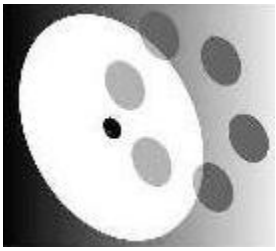
MMTO EtherCAT User's Guide

D. Clark

March 31, 2010



Version 0.1

EtherCAT®

IgH

1. Introduction

This document describes getting, installing, and using the IgH (Ingenieurgemeinschaft) EtherCAT master software and EtherCAT hardware from Beckhoff Automation for use in MMT systems using a Fedora 12 Linux PC. It is not intended as an exhaustive description or detailed “how-to”, rather it is more of a quick-start to assist knowledgeable persons in getting EtherCAT applications and hardware working. Let's get going!

2. Prerequisites

First, you need a Linux PC running Fedora 12. You should bring the machine completely up to date, and apply all yum updates as necessary to have a fresh installation ready to use. As of this writing, the test machine *schwarzwald* is running `Linux 2.6.32.9-70.fc12.i686`.

2.1 Hardware Prerequisites

Next, your PC should have at least two Ethernet ports, at least one of which is an RJ-45 jack; the other could easily be a wireless connection. The port with the RJ-45 will be dedicated to the EtherCAT connection, and you will need a network connection on the other port for connecting to the internet for downloading software, updates, and other uses. While EtherCAT can handle passing ordinary packet traffic across the Ethernet interface, this is an advanced use case that is more complicated than necessary -- network cards are inexpensive, and your motherboard may even already have two Ethernet ports available.

Once you have a running Fedora 12 system with a working internet connection, it's time to install some software to prepare for running EtherCAT.

2.2 Software Prerequisites

MMTO uses the IgH EtherCAT master software from www.etherlab.org. In order to effectively download and use the software, some extra packages are required to get, build, install, and create documentation for the software. It is recommended to go ahead and install all these packages before starting work with EtherCAT rather than discover later something is needed and have to go back and fix it before continuing. So, what do we need?

For building the IgH master software and documentation, we need a) the source for your running kernel, b) *gcc* toolchain, c) *autoconf* and *automake* tools, d)*pdflatex* or equivalent for dealing with LaTex files, e) *doxygen* to generate the C source code documentation, and f)*xfig* and *graphviz* for generating documentation graphics, and g) l*ibtool*, the GNU portable library tool.

The easiest way to get the kernel source is to execute:

```
user@machine> yumdownloader --source kernel
```

Yum will then acquire and install the kernel source at the location `/usr/src/kernels/$uname`. The next step is to create a symbolic link to the kernel source:

```
user@machine> ln -s /usr/src/kernels/$uname linux
```

If you have automatic updates enabled, as is usually the case with a standard Fedora installation, kernel updates will probably update the /kernels directory with the new kernel source, but will not change the linux symlink. You'll have to do that yourself should the need arise. If you reboot after the kernel update, chances are you'll boot into a kernel that no longer has your EtherCAT installation available

and you'll need to rebuild and re-install the software to get it working again. This is not too difficult or time-consuming, just beware it could come along to make things stop working for you.

Your vanilla Fedora installation should come with the *gcc* toolchain by default, but the EtherCAT software is built and distributed with the *autoconf* and *automake* tools, so make sure you have those available to you. Also, you may need *libtool*, as from the '`make install`' step we'll execute later, we have the following message:

Libraries have been installed in:

/opt/etherlab/lib

If you ever happen to want to link against installed libraries

in a given directory, LIBDIR, you must either use libtool, and

specify the full pathname of the library, or use the `-LLIBDIR'

flag during linking and do at least one of the following:

- add LIBDIR to the `LD_LIBRARY_PATH' environment variable
  during execution
- add LIBDIR to the `LD_RUN_PATH' environment variable
  during linking
- use the `-Wl,-rpath -Wl,LIBDIR' linker flag
- have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for

more information, such as the ld(1) and ld.so(8) manual pages.

Now, for building documentation, we need some more packages. In particular, for building the C source code documentation, *doxygen* is needed. The pdf documentation for the EtherCAT software requires *pdflatex*, since the primary documentation source file is a LaTex file. In addition, some of the documentation LaTex inputs are graphics files that in turn require the *graphviz* and *xfig* packages for conversion.

So:

```
user@machine> yum install doxygen | graphviz.i686 | xfig.i686
```

For dealing with LaTex, try using LyX, a LaTex document editor, or whatever is your preference.

Now we are ready to get the actual EtherCAT software and start working with it. We need one final package; IgH has chosen to host their source code on SourceForge as a mercurial repository, so we need to install the *mercurial* package. Mercurial is a Python-based source control management (SCM) tool similar to trac and git. It is free, and can be either installed via yum or the vendor website at http://mercurial.selenic.com/.

3. Installation of the EtherCAT Master Software

IgH maintains a website, as mentioned earlier, at www.etherlab.org. Navigation through the website brings you to the page http://etherlab.org/en/ethercat/index.php, where there is mention of the SourceForge repository, as well as a list of other available software downloads. Don't bother with anything other than the repository versions, as the others are all older versions that are unlikely to work with newer kernels but haven't been taken down to support legacy users.

