Using the Common Device Interface in TINE

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

Abstract

An accelerator control system must in general support a variety of hardware devices and field busses. The Common Device Interface (CDI) is designed to provide the control system engineer with an easy-to-use-and-understand interface for accessing data from the hardware devices, independent from the underlying field bus. The concept behind CDI was initially presented in PCaPAC 2005[1]. Since that time CDI has undergone numerous refinements which render the writing of a plug-and-play CDI bus plug a straightforward procedure on the one hand, and allow the server developer an intuitive interface for acquiring or setting hardware data either synchronously or asynchronously and from either single or multiple devices.

We note here some of the differences in philosophy between CDI and asynDriver [2] (used by EPICS) and DOOCS[3] device drivers, and we report on the first operational results using CDI on several different TINE[4] platforms (including, Windows, Linux, and Java). Although CDI can be trivially hooked into a TINE server and offers the TINE client interface to the hardware, it is not tightly bound to TINE and could in principle be used independently.

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 hardware 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. Some bus drivers are single master, others are multi-master. 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.

Prior to CDI, the TINE control system did not have an explicit hardware layer for device servers. For TINE device servers which are EPICS [4] IOCs running Epics2Tine [5] then the device drivers are automatically EPICS drivers (most likely aSyn drivers). For TINE device servers which are DOOCS [6] servers, then the device drivers are DOOCS drivers (which follow the UNIX device server model). By and large the vast majority of TINE device servers at DESY are native TINE servers and follow a "do it yourself" ansatz. Namely, one uses the drivers "which come with the hardware" or (more likely) one uses the in-house SEDAC drivers. This has always proved a viable approach as the bulk of the hardware for HERA and its pre-accelerators is SEDAC. That will change with the advent of PETRA III and is already no longer true with FLASH. There will be considerably more front end hardware using CANOpen devices and TwinCat devices, along with legacy SEDAC, GPIB, rs232, vme, etc.

Application programmers will then either have to become familiar with a bus interface API for each of these bus types or rely on CDI to provide a common interface for all.

The Common Device Interface (CDI) first presented in PCaPAC 2005 [1] has now reached a new level of maturity and is in operation on a couple of test stands at DESY.

CDI API

As all TINE developers are familiar with the TINE client API for accessing data from device servers, CDI strives to leverage this knowledge by offering the same API for accessing data from the hardware bus. In this case a device server running on a Front End Computer (FEC) is a client to its attached hardware.

CDI itself offers a CDI-native API which is TINE-similar. However, by and large developers will want to make use of precisely the same TINE client API calls as used when accessing data from any other end-point in the control system.

It is worth a bit of time to review the TINE client API, as it does not follow the (seemingly ubiquitous) 'get(), set(), and monitor()' APIs so loved by other control systems. Instead TINE deals with 'calls' in the sense of Remote Procedure Calls or Remote Method Invocation, which are passed via data 'links'. A data link can be either synchronous or asynchronous and calls a TINE property at a TINE endpoint. A TINE endpoint is in turn determined by a namespace consisting of device context, device server, and device name. Schematically, a call will attempt to access:

/<context>/<server>/<device> [<property>].

and will optionally send an input data object to the target and/or request an output data object to be returned. A TINE property should rather be thought of as a 'method'. If data is both sent to and received from the target, this exchange of data objects occurs atomically. Note also the requested data access (read or write) is separated from the data objects. That is, a 'read' call which needs to send data to the target is still a 'read' call!

The synchronous API call is ExecLink() (short for 'execute link') and has the following basic prototype:

ExecLink*(name, property, dout, din, access, timeout)*

The corresponding asynchronous call AttachLink() has additional monitoring parameters which supply callback information as well as the monitoring 'mode', which can contain a wide variety of monitoring instructions.

AttachLink*(name, property, dout, din, access, pollrate, callback, callbackId, mode)*

One can of course wrap these calls with get(), set() and monitor(), but will loose generality in doing so. Indeed some general features are then difficult to re-introduce, such as a set-get atomic operation or starting a monitor with instructions to stop processing until the first update, etc.

