COMMON DEVICE INTERFACE 2.0

P. Duval, H. Wu, DESY, Hamburg, Germany
J. Bobnar, Cosylab, Ljubljana, Slovenia

Abstract

The Common Device Interface (CDI) is a popular device layer [1] in TINE control systems [2]. Indeed, a de facto device server (more specifically a 'property server') can be instantiated merely by supplying a hardware address database, somewhat reminiscent of an epics IOC. It has in fact become quite popular to do precisely this, although the original design intent anticipated embedding CDI as a hardware layer within a dedicated device server. When control system client applications and central services communicate directly to a CDI server, this places the burden of providing useable, viewable data (and in an efficient manner) squarely on CDI and its address database. In its initial release variant, any modifications to this hardware database needed to be made on the file system used by the CDI device server (and only when the CDI device server was not running). In this report we shall describe some of the many new features of CDI release 2.0, which have drawn on the user/developer experience over the past eight years.

CDI AND TINE

Although the Common Device Interface (CDI) can be used outside of the TINE control system it is nonetheless strongly coupled to the TINE libraries as well as the TINE application programmer's interface (API) and the TINE naming convention and hierarchy. It is worthwhile to discuss some of these aspects in order to better understand the discussion that follows.

TINE itself does not require any specific hardware device layer in order to provide control system services. On the other hand TINE is weakly coupled to CDI in that specific CDI hooks are embedded in the TINE library. This in turn allows a TINE device server to utilize embedded CDI services for hardware access.

CDI Hardware Server

A very simple manifestation of embedded CDI services is the so-called CDI hardware server, which is essentially a generic TINE device server providing access to the hardware devices contained in the CDI database. Such a server provides no additional device control intelligence beyond that which can be configured in the database. Originally it was imagined that although the hardware server would be a very useful tool for testing hardware, developers would design device servers based on direct data acquisition via embedded services within a single framework. In practice, the hardware server itself has become the mainstay of hardware access for most TINE device servers. In most cases a device server with specific control intelligence is designed as an effective middle layer server communicating with a front-end CDI hardware server. In many cases, however, client applications communicate directly with CDI servers and there is no additional device server in the picture at all!

At this juncture we should point out that the CDI hardware server should properly be termed a property server and not a device server. Control systems are sometimes categorized into those which provide a database-driven paradigm (such as EPICS [3]) or a device-server paradigm (such as DOOCS [4] or TANGO [5]). Although TINE falls into the device-server camp it also supports property servers. Traditional device-servers treat instances of equipment as named devices and these devices have properties which one can access via the device server. A property server on the other hand considers services and information to be designated as properties located on some host, and such a service property will likely apply to a set of keywords.

The services a CDI server offers of course include bus access properties, such as sending and receiving on a hardware bus. For such properties, the keywords correspond directly to named hardware addresses, referred to as CDI devices. Other services include bus and template information as well as database management services.

CDI SPECIFICS

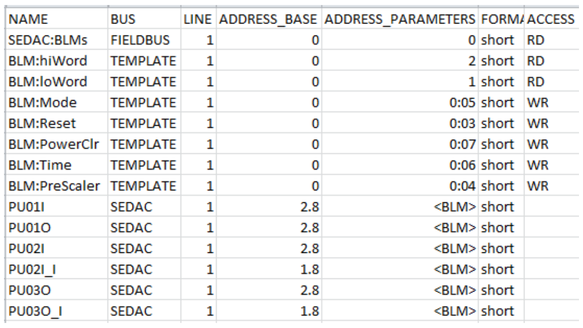
CDI operates on a plug-and-play basis, making use of bus-plug interfaces to the device hardware. A bus plug is a hardware specific shared library which encapsulates the details of the hardware bus I/O behind the CDI API. The CDI shared library is told which bus plug libraries to load via a CDI manifest database.

A CDI address database provides the cross-reference information necessary to instantiate the named hardware devices that CDI will export. The use of CDI address templates can greatly facilitate this instantiation. As a database format, CDI uses comma-separated value (CSV) files, which are easy to view in any text editor and fit seamlessly into any spreadsheet application such as Excel.

A CDI address database snippet is shown in Fig. 1. Here one can see how templates make life easy. Templates are defined by specifying the bus name 'TEMPLATE' and providing both the template name and template field name separated by a colon. If a device instance such as 'PU01I' specifies a template <BLM> in its address parameters then it will automatically expand into multiple CDI I/O devices given by instance name and template field name separated by a dot. The bus itself in this case is the in-house DESY bus SEDAC. The initial entry gives the special bus name 'FIELDBUS' which

ISBN 978-3-95450-146-5

provides a method for giving a name to the bus on line SEDAC line 1, namely 'BLMs'. Not shown in the example is another special bus name BITFIELD which can be used to define the elements of a TINE bitfield which can be applied to read-back data.



