IgH EtherCAT® Master 1.6.3 Documentation

Dipl.-Ing. (FH) Florian Pose, fp@igh.de
Ingenieurgemeinschaft IgH

Essen, March 13, 2025
Revision 1.6.3-15-gb9b49333

Contents

List of Tables

List of Figures

Conventions

The following typographic conventions are used:

- *Italic face* is used for newly introduced terms and file names.
- `Typewriter face` is used for code examples and command line output.
- **`Bold typewriter face`** is used for user input in command lines.

Data values and addresses are usually specified as hexadecimal values. These are marked in the *C* programming language style with the prefix `0x` (example: `0x88A4`). Unless otherwise noted, address values are specified as byte addresses.

Function names are always printed with parentheses, but without parameters. So, if a function `ecrt_request_master()` has empty parentheses, this shall not imply that it has no parameters.

If shell commands have to be entered, this is marked by a dollar prompt:

```
$
```

Further, if a shell command has to be entered as the superuser, the prompt is a mesh:

```
#
```

1 The IgH EtherCAT Master

This chapter covers some general information about the EtherCAT master.

1.1 Feature Summary

The list below gives a short summary of the master features.

- Designed as a kernel module for Linux from version 2.6 (or newer).
- Implemented according to IEC 61158-12 [2] [3].
- Comes with EtherCAT-capable native drivers for several common Ethernet chips, as well as a generic driver for all chips supported by the Linux kernel.
  - The native drivers operate the hardware without interrupts.
  - Native drivers for additional Ethernet hardware can easily be implemented using the common device interface (see section 4.6) provided by the master module.
  - For any other hardware, the generic driver can be used. It uses the lower layers of the Linux network stack.
- The master module supports multiple EtherCAT masters running in parallel.
- The master code supports any Linux realtime extension through its independent architecture.
  - RTAI [11] (including LXRT via RTDM), ADEOS, RT-Preempt [12], Xenomai (including RTDM), etc.
  - It runs well even without realtime extensions.
- Common "Application Interface" for applications, that want to use EtherCAT functionality (see chapter 3).
- *Domains* are introduced, to allow grouping of process data transfers with different slave groups and task periods.
  - Handling of multiple domains with different task periods.
  - Automatic calculation of process data mapping, FMMU and sync manager configuration within each domain.
- Communication through several finite state machines.

  - Automatic bus scanning after topology changes.
  - Bus monitoring during operation.
  - Automatic reconfiguration of slaves (for example after power failure) during operation.
- Distributed Clocks support (see section 3.5).
  - Configuration of the slave's DC parameters through the application interface.
  - Synchronization (offset and drift compensation) of the distributed slave clocks to the reference clock.
  - Optional synchronization of the reference clock to the master clock or the other way round.
- CANopen over EtherCAT (CoE)
  - SDO upload, download and information service.
  - Slave configuration via SDOs.
  - SDO access from userspace and from the application.
- Ethernet over EtherCAT (EoE)
  - Transparent use of EoE slaves via virtual network interfaces.
  - Natively supports either a switched or a routed EoE network architecture.
- Vendor-specific over EtherCAT (VoE)
  - Communication with vendor-specific mailbox protocols via the API.
- File Access over EtherCAT (FoE)
  - Loading and storing files via the command-line tool.
  - Updating a slave's firmware can be done easily.
- Servo Profile over EtherCAT (SoE)
  - Implemented according to IEC 61800-7 [16].
  - Storing IDN configurations, that are written to the slave during startup.
  - Accessing IDNs via the command-line tool.
  - Accessing IDNs at runtime via the user-space library.
- Userspace command-line-tool "ethercat" (see section 7.1)
  - Detailed information about master, slaves, domains and bus configuration.
  - Setting the master's debug level.
  - Reading/Writing alias addresses.
  - Listing slave configurations.
  - Viewing process data.

  - SDO download/upload; listing SDO dictionaries.
  - Loading and storing files via FoE.
  - SoE IDN access.
  - Access to slave registers.
  - Slave SII (EEPROM) access.
  - Controlling application-layer states.
  - Generation of slave description XML and C-code from existing slaves.
- Seamless system integration though LSB compliance.
  - Master and network device configuration via sysconfig files.
  - Init script for master control.
  - Service file for systemd.
- Virtual read-only network interface for monitoring and debugging purposes.

1.2 License

The master code is released under the terms and conditions of the GNU General Public License (GPL [4]), version 2. Other developers, that want to use EtherCAT with Linux systems, are invited to use the master code or even participate on development.

To allow dynamic linking of userspace application against the master's application interface (see chapter 3), the userspace library (see section 7.2) is licensed under the terms and conditions of the GNU Lesser General Public License (LGPL [5]), version 2.1.

2 Architecture

The EtherCAT master is integrated into the Linux kernel. This was an early design decision, which has been made for several reasons:

- Kernel code has significantly better realtime characteristics, i. e. less latency than userspace code. It was foreseeable, that a fieldbus master has a lot of cyclic work to do. Cyclic work is usually triggered by timer interrupts inside the kernel. The execution delay of a function that processes timer interrupts is less, when it resides in kernelspace, because there is no need of time-consuming context switches to a userspace process.
- It was also foreseeable, that the master code has to directly communicate with the Ethernet hardware. This has to be done in the kernel anyway (through network device drivers), which is one more reason for the master code being in kernelspace.

Figure 2.1 gives a general overview of the master architecture.

The components of the master environment are described below:

Master Module Kernel module containing one or more EtherCAT master instances (see section 2.1), the "Device Interface" (see section 4.6) and the "Application Interface" (see chapter 3).

Device Modules EtherCAT-capable Ethernet device driver modules, that offer their devices to the EtherCAT master via the device interface (see section 4.6). These modified network drivers can handle network devices used for EtherCAT operation and "normal" Ethernet devices in parallel. A master can accept a certain device and then is able to send and receive EtherCAT frames. Ethernet devices declined by the master module are connected to the kernel's network stack as usual.

Application A program that uses the EtherCAT master (usually for cyclic exchange of process data with EtherCAT slaves). These programs are not part of the EtherCAT master code[1], but have to be generated or written by the user. An application can request a master through the application interface (see chapter 3). If this succeeds, it has the control over the master: It can provide a bus configuration and exchange process data. Applications can be kernel modules (that use the kernel application interface directly) or userspace programs, that use the application interface via the EtherCAT library (see section 7.2), or the RTDM library (see section 7.3).

[1] Although there are some examples provided in the *examples/* directory.

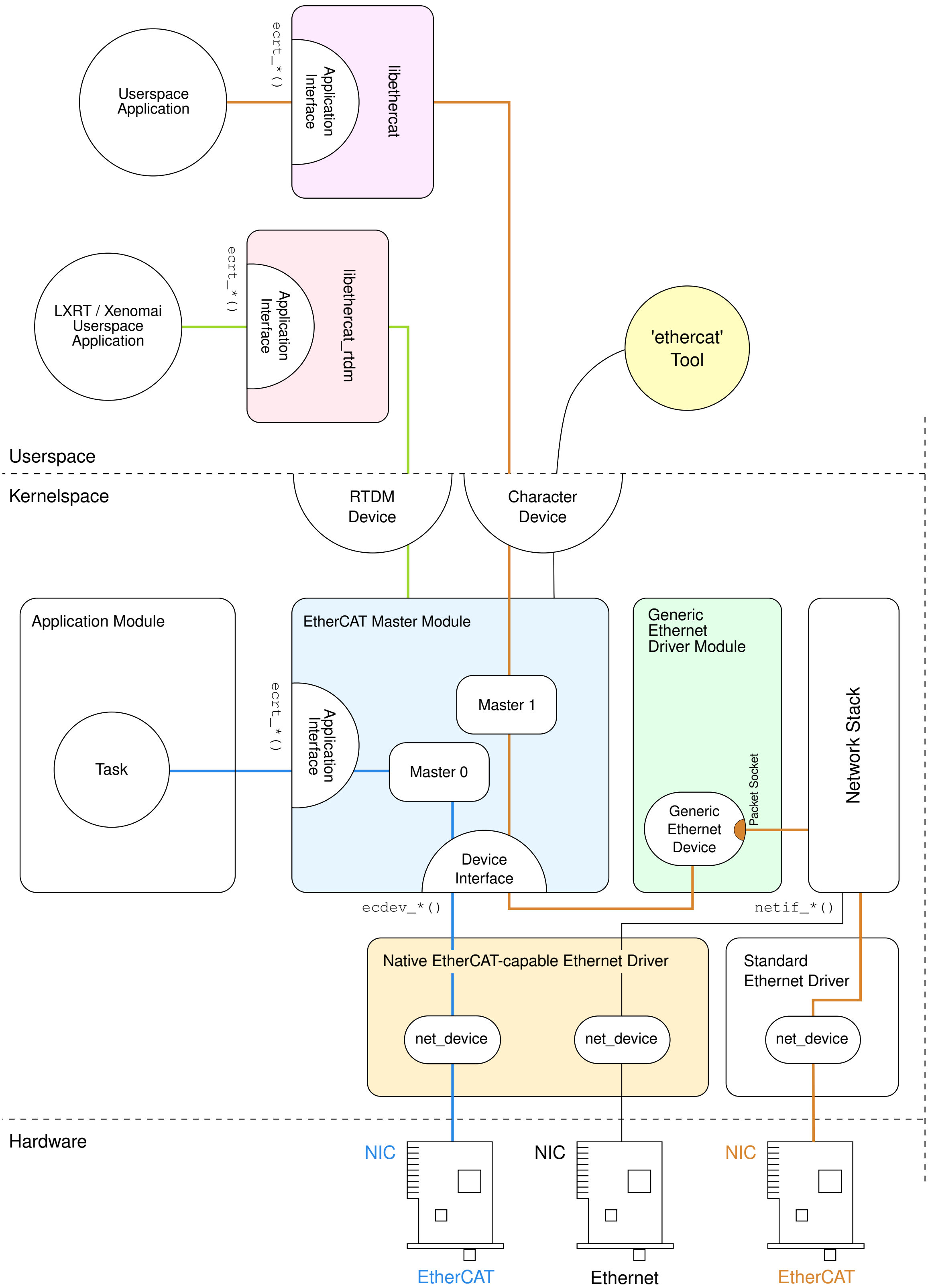

Figure 2.1: Master Architecture

2.1 Master Module

The EtherCAT master kernel module *ec_master* can contain multiple master instances. Each master waits for certain Ethernet device(s) identified by its MAC address(es). These addresses have to be specified on module loading via the *main_devices* (and optional: *backup_devices*) module parameter. The number of master instances to initialize is taken from the number of MAC addresses given.

The below command loads the master module with a single master instance that waits for one Ethernet device with the MAC address `00:0E:0C:DA:A2:20`. The master will be accessible via index 0.

```
# modprobe ec_master main_devices=00:0E:0C:DA:A2:20
```

MAC addresses for multiple masters have to be separated by commas:

```
# modprobe ec_master main_devices=00:0E:0C:DA:A2:20,00:e0:81:71:d5:1c
```

The two masters can be addressed by their indices 0 and 1 respectively (see Figure 2.2). The master index is needed for the `ecrt_request_master()` function of the application interface (see chapter 3) and the `--master` option of the *ethercat* command-line tool (see section 7.1), which defaults to 0.

Debug Level The master module also has a parameter *debug_level* to set the initial debug level for all masters (see also subsection 7.1.7).

Init Script In most cases it is not necessary to load the master module and the Ethernet driver modules manually. There is an init script available, so the master can be started as a service (see section 7.4). For systems that are managed by systemd [7], there is also a service file available.

Syslog The master module outputs information about its state and events to the kernel ring buffer. These also end up in the system logs. The above module loading command should result in the messages below:

```
# dmesg | tail -2
EtherCAT: Master driver 1.6.3
EtherCAT: 2 masters waiting for devices.

# tail -2 /var/log/messages
Jul  4 10:22:45 ethercat kernel: EtherCAT: Master driver 1.6.3
Jul  4 10:22:45 ethercat kernel: EtherCAT: 2 masters waiting
                                 for devices.
```

Master output is prefixed with `EtherCAT` which makes searching the logs easier.

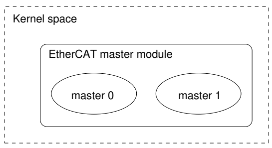

Figure 2.2: Multiple masters in one module

2.2 Master Phases

Every EtherCAT master provided by the master module (see section 2.1) runs through several phases (see Figure 2.3):

Orphaned phase This mode takes effect, when the master still waits for its Ethernet device(s) to connect. No bus communication is possible until then.

Idle phase takes effect when the master has accepted all required Ethernet devices, but is not requested by any application yet. The master runs its state machine (see section 5.3), that automatically scans the bus for slaves and executes pending operations from the userspace interface (for example SDO access). The command-line tool can be used to access the bus, but there is no process data exchange because of the missing bus configuration.

Operation phase The master is requested by an application that can provide a bus configuration and exchange process data.

2.3 Process Data

This section shall introduce a few terms and ideas how the master handles process data.

Process Data Image Slaves offer their inputs and outputs by presenting the master so-called "Process Data Objects" (PDOs). The available PDOs can be either determined by reading out the slave's TxPDO and RxPDO SII categories from the E^{2}PROM (in case of fixed PDOs) or by reading out the appropriate CoE objects (see section 6.2), if available. The application can register the PDOs' entries for exchange during cyclic operation. The sum of all registered PDO entries defines the "process data image", which is exchanged via datagrams with "logical" memory access (like LWR, LRD or LRW) introduced in [2, sec. 5.4].

Process Data Domains The process data image can be easily managed by creating so-called "domains", which allow grouped PDO exchange. They also take care of managing the datagram structures needed to exchange the PDOs. Domains are mandatory for process data exchange, so there has to be at least one. They were introduced for the following reasons:

- The maximum size of a datagram is limited due to the limited size of an Ethernet frame: The maximum data size is the Ethernet data field size minus the EtherCAT frame header, EtherCAT datagram header and EtherCAT datagram footer: $1500 - 2 - 12 - 2 = 1484$ octets. If the size of the process data image exceeds this limit, multiple frames have to be sent, and the image has to be partitioned for the use of multiple datagrams. A domain manages this automatically.

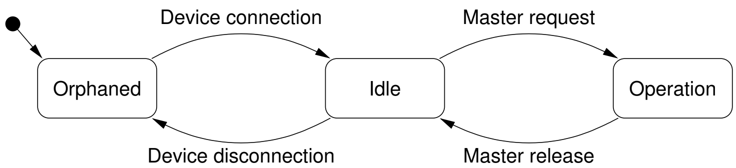

Figure 2.3: Master phases and transitions

- Not every PDO has to be exchanged with the same frequency: The values of PDOs can vary slowly over time (for example temperature values), so exchanging them with a high frequency would just waste bus bandwidth. For this reason, multiple domains can be created, to group different PDOs and so allow separate exchange.

There is no upper limit for the number of domains, but each domain occupies one FMMU in each slave involved, so the maximum number of domains is de facto limited by the slaves.

FMMU Configuration An application can register PDO entries for exchange. Every PDO entry and its parent PDO is part of a memory area in the slave's physical memory, that is protected by a sync manager [2, sec. 6.7] for synchronized access. In order to make a sync manager react on a datagram accessing its memory, it is necessary to access the last byte covered by the sync manager. Otherwise the sync manager will not react on the datagram and no data will be exchanged. That is why the whole synchronized memory area has to be included into the process data image: For example, if a certain PDO entry of a slave is registered for exchange with a certain domain, one FMMU will be configured to map the complete sync-manager-protected memory, the PDO entry resides in. If a second PDO entry of the same slave is registered for process data exchange within the same domain, and it resides in the same sync-manager-protected memory as the first one, the FMMU configuration is not altered, because the desired memory is already part of the domain's process data image. If the second PDO entry would belong to another sync-manager-protected area, this complete area would also be included into the domains process data image.

Figure 2.4 gives an overview, how FMMUs are configured to map physical memory to logical process data images.

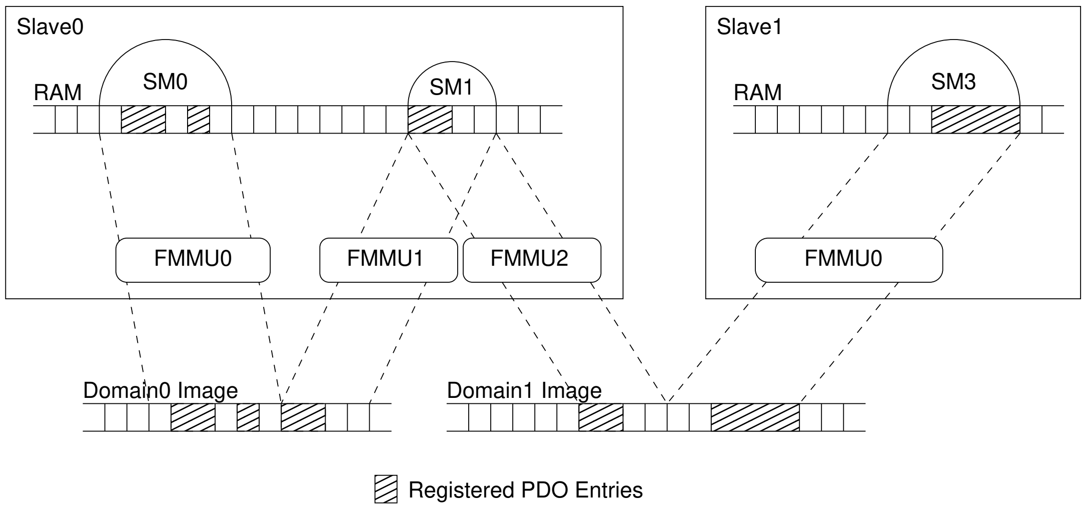

Figure 2.4: FMMU Configuration

3 Application Interface

The application interface provides functions and data structures for applications to access an EtherCAT master. The complete documentation of the interface is included as Doxygen [13] comments in the header file *include/ecrt.h* (see section 3.6). It can either be read directly from the file comments, or as a more comfortable HTML documentation. The HTML generation is described in section 9.3.

The following sections cover a general description of the application interface.

Every application should use the master in two steps:

Configuration The master is requested and the configuration is applied. For example, domains are created, slaves are configured and PDO entries are registered (see section 3.1).

Operation Cyclic code is run and process data are exchanged (see section 3.2).

Example Applications There are a few example applications in the *examples/* subdirectory of the master code. They are documented in the source code.

3.1 Master Configuration

The bus configuration is supplied via the application interface. Figure 3.1 gives an overview of the objects, that can be configured by the application.

3.1.1 Slave Configuration

The application has to tell the master about the expected bus topology. This can be done by creating "slave configurations". A slave configuration can be seen as an expected slave. When a slave configuration is created, the application provides the bus position (see below), vendor id and product code.

When the bus configuration is applied, the master checks, if there is a slave with the given vendor id and product code at the given position. If this is the case, the slave configuration is "attached" to the real slave on the bus and the slave is configured according to the settings provided by the application. The state of a slave configuration can either be queried via the application interface or via the command-line tool (see subsection 7.1.3).

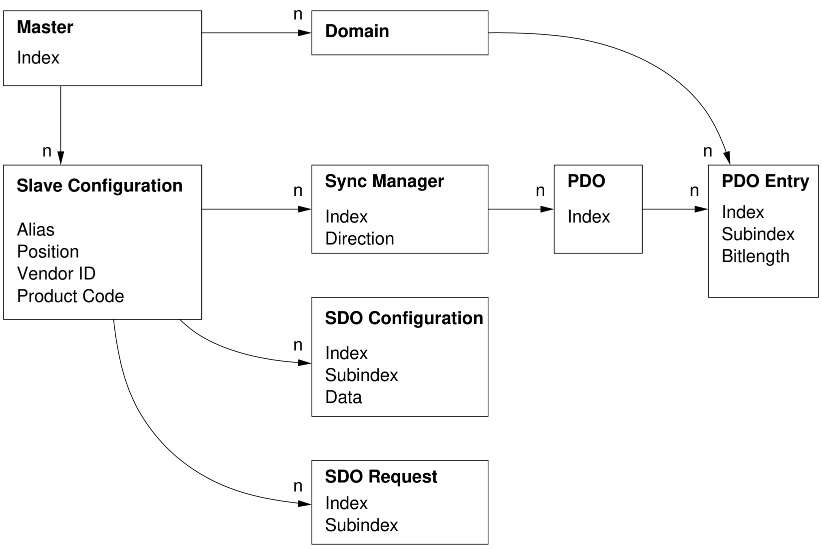

Figure 3.1: Master Configuration

Slave Position The slave position has to be specified as a tuple of "alias" and "position". This allows addressing slaves either via an absolute bus position, or a stored identifier called "alias", or a mixture of both. The alias is a 16-bit value stored in the slave's E^2PROM. It can be modified via the command-line tool (see subsection 7.1.2). Table 3.1 shows, how the values are interpreted.

Table 3.1: Specifying a Slave Position

Alias	Position	Interpretation
`0`	`0 – 65535`	Position addressing. The position parameter is interpreted as the absolute ring position in the bus.
`1 – 65535`	`0 – 65535`	Alias addressing. The position parameter is interpreted as relative position after the first slave with the given alias address.

Figure 3.2 shows an example of how slave configurations are attached. Some of the configurations were attached, while others remain detached. The below lists gives the reasons beginning with the top slave configuration.

1. A zero alias means to use simple position addressing. Slave 1 exists and vendor id and product code match the expected values.
2. Although the slave with position 0 is found, the product code does not match, so the configuration is not attached.
3. The alias is non-zero, so alias addressing is used. Slave 2 is the first slave with alias `0x2000`. Because the position value is zero, the same slave is used.
4. There is no slave with the given alias, so the configuration can not be attached.
5. Slave 2 is again the first slave with the alias `0x2000`, but position is now 1, so slave 3 is attached.

If the master sources are configured with `--enable-wildcards`, then `0xffffffff` matches every vendor ID and/or product code.

3.2 Cyclic Operation

To enter cyclic operation mode, the master has to be "activated" to calculate the process data image and apply the bus configuration for the first time. After activation, the application is in charge to send and receive frames. The configuration can not be changed after activation.

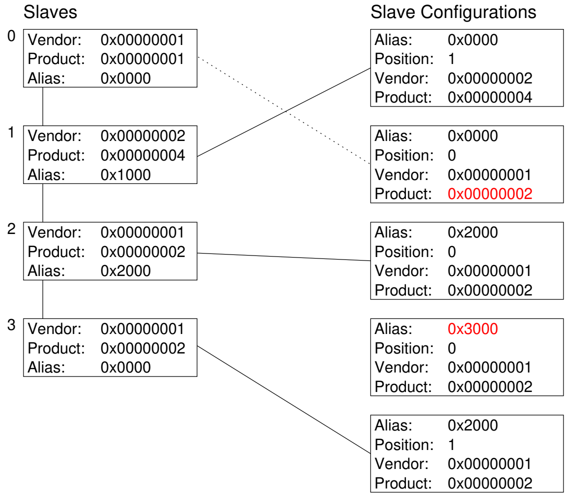

Figure 3.2: Slave Configuration Attachment

3.3 VoE Handlers

During the configuration phase, the application can create handlers for the VoE mailbox protocol described in section 6.3. One VoE handler always belongs to a certain slave configuration, so the creation function is a method of the slave configuration.

A VoE handler manages the VoE data and the datagram used to transmit and receive VoE messages. Is contains the state machine necessary to transfer VoE messages.

The VoE state machine can only process one operation at a time. As a result, either a read or write operation may be issued at a time[1]. After the operation is initiated, the handler must be executed cyclically until it is finished. After that, the results of the operation can be retrieved.

A VoE handler has an own datagram structure, that is marked for exchange after each execution step. So the application can decide, how many handlers to execute before sending the corresponding EtherCAT frame(s).

For more information about the use of VoE handlers see the documentation of the application interface functions and the example applications provided in the *examples/* directory.

3.4 Concurrent Master Access

In some cases, one master is used by several instances, for example when an application does cyclic process data exchange, and there are EoE-capable slaves that require to exchange Ethernet data with the kernel (see section 6.1). For this reason, the master is a shared resource, and access to it has to be sequentialized. This is usually done by locking with semaphores, or other methods to protect critical sections.

The master itself can not provide locking mechanisms, because it has no chance to know the appropriate kind of lock. For example if the application is in kernelspace and uses RTAI functionality, ordinary kernel semaphores would not be sufficient. For that, an important design decision was made: The application that reserved a master must have the total control, therefore it has to take responsibility for providing the appropriate locking mechanisms. If another instance wants to access the master, it has to request the bus access via callbacks, that have to be provided by the application. Moreover the application can deny access to the master if it considers it to be awkward at the moment.

Figure 3.3 exemplary shows, how two processes share one master: The application's cyclic task uses the master for process data exchange, while the master-internal EoE process uses it to communicate with EoE-capable slaves. Both have to access the bus from time to time, but the EoE process does this by "asking" the application to do the bus access for it. In this way, the application can use the appropriate locking

[1] If simultaneous sending and receiving is desired, two VoE handlers can be created for the slave configuration.

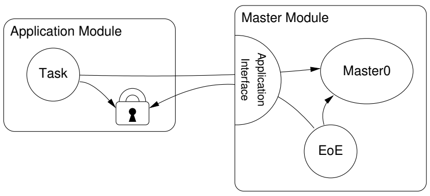

Figure 3.3: Concurrent Master Access

mechanism to avoid accessing the bus at the same time. See the application interface documentation (chapter 3) for how to use these callbacks.

3.5 Distributed Clocks

From version 1.5, the master supports EtherCAT's "Distributed Clocks" feature. It is possible to synchronize the slave clocks on the bus to the "reference clock" (which is the local clock of the first slave with DC support) and to synchronize the reference clock to the "master clock" (which is the local clock of the master). All other clocks on the bus (after the reference clock) are considered as "slave clocks" (see Figure 3.4).

Local Clocks Any EtherCAT slave that supports DC has a local clock register with nanosecond resolution. If the slave is powered, the clock starts from zero, meaning that when slaves are powered on at different times, their clocks will have different values. These "offsets" have to be compensated by the distributed clocks mechanism. On the other hand, the clocks do not run exactly with the same speed, since the used quarts units have a natural frequency deviation. This deviation is usually very small, but over longer periods, the error would accumulate and the difference between local clocks would grow. This clock "drift" has also to be compensated by the DC mechanism.

Application Time The common time base for the bus has to be provided by the application. This application time t_{app} is used

1. to configure the slaves' clock offsets (see below),
2. to program the slave's start times for sync pulse generation (see below).
3. to synchronize the reference clock to the master clock (optional).

Offset Compensation For the offset compensation, each slave provides a "System Time Offset" register t_{off}, that is added to the internal clock value t_{int} to get the "System Time" t_{sys}:

$$\begin{aligned} t_{\text{sys}} &= t_{\text{int}} + t_{\text{off}} \\ \Rightarrow t_{\text{int}} &= t_{\text{sys}} - t_{\text{off}} \end{aligned} \tag{3.1}$$

The master reads the values of both registers to calculate a new system time offset in a way, that the resulting system time shall match the master's application time t_{app}:

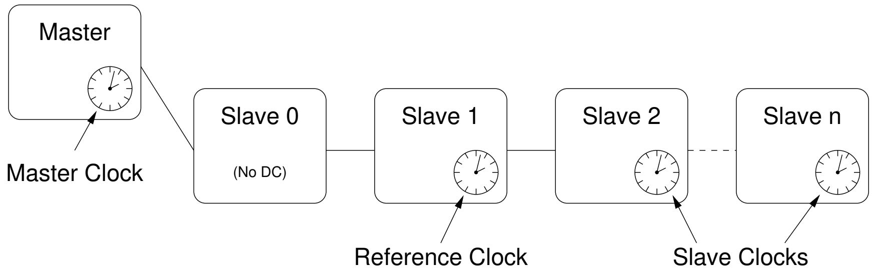

Figure 3.4: Distributed Clocks

$$
\begin{aligned}
t_{\text{sys}} &\stackrel{!}{=} t_{\text{app}} && (3.2)\\
\Rightarrow t_{\text{int}} + t_{\text{off}} &\stackrel{!}{=} t_{\text{app}}\\
\Rightarrow t_{\text{off}} &= t_{\text{app}} - t_{\text{int}}\\
\Rightarrow t_{\text{off}} &= t_{\text{app}} - (t_{\text{sys}} - t_{\text{off}})\\
\Rightarrow t_{\text{off}} &= t_{\text{app}} - t_{\text{sys}} + t_{\text{off}} && (3.3)
\end{aligned}
$$

The small time offset error resulting from the different times of reading and writing the registers will be compensated by the drift compensation.

Drift Compensation The drift compensation is possible due to a special mechanism in each DC-capable slave: A write operation to the "System time" register will cause the internal time control loop to compare the written time (minus the programmed transmission delay, see below) to the current system time. The calculated time error will be used as an input to the time controller, that will tune the local clock speed to be a little faster or slower[2], according to the sign of the error.

Transmission Delays The Ethernet frame needs a small amount of time to get from slave to slave. The resulting transmission delay times accumulate on the bus and can reach microsecond magnitude and thus have to be considered during the drift compensation. EtherCAT slaves supporting DC provide a mechanism to measure the transmission delays: For each of the four slave ports there is a receive time register. A write operation to the receive time register of port 0 starts the measuring and the current system time is latched and stored in a receive time register once the frame is received on the corresponding port. The master can read out the relative receive times, then calculate time delays between the slaves (using its knowledge of the bus topology), and finally calculate the time delays from the reference clock to each slave. These values are programmed into the slaves' transmission delay registers. In this way, the drift compensation can reach nanosecond synchrony.