We want to clone the repository to start with a clean slate, so execute:

```
hg clone http://etherlabmaster.hg.sourceforge.net:8000/hgroot/etherlabmaster/etherlabmaster \
      ethercat-default
```

All mercurial commands begin with 'hg', so the above clones the repository files into the directory ethercat-default, relative to wherever the hg clone command is executed, so if you entered:

```
/home/user> hg clone...
```

you would have a subdirectory named `ethercat-default` under `/home/user`. There is no rule to use the above as the directory name, you're free to use whichever you wish, so long as you have write permission to the parent directory. Mercurial also offers a way to compare versions and update your local repository. For example, the current parent of the software on *schwarzwald* is:

```
[dclark@schwarzwald ethercat-default]$ hg parent
changeset:   1882:8ae8cc56d910
tag:         tip
user:        Florian Pose <fp@igh-essen.com>
date:        Fri Mar 19 12:46:57 2010 +0100
summary:     Minor fix.
```

The above summarizes the current changeset, date of the last update, and a comment about that changeset. The EtherCAT repository changes on almost a weekly basis, so it's worth checking to see if a more recent version addresses any problems you might have. To get and apply updates from the repository, it's very simple--

First, pull the latest changeset from the repository:

```
[dclark@schwarzwald ethercat-default]$ hg pull \
      http://etherlabmaster.hg.sourceforge.net:8000/hgroot/etherlabmaster/etherlabmaster
```

This will bring the latest changes into the project database that mercurial uses for change tracking. But, the source software in your local copy *has not (yet) been changed*. To apply the current changeset:

```
[dclark@schwarzwald ethercat-default]$ hg update
```

Then all relevant files will be updated. You can revert them, as well. See the tutorials at http://mercurial.selenic.com/wiki/QuickStart for more complete information.

It's worth browsing the SourceForge repository for more information on the software version. An interesting way to look at it is to click “graph” from the selections near the top of the page, which will bring up a graphical summary of all the changes, branches, and merges to the software, along with text noting the person, data, and changeset summary comment. At this writing, the current tip in the repository is changeset 1889:131f655c03d3, so the version running on *schwarzwald* is somewhat out of

date.

A clone of the repository will have the `'dist'` version of the software in it. Since the *automake* and autoconf tools are used by the authors to distribute the software, it will be missing some necessary folders for configuring and building the software. Running the ***bootstrap.sh*** script will regenerate these necessary files and you'll be ready to configure and build the software.

At this point, you're probably wondering, “where is the documentation for building the software?”, and the answer is: in the distribution file tree, as a LaTex source document. Oddly, you must build the software, via `'make doc'` to get a pdf version of the documentation, so the process is somewhat circular. You can, however, cheat a bit by downloading the pdf documentation of the master software listed on the download page and get ahead of the game.

The pdf documentation lists a number of important switches for configuring the master software. For use by MMTO, we are mainly interested in building the software with a generic network driver, ***not*** the available native driver, which only supports a few hardware types, notably the Intel Pro1000e and 8139too network interfaces, which are commonly available. An older native driver for the Intel Pro100e is available, but not supported in recent kernels. For the cost of a single digits to a few tens of microseconds of latency in the data transfer (depending on your hardware), the generic driver makes it possible to support any network driver by use of the underlying kernel modules and traversing the lower levels of the network stack. The test PC *schwarzwald* has successfully run EtherCAT transfers at 10kHz, and it is unknown what the ultimate limit is (probably around 80 to 100kHz). This is much higher than envisioned maximum rates of 1kHz, so pursuing native network interface drivers and hardware is not recommended at this time. Data transfer time is, of course, highly dependent on the computer hardware and CPU load, so plan to test your hardware to ensure it can support the highest data rates under worst-case conditions.

Now we are ready to configure the EtherCAT software. Below is the configuration used for the most recent build on *schwarzwald*:

```
$ ./configure --enable-generic=yes --enable-8139too=no —with-linux-dir=/usr/s rc/linux
```

The default configuration options allow a native 8139too driver to be be built, with generic driver turned off, so you must set these explicitly. It's also wise to point to the linux sources you want to use, as well.

Now to build the software:

```
/home/user/ethercat-default>make
```

```
/home/user/ethercat-default>make modules
```

For building documentation (optional):

```
/home/user/ethercat-default>make doc
```

Now you need to be root, and execute:

```
/home/user/ethercat-default# make install
```

```
/home/user/ethercat-default# make modules_install
```

The documentation claims that you should run depmod after modules_install, but on s*chwarzwald* it appears to have been called in the make script, so if your module dependencies don't seem to have made it into the kernel, you might try it.

It should be noted that a few other make targets are provided, among them `'clean'`, `'dist'`, and

'all'. See the Makefile in the /ethercat-default directory for all the targets.

4. Completing the Installation

Once the EtherCAT master software is built and installed, it's time to apply some setup information so the master is attached to the correct Ethernet interface, and the command-line tool is available for use.

The EtherCAT master when started looks for a configuration file, /etc/sysconfig/ethercat for variables that define which interface to use and the master device kernel module to load. For *schwarzwald*, the relevant portion of the file is:

```
MASTER0_DEVICE="00:30:1b:ae:36:0e"
```

and

```
DEVICE_MODULES="generic"
```

The MASTER0_DEVICE is set to the MAC address of the desired network interface hardware, and the module name is identified with the DEVICE_MODULES value. The file is installed with default values during make install, and is richly commented to assist in getting the correct information.

