EtherCAT DRIVER AND TOOLS FOR EPICS AND LINUX AT PSI

D. Maier-Manojlovic, Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

Abstract

A combined EPICS/Linux driver package has been developed at PSI, to allow for simple and mostly automatic setup of various EtherCAT configurations. The driver is capable of automatic scan of the existing devices and modules, followed by self-configuration and finally autonomous operation of the EtherCAT bus real-time loop. Additionally, the driver package supports the user PLC to manipulate EtherCAT data in real time, implements fast real-time (single cycle) slave-to-slave communication (skipping EPICS layer or PLC completely), features guaranteed one-shot trigger signals otherwise not supported by EPICS and much more.

INTRODUCTION

For the modules and devices equipped with the EtherCAT bus interface [1], a general, real-time software interface was needed for the integration in the existing accelerator control system, both for existing facilities like SLS (Swiss Light Source) and HIPA (High Intensity Proton Accelerator), and for facilities and systems being built at the time this document was created, such as SwissFEL [2] (Swiss X-Ray Free Electron Laser).

First, we have tested the existing solutions, both related and unrelated to our controls system of choice, EPICS. Unfortunately, none of the existing commercial and non-commercial solutions we have reviewed and tested was able to cover and satisfy all of the requirements for the EtherCAT support at PSI.

CONCEPTS

Providing full support for such a wide range of systems and applications in a single package presented a problem since not every requirement or possible usage scenario could have been satisfied with a single piece of software.

EPICS control system support requires its own type of dedicated device support driver. Unlike its kernel counterparts, EPICS driver has to run in Linux userspace, since EPICS system itself is a userspace application. Aside from EPICS, the system has to support other types of applications.

To make things more complicated, the applications that are supposed to use the system are running in both userspace and kernelspace. This, of course, requires distinctively different structure of the supporting interfaces and practically double the work needed to create the system and maintain it later. Real-time applications can be created to run in either userspace or kernelspace, which in turns mean at least two separate local APIs had to be created.

EtherCAT Data Addressing

To describe an address of a given EtherCAT data entry, the following IDs have to be included:

- *master number* (since there can be multiple masters running on the same host),
- *domain number* (domain is an arbitrary, user-defined collection of PDO (Process Data Object) entries sharing the same buffer memory, EtherCAT packages and network exchange frame rate),
- *slave number* (slave is simply another name for an EtherCAT Module),
- *synchronization manager number* (synchronization managers, also known as SyncManagers or SMs, group EtherCAT PDO objects by their exchange direction (input/output) and other, manufacturer or end-user defined criteria),
- *process data object number* (process data objects, or PDOs, group entries by some arbitrary purpose defined by the manufacturer of the EtherCAT module, or if a module supports it, by end-users) or process data object entry number (process data object entries, or PDO Entries, hold the actual data exchanged by the module).

To solve this problem, we have devised a new addressing schema for EtherCAT data, as depicted in Fig. 1.

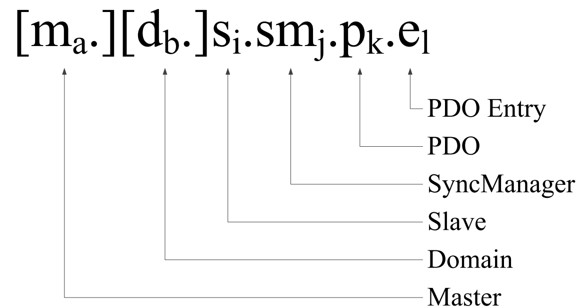

Figure 1: New schema for EtherCAT data addressing.

Master (**m**$_a$) and domain (**d**$_b$) can be omitted when a=0 or b=0, i.e., when the first, default master and domain are used. Similarly, the PDO entry (**e**$_l$), PDO (**p**$_k$) or SyncManager (**sm**$_j$) can be omitted as well (in that order), when the user wants to address multiple entries included in a larger parent container, instead of a single entry.