Checking Synchrony DC-capable slaves provide the 32-bit "System time difference" register at address `0x092c`, where the system time difference of the last drift compensation is stored in nanosecond resolution and in sign-and-magnitude coding[3]. To check for bus synchrony, the system time difference registers can also be cyclically read via the command-line-tool (see subsection 7.1.16):

```
$ watch -n0 "ethercat reg_read -p4 -tsm32 0x92c"
```

[2] The local slave clock will be incremented either with 9 ns, 10 ns or 11 ns every 10 ns.
[3] This allows broadcast-reading all system time difference registers on the bus to get an upper approximation

Sync Signals Synchronous clocks are only the prerequisite for synchronous events on the bus. Each slave with DC support provides two "sync signals", that can be programmed to create events, that will for example cause the slave application to latch its inputs on a certain time. A sync event can either be generated once or cyclically, depending on what makes sense for the slave application. Programming the sync signals is a matter of setting the so-called "AssignActivate" word and the sync signals' cycle- and shift times. The AssignActivate word is slave-specific and has to be taken from the XML slave description (`Device` → `Dc`), where also typical sync signal configurations "OpModes" can be found.

3.6 Application Interface Header

The application interface of the EtherCAT master is defined in the header file *include/ecrt.h* (acronym for "EtherCAT Real-Time") which is listed in this section. The calling conventions of all methods are documented in the comments of this header. There is also a Doxygen-generated [13] online version at https://docs.etherlab.org.

Listing 3.1: Application Interface Header ecrt.h

```
/******************************************************************************
 *
 *  Copyright (C) 2006-2024  Florian Pose, Ingenieurgemeinschaft IgH
 *
 *  This file is part of the IgH EtherCAT master userspace library.
 *
 *  The IgH EtherCAT master userspace library is free software; you can
 *  redistribute it and/or modify it under the terms of the GNU Lesser General
 *  Public License as published by the Free Software Foundation; version 2.1
 *  of the License.
 *
 *  The IgH EtherCAT master userspace library is distributed in the hope that
 *  it will be useful, but WITHOUT ANY WARRANTY; without even the implied
 *  warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public License
 *  along with the IgH EtherCAT master userspace library. If not, see
 *  <http://www.gnu.org/licenses/>.
 *
 *****************************************************************************/

/** \file
 *
 * EtherCAT master application interface.
 *
 * \defgroup ApplicationInterface EtherCAT Application Interface
 *
 * EtherCAT interface for realtime applications. This interface is designed
 * for realtime modules that want to use EtherCAT. There are functions to
 * request a master, to map process data, to communicate with slaves via CoE
 * and to configure and activate the bus.
 *
 *
 * Changes in version 1.6.0:
 *
```

```
 * - Added the ecrt_master_scan_progress() method, the
 *   ec_master_scan_progress_t structure and the EC_HAVE_SCAN_PROGRESS
 *   definition to check for its existence.
 * - Added the EoE configuration methods ecrt_slave_config_eoe_mac_address(),
 *   ecrt_slave_config_eoe_ip_address(), ecrt_slave_config_eoe_subnet_mask(),
 *   ecrt_slave_config_eoe_default_gateway(),
 *   ecrt_slave_config_eoe_dns_address(),
 *   ecrt_slave_config_eoe_hostname() and the EC_HAVE_SET_IP
 *   definition to check for its existence.
 * - Added ecrt_slave_config_state_timeout() to set the application-layer
 *   state change timeout and EC_HAVE_STATE_TIMEOUT to check for its
 *   existence.
 *
 * Changes since version 1.5.2:
 *
 * - Added the ecrt_slave_config_flag() method and the EC_HAVE_FLAGS
 *   definition to check for its existence.
 * - Added SoE IDN requests, including the datatype ec_soe_request_t and the
 *   methods ecrt_slave_config_create_soe_request(),
 *   ecrt_soe_request_object(), ecrt_soe_request_timeout(),
 *   ecrt_soe_request_data(), ecrt_soe_request_data_size(),
 *   ecrt_soe_request_state(), ecrt_soe_request_write() and
 *   ecrt_soe_request_read(). Use the EC_HAVE_SOE_REQUESTS to check, if the
 *   functionality is available.
 *
 * Changes in version 1.5.2:
 *
 * - Added redundancy_active flag to ec_domain_state_t.
 * - Added ecrt_master_link_state() method and ec_master_link_state_t to query
 *   the state of a redundant link.
 * - Added the EC_HAVE_REDUNDANCY define, to check, if the interface contains
 *   redundancy features.
 * - Added ecrt_sdo_request_index() to change SDO index and subindex after
 *   request creation.
 * - Added interface for retrieving CoE emergency messages, i. e.
 *   ecrt_slave_config_emerg_size(), ecrt_slave_config_emerg_pop(),
 *   ecrt_slave_config_emerg_clear(), ecrt_slave_config_emerg_overruns() and
 *   the defines EC_HAVE_EMERGENCY and EC_COE_EMERGENCY_MSG_SIZE.
 * - Added interface for direct EtherCAT register access: Added data type
 *   ec_reg_request_t and methods ecrt_slave_config_create_reg_request(),
 *   ecrt_reg_request_data(), ecrt_reg_request_state(),
 *   ecrt_reg_request_write(), ecrt_reg_request_read() and the feature flag
 *   EC_HAVE_REG_ACCESS.
 * - Added method to select the reference clock,
 *   ecrt_master_select_reference_clock() and the feature flag
 *   EC_HAVE_SELECT_REF_CLOCK to check, if the method is available.
 * - Added method to get the reference clock time,
 *   ecrt_master_reference_clock_time() and the feature flag
 *   EC_HAVE_REF_CLOCK_TIME to have the possibility to synchronize the master
 *   clock to the reference clock.
 * - Changed the data types of the shift times in ecrt_slave_config_dc() to
 *   int32_t to correctly display negative shift times.
 * - Added ecrt_slave_config_reg_pdo_entry_pos() and the feature flag
 *   EC_HAVE_REG_BY_POS for registering PDO entries with non-unique indices
 *   via their positions in the mapping.
 *
 * Changes in version 1.5:
 *
 * - Added the distributed clocks feature and the respective method
 *   ecrt_slave_config_dc() to configure a slave for cyclic operation, and
 *   ecrt_master_application_time(), ecrt_master_sync_reference_clock() and
 *   ecrt_master_sync_slave_clocks() for offset and drift compensation. The
 *   EC_TIMEVAL2NANO() macro can be used for epoch time conversion, while the
 *   ecrt_master_sync_monitor_queue() and ecrt_master_sync_monitor_process()
 *   methods can be used to monitor the synchrony.
 * - Improved the callback mechanism. ecrt_master_callbacks() now takes two
```

```
 *   callback functions for sending and receiving datagrams.
 *   ecrt_master_send_ext() is used to execute the sending of non-application
 *   datagrams.
 * - Added watchdog configuration (method ecrt_slave_config_watchdog(),
 *   #ec_watchdog_mode_t, \a watchdog_mode parameter in ec_sync_info_t and
 *   ecrt_slave_config_sync_manager()).
 * - Added ecrt_slave_config_complete_sdo() method to download an SDO during
 *   configuration via CompleteAccess.
 * - Added ecrt_master_deactivate() to remove the master configuration.
 * - Added ecrt_open_master() and ecrt_master_reserve() separation for
 *   userspace.
 * - Added master information interface (methods ecrt_master(),
 *   ecrt_master_get_slave(), ecrt_master_get_sync_manager(),
 *   ecrt_master_get_pdo() and ecrt_master_get_pdo_entry()) to get information
 *   about the currently connected slaves and the PDO entries provided.
 * - Added ecrt_master_sdo_download(), ecrt_master_sdo_download_complete() and
 *   ecrt_master_sdo_upload() methods to let an application transfer SDOs
 *   before activating the master.
 * - Changed the meaning of the negative return values of
 *   ecrt_slave_config_reg_pdo_entry() and ecrt_slave_config_sdo*().
 * - Implemented the Vendor-specific over EtherCAT mailbox protocol. See
 *   ecrt_slave_config_create_voe_handler().
 * - Renamed ec_sdo_request_state_t to #ec_request_state_t, because it is also
 *   used by VoE handlers.
 * - Removed 'const' from argument of ecrt_sdo_request_state(), because the
 *   userspace library has to modify object internals.
 * - Added 64-bit data access macros.
 * - Added ecrt_slave_config_idn() method for storing SoE IDN configurations,
 *   and ecrt_master_read_idn() and ecrt_master_write_idn() to read/write IDNs
 *   ad-hoc via the user-space library.
 * - Added ecrt_master_reset() to initiate retrying to configure slaves.
 *
 * @{
 */

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

#ifndef __ECRT_H__
#define __ECRT_H__

#ifdef __KERNEL__
#include <asm/byteorder.h>
#include <linux/types.h>
#include <linux/time.h>
#include <linux/in.h> // struct in_addr
#else
#include <stdlib.h> // for size_t
#include <stdint.h>
#include <sys/time.h> // for struct timeval
#include <netinet/in.h> // struct in_addr
#endif

/******************************************************************************
 * Global definitions
 *****************************************************************************/

/** EtherCAT realtime interface major version number.
 */
#define ECRT_VER_MAJOR 1

/** EtherCAT realtime interface minor version number.
 */
#define ECRT_VER_MINOR 6

/** EtherCAT realtime interface version word generator.
 */
```

```
#define ECRT_VERSION(a, b) (((a) << 8) + (b))

/** EtherCAT realtime interface version word.
 */
#define ECRT_VERSION_MAGIC ECRT_VERSION(ECRT_VER_MAJOR, ECRT_VER_MINOR)

/******************************************************************************
 * Feature flags
 *****************************************************************************/

/** Defined, if the redundancy features are available.
 *
 * I. e. if the \a redundancy_active flag in ec_domain_state_t and the
 * ecrt_master_link_state() method are available.
 */
#define EC_HAVE_REDUNDANCY

/** Defined, if the CoE emergency ring feature is available.
 *
 * I. e. if the ecrt_slave_config_emerg_*() methods are available.
 */
#define EC_HAVE_EMERGENCY

/** Defined, if the register access interface is available.
 *
 * I. e. if the methods ecrt_slave_config_create_reg_request(),
 * ecrt_reg_request_data(), ecrt_reg_request_state(), ecrt_reg_request_write()
 * and ecrt_reg_request_read() are available.
 */
#define EC_HAVE_REG_ACCESS

/** Defined if the method ecrt_master_select_reference_clock() is available.
 */
#define EC_HAVE_SELECT_REF_CLOCK

/** Defined if the method ecrt_master_reference_clock_time() is available.
 */
#define EC_HAVE_REF_CLOCK_TIME

/** Defined if the method ecrt_slave_config_reg_pdo_entry_pos() is available.
 */
#define EC_HAVE_REG_BY_POS

/** Defined if the method ecrt_master_sync_reference_clock_to() is available.
 */
#define EC_HAVE_SYNC_TO

/** Defined if the method ecrt_slave_config_flag() is available.
 */
#define EC_HAVE_FLAGS

/** Defined if the methods ecrt_slave_config_create_soe_request(),
 * ecrt_soe_request_object(), ecrt_soe_request_timeout(),
 * ecrt_soe_request_data(), ecrt_soe_request_data_size(),
 * ecrt_soe_request_state(), ecrt_soe_request_write() and
 * ecrt_soe_request_read() and the datatype ec_soe_request_t are available.
 */
#define EC_HAVE_SOE_REQUESTS

/** Defined, if the method ecrt_master_scan_progress() and the
 * ec_master_scan_progress_t structure are available.
 */
#define EC_HAVE_SCAN_PROGRESS

/** Defined, if the methods ecrt_slave_config_eoe_mac_address(),
 * ecrt_slave_config_eoe_ip_address(), ecrt_slave_config_eoe_subnet_mask(),
```

```
 * ecrt_slave_config_eoe_default_gateway(),
 * ecrt_slave_config_eoe_dns_address(), ecrt_slave_config_eoe_hostname() are
 * available.
 */
#define EC_HAVE_SET_IP

/** Defined, if the method ecrt_slave_config_state_timeout() is available.
 */
#define EC_HAVE_STATE_TIMEOUT

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

/** Symbol visibility control macro.
 */
#ifndef EC_PUBLIC_API
# if defined(ethercat_EXPORTS) && !defined(__KERNEL__)
#  define EC_PUBLIC_API __attribute__ ((visibility ("default")))
# else
#  define EC_PUBLIC_API
# endif
#endif

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

/** End of list marker.
 *
 * This can be used with ecrt_slave_config_pdos().
 */
#define EC_END ~0U

/** Maximum number of sync managers per slave.
 */
#define EC_MAX_SYNC_MANAGERS 16

/** Maximum string length.
 *
 * Used in ec_slave_info_t.
 */
#define EC_MAX_STRING_LENGTH 64

/** Maximum number of slave ports. */
#define EC_MAX_PORTS 4

/** Timeval to nanoseconds conversion.
 *
 * This macro converts a Unix epoch time to EtherCAT DC time.
 *
 * \see void ecrt_master_application_time()
 *
 * \param TV struct timeval containing epoch time.
 */
#define EC_TIMEVAL2NANO(TV) \
    (((TV).tv_sec - 946684800ULL) * 1000000000ULL + (TV).tv_usec * 1000ULL)

/** Size of a CoE emergency message in byte.
 *
 * \see ecrt_slave_config_emerg_pop().
 */
#define EC_COE_EMERGENCY_MSG_SIZE 8

/******************************************************************************
 * Data types
 *****************************************************************************/

struct ec_master;
typedef struct ec_master ec_master_t; /**< \see ec_master */
```

```
struct ec_slave_config;
typedef struct ec_slave_config ec_slave_config_t; /**< \see ec_slave_config */

struct ec_domain;
typedef struct ec_domain ec_domain_t; /**< \see ec_domain */

struct ec_sdo_request;
typedef struct ec_sdo_request ec_sdo_request_t; /**< \see ec_sdo_request. */

struct ec_soe_request;
typedef struct ec_soe_request ec_soe_request_t; /**< \see ec_soe_request. */

struct ec_voe_handler;
typedef struct ec_voe_handler ec_voe_handler_t; /**< \see ec_voe_handler. */

struct ec_reg_request;
typedef struct ec_reg_request ec_reg_request_t; /**< \see ec_reg_request. */

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

/** Master state.
 *
 * This is used for the output parameter of ecrt_master_state().
 *
 * \see ecrt_master_state().
 */
typedef struct {
    unsigned int slaves_responding; /**< Sum of responding slaves on all
                                      Ethernet devices. */
    unsigned int al_states : 4; /**< Application-layer states of all slaves.
                                  The states are coded in the lower 4 bits.
                                  If a bit is set, it means that at least one
                                  slave in the network is in the corresponding
                                  state:
                                  - Bit 0: \a INIT
                                  - Bit 1: \a PREOP
                                  - Bit 2: \a SAFEOP
                                  - Bit 3: \a OP */
    unsigned int link_up : 1; /**< \a true, if at least one Ethernet link is
                                up. */
} ec_master_state_t;

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

/** Redundant link state.
 *
 * This is used for the output parameter of ecrt_master_link_state().
 *
 * \see ecrt_master_link_state().
 */
typedef struct {
    unsigned int slaves_responding; /**< Sum of responding slaves on the given
                                      link. */
    unsigned int al_states : 4; /**< Application-layer states of the slaves on
                                  the given link. The states are coded in the
                                  lower 4 bits. If a bit is set, it means
                                  that at least one slave in the network is in
                                  the corresponding state:
                                  - Bit 0: \a INIT
                                  - Bit 1: \a PREOP
                                  - Bit 2: \a SAFEOP
                                  - Bit 3: \a OP */
    unsigned int link_up : 1; /**< \a true, if the given Ethernet link is up.
                               */
} ec_master_link_state_t;
```

```

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

/** Slave configuration state.
 *
 * This is used as an output parameter of ecrt_slave_config_state().
 *
 * \see ecrt_slave_config_state().
 */
typedef struct  {
    unsigned int online : 1; /**< The slave is online. */
    unsigned int operational : 1; /**< The slave was brought into \a OP state
                                    using the specified configuration. */
    unsigned int al_state : 4; /**< The application-layer state of the slave.
                                 - 1: \a INIT
                                 - 2: \a PREOP
                                 - 4: \a SAFEOP
                                 - 8: \a OP

                                 Note that each state is coded in a different
                                 bit! */
} ec_slave_config_state_t;

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

/** Master information.
 *
 * This is used as an output parameter of ecrt_master().
 *
 * \see ecrt_master().
 */
typedef struct {
    unsigned int slave_count; /**< Number of slaves in the network. */
    unsigned int link_up : 1; /**< \a true, if the network link is up. */
    uint8_t scan_busy; /**< \a true, while the master is scanning the network.
                         */
    uint64_t app_time; /**< Application time. */
} ec_master_info_t;

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

/** Master scan progress information.
 *
 * This is used as an output parameter of ecrt_master_scan_progress().
 *
 * \see ecrt_master_scan_progress().
 */
typedef struct {
    unsigned int slave_count; /**< Number of slaves detected. */
    unsigned int scan_index; /**< Index of the slave that is currently
                               scanned.  If it is less than the \a
                               slave_count, the network scan is in progress.
                             */
} ec_master_scan_progress_t;

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

/** EtherCAT slave port descriptor.
 */
typedef enum {
    EC_PORT_NOT_IMPLEMENTED, /**< Port is not implemented. */
    EC_PORT_NOT_CONFIGURED, /**< Port is not configured. */
    EC_PORT_EBUS, /**< Port is an E-Bus. */
    EC_PORT_MII /**< Port is a MII. */
} ec_slave_port_desc_t;

```

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

/** EtherCAT slave port information.
 */
typedef struct {
    uint8_t link_up; /**< Link detected. */
    uint8_t loop_closed; /**< Loop closed. */
    uint8_t signal_detected; /**< Detected signal on RX port. */
} ec_slave_port_link_t;

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

/** Slave information.
 *
 * This is used as an output parameter of ecrt_master_get_slave().
 *
 * \see ecrt_master_get_slave().
 */
typedef struct {
    uint16_t position; /**< Offset of the slave in the ring. */
    uint32_t vendor_id; /**< Vendor-ID stored on the slave. */
    uint32_t product_code; /**< Product-Code stored on the slave. */
    uint32_t revision_number; /**< Revision-Number stored on the slave. */
    uint32_t serial_number; /**< Serial-Number stored on the slave. */
    uint16_t alias; /**< The slaves alias if not equal to 0. */
    int16_t current_on_ebus; /**< Used current in mA. */
    struct {
        ec_slave_port_desc_t desc; /**< Physical port type. */
        ec_slave_port_link_t link; /**< Port link state. */
        uint32_t receive_time; /**< Receive time on DC transmission delay
                                 measurement. */
        uint16_t next_slave; /**< Ring position of next DC slave on that
                               port. */
        uint32_t delay_to_next_dc; /**< Delay [ns] to next DC slave. */
    } ports[EC_MAX_PORTS]; /**< Port information. */
    uint8_t al_state; /**< Current state of the slave. */
    uint8_t error_flag; /**< Error flag for that slave. */
    uint8_t sync_count; /**< Number of sync managers. */
    uint16_t sdo_count; /**< Number of SDOs. */
    char name[EC_MAX_STRING_LENGTH]; /**< Name of the slave. */
} ec_slave_info_t;

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

/** Domain working counter interpretation.
 *
 * This is used in ec_domain_state_t.
 */
typedef enum {
    EC_WC_ZERO = 0,   /**< No registered process data were exchanged. */
    EC_WC_INCOMPLETE, /**< Some of the registered process data were
                        exchanged. */
    EC_WC_COMPLETE    /**< All registered process data were exchanged. */
} ec_wc_state_t;

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

/** Domain state.
 *
 * This is used for the output parameter of ecrt_domain_state().
 */
typedef struct {
    unsigned int working_counter; /**< Value of the last working counter. */
    ec_wc_state_t wc_state; /**< Working counter interpretation. */
    unsigned int redundancy_active; /**< Redundant link is in use. */
} ec_domain_state_t;
```

```

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

/** Direction type for PDO assignment functions.
 */
typedef enum {
    EC_DIR_INVALID, /**< Invalid direction. Do not use this value. */
    EC_DIR_OUTPUT, /**< Values written by the master. */
    EC_DIR_INPUT, /**< Values read by the master. */
    EC_DIR_COUNT /**< Number of directions. For internal use only. */
} ec_direction_t;

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

/** Watchdog mode for sync manager configuration.
 *
 * Used to specify, if a sync manager's watchdog is to be enabled.
 */
typedef enum {
    EC_WD_DEFAULT, /**< Use the default setting of the sync manager. */
    EC_WD_ENABLE, /**< Enable the watchdog. */
    EC_WD_DISABLE, /**< Disable the watchdog. */
} ec_watchdog_mode_t;

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

/** PDO entry configuration information.
 *
 * This is the data type of the \a entries field in ec_pdo_info_t.
 *
 * \see ecrt_slave_config_pdos().
 */
typedef struct {
    uint16_t index; /**< PDO entry index. */
    uint8_t subindex; /**< PDO entry subindex. */
    uint8_t bit_length; /**< Size of the PDO entry in bit. */
} ec_pdo_entry_info_t;

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

/** PDO configuration information.
 *
 * This is the data type of the \a pdos field in ec_sync_info_t.
 *
 * \see ecrt_slave_config_pdos().
 */
typedef struct {
    uint16_t index; /**< PDO index. */
    unsigned int n_entries; /**< Number of PDO entries in \a entries to map.
                              Zero means, that the default mapping shall be
                              used (this can only be done if the slave is
                              present at configuration time). */
    ec_pdo_entry_info_t const *entries; /**< Array of PDO entries to map. Can
                                          either be \a NULL, or must contain
                                          at least \a n_entries values. */
} ec_pdo_info_t;

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

/** Sync manager configuration information.
 *
 * This can be use to configure multiple sync managers including the PDO
 * assignment and PDO mapping. It is used as an input parameter type in
 * ecrt_slave_config_pdos().
 */
typedef struct {
```

```
    uint8_t index; /**< Sync manager index. Must be less
                     than #EC_MAX_SYNC_MANAGERS for a valid sync manager,
                     but can also be \a 0xff to mark the end of the list. */
    ec_direction_t dir; /**< Sync manager direction. */
    unsigned int n_pdos; /**< Number of PDOs in \a pdos. */
    ec_pdo_info_t const *pdos; /**< Array with PDOs to assign. This must
                                 contain at least \a n_pdos PDOs. */
    ec_watchdog_mode_t watchdog_mode; /**< Watchdog mode. */
} ec_sync_info_t;

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

/** List record type for PDO entry mass-registration.
 *
 * This type is used for the array parameter of the
 * ecrt_domain_reg_pdo_entry_list()
 */
typedef struct {
    uint16_t alias; /**< Slave alias address. */
    uint16_t position; /**< Slave position. */
    uint32_t vendor_id; /**< Slave vendor ID. */
    uint32_t product_code; /**< Slave product code. */
    uint16_t index; /**< PDO entry index. */
    uint8_t subindex; /**< PDO entry subindex. */
    unsigned int *offset; /**< Pointer to a variable to store the PDO entry's
                            (byte-)offset in the process data. */
    unsigned int *bit_position; /**< Pointer to a variable to store a bit
                                  position (0-7) within the \a offset. Can be
                                  NULL, in which case an error is raised if
                                  the PDO entry does not byte-align. */
} ec_pdo_entry_reg_t;

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

/** Request state.
 *
 * This is used as return type for ecrt_sdo_request_state() and
 * ecrt_voe_handler_state().
 */
typedef enum {
    EC_REQUEST_UNUSED, /**< Not requested. */
    EC_REQUEST_BUSY, /**< Request is being processed. */
    EC_REQUEST_SUCCESS, /**< Request was processed successfully. */
    EC_REQUEST_ERROR, /**< Request processing failed. */
} ec_request_state_t;

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

/** Application-layer state.
 */
typedef enum {
    EC_AL_STATE_INIT = 1, /**< Init. */
    EC_AL_STATE_PREOP = 2, /**< Pre-operational. */
    EC_AL_STATE_SAFEOP = 4, /**< Safe-operational. */
    EC_AL_STATE_OP = 8, /**< Operational. */
} ec_al_state_t;

/******************************************************************************
 * Global functions
 *****************************************************************************/

#ifdef __cplusplus
extern "C" {
#endif

/** Returns the version magic of the realtime interface.
```

```
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return Value of ECRT_VERSION_MAGIC() at EtherCAT master compile time.
 */
EC_PUBLIC_API unsigned int ecrt_version_magic(void);

/** Requests an EtherCAT master for realtime operation.
 *
 * Before an application can access an EtherCAT master, it has to reserve one
 * for exclusive use.
 *
 * In userspace, this is a convenience function for ecrt_open_master() and
 * ecrt_master_reserve().
 *
 * This function has to be the first function an application has to call to
 * use EtherCAT. The function takes the index of the master as its argument.
 * The first master has index 0, the n-th master has index n - 1. The number
 * of masters has to be specified when loading the master module.
 *
 * \apiusage{master_idle,blocking}
 *
 * \return Pointer to the reserved master, otherwise \a NULL.
 */
EC_PUBLIC_API ec_master_t *ecrt_request_master(
        unsigned int master_index /**< Index of the master to request. */
        );

#ifndef __KERNEL__

/** Opens an EtherCAT master for userspace access.
 *
 * This function has to be the first function an application has to call to
 * use EtherCAT. The function takes the index of the master as its argument.
 * The first master has index 0, the n-th master has index n - 1. The number
 * of masters has to be specified when loading the master module.
 *
 * For convenience, the function ecrt_request_master() can be used.
 *
 * \apiusage{master_idle,blocking}
 *
 * \return Pointer to the opened master, otherwise \a NULL.
 */
EC_PUBLIC_API ec_master_t *ecrt_open_master(
        unsigned int master_index /**< Index of the master to request. */
        );

#endif // #ifndef __KERNEL__

/** Releases a requested EtherCAT master.
 *
 * After use, a master it has to be released to make it available for other
 * applications.
 *
 * This method frees all created data structures. It should not be called in
 * realtime context.
 *
 * If the master was activated, ecrt_master_deactivate() is called internally.
 *
 * \apiusage{master_any,blocking}
 */
EC_PUBLIC_API void ecrt_release_master(
        ec_master_t *master /**< EtherCAT master */
        );