The EtherCAT master offers a character device interface for user-space applications, so it's useful to make a udev rule to allow read/write access for them. The rule used on *schwarzwald* is the file /etc/udev/rules.d/99-EtherCAT.rules, and contains only:

```
KERNEL=="EtherCAT[0-9]", MODE="0666"
```

Finally, a command-line tool is available for interacting with the master (e.g. starting, stopping, setting debug levels, and most importantly, utilities). The tool is installed in /opt/etherlab/bin/ethercat, so it's convenient to make a symlink to the tool binary, such as:

```
ethercat-tool -> /opt/etherlab/bin/ethercat
```

this makes it a bit easier to remember what you're doing, since a lot of different things within the system are called, “ethercat”.

The EtherCAT master posts messages to the syslog, so you can use dmesg to get relevant information about the master's state. For example, since we've completed the above, we can start the master (as root):

```
[root@schwarzwald dclark]# /etc/init.d/ethercat start

Starting EtherCAT master devel  done
```

Obviously in a production setup we probably want the EtherCAT to start at boot time, so set your machine up appropriately. If the startup succeeds, we should see the “done” message above, and we should have dmesg entries for the process, which are always prefaced with EtherCAT to make searching the log easier:

```
EtherCAT: Master driver devel 8ae8cc56d910

EtherCAT: 1 master waiting for devices.

ec_generic: EtherCAT master generic Ethernet device module devel 8ae8cc56d910

EtherCAT: Accepting device 00:30:1B:AE:36:0E for master 0.
```

```
ec_generic: Binding socket to interface 3 (eth2).

EtherCAT: Starting EtherCAT-IDLE thread.

EtherCAT: Link state changed to UP.

EtherCAT: 3 slave(s) responding.

EtherCAT: Slave states: PREOP.

EtherCAT: Scanning bus.

EtherCAT: Bus scanning completed in 207 ms.
```

Notice that the EtherCAT master always reports its parent changeset value, so you can always tell without difficulty the version that's running. It will start the master code, bring up the interface with the MAC address in the sysconfig file, and scan the bus for any slaves that might be active before moving to an idle state waiting for a master instance to be requested (more on this later).

5. Using the Command-line Tool

The ethercat-tool above has several useful utilities for getting information about the slaves attached to the bus. Here, I will mention a few of the more useful ones. See the documentation for more information about what commands are available.

One of the first things the user will want to know is that the slaves are connected, and which slaves are connected and in what order; the command-line tool makes this possible in a couple of different ways. The simplest is a listing of the slaves:

```
[dclark@schwarzwald ~]$ ./ethercat-tool slaves

0  0:0  PREOP  +  EK1100 EtherCAT-Koppler (2A E-Bus)

1  0:1  PREOP  +  EL2004 4Ch. Dig. Output 24V, 0.5A

2  0:2  PREOP  +  EL1014 4Ch. Dig. Input 24V, 10µs
```

What is reported here is the bus segment:position for each slave (0:0 through 0:2), the slave's application-layer state, and the slave's model and function. For more complex arrangements, you might want to know about other bus segments and slave connections in a graphical manner. Consider the following:

```
[dclark@schwarzwald ~]$ ./ethercat-tool graph | dot -Tpng > slaves7.png
```

which produces a graphical output of the bus connections, the slaves at each position, and the bus delay time to each slave (next page):

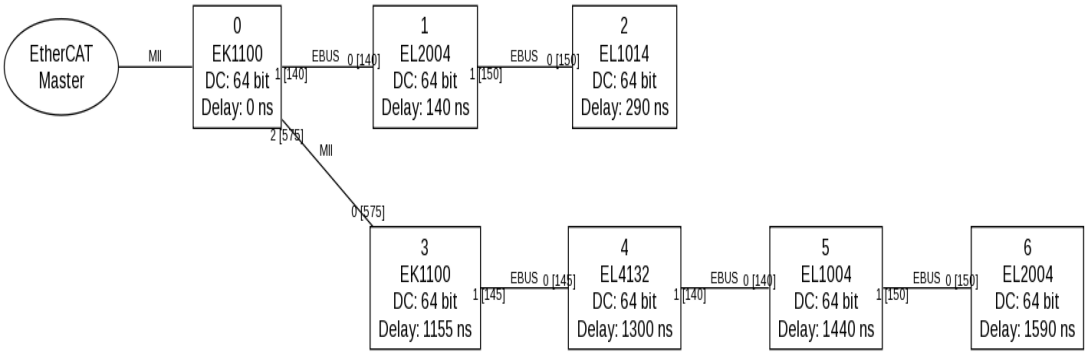


Figure 1. EtherCAT bus graph

Here we see visually the interconnection of the slaves, their position values, slave type, distributed clock (DC) width, and output delay. In this example the bus extension RJ-45 from the bus coupler EK1100 at position 0 was used to connect a second EK110 with 3 slaves attached to it to produce a tree arrangement. EtherCAT allows a mix of topology variations, such as star and ring connections, depending on the needs of the application.

