IgH EtherCAT® Master 1.1 Documentation

Florian Pose, fp@igh-essen.com
Ingenieurgemeinschaft IgH

Essen, September 1, 2006
Revision 531

Contents

Conventions x

1 **The IgH EtherCAT Master** **1**
1.1 Feature Summary 1
1.2 License 2
1.3 General Master Architecture 3
1.3.1 Handling of Process Data 5
1.3.2 Operation Modes 6
1.4 Device Modules 8
1.4.1 Network Driver Basics 8
1.4.2 EtherCAT Network Drivers 10
1.4.3 Device Selection 12
1.4.4 The Device Interface 12
1.4.5 Patching Network Drivers 15
1.5 The Master Module 21
1.5.1 Class Reference 22
1.5.2 The Realtime Interface 42
1.5.3 Slave Addressing 48
1.5.4 Concurrent Master Access 49
1.6 The Master's State Machines 50
1.6.1 State Machine Theory 51
1.6.2 The Master's State Model 53
1.6.3 The Operation State Machine 56
1.6.4 The Idle State Machine 58
1.6.5 The Slave Scan State Machine 59
1.6.6 The Slave Configuration State Machine 62
1.6.7 The State Change State Machine 64
1.6.8 The SII State Machine 65
1.7 Mailbox Protocol Implementations 66
1.7.1 Ethernet-over-EtherCAT (EoE) 66
1.7.2 CANopen-over-EtherCAT (CoE) 70
1.8 User Space 72
1.8.1 The Sysfs Interface 72
1.8.2 User Space Tools 76
1.8.3 System Integration 77
1.8.4 Monitoring and Debugging 79

1.9 Timing Aspects . . . 79
1.9.1 Realtime Interface Profiling . . . 79
1.9.2 Bus Cycle Measuring . . . 80

2 Using the EtherCAT Master **83**
2.1 Compiling and Installing . . . 83
2.2 A Minimal Example Module . . . 84
2.3 An RTAI Example Module . . . 88
2.4 Concurrency Example . . . 91

Bibliography **97**

Glossary **99**

Index **101**

List of Tables

1.1 A typical state transition table 52
1.2 Profiling of a Realtime Cycle on a 2.0 GHz Processor 80

List of Figures

1.1 Master architecture . . . 4
1.2 FMMU configuration for serveral domains . . . 6
1.3 Master modes and transitions . . . 7
1.4 Interrupt Operation versus Interrupt-less Operation . . . 11
1.5 Multiple masters in one module . . . 21
1.6 UML class diagram with associations . . . 23
1.7 Master UML class diagram . . . 24
1.8 Slave UML class diagram . . . 27
1.9 Device UML class diagram . . . 30
1.10 Datagram UML class diagram . . . 32
1.11 Domain UML class diagram . . . 34
1.12 Finite State Machine UML class diagram . . . 36
1.13 EoE UML class diagram . . . 39
1.14 Debug UML class diagram . . . 42
1.15 Concurrent master access . . . 49
1.16 A typical state transition diagram . . . 52
1.17 Transition diagram of the operation state machine . . . 56
1.18 Transition diagram of the idle state machine . . . 58
1.19 Transition diagram of the slave scan state machine . . . 60
1.20 Transition diagram of the slave configuration state machine . . . 62
1.21 Transition diagram of the state change state machine . . . 64
1.22 Transition diagram of the SII state machine . . . 65
1.23 Transition diagram of the EoE state machine . . . 69
1.24 Transition diagram of the CoE download state machine . . . 71

List of Listings

2.1 Minimal variables . . . 85
2.2 Minimal init function . . . 86
2.3 Minimal cleanup function . . . 87
2.4 Minimal cyclic function . . . 87
2.5 RTAI task declaration . . . 89
2.6 RTAI module init function . . . 89
2.7 RTAI module cleanup function . . . 90
2.8 RTAI module cyclic function . . . 91
2.9 RTAI semaphore for concurrent access . . . 92
2.10 RTAI locking callbacks for concurrent access . . . 92
2.11 Module init function for concurrent access . . . 92
2.12 RTAI cyclic function for concurrent access . . . 93
2.13 RTAI module cleanup function for concurrent access . . . 94
2.14 Variables for jitter reduction . . . 94
2.15 Cyclic function with reduced jitter . . . 95
2.16 Request callback for reduced jitter . . . 95

Conventions

The following typographic conventions are used:

- *Italic face* is used for newly introduced terms, file names, parameter names and in-text source code elements.
- `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 specified as hexadecimal values with the prefix 0x. Example: 0x88A4. Unless otherwise noted, address values are specified as byte addresses.

Concerning bit operations, the phrase "setting a bit", stands for setting the bit to 1, "clearing a bit" means setting it to 0, respectively.