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

```
 * Master methods
 ****************************************************************************/

#ifndef __KERNEL__

/** Reserves an EtherCAT master for realtime operation.
 *
 * Before an application can use PDO/domain registration functions or SDO
 * request functions on the master, it has to reserve one for exclusive use.
 *
 * \apiusage{master_idle,blocking}
 *
 * \return 0 in case of success, else < 0
 */
EC_PUBLIC_API int ecrt_master_reserve(
        ec_master_t *master /**< EtherCAT master */
        );

#endif // #ifndef __KERNEL__

#ifdef __KERNEL__

/** Sets the locking callbacks.
 *
 * For concurrent master access, i. e. if other instances than the application
 * want to send and receive datagrams on the network, the application has to
 * provide a callback mechanism. This method takes two function pointers as
 * its parameters. Asynchronous master access (like EoE processing) is only
 * possible if the callbacks have been set.
 *
 * The task of the send callback (\a send_cb) is to decide, if the network
 * hardware is currently accessible and whether or not to call the
 * ecrt_master_send_ext() method.
 *
 * The task of the receive callback (\a receive_cb) is to decide, if a call to
 * ecrt_master_receive() is allowed and to execute it respectively.
 *
 * \apiusage{master_idle,blocking}
 *
 * \attention This method has to be called before ecrt_master_activate().
 */
void ecrt_master_callbacks(
        ec_master_t *master, /**< EtherCAT master */
        void (*send_cb)(void *), /**< Datagram sending callback. */
        void (*receive_cb)(void *), /**< Receive callback. */
        void *cb_data /**< Arbitrary pointer passed to the callback functions.
                       */
        );

#endif /* __KERNEL__ */

/** Creates a new process data domain.
 *
 * For process data exchange, at least one process data domain is needed.
 * This method creates a new process data domain and returns a pointer to the
 * new domain object. This object can be used for registering PDOs and
 * exchanging them in cyclic operation.
 *
 * This method allocates memory and should be called in non-realtime context
 * before ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return Pointer to the new domain on success, else NULL.
 */
EC_PUBLIC_API ec_domain_t *ecrt_master_create_domain(
```

```
        ec_master_t *master /**< EtherCAT master. */
        );

/** Obtains a slave configuration.
 *
 * Creates a slave configuration object for the given \a alias and \a position
 * tuple and returns it. If a configuration with the same \a alias and \a
 * position already exists, it will be re-used. In the latter case, the given
 * vendor ID and product code are compared to the stored ones. On mismatch, an
 * error message is raised and the function returns \a NULL.
 *
 * Slaves are addressed with the \a alias and \a position parameters.
 * - If \a alias is zero, \a position is interpreted as the desired slave's
 *   ring position.
 * - If \a alias is non-zero, it matches a slave with the given alias. In this
 *   case, \a position is interpreted as ring offset, starting from the
 *   aliased slave, so a position of zero means the aliased slave itself and a
 *   positive value matches the n-th slave behind the aliased one.
 *
 * If the slave with the given address is found during the configuration,
 * its vendor ID and product code are matched against the given value. On
 * mismatch, the slave is not configured and an error message is raised.
 *
 * If different slave configurations are pointing to the same slave during
 * configuration, a warning is raised and only the first configuration is
 * applied.
 *
 * This method allocates memory and should be called in non-realtime context
 * before ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval >0 Pointer to the slave configuration structure.
 * \retval NULL in the error case.
 */
EC_PUBLIC_API ec_slave_config_t *ecrt_master_slave_config(
        ec_master_t *master, /**< EtherCAT master */
        uint16_t alias, /**< Slave alias. */
        uint16_t position, /**< Slave position. */
        uint32_t vendor_id, /**< Expected vendor ID. */
        uint32_t product_code /**< Expected product code. */
        );

/** Selects the reference clock for distributed clocks.
 *
 * If this method is not called for a certain master, or if the slave
 * configuration pointer is NULL, then the first slave with DC functionality
 * will provide the reference clock.
 *
 * \apiusage{master_idle,blocking}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_master_select_reference_clock(
        ec_master_t *master, /**< EtherCAT master. */
        ec_slave_config_t *sc /**< Slave config of the slave to use as the
                               * reference slave (or NULL). */
        );

/** Obtains master information.
 *
 * No memory is allocated on the heap in this function.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \attention The pointer to this structure must point to a valid variable.
```

```
 *
 * \return 0 in case of success, else < 0
 */
EC_PUBLIC_API int ecrt_master(
        ec_master_t *master, /**< EtherCAT master */
        ec_master_info_t *master_info /**< Structure that will output the
                                        information */
        );

/** Obtains network scan progress information.
 *
 * No memory is allocated on the heap in this function.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \attention The pointer to this structure must point to a valid variable.
 *
 * \return 0 in case of success, else < 0
 */
EC_PUBLIC_API int ecrt_master_scan_progress(
        ec_master_t *master, /**< EtherCAT master */
        ec_master_scan_progress_t *progress /**< Structure that will output
                                              the progress information. */
        );

/** Obtains slave information.
 *
 * Tries to find the slave with the given ring position. The obtained
 * information is stored in a structure. No memory is allocated on the heap in
 * this function.
 *
 * \apiusage{master_any,blocking}
 *
 * \attention The pointer to this structure must point to a valid variable.
 *
 * \return 0 in case of success, else < 0
 */
EC_PUBLIC_API int ecrt_master_get_slave(
        ec_master_t *master, /**< EtherCAT master */
        uint16_t slave_position, /**< Slave position. */
        ec_slave_info_t *slave_info /**< Structure that will output the
                                      information */
        );

#ifndef __KERNEL__

/** Returns the proposed configuration of a slave's sync manager.
 *
 * Fills a given ec_sync_info_t structure with the attributes of a sync
 * manager. The \a pdos field of the return value is left empty. Use
 * ecrt_master_get_pdo() to get the PDO information.
 *
 * \apiusage{master_any,blocking}
 *
 * \return zero on success, else non-zero
 */
EC_PUBLIC_API int ecrt_master_get_sync_manager(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint8_t sync_index, /**< Sync manager index. Must be less
                              than #EC_MAX_SYNC_MANAGERS. */
        ec_sync_info_t *sync /**< Pointer to output structure. */
        );

/** Returns information about a currently assigned PDO.
 *
```

```
 * Fills a given ec_pdo_info_t structure with the attributes of a currently
 * assigned PDO of the given sync manager. The \a entries field of the return
 * value is left empty. Use ecrt_master_get_pdo_entry() to get the PDO
 * entry information.
 *
 * \apiusage{master_any,blocking}
 *
 * \retval zero on success, else non-zero
 */
EC_PUBLIC_API int ecrt_master_get_pdo(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint8_t sync_index, /**< Sync manager index. Must be less
                                 than #EC_MAX_SYNC_MANAGERS. */
        uint16_t pos, /**< Zero-based PDO position. */
        ec_pdo_info_t *pdo /**< Pointer to output structure. */
        );

/** Returns information about a currently mapped PDO entry.
 *
 * Fills a given ec_pdo_entry_info_t structure with the attributes of a
 * currently mapped PDO entry of the given PDO.
 *
 * \apiusage{master_any,blocking}
 *
 * \retval zero on success, else non-zero
 */
EC_PUBLIC_API int ecrt_master_get_pdo_entry(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint8_t sync_index, /**< Sync manager index. Must be less
                                 than #EC_MAX_SYNC_MANAGERS. */
        uint16_t pdo_pos, /**< Zero-based PDO position. */
        uint16_t entry_pos, /**< Zero-based PDO entry position. */
        ec_pdo_entry_info_t *entry /**< Pointer to output structure. */
        );

#endif /* #ifndef __KERNEL__ */

/** Executes an SDO download request to write data to a slave.
 *
 * This request is processed by the master state machine. This method blocks,
 * until the request has been processed and may not be called in realtime
 * context.
 *
 * \apiusage{master_any,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_master_sdo_download(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint16_t index, /**< Index of the SDO. */
        uint8_t subindex, /**< Subindex of the SDO. */
        const uint8_t *data, /**< Data buffer to download. */
        size_t data_size, /**< Size of the data buffer. */
        uint32_t *abort_code /**< Abort code of the SDO download. */
        );

/** Executes an SDO download request to write data to a slave via complete
 * access.
 *
 * This request is processed by the master state machine. This method blocks,
 * until the request has been processed and may not be called in realtime
 * context.
```

```
 *
 * \apiusage{master_any,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_master_sdo_download_complete(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint16_t index, /**< Index of the SDO. */
        const uint8_t *data, /**< Data buffer to download. */
        size_t data_size, /**< Size of the data buffer. */
        uint32_t *abort_code /**< Abort code of the SDO download. */
        );

/** Executes an SDO upload request to read data from a slave.
 *
 * This request is processed by the master state machine. This method blocks,
 * until the request has been processed and may not be called in realtime
 * context.
 *
 * \apiusage{master_any,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_master_sdo_upload(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint16_t index, /**< Index of the SDO. */
        uint8_t subindex, /**< Subindex of the SDO. */
        uint8_t *target, /**< Target buffer for the upload. */
        size_t target_size, /**< Size of the target buffer. */
        size_t *result_size, /**< Uploaded data size. */
        uint32_t *abort_code /**< Abort code of the SDO upload. */
        );

/** Executes an SoE write request.
 *
 * Starts writing an IDN and blocks until the request was processed, or an
 * error occurred.
 *
 * \apiusage{master_any,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_master_write_idn(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint8_t drive_no, /**< Drive number. */
        uint16_t idn, /**< SoE IDN (see ecrt_slave_config_idn()). */
        const uint8_t *data, /**< Pointer to data to write. */
        size_t data_size, /**< Size of data to write. */
        uint16_t *error_code /**< Pointer to variable, where an SoE error code
                               can be stored. */
        );

/** Executes an SoE read request.
 *
 * Starts reading an IDN and blocks until the request was processed, or an
 * error occurred.
 *
 * \apiusage{master_any,blocking}
 *
 * \retval  0 Success.
```

```
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_master_read_idn(
        ec_master_t *master, /**< EtherCAT master. */
        uint16_t slave_position, /**< Slave position. */
        uint8_t drive_no, /**< Drive number. */
        uint16_t idn, /**< SoE IDN (see ecrt_slave_config_idn()). */
        uint8_t *target, /**< Pointer to memory where the read data can be
                           stored. */
        size_t target_size, /**< Size of the memory \a target points to. */
        size_t *result_size, /**< Actual size of the received data. */
        uint16_t *error_code /**< Pointer to variable, where an SoE error code
                               can be stored. */
        );

/** Finishes the configuration phase and prepares for cyclic operation.
 *
 * This function tells the master that the configuration phase is finished and
 * the realtime operation will begin. The function allocates internal memory
 * for the domains and calculates the logical FMMU addresses for domain
 * members. It tells the master state machine that the configuration is
 * now to be applied to the network.
 *
 * \apiusage{master_idle,blocking}
 *
 * \attention After this function has been called, the realtime application is
 * in charge of cyclically calling ecrt_master_send() and
 * ecrt_master_receive() to ensure network communication. Before calling this
 * function, the master thread is responsible for that, so these functions may
 * not be called! The method itself allocates memory and should not be called
 * in realtime context.
 *
 * \return 0 in case of success, else < 0
 */
EC_PUBLIC_API int ecrt_master_activate(
        ec_master_t *master /**< EtherCAT master. */
        );

/** Deactivates the master.
 *
 * Removes the master configuration. All objects created by
 * ecrt_master_create_domain(), ecrt_master_slave_config(), ecrt_domain_data()
 * ecrt_slave_config_create_sdo_request() and
 * ecrt_slave_config_create_voe_handler() are freed, so pointers to them
 * become invalid.
 *
 * \apiusage{master_op,blocking}
 *
 * This method should not be called in realtime context.
 * \return 0 on success, otherwise negative error code.
 * \retval 0 Success.
 * \retval -EINVAL Master has not been activated before.
 */
EC_PUBLIC_API int ecrt_master_deactivate(
        ec_master_t *master /**< EtherCAT master. */
        );

/** Set interval between calls to ecrt_master_send().
 *
 * This information helps the master to decide, how much data can be appended
 * to a frame by the master state machine. When the master is configured with
 * --enable-hrtimers, this is used to calculate the scheduling of the master
 * thread.
 *
 * \apiusage{master_idle,blocking}
 *
```

```
 * \retval 0 on success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_master_set_send_interval(
        ec_master_t *master, /**< EtherCAT master. */
        size_t send_interval /**< Send interval in us */
        );

/** Sends all datagrams in the queue.
 *
 * This method takes all datagrams, that have been queued for transmission,
 * puts them into frames, and passes them to the Ethernet device for sending.
 *
 * Has to be called cyclically by the application after ecrt_master_activate()
 * has returned.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Zero on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_master_send(
        ec_master_t *master /**< EtherCAT master. */
        );

/** Fetches received frames from the hardware and processes the datagrams.
 *
 * Queries the network device for received frames by calling the interrupt
 * service routine. Extracts received datagrams and dispatches the results to
 * the datagram objects in the queue. Received datagrams, and the ones that
 * timed out, will be marked, and dequeued.
 *
 * Has to be called cyclically by the realtime application after
 * ecrt_master_activate() has returned.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Zero on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_master_receive(
        ec_master_t *master /**< EtherCAT master. */
        );

#ifdef __KERNEL__
/** Sends non-application datagrams.
 *
 * This method has to be called in the send callback function passed via
 * ecrt_master_callbacks() to allow the sending of non-application datagrams.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Zero on success, otherwise negative error code.
 * \retval -EAGAIN Lock could not be acquired, try again later.
 */
int ecrt_master_send_ext(
        ec_master_t *master /**< EtherCAT master. */
        );
#endif

/** Reads the current master state.
 *
 * Stores the master state information in the given \a state structure.
 *
 * This method returns a global state. For the link-specific states in a
 * redundant network topology, use the ecrt_master_link_state() method.
 *
 * \apiusage{master_any,rt_safe}
```

```
 *
 * \return Zero on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_master_state(
        const ec_master_t *master, /**< EtherCAT master. */
        ec_master_state_t *state /**< Structure to store the information. */
        );

/** Reads the current state of a redundant link.
 *
 * Stores the link state information in the given \a state structure.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return Zero on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_master_link_state(
        const ec_master_t *master, /**< EtherCAT master. */
        unsigned int dev_idx, /**< Index of the device (0 = main device, 1 =
                                first backup device, ...). */
        ec_master_link_state_t *state /**< Structure to store the information.
                                       */
        );

/** Sets the application time.
 *
 * The master has to know the application's time when operating slaves with
 * distributed clocks. The time is not incremented by the master itself, so
 * this method has to be called cyclically.
 *
 * \attention The time passed to this method is used to calculate the phase of
 * the slaves' SYNC0/1 interrupts. It should be called constantly at the same
 * point of the realtime cycle. So it is recommended to call it at the start
 * of the calculations to avoid deviancies due to changing execution times.
 * Avoid calling this method before the realtime cycle is established.
 *
 * The time is used when setting the slaves' <tt>System Time Offset</tt> and
 * <tt>Cyclic Operation Start Time</tt> registers and when synchronizing the
 * DC reference clock to the application time via
 * ecrt_master_sync_reference_clock().
 *
 * The time is defined as nanoseconds from 2000-01-01 00:00. Converting an
 * epoch time can be done with the EC_TIMEVAL2NANO() macro, but is not
 * necessary, since the absolute value is not of any interest.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Zero on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_master_application_time(
        ec_master_t *master, /**< EtherCAT master. */
        uint64_t app_time /**< Application time. */
        );

/** Queues the DC reference clock drift compensation datagram for sending.
 *
 * The reference clock will by synchronized to the application time provided
 * by the last call off ecrt_master_application_time().
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Zero on success, otherwise negative error code.
 * \retval 0 Success.
 * \retval -ENXIO No reference clock found.
 */
EC_PUBLIC_API int ecrt_master_sync_reference_clock(
```

```
        ec_master_t *master /**< EtherCAT master. */
        );

/** Queues the DC reference clock drift compensation datagram for sending.
 *
 * The reference clock will by synchronized to the time passed in the
 * sync_time parameter.
 *
 * Has to be called by the application after ecrt_master_activate()
 * has returned.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Zero on success, otherwise negative error code.
 * \retval 0 Success.
 * \retval -ENXIO No reference clock found.
 */
EC_PUBLIC_API int ecrt_master_sync_reference_clock_to(
        ec_master_t *master, /**< EtherCAT master. */
        uint64_t sync_time /**< Sync reference clock to this time. */
        );

/** Queues the DC clock drift compensation datagram for sending.
 *
 * All slave clocks synchronized to the reference clock.
 *
 * Has to be called by the application after ecrt_master_activate()
 * has returned.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 * \retval 0 Success.
 * \retval -ENXIO No reference clock found.
 */
EC_PUBLIC_API int ecrt_master_sync_slave_clocks(
        ec_master_t *master /**< EtherCAT master. */
        );

/** Get the lower 32 bit of the reference clock system time.
 *
 * This method can be used to synchronize the master to the reference clock.
 *
 * The reference clock system time is queried via the
 * ecrt_master_sync_slave_clocks() method, that reads the system time of the
 * reference clock and writes it to the slave clocks (so be sure to call it
 * cyclically to get valid data).
 *
 * \attention The returned time is the system time of the reference clock
 * minus the transmission delay of the reference clock.
 *
 * Calling this method makes only sense in realtime context (after master
 * activation), when the ecrt_master_sync_slave_clocks() method is called
 * cyclically.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \retval 0 success, system time was written into \a time.
 * \retval -ENXIO No reference clock found.
 * \retval -EIO Slave synchronization datagram was not received.
 */
EC_PUBLIC_API int ecrt_master_reference_clock_time(
        const ec_master_t *master, /**< EtherCAT master. */
        uint32_t *time /**< Pointer to store the queried system time. */
        );
```

```
/** Queues the DC synchrony monitoring datagram for sending.
 *
 * The datagram broadcast-reads all "System time difference" registers (\a
 * 0x092c) to get an upper estimation of the DC synchrony. The result can be
 * checked with the ecrt_master_sync_monitor_process() method.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Zero on success, otherwise a negative error code.
 */
EC_PUBLIC_API int ecrt_master_sync_monitor_queue(
        ec_master_t *master /**< EtherCAT master. */
        );

/** Processes the DC synchrony monitoring datagram.
 *
 * If the sync monitoring datagram was sent before with
 * ecrt_master_sync_monitor_queue(), the result can be queried with this
 * method.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Upper estimation of the maximum time difference in ns, -1 on error.
 * \retval (uint32_t)-1 Error.
 */
EC_PUBLIC_API uint32_t ecrt_master_sync_monitor_process(
        const ec_master_t *master /**< EtherCAT master. */
        );

/** Retry configuring slaves.
 *
 * Via this method, the application can tell the master to bring all slaves to
 * OP state. In general, this is not necessary, because it is automatically
 * done by the master. But with special slaves, that can be reconfigured by
 * the vendor during runtime, it can be useful.
 *
 * Calling this method only makes sense in realtime context (after
 * activation), because slaves will not be configured before.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_master_reset(
        ec_master_t *master /**< EtherCAT master. */
        );

/*****************************************************************************
 * Slave configuration methods
 ****************************************************************************/

/** Configure a sync manager.
 *
 * Sets the direction of a sync manager. This overrides the direction bits
 * from the default control register from SII.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return zero on success, else non-zero
 */
EC_PUBLIC_API int ecrt_slave_config_sync_manager(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint8_t sync_index, /**< Sync manager index. Must be less
```

```
                              than #EC_MAX_SYNC_MANAGERS. */
        ec_direction_t direction, /**< Input/Output. */
        ec_watchdog_mode_t watchdog_mode /** Watchdog mode. */
        );

/** Configure a slave's watchdog times.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_slave_config_watchdog(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint16_t watchdog_divider, /**< Number of 40 ns intervals (register
                                    0x0400). Used as a base unit for all
                                    slave watchdogs^. If set to zero, the
                                    value is not written, so the default is
                                    used. */
        uint16_t watchdog_intervals /**< Number of base intervals for sync
                                     manager watchdog (register 0x0420). If
                                     set to zero, the value is not written,
                                     so the default is used. */
        );

/** Add a PDO to a sync manager's PDO assignment.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \see ecrt_slave_config_pdos()
 * \return zero on success, else non-zero
 */
EC_PUBLIC_API int ecrt_slave_config_pdo_assign_add(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint8_t sync_index, /**< Sync manager index. Must be less
                              than #EC_MAX_SYNC_MANAGERS. */
        uint16_t index /**< Index of the PDO to assign. */
        );

/** Clear a sync manager's PDO assignment.
 *
 * This can be called before assigning PDOs via
 * ecrt_slave_config_pdo_assign_add(), to clear the default assignment of a
 * sync manager.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \see ecrt_slave_config_pdos()
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_slave_config_pdo_assign_clear(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint8_t sync_index /**< Sync manager index. Must be less
                              than #EC_MAX_SYNC_MANAGERS. */
        );

/** Add a PDO entry to the given PDO's mapping.
 *
```

```
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \see ecrt_slave_config_pdos()
 * \return zero on success, else non-zero
 */
EC_PUBLIC_API int ecrt_slave_config_pdo_mapping_add(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint16_t pdo_index, /**< Index of the PDO. */
        uint16_t entry_index, /**< Index of the PDO entry to add to the PDO's
                                mapping. */
        uint8_t entry_subindex, /**< Subindex of the PDO entry to add to the
                                  PDO's mapping. */
        uint8_t entry_bit_length /**< Size of the PDO entry in bit. */
        );

/** Clear the mapping of a given PDO.
 *
 * This can be called before mapping PDO entries via
 * ecrt_slave_config_pdo_mapping_add(), to clear the default mapping.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \see ecrt_slave_config_pdos()
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_slave_config_pdo_mapping_clear(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint16_t pdo_index /**< Index of the PDO. */
        );

/** Specify a complete PDO configuration.
 *
 * This function is a convenience wrapper for the functions
 * ecrt_slave_config_sync_manager(), ecrt_slave_config_pdo_assign_clear(),
 * ecrt_slave_config_pdo_assign_add(), ecrt_slave_config_pdo_mapping_clear()
 * and ecrt_slave_config_pdo_mapping_add(), that are better suitable for
 * automatic code generation.
 *
 * The following example shows, how to specify a complete configuration,
 * including the PDO mappings. With this information, the master is able to
 * reserve the complete process data, even if the slave is not present at
 * configuration time:
 *
 * \code
 * ec_pdo_entry_info_t el3162_channel1[] = {
 *     {0x3101, 1,  8}, // status
 *     {0x3101, 2, 16}  // value
 * };
 *
 * ec_pdo_entry_info_t el3162_channel2[] = {
 *     {0x3102, 1,  8}, // status
 *     {0x3102, 2, 16}  // value
 * };
 *
 * ec_pdo_info_t el3162_pdos[] = {
 *     {0x1A00, 2, el3162_channel1},
 *     {0x1A01, 2, el3162_channel2}
 * };
 *
 * ec_sync_info_t el3162_syncs[] = {
```

```
 *     {2, EC_DIR_OUTPUT},
 *     {3, EC_DIR_INPUT, 2, el3162_pdos},
 *     {0xff}
 * };
 *
 * if (ecrt_slave_config_pdos(sc_ana_in, EC_END, el3162_syncs)) {
 *     // handle error
 * }
 * \endcode
 *
 * The next example shows, how to configure the PDO assignment only. The
 * entries for each assigned PDO are taken from the PDO's default mapping.
 * Please note, that PDO entry registration will fail, if the PDO
 * configuration is left empty and the slave is offline.
 *
 * \code
 * ec_pdo_info_t pdos[] = {
 *     {0x1600}, // Channel 1
 *     {0x1601}  // Channel 2
 * };
 *
 * ec_sync_info_t syncs[] = {
 *     {3, EC_DIR_INPUT, 2, pdos},
 * };
 *
 * if (ecrt_slave_config_pdos(slave_config_ana_in, 1, syncs)) {
 *     // handle error
 * }
 * \endcode
 *
 * Processing of \a syncs will stop, if
 * - the number of processed items reaches \a n_syncs, or
 * - the \a index member of an ec_sync_info_t item is 0xff. In this case,
 *   \a n_syncs should set to a number greater than the number of list items;
 *   using EC_END is recommended.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return zero on success, else non-zero
 */
EC_PUBLIC_API int ecrt_slave_config_pdos(
        ec_slave_config_t *sc, /**< Slave configuration. */
        unsigned int n_syncs, /**< Number of sync manager configurations in
                                \a syncs. */
        const ec_sync_info_t syncs[] /**< Array of sync manager
                                       configurations. */
        );

/** Registers a PDO entry for process data exchange in a domain.
 *
 * Searches the assigned PDOs for the given PDO entry. An error is raised, if
 * the given entry is not mapped. Otherwise, the corresponding sync manager
 * and FMMU configurations are provided for slave configuration and the
 * respective sync manager's assigned PDOs are appended to the given domain,
 * if not already done. The offset of the requested PDO entry's data inside
 * the domain's process data is returned. Optionally, the PDO entry bit
 * position (0-7) can be retrieved via the \a bit_position output parameter.
 * This pointer may be \a NULL, in this case an error is raised if the PDO
 * entry does not byte-align.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
```

```
 * \apiusage{master_idle,blocking}
 *
 * \retval >=0 Success: Offset of the PDO entry's process data.
 * \retval  <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_reg_pdo_entry(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint16_t entry_index, /**< Index of the PDO entry to register. */
        uint8_t entry_subindex, /**< Subindex of the PDO entry to register. */
        ec_domain_t *domain, /**< Domain. */
        unsigned int *bit_position /**< Optional address if bit addressing
                                     is desired */
        );

/** Registers a PDO entry using its position.
 *
 * Similar to ecrt_slave_config_reg_pdo_entry(), but not using PDO indices but
 * offsets in the PDO mapping, because PDO entry indices may not be unique
 * inside a slave's PDO mapping. An error is raised, if
 * one of the given positions is out of range.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval >=0 Success: Offset of the PDO entry's process data.
 * \retval  <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_reg_pdo_entry_pos(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint8_t sync_index, /**< Sync manager index. */
        unsigned int pdo_pos, /**< Position of the PDO inside the SM. */
        unsigned int entry_pos, /**< Position of the entry inside the PDO. */
        ec_domain_t *domain, /**< Domain. */
        unsigned int *bit_position /**< Optional address if bit addressing
                                     is desired */
        );

/** Configure distributed clocks.
 *
 * Sets the AssignActivate word and the cycle and shift times for the sync
 * signals.
 *
 * The AssignActivate word is vendor-specific and can be taken from the XML
 * device description file (Device -> Dc -> AssignActivate). Set this to zero,
 * if the slave shall be operated without distributed clocks (default).
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \attention The \a sync1_shift time is ignored.
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_slave_config_dc(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint16_t assign_activate, /**< AssignActivate word. */
        uint32_t sync0_cycle, /**< SYNC0 cycle time [ns]. */
        int32_t sync0_shift, /**< SYNC0 shift time [ns]. */
        uint32_t sync1_cycle, /**< SYNC1 cycle time [ns]. */
        int32_t sync1_shift /**< SYNC1 shift time [ns]. */
        );

/** Add an SDO configuration.
```

```
 *
 * An SDO configuration is stored in the slave configuration object and is
 * downloaded to the slave whenever the slave is being configured by the
 * master. This usually happens once on master activation, but can be repeated
 * subsequently, for example after the slave's power supply failed.
 *
 * \attention The SDOs for PDO assignment (\p 0x1C10 - \p 0x1C2F) and PDO
 * mapping (\p 0x1600 - \p 0x17FF and \p 0x1A00 - \p 0x1BFF) should not be
 * configured with this function, because they are part of the slave
 * configuration done by the master. Please use ecrt_slave_config_pdos() and
 * friends instead.
 *
 * This is the generic function for adding an SDO configuration. Please note
 * that the this function does not do any endianness correction. If
 * datatype-specific functions are needed (that automatically correct the
 * endianness), have a look at ecrt_slave_config_sdo8(),
 * ecrt_slave_config_sdo16() and ecrt_slave_config_sdo32().
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_sdo(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint16_t index, /**< Index of the SDO to configure. */
        uint8_t subindex, /**< Subindex of the SDO to configure. */
        const uint8_t *data, /**< Pointer to the data. */
        size_t size /**< Size of the \a data. */
        );

/** Add a configuration value for an 8-bit SDO.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \see ecrt_slave_config_sdo().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_sdo8(
        ec_slave_config_t *sc, /**< Slave configuration */
        uint16_t sdo_index, /**< Index of the SDO to configure. */
        uint8_t sdo_subindex, /**< Subindex of the SDO to configure. */
        uint8_t value /**< Value to set. */
        );

/** Add a configuration value for a 16-bit SDO.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \see ecrt_slave_config_sdo().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_sdo16(
```

```
        ec_slave_config_t *sc, /**< Slave configuration */
        uint16_t sdo_index, /**< Index of the SDO to configure. */
        uint8_t sdo_subindex, /**< Subindex of the SDO to configure. */
        uint16_t value /**< Value to set. */
        );

/** Add a configuration value for a 32-bit SDO.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \see ecrt_slave_config_sdo().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_sdo32(
        ec_slave_config_t *sc, /**< Slave configuration */
        uint16_t sdo_index, /**< Index of the SDO to configure. */
        uint8_t sdo_subindex, /**< Subindex of the SDO to configure. */
        uint32_t value /**< Value to set. */
        );

/** Add configuration data for a complete SDO.
 *
 * The SDO data are transferred via CompleteAccess. Data for the first
 * subindex (0) have to be included.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \see ecrt_slave_config_sdo().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_complete_sdo(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint16_t index, /**< Index of the SDO to configure. */
        const uint8_t *data, /**< Pointer to the data. */
        size_t size /**< Size of the \a data. */
        );

/** Set the size of the CoE emergency ring buffer.
 *
 * The initial size is zero, so all messages will be dropped. This method can
 * be called even after master activation, but it will clear the ring buffer!
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return 0 on success, or negative error code.
 */
EC_PUBLIC_API int ecrt_slave_config_emerg_size(
        ec_slave_config_t *sc, /**< Slave configuration. */
        size_t elements /**< Number of records of the CoE emergency ring. */
        );

/** Read and remove one record from the CoE emergency ring buffer.
 *
```

```
 * A record consists of 8 bytes:
 *
 * Byte 0-1: Error code (little endian)
 * Byte   2: Error register
 * Byte 3-7: Data
 *
 * Calling this method makes only sense in realtime context (after master
 * activation).
 *
 * \return 0 on success (record popped), or negative error code (i.e.
 * -ENOENT, if ring is empty).
 *
 * \apiusage{master_op,any_context}
 */
EC_PUBLIC_API int ecrt_slave_config_emerg_pop(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint8_t *target /**< Pointer to target memory (at least
                          EC_COE_EMERGENCY_MSG_SIZE bytes). */
        );

/** Clears CoE emergency ring buffer and the overrun counter.
 *
 * Calling this method makes only sense in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,any_context}
 *
 * \return 0 on success, or negative error code.
 *
 */
EC_PUBLIC_API int ecrt_slave_config_emerg_clear(
        ec_slave_config_t *sc /**< Slave configuration. */
        );

/** Read the number of CoE emergency overruns.
 *
 * The overrun counter will be incremented when a CoE emergency message could
 * not be stored in the ring buffer and had to be dropped. Call
 * ecrt_slave_config_emerg_clear() to reset the counter.
 *
 * Calling this method makes only sense in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,any_context}
 *
 * \return Number of overruns since last clear, or negative error code.
 *
 */
EC_PUBLIC_API int ecrt_slave_config_emerg_overruns(
        const ec_slave_config_t *sc /**< Slave configuration. */
        );

/** Create an SDO request to exchange SDOs during realtime operation.
 *
 * The created SDO request object is freed automatically when the master is
 * released.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return New SDO request, or NULL on error.
 */
EC_PUBLIC_API ec_sdo_request_t *ecrt_slave_config_create_sdo_request(
        ec_slave_config_t *sc, /**< Slave configuration. */
```

```
        uint16_t index, /**< SDO index. */
        uint8_t subindex, /**< SDO subindex. */
        size_t size /**< Data size to reserve. */
        );

/** Create an SoE request to exchange SoE IDNs during realtime operation.
 *
 * The created SoE request object is freed automatically when the master is
 * released.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return New SoE request, or NULL on error.
 */
EC_PUBLIC_API ec_soe_request_t *ecrt_slave_config_create_soe_request(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint8_t drive_no, /**< Drive number. */
        uint16_t idn, /**< Sercos ID-Number. */
        size_t size /**< Data size to reserve. */
        );

/** Create an VoE handler to exchange vendor-specific data during realtime
 * operation.
 *
 * The number of VoE handlers per slave configuration is not limited, but
 * usually it is enough to create one for sending and one for receiving, if
 * both can be done simultaneously.
 *
 * The created VoE handler object is freed automatically when the master is
 * released.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return New VoE handler, or NULL on error.
 */
EC_PUBLIC_API ec_voe_handler_t *ecrt_slave_config_create_voe_handler(
        ec_slave_config_t *sc, /**< Slave configuration. */
        size_t size /**< Data size to reserve. */
        );

/** Create a register request to exchange EtherCAT register contents during
 * realtime operation.
 *
 * This interface should not be used to take over master functionality,
 * instead it is intended for debugging and monitoring reasons.
 *
 * The created register request object is freed automatically when the master
 * is released.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \return New register request, or NULL on error.
 */
EC_PUBLIC_API ec_reg_request_t *ecrt_slave_config_create_reg_request(
        ec_slave_config_t *sc, /**< Slave configuration. */
        size_t size /**< Data size to reserve. */
        );
```

```

/** Outputs the state of the slave configuration.
 *
 * Stores the state information in the given \a state structure. The state
 * information is updated by the master state machine, so it may take a few
 * cycles, until it changes.
 *
 * \attention If the state of process data exchange shall be monitored in
 * realtime, ecrt_domain_state() should be used.
 *
 * \apiusage{master_op,rt_safe}
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_state(
        const ec_slave_config_t *sc, /**< Slave configuration */
        ec_slave_config_state_t *state /**< State object to write to. */
        );

/** Add an SoE IDN configuration.
 *
 * A configuration for a Sercos-over-EtherCAT IDN is stored in the slave
 * configuration object and is written to the slave whenever the slave is
 * being configured by the master. This usually happens once on master
 * activation, but can be repeated subsequently, for example after the slave's
 * power supply failed.
 *
 * The \a idn parameter can be separated into several sections:
 *  - Bit 15: Standard data (0) or Product data (1)
 *  - Bit 14 - 12: Parameter set (0 - 7)
 *  - Bit 11 - 0: Data block number (0 - 4095)
 *
 * Please note that the this function does not do any endianness correction.
 * Multi-byte data have to be passed in EtherCAT endianness (little-endian).
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_idn(
        ec_slave_config_t *sc, /**< Slave configuration. */
        uint8_t drive_no, /**< Drive number. */
        uint16_t idn, /**< SoE IDN. */
        ec_al_state_t state, /**< AL state in which to write the IDN (PREOP or
                               SAFEOP). */
        const uint8_t *data, /**< Pointer to the data. */
        size_t size /**< Size of the \a data. */
        );

/** Adds a feature flag to a slave configuration.
 *
 * Feature flags are a generic way to configure slave-specific behavior.
 *
 * Multiple calls with the same slave configuration and key will overwrite the
 * configuration.
 *
 * The following flags may be available:
 * - AssignToPdi: Zero (default) keeps the slave information interface (SII)
```

```
 *   assigned to EtherCAT (except during transition to PREOP). Non-zero
 *   assigns the SII to the slave controller side before going to PREOP and
 *   leaves it there until a write command happens.
 * - WaitBeforeSAFEOPms: Number of milliseconds to wait before commanding the
 *   transition from PREOP to SAFEOP. This can be used as a workaround for
 *   slaves that need a little time to initialize.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_flag(
        ec_slave_config_t *sc, /**< Slave configuration. */
        const char *key, /**< Key as null-terminated ASCII string. */
        int32_t value /**< Value to store. */
        );

/** Sets the link/MAC address for Ethernet-over-EtherCAT (EoE) operation.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * The MAC address is stored in the slave configuration object and will be
 * written to the slave during the configuration process.
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_eoe_mac_address(
        ec_slave_config_t *sc, /**< Slave configuration. */
        const unsigned char *mac_address /**< MAC address. */
        );

/** Sets the IP address for Ethernet-over-EtherCAT (EoE) operation.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * The IP address is stored in the slave configuration object and will be
 * written to the slave during the configuration process.
 *
 * The IP address is passed by-value as a `struct in_addr`. This structure
 * contains the 32-bit IPv4 address in network byte order (big endian).
 *
 * A string-represented IPv4 address can be converted to a `struct in_addr`
 * for example via the POSIX function `inet_pton()` (see man 3 inet_pton):
 *
 * \code{.c}
 *     #include <arpa/inet.h>
 *     struct in_addr addr;
 *     if (inet_aton("192.168.0.1", &addr) == 0) {
 *         fprintf(stderr, "Failed to convert IP address.\n");
 *         return -1;
 *     }
 *     if (ecrt_slave_config_eoe_ip_address(sc, addr)) {
 *         fprintf(stderr, "Failed to set IP address.\n");
 *         return -1;
 *     }
 * \endcode
 *
```

```
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_eoe_ip_address(
        ec_slave_config_t *sc, /**< Slave configuration. */
        struct in_addr ip_address /**< IPv4 address. */
        );

/** Sets the subnet mask for Ethernet-over-EtherCAT (EoE) operation.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * The subnet mask is stored in the slave configuration object and will be
 * written to the slave during the configuration process.
 *
 * The subnet mask is passed by-value as a `struct in_addr`. This structure
 * contains the 32-bit mask in network byte order (big endian).
 *
 * See ecrt_slave_config_eoe_ip_address() on how to convert string-coded masks
 * to `struct in_addr`.
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_eoe_subnet_mask(
        ec_slave_config_t *sc, /**< Slave configuration. */
        struct in_addr subnet_mask /**< IPv4 subnet mask. */
        );

/** Sets the gateway address for Ethernet-over-EtherCAT (EoE) operation.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * The gateway address is stored in the slave configuration object and will be
 * written to the slave during the configuration process.
 *
 * The address is passed by-value as a `struct in_addr`. This structure
 * contains the 32-bit IPv4 address in network byte order (big endian).
 *
 * See ecrt_slave_config_eoe_ip_address() on how to convert string-coded IPv4
 * addresses to `struct in_addr`.
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_eoe_default_gateway(
        ec_slave_config_t *sc, /**< Slave configuration. */
        struct in_addr gateway_address /**< Gateway's IPv4 address. */
        );

/** Sets the IPv4 address of the DNS server for Ethernet-over-EtherCAT (EoE)
 * operation.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * The DNS server address is stored in the slave configuration object and will
```

```
 * be written to the slave during the configuration process.
 *
 * The address is passed by-value as a ‘struct in_addr‘. This structure
 * contains the 32-bit IPv4 address in network byte order (big endian).
 *
 * See ecrt_slave_config_eoe_ip_address() on how to convert string-coded IPv4
 * addresses to ‘struct in_addr‘.
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_eoe_dns_address(
        ec_slave_config_t *sc, /**< Slave configuration. */
        struct in_addr dns_address /**< IPv4 address of the DNS server. */
        );

/** Sets the host name for Ethernet-over-EtherCAT (EoE) operation.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * The host name is stored in the slave configuration object and will
 * be written to the slave during the configuration process.
 *
 * The maximum size of the host name is 32 bytes (including the zero
 * terminator).
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_eoe_hostname(
        ec_slave_config_t *sc, /**< Slave configuration. */
        const char *name /**< Zero-terminated host name. */
        );

/** Sets the application-layer state transition timeout in ms.
 *
 * Change the maximum allowed time for a slave to make an application-layer
 * state transition for the given state transition (for example from PREOP to
 * SAFEOP). The default values are defined in ETG.2000.
 *
 * A timeout value of zero ms will restore the default value.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 *
 * \retval  0 Success.
 * \retval <0 Error code.
 */
EC_PUBLIC_API int ecrt_slave_config_state_timeout(
        ec_slave_config_t *sc, /**< Slave configuration. */
        ec_al_state_t from_state, /**< Initial state. */
        ec_al_state_t to_state, /**< Target state. */
        unsigned int timeout_ms /**< Timeout in [ms]. */
        );

/*****************************************************************************
 * Domain methods
 ****************************************************************************/