NAME	BUS	LINE	ADDRESS_BASE	ADDRESS_PARAMETERS	FORMA	ACCESS
SEDAC:BLMs	FIELDBUS	1	0	0	short	RD
BLM:hiWord	TEMPLATE	1	0	2	short	RD
BLM:loWord	TEMPLATE	1	0	1	short	RD
BLM:Mode	TEMPLATE	1	0	0:05	short	WR
BLM:Reset	TEMPLATE	1	0	0:03	short	WR
BLM:PowerClr	TEMPLATE	1	0	0:07	short	WR
BLM:Time	TEMPLATE	1	0	0:06	short	WR
BLM:PreScaler	TEMPLATE	1	0	0:04	short	WR
PU01I	SEDAC	1	2.8	<BLM>	short	
PU01O	SEDAC	1	2.8	<BLM>	short	
PU02I	SEDAC	1	2.8	<BLM>	short	
PU02I_I	SEDAC	1	1.8	<BLM>	short	
PU03O	SEDAC	1	2.8	<BLM>	short	
PU03O_I	SEDAC	1	1.8	<BLM>	short	

Figure 1: CDI address database snippet.

The database snippet shown in Fig. 1 is simplified and omits many optional columns which could be used to specify other I/O instructions including data masks and calibration rules. As noted above, a well-configured database can often establish a CDI server which provides finished, ready-to-use data, obviating the need to develop any other device server. In the past, this was usually only true when the targeted hardware was 'simple', i.e. where slow data access was sufficient and multiple clients to the CDI server could be tolerated.

With the advent of scheduling, asynchronous listeners, and the asynchronous triggering from bus plugs found in CDI 2.0, the number of cases where a CDI server alone is sufficient for control purposes has greatly expanded. We shall come to these topics in more detail below.

One further important detail concerning the property server nature of a CDI server should be mentioned. Namely the fields of the template devices alluded to above are themselves registered as properties, whose keyword lists consist of those instances making use of the template. To continue with Fig. 1 for example, the CDI server would also export a property 'Mode' and list all of the PU01I, PU01O, etc. instances as keyword devices. More importantly it would treat this property (referred to as a CDI extended property) as a multi-channel array (MCA) property, which would participate in the TINE MCA contract coercion [6] used to provide efficient data transfer, server to client. Underneath the hood, any call to a CDI extended property maps to the corresponding full CDI device name ('*instance.template-field*') and the bus property 'RECV.CLBR', which translates into 'receive on the bus and apply any calibration rules'.

Asynchronous Listeners

If a call to a CDI server always ended up making a synchronous bus i/o operation this could lead to bottle necks and inefficiencies depending on the amount of data and the read-back intervals involved. To this end, a CDI server will recognize when a client establishes an asynchronous contract and then establish a local asynchronous static listener, which will regularly receive updates for the requested device from the hardware and report these results to the caller. In fact, when an asynchronous listener is in play, any synchronous request from a caller will return the most recently acquired data without any additional hardware I/O. This of course does not apply to SEND operations on the bus, but as most server I/O tends to be read-backs this mechanism makes the CDI server very efficient at servicing multiple clients.

A CDI server can now easily specify in its address database which properties (or template fields) should have an automatic listener applied at start time.

Scheduling

When an asynchronous listener is in place, then the CDI library can itself monitor read-back values and signal a data-change event. This is done by calling the TINE scheduler and is referred to as scheduling a property. CDI will generally monitor data at an interval provided in the address database, although it can also react to asynchronous events coming from the bus plug. Scheduling is a good way to avoid latency and deliver data to listening clients in a timely manner.

Local Histories and Alarms

To further the cause for creating a viable device server (or rather, property server) merely by applying the proper database settings, we note here that TINE local histories of any designated CDI device can be easily incorporated into the database. Raising alarms is another matter and requires a separate TINE alarm watch database [2]. However, this latter task is hardly daunting.

Remote Database Access

One of most exciting features of CDI 2.0 is the ability to remotely access and modify the database of a running CDI server by making use of exported database properties. Although reading the database is allowed by anyone, writing to the database requires traversing TINE security. Should a local database be updated, a backup is made of the most recent database and a rollback to the previous database is also easily available in CDI 2.0.