Suppose now that we wanted more specific information about the slaves. Another useful command is:

```
[dc1ark@schwarzwa1d ~]$ ./ethercat-too1 pdos
=== Master 0, S1ave 1 ===
SM0: PhysAddr 0x0f00, Defau1tSize    0, Contro1Register 0x44, Enab1e 9
  RxPDO 0x1600 "Channe1 1"
    PDO entry 0x7000:01,  1 bit, "Output"
  RxPDO 0x1601 "Channe1 2"
    PDO entry 0x7010:01,  1 bit, "Output"
  RxPDO 0x1602 "Channe1 3"
    PDO entry 0x7020:01,  1 bit, "Output"
  RxPDO 0x1603 "Channe1 4"
    PDO entry 0x7030:01,  1 bit, "Output"
=== Master 0, S1ave 2 ===
SM0: PhysAddr 0x1000, Defau1tSize    0, Contro1Register 0x00, Enab1e 0
  TxPDO 0x1a00 "Channe1 1"
    PDO entry 0x6000:01,  1 bit, "Input"
```

```
  TxPDO 0x1a01 "Channel 2"
    PDO entry 0x6010:01,  1 bit, "Input"
  TxPDO 0x1a02 "Channel 3"
    PDO entry 0x6020:01,  1 bit, "Input"
  TxPDO 0x1a03 "Channel 4"
    PDO entry 0x6030:01,  1 bit, "Input"
<snip...>
=== Master 0, Slave 4 ===
SM0: PhysAddr 0x1800, DefaultSize  246, ControlRegister 0x26, Enable 1
SM1: PhysAddr 0x18f6, DefaultSize  246, ControlRegister 0x22, Enable 1
SM2: PhysAddr 0x1000, DefaultSize    4, ControlRegister 0x24, Enable 1
  RxPDO 0x1600 "RxPDO 01 mapping"
    PDO entry 0x3001:01, 16 bit, "Output"
  RxPDO 0x1601 "RxPDO 02 mapping"
    PDO entry 0x3002:01, 16 bit, "Output"
SM3: PhysAddr 0x1100, DefaultSize    0, ControlRegister 0x20, Enable 0
```

The above lists the Process Data Objects (PDO) that are available for each slave. The PDO serves as a dictionary for collecting entries that represent i/o access to the outputs and control registers for the slaves. For slaves that output signals, these PDOs entries are listed as RxPDOs, since (from the view of the slave) they receive data from the master to set their output terminals. Input slaves similarly map input terminals as TxPDOs, since they write frame bits during the data exchange with the master. Access to the slave i/o registers is protected by a sync manager (SM) that acts as a dual-port buffer to manage the transfer of data from the ethercat datagram as it passes through the hardware to the i/o side. Addresses to slaves can be remapped with aliasing; see the EtherCAT pdf documentation for an explanation of this.

There is a similar command, and output data from 'ethercat-tool sdos' to list the Service Data Objects supported by the slaves.

Perhaps the most useful command of all is this one:

```
[clark@schwarzwald ~]$ ./ethercat-tool cstruct > slaves7.h
```

Which produces a C listing of all the slaves' PDO mapping information for inclusion into user-written application software. This saves valuable time looking up information that is needed to construct a complete EtherCAT application by pulling in all the necessary data from the slaves. Armed with this C structure listing, you can develop your application with the correct configuration entries, and use it during runtime to check for changes to the bus. If a slave was added, or failed in service, for example, the application software would know about it as soon as the configuration data no longer matched the physical bus – provided some simple checks are included.

The C output is shown below:

```
/* Master 0, Slave 1, "EL2004"
 * Vendor ID:       0x00000002
 * Product code:    0x07d43052
 * Revision number: 0x00100000
 */
ec_pdo_entry_info_t slave_1_pdo_entries[] = {
   {0x7000, 0x01, 1}, /* Output */
   {0x7010, 0x01, 1}, /* Output */
   {0x7020, 0x01, 1}, /* Output */
   {0x7030, 0x01, 1}, /* Output */
};
ec_pdo_info_t slave_1_pdos[] = {
   {0x1600, 1, slave_1_pdo_entries + 0}, /* Channel 1 */
   {0x1601, 1, slave_1_pdo_entries + 1}, /* Channel 2 */
   {0x1602, 1, slave_1_pdo_entries + 2}, /* Channel 3 */
   {0x1603, 1, slave_1_pdo_entries + 3}, /* Channel 4 */
};
ec_sync_info_t slave_1_syncs[] = {
   {0, EC_DIR_OUTPUT, 4, slave_1_pdos + 0, EC_WD_ENABLE},
};
<snip...>
/* Master 0, Slave 4, "EL4132"
 * Vendor ID:       0x00000002
 * Product code:    0x10243052
 * Revision number: 0x03fa0000
 */
ec_pdo_entry_info_t slave_4_pdo_entries[] = {
   {0x3001, 0x01, 16}, /* Output */
```

```
    {0x3002, 0x01, 16}, /* Output */
};
 ec_pdo_info_t slave_4_pdos[] = {
    {0x1600, 1, slave_4_pdo_entries + 0}, /* RxPDO 01 mapping */
    {0x1601, 1, slave_4_pdo_entries + 1}, /* RxPDO 02 mapping */
};
ec_sync_info_t slave_4_syncs[] = {
    {0, EC_DIR_OUTPUT, 0, NULL, EC_WD_DISABLE},
    {1, EC_DIR_INPUT, 0, NULL, EC_WD_DISABLE},
    {2, EC_DIR_OUTPUT, 2, slave_4_pdos + 0, EC_WD_DISABLE},
    {3, EC_DIR_INPUT, 0, NULL, EC_WD_DISABLE},
};
```

6. A Simple Application Example

The EtherCAT master installation tree includes several examples of both user- and kernel-space applications, including more advanced topics such as distributed clocks. For the purposes of this guide, we are only concerned with user-space applications that don't (necessarily) need precise timing, or to run in a real-time context. Refer to the examples and the doxygen source code documentation for information, if interested.