```

```
/** Registers a bunch of PDO entries for a domain.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \see ecrt_slave_config_reg_pdo_entry()
 *
 * \attention The registration array has to be terminated with an empty
 *            structure, or one with the \a index field set to zero!
 *
 * \apiusage{master_idle,blocking}
 *
 * \return 0 on success, else non-zero.
 */
EC_PUBLIC_API int ecrt_domain_reg_pdo_entry_list(
        ec_domain_t *domain, /**< Domain. */
        const ec_pdo_entry_reg_t *pdo_entry_regs /**< Array of PDO
                                                   registrations. */
        );

/** Returns the current size of the domain's process data.
 *
 * The domain size is calculated after master activation.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Size of the process data image, or a negative error code.
 */
EC_PUBLIC_API size_t ecrt_domain_size(
        const ec_domain_t *domain /**< Domain. */
        );

#ifdef __KERNEL__

/** Provide external memory to store the domain's process data.
 *
 * Call this after all PDO entries have been registered and before activating
 * the master.
 *
 * The size of the allocated memory must be at least ecrt_domain_size(), after
 * all PDO entries have been registered.
 *
 * This method has to be called in non-realtime context before
 * ecrt_master_activate().
 *
 * \apiusage{master_idle,blocking}
 */
void ecrt_domain_external_memory(
        ec_domain_t *domain, /**< Domain. */
        uint8_t *memory /**< Address of the memory to store the process
                          data in. */
        );

#endif /* __KERNEL__ */

/** Returns the domain's process data.
 *
 * - In kernel context: If external memory was provided with
 * ecrt_domain_external_memory(), the returned pointer will contain the
 * address of that memory. Otherwise it will point to the internally allocated
 * memory. In the latter case, this method may not be called before
 * ecrt_master_activate().
 *
 * - In userspace context: This method has to be called after
 * ecrt_master_activate() to get the mapped domain process data memory.
 *
```

```
 * \apiusage{master_op,rt_safe}
 *
 * \return Pointer to the process data memory.
 */
EC_PUBLIC_API uint8_t *ecrt_domain_data(
        const ec_domain_t *domain /**< Domain. */
        );

/** Determines the states of the domain's datagrams.
 *
 * Evaluates the working counters of the received datagrams and outputs
 * statistics, if necessary. This must be called after ecrt_master_receive()
 * is expected to receive the domain datagrams in order to make
 * ecrt_domain_state() return the result of the last process data exchange.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_domain_process(
        ec_domain_t *domain /**< Domain. */
        );

/** (Re-)queues all domain datagrams in the master's datagram queue.
 *
 * Call this function to mark the domain's datagrams for exchanging at the
 * next call of ecrt_master_send().
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_domain_queue(
        ec_domain_t *domain /**< Domain. */
        );

/** Reads the state of a domain.
 *
 * Stores the domain state in the given \a state structure.
 *
 * Using this method, the process data exchange can be monitored in realtime.
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_domain_state(
        const ec_domain_t *domain, /**< Domain. */
        ec_domain_state_t *state /**< Pointer to a state object to store the
                                   information. */
        );

/*****************************************************************************
 * SDO request methods.
 ****************************************************************************/

/** Set the SDO index and subindex.
 *
 * \attention If the SDO index and/or subindex is changed while
 * ecrt_sdo_request_state() returns EC_REQUEST_BUSY, this may lead to
 * unexpected results.
 *
 * This method is meant to be called in realtime context (after master
 * activation). To initialize the SDO request, the index and subindex can be
 * set via ecrt_slave_config_create_sdo_request().
 *
```

```
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_sdo_request_index(
        ec_sdo_request_t *req, /**< SDO request. */
        uint16_t index, /**< SDO index. */
        uint8_t subindex /**< SDO subindex. */
        );

/** Set the timeout for an SDO request.
 *
 * If the request cannot be processed in the specified time, if will be marked
 * as failed.
 *
 * The timeout is permanently stored in the request object and is valid until
 * the next call of this method.
 *
 * The timeout should be defined in non-realtime context, but can also be
 * changed afterwards.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_sdo_request_timeout(
        ec_sdo_request_t *req, /**< SDO request. */
        uint32_t timeout /**< Timeout in milliseconds. Zero means no
                           timeout. */
        );

/** Access to the SDO request's data.
 *
 * This function returns a pointer to the request's internal SDO data memory.
 *
 * - After a read operation was successful, integer data can be evaluated
 *   using the EC_READ_*() macros as usual. Example:
 *   \code
 *   uint16_t value = EC_READ_U16(ecrt_sdo_request_data(sdo)));
 *   \endcode
 * - If a write operation shall be triggered, the data have to be written to
 *   the internal memory. Use the EC_WRITE_*() macros, if you are writing
 *   integer data. Be sure, that the data fit into the memory. The memory size
 *   is a parameter of ecrt_slave_config_create_sdo_request().
 *   \code
 *   EC_WRITE_U16(ecrt_sdo_request_data(sdo), 0xFFFF);
 *   \endcode
 *
 * \attention The return value can be invalid during a read operation, because
 * the internal SDO data memory could be re-allocated if the read SDO data do
 * not fit inside.
 *
 * This method is meant to be called in realtime context (after master
 * activation), but can also be used to initialize data before.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return Pointer to the internal SDO data memory.
 *
 */
EC_PUBLIC_API uint8_t *ecrt_sdo_request_data(
        const ec_sdo_request_t *req /**< SDO request. */
        );

/** Returns the current SDO data size.
 *
```

```
 * When the SDO request is created, the data size is set to the size of the
 * reserved memory. After a read operation the size is set to the size of the
 * read data. The size is not modified in any other situation.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return SDO data size in bytes.
 *
 */
EC_PUBLIC_API size_t ecrt_sdo_request_data_size(
        const ec_sdo_request_t *req /**< SDO request. */
        );

/** Get the current state of the SDO request.
 *
 * The user-space implementation fetches incoming data and stores the received
 * data size in the request object, so the request is not const.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Request state.
 *
 */
EC_PUBLIC_API ec_request_state_t ecrt_sdo_request_state(
#ifdef __KERNEL__
        const
#endif
        ec_sdo_request_t *req /**< SDO request. */
        );

/** Schedule an SDO write operation.
 *
 * \attention This method may not be called while ecrt_sdo_request_state()
 * returns EC_REQUEST_BUSY.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 * \retval -EINVAL Invalid input data, e.g. data size == 0.
 * \retval -ENOBUFS Reserved memory in ecrt_slave_config_create_sdo_request()
 *                  too small.
 */
EC_PUBLIC_API int ecrt_sdo_request_write(
        ec_sdo_request_t *req /**< SDO request. */
        );

/** Schedule an SDO read operation.
 *
 * \attention This method may not be called while ecrt_sdo_request_state()
 * returns EC_REQUEST_BUSY.
 *
 * \attention After calling this function, the return value of
 * ecrt_sdo_request_data() must be considered as invalid while
 * ecrt_sdo_request_state() returns EC_REQUEST_BUSY.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
```

```
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_sdo_request_read(
        ec_sdo_request_t *req /**< SDO request. */
        );

/*****************************************************************************
 * SoE request methods.
 ****************************************************************************/

/** Set the request's drive and Sercos ID numbers.
 *
 * \attention If the drive number and/or IDN is changed while
 * ecrt_soe_request_state() returns EC_REQUEST_BUSY, this may lead to
 * unexpected results.
 *
 * This method is meant to be called in realtime context (after master
 * activation). To initialize the SoE request, the drive_no and IDN can be
 * set via ecrt_slave_config_create_soe_request().
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_soe_request_idn(
        ec_soe_request_t *req, /**< IDN request. */
        uint8_t drive_no, /**< SDO index. */
        uint16_t idn /**< SoE IDN. */
        );

/** Set the timeout for an SoE request.
 *
 * If the request cannot be processed in the specified time, if will be marked
 * as failed.
 *
 * The timeout is permanently stored in the request object and is valid until
 * the next call of this method.
 *
 * The timeout should be defined in non-realtime context, but can also be
 * changed afterwards.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_soe_request_timeout(
        ec_soe_request_t *req, /**< SoE request. */
        uint32_t timeout /**< Timeout in milliseconds. Zero means no
                           timeout. */
        );

/** Access to the SoE request's data.
 *
 * This function returns a pointer to the request's internal IDN data memory.
 *
 * - After a read operation was successful, integer data can be evaluated
 *   using the EC_READ_*() macros as usual. Example:
 *   \code
 *   uint16_t value = EC_READ_U16(ecrt_soe_request_data(idn_req)));
 *   \endcode
 * - If a write operation shall be triggered, the data have to be written to
 *   the internal memory. Use the EC_WRITE_*() macros, if you are writing
 *   integer data. Be sure, that the data fit into the memory. The memory size
```

```
 *   is a parameter of ecrt_slave_config_create_soe_request().
 *   \code
 *   EC_WRITE_U16(ecrt_soe_request_data(idn_req), 0xFFFF);
 *   \endcode
 *
 * \attention The return value can be invalidated during a read operation,
 * because the internal IDN data memory could be re-allocated if the read IDN
 * data do not fit inside.
 *
 * This method is meant to be called in realtime context (after master
 * activation), but can also be used to initialize data before.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return Pointer to the internal IDN data memory.
 *
 */
EC_PUBLIC_API uint8_t *ecrt_soe_request_data(
        const ec_soe_request_t *req /**< SoE request. */
        );

/** Returns the current IDN data size.
 *
 * When the SoE request is created, the data size is set to the size of the
 * reserved memory. After a read operation the size is set to the size of the
 * read data. The size is not modified in any other situation.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return IDN data size in bytes.
 */
EC_PUBLIC_API size_t ecrt_soe_request_data_size(
        const ec_soe_request_t *req /**< SoE request. */
        );

/** Get the current state of the SoE request.
 *
 * \return Request state.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * In the user-space implementation, the method fetches the size of the
 * incoming data, so the request object is not const.
 *
 * \apiusage{master_op,rt_safe}
 */
EC_PUBLIC_API ec_request_state_t ecrt_soe_request_state(
#ifdef __KERNEL__
        const
#endif
        ec_soe_request_t *req /**< SoE request. */
        );

/** Schedule an SoE IDN write operation.
 *
 * \attention This method may not be called while ecrt_soe_request_state()
 * returns EC_REQUEST_BUSY.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 * \retval -EINVAL Invalid input data, e.g. data size == 0.
```

```
 * \retval -ENOBUFS Reserved memory in ecrt_slave_config_create_soe_request()
 *                too small.
 */
EC_PUBLIC_API int ecrt_soe_request_write(
        ec_soe_request_t *req /**< SoE request. */
        );

/** Schedule an SoE IDN read operation.
 *
 * \attention This method may not be called while ecrt_soe_request_state()
 * returns EC_REQUEST_BUSY.
 *
 * \attention After calling this function, the return value of
 * ecrt_soe_request_data() must be considered as invalid while
 * ecrt_soe_request_state() returns EC_REQUEST_BUSY.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_soe_request_read(
        ec_soe_request_t *req /**< SoE request. */
        );

/*****************************************************************************
 * VoE handler methods.
 ****************************************************************************/

/** Sets the VoE header for future send operations.
 *
 * A VoE message shall contain a 4-byte vendor ID, followed by a 2-byte vendor
 * type at as header. These numbers can be set with this function. The values
 * are valid and will be used for future send operations until the next call
 * of this method.
 *
 * This method is meant to be called in non-realtime context (before master
 * activation) to initialize the header data, but it is also safe to
 * change the header later on in realtime context.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_voe_handler_send_header(
        ec_voe_handler_t *voe, /**< VoE handler. */
        uint32_t vendor_id, /**< Vendor ID. */
        uint16_t vendor_type /**< Vendor-specific type. */
        );

/** Reads the header data of a received VoE message.
 *
 * This method can be used to get the received VoE header information after a
 * read operation has succeeded.
 *
 * The header information is stored at the memory given by the pointer
 * parameters.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
```

```
 */
EC_PUBLIC_API int ecrt_voe_handler_received_header(
        const ec_voe_handler_t *voe, /**< VoE handler. */
        uint32_t *vendor_id, /**< Vendor ID. */
        uint16_t *vendor_type /**< Vendor-specific type. */
        );

/** Access to the VoE handler's data.
 *
 * This function returns a pointer to the VoE handler's internal memory, that
 * points to the actual VoE data right after the VoE header (see
 * ecrt_voe_handler_send_header()).
 *
 * - After a read operation was successful, the memory contains the received
 *   data. The size of the received data can be determined via
 *   ecrt_voe_handler_data_size().
 * - Before a write operation is triggered, the data have to be written to the
 *   internal memory. Be sure, that the data fit into the memory. The reserved
 *   memory size is a parameter of ecrt_slave_config_create_voe_handler().
 *
 * \attention The returned pointer is not necessarily persistent: After a read
 * operation, the internal memory may have been reallocated. This can be
 * avoided by reserving enough memory via the \a size parameter of
 * ecrt_slave_config_create_voe_handler().
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return Pointer to the internal memory.
 */
EC_PUBLIC_API uint8_t *ecrt_voe_handler_data(
        const ec_voe_handler_t *voe /**< VoE handler. */
        );

/** Returns the current data size.
 *
 * The data size is the size of the VoE data without the header (see
 * ecrt_voe_handler_send_header()).
 *
 * When the VoE handler is created, the data size is set to the size of the
 * reserved memory. At a write operation, the data size is set to the number
 * of bytes to write. After a read operation the size is set to the size of
 * the read data. The size is not modified in any other situation.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return Data size in bytes.
 */
EC_PUBLIC_API size_t ecrt_voe_handler_data_size(
        const ec_voe_handler_t *voe /**< VoE handler. */
        );

/** Start a VoE write operation.
 *
 * After this function has been called, the ecrt_voe_handler_execute() method
 * must be called in every realtime cycle as long as it returns
 * EC_REQUEST_BUSY. No other operation may be started while the handler is
 * busy.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 * \retval -ENOBUFS Reserved memory in ecrt_slave_config_create_voe_handler
 *                  too small.
```

```
 */
EC_PUBLIC_API int ecrt_voe_handler_write(
        ec_voe_handler_t *voe, /**< VoE handler. */
        size_t size /**< Number of bytes to write (without the VoE header). */
        );

/** Start a VoE read operation.
 *
 * After this function has been called, the ecrt_voe_handler_execute() method
 * must be called in every realtime cycle as long as it returns
 * EC_REQUEST_BUSY. No other operation may be started while the handler is
 * busy.
 *
 * The state machine queries the slave's send mailbox for new data to be send
 * to the master. If no data appear within the EC_VOE_RESPONSE_TIMEOUT
 * (defined in master/voe_handler.c), the operation fails.
 *
 * On success, the size of the read data can be determined via
 * ecrt_voe_handler_data_size(), while the VoE header of the received data
 * can be retrieved with ecrt_voe_handler_received_header().
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_voe_handler_read(
        ec_voe_handler_t *voe /**< VoE handler. */
        );

/** Start a VoE read operation without querying the sync manager status.
 *
 * After this function has been called, the ecrt_voe_handler_execute() method
 * must be called in every realtime cycle as long as it returns
 * EC_REQUEST_BUSY. No other operation may be started while the handler is
 * busy.
 *
 * The state machine queries the slave by sending an empty mailbox. The slave
 * fills its data to the master in this mailbox. If no data appear within the
 * EC_VOE_RESPONSE_TIMEOUT (defined in master/voe_handler.c), the operation
 * fails.
 *
 * On success, the size of the read data can be determined via
 * ecrt_voe_handler_data_size(), while the VoE header of the received data
 * can be retrieved with ecrt_voe_handler_received_header().
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 */
EC_PUBLIC_API int ecrt_voe_handler_read_nosync(
        ec_voe_handler_t *voe /**< VoE handler. */
        );

/** Execute the handler.
 *
 * This method executes the VoE handler. It has to be called in every realtime
 * cycle as long as it returns EC_REQUEST_BUSY.
 *
 * \return Handler state.
 *
```

```
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 */
EC_PUBLIC_API ec_request_state_t ecrt_voe_handler_execute(
        ec_voe_handler_t *voe /**< VoE handler. */
        );

/*****************************************************************************
 * Register request methods.
 ****************************************************************************/

/** Access to the register request's data.
 *
 * This function returns a pointer to the request's internal memory.
 *
 * - After a read operation was successful, integer data can be evaluated
 *   using the EC_READ_*() macros as usual. Example:
 *   \code
 *   uint16_t value = EC_READ_U16(ecrt_reg_request_data(reg_request)));
 *   \endcode
 * - If a write operation shall be triggered, the data have to be written to
 *   the internal memory. Use the EC_WRITE_*() macros, if you are writing
 *   integer data. Be sure, that the data fit into the memory. The memory size
 *   is a parameter of ecrt_slave_config_create_reg_request().
 *   \code
 *   EC_WRITE_U16(ecrt_reg_request_data(reg_request), 0xFFFF);
 *   \endcode
 *
 * This method is meant to be called in realtime context (after master
 * activation), but can also be used to initialize data before.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \return Pointer to the internal memory.
 *
 */
EC_PUBLIC_API uint8_t *ecrt_reg_request_data(
        const ec_reg_request_t *req /**< Register request. */
        );

/** Get the current state of the register request.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return Request state.
 *
 */
EC_PUBLIC_API ec_request_state_t ecrt_reg_request_state(
        const ec_reg_request_t *req /**< Register request. */
    );

/** Schedule an register write operation.
 *
 * \attention This method may not be called while ecrt_reg_request_state()
 * returns EC_REQUEST_BUSY.
 *
 * \attention The \a size parameter is truncated to the size given at request
 * creation.
 *
 * This method is meant to be called in realtime context (after master
```

```
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 * \retval -ENOBUFS Reserved memory in ecrt_slave_config_create_reg_request
 *                  too small.
 */
EC_PUBLIC_API int ecrt_reg_request_write(
        ec_reg_request_t *req, /**< Register request. */
        uint16_t address, /**< Register address. */
        size_t size /**< Size to write. */
        );

/** Schedule a register read operation.
 *
 * \attention This method may not be called while ecrt_reg_request_state()
 * returns EC_REQUEST_BUSY.
 *
 * \attention The \a size parameter is truncated to the size given at request
 * creation.
 *
 * This method is meant to be called in realtime context (after master
 * activation).
 *
 * \apiusage{master_op,rt_safe}
 *
 * \return 0 on success, otherwise negative error code.
 * \retval -ENOBUFS Reserved memory in ecrt_slave_config_create_reg_request
 *                  too small.
 */
EC_PUBLIC_API int ecrt_reg_request_read(
        ec_reg_request_t *req, /**< Register request. */
        uint16_t address, /**< Register address. */
        size_t size /**< Size to write. */
        );

/*****************************************************************************
 * Bitwise read/write macros
 ****************************************************************************/

/** Read a certain bit of an EtherCAT data byte.
 *
 * \param DATA EtherCAT data pointer
 * \param POS bit position
 */
#define EC_READ_BIT(DATA, POS) ((*((uint8_t *) (DATA)) >> (POS)) & 0x01)

/** Write a certain bit of an EtherCAT data byte.
 *
 * \param DATA EtherCAT data pointer
 * \param POS bit position
 * \param VAL new bit value
 */
#define EC_WRITE_BIT(DATA, POS, VAL) \
    do { \
        if (VAL) *((uint8_t *) (DATA)) |=  (1 << (POS)); \
        else     *((uint8_t *) (DATA)) &= ~(1 << (POS)); \
    } while (0)

/*****************************************************************************
 * Byte-swapping functions for user space
 ****************************************************************************/

#ifndef __KERNEL__
```

```
#if __BYTE_ORDER == __LITTLE_ENDIAN

#define le16_to_cpu(x) x
#define le32_to_cpu(x) x
#define le64_to_cpu(x) x

#define cpu_to_le16(x) x
#define cpu_to_le32(x) x
#define cpu_to_le64(x) x

#elif __BYTE_ORDER == __BIG_ENDIAN

#define swap16(x) \
        ((uint16_t)( \
        (((uint16_t)(x) & 0x00ffU) << 8) | \
        (((uint16_t)(x) & 0xff00U) >> 8) ))
#define swap32(x) \
        ((uint32_t)( \
        (((uint32_t)(x) & 0x000000ffUL) << 24) | \
        (((uint32_t)(x) & 0x0000ff00UL) <<  8) | \
        (((uint32_t)(x) & 0x00ff0000UL) >>  8) | \
        (((uint32_t)(x) & 0xff000000UL) >> 24) ))
#define swap64(x) \
        ((uint64_t)( \
        (((uint64_t)(x) & 0x00000000000000ffULL) << 56) | \
        (((uint64_t)(x) & 0x000000000000ff00ULL) << 40) | \
        (((uint64_t)(x) & 0x0000000000ff0000ULL) << 24) | \
        (((uint64_t)(x) & 0x00000000ff000000ULL) <<  8) | \
        (((uint64_t)(x) & 0x000000ff00000000ULL) >>  8) | \
        (((uint64_t)(x) & 0x0000ff0000000000ULL) >> 24) | \
        (((uint64_t)(x) & 0x00ff000000000000ULL) >> 40) | \
        (((uint64_t)(x) & 0xff00000000000000ULL) >> 56) ))

#define le16_to_cpu(x) swap16(x)
#define le32_to_cpu(x) swap32(x)
#define le64_to_cpu(x) swap64(x)

#define cpu_to_le16(x) swap16(x)
#define cpu_to_le32(x) swap32(x)
#define cpu_to_le64(x) swap64(x)

#endif

#define le16_to_cpup(x) le16_to_cpu(*((uint16_t *)(x)))
#define le32_to_cpup(x) le32_to_cpu(*((uint32_t *)(x)))
#define le64_to_cpup(x) le64_to_cpu(*((uint64_t *)(x)))

#endif /* ifndef __KERNEL__ */

/*****************************************************************************
 * Read macros
 *****************************************************************************/

/** Read an 8-bit unsigned value from EtherCAT data.
 *
 * \return EtherCAT data value
 */
#define EC_READ_U8(DATA) \
    ((uint8_t) *((uint8_t *) (DATA)))

/** Read an 8-bit signed value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_S8(DATA) \
```

```
    ((int8_t) *((uint8_t *) (DATA)))

/** Read a 16-bit unsigned value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_U16(DATA) \
    ((uint16_t) le16_to_cpup((void *) (DATA)))

/** Read a 16-bit signed value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_S16(DATA) \
    ((int16_t) le16_to_cpup((void *) (DATA)))

/** Read a 32-bit unsigned value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_U32(DATA) \
    ((uint32_t) le32_to_cpup((void *) (DATA)))

/** Read a 32-bit signed value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_S32(DATA) \
    ((int32_t) le32_to_cpup((void *) (DATA)))

/** Read a 64-bit unsigned value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_U64(DATA) \
    ((uint64_t) le64_to_cpup((void *) (DATA)))

/** Read a 64-bit signed value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_S64(DATA) \
    ((int64_t) le64_to_cpup((void *) (DATA)))

/******************************************************************************
 * Floating-point read functions and macros (userspace only)
 *****************************************************************************/

#ifndef __KERNEL__

/** Read a 32-bit floating-point value from EtherCAT data.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \param data EtherCAT data pointer
 * \return EtherCAT data value
 */
EC_PUBLIC_API float ecrt_read_real(const void *data);

/** Read a 32-bit floating-point value from EtherCAT data.
```

```
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_REAL(DATA) ecrt_read_real(DATA)

/** Read a 64-bit floating-point value from EtherCAT data.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \param data EtherCAT data pointer
 * \return EtherCAT data value
 */
EC_PUBLIC_API double ecrt_read_lreal(const void *data);

/** Read a 64-bit floating-point value from EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \return EtherCAT data value
 */
#define EC_READ_LREAL(DATA) ecrt_read_lreal(DATA)

#endif // ifndef __KERNEL__

/*****************************************************************************
 * Write macros
 ****************************************************************************/

/** Write an 8-bit unsigned value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_U8(DATA, VAL) \
    do { \
        *((uint8_t *)(DATA)) = ((uint8_t) (VAL)); \
    } while (0)

/** Write an 8-bit signed value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_S8(DATA, VAL) EC_WRITE_U8(DATA, VAL)

/** Write a 16-bit unsigned value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_U16(DATA, VAL) \
    do { \
        *((uint16_t *) (DATA)) = cpu_to_le16((uint16_t) (VAL)); \
    } while (0)

/** Write a 16-bit signed value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_S16(DATA, VAL) EC_WRITE_U16(DATA, VAL)

/** Write a 32-bit unsigned value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
```

```
 */
#define EC_WRITE_U32(DATA, VAL) \
    do { \
        *((uint32_t *) (DATA)) = cpu_to_le32((uint32_t) (VAL)); \
    } while (0)

/** Write a 32-bit signed value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_S32(DATA, VAL) EC_WRITE_U32(DATA, VAL)

/** Write a 64-bit unsigned value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_U64(DATA, VAL) \
    do { \
        *((uint64_t *) (DATA)) = cpu_to_le64((uint64_t) (VAL)); \
    } while (0)

/** Write a 64-bit signed value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_S64(DATA, VAL) EC_WRITE_U64(DATA, VAL)

/*****************************************************************************
 * Floating-point write functions and macros (userspace only)
 ****************************************************************************/

#ifndef __KERNEL__

/** Write a 32-bit floating-point value to EtherCAT data.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \param data EtherCAT data pointer
 * \param value new value
 */
EC_PUBLIC_API void ecrt_write_real(void *data, float value);

/** Write a 32-bit floating-point value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
#define EC_WRITE_REAL(DATA, VAL) ecrt_write_real(DATA, VAL)

/** Write a 64-bit floating-point value to EtherCAT data.
 *
 * \apiusage{master_any,rt_safe}
 *
 * \param data EtherCAT data pointer
 * \param value new value
 */
EC_PUBLIC_API void ecrt_write_lreal(void *data, double value);

/** Write a 64-bit floating-point value to EtherCAT data.
 *
 * \param DATA EtherCAT data pointer
 * \param VAL new value
 */
```

```
#define EC_WRITE_LREAL(DATA, VAL) ecrt_write_lreal(DATA, VAL)

#endif // ifndef __KERNEL__

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

#ifdef __cplusplus
}
#endif

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

/** @} */

#endif
```

3.7 Userspace Application Example

There are multiple examples of how to use the application interface included in the master sources (under *examples/*). This section lists a very common application, the usage of the master from the user-space. The example code reserves an EtherCAT master, creates slave configurations and domains and goes into cyclic mode, where the `cyclic_task()` function is called repeatedly. For more general information on how to do real-time programming under Linux, please have a look at the code examples in https://gitlab.com/etherlab.org/realtime.