Using these API calls to access the hardware to access the local hardware becomes a simple matter if the endpoint uses the device context "localhost", and the device server "cdi". Special parsing of the full device name also allows multiple endpoints with a single call. For instance, a call to "/localhost/cdi/#1" would access only the device registered as device number 1. On the other hand a call to "/localhost/cdi/#1-#100" or "/localhost/cdi/#1,#3-#10,#99" would identify the individual registered devices and access them as a group. The CDI property space include the properties "RECV" for receiving (reading) data from the device, "SEND" for sending (writing) data to the device, "RECV.SEND.ATOM" and "SEND.RECV.ATOM" for issuing a pair-wise read-write or write-read operation which is guaranteed to be atomic, "RECV.CLBR" and "SEND.RECV.CLBR" for returning data which has been calibrated according to the registered calibration rules, and such properties as "ADDR" and "BUSNAME" which return information about the endpoint device.

For the sake of an example, a CDI monitor call in C might look like

```
dout.dArrayLength = 100;
dout.dFormat = CF_UINT16;
dout.data.sptr = rbData;
AttachLink("/localhost/cdi/#1-#100", "RECV.CLBR,
               &dout, NULL, 1000, cb);
```

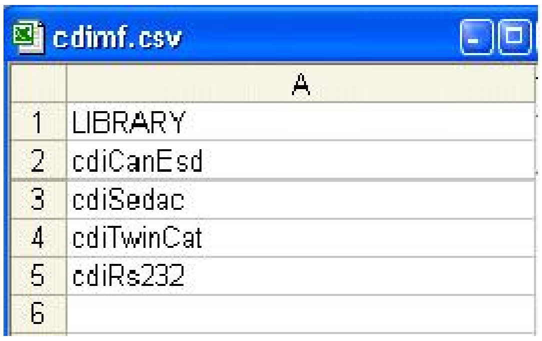
which would read devices 1 to 100, calibrate the data, fill in the read-back buffer rbData and call the callback routine cb at 1 Hz.

Needless to say, there is a similar interface for other languages such as java, Visual Basic, or LabView. We should also point out that CDI devices are registered both with a device name and a device number. The registered device name can likewise be used in the above calls, which might be desirable when browsing the registered hardware remotely. Generally speaking, a server developer will be more inclined to use the device number when accessing the attached hardware, since the actual name of temperature sensor, sputter pump, BPM or whatever in question is mostly irrelevant at that level of data access.

CDI DETAILS

CDI operates on a plug-and-play basis, and adding a new bus interface plug to CDI only involves writing a new bus interface plug. The CDI shared library needs only to be compiled and installed once for the platform in question. On windows for instance this will be cdi32.dll (or cdi64.dll) and on Unix systems libcdi.so. Application platforms such as java, VB, or LabView will also access this same shared library.

When the library loads, it will look for a CDI bus manifest file, which is a simple comma-separated-value file and can be a simple as a single column with a list of bus plug libraries, as shown below in figure 1.


cdimf.csv

	A
1	LIBRARY
2	cdiCanEsd
3	cdiSedac
4	cdiTwinCat
5	cdiRs232
6	

Figure 1. Example of a CDI bus manifest file.

CDI will then call cdiLoadLib() for each bus plug entry in the manifest list, for instance

cdiLoadLib("cdiCanEsd.dll");

on windows or

cdiLoadLib("libcdiCanEsd.so");

on Unix platforms, etc. If the library loads successfully, it will (via its prologue code) register its name and all of its bus handlers with CDI, which itself has no a priori knowledge of any hardware bus interface.