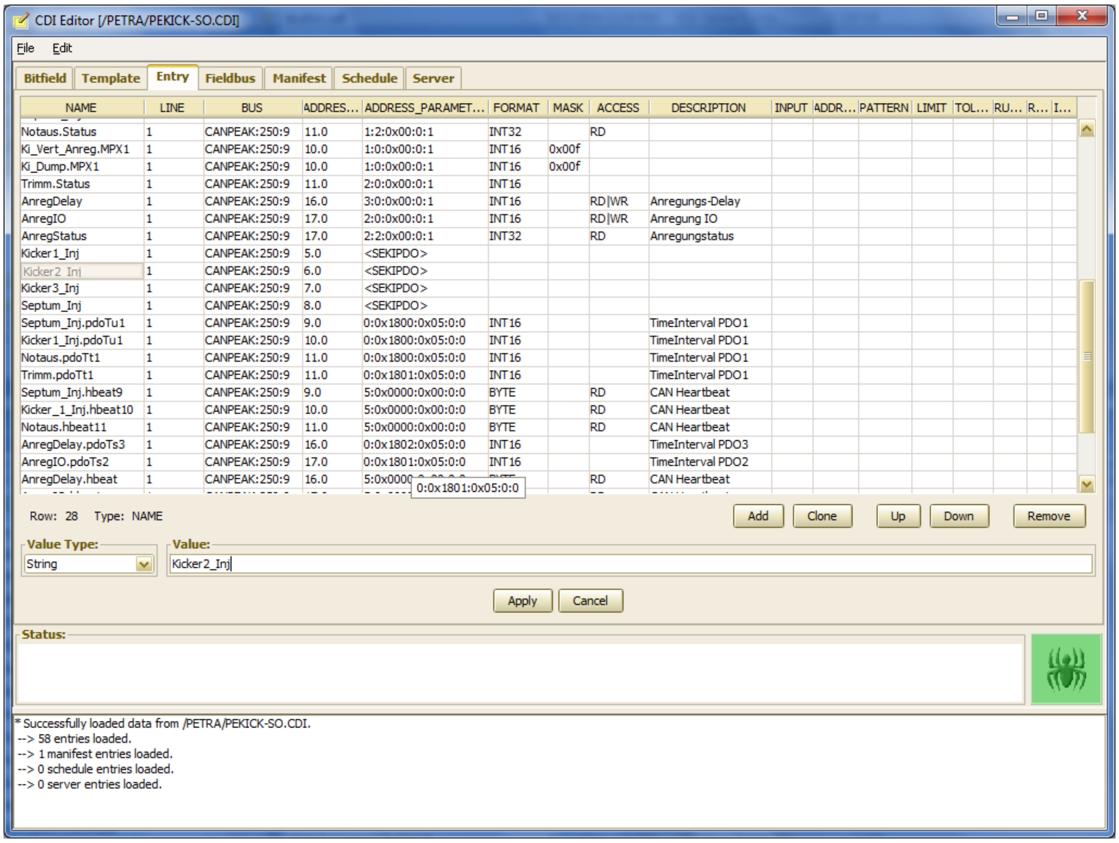
The ability to update a CDI database via the CDI server itself simplifies matters to no end when one is dealing with embedded or semi-embedded CDI servers on, say, a PC104 card. In the old days, this typically required a secure login/secure copy of the new database and a server re-start.

If enabled, the CDI server can also be remotely 'reset', whereby the wrapping TINE server unloads all CDI libraries and returns the running server to its original uninitialized state and then re-initializes.

CDI EDITOR

As seen above, a CDI database makes good use of CSV files. Although an accomplished spreadsheet user might be comfortable with this, working with detailed

ISBN 978-3-95450-146-5


CDI Editor [/PETRA/PEKICK-SO.CDI]