Listing 3.2: Userspace application example example/user/main.c

```
/*****************************************************************************
 *
 *  Copyright (C) 2007-2009  Florian Pose, Ingenieurgemeinschaft IgH
 *
 *  This file is part of the IgH EtherCAT Master.
 *
 *  The IgH EtherCAT Master is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License version 2, as
 *  published by the Free Software Foundation.
 *
 *  The IgH EtherCAT Master is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General
 *  Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with the IgH EtherCAT Master; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 ****************************************************************************/

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h> /* clock_gettime() */
```

```
#include <sys/mman.h> /* mlockall() */
#include <sched.h> /* sched_setscheduler() */

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

#include "ecrt.h"

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

/** Task period in ns. */
#define PERIOD_NS   (1000000)

#define MAX_SAFE_STACK (8 * 1024) /* The maximum stack size which is
                                     guranteed safe to access without
                                     faulting */

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

/* Constants */
#define NSEC_PER_SEC (1000000000)
#define FREQUENCY (NSEC_PER_SEC / PERIOD_NS)

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

// 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_ana_in = NULL;
static ec_slave_config_state_t sc_ana_in_state = {};

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

// process data
static uint8_t *domain1_pd = NULL;

#define BusCouplerPos  0, 0
#define DigOutSlavePos 0, 2
#define AnaInSlavePos  0, 3
#define AnaOutSlavePos 0, 4

#define Beckhoff_EK1100 0x00000002, 0x044c2c52
#define Beckhoff_EL2004 0x00000002, 0x07d43052
#define Beckhoff_EL2032 0x00000002, 0x07f03052
#define Beckhoff_EL3152 0x00000002, 0x0c503052
#define Beckhoff_EL3102 0x00000002, 0x0c1e3052
#define Beckhoff_EL4102 0x00000002, 0x10063052

// offsets for PDO entries
static unsigned int off_ana_in_status;
static unsigned int off_ana_in_value;
static unsigned int off_ana_out;
static unsigned int off_dig_out;

const static ec_pdo_entry_reg_t domain1_regs[] = {
    {AnaInSlavePos,  Beckhoff_EL3102, 0x3101, 1, &off_ana_in_status},
    {AnaInSlavePos,  Beckhoff_EL3102, 0x3101, 2, &off_ana_in_value},
    {AnaOutSlavePos, Beckhoff_EL4102, 0x3001, 1, &off_ana_out},
    {DigOutSlavePos, Beckhoff_EL2032, 0x3001, 1, &off_dig_out},
    {}
};

static unsigned int counter = 0;
```

```
static unsigned int blink = 0;

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

// Analog in --------------------------

static const ec_pdo_entry_info_t el3102_pdo_entries[] = {
    {0x3101, 1,  8}, // channel 1 status
    {0x3101, 2, 16}, // channel 1 value
    {0x3102, 1,  8}, // channel 2 status
    {0x3102, 2, 16}, // channel 2 value
    {0x6401, 1, 16}, // channel 1 value (alt.)
    {0x6401, 2, 16}  // channel 2 value (alt.)
};

static const ec_pdo_info_t el3102_pdos[] = {
    {0x1A00, 2, el3102_pdo_entries},
    {0x1A01, 2, el3102_pdo_entries + 2}
};

static const ec_sync_info_t el3102_syncs[] = {
    {2, EC_DIR_OUTPUT},
    {3, EC_DIR_INPUT, 2, el3102_pdos},
    {0xff}
};

// Analog out -------------------------

static const ec_pdo_entry_info_t el4102_pdo_entries[] = {
    {0x3001, 1, 16}, // channel 1 value
    {0x3002, 1, 16}, // channel 2 value
};

static const ec_pdo_info_t el4102_pdos[] = {
    {0x1600, 1, el4102_pdo_entries},
    {0x1601, 1, el4102_pdo_entries + 1}
};

static const ec_sync_info_t el4102_syncs[] = {
    {2, EC_DIR_OUTPUT, 2, el4102_pdos},
    {3, EC_DIR_INPUT},
    {0xff}
};

// Digital out ------------------------

static const ec_pdo_entry_info_t el2004_channels[] = {
    {0x3001, 1, 1}, // Value 1
    {0x3001, 2, 1}, // Value 2
    {0x3001, 3, 1}, // Value 3
    {0x3001, 4, 1}  // Value 4
};

static const ec_pdo_info_t el2004_pdos[] = {
    {0x1600, 1, &el2004_channels[0]},
    {0x1601, 1, &el2004_channels[1]},
    {0x1602, 1, &el2004_channels[2]},
    {0x1603, 1, &el2004_channels[3]}
};

static const ec_sync_info_t el2004_syncs[] = {
    {0, EC_DIR_OUTPUT, 4, el2004_pdos},
    {1, EC_DIR_INPUT},
    {0xff}
};
```

```c
/****************************************************************************/

void check_domain1_state(void)
{
    ec_domain_state_t ds;

    ecrt_domain_state(domain1, &ds);

    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);
    }

    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(void)
{
    ec_slave_config_state_t s;

    ecrt_slave_config_state(sc_ana_in, &s);

    if (s.al_state != sc_ana_in_state.al_state) {
        printf("AnaIn:␣State␣0x%02X.\n", s.al_state);
    }
    if (s.online != sc_ana_in_state.online) {
        printf("AnaIn:␣%s.\n", s.online ? "online" : "offline");
    }
    if (s.operational != sc_ana_in_state.operational) {
        printf("AnaIn:␣%soperational.\n", s.operational ? "" : "Not␣");
    }

    sc_ana_in_state = s;
}

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

void cyclic_task()
{
    // receive process data
    ecrt_master_receive(master);
```

```
    ecrt_domain_process(domain1);

    // check process data state
    check_domain1_state();

    if (counter) {
        counter--;
    } else { // do this at 1 Hz
        counter = FREQUENCY;

        // calculate new process data
        blink = !blink;

        // check for master state (optional)
        check_master_state();

        // check for slave configuration state(s) (optional)
        check_slave_config_states();
    }

#if 0
    // read process data
    printf("AnaIn:␣state␣%u␣value␣%u\n",
            EC_READ_U8(domain1_pd + off_ana_in_status),
            EC_READ_U16(domain1_pd + off_ana_in_value));
#endif

#if 1
    // write process data
    EC_WRITE_U8(domain1_pd + off_dig_out, blink ? 0x06 : 0x09);
#endif

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

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

void stack_prefault(void)
{
    unsigned char dummy[MAX_SAFE_STACK];

    memset(dummy, 0, MAX_SAFE_STACK);
}

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

int main(int argc, char **argv)
{
    ec_slave_config_t *sc;
    struct timespec wakeup_time;
    int ret = 0;

    master = ecrt_request_master(0);
    if (!master) {
        return -1;
    }

    domain1 = ecrt_master_create_domain(master);
    if (!domain1) {
        return -1;
    }

    if (!(sc_ana_in = ecrt_master_slave_config(
                    master, AnaInSlavePos, Beckhoff_EL3102))) {
```

```
        fprintf(stderr, "Failed to get slave configuration.\n");
        return -1;
    }

    printf("Configuring PDOs...\n");
    if (ecrt_slave_config_pdos(sc_ana_in, EC_END, el3102_syncs)) {
        fprintf(stderr, "Failed to configure PDOs.\n");
        return -1;
    }

    if (!(sc = ecrt_master_slave_config(
                    master, AnaOutSlavePos, Beckhoff_EL4102))) {
        fprintf(stderr, "Failed to get slave configuration.\n");
        return -1;
    }

    if (ecrt_slave_config_pdos(sc, EC_END, el4102_syncs)) {
        fprintf(stderr, "Failed to configure PDOs.\n");
        return -1;
    }

    if (!(sc = ecrt_master_slave_config(
                    master, DigOutSlavePos, Beckhoff_EL2032))) {
        fprintf(stderr, "Failed to get slave configuration.\n");
        return -1;
    }

    if (ecrt_slave_config_pdos(sc, EC_END, el2004_syncs)) {
        fprintf(stderr, "Failed to configure PDOs.\n");
        return -1;
    }

    // Create configuration for bus coupler
    sc = ecrt_master_slave_config(master, BusCouplerPos, Beckhoff_EK1100);
    if (!sc) {
        return -1;
    }

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

    printf("Activating master...\n");
    if (ecrt_master_activate(master)) {
        return -1;
    }

    if (!(domain1_pd = ecrt_domain_data(domain1))) {
        return -1;
    }

    /* Set priority */

    struct sched_param param = {};
    param.sched_priority = sched_get_priority_max(SCHED_FIFO);

    printf("Using priority %i.\n", param.sched_priority);
    if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
        perror("sched_setscheduler failed");
    }

    /* Lock memory */

    if (mlockall(MCL_CURRENT | MCL_FUTURE) == -1) {
        fprintf(stderr, "Warning: Failed to lock memory: %s\n",
```

```
                strerror(errno));
    }

    stack_prefault();

    printf("Starting␣RT␣task␣with␣dt=%u␣ns.\n", PERIOD_NS);

    clock_gettime(CLOCK_MONOTONIC, &wakeup_time);
    wakeup_time.tv_sec += 1; /* start in future */
    wakeup_time.tv_nsec = 0;

    while (1) {
        ret = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                &wakeup_time, NULL);
        if (ret) {
            fprintf(stderr, "clock_nanosleep():␣%s\n", strerror(ret));
            break;
        }

        cyclic_task();

        wakeup_time.tv_nsec += PERIOD_NS;
        while (wakeup_time.tv_nsec >= NSEC_PER_SEC) {
            wakeup_time.tv_nsec -= NSEC_PER_SEC;
            wakeup_time.tv_sec++;
        }
    }

    return ret;
}

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

4 Ethernet Devices

The EtherCAT protocol is based on the Ethernet standard, so a master relies on standard Ethernet hardware to communicate with the bus.

The term *device* is used as a synonym for Ethernet network interface hardware.

Native Ethernet Device Drivers There are native device driver modules (see section 4.2) that handle Ethernet hardware, which a master can use to connect to an EtherCAT bus. They offer their Ethernet hardware to the master module via the device interface (see section 4.6) and must be capable to prepare Ethernet devices either for EtherCAT (realtime) operation or for "normal" operation using the kernel's network stack. The advantage of this approach is that the master can operate nearly directly on the hardware, which allows a high performance. The disadvantage is, that there has to be an EtherCAT-capable version of the original Ethernet driver.

Generic Ethernet Device Driver From master version 1.5, there is a generic Ethernet device driver module (see section 4.3), that uses the lower layers of the network stack to connect to the hardware. The advantage is, that arbitrary Ethernet hardware can be used for EtherCAT operation, independently of the actual hardware driver (so all Linux Ethernet drivers are supported without modifications). The disadvantage is, that this approach does not support realtime extensions like RTAI, because the Linux network stack is addressed. Moreover the performance is a little worse than the native approach, because the Ethernet frame data have to traverse the network stack.

4.1 Network Driver Basics

EtherCAT relies on Ethernet hardware and the master needs a physical Ethernet device to communicate with the bus. Therefore it is necessary to understand how Linux handles network devices and their drivers, respectively.

Tasks of a Network Driver Network device drivers usually handle the lower two layers of the OSI model, that is the physical layer and the data-link layer. A network device itself natively handles the physical layer issues: It represents the hardware to connect to the medium and to send and receive data in the way, the physical layer protocol describes. The network device driver is responsible for getting data from the kernel's networking stack and forwarding it to the hardware, that does the physical transmission. If data is received by the hardware respectively, the driver is notified (usually by means of an interrupt) and has to read the data from the hardware memory and forward it to the network stack. There are a few more tasks, a network device driver has to handle, including queue control, statistics and device dependent features.

Driver Startup Usually, a driver searches for compatible devices on module loading. For PCI drivers, this is done by scanning the PCI bus and checking for known device IDs. If a device is found, data structures are allocated and the device is taken into operation.

Interrupt Operation A network device usually provides a hardware interrupt that is used to notify the driver of received frames and success of transmission, or errors, respectively. The driver has to register an interrupt service routine (ISR), that is executed each time, the hardware signals such an event. If the interrupt was thrown by the own device (multiple devices can share one hardware interrupt), the reason for the interrupt has to be determined by reading the device's interrupt register. For example, if the flag for received frames is set, frame data has to be copied from hardware to kernel memory and passed to the network stack.

The `net_device` Structure The driver registers a `net_device` structure for each device to communicate with the network stack and to create a "network interface". In case of an Ethernet driver, this interface appears as *ethX*, where X is a number assigned by the kernel on registration. The `net_device` structure receives events (either from userspace or from the network stack) via several callbacks, which have to be set before registration. Not every callback is mandatory, but for reasonable operation the ones below are needed in any case:

`open()` This function is called when network communication has to be started, for example after a command `ip link set ethX up` from userspace. Frame reception has to be enabled by the driver.

`stop()` The purpose of this callback is to "close" the device, i. e. make the hardware stop receiving frames.

`hard_start_xmit()` This function is called for each frame that has to be transmitted. The network stack passes the frame as a pointer to an `sk_buff` structure ("socket buffer", see below), which has to be freed after sending.

`get_stats()` This call has to return a pointer to the device's `net_device_stats` structure, which permanently has to be filled with frame statistics. This means, that every time a frame is received, sent, or an error happened, the appropriate counter in this structure has to be increased.

The actual registration is done with the `register_netdev()` call, unregistering is done with `unregister_netdev()`.

The `netif` Interface All other communication in the direction interface → network stack is done via the `netif_*()` calls. For example, on successful device opening, the network stack has to be notified, that it can now pass frames to the interface. This is done by calling `netif_start_queue()`. After this call, the `hard_start_xmit()` callback can be called by the network stack. Furthermore a network driver usually manages a frame transmission queue. If this gets filled up, the network stack has to be told to stop passing further frames for a while. This happens with a call to `netif_stop_queue()`. If some frames have been sent, and there is enough space again to queue new frames, this can be notified with `netif_wake_queue()`. Another important call is `netif_receive_skb()`[1]: It passes a frame to the network stack, that was just received by the device. Frame data has to be included in a so-called "socket buffer" for that (see below).

Socket Buffers Socket buffers are the basic data type for the whole network stack. They serve as containers for network data and are able to quickly add data headers and footers, or strip them off again. Therefore a socket buffer consists of an allocated buffer and several pointers that mark beginning of the buffer (`head`), beginning of data (`data`), end of data (`tail`) and end of buffer (`end`). In addition, a socket buffer holds network header information and (in case of received data) a pointer to the `net_device`, it was received on. There exist functions that create a socket buffer (`dev_alloc_skb()`), add data either from front (`skb_push()`) or back (`skb_put()`), remove data from front (`skb_pull()`) or back (`skb_trim()`), or delete the buffer (`kfree_skb()`). A socket buffer is passed from layer to layer, and is freed by the layer that uses it the last time. In case of sending, freeing has to be done by the network driver.

4.2 Native EtherCAT Device Drivers

There are a few requirements, that applies to Ethernet hardware when used with a native Ethernet driver with EtherCAT functionality.

Dedicated Hardware For performance and realtime purposes, the EtherCAT master needs direct and exclusive access to the Ethernet hardware. This implies that the network device must not be connected to the kernel's network stack as usual, because the kernel would try to use it as an ordinary Ethernet device.

Interrupt-less Operation EtherCAT frames travel through the logical EtherCAT ring and are then sent back to the master. Communication is highly deterministic: A frame is sent and will be received again after a constant time, so there is no need to

[1] This function is part of the NAPI ("New API"), that replaces the kernel 2.4 technique for interfacing to the network stack (with `netif_rx()`). NAPI is a technique to improve network performance on Linux. Read more in http://www.cyberus.ca/~hadi/usenix-paper.tgz.

notify the driver about frame reception: The master can instead query the hardware for received frames, if it expects them to be already received.

Figure 4.1 shows two workflows for cyclic frame transmission and reception with and without interrupts.

In the left workflow "Interrupt Operation", the data from the last cycle is first processed and a new frame is assembled with new datagrams, which is then sent. The cyclic work is done for now. Later, when the frame is received again by the hardware, an interrupt is triggered and the ISR is executed. The ISR will fetch the frame data from the hardware and initiate the frame dissection: The datagrams will be processed, so that the data is ready for processing in the next cycle.

In the right workflow "Interrupt-less Operation", there is no hardware interrupt enabled. Instead, the hardware will be polled by the master by executing the ISR. If the frame has been received in the meantime, it will be dissected. The situation is now the same as at the beginning of the left workflow: The received data is processed and a new frame is assembled and sent. There is nothing to do for the rest of the cycle.

The interrupt-less operation is desirable, because hardware interrupts are not conducive in improving the driver's realtime behaviour: Their indeterministic incidences contribute to increasing the jitter. Besides, if a realtime extension (like RTAI) is used, some additional effort would have to be made to prioritize interrupts.

Ethernet and EtherCAT Devices Another issue lies in the way Linux handles devices of the same type. For example, a PCI driver scans the PCI bus for devices it can handle. Then it registers itself as the responsible driver for all of the devices found. The problem is, that an unmodified driver can not be told to ignore a device because it will be used for EtherCAT later. There must be a way to handle multiple devices of the same type, where one is reserved for EtherCAT, while the other is treated as an ordinary Ethernet device.

For all this reasons, the author decided that the only acceptable solution is to modify standard Ethernet drivers in a way that they keep their normal functionality, but gain the ability to treat one or more of the devices as EtherCAT-capable.

Below are the advantages of this solution:

- No need to tell the standard drivers to ignore certain devices.
- One networking driver for EtherCAT and non-EtherCAT devices.
- No need to implement a network driver from scratch and running into issues, the former developers already solved.

The chosen approach has the following disadvantages:

- The modified driver gets more complicated, as it must handle EtherCAT and non-EtherCAT devices.
- Many additional case differentiations in the driver code.
- Changes and bug fixes on the standard drivers have to be ported to the EtherCAT-capable versions from time to time.

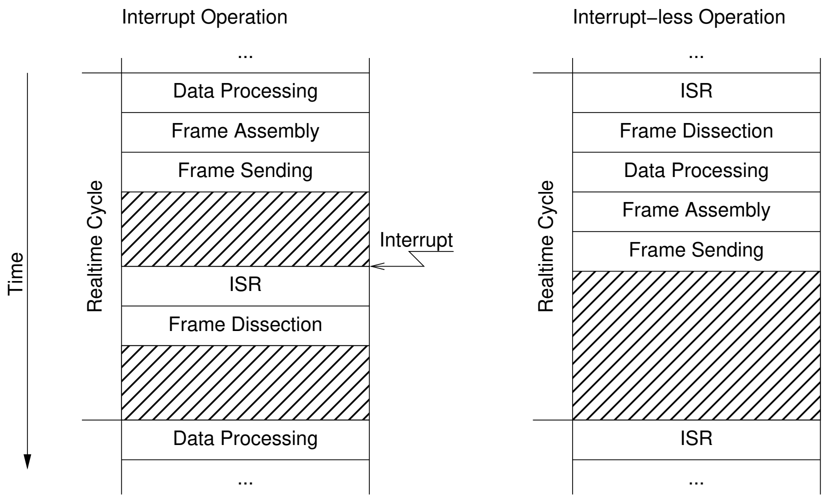

Figure 4.1: Interrupt Operation versus Interrupt-less Operation

4.3 Generic EtherCAT Device Driver

Since there are approaches to enable the complete Linux kernel for realtime operation [12], it is possible to operate without native implementations of EtherCAT-capable Ethernet device drivers and use the Linux network stack instead. Figure 2.1 shows the "Generic Ethernet Driver Module", that connects to local Ethernet devices via the network stack. The kernel module is named `ec_generic` and can be loaded after the master module like a native EtherCAT-capable Ethernet driver.

The generic device driver scans the network stack for interfaces, that have been registered by Ethernet device drivers. It offers all possible devices to the EtherCAT master. If the master accepts a device, the generic driver creates a packet socket (see `man 7 packet`) with `socket_type` set to `SOCK_RAW`, bound to that device. All functions of the device interface (see section 4.6) will then operate on that socket.

Below are the advantages of this solution:

- Any Ethernet hardware, that is covered by a Linux Ethernet driver can be used for EtherCAT.
- No modifications have to be made to the actual Ethernet drivers.

The generic approach has the following disadvantages:

- The performance is a little worse than the native approach, because the frame data have to traverse the lower layers of the network stack.
- It is not possible to use in-kernel realtime extensions like RTAI with the generic driver, because the network stack code uses dynamic memory allocations and other things, that could cause the system to freeze in realtime context.

Device Activation In order to send and receive frames through a socket, the Ethernet device linked to that socket has to be activated, otherwise all frames will be rejected. Activation has to take place before the master module is loaded and can happen in several ways:

- Ad-hoc, using the command `ip link set dev ethX up` (or the older `ifconfig ethX up`),
- Configured, depending on the distribution, for example using `ifcfg` files (`/etc/sysconfig/network/ifcfg-ethX`) in openSUSE and others. This is the better choice, if the EtherCAT master shall start at system boot time. Since the Ethernet device shall only be activated, but no IP address etc. shall be assigned, it is enough to use `STARTMODE=auto` as configuration.

4.4 Providing Ethernet Devices

After loading the master module, additional module(s) have to be loaded to offer devices to the master(s) (see section 4.6). The master module knows the devices to choose from the module parameters (see section 2.1). If the init script is used to start the master, the drivers and devices to use can be specified in the sysconfig file (see subsection 7.4.2).

Modules offering Ethernet devices can be

- native EtherCAT-capable network driver modules (see section 4.2) or
- the generic EtherCAT device driver module (see section 4.3).

4.5 Redundancy

Redundant bus operation means, that there is more than one Ethernet connection from the master to the slaves. Process data exchange datagrams are sent out on every master link, so that the exchange is still complete, even if the bus is disconnected somewhere in between.

Prerequisite for fully redundant bus operation is, that every slave can be reached by at least one master link. In this case a single connection failure (i. e. cable break) will never lead to incomplete process data. Double-faults can not be handled with two Ethernet devices.

Redundancy is configured with the `--with-devices` switch at configure time (see chapter 9) and using the `backup_devices` parameter of the `ec_master` kernel module (see section 2.1) or the appropriate variable `MASTERx_BACKUP` in the (sys-)config file (see subsection 7.4.2).

Bus scanning is done after a topology change on any Ethernet link. The application interface (see chapter 3) and the command-line tool (see section 7.1) both have methods to query the status of the redundant operation.

4.6 EtherCAT Device Interface

An anticipation to the section about the master module (section 2.1) has to be made in order to understand the way, a network device driver module can connect a device to a specific EtherCAT master.

The master module provides a "device interface" for network device drivers. To use this interface, a network device driver module must include the header *devices/ecdev.h*, coming with the EtherCAT master code. This header offers a function interface for EtherCAT devices. All functions of the device interface are named with the prefix `ecdev`.

The documentation of the device interface can be found in the header file or in the appropriate module of the interface documentation (see section 9.3 for generation instructions).

4.7 Patching Native Network Drivers

This section will describe, how to make a standard Ethernet driver EtherCAT-capable, using the native approach (see section 4.2). Unfortunately, there is no standard procedure to enable an Ethernet driver for use with the EtherCAT master, but there are a few common techniques.

1. A first simple rule is, that `netif_*()` calls must be avoided for all EtherCAT devices. As mentioned before, EtherCAT devices have no connection to the network stack, and therefore must not call its interface functions.
2. Another important thing is, that EtherCAT devices should be operated without interrupts. So any calls of registering interrupt handlers and enabling interrupts at hardware level must be avoided, too.
3. The master does not use a new socket buffer for each send operation: Instead there is a fix one allocated on master initialization. This socket buffer is filled with an EtherCAT frame with every send operation and passed to the `hard_start_xmit()` callback. For that it is necessary, that the socket buffer is not be freed by the network driver as usual.

An Ethernet driver usually handles several Ethernet devices, each described by a `net_device` structure with a `priv_data` field to attach driver-dependent data to the structure. To distinguish between normal Ethernet devices and the ones used by EtherCAT masters, the private data structure used by the driver could be extended by a pointer, that points to an `ec_device_t` object returned by `ecdev_offer()` (see section 4.6) if the device is used by a master and otherwise is zero.

The RealTek RTL-8139 Fast Ethernet driver is a "simple" Ethernet driver and can be taken as an example to patch new drivers. The interesting sections can be found by searching the string "ecdev" in the file *devices/8139too-2.6.24-ethercat.c*.

5 State Machines

Many parts of the EtherCAT master are implemented as *finite state machines* (FSMs). Though this leads to a higher grade of complexity in some aspects, is opens many new possibilities.

The below short code example exemplary shows how to read all slave states and moreover illustrates the restrictions of “sequential” coding:

```
ec_datagram_brd(datagram, 0x0130, 2); // prepare datagram
if (ec_master_simple_io(master, datagram)) return -1;
slave_states = EC_READ_U8(datagram->data); // process datagram
```

The *ec_master_simple_io()* function provides a simple interface for synchronously sending a single datagram and receiving the result[1]. Internally, it queues the specified datagram, invokes the *ec_master_send_datagrams()* function to send a frame with the queued datagram and then waits actively for its reception.

This sequential approach is very simple, reflecting in only three lines of code. The disadvantage is, that the master is blocked for the time it waits for datagram reception. There is no difficulty when only one instance is using the master, but if more instances want to (synchronously[2]) use the master, it is inevitable to think about an alternative to the sequential model.

Master access has to be sequentialized for more than one instance wanting to send and receive datagrams synchronously. With the present approach, this would result in having one phase of active waiting for each instance, which would be non-acceptable especially in realtime circumstances, because of the huge time overhead.

A possible solution is, that all instances would be executed sequentially to queue their datagrams, then give the control to the next instance instead of waiting for the datagram reception. Finally, bus IO is done by a higher instance, which means that all queued datagrams are sent and received. The next step is to execute all instances again, which then process their received datagrams and issue new ones.

This approach results in all instances having to retain their state, when giving the control back to the higher instance. It is quite obvious to use a *finite state machine* model in this case. section 5.1 will introduce some of the theory used, while the

[1] For all communication issues have been meanwhile sourced out into state machines, the function is deprecated and stopped existing. Nevertheless it is adequate for showing it’s own restrictions.

[2] At this time, synchronous master access will be adequate to show the advantages of an FSM. The asynchronous approach will be discussed in section 6.1

listings below show the basic approach by coding the example from above as a state machine:

```
// state 1
ec_datagram_brd(datagram, 0x0130, 2); // prepare datagram
ec_master_queue(master, datagram); // queue datagram
next_state = state_2;
// state processing finished
```

After all instances executed their current state and queued their datagrams, these are sent and received. Then the respective next states are executed:

```
// state 2
if (datagram->state != EC_DGRAM_STATE_RECEIVED) {
        next_state = state_error;
        return; // state processing finished
}
slave_states = EC_READ_U8(datagram->data); // process datagram
// state processing finished.
```

See section 5.2 for an introduction to the state machine programming concept used in the master code.

5.1 State Machine Theory

A finite state machine [9] is a model of behavior with inputs and outputs, where the outputs not only depend on the inputs, but the history of inputs. The mathematical definition of a finite state machine (or finite automaton) is a six-tuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$, with

- the input alphabet Σ, with $\Sigma \neq \emptyset$, containing all input symbols,
- the output alphabet Γ, with $\Gamma \neq \emptyset$, containing all output symbols,
- the set of states S, with $S \neq \emptyset$,
- the set of initial states s_0 with $s_0 \subseteq S, s_0 \neq \emptyset$
- the transition function $\delta : S \times \Sigma \rightarrow S \times \Gamma$
- the output function ω.

The state transition function δ is often specified by a *state transition table*, or by a *state transition diagram*. The transition table offers a matrix view of the state machine behavior (see Table 5.1). The matrix rows correspond to the states ($S = \{s_0, s_1, s_2\}$) and the columns correspond to the input symbols ($\Gamma = \{a, b, \varepsilon\}$). The table contents in a certain row i and column j then represent the next state (and possibly the output) for the case, that a certain input symbol σ_j is read in the state s_i.

Table 5.1: A typical state transition table

	a	b	ε
s_0	s_1	s_1	s_2
s_1	s_2	s_1	s_0
s_2	s_0	s_0	s_0

The state diagram for the same example looks like the one in Figure 5.1. The states are represented as circles or ellipses and the transitions are drawn as arrows between them. Close to a transition arrow can be the condition that must be fulfilled to allow the transition. The initial state is marked by a filled black circle with an arrow pointing to the respective state.

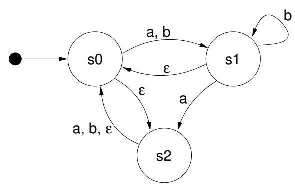

Figure 5.1: A typical state transition diagram

Deterministic and non-deterministic state machines A state machine can be deterministic, meaning that for one state and input, there is one (and only one) following state. In this case, the state machine has exactly one starting state. Non-deterministic state machines can have more than one transitions for a single state-input combination. There is a set of starting states in the latter case.

Moore and Mealy machines There is a distinction between so-called *Moore machines*, and *Mealy machines*. Mathematically spoken, the distinction lies in the output function ω: If it only depends on the current state ($\omega : S \to \Gamma$), the machine corresponds to the "Moore Model". Otherwise, if ω is a function of a state and the input alphabet ($\omega : S \times \Sigma \to \Gamma$) the state machine corresponds to the "Mealy model". Mealy machines are the more practical solution in most cases, because their design allows machines with a minimum number of states. In practice, a mixture of both models is often used.

Misunderstandings about state machines There is a phenomenon called "state explosion", that is often taken as a counter-argument against general use of state machines in complex environments. It has to be mentioned, that this point is misleading [10]. State explosions happen usually as a result of a bad state machine design: Common mistakes are storing the present values of all inputs in a state, or not dividing a complex state machine into simpler sub state machines. The EtherCAT master uses several state machines, that are executed hierarchically and so serve as sub state machines. These are also described below.

5.2 The Master's State Model

This section will introduce the techniques used in the master to implement state machines.

State Machine Programming There are certain ways to implement a state machine in C code. An obvious way is to implement the different states and actions by one big case differentiation:

```
enum {STATE_1, STATE_2, STATE_3};
int state = STATE_1;

void state_machine_run(void *priv_data) {
        switch (state) {
                case STATE_1:
                        action_1();
                        state = STATE_2;
                        break;
                case STATE_2:
                        action_2()
                        if (some_condition) state = STATE_1;
                        else state = STATE_3;
                        break;
                case STATE_3:
                        action_3();
```

```
                        state = STATE_1;
                        break;
        }
}
```

For small state machines, this is an option. The disadvantage is, that with an increasing number of states the code soon gets complex and an additional case differentiation is executed each run. Besides, lots of indentation is wasted.

The method used in the master is to implement every state in an own function and to store the current state function with a function pointer:

```
void (*state)(void *) = state1;

void state_machine_run(void *priv_data) {
        state(priv_data);
}

void state1(void *priv_data) {
        action_1();
        state = state2;
}

void state2(void *priv_data) {
        action_2();
        if (some_condition) state = state1;
        else state = state2;
}

void state3(void *priv_data) {
        action_3();
        state = state1;
}
```

In the master code, state pointers of all state machines[3] are gathered in a single object of the `ec_fsm_master_t` class. This is advantageous, because there is always one instance of every state machine available and can be started on demand.

Mealy and Moore If a closer look is taken to the above listing, it can be seen that the actions executed (the "outputs" of the state machine) only depend on the current state. This accords to the "Moore" model introduced in section 5.1. As mentioned, the "Mealy" model offers a higher flexibility, which can be seen in the listing below:

```
void state7(void *priv_data) {
```

[3]All except for the EoE state machine, because multiple EoE slaves have to be handled in parallel. For this reason each EoE handler object has its own state pointer.

```
2         if (some_condition) {
3                 action_7a();
4                 state = state1;
5         }
6         else {
7                 action_7b();
8                 state = state8;
9         }
10 }
```

③ + ⑦ The state function executes the actions depending on the state transition, that is about to be done.

The most flexible alternative is to execute certain actions depending on the state, followed by some actions dependent on the state transition:

```
void state9(void *priv_data) {
        action_9();
        if (some_condition) {
                action_9a();
                state = state7;
        }
        else {
                action_9b();
                state = state10;
        }
}
```

This model is often used in the master. It combines the best aspects of both approaches.

Using Sub State Machines To avoid having too much states, certain functions of the EtherCAT master state machine have been sourced out into sub state machines. This helps to encapsulate the related workflows and moreover avoids the "state explosion" phenomenon described in section 5.1. If the master would instead use one big state machine, the number of states would be a multiple of the actual number. This would increase the level of complexity to a non-manageable grade.

Executing Sub State Machines If a state machine starts to execute a sub state machine, it usually remains in one state until the sub state machine terminates. This is usually done like in the listing below, which is taken out of the slave configuration state machine code:

```
void ec_fsm_slaveconf_safeop(ec_fsm_t *fsm)
{
        fsm->change_state(fsm); // execute state change
```

```
4                                       // sub state machine
5
6           if (fsm->change_state == ec_fsm_error) {
7                   fsm->slave_state = ec_fsm_end;
8                   return;
9           }
10
11          if (fsm->change_state != ec_fsm_end) return;
12
13          // continue state processing
14          ...
```

(3) `change_state` is the state pointer of the state change state machine. The state function, the pointer points on, is executed...

(6) ...either until the state machine terminates with the error state ...

(11) ...or until the state machine terminates in the end state. Until then, the "higher" state machine remains in the current state and executes the sub state machine again in the next cycle.

State Machine Descriptions The below sections describe every state machine used in the EtherCAT master. The textual descriptions of the state machines contain references to the transitions in the corresponding state transition diagrams, that are marked with an arrow followed by the name of the successive state. Transitions caused by trivial error cases (i. e. no response from slave) are not described explicitly. These transitions are drawn as dashed arrows in the diagrams.

5.3 The Master State Machine

The master state machine is executed in the context of the master thread. Figure 5.2 shows its transition diagram. Its purposes are:

Bus monitoring The bus topology is monitored. If it changes, the bus is (re-)scanned.

Slave configuration The application-layer states of the slaves are monitored. If a slave is not in the state it supposed to be, the slave is (re-)configured.

Request handling Requests (either originating from the application or from external sources) are handled. A request is a job that the master shall process asynchronously, for example an SII access, SDO access, or similar.

5.4 The Slave Scan State Machine

The slave scan state machine, which can be seen in Figure 5.3, leads through the process of reading desired slave information.

The scan process includes the following steps:

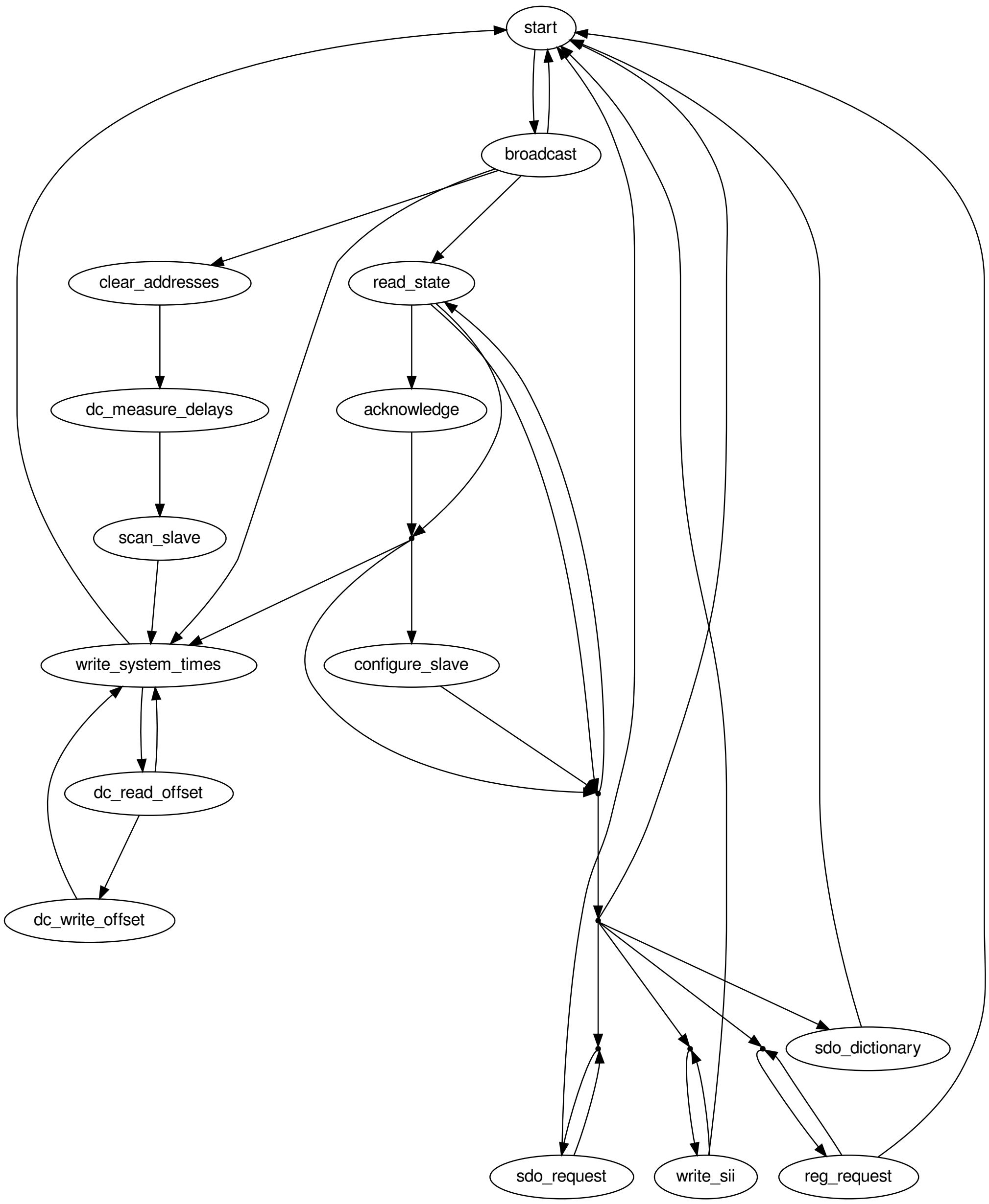

Figure 5.2: Transition diagram of the master state machine

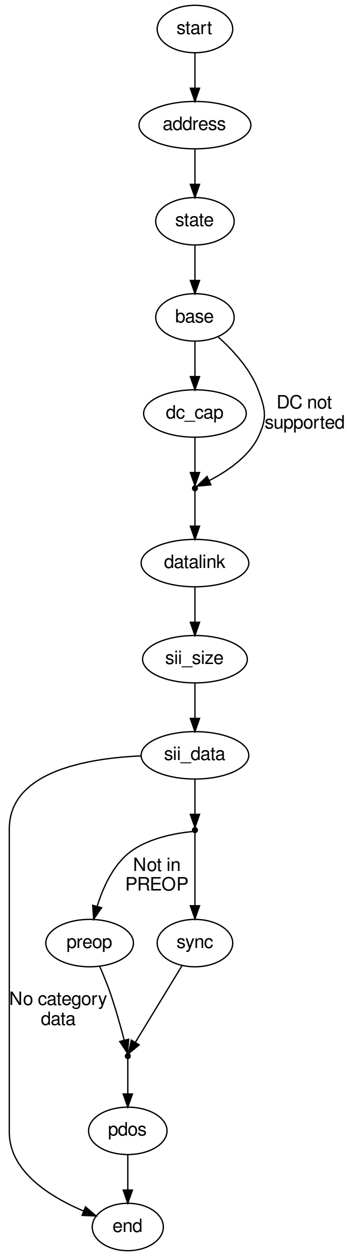

Figure 5.3: Transition diagram of the slave scan state machine

Node Address The node address is set for the slave, so that it can be node-addressed for all following operations.

AL State The initial application-layer state is read.

Base Information Base information (like the number of supported FMMUs) is read from the lower physical memory.

Data Link Information about the physical ports is read.

SII Size The size of the SII contents is determined to allocate SII image memory.

SII Data The SII contents are read into the master's image.

PREOP If the slave supports CoE, it is set to PREOP state using the State change FSM (see section 5.6) to enable mailbox communication and read the PDO configuration via CoE.

PDOs The PDOs are read via CoE (if supported) using the PDO Reading FSM (see section 5.8). If this is successful, the PDO information from the SII (if any) is overwritten.

5.5 The Slave Configuration State Machine

The slave configuration state machine, which can be seen in Figure 5.4, leads through the process of configuring a slave and bringing it to a certain application-layer state.

INIT The state change FSM is used to bring the slave to the INIT state.

FMMU Clearing To avoid that the slave reacts on any process data, the FMMU configuration are cleared. If the slave does not support FMMUs, this state is skipped. If INIT is the requested state, the state machine is finished.

Mailbox Sync Manager Configuration If the slaves support mailbox communication, the mailbox sync managers are configured. Otherwise this state is skipped.

PREOP The state change FSM is used to bring the slave to PREOP state. If this is the requested state, the state machine is finished.

SDO Configuration If there is a slave configuration attached (see section 3.1), and there are any SDO configurations that are provided by the application, these are sent to the slave.

PDO Configuration The PDO configuration state machine is executed to apply all necessary PDO configurations.

PDO Sync Manager Configuration If any PDO sync managers exist, they are configured.

FMMU Configuration If there are FMMUs configurations supplied by the application (i. e. if the application registered PDO entries), they are applied.

SAFEOP The state change FSM is used to bring the slave to SAFEOP state. If this is the requested state, the state machine is finished.

OP The state change FSM is used to bring the slave to OP state. If this is the requested state, the state machine is finished.

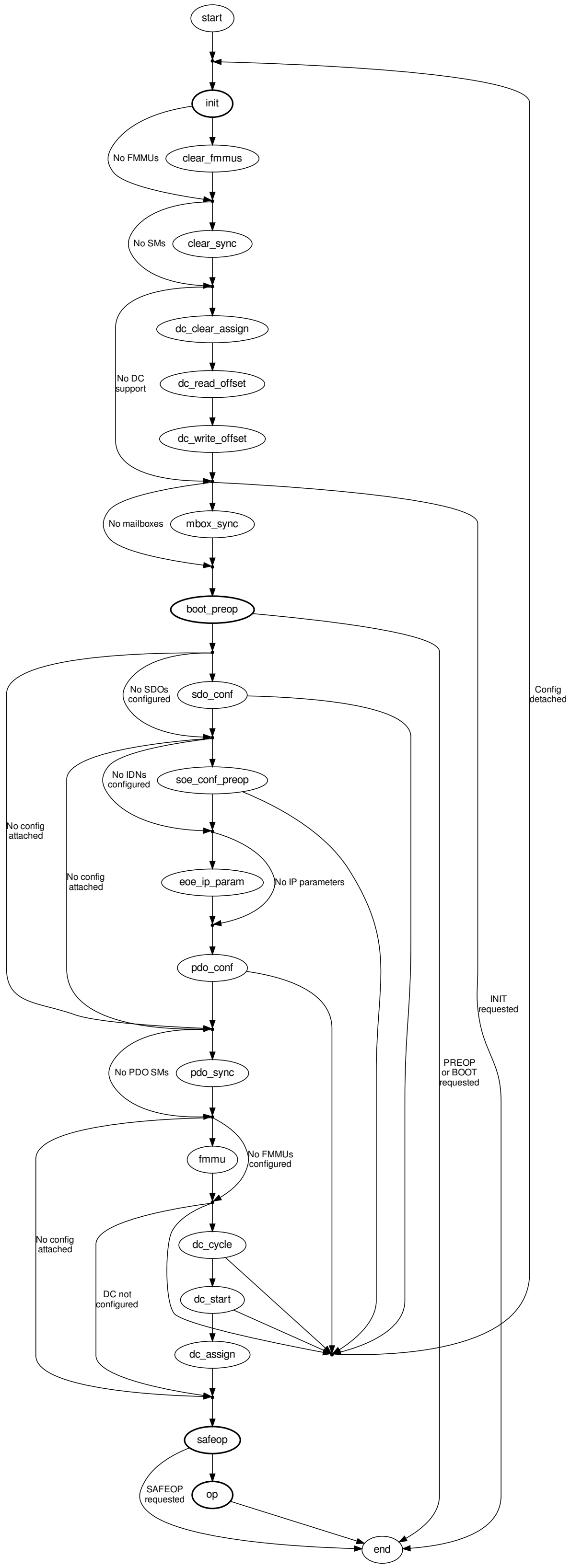

Figure 5.4: Transition diagram of the slave configuration state machine

5.6 The State Change State Machine

The state change state machine, which can be seen in Figure 5.5, leads through the process of changing a slave's application-layer state. This implements the states and transitions described in [3, sec. 6.4.1].

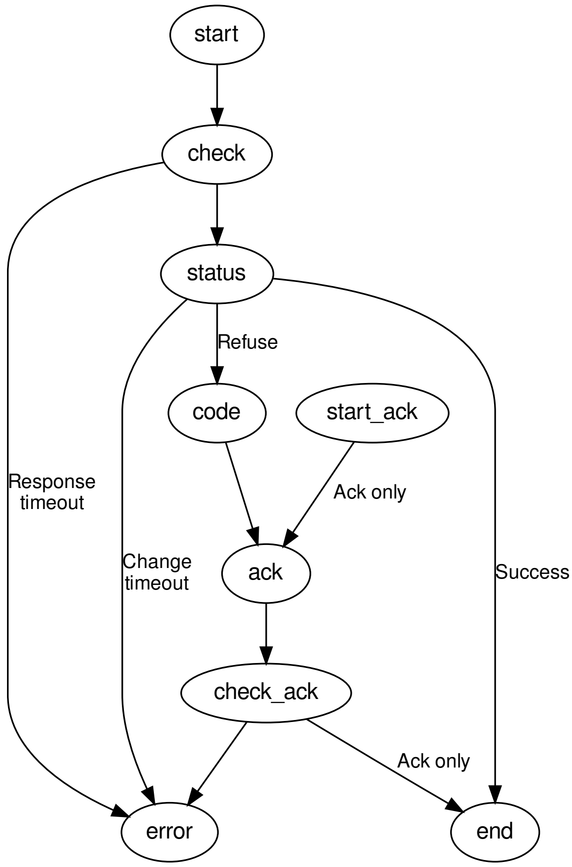

Figure 5.5: Transition Diagram of the State Change State Machine

Start The new application-layer state is requested via the "AL Control Request" register (see [3, sec. 5.3.1]).

Check for Response Some slave need some time to respond to an AL state change command, and do not respond for some time. For this case, the command is issued again, until it is acknowledged.

Check AL Status If the AL State change datagram was acknowledged, the "AL Control Response" register (see [3, sec. 5.3.2]) must be read out until the slave changes the AL state.

AL Status Code If the slave refused the state change command, the reason can be read from the "AL Status Code" field in the "AL State Changed" registers (see [3, sec. 5.3.3]).

Acknowledge State If the state change was not successful, the master has to acknowledge the old state by writing to the "AL Control request" register again.

Check Acknowledge After sending the acknowledge command, it has to read out the "AL Control Response" register again.

The "start_ack" state is a shortcut in the state machine for the case, that the master wants to acknowledge a spontaneous AL state change, that was not requested.

5.7 The SII State Machine

The SII state machine (shown in Figure 5.6) implements the process of reading or writing SII data via the Slave Information Interface described in [2, sec. 6.4].

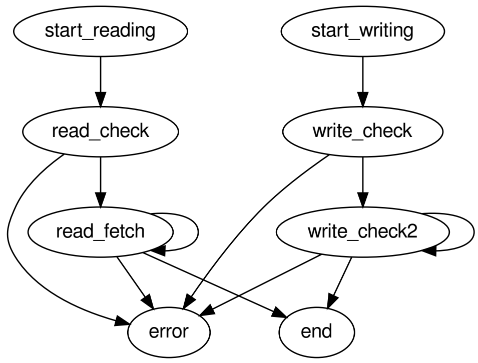

Figure 5.6: Transition Diagram of the SII State Machine

This is how the reading part of the state machine works:

Start Reading The read request and the requested word address are written to the SII attribute.

Check Read Command If the SII read request command has been acknowledged, a timer is started. A datagram is issued, that reads out the SII attribute for state and data.

Fetch Data If the read operation is still busy (the SII is usually implemented as an E^2PROM), the state is read again. Otherwise the data are copied from the datagram.

The writing part works nearly similar:

Start Writing A write request, the target address and the data word are written to the SII attribute.

Check Write Command If the SII write request command has been acknowledged, a timer is started. A datagram is issued, that reads out the SII attribute for the state of the write operation.

Wait while Busy If the write operation is still busy (determined by a minimum wait time and the state of the busy flag), the state machine remains in this state to avoid that another write operation is issued too early.

5.8 The PDO State Machines

The PDO state machines are a set of state machines that read or write the PDO assignment and the PDO mapping via the "CoE Communication Area" described in [3, sec. 5.6.7.4]. For the object access, the CANopen over EtherCAT access primitives are used (see section 6.2), so the slave must support the CoE mailbox protocol.

PDO Reading FSM This state machine (Figure 5.7) has the purpose to read the complete PDO configuration of a slave. It reads the PDO assignment for each Sync Manager and uses the PDO Entry Reading FSM (Figure 5.8) to read the mapping for each assigned PDO.

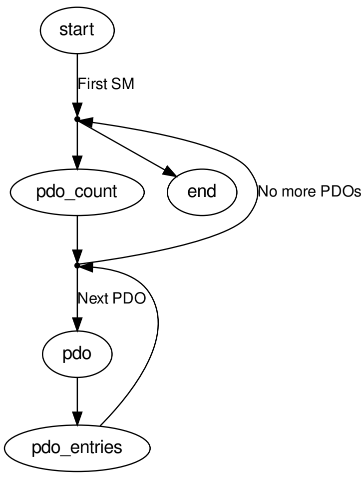

Figure 5.7: Transition Diagram of the PDO Reading State Machine

Basically it reads the every Sync manager's PDO assignment SDO's (`0x1C1x`) number of elements to determine the number of assigned PDOs for this sync manager and then reads out the subindices of the SDO to get the assigned PDO's indices. When a PDO index is read, the PDO Entry Reading FSM is executed to read the PDO's mapped PDO entries.

PDO Entry Reading FSM This state machine (Figure 5.8) reads the PDO mapping (the PDO entries) of a PDO. It reads the respective mapping SDO (`0x1600` – `0x17ff`, or `0x1a00` – `0x1bff`) for the given PDO by reading first the subindex zero (number of elements) to determine the number of mapped PDO entries. After that, each subindex is read to get the mapped PDO entry index, subindex and bit size.

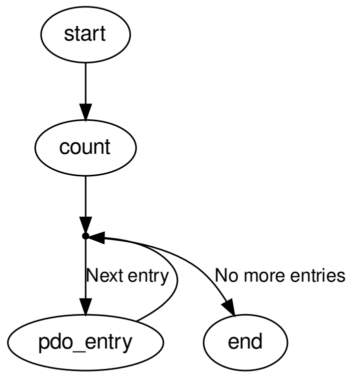

Figure 5.8: Transition Diagram of the PDO Entry Reading State Machine

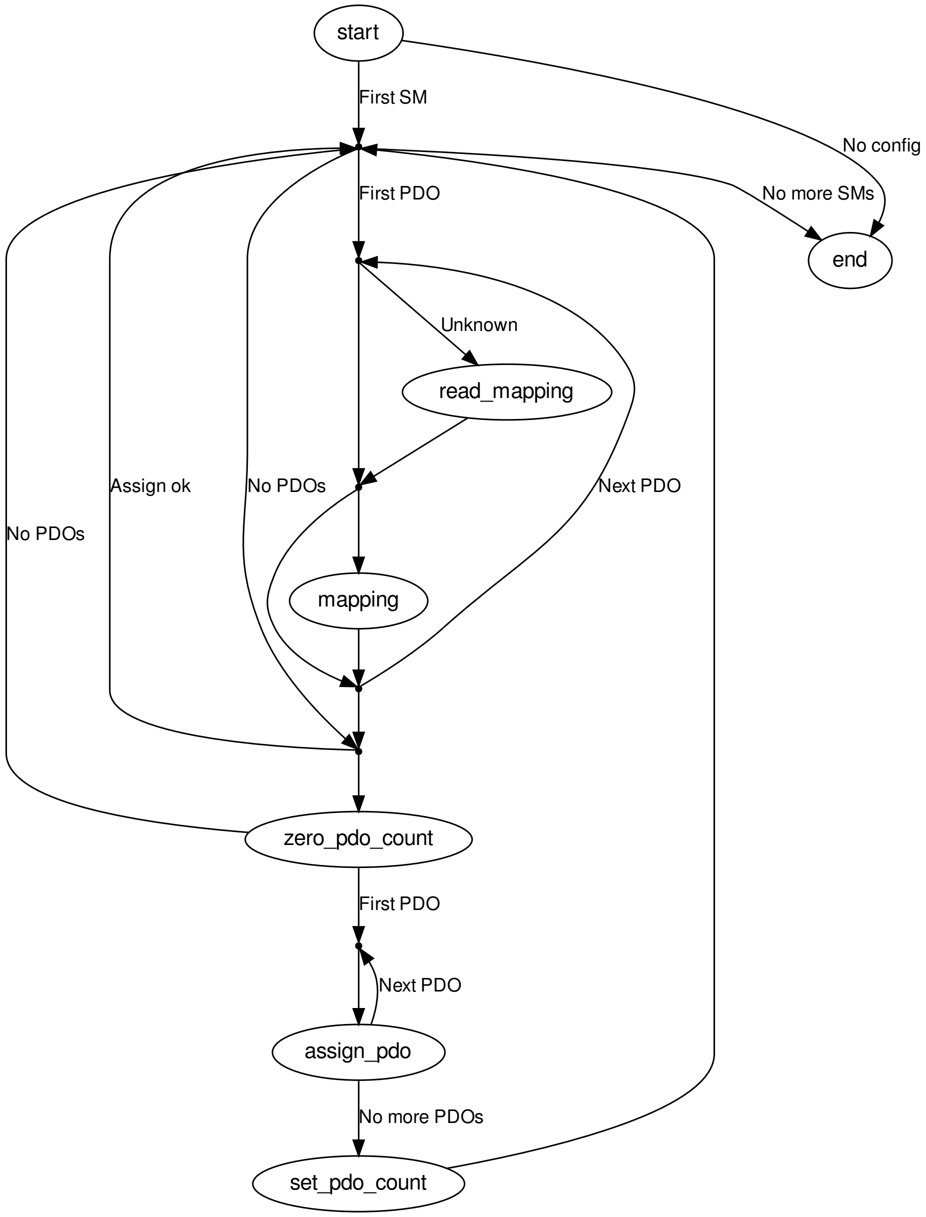

Figure 5.9: Transition Diagram of the PDO Configuration State Machine

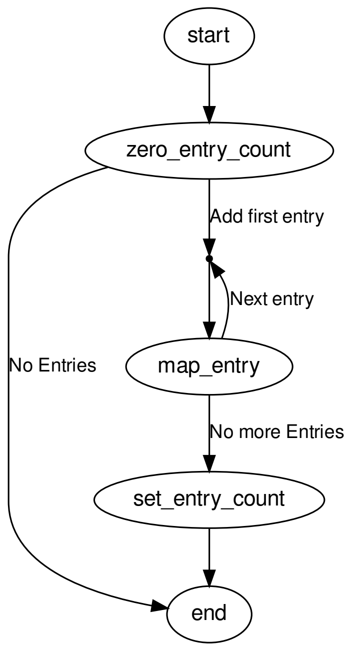

Figure 5.10: Transition Diagram of the PDO Entry Configuration State Machine

6 Mailbox Protocol Implementations

The EtherCAT master implements the CANopen over EtherCAT (CoE), Ethernet over EtherCAT (EoE), File-access over EtherCAT (FoE), Vendor-specific over EtherCAT (VoE) and Servo Profile over EtherCAT (SoE) mailbox protocols. See the below sections for details.

6.1 Ethernet over EtherCAT (EoE)

The EtherCAT master implements the Ethernet over EtherCAT mailbox protocol [3, sec. 5.7] to enable the tunneling of Ethernet frames to special slaves, that can either have physical Ethernet ports to forward the frames to, or have an own IP stack to receive the frames.

Virtual Network Interfaces The master creates a virtual EoE network interface for every EoE-capable slave. These interfaces are called either

eoeXsY for a slave without an alias address (see subsection 7.1.2), where X is the master index and Y is the slave's ring position, or

eoeXaY for a slave with a non-zero alias address, where X is the master index and Y is the decimal alias address.

For some hints on how to configure these virtual interfaces, see subsection 6.1.1.

Frames sent to these interfaces are forwarded to the associated slaves by the master. Frames, that are received by the slaves, are fetched by the master and forwarded to the virtual interfaces.

This bears the following advantages:

- Flexibility: The user can decide, how the EoE-capable slaves are interconnected with the rest of the world.
- Standard tools can be used to monitor the EoE activity and to configure the EoE interfaces.
- The Linux kernel's layer-2-bridging implementation (according to the IEEE 802.1D MAC Bridging standard) can be used natively to bridge Ethernet traffic between EoE-capable slaves.
- The Linux kernel's network stack can be used to route packets between EoE-capable slaves and to track security issues, just like having physical network interfaces.

EoE Handlers The virtual EoE interfaces and the related functionality is encapsulated in the `ec_eoe_t` class. An object of this class is called "EoE handler". For example the master does not create the network interfaces directly: This is done inside the constructor of an EoE handler. An EoE handler additionally contains a frame queue. Each time, the kernel passes a new socket buffer for sending via the interface's `hard_start_xmit()` callback, the socket buffer is queued for transmission by the EoE state machine (see below). If the queue gets filled up, the passing of new socket buffers is suspended with a call to `netif_stop_queue()`.

Creation of EoE Handlers During bus scanning (see section 5.4), the master determines the supported mailbox protocols for each slave. This is done by examining the "Supported Mailbox Protocols" mask field at word address 0x001C of the SII. If bit 1 is set, the slave supports the EoE protocol. In this case, an EoE handler is created for that slave.

EoE State Machine Every EoE handler owns an EoE state machine, that is used to send frames to the corresponding slave and receive frames from the it via the EoE communication primitives. This state machine is showed in Figure 6.1.

RX_START The beginning state of the EoE state machine. A mailbox check datagram is sent, to query the slave's mailbox for new frames. → RX_CHECK

RX_CHECK The mailbox check datagram is received. If the slave's mailbox did not contain data, a transmit cycle is started. → TX_START

If there are new data in the mailbox, a datagram is sent to fetch the new data. → RX_FETCH

RX_FETCH The fetch datagram is received. If the mailbox data do not contain a "EoE Fragment request" command, the data are dropped and a transmit sequence is started. → TX_START

If the received Ethernet frame fragment is the first fragment, a new socket buffer is allocated. In either case, the data are copied into the correct position of the socket buffer.

If the fragment is the last fragment, the socket buffer is forwarded to the network stack and a transmit sequence is started. → TX_START

Otherwise, a new receive sequence is started to fetch the next fragment. → RX_START

TX_START The beginning state of a transmit sequence. It is checked, if the transmission queue contains a frame to send. If not, a receive sequence is started. → RX_START

If there is a frame to send, it is dequeued. If the queue was inactive before (because it was full), the queue is woken up with a call to *netif_wake_queue()*. The first fragment of the frame is sent. → TX_SENT

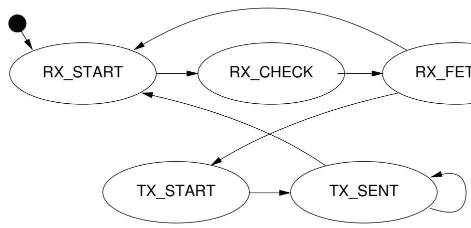

Figure 6.1: Transition Diagram of the EoE State Machine

TX_SENT It is checked, if the first fragment was sent successfully. If the current frame consists of further fragments, the next one is sent. → TX_SENT

If the last fragment was sent, a new receive sequence is started. → RX_START

EoE Processing To execute the EoE state machine of every active EoE handler, there must be a cyclic process. The easiest solution would be to execute the EoE state machines synchronously with the master state machine (see section 5.3). This approach has the following disadvantage:

Only one EoE fragment could be sent or received every few cycles. This causes the data rate to be very low, because the EoE state machines are not executed in the time between the application cycles. Moreover, the data rate would be dependent on the period of the application task.

To overcome this problem, an own cyclic process is needed to asynchronously execute the EoE state machines. For that, the master owns a kernel timer, that is executed each timer interrupt. This guarantees a constant bandwidth, but poses the new problem of concurrent access to the master. The locking mechanisms needed for this are introduced in section 3.4.

6.1.1 EoE Interface Configuration

The configuration of the EoE network interfaces is a matter of using standard Linux networking infrastructure commands like `ifconfig`, `ip` and `brctl`. Though this lies not in the scope of this document, some hints and examples are provided in this section.

In the below examples it is assumed, that there are two slaves (0 and 1) with EoE support in the bus. The first decision to make is whether to use a bridged or routed environment.

Bridging A common solution is to create a bridge containing all EoE interfaces:

```
$ brctl addbr br0
$ ip addr add 192.168.100.1/24 dev br0
$ brctl addif br0 eoe0s0
$ brctl addif br0 eoe0s1
```

The above example allows to access IPv4 nodes using subnet 192.168.100.0/24 connected to the EtherCAT bus via EoE. Please note, that the example only contains ad-hoc configuration commands: If the bus topology changes, the EoE interfaces are re-created and have to be added to the bridge again. Therefore it is highly recommended to use the networking configuration infrastructure of the used Linux distribution to store this configuration permanently, so that appearing EoE devices are added automatically.

Routing Another possibility is to create an IP subnet for each EoE interface:

```
$ ip addr add 192.168.200.1/24 dev eoe0s0
$ ip addr add 192.168.201.1/24 dev eoe0s1
$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

This example is again only an ad-hoc configuration (see above). Please note, that it is necessary to set the default gateways properly on the IP nodes connected to the EoE slaves, if they shall be able to communicate between the different EoE interfaces / IP networks.

Setting IP Parameters If IP address and other parameters of the EoE remote nodes (not the EoE interfaces on the master side) have to be set, this can be achieved via the `ethercat ip` command-line tool (see subsection 7.1.13).

6.2 CANopen over EtherCAT (CoE)

The CANopen over EtherCAT protocol [3, sec. 5.6] is used to configure slaves and exchange data objects on application level.

SDO Download State Machine The best time to apply SDO configurations is during the slave's PREOP state, because mailbox communication is already possible and slave's application will start with updating input data in the succeeding SAFEOP state. Therefore the SDO configuration has to be part of the slave configuration state machine (see section 5.5): It is implemented via an SDO download state machine, that is executed just before entering the slave's SAFEOP state. In this way, it is guaranteed that the SDO configurations are applied each time, the slave is reconfigured.

The transition diagram of the SDO Download state machine can be seen in Figure 6.2.

START The beginning state of the CoE download state machine. The "SDO Download Normal Request" mailbox command is sent. → REQUEST

REQUEST It is checked, if the CoE download request has been received by the slave. After that, a mailbox check command is issued and a timer is started. → CHECK

CHECK If no mailbox data is available, the timer is checked.

- If it timed out, the SDO download is aborted. → ERROR
- Otherwise, the mailbox is queried again. → CHECK

If the mailbox contains new data, the response is fetched. → RESPONSE

RESPONSE If the mailbox response could not be fetched, the data is invalid, the wrong protocol was received, or a "Abort SDO Transfer Request" was received, the SDO download is aborted. → ERROR

If a "SDO Download Normal Response" acknowledgement was received, the SDO download was successful. → END

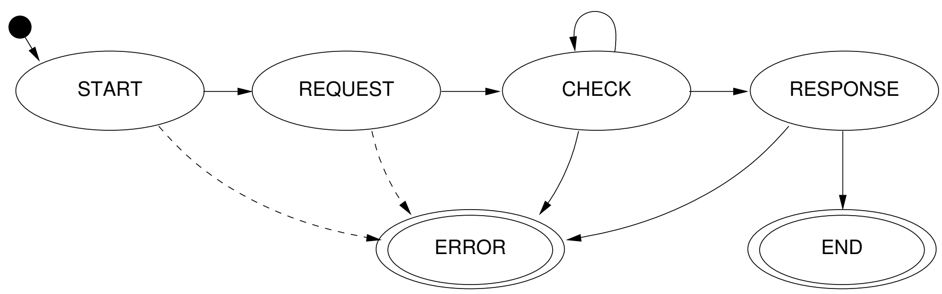

Figure 6.2: Transition diagram of the CoE download state machine

END The SDO download was successful.

ERROR The SDO download was aborted due to an error.

6.3 Vendor specific over EtherCAT (VoE)

The VoE protocol opens the possibility to implement a vendor-specific mailbox communication protocol. VoE mailbox messages are prepended by a VoE header containing a 32-bit vendor ID and a 16-bit vendor-type. There are no more constraints regarding this protocol.

The EtherCAT master allows to create multiple VoE handlers per slave configuration via the application interface (see chapter 3). These handlers contain the state machine necessary for the communication via VoE.

For more information about using VoE handlers, see section 3.3 or the example applications provided in the *examples/* subdirectory.

6.4 Servo Profile over EtherCAT (SoE)

The SoE protocol implements the Service Channel layer, specified in IEC 61800-7 [16] via EtherCAT mailboxes.

The SoE protocol is quite similar to the CoE protocol (see section 6.2). Instead of SDO indices and subindices, so-called identification numbers (IDNs) identify parameters.

The implementation covers the “SCC Read” and “SCC Write” primitives, each with the ability to fragment data.

There are several ways to use the SoE implementation:

- Reading and writing IDNs via the command-line tool (see subsection 7.1.21).
- Storing configurations for arbitrary IDNs via the application interface (see chapter 3, i. e. `ecrt_slave_config_idn()`). These configurations are written to the slave during configuration in PREOP state, before going to SAFEOP.
- The user-space library (see section 7.2), offers functions to read/write IDNs in blocking mode (`ecrt_master_read_idn()`, `ecrt_master_write_idn()`).

7 Userspace Interfaces

For the master runs as a kernel module, accessing it is natively limited to analyzing Syslog messages and controlling using *modutils*.

It was necessary to implement further interfaces, that make it easier to access the master from userspace and allow a finer influence. It should be possible to view and to change special parameters at runtime.

Bus visualization is another point: For development and debugging purposes it is necessary to show the connected slaves with a single command, for instance (see section 7.1).

The application interface has to be available in userspace, to allow userspace programs to use EtherCAT master functionality. This was implemented via a character device and a userspace library (see section 7.2).

Another aspect is automatic startup and configuration. The master must be able to automatically start up with a persistent configuration (see section 7.4).

A last thing is monitoring EtherCAT communication. For debugging purposes, there had to be a way to analyze EtherCAT datagrams. The best way would be with a popular network analyzer, like Wireshark [8] or others (see section 7.5).

This chapter covers all these points and introduces the interfaces and tools to make all that possible.

7.1 Command-line Tool

7.1.1 Character Devices

Each master instance will get a character device as a userspace interface. The devices are named */dev/EtherCATx*, where $x \in \{0 \ldots n\}$ is the index of the master.

Device Node Creation The character device nodes are automatically created, if the `udev` Package is installed. See section 9.5 for how to install and configure it.

7.1.2 Setting Alias Addresses

```
ethercat alias [OPTIONS] <ALIAS>

Write alias addresses.

Arguments:
  ALIAS must be an unsigned 16 bit number. Zero means
        removing an alias address.

If multiple slaves are selected, the --force option
is required.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --force    -f          Acknowledge writing aliases of
                         multiple slaves.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.3 Displaying the Bus Configuration

```
ethercat config [OPTIONS]

Show slave configurations.

Without the --verbose option, slave configurations are
output one-per-line. Example:

1001:0  0x0000003b/0x02010000  3  OP
|       |                      |  |
|       |                      |  \- Application-layer
|       |                      |     state of the attached
|       |                      |     slave, or '-', if no
|       |                      |     slave is attached.
|       |                      \- Absolute decimal ring
|       |                         position of the attached
|       |                         slave, or '-' if none
|       |                         attached.
|       \- Expected vendor ID and product code (both
|          hexadecimal).
\- Alias address and relative position (both decimal).

With the --verbose option given, the configured PDOs and
SDOs are output in addition.

Configuration selection:
  Slave configurations can be selected with
```

```
  the --alias and --position parameters as follows:

  1) If neither the --alias nor the --position option
     is given, all slave configurations are displayed.
  2) If only the --position option is given, an alias
     of zero is assumed (see 4)).
  3) If only the --alias option is given, all slave
     configurations with the given alias address
     are displayed.
  4) If both the --alias and the --position option are
     given, the selection can match a single
     configuration, that is displayed, if it exists.

Command-specific options:
  --alias    -a <alias>  Configuration alias (see above).
  --position -p <pos>    Relative position (see above).
  --verbose  -v          Show detailed configurations.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.4 Display CRC Error Counters

```
ethercat crc
ethercat crc reset

CRC error register diagnosis.

CRC - CRC Error Counter                0x300, 0x302, 0x304, 0x306
PHY - Physical Interface Error Counter 0x301, 0x303, 0x305, 0x307
FWD - Forwarded RX Error Counter       0x308, 0x309, 0x30a, 0x30b
NXT - Next slave
```

7.1.5 Output PDO information in C Language

```
ethercat cstruct [OPTIONS]

Generate slave PDO information in C language.

The output C code can be used directly with the
ecrt_slave_config_pdos() function of the application
interface.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.6 Displaying Process Data

```
ethercat data [OPTIONS]

Output binary domain process data.

Data of multiple domains are concatenated.

Command-specific options:
  --domain -d <index>  Positive numerical domain index.
                       If omitted, data of all domains
                       are output.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.7 Setting a Master's Debug Level

```
ethercat debug <LEVEL>

Set the master's debug level.

Debug messages are printed to syslog.

Arguments:
  LEVEL can have one of the following values:
        0 for no debugging output,
        1 for some debug messages, or
        2 for printing all frame contents (use with caution!).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.8 Configured Domains

```
ethercat domains [OPTIONS]

Show configured domains.

Without the --verbose option, the domains are displayed
one-per-line. Example:

Domain0: LogBaseAddr 0x00000000, Size   6, WorkingCounter 0/1

The domain's base address for the logical datagram
(LRD/LWR/LRW) is displayed followed by the domain's
process data size in byte. The last values are the current
datagram working counter sum and the expected working
counter sum. If the values are equal, all PDOs were
exchanged during the last cycle.
```

```
If the --verbose option is given, the participating slave
configurations/FMMUs and the current process data are
additionally displayed:

Domain1: LogBaseAddr 0x00000006, Size   6, WorkingCounter 0/1
  SlaveConfig 1001:0, SM3 ( Input), LogAddr 0x00000006, Size 6
    00 00 00 00 00 00

The process data are displayed as hexadecimal bytes.

Command-specific options:
  --domain  -d <index>  Positive numerical domain index.
                        If omitted, all domains are
                        displayed.

  --verbose -v          Show FMMUs and process data
                        in addition.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.9 SDO Access

```
ethercat download [OPTIONS] <INDEX> <SUBINDEX> <VALUE>
 [OPTIONS] <INDEX> <VALUE>

Write an SDO entry to a slave.

This command requires a single slave to be selected.

The data type of the SDO entry is taken from the SDO
dictionary by default. It can be overridden with the
--type option. If the slave does not support the SDO
information service or the SDO is not in the dictionary,
the --type option is mandatory.

The second call (without <SUBINDEX>) uses the complete
access method.

These are valid data types to use with
the --type option:
  bool,
  int8, int16, int32, int64,
  uint8, uint16, uint32, uint64,
  float, double,
  string, octet_string, unicode_string.
For sign-and-magnitude coding, use the following types:
  sm8, sm16, sm32, sm64

Arguments:
  INDEX    is the SDO index and must be an unsigned
```

```
          16 bit number.
  SUBINDEX is the SDO entry subindex and must be an
          unsigned 8 bit number.
  VALUE    is the value to download and must correspond
          to the SDO entry datatype (see above). Use
          '-' to read from standard input.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --type     -t <type>   SDO entry data type (see above).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.


ethercat upload [OPTIONS] <INDEX> <SUBINDEX>

Read an SDO entry from a slave.

This command requires a single slave to be selected.

The data type of the SDO entry is taken from the SDO
dictionary by default. It can be overridden with the
--type option. If the slave does not support the SDO
information service or the SDO is not in the dictionary,
the --type option is mandatory.

These are valid data types to use with
the --type option:
  bool,
  int8, int16, int32, int64,
  uint8, uint16, uint32, uint64,
  float, double,
  string, octet_string, unicode_string.
For sign-and-magnitude coding, use the following types:
  sm8, sm16, sm32, sm64

Arguments:
  INDEX    is the SDO index and must be an unsigned
          16 bit number.
  SUBINDEX is the SDO entry subindex and must be an
          unsigned 8 bit number.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --type     -t <type>   SDO entry data type (see above).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.10 EoE Statistics

```
ethercat eoe

Display Ethernet over EtherCAT statictics.

The TxRate and RxRate are displayed in Byte/s.
```

7.1.11 File-Access over EtherCAT

```
ethercat foe_read [OPTIONS] <SOURCEFILE>

Read a file from a slave via FoE.

This command requires a single slave to be selected.

Arguments:
  SOURCEFILE is the name of the source file on the slave.

Command-specific options:
  --output-file -o <file>   Local target filename. If
                            '-' (default), data are
                            printed to stdout.
  --alias       -a <alias>
  --position    -p <pos>    Slave selection. See the help
                            of the 'slaves' command.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

```
ethercat foe_write [OPTIONS] <FILENAME>

Store a file on a slave via FoE.

This command requires a single slave to be selected.

Arguments:
  FILENAME can either be a path to a file, or '-'. In
           the latter case, data are read from stdin and
           the --output-file option has to be specified.

Command-specific options:
  --output-file -o <file>   Target filename on the slave.
                            If the FILENAME argument is
                            '-', this is mandatory.
                            Otherwise, the basename() of
                            FILENAME is used by default.
  --alias       -a <alias>
  --position    -p <pos>    Slave selection. See the help
                            of the 'slaves' command.
```

```
Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.12 Creating Topology Graphs

```
ethercat graph [OPTIONS]
ethercat graph [OPTIONS] <INFO>

Output the bus topology as a graph.

The bus is output in DOT language (see
http://www.graphviz.org/doc/info/lang.html), which can
be processed with the tools from the Graphviz
package. Example:

  ethercat graph | dot -Tsvg > bus.svg

See 'man dot' for more information.

Additional information at edges and nodes is selected via
the first argument:
  DC  - DC timing
  CRC - CRC error register information
```

7.1.13 Setting Ethernet-over-EtherCAT IP Parameters

Slaves can have own IP stack implementations accessible via EoE. Since some of them do not provide other mechanisms to set IP parameters (because they only have an EtherCAT interface), there is a possibility to set the below parameters via EoE:

- Ethernet MAC address[1],
- IPv4 address,
- IPv4 subnet mask,
- IPv4 default gateway,
- IPv4 DNS server,
- DNS host name.

```
ethercat ip [OPTIONS] <ARGS>

Set EoE IP parameters.

This command requires a single slave to be selected.

IP parameters can be appended as argument pairs:

  ip_address <IPv4>[/prefix]  IP address (optionally with
```

[1] The MAC address of the virtual EoE remote interface, not the one of the EtherCAT interface.

```
                                  decimal subnet prefix)
  mac_address <MAC>             Link-layer address (may contain
                                  colons or hyphens)
  default_gateway <IPv4>        Default gateway
  dns_address <IPv4>            DNS server address
  hostname <hostname>           Host name (max. 32 byte)

IPv4 adresses can be given either in dot notation or as
hostnames, which will be automatically resolved.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.14 Master and Ethernet Devices

```
ethercat master [OPTIONS]

Show master and Ethernet device information.

Command-specific options:
  --master -m <indices>  Master indices. A comma-separated
                         list with ranges is supported.
                         Example: 1,4,5,7-9. Default: - (all).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.15 Sync Managers, PDOs and PDO Entries

```
ethercat pdos [OPTIONS]

List Sync managers, PDO assignment and mapping.

For the default skin (see --skin option) the information
is displayed in three layers, which are
indented accordingly:

1) Sync managers - Contains the sync manager information
   from the SII: Index, physical start address, default
   size, control register and enable word. Example:

   SM3: PhysAddr 0x1100, DefaultSize 0, ControlRegister 0x20, Enable
      1

2) Assigned PDOs - PDO direction, hexadecimal index and
```

```
   the PDO name, if available. Note that a 'Tx' and 'Rx'
   are seen from the slave's point of view. Example:

   TxPDO 0x1a00 "Channel1"

3) Mapped PDO entries - PDO entry index and subindex (both
   hexadecimal), the length in bit and the description, if
   available. Example:

   PDO entry 0x3101:01, 8 bit, "Status"

Note, that the displayed PDO assignment and PDO mapping
information can either originate from the SII or from the
CoE communication area.

The "etherlab" skin outputs a template configuration
for EtherLab's generic EtherCAT slave block.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --skin     -s <skin>   Choose output skin. Possible values are
                         "default" and "etherlab".

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.16 Register Access

```
ethercat reg_read [OPTIONS] <ADDRESS> [SIZE]

Output a slave's register contents.

This command requires a single slave to be selected.

Arguments:
  ADDRESS is the register address. Must
          be an unsigned 16 bit number.
  SIZE    is the number of bytes to read and must also be
          an unsigned 16 bit number. ADDRESS plus SIZE
          may not exceed 64k. The size is ignored (and
          can be omitted), if a selected data type
          implies a size.

These are valid data types to use with
the --type option:
  bool,
  int8, int16, int32, int64,
  uint8, uint16, uint32, uint64,
  float, double,
  string, octet_string, unicode_string.
```

```
For sign-and-magnitude coding, use the following types:
  sm8, sm16, sm32, sm64

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --type     -t <type>   Data type (see above).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

```
ethercat reg_write [OPTIONS] <OFFSET> <DATA>

Write data to a slave's registers.

This command requires a single slave to be selected.

Arguments:
  ADDRESS is the register address to write to.
  DATA    depends on whether a datatype was specified
          with the --type option: If not, DATA must be
          either a path to a file with data to write,
          or '-', which means, that data are read from
          stdin. If a datatype was specified, VALUE is
          interpreted respective to the given type.

These are valid data types to use with
the --type option:
  bool,
  int8, int16, int32, int64,
  uint8, uint16, uint32, uint64,
  float, double,
  string, octet_string, unicode_string.
For sign-and-magnitude coding, use the following types:
  sm8, sm16, sm32, sm64

Command-specific options:
  --alias     -a <alias>
  --position  -p <pos>    Slave selection. See the help of
                          the 'slaves' command.
  --type      -t <type>   Data type (see above).
  --emergency -e          Send as emergency request.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.17 Trigger a Bus Scan

```
ethercat rescan
```

```
Rescan the bus.

Command a bus rescan. Gathered slave information will be
forgotten and slaves will be read in again.
```

7.1.18 SDO Dictionary

```
ethercat sdos [OPTIONS]

List SDO dictionaries.

SDO dictionary information is displayed in two layers,
which are indented accordingly:

1) SDOs - Hexadecimal SDO index and the name. Example:

   SDO 0x1018, "Identity object"

2) SDO entries - SDO index and SDO entry subindex (both
   hexadecimal) followed by the access rights (see
   below), the data type, the length in bit, and the
   description. Example:

   0x1018:01, rwrwrw, uint32, 32 bit, "Vendor id"

The access rights are specified for the AL states PREOP,
SAFEOP and OP. An 'r' means, that the entry is readable
in the corresponding state, an 'w' means writable,
respectively. If a right is not granted, a dash '-' is
shown.

If the --quiet option is given, only the SDOs are output.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --quiet    -q          Only output SDOs (without the
                         SDO entries).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.19 SII Access

It is possible to directly read or write the complete SII contents of the slaves. This was introduced for the reasons below:

- The format of the SII data is still in development and categories can be added in the future. With read and write access, the complete memory contents can be easily backed up and restored.

- Some SII data fields have to be altered (like the alias address). A quick writing must be possible for that.
- Through reading access, analyzing category data is possible from userspace.

```
ethercat sii_read [OPTIONS]

Output a slave's SII contents.

This command requires a single slave to be selected.

Without the --verbose option, binary SII contents are
output.

With the --verbose option given, a textual representation
of the data is output, that is separated by SII category
names.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --verbose  -v          Output textual data with
                         category names.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

Reading out SII data is as easy as other commands. Though the data are in binary format, analysis is easier with a tool like *hexdump*:

```
$ ethercat sii_read --position 3 | hexdump
0000000 0103 0000 0000 0000 0000 0000 0000 008c
0000010 0002 0000 3052 07f0 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
...
```

Backing up SII contents can easily done with a redirection:

```
$ ethercat sii_read --position 3 > sii-of-slave3.bin
```

To download SII contents to a slave, writing access to the master's character device is necessary (see subsection 7.1.1).

```
ethercat sii_write [OPTIONS] <FILENAME>

Write SII contents to a slave.

This command requires a single slave to be selected.

The file contents are checked for validity and integrity.
These checks can be overridden with the --force option.
```

```
Arguments:
  FILENAME must be a path to a file that contains a
           positive number of words. If it is '-',
           data are read from stdin.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --force    -f          Override validity checks.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

```
# ethercat sii_write --position 3 sii-of-slave3.bin
```

The SII contents will be checked for validity and then sent to the slave. The write operation may take a few seconds.

7.1.20 Slaves on the Bus

Slave information can be gathered with the subcommand `slaves`:

```
ethercat slaves [OPTIONS]

Display slaves on the bus.

If the --verbose option is not given, the slaves are
displayed one-per-line. Example:

1  5555:0  PREOP  +  EL3162 2C. Ana. Input 0-10V
|  |    |  |      |  |
|  |    |  |      |  \- Name from the SII if available,
|  |    |  |      |     otherwise vendor ID and product
|  |    |  |      |     code (both hexadecimal).
|  |    |  |      \- Error flag. '+' means no error,
|  |    |  |         'E' means that scan or
|  |    |  |         configuration failed.
|  |    |  \- Current application-layer state.
|  |    \- Decimal relative position to the last
|  |       slave with an alias address set.
|  \- Decimal alias address of this slave (if set),
|     otherwise of the last slave with an alias set,
|     or zero, if no alias was encountered up to this
|     position.
\- Absolute ring position in the bus.

If the --verbose option is given, a detailed (multi-line)
description is output for each slave.
```

```
Slave selection:
  Slaves for this and other commands can be selected with
  the --alias and --position parameters as follows:

  1) If neither the --alias nor the --position option
     is given, all slaves are selected.
  2) If only the --position option is given, it is
     interpreted as an absolute ring position and
     a slave with this position is matched.
  3) If only the --alias option is given, all slaves
     with the given alias address and subsequent
     slaves before a slave with a different alias
     address match (use -p0 if only the slaves
     with the given alias are desired, see 4)).
  4) If both the --alias and the --position option are
     given, the latter is interpreted as relative
     position behind any slave with the given alias.

Command-specific options:
  --alias    -a <alias>  Slave alias (see above).
  --position -p <pos>    Slave position (see above).
  --verbose  -v          Show detailed slave information.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

Below is a typical output:

```
$ ethercat slaves
0      0:0  PREOP  +  EK1100 Ethernet Kopplerklemme (2A E-Bus)
1   5555:0  PREOP  +  EL3162 2K. Ana. Eingang 0-10V
2   5555:1  PREOP  +  EL4102 2K. Ana. Ausgang 0-10V
3   5555:2  PREOP  +  EL2004 4K. Dig. Ausgang 24V, 0,5A
```

7.1.21 SoE IDN Access

```
ethercat soe_read [OPTIONS] <IDN>
ethercat soe_read [OPTIONS] <DRIVE> <IDN>

Read an SoE IDN from a slave.

This command requires a single slave to be selected.

Arguments:
  DRIVE    is the drive number (0 - 7). If omitted, 0 is assumed.
  IDN      is the IDN and must be either an unsigned
           16 bit number acc. to IEC 61800-7-204:
             Bit 15: (0) Standard data, (1) Product data
             Bit 14 - 12: Parameter set (0 - 7)
             Bit 11 - 0: Data block number
```

```
           or a string like 'P-0-150'.

Data of the given IDN are read and displayed according to
the given datatype, or as raw hex bytes.

These are valid data types to use with
the --type option:
  bool,
  int8, int16, int32, int64,
  uint8, uint16, uint32, uint64,
  float, double,
  string, octet_string, unicode_string.
For sign-and-magnitude coding, use the following types:
  sm8, sm16, sm32, sm64

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --type     -t <type>   Data type (see above).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.


ethercat soe_write [OPTIONS] <IDN> <VALUE>
ethercat soe_write [OPTIONS] <DRIVE> <IDN> <VALUE>

Write an SoE IDN to a slave.

This command requires a single slave to be selected.

Arguments:
  DRIVE    is the drive number (0 - 7). If omitted, 0 is assumed.
  IDN      is the IDN and must be either an unsigned
           16 bit number acc. to IEC 61800-7-204:
             Bit 15: (0) Standard data, (1) Product data
             Bit 14 - 12: Parameter set (0 - 7)
             Bit 11 - 0: Data block number
           or a string like 'P-0-150'.
  VALUE    is the value to write (see below).

The VALUE argument is interpreted as the given data type
(--type is mandatory) and written to the selected slave.

These are valid data types to use with
the --type option:
  bool,
  int8, int16, int32, int64,
  uint8, uint16, uint32, uint64,
  float, double,
  string, octet_string, unicode_string.
For sign-and-magnitude coding, use the following types:
```

```
  sm8, sm16, sm32, sm64

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
  --type     -t <type>   Data type (see above).

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.22 Requesting Application-Layer States

```
ethercat states [OPTIONS] <STATE>

Request application-layer states.