For building and linking an example program discussed in the pdf documentation, we have the following gcc command, taken from the pdf guide:

```
gcc ethercat . c -o ectest -I / opt / etherlab / include “
    -L / opt / etherlab / lib - lethercat “
    -Wl , - - rpath -Wl ,/ opt / etherlab / lib
```

Make sure you have the -L and -l flags set correctly, or the linker will fail. These are shown in read above.

The lexample using the ethercat library can also be linked statically like so:

```
gcc - static ectest . c -o ectest -I / opt / etherlab / include “
    / opt / etherlab / lib / libethercat
```

The first set of EtherCAT hardware available for testing and development consisted of an EK1100 bus coupler, an EL1014 digital output, and an EL2004 digital output module. Constructing an EtherCAT application requires the following elements:

1. Include “ecrt.h”, the user-space real-time interface library. Other interfaces are available and documented in the *doxygen* html reference.
2. Declaration of all global variables relating to the ethercat master and the application's implementation of the interface to the hardware.

3. Either explicit declaration of the PDOs of the hardware to be used, or applying the PDO configuration data at run time.
4. Requesting a master instance from the master kernel module. You can have more than one master instance that share access to the master module, but locking the resource to prevent conflicts is up to you.
5. Registration of the PDO entries, and collecting the relevant data into domains.
6. Activating the master and filling in the domain data.
7. Applying the configuration specified.
8. Cyclically exchanging the data with the slaves through the domain and relevant calls to the ecrt library.
9. Releasing the master when finished.

We will use the initial test hardware to perform all of the above in an example application. First, let's review how the PDOs, Process Data Domains, and the master are related:

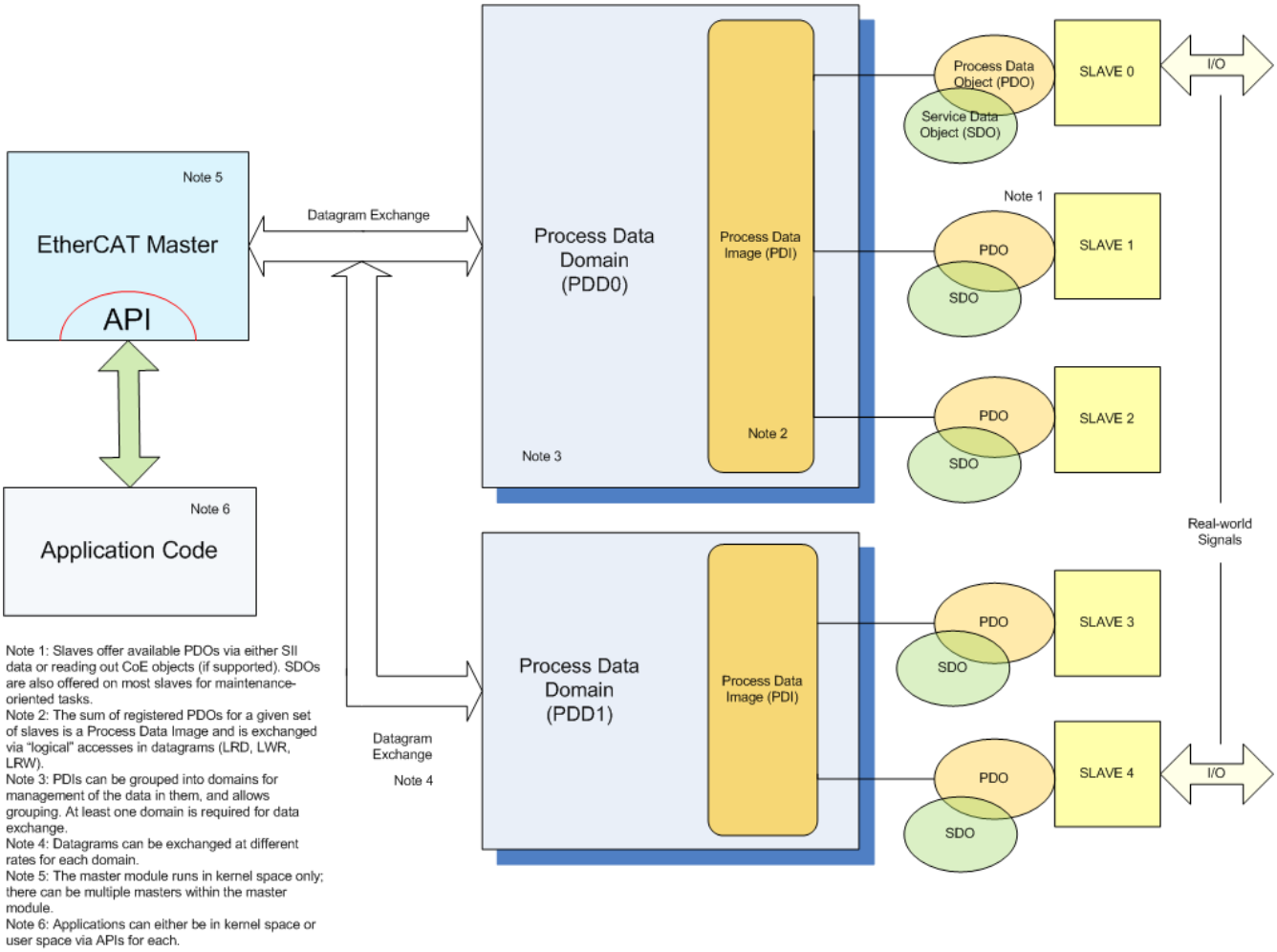


Figure 2. Relationship of slaves and data exchange objects.

PDO data are grouped into a Process Data Image. The sum of all PDOs that make up the image in turn are collected into a Process Data Domain. A master may have more than one, but *must* have at least one, domain configured in order to exchange data. Data domains make it convenient to group data in terms of their access requirements; process data that changes slowly (e.g. thermocouples) may need

access to its domain much less often than a domain that contains a servo loop, for example. So bandwidth in the ethernet link can be preserved by only exchanging datagrams when necessary.

The master handles the work of assembling and disassembling Ethernet frames from the process data image; payloads that require more than one Ethernet frame (1486 bytes) to transmit or receive are handled automatically. Other than registration of the PDO entries into the process data domain, the application need not be concerned with details of accessing data, since the ecrt library exposes functions to handle writing to and reading from the domain.

Here is the simplest possible EtherCAT application to check if you have everything working:

```
#include <stdio.h>
#include <stdlib.h>

#include "ecrt.h"

int main(void) {
  static ec_master_t *master = NULL;

        printf("Starting ecat test.\n");
        master = ecrt_request_master(0);
        if (!master) {
          printf("Failed to get a master instance!\n");
          return EXIT_FAILURE;
        }
        else {
          ecrt_release_master(master);
          printf("Master get/release ok.\n");
          return EXIT_SUCCESS;
        }

}
```

Listing 1. “Hello world” EtherCAT sample.