After the manifest has been read, CDI will look for a CDI device database and, if found, read it and register all devices and device information contained within. The registered information will include the bus name and address of the device and any accompanying bus parameters (such as bus speed) along with its assigned device name and number, as well as data access parameters and calibration rules. The supported calibration operations include addition (and subtraction), multiplication, and exponentiation along with bit shifting and modulo arithmetic on integer values. There is no limit to the number of calibration rules which can be applied, and they can be applied in any order, although care must be taken when mixing possible floating point rules such as multiplication or exponentiation with purely integer rules such as modulo arithmetic or bit shifting, so that the outcome of the calibration makes sense. We note here for completeness that CDI can operate without a database, but then all devices must be registered via API calls from with the server application.

It should also be pointed out that writing a new bus plug for CDI is a relatively straightforward process. CDI itself will do nothing but load the bus plug library. It is the duty of the bus plug to register itself with CDI. There are a handful of CDI routines the bus plug should make use of in order to function properly. These are essentially all registration routines which provide CDI with the bus handlers for accessing its bus hardware (calls to open the bus, read and write to the bus, and close the bus). If a new hardware device has a driver API for the target platform, then the task of writing a bus plug is no more complicated than wrapping the appropriate API calls within the CDI bus plug handlers.

To this end, a “bus” plug does not itself have to interface directly to hardware, but could simply provide access to more complicated, generic hardware entities, such as “stepper motor” or “oscilloscope”. These entities could themselves access the hardware directly (using CDI in a nested manner) and incorporate the “business logic” which handles the generic functionality the all “stepper motors” or all “oscilloscopes” have.

REMOTE CDI

TINE servers which make use of CDI will automatically offer the full palette of device access available locally to remote clients. In other words, CDI will export a de facto device server (unless operating in stand-alone mode) which makes the hardware available for remote access. A remote client wishing to access the device hardware directly can do so by contacting the endpoint using the device context of the registered TINE server and the device server name given by the Front End Controller (FEC) name appended with the suffix “.CDI”. In other words, a Beam Position Monitor FEC, called BPM which registers itself in context PETRA will automatically export a device server called BPM.CDI also in context PETRA. The local server will access its hardware using, for example, the endpoint

“/localhost/cdi/#1-#100”

whereas a remote client could access the hardware via

“/PETRA/BPM.CDI/#1-#100”

Indeed, remotely it might make more sense to use device names and issue the call as

“/PETRA/BPM.CDI/BPM/OR1 – NL25”

assuming that “OR1” is the registered name for device 1 and “NL25” is the registered name for device 2.

Although these remote services are automatically present and do not require coding, accessing the hardware devices remotely in this manner should be thought of as a debugging service. On some (rare) occasions, when the server itself has no more complicated duties than to read out a number of, say, temperature sensors, calibrate the results and offer them, then the default CDI server can be used almost as is (It would probably be prudent in this case to alias the CDI property “RECV.CLBR” with “Temperature”, for example).

CURRENT STATUS

CDI is now being tested at DESY for isolated but relevant cases in HERA and PETRA, and is so far proving to be stable as easy to use as advertised. The existing bus plugs include three varieties of SEDAC bus plugs (which suggests that these should rather be thought of as bus interface plugs) for both Windows and Linux, two varieties of CANOpen for Windows and Linux, RS232 for Windows and Linux and most recently for the TwinCat interface from Beckoff [7] (for Windows).

Current efforts will now focus on providing database management tools which allow rapid and straightforward creation and checking of the CDI database.

REFERENCES

[1] R. Bacher, et. al., “Common Device Access for Accelerator Controls”, Proceedings PCaPAC 2005.

[2] M. Kraimer, E. Norum, and M. Rivers, “asynDriver: Asynchronous Driver Support”, http://www.aps.anl.gov/epics/modules/soft/asyn.

[3] http://tine.desy.de.

[4] http://www.aps.anl.gov/epics.

[5] Z. Kakucs, P. Duval, M. Clausen, “An EPICS to TINE Translator”, ICALEPCS 2001.

[6] http://doocs.desy.de

[7] http://www.beckhoff.com/english