NAME	LINE	BUS	ADDRES...	ADDRESS_PARAMET...	FORMAT	MASK	ACCESS	DESCRIPTION	INPUT	ADDR...	PATTERN	LIMIT	TOL...	RU...	R...	I...
Notaus.Status	1	CANPEAK:250:9	11.0	1:2:0x00:0:1	INT32		RD									
Ki_Vert_Anreg.MPX1	1	CANPEAK:250:9	10.0	1:0:0x00:0:1	INT16	0x00f										
Ki_Dump.MPX1	1	CANPEAK:250:9	10.0	1:0:0x00:0:1	INT16	0x00f										
Trimm.Status	1	CANPEAK:250:9	11.0	2:0:0x00:0:1	INT16											
AnregDelay	1	CANPEAK:250:9	16.0	3:0:0x00:0:1	INT16		RD|WR	Anregungs-Delay								
AnregIO	1	CANPEAK:250:9	17.0	2:0:0x00:0:1	INT16		RD|WR	Anregung IO								
AnregStatus	1	CANPEAK:250:9	17.0	2:2:0x00:0:1	INT32		RD	Anregungstatus								
Kicker1_Inj	1	CANPEAK:250:9	5.0	<SEKIPDO>												
Kicker2_Inj	1	CANPEAK:250:9	6.0	<SEKIPDO>												
Kicker3_Inj	1	CANPEAK:250:9	7.0	<SEKIPDO>												
Septum_Inj	1	CANPEAK:250:9	8.0	<SEKIPDO>												
Septum_Inj.pdoTu1	1	CANPEAK:250:9	9.0	0:0x1800:0x05:0:0	INT16			TimeInterval PDO1								
Kicker1_Inj.pdoTu1	1	CANPEAK:250:9	10.0	0:0x1800:0x05:0:0	INT16			TimeInterval PDO1								
Notaus.pdoTt1	1	CANPEAK:250:9	11.0	0:0x1800:0x05:0:0	INT16			TimeInterval PDO1								
Trimm.pdoTt1	1	CANPEAK:250:9	11.0	0:0x1801:0x05:0:0	INT16			TimeInterval PDO1								
Septum_Inj.hbeat9	1	CANPEAK:250:9	9.0	5:0x0000:0x00:0:0	BYTE		RD	CAN Heartbeat								
Kicker_1_Inj.hbeat10	1	CANPEAK:250:9	10.0	5:0x0000:0x00:0:0	BYTE		RD	CAN Heartbeat								
Notaus.hbeat11	1	CANPEAK:250:9	11.0	5:0x0000:0x00:0:0	BYTE		RD	CAN Heartbeat								
AnregDelay.pdoTs3	1	CANPEAK:250:9	16.0	0:0x1802:0x05:0:0	INT16			TimeInterval PDO3								
AnregIO.pdoTs2	1	CANPEAK:250:9	17.0	0:0x1801:0x05:0:0	INT16			TimeInterval PDO2								
AnregDelay.hbeat	1	CANPEAK:250:9	16.0	5:0x0000	[illegible]		RD	CAN Heartbeat								

0:0x1801:0x05:0:0

Row: 28 Type: NAME

Value Type: String

Value: Kicker2_Inj

```
* Successfully loaded data from /PETRA/PEKICK-SO.CDI.
--> 58 entries loaded.
--> 1 manifest entries loaded.
--> 0 schedule entries loaded.
--> 0 server entries loaded.
```

Figure 2: Example of the CDI Editor. From the title bar one can see that the database was acquired directly from a server called /PETRA/PEKICK-SO.CDI. The above view focuses on the device *entries*.

information in a large spreadsheet can be tedious and error prone. To this end, a CDI editor is now available, which shows various aspects within their contexts and, most importantly, checks for consistency where it can.

As a CDI 2.0 server offers remote access to its database, the CDI editor can either work with a local database on the local file system or acquire the database from any running CDI server.

The user can browse and edit in a clear, intuitive manner all aspects of the database and then have the option of updating to the targeted server if he has the access rights to do so.

An example is shown in Fig. 2 above.

CDI DEPLOYMENT TOOL

Another issue which has surfaced over the past eight years of CDI usage is that of distributed databases. Namely, there are situations where not only is a real device server necessary, but said device server needs to acquire data from multiple, distributed, CDI hardware servers, each containing instances of the same hardware type. A simple modification to a template then involves updating not one, but numerous CDI address databases (not an enviable task, even making use of the CDI editor).

To simplify such operations the CDI 2.0 package also features a CDI deployment tool. This utility is able to generate the multiple CDI address files from a single master address database. An additional requirement here is that the master address database must have a column labelled TARGET, which should contain the destination CDI server address (in TINE */Context/Server* notation) for each hardware instance. The deployment tool is able to then generate a set of database files on the local file system in a directory structure based on the *Context* and *Server* information provided. Or, it is able to make use of the remote database access described above to push the specific database to the targeted server.

CONCLUSION

CDI has reached a new level of maturity and continues to be a major workhorse in TINE control systems. The feature-set of release 2.0 makes CDI considerably more versatile, and the ability to generate a hardware address database is now vastly simplified through the CDI editor.

REFERENCES

[1] P. Duval and H. Wu, "Using the Common Device Interface in TINE", Proc. PCaPAC'06, http://jacow.org/.
[2] TINE website: http://tine.desy.de/.
[3] EPICS website: http://www.aps.anl.gov/epics/.
[4] DOOCS website: http://doocs.desy.de/.
[5] TANGO website: http://www.tango-controls.org/.
[6] P. Duval and S. Herb, "The TINE Control System Protocol: How to achieve high scalability and performance", Proc. PCaPAC'10, http://jacow.org/.

ISBN 978-3-95450-146-5