Here, we merely include the ecrt library and request, then release, a master instance. Not really very useful, is it? Well, how about a more..*involved* example? Below, we have a complete example that reads an input module and copies the input data to an output module. Relevant comments are inserted in the test in RED.

```
#include <stdio.h>
#include <unistd.h>

/****************************************************************************/
//EtherCAT
#include "ecrt.h"  YOU WILL NEED THIS LIBRARY

/****************************************************************************/
// Optional features

#define PRECONFIGURE_PDOS 1  //use configuration in build
#define CONFIGURE_PDOS 0     //configure at run-time


/****************************************************************************/
// EtherCAT
```

```
static ec_master_t *master = NULL;
static ec_master_state_t master_state = {};

static ec_domain_t *domain1 = NULL;
static ec_domain_state_t domain1_state = {};

static ec_slave_config_t *sc_dig_out = NULL;
static ec_slave_config_t *sc_dig_in = NULL;

/****************************************************************************/
#define NUMSLAVES 2
// process data
static uint8_t *domain1_pd = NULL;

HERE WE BEGIN TO DEFINE THE PDO ENTRIES AND WHERE THEY WILL APPEAR IN THE PROCESS
DATA DOMAIN

//definitions for the bus connections and expected slave positions
#define BusCouplerPos  0, 0
#define DigOutSlavePos 0, 1
#define DigInSlavePos  0, 2

#define Beckhoff_EK1100 0x00000002, 0x044c2c52
#define Beckhoff_EL2004 0x00000002, 0x07d43052
#define Beckhoff_EL1014 0x00000002, 0x03f63052

// offsets for PDO entries
static unsigned int off_dig_out;
static unsigned int off_dig_in;

NOTE THAT THE DOMAIN REGISTRATION REQUIRES ONLY THE BASE ADDRESS OF THE I/O ON THE
MODULES (SEE BELOW)
const static ec_pdo_entry_reg_t domain1_regs[] = {
    {DigOutSlavePos, Beckhoff_EL2004, 0x7000, 1, &off_dig_out},
    {DigInSlavePos, Beckhoff_EL1014, 0x6000, 1, &off_dig_in},
    {}
};

static char slaves_up = 0;
/****************************************************************************/

SUPPOSING THAT WE HAD A CONFIGURATION IN HAND, WE CAN SIMPLY INCLUDE IT IN THE
SOURCE, LIKE THIS

#if PRECONFIGURE_PDOS
/* Master 0, Slave 1, "EL2004"
 * Vendor ID:       0x00000002
 * Product code:    0x07d43052
 * Revision number: 0x00100000
 */
BELOW IS THE BASE ADDRESS AND THE BIT OFFSET AND SIZE OF THE DATA
ec_pdo_entry_info_t EL2004_pdo_entries[] = {
   {0x7000, 0x01, 1}, /* Output */
   {0x7010, 0x01, 1}, /* Output */
   {0x7020, 0x01, 1}, /* Output */
   {0x7030, 0x01, 1}, /* Output */
};
THE PDO ENTRY INFO AND PDO ENTRY ENTITIES FORM A TREE
```

```
ec_pdo_info_t EL2004_pdos[] = {
   {0x1600, 1, EL2004_pdo_entries + 0}, /* Channel 1 */
   {0x1601, 1, EL2004_pdo_entries + 1}, /* Channel 2 */
   {0x1602, 1, EL2004_pdo_entries + 2}, /* Channel 3 */
   {0x1603, 1, EL2004_pdo_entries + 3}, /* Channel 4 */
};

ec_sync_info_t EL2004_syncs[] = {
   {0, EC_DIR_OUTPUT, 4, EL2004_pdos + 0, EC_WD_ENABLE},
   {0xff}
};

/* Master 0, Slave 2, "EL1014"
 * Vendor ID:       0x00000002
 * Product code:    0x03f63052
 * Revision number: 0x00100000
 */

ec_pdo_entry_info_t EL1014_pdo_entries[] = {
   {0x6000, 0x01, 1}, /* Input */
   {0x6010, 0x01, 1}, /* Input */
   {0x6020, 0x01, 1}, /* Input */
   {0x6030, 0x01, 1}, /* Input */
};

ec_pdo_info_t EL1014_pdos[] = {
   {0x1a00, 1, EL1014_pdo_entries + 0}, /* Channel 1 */
   {0x1a01, 1, EL1014_pdo_entries + 1}, /* Channel 2 */
   {0x1a02, 1, EL1014_pdo_entries + 2}, /* Channel 3 */
   {0x1a03, 1, EL1014_pdo_entries + 3}, /* Channel 4 */
};

ec_sync_info_t EL1014_syncs[] = {
   {0, EC_DIR_INPUT, 4, EL1014_pdos + 0, EC_WD_DISABLE},
   {0xff}
};

#endif

/*****************************************************************************/
void check_domain1_state(void)
{
    ec_domain_state_t ds;

    ecrt_domain_state(domain1, &ds);
#if 0
    if (ds.working_counter != domain1_state.working_counter)
        printf("Domain1: WC %u.\n", ds.working_counter);
    if (ds.wc_state != domain1_state.wc_state)
        printf("Domain1: State %u.\n", ds.wc_state);
#endif
    domain1_state = ds;
}

/*****************************************************************************/

void check_master_state(void)
{
```

```
    ec_master_state_t ms;

    ecrt_master_state(master, &ms);

    if (ms.slaves_responding != master_state.slaves_responding)
        printf("%u slave(s).\n", ms.slaves_responding);
    if (ms.al_states != master_state.al_states)
        printf("AL states: 0x%02X.\n", ms.al_states);
    if (ms.link_up != master_state.link_up)
        printf("Link is %s.\n", ms.link_up ? "up" : "down");

    master_state = ms;
}

/*****************************************************************************/
void check_slave_config_states()
{
    ec_slave_config_state_t s;
    ecrt_slave_config_state(sc_dig_out, &s);
    if (slaves_up < 1 && s.al_state != 0x08) {
    printf("DigOut: State 0x%02X.\n", s.al_state);
    }
    if (slaves_up < 1 && s.al_state == 0x08) {
      slaves_up = 1;
    }
    ecrt_slave_config_state(sc_dig_in, &s);
    if (slaves_up < 2 && s.al_state != 0x08) {
      printf("DigIn: State 0x%02X.\n", s.al_state);
    }
    if (slaves_up < 2 && s.al_state == 0x08) {
      slaves_up = 2;
    }

}
/*****************************************************************************/
int main(void) //(int argc, char **argv)
{
    unsigned int j, in, out, op_flag;

    ec_slave_config_t *sc;
    ec_master_state_t ms;
FIRST, REQUEST A MASTER INSTANCE
    master = ecrt_request_master(0);
    if (!master)
      { fprintf(stderr, "Unable to get requested master.\n");
        return -1;
      }
THEN, CREATE A DOMAIN
    domain1 = ecrt_master_create_domain(master);
    if (!domain1)
      { fprintf(stderr, "Unable to create process data domain.\n");
        return -1;
      }

#if CONFIGURE_PDOS
    printf("Configuring PDOs...\n");

    if (!(sc = ecrt_master_slave_config(
```

```
                    master, DigOutSlavePos, Beckhoff_EL2004))) {
        fprintf(stderr, "Failed to get EL2004 configuration.\n");
        return -1;
    }

    if (!(sc = ecrt_master_slave_config(
                        master, DigInSlavePos, Beckhoff_EL1014))) {
            fprintf(stderr, "Failed to get EL1014 configuration.\n");
            return -1;
        }

#endif
EITHER CONFIGURE THE PDOS AT RUNTIME (AS ABOVE),OR DO IT WITH THE PRECONFIGURED
INFO (AS BELOW)
    // Create configuration for the bus coupler
    sc = ecrt_master_slave_config(master, BusCouplerPos, Beckhoff_EK1100);
    if (!sc) {
      fprintf(stderr, "Failed to get EK1100 configuration.\n");
      return -1;
    }

    sc_dig_out = ecrt_master_slave_config(
                                master, DigOutSlavePos, Beckhoff_EL2004);

    sc_dig_in = ecrt_master_slave_config(
                            master, DigInSlavePos, Beckhoff_EL1014);

    if (ecrt_domain_reg_pdo_entry_list(domain1, domain1_regs)) {
        fprintf(stderr, "PDO entry registration failed!\n");
        return -1;
    }

ACTIVATE THE MASTER. DO NOT APPLY ANY CONFIGURATION AFTER THIS, IT WON'T WORK
    printf("Activating master...");
        if (ecrt_master_activate(master)) {
          fprintf(stderr,"activation failed.\n");
          return -1;
        }
    printf("ok!\n");
INITIALIZE THE PROCESS DOMAIN MEMORY (FOR USER-SPACE APPS)
    if (!(domain1_pd = ecrt_domain_data(domain1))) {
         fprintf(stderr,"Domain data initialization failed.\n");
         return -1;
       }
       printf("Domain data registered ok.\n");

    check_master_state();
    check_domain1_state();

    in = 0; out = 0; op_flag = 0;
ONCE THE MASTER IS ACTIVATED, THE APP IS IN CHARGE OF EXCHANGING DATA THROUGH
EXPLICIT CALLS TO THE ECRT LIBRARY (DONE IN THE IDLE STATE BY THE MASTER)
#if 1
    for (j=0;j<3000;j++)
      {
        ecrt_master_receive(master); RECEIVE A FRAME
        ecrt_domain_process(domain1); DETERMINE THE DATAGRAM STATES
       // check_slave_config_states();
```

```
        if (!op_flag) {
          check_domain1_state();
        }
#if 0
        if (slaves_up == 2 && !op_flag) {
          printf("All slaves reached OP state at %d cycles.\n", j);
          op_flag = 1;
        }
#endif
        if (domain1_state.wc_state == EC_WC_COMPLETE && !op_flag) {
          printf("Domain is up at %d cycles.\n", j);
          op_flag = 1;
        }

        usleep(1000); WAIT 1mS

        in = EC_READ_U8(domain1_pd + off_dig_in) & 0x0F; READ DATA
        if (in != out) {
          EC_WRITE_U8(domain1_pd + off_dig_out, in);     WRITE DATA
          out = in;
          printf("Input data is now: %x\n", in);
        }

        // send process data
        ecrt_domain_queue(domain1); MARK THE DOMAIN DATA AS READY FOR EXCHANGE
        ecrt_master_send(master);   SEND ALL QUEUED DATAGRAMS

      }

    //dummy to keep abrt from seeing the last datagram (which won't match)
    ecrt_master_receive(master);
    ecrt_domain_process(domain1);
#endif

    printf("...Done. Releasing the master!\n");
    ecrt_release_master(master); RELEASE THE MASTER INSTANCE
    return 0;
}

/****************************************************************************/
```