The possible modifiers or addressing modes (as of v2.0.6) are:

- `[.o<offset>]` - **forced offset** (in bytes). This allows shifting of the starting address of the PDO entry in the buffer, effectively allowing for partial reading of the entries or using certain "tricks" to reach otherwise unreachable or hard to reach content
- `[.b<bitnr>]` – **forced bit extraction**, allows extraction of single bits from any larger PDO entry data, regardless of its original type or length
- `[.r<domregnr>]` – **domain register addressing**, replaces address modifiers *s, sm, p* and *e*, using rela-

Copyright © 2018 CC-BY-3.0 and by the respective authors

tive entry addressing inside a domain instead. This allows end-user to simply address registers inside a module, SyncManager or PDO without having to remember or know all parts of the EtherCAT address, or to know the internal structure of a module

- `[.lr<entryrelnr>]` – **local register addressing**, replaces any (group) of the address modifiers s, sm and p, allowing for local relative addressing of all entries inside a slave, inside a SyncManager, or inside a PDO regardless of their actual parent container or containers
- `[t<type>]` or `[t=<type>]` provides means for **forced typecasting or type override**, changing the default type of the data entry when applied. Many typecasts are provided for this purpose, such as *int/uint* (8-, 16-, 32-, 64-bits), and also *float*, *double*, *BCD*, etc.
- `[.l<length>]` – **length modifier**, in bytes. Used primarily to define the length of *stringin/stringout* EPICS records, but can be used for any other buffer extraction, also

EPICS SUPPORT

Since at PSI the EPICS control system is almost exclusively used for the accelerators and device control, integrating EPICS support was a top priority.

As an example of a typical system controlling EtherCAT components, the EPICS Core is running on the Ioxos IFC 1210 Board [3], equipped with two separate Ethernet interfaces, a PowerPC P2020 CPU and the VME Bus backplane. The operating system used on these systems is a custom built Linux with the appropriate PREEMPT-RT patch.

Since EPICS has its own interface for device drivers, a special EPICS userspace driver had to be developed, using high priority real-time threads for the control loop. Without the real-time capabilities provided by the PREEMPT-RT Linux, timing and execution of the control loop would be less reliable and hence not real-time capable. However, if somewhat increased level of jitter is acceptable, the EtherCAT driver can run on a non-real-time Linux, i.e., without the PREEMPT-RT Patch installed.

Another problem that we have to solve was the fact that EtherCAT modules (slaves) are not always accepting the write values –for example, a write request in a certain cycle may fail for a number of reasons, and that means that the Ethernet packet on a return trip may contain register values which differ from the values stored in the write buffer of the driver.

This means that not only the refreshed read values, but also the write values has to be transferred back to the write buffer in order to overwrite the obsolete and potentially not matching values at the end of every cycle.

Yet, the newly received write values, unlike new read values, cannot be simply copied over the old values in the buffer, since that would effectively overwrite the new write request values which were already accepted from EPICS or other clients since the beginning of the last cycle. To solve this, a multithreaded double-buffering with the bit granularity write-mask for write requests was implemented (Fig. 2).

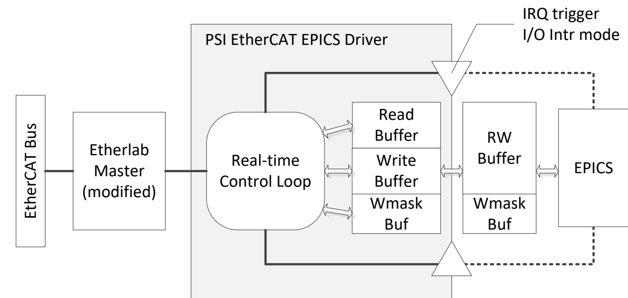

Figure 2: PSI EPICS EtherCAT driver structure.

EPICS Records

All EPICS records use a single driver type (DTYP) called **ecat1** (PSI multi-client and EPICS EtherCAT driver) **ecat2** (PSI pure EPICS EtherCAT driver). Scan rates for records can be set to any valid EPICS scan rate (from parts of a second to multiple seconds), including *Passive* and *I/O Intr.*

Register Values Typecasting

All of the extended addressing modes and modifiers, including typecasting (t=type in Input or Output field of a record), can be used in EPICS as well. Almost all combinations of provided typecasts and modificators are allowed. Using this feature it is possible to achieve high level of flexibility even in EPICS, with its rigidly defined record types.

For example, it is possible to extract a single bit from an *mbbi* record without having to create additional *calc* records, or having to recalculate the complete bit field extracted from a record of any length. Type override and other modifiers can be used for all record types, including array-type records (*aai/aao*). Length modifier (`.l<length>`) is used to define the length of the string for string-type records (*stringin/stringout*).

