink() and supply the callback functions and logic applicable to the device access.

We reiterate: The CDI configuration database is a simple database which matches a device name to a hardware address and set of bus parameters. It does not specify how or how often a hardware device is to be accessed. Nor does it specify what actions are to be taken following data access or hardware events.

It is of course tempting to take this a step further, where these kinds of things (or at least some common subset of them) are indeed specified. We are intentionally avoiding these issues at this juncture. Largely, this sort of second-level configuration will be dealt with via the TINE server wizard [5], which will generate code based on the developer's wishes. We tend to favor this approach since there is frequently a large amount of 'tweaking' which is apt to take place by the developer when he is commissioning his server. Once the necessary hardware IO goes beyond the simplest 'read', 'write', or 'poll' operations, the server developer is often confronted with a wide variety of adjustable parameters involving everything from timing and delays to various ways of combining, filtering or massaging the arriving bus data. Although making adjustments in a database is in principle easier than making software changes (at least the debug turn-around time is shorter!), one very often reaches the point where 'you've got to go in and code something anyway'.

FUTURE DEVELOPMENTS

Currently, CDI is still in its infancy. The three field busses SEDAC, CAN, and RS232 have been tested extensively on Linux, and to some extent on Windows. Another popular bus in use by front-end hardware at DESY is GPIB, which will soon be incorporated into CDI. Likewise, the VME bus will also be added to the CDI repertoire, especially as the package begins being tested on VxWorks.

As the second-tier CDI library is written in C, it is easily ported to other platforms. Of course the hardware drivers themselves must be available on the platform in question in order to use the CDI. The device management in CDI depends on thread and thread-synchronization operations which must also be made available to the CDI library. The CDI thread operations essentially wrap POSIX thread operations for UNIX systems and Windows thread operations for Windows systems. Porting CDI to VxWorks will involve wrapping the VxWorks task operations. This needs to be tested.

Using CDI in Java will involve providing a JNI interface to the CDI library which must be available on the platform in question. Likewise this needs to be tested.

CONCLUSION

The current version of TINE CDI in use at DESY serves as an existence proof of the viability of offering the TINE client API to server developers for accessing server hardware. It offers remarkable simplicity to the developer for 'simple' operations, yet at the same time allows the developer to access the full range of the hardware capabilities of the bus if necessary.

Future projects at DESY, such as PETRA III, will make extensive use of TINE CDI. That is planned. Even at the present time however, new elements in the existing accelerator control are using a wider variety of field busses. In other words: "It's not all SEDAC!" Thus it is already an attractive idea to regain the degree of uniformity that CDI offers at the device server level. So there is a rich test bed where we can hone and refine the current CDI library for its use in the next generation of DESY accelerators.

REFERENCES

[1] http://www.aps.anl.gov/epics
[2] M. Mutoh, et al., "Development of Generalized Device Layer for the COACK System", PCaPAC 2002, Frascati, Italy
[3] http://tesla.desy.de/doocs/doocs.html
[4] http://desyntwww.desy.de
[5] P.Duval and V.Yarygin, "The Use of Wizards in Creating Control Applications", ICALEPCS 2001, San Jose, CA.