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

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

```
host>
```

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

```
host#
```

1 The IgH EtherCAT Master

This section will first introduce the master's general features and the concepts used for master development and will then explain the master's general architecture and offer details of the different modules. In addition, it will cover state machine definitions, mailbox protocol implementation and the user space interface. The last section will deal with timing aspects.

1.1 Feature Summary

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

- The master runs as a kernel module for Linux 2.6.
- It comes with EtherCAT-capable network driver for RealTek RTL8139 (and compatible) network interface cards.
  - The Ethernet hardware is operated without interrupts.
  - Drivers for additional Ethernet hardware can easily be implemented due to a common device interface provided by the master.
- The master module supports multiple EtherCAT masters on one machine.
- The master code supports any Linux realtime extension through its independent architecture.
  - RTAI, ADEOS, etc.
  - It runs well even without realtime extensions.
- Common "realtime interface" for modules, that want to use EtherCAT functionality.
  - Synchronous and asynchronous sending and receiving of frames is supported.
  - Avoidance of unnecessary copy operations for process data.
- *Domains* are introduced, to allow grouping of process data objects.

  - Handling of multiple domains with different sampling rates.
  - Automatic calculation of process data mapping, FMMU and sync manager configuration within each domain.

- Communication through serveral finite state machines.
  - Bus monitoring possible during realtime operation.
  - Automatic reconfiguration of slaves on bus power failure during realtime operation.
  - Controlling of single slaves during realtime operation.

- Master idle mode.
  - Automatic scanning of slaves upon topology changes.
  - Bus visualisation and EoE processing without a realtime module connected.

- Implementation of the CANopen-over-EtherCAT (CoE) protocol.
  - Configuration of CoE-capable slaves via SDO interface.

- Implementation of the Ethernet-over-EtherCAT (EoE) protocol.
  - Each master creates virtual network devices that are automatically coupled to EoE-capable slaves found.
  - This implementation natively supports either a switched or a routed EoE network architecture.

- User space interface via the System Filesystem (Sysfs).
  - User space tool for bus visualisation.
  - Slave E^2PROM image reading and writing.

- Seamless system integration though LSB compliance.
  - Master and network device configuration via Sysconfig files.
  - Linux Standard Base compatible init script for master control.

- 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 [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.

1.3 General Master Architecture

The EtherCAT master is integrated into the Linux 2.6 kernel. This was an early design decision, which has been made for serveral reasons:

- Kernel code has significantly better realtime characteristics, i. e. less jitter than user space 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 kernel space, because there is no need of time-consuming context switches to a user space 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 kernel space.

A general overview of the master architecture can be seen in figure 1.1.

Master Module The EtherCAT master mainly consists of the master module, containing one or more EtherCAT masters (section 1.5), the “Device Interface” (section 1.4.4) and the “Realtime Interface” (section 1.5.2).

Device Modules Furthermore there are EtherCAT-capable network device driver modules, that connect to the EtherCAT master via the device interface. These modified network drivers can handle both network devices used for EtherCAT operation and “normal” Ethernet devices. The common case is, that the master module offers a single EtherCAT master: An EtherCAT-capable network device driver module connects one network device to this master, that is now able to send and receive EtherCAT frames, while all other network devices handled by the network driver get connected to the kernel's network stack as usual.

Realtime Modules A “realtime module” is a kernel module, that uses the EtherCAT master for cyclic exchange of process data with EtherCAT slaves. Realtime modules are not part of the EtherCAT master code[1], so anybody wanting to use the master has to write one. A realtime module can “request” a master through the realtime interface. If this succeeds, the module has the control over the master. It can now configure slaves and set up a process data image (see section 1.3.1) for cyclic exchange. This cyclic code has to be provided by the realtime module, so it is in hands of the developer, which mechanism to use for this. Moreover he has to decide, whether or not using a Linux realtime extension.

[1] Although there are serveral examples provided in the *examples* directory, see chapter 2 for more information