GENERAL SUPPORT

To support the local and remote applications wishing to connect to and use the EtherCAT hardware, the standalone, multi-client version of the driver package was created (ecat1). In this package three different subsystems have been developed in order to allow application developers a highly flexible way to access the EtherCAT slaves and their data:

- Kernelspace API
- Userspace API
- Modules, SMs, PDOs and PDO entries as *procfs* directory/file structure

Additionally, the driver automatically constructs and maintains *procfs* trees throughout its operation, and takes care of double/triple triple-buffering and write-masking

Copyright © 2018 CC-BY-3.0 and by the respective authors

process needed for data exchange with the client applications. Description of the each of the access modes is presented below.

Kernelspace API

Kernelspace API (Fig. 3) is a set of functions providing the easy access to driver control loop parameters and EtherCAT data entries.

The driver provides an internal real-time control loop for buffering and exchange of TCP packets over Ethernet, Timing of the control loop is based on the host high resolution timers, but can be driven by an external source as well, such as a timing system input.

Multiple kernelspace applications and/or drivers can use the API.

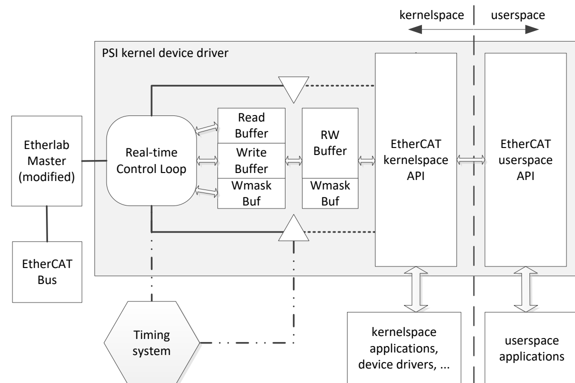

Figure 3: PSI EtherCAT driver structure.

Userspace API

Userspace API (Fig. 3) is an API, i.e., a set of functions providing (almost) the same functionality found in the Kernel API. The functions library can be used statically or dynamically with userspace applications as needed.

One of the Userspace applications that had to be covered is EPICS itself, so a native EPICS driver using this API has been developed as a part of the package. When this EPICS driver is used, both EPICS and other client applications (running in both kernel- and userspace) can be used concurrently.

EtherCAT Data in "procfs tree"

To allow even more applications to access the EtherCAT data, but without the need for an API or a dedicated remote server and client, we have developed the concept of "*procfs trees*" to represent the tree structure of EtherCAT modules and their components. *procfs trees* are a series of virtual "directories" and "files" constructed on-the-fly by the drivers in the Linux host *procfs* file system.

Each directory represents some kind of a parent container, such as a slave, a SyncManager, a PDO, a domain or a master (Fig. 4). Each virtual file in these directories represents either a direct representation of an EtherCAT PDO entry, or a utility file representing the data about the system or about the containers present.

There is also a special *cmd* entry provided in the *procfs tree*, allowing a CLI or application to interactively talk to the driver and sending commands (for example, add entry, add PDO, add slave entries, list data, etc.). Also, entry data can be read or changed using the CLI as well.

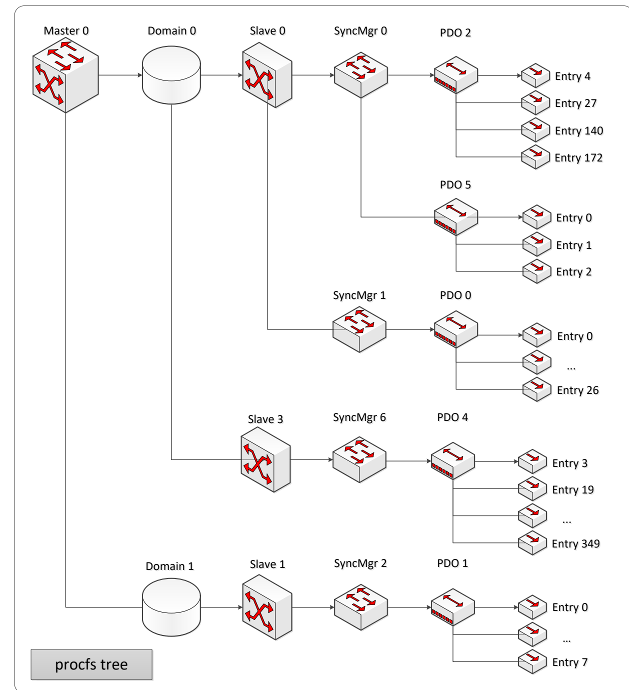

Figure 4: *procfs tree* general structure.

EXTENSIONS AND UTILITIES

The PSI EtherCAT support package offers several extensions and tools. The most important ones are described below.