Listing 2. More complicated than “Hello world” example.

The above is based on one of the examples in the ethercat-default directory. The same code from that example can be reworked into one that uses Linux timers to fire a cyclic task, which again, is responsible for exchanging data:

```
      void cyclic_task()
      {
       static int i;
       static unsigned int in, old_in, cycle_led;

          // receive process data
          ecrt_master_receive(master);
          ecrt_domain_process(domain1);

          if (counter) {
              counter--;
```

```
        } else { // do this at 1 Hz
            counter = FREQUENCY;
            // calculate new process data
           blink++;
        }
        // read process data
        in = EC_READ_U8(domain1_pd + off_dig_in) & 0x0F;
        if (old_in != in) {
          printf("Input is now %x\n", in);
          old_in = in;
        }
        // write process data
        cycle_led = !cycle_led;
        if (cycle_led) {
          blink |= 0x08;
        }
        else {
          blink &= 0x07;
        }
        EC_WRITE_U8(domain1_pd + off_dig_out, blink);

        // send process data
        ecrt_domain_queue(domain1);
        ecrt_master_send(master);
    }
```

Listing 3. Cyclic task based on a Linux timer.

Once again, the process data handling is surrounded by the calls:

```
// receive process data
ecrt_master_receive (master);
ecrt_domain_process (domain1);
```

and

```
// send process data
ecrt_domain_queue(domain1);
ecrt_master_send(master);
```

So any application should do whatever data processing is necessary in between the receive and send sections.

The ecrt_master_activate(...) and ecrt_domain_data(...) calls by themselves, it should be noted, don't apply the bus configuration immediately. The master needs several cyclic exchanges to fully apply all the configuration entries to all the slaves, so you shouldn't expect your application to immediately yield up process data as soon as the cyclic task is running. The “real-time” way to check that the bus is fully configured and data are being exchanged is with a call to ecrt_domain_state(...). If the wc_state member is set to the value EC_WC_COMPLETE, then all slaves are in exchanging data.

7. Reference Information

An archive of the EtherCAT version 1.5 guide, EtherCAT presentations, and other information is available at www. mmto.org/~dclark/Reports/EtherCATdocs for further reading. Additional help is available on the etherlab-users mailing list. Register for the list at the URL:

http://lists.etherlab.org/mailman/listinfo/etherlab-users

Questions posted on the list are often answered by Florian Pose, the main author of the IgH EtherCAT software, and finally, as always, check out the source code documentation in the doxygen html reference.