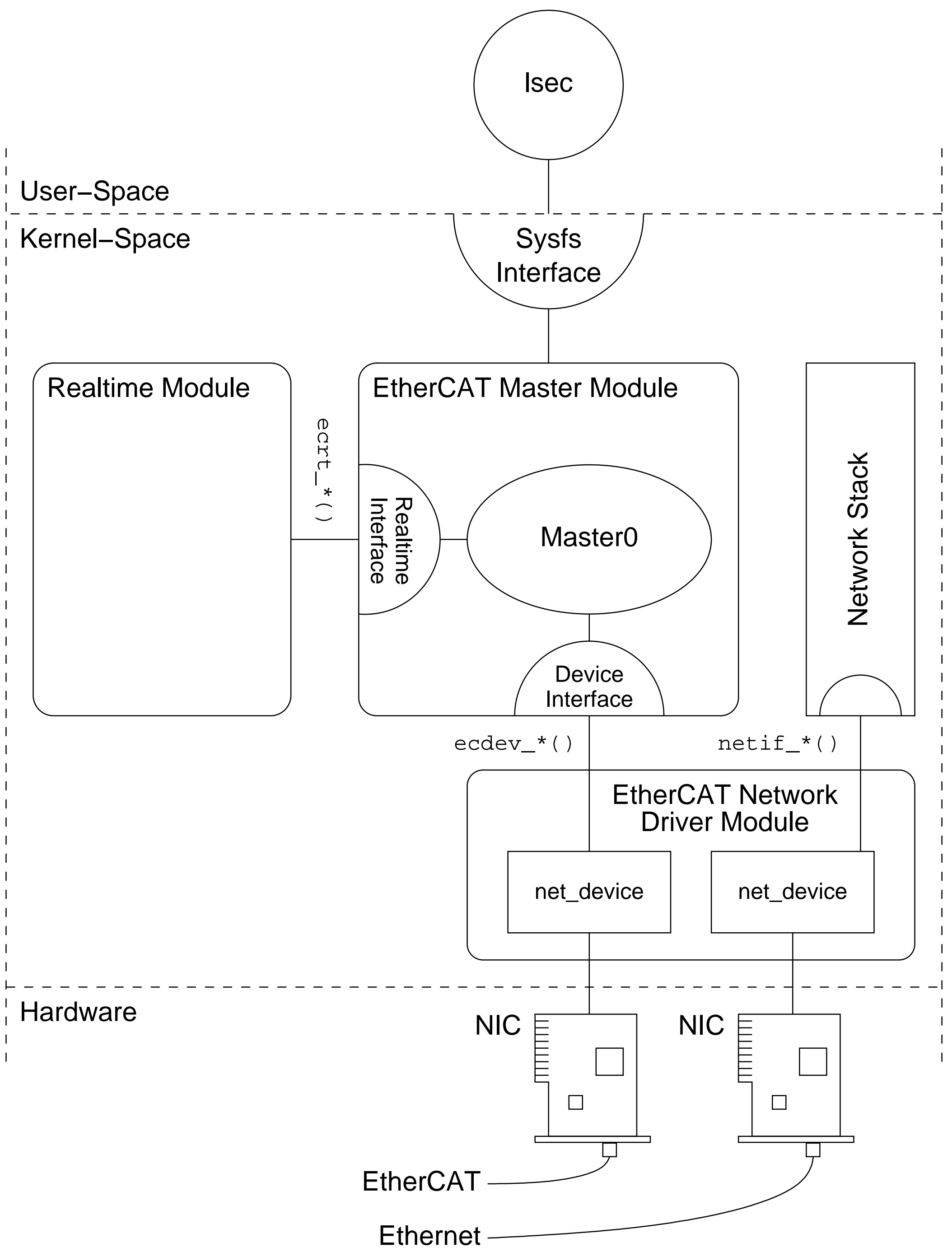

Figure 1.1: Master architecture

Why "Realtime" Module? The name shall not imply, that a linux realtime extension is mandatory: The master runs well even without realtime extensions, as section 2.2 shows. However, the code using the master is time-critical, because process data IO has to be done in cyclic work. Some EtherCAT slaves support watchdog units, that stop driving the outputs when process data was not exchanged for some time. So the names "realtime interface" and "realtime module" are quite appropriate.

1.3.1 Handling of Process Data

Process Data Image The slaves offer their inputs and outputs by presenting the master so-called "Process Data Objects" (PDOs). The available PDOs can be determined by reading out the slave's TXPDO and RXPDO E^2PROM categories. The realtime module can register the PDOs for data exchange during cyclic operation. The sum of all registered PDOs defines the "process data image", which is exchanged via the "Logical ReadWrite" datagrams introduced in [2, section 5.4.2.4].

Process Data Domains The process data image can be easily managed by creating co-called "domains", which group PDOs and allocate the datagrams needed to exchange them. 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 "Logical ReadWrite" 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.
- 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 also limited by the slaves' capabilities.

FMMU Configuration A realtime module can register PDOs for process data exchange. Every PDO is part of a memory area in the slave's physical memory, that is protected by a sync manager [2, section 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 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 resides in. If a second PDO of the same slave is registered for process data exchange within the same domain, and this PDO resides in the same sync-manager-protected memory as the first PDO, the FMMU configuration is not touched, because the appropriate memory is already part of the domain's process data image. If the second PDO belongs to another sync-manager-protected area, this complete area is also included into the domains process data image. See figure 1.2 for an overview, how FMMU's are configured to map physical memory to logical process data images.