Arguments:
  STATE can be 'INIT', 'PREOP', 'BOOT', 'SAFEOP', or 'OP'.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.

Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.1.23 Displaying the Master Version

```
ethercat version [OPTIONS]

Show version information.
```

7.1.24 Generating Slave Description XML

```
ethercat xml [OPTIONS]

Generate slave information XML.

Note that the PDO information can either originate
from the SII or from the CoE communication area. For
slaves, that support configuring PDO assignment and
mapping, the output depends on the last configuration.

Command-specific options:
  --alias    -a <alias>
  --position -p <pos>    Slave selection. See the help of
                         the 'slaves' command.
```

```
Numerical values can be specified either with decimal (no
prefix), octal (prefix '0') or hexadecimal (prefix '0x') base.
```

7.2 Userspace Library

The native application interface (see chapter 3) resides in kernelspace and hence is only accessible from inside the kernel. To make the application interface available from userspace programs, a userspace library has been created, that can be linked to programs under the terms and conditions of the LGPL, version 2 [5].

The library is named *libethercat*. Its sources reside in the *lib/* subdirectory and are build by default when using `make`. It is installed in the *lib/* path below the installation prefix as *libethercat.a* (for static linking), *libethercat.la* (for the use with *libtool*) and *libethercat.so* (for dynamic linking).

For running an application without actual EtherCAT hardware or for simulation purposes, there is a special library called *libfakeethercat* (see subsection 7.2.4).

7.2.1 Using the Library

The application interface header *ecrt.h* (see section 3.6) can be used both in kernel and in user context.

The following minimal example shows how to build a program with EtherCAT functionality. An entire example can be found in the *examples/user/* path of the master sources and in section 3.7.

```
#include <ecrt.h>

int main(void)
{
    ec_master_t *master = ecrt_request_master(0);

    if (!master)
        return 1; // error

    pause(); // wait for signal
    return 0;
}
```

The program can be compiled and dynamically linked to the library with the below command:

Listing 7.1: Linker command for using the userspace library

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

The library can also be linked statically to the program:

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

Please keep in mind, that your application has to be licensed under GPLv2 then, because the LGPL does only allow dynamic linking.

7.2.2 Implementation

Basically the kernel API was transferred into userspace via the master character device (see chapter 2, Figure 2.1 and subsection 7.1.1).

The function calls of the kernel API are mapped to the userspace via an `ioctl()` interface. The userspace API functions share a set of generic `ioctl()` calls. The kernel part of the interface calls the according API functions directly, what results in a minimum additional delay (see subsection 7.2.3).

For performance reasons, the actual domain process data (see section 2.3) are not copied between kernel and user memory on every access: Instead, the data are memory-mapped to the userspace application. Once the master is configured and activated, the master module creates one process data memory area spanning all domains and maps it to userspace, so that the application can directly access the process data. As a result, there is no additional delay when accessing process data from userspace.

Kernel/User API Differences Because of the memory-mapping of the process data, the memory is managed internally by the library functions. As a result, it is not possible to provide external memory for domains, like in the kernel API. The corresponding functions are only available in kernelspace. This is the only difference when using the application interface in userspace.

7.2.3 Timing

An interesting aspect is the timing of the userspace library calls compared to those of the kernel API. Table 7.1 shows the call times and standard deviancies of typical (and time-critical) API functions measured on an Intel Pentium 4 M CPU with 2.2 GHz and a standard 2.6.26 kernel.

The test results show, that for this configuration, the userspace API causes about 1 µs additional delay for each function, compared to the kernel API.

Table 7.1: Application Interface Timing Comparison

Function	Kernelspace		Userspace	
	$\mu(t)$	$\sigma(t)$	$\mu(t)$	$\sigma(t)$
`ecrt_master_receive()`	1.1 µs	0.3 µs	2.2 µs	0.5 µs
`ecrt_domain_process()`	< 0.1 µs	< 0.1 µs	1.0 µs	0.2 µs
`ecrt_domain_queue()`	< 0.1 µs	< 0.1 µs	1.0 µs	0.1 µs
`ecrt_master_send()`	1.8 µs	0.2 µs	2.5 µs	0.5 µs

7.2.4 Simulation / Fake Library

Sometimes is is handy to run your EtherCAT realtime application without an actual EtherCAT network connected, for example for test purposes. Though it is possible to spin up an EtherCAT master and to connect it to a loopback device, this step is not always wanted.

The EtherCAT master (since version 1.6.1) comes with a library *libfakeethercat* that comes with a reasonable subset of the EtherCAT application interface (see chapter 3).

The `ecrt` method implementation in the fake library will just accept your input and behave as if everything would be fine. Without further steps, the process data will be all-zero then.

As a special feature, the *libfakeethercat* will create RtIPC [18] endpoints for registered PDO entries to enable a simulation interface. Another application that either uses RtIPC directly or another (inverted) instance of *libfakeethercat* will then connect to these endpoints and thus create the possibility to provide simulated values to your pristine application.

The fake library functions an usage is documented in Doxygen [13] and the most recent version can be found online: https://docs.etherlab.org/ethercat/1.6/doxygen/libfakeethercat.html

7.3 RTDM Interface

When using the userspace interfaces of realtime extensions like Xenomai or RTAI, the use of *ioctl()* is not recommended, because it may disturb realtime operation. To accomplish this, the Real-Time Device Model (RTDM) [17] has been developed. The master module provides an RTDM interface (see Figure 2.1) in addition to the normal character device, if the master sources were configured with `--enable-rtdm` (see chapter 9).

To force an application to use the RTDM interface instead of the normal character device, it has to be linked with the *libethercat_rtdm* library instead of *libethercat*. The use of the *libethercat_rtdm* is transparent, so the EtherCAT header *ecrt.h* (see section 3.6) with the complete API can be used as usual.

To make the example in Listing 7.1 use the RTDM library, the linker command has to be altered as follows:

```
gcc ethercat-with-rtdm.c -o ectest -I/opt/etherlab/include \
    -L/opt/etherlab/lib -lethercat_rtdm \
    -Wl,--rpath -Wl,/opt/etherlab/lib
```

7.4 System Integration

To integrate the EtherCAT master as a service into a running system, it comes with an init script and a sysconfig file, that are described below. Modern systems may be managed by systemd [7]. Integration of the master with systemd is described in subsection 7.4.4.

7.4.1 Init Script

The EtherCAT master init script conforms to the requirements of the "Linux Standard Base" (LSB, [6]). The script is installed to *etc/init.d/ethercat* below the installation prefix and has to be copied (or better: linked) to the appropriate location (see chapter 9), before the master can be inserted as a service. Please note, that the init script depends on the sysconfig file described below.

To provide service dependencies (i. e. which services have to be started before others) inside the init script code, LSB defines a special comment block. System tools can extract this information to insert the EtherCAT init script at the correct place in the startup sequence:

```
# Default-Stop:       0 1 2 6
# Short-Description:  EtherCAT master
# Description:        EtherCAT master @VERSION@
### END INIT INFO

#------------------------------------------------------------------------------

ETHERCATCTL="@sbindir@/ethercatctl -c @sysconfdir@/sysconfig/ethercat"

#------------------------------------------------------------------------------
```

7.4.2 Sysconfig File

For persistent configuration, the init script uses a sysconfig file installed to *etc/sysconfig/ethercat* (below the installation prefix), that is mandatory for the init script. The sysconfig file contains all configuration variables needed to operate one or more masters. The documentation is inside the file and included below:

```
#
# The MASTER<X>_DEVICE variable specifies the Ethernet device for a master
# with index 'X'.
#
# Specify the MAC address (hexadecimal with colons) of the Ethernet device to
# use. Example: "00:00:08:44:ab:66"
#
# Alternatively, a network interface name can be specified. The interface
# name will be resolved to a MAC address using the 'ip' command.
# Example: "eth0"
#
# The broadcast address "ff:ff:ff:ff:ff:ff" has a special meaning: It tells
# the master to accept the first device offered by any Ethernet driver.
#
# The MASTER<X>_DEVICE variables also determine, how many masters will be
# created: A non-empty variable MASTER0_DEVICE will create one master, adding a
# non-empty variable MASTER1_DEVICE will create a second master, and so on.
#
# Examples:
# MASTER0_DEVICE="00:00:08:44:ab:66"
# MASTER0_DEVICE="eth0"
#
MASTER0_DEVICE=""
#MASTER1_DEVICE=""

#
# Backup Ethernet devices
#
# The MASTER<X>_BACKUP variables specify the devices used for redundancy. They
# behaves nearly the same as the MASTER<X>_DEVICE variable, except that it
# does not interpret the ff:ff:ff:ff:ff:ff address.
#
#MASTER0_BACKUP=""

#
# Ethernet driver modules to use for EtherCAT operation.
#
# Specify a non-empty list of Ethernet drivers, that shall be used for
# EtherCAT operation.
#
# Except for the generic Ethernet driver module, the init script will try to
# unload the usual Ethernet driver modules in the list and replace them with
# the EtherCAT-capable ones. If a certain (EtherCAT-capable) driver is not
# found, a warning will appear.
#
# Possible values: 8139too, e100, e1000, e1000e, r8169, generic, ccat, igb, igc,
    genet, dwmac-intel, stmmac-pci.
# Separate multiple drivers with spaces.
# A list of all matching kernel versions can be found here:
# https://docs.etherlab.org/ethercat/1.6/doxygen/devicedrivers.html
#
# Note: The e100, e1000, e1000e, r8169, ccat, igb and igc drivers are not built by
# default. Enable them with the --enable-<driver> configure switches.
#
DEVICE_MODULES=""

# If you have any issues about network interfaces not being configured
# properly, systemd may need some additional infos about your setup.
# Have a look at the service file, you'll find some details there.
#

#
# List of interfaces to bring up and down automatically.
#
# Specify a space-separated list of interface names (such as eth0 or
# enp0s1) that shall be brought up on `ethercatctl start` and down on
```

```
# 'ethercatctl stop'.
#
# When using the generic driver, the corresponding Ethernet device has to be
# activated before the master is started, otherwise all frames will time out.
# This the perfect use-case for 'UPDOWN_INTERFACES'.
#
UPDOWN_INTERFACES=""

#
# Flags for loading kernel modules.
#
# This can usually be left empty. Adjust this variable, if you have problems
# with module loading.
#
#MODPROBE_FLAGS="-b"

#------------------------------------------------------------------------------
```

For systems managed by systemd (see subsection 7.4.4), the sysconfig file has moved to `/etc/ethercat.conf`. Both versions are part of the master sources and are meant to used alternatively.

7.4.3 Starting the Master as a Service

After the init script and the sysconfig file are placed into the right location, the EtherCAT master can be inserted as a service. The different Linux distributions offer different ways to mark a service for starting and stopping in certain runlevels. For example, SUSE Linux provides the *insserv* command:

```
# insserv ethercat
```

The init script can also be used for manually starting and stopping the EtherCAT master. It has to be executed with one of the parameters `start`, `stop`, `restart` or `status`.

```
# /etc/init.d/ethercat restart
Shutting down EtherCAT master                    done
Starting EtherCAT master                         done
```

7.4.4 Integration with systemd

Distributions using *systemd* instead of the SysV init system are using service files to describe how a service is to be maintained. Listing 7.2 lists the master's service file:

Listing 7.2: Service file

```
#
# EtherCAT master kernel modules
#

[Unit]
Description=EtherCAT Master Kernel Modules
```

```
# Fine tuning of the startup dependencies below are recommended
# to provide a reliable startup routine.
# The dependencies below can be either uncommented after copying
# this file to /etc/systemd/system or by creating overrides:
# Copy the needed dependencies into
# /etc/systemd/system/ethercat.service.d/50-dependencies.conf
# in a [Unit] section.

#
# Uncomment this, if the generic Ethernet driver is used. It assures, that the
# network interfaces are configured, before the master starts.
#
#Requires=network.target # Stop master, if network is stopped
#After=network.target # Start master, after network is ready

#
# Uncomment this, if a native Ethernet driver is used. It assures, that the
# network interfaces are configured, after the Ethernet drivers have been
# replaced. Otherwise, the networking configuration tools could be confused.
#
#Before=network-pre.target
#Wants=network-pre.target

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=@sbindir@/ethercatctl start
ExecStop=@sbindir@/ethercatctl stop

[Install]
WantedBy=multi-user.target
```

The *systemctl* command is used to load and unload the master and network driver modules in a similar way to the former init script (subsection 7.4.1).

```
# systemctl start ethercat
```

When using systemd and/or the *systemctl* command, the master configuration must be in `/etc/ethercat.conf` instead of `/etc/sysconfig/ethercat`! The latter is ignored. The configuration options are exactly the same.

7.5 Debug Interfaces

EtherCAT buses can always be monitored by inserting a switch between master and slaves. This allows to connect another PC with a network monitor like Wireshark [8], for example. It is also possible to listen to local network interfaces on the machine running the EtherCAT master directly. If the generic Ethernet driver (see section 4.3) is used, the network monitor can directly listen on the network interface connected to the EtherCAT bus.

When using native Ethernet drivers (see section 4.2), there are no local network interfaces to listen to, because the Ethernet devices used for EtherCAT are not registered at the network stack. For that case, so-called "debug interfaces" are supported, which are virtual network interfaces allowing to capture EtherCAT traffic with a network monitor (like Wireshark or tcpdump) running on the master machine without using external hardware. To use this functionality, the master sources have to be configured with the `--enable-debug-if` switch (see chapter 9).

Every EtherCAT master registers a read-only network interface per attached physical Ethernet device. The network interfaces are named *ecdbgmX* for the main device, and *ecdbgbX* for the backup device, where X is the master index. The below listing shows a debug interface among some standard network interfaces:

```
# ip link
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop qlen 1000
    link/ether 00:13:46:3b:ad:d7 brd ff:ff:ff:ff:ff:ff
8: ecdbgm0: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast
                                             qlen 1000
    link/ether 00:04:61:03:d1:01 brd ff:ff:ff:ff:ff:ff
```

While a debug interface is enabled, all frames sent or received to or from the physical device are additionally forwarded to the debug interface by the corresponding master. Network interfaces can be enabled with the below command:

```
# ip link set dev ecdbgm0 up
```

Please note, that the frame rate can be very high. With an application connected, the debug interface can produce thousands of frames per second.

Attention The socket buffers needed for the operation of debug interfaces have to be allocated dynamically. Some Linux realtime extensions (like RTAI) do not allow this in realtime context!

8 Timing Aspects

Although EtherCAT's timing is highly deterministic and therefore timing issues are rare, there are a few aspects that can (and should be) dealt with.

8.1 Application Interface Profiling

One of the most important timing aspects are the execution times of the application interface functions, that are called in cyclic context. These functions make up an important part of the overall timing of the application. To measure the timing of the functions, the following code was used:

```
c0 = get_cycles();
ecrt_master_receive(master);
c1 = get_cycles();
ecrt_domain_process(domain1);
c2 = get_cycles();
ecrt_master_run(master);
c3 = get_cycles();
ecrt_master_send(master);
c4 = get_cycles();
```

Between each call of an interface function, the CPU timestamp counter is read. The counter differences are converted to µs with help of the `cpu_khz` variable, that contains the number of increments per ms.

For the actual measuring, a system with a 2.0 GHz CPU was used, that ran the above code in an RTAI thread with a period of 100 µs. The measuring was repeated $n = 100$ times and the results were averaged. These can be seen in Table 8.1.

Table 8.1: Profiling of an Application Cycle on a 2.0 GHz Processor

Element	Mean Duration [s]	Standard Deviancy [µs]
ecrt_master_receive()	8.04	0.48
ecrt_domain_process()	0.14	0.03
ecrt_master_run()	0.29	0.12
ecrt_master_send()	2.18	0.17
Complete Cycle	10.65	0.69

It is obvious, that the functions accessing hardware make up the lion's share. The *ec_master_receive()* executes the ISR of the Ethernet device, analyzes datagrams and copies their contents into the memory of the datagram objects. The *ec_master_send()* assembles a frame out of different datagrams and copies it to the hardware buffers. Interestingly, this makes up only a quarter of the receiving time.

The functions that only operate on the masters internal data structures are very fast ($\Delta t < 1$ µs). Interestingly the runtime of *ec_domain_process()* has a small standard deviancy relative to the mean value, while this ratio is about twice as big for *ec_master_run()*: This probably results from the latter function having to execute code depending on the current state and the different state functions are more or less complex.

For a realtime cycle makes up about 10 µs, the theoretical frequency can be up to 100 kHz. For two reasons, this frequency keeps being theoretical:

1. The processor must still be able to run the operating system between the realtime cycles.
2. The EtherCAT frame must be sent and received, before the next realtime cycle begins. The determination of the bus cycle time is difficult and covered in section 8.2.

8.2 Bus Cycle Measuring

For measuring the time, a frame is "on the wire", two timestamps must be taken:

1. The time, the Ethernet hardware begins with physically sending the frame.
2. The time, the frame is completely received by the Ethernet hardware.

Both times are difficult to determine. The first reason is, that the interrupts are disabled and the master is not notified, when a frame is sent or received (polling would distort the results). The second reason is, that even with interrupts enabled, the time from the event to the notification is unknown. Therefore the only way to confidently determine the bus cycle time is an electrical measuring.

Anyway, the bus cycle time is an important factor when designing realtime code, because it limits the maximum frequency for the cyclic task of the application. In practice, these timing parameters are highly dependent on the hardware and often a trial and error method must be used to determine the limits of the system.

The central question is: What happens, if the cycle frequency is too high? The answer is, that the EtherCAT frames that have been sent at the end of the cycle are not yet received, when the next cycle starts. First this is noticed by *ecrt_domain_process()*, because the working counter of the process data datagrams were not increased. The function will notify the user via Syslog[1]. In this case, the process data keeps being the

[1] To limit Syslog output, a mechanism has been implemented, that outputs a summarized notification at maximum once a second.

same as in the last cycle, because it is not erased by the domain. When the domain datagrams are queued again, the master notices, that they are already queued (and marked as sent). The master will mark them as unsent again and output a warning, that datagrams were “skipped”.

On the mentioned 2.0 GHz system, the possible cycle frequency can be up to 25 kHz without skipped frames. This value can surely be increased by choosing faster hardware. Especially the RealTek network hardware could be replaced by a faster one. Besides, implementing a dedicated ISR for EtherCAT devices would also contribute to increasing the latency. These are two points on the author’s to-do list.

9 Installation

9.1 Getting the Software

There are several ways to get the master software:

1. An official release (for example 1.6.3), can be downloaded from the master's website[1] at the EtherLab project [1] as a tarball.
2. The most recent development revision (and moreover any other revision) can be obtained via the Git [14] repository on the master's project page on GitLab.com[2]. The whole repository can be cloned with the command

   ```
   git clone https://gitlab.com/etherlab.org/ethercat.git
      local-dir
   ```

3. Without a local Git installation, tarballs of arbitrary revisions can be downloaded via the "Download" button on GitLab.

9.2 Building the Software

After downloading a tarball or cloning the repository as described in section 9.1, the sources have to be prepared and configured for the build process.

When a tarball was downloaded, it has to be extracted with the following commands:

```
$ tar xjf ethercat-1.6.3.tar.bz2
$ cd ethercat-1.6.3/
```

The software configuration is managed with Autoconf [15] so the released versions contain a `configure` shell script, that has to be executed for configuration (see below).

Bootstrap When downloading or cloning directly from the repository, the `configure` script does not yet exist. It can be created via the `bootstrap.sh` script in the master sources. The autoconf and automake packages are required for this.

[1] https://etherlab.org/ethercat
[2] https://gitlab.com/etherlab.org/ethercat

Configuration and Build The configuration and the build process follow the below commands:

```
$ ./configure
$ make
$ make modules
```

Table 9.1 lists important configuration switches and options.

Table 9.1: Configuration options

Option/Switch	Description	Default
`--prefix`	Installation prefix	*/opt/etherlab*
`--with-linux-dir`	Linux kernel sources	Use running kernel
`--with-module-dir`	Subdirectory in the kernel module tree, where the EtherCAT kernel modules shall be installed.	*ethercat*
`--enable-generic`	Build the generic Ethernet driver (see section 4.3).	yes
`--enable-8139too`	Build the 8139too driver	yes
`--with-8139too-kernel`	8139too kernel	†
`--enable-e100`	Build the e100 driver	no
`--with-e100-kernel`	e100 kernel	†
`--enable-e1000`	Enable e1000 driver	no
`--with-e1000-kernel`	e1000 kernel	†
`--enable-e1000e`	Enable e1000e driver	no
`--with-e1000e-kernel`	e1000e kernel	†
`--enable-r8169`	Enable r8169 driver	no
`--with-r8169-kernel`	r8169 kernel	†
`--enable-ccat`	Enable ccat driver (independent of kernel version)	no
`--enable-igb`	Enable igb driver	no
`--with-igb-kernel`	igb kernel	†
`--enable-kernel`	Build the master kernel modules	yes
`--enable-rtdm`	Create the RTDM interface (RTAI or Xenomai directory needed, see below)	no
`--with-rtai-dir`	RTAI path (for RTAI examples and RTDM interface)	
`--with-xenomai-dir`	Xenomai path (for Xenomai examples and RTDM interface)	
`--with-devices`	Number of Ethernet devices for redundant operation (> 1 switches redundancy on)	1

Option/Switch	Description	Default
`--with-systemdsystemunitdir`	Systemd unit directory ("no" disables service file installation)	auto
`--enable-debug-if`	Create a debug interface for each master	no
`--enable-debug-ring`	Create a debug ring to record frames	no
`--enable-eoe`	Enable EoE support	yes
`--enable-cycles`	Use CPU timestamp counter. Enable this on Intel architecture to get finer timing calculation.	no
`--enable-hrtimer`	Use high-resolution timer to let the master state machine sleep between sending frames.	no
`--enable-regalias`	Read alias address from register	no
`--enable-tool`	Build the command-line tool "ethercat" (see section 7.1)	yes
`--enable-userlib`	Build the userspace library	yes
`--enable-tty`	Build the TTY driver	no
`--enable-wildcards`	Enable *0xffffffff* to be wildcards for vendor ID and product code	no
`--enable-sii-assign`	Enable assigning SII access to the PDI layer during slave configuration	no
`--enable-rt-syslog`	Enable syslog statements in realtime context	yes

† If this option is not specified, the kernel version to use is extracted from the Linux kernel sources.

9.3 Building the Interface Documentation

The source code is documented using Doxygen [13]. To build the HTML documentation, the Doxygen software has to be installed. The below command will generate the documents in the subdirectory *doxygen-output*:

```
$ make doc
```

The interface documentation can be viewed by pointing a browser to the file *doxygen-output/html/index.html*. The functions and data structures of the application interface are covered by an own module "Application Interface".

9.4 Installing the Software

The below commands have to be entered as *root*: the first one will install the EtherCAT header, service scripts (systemd or init.d) and the userspace tool to the prefix path. The second one will install the kernel modules to the kernel's modules directory. The final `depmod` call is necessary to include the kernel modules into the *modules.dep* file to make it available to the `modprobe` command, used by the service scripts.

```
# make install
# make modules_install
# depmod
```

If the target kernel's modules directory is not under */lib/modules*, a different destination directory can be specified with the `DESTDIR` make variable. For example:

```
# make DESTDIR=/vol/nfs/root modules_install
```

This command will install the compiled kernel modules to */vol/nfs/root/lib/modules*, prepended by the kernel release.

Now the sysconfig file `/etc/sysconfig/ethercat` (see subsection 7.4.2), or the configuration file */etc/ethercat.conf*, if using systemd, has to be customized. The minimal customization is to set the `MASTER0_DEVICE` variable to the MAC address of the Ethernet device to use (or `ff:ff:ff:ff:ff:ff` to use the first device offered) and selecting the driver(s) to load via the `DEVICE_MODULES` variable.

After the basic configuration is done, the master can be started with the below command:

```
# systemctl start ethercat
```

When using init.d, the following command can be used alternatively:

```
# /etc/init.d/ethercat start
```

At this time, the operation of the master can be observed by viewing the Syslog messages, which should look like the ones below. If EtherCAT slaves are connected to the master's EtherCAT device, the activity indicators should begin to flash.

```
EtherCAT: Master driver 1.6.3
EtherCAT: 1 master waiting for devices.
EtherCAT Intel(R) PRO/1000 Network Driver - version 6.0.60-k2
Copyright (c) 1999-2005 Intel Corporation.
PCI: Found IRQ 12 for device 0000:01:01.0
PCI: Sharing IRQ 12 with 0000:00:1d.2
PCI: Sharing IRQ 12 with 0000:00:1f.1
EtherCAT: Accepting device 00:0E:0C:DA:A2:20 for master 0.
EtherCAT: Starting master thread.
ec_e1000: ec0: e1000_probe: Intel(R) PRO/1000 Network
```

```
11              Connection
12 ec_e1000: ec0: e1000_watchdog_task: NIC Link is Up 100 Mbps
13              Full Duplex
14 EtherCAT: Link state changed to UP.
15 EtherCAT: 7 slave(s) responding.
16 EtherCAT: Slave states: PREOP.
17 EtherCAT: Scanning bus.
18 EtherCAT: Bus scanning completed in 431 ms.
```

1 – 2 The master module is loading, and one master is initialized.

3 – 8 The EtherCAT-capable e1000 driver is loading. The master accepts the device with the address `00:0E:0C:DA:A2:20`.

9 – 16 The master goes to idle phase, starts its state machine and begins scanning the bus.

9.5 Automatic Device Node Creation

The `ethercat` command-line tool (see section 7.1) communicates with the master via a character device. The corresponding device nodes are created automatically, if the udev daemon is running. Note, that on some distributions, the `udev` package is not installed by default.

The device nodes will be created with mode `0660` and group `root` by default. If "normal" users shall have reading access, a udev rule file (for example */etc/udev/rules.d/99-EtherCAT.rules*) has to be created with the following contents:

```
KERNEL=="EtherCAT[0-9]*", MODE="0664"
```

After the udev rule file is created and the EtherCAT master is restarted with `/etc/init.d/ethercat restart`, the device node will be automatically created with the desired rights:

```
# ls -l /dev/EtherCAT0
crw-rw-r--  1 root root 252, 0 2008-09-03 16:19 /dev/EtherCAT0
```

Now, the `ethercat` tool can be used (see section 7.1) even as a non-root user.

If non-root users shall have writing access, the following udev rule can be used instead:

```
KERNEL=="EtherCAT[0-9]*", MODE="0664", GROUP="users"
```

Bibliography

[1] Ingenieurgemeinschaft IgH: EtherLab – Open Source Toolkit for rapid realtime code generation under Linux with Simulink/RTW and EtherCAT technology. https://etherlab.org, 2024.

[2] IEC 61158-4-12: Data-link Protocol Specification. International Electrotechnical Commission (IEC), 2005.

[3] IEC 61158-6-12: Application Layer Protocol Specification. International Electrotechnical Commission (IEC), 2005.

[4] GNU General Public License, Version 2. http://www.gnu.org/licenses/gpl-2.0.html. October 15, 2008.

[5] GNU Lesser General Public License, Version 2.1. http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html. October 15, 2008.

[6] Linux Standard Base. http://www.linuxfoundation.org/en/LSB. August 9, 2006.

[7] systemd System and Service Manager http://freedesktop.org/wiki/Software/systemd. January 18, 2013.

[8] Wireshark. http://www.wireshark.org. 2008.

[9] *Hopcroft, J. E. / Ullman, J. D.*: Introduction to Automata Theory, Languages and Computation. Adison-Wesley, Reading, Mass. 1979.

[10] *Wagner, F. / Wolstenholme, P.*: State machine misunderstandings. In: IEE journal "Computing and Control Engineering", 2004.

[11] RTAI. The RealTime Application Interface for Linux from DIAPM. https://www.rtai.org, 2010.

[12] RT PREEMPT HOWTO. http://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO, 2010.

[13] Doxygen. Source code documentation generator tool. http://www.stack.nl/~dimitri/doxygen, 2008.

[14] Git SCM. https://git-scm.com, 2021.

[15] Autoconf – GNU Project – Free Software Foundation (FSF). http://www.gnu.org/software/autoconf, 2010.

[16] IEC 61800-7-304: Adjustable speed electrical power drive systems - Part 7-300: Generic interface and use of profiles for power drive systems - Mapping of profiles to network technologies. International Electrotechnical Commission (IEC), 2007.

[17] *J. Kiszka*: The Real-Time Driver Model and First Applications. http://svn.gna.org/svn/xenomai/tags/v2.4.0/doc/nodist/pdf/RTDM-and-Applications.pdf, 2013.

[18] Real-Time Inter-Process-Communication library. Part of the EtherLab toolkit. https://gitlab.com/etherlab.org/rtipc, 2024.

Glossary

ADEOS Adaptive Domain Environment for Operating Systems, page 1

CoE CANopen over EtherCAT, Mailbox Protocol, page 107

ecdev EtherCAT Device, page 83

EoE Ethernet over EtherCAT, Mailbox Protocol, page 103

FSM Finite State Machine, page 85

ISR Interrupt Service Routine, page 78

LSB Linux Standard Base, page 3

PCI Peripheral Component Interconnect, Computer Bus, page 80

RTAI Realtime Application Interface, page 1

Index