Slave-to-slave Communication

It is often the case that the data on the EtherCAT bus has to be transferred from one EtherCAT device or module to another, preferably in real-time. For this kind of communication, two different directions of transfer can be observed, *upstream* and *downstream*.

Upstream slave-to-slave communication is transfer of data from a module further away on the EtherCAT bus from the master to a module closer to the master. Downstream communication is the transfer from a closer module to one further "down the stream" from the master. The stream in this case represents the path an EtherCAT TCP packet is travelling, and its direction remains constant as long as there are no physical changes in the bus configuration and modules present.

From real time point of view, this communication is highly deterministic, yet not identical – downstream communication (send on one module, receive on another) can, theoretically, be done in the same bus cycle, hence costing exactly zero bus cycles to execute. Upstream communication, due to the way TCP packets are handled by the EtherCAT, will take exactly one bus cycle to complete.

We have decided to implement the slave-to-slave communication (*sts*) with constant cost of completion, in this

Copyright © 2018 CC-BY-3.0 and by the respective authors

case, exactly one bus cycle for both upstream and downstream communication requests.

In EPICS, *sts*-communication transaction requests can be inserted as follows:

```
ecat2sts <source> <destination>
```

For example:

```
ecat2sts r8 r0
ecat2sts r2.b0 r0.b6
ecat2sts s2.sm0.p1.e0 s1.sm0.p1.e0
ecat2stss3.sm3.p0.e10.b3 s4.sm2.p1.e0.b7
```

As can be seen in examples above, any valid addressing mode and/or modifier can be used for source and destination. API access is done by calling a function to register a transaction request, but the addressing remains the same.

Support for Programmable EtherCAT Modules

The PSI EtherCAT drivers and utilities also support setting up and live programming of programmable EtherCAT modules and devices, such as, for example, EtherCAT network bridges (EL6692, EL6695), motor controllers, and so on.

From EPICS, any module can be programmed by using *ecat2cfgslave* set of commands, for example:

`ecat2cfgslave sm <arguments...>` - configures one Sync Manager for the given slave.

`ecat2cfgslave sm_clear_pdos <arguments…>` - clears (i.e., deletes) all PDOs for a given Sync Manager (SM).

`ecat2cfgslave sm_add_pdo <arguments…>` - adds a PDO with index pdoindex to a Sync Manager.

`ecat2cfgslave pdo_clear_entries <arguments…>` - clears (i.e., deletes) all PDO entries associated with the given PDO.

`ecat2cfgslave pdo_add_entry <arguments…>` - creates a new PDO entry and associates it with the given PDO.

Programmable network bridge EtherCAT modules, such as EL6692 or EL6695, have their own, simplified commands for programming entries:

```
ecat2cfgEL6692 <netbridge_nr> in/out <numberofbits>
```

Support for User PLC in Dynamic Libraries

During the development and testing of EtherCAT drivers, the need for user-defined PLCs has been identified. In addition to slave-to-slave communication mentioned earlier, users sometimes wanted to be able to directly manipulate the EtherCAT data themselves.

A possibility to create a C/C++ based PLC, placed in a user-created dynamic library that is automatically detected and loaded, if needed was added.

This way, there is no need to change and recompile the driver directly, user only has to create a dynamic library with certain functions present, and it will be called (with a callback-function to prevent real-time jitter), inside each EtherCAT cycle.

Now the users have the possibility to directly manipulate EtherCAT module values in each cycle, in real-time, without the need of a slow, interpreted script based solution, or doing it using EPICS record, which is also an order of magnitude slower than our solution.

CONCLUSION

In this paper, we have presented the PSI EtherCAT driver packages and described the package components.

The system using ecat2 driver is already successfully used at PSI for several years, for all systems controlling EtherCAT modules from EPICS and from real-time applications. The other version of the driver, ecat1, is currently undergoing final testing at PSI under full-load and real-life conditions.

As is usual with such systems, it is to be expected that changes will be made to this package in the future to accommodate needs and new requirements of expanding number of users of the system, the existing features will be extended and streamlined and the new features and components will be added.

REFERENCES

[1] Beckhoff GmbH, http://www.beckhoff.de/.

[2] Paul Scherrer Institut (PSI), SwissFEL, http://www.psi.ch/media/swissfel/.

[3] Ioxos Technologies, IFC 1210 – P2020 Intelligent FPGA Controller, http://www.ioxos.ch/.

Copyright © 2018 CC-BY-3.0 and by the respective authors