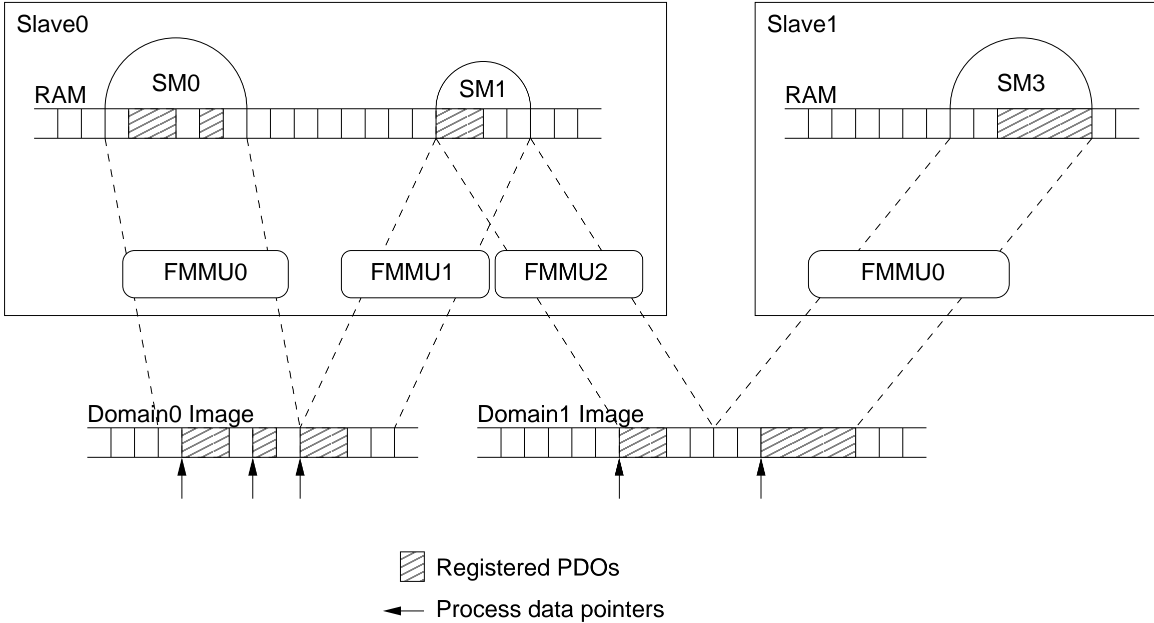

Figure 1.2: FMMU configuration for serveral domains

Process Data Pointers The figure also demonstrates the way, the realtime module can access the exchanged process data: At PDO registration, the realtime module has to provide the address of a process data pointer. Upon calculation of the domain image and allocation of process data memory, this pointer is redirected to the appropriate location inside the domain's process data memory and can later be easily dereferenced by the module code.

1.3.2 Operation Modes

The EtherCAT master has serveral modes of operation:

Orphaned Mode This mode takes effect, when the master has no EtherCAT-capable network device connected. No bus communication is possible, so this mode is not of further interest.

Idle Mode takes effect when the master is unused (i. e. there is no realtime module, that reserved the master). In this case, the master has the ability to scan the bus by itsself and generously allow bus access from user space. This mode is meant for maintenance and visualisation.

Operation Mode The master is reserved for exclusive access by a realtime module. In this mode, the master is adjusted for availability and monitoring. Access from user space is very restrictive and mostly limited to reading direction.

Figure 1.3 shows the three modes and the possible mode transitions.

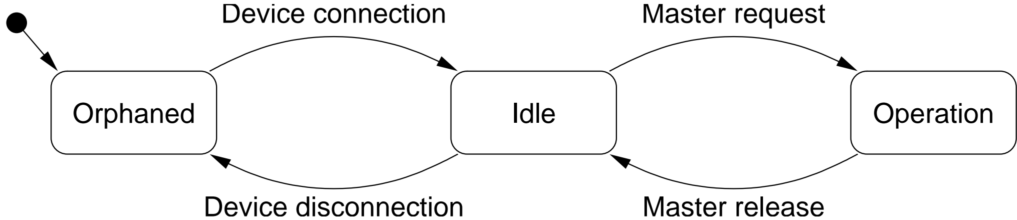

Figure 1.3: Master modes and transitions

1.3.2.1 Idle Mode

The master enters idle mode upon connection of a device module (see section 1.4) or releasing by a realtime module. The master owns a kernel workqueue and a suitable work structure, which is used to cyclically process the “Idle state machine” (see section 1.6.4). This state machine automatically scans the bus for slaves (and re-scans upon topology changes), configures slaves for idle operation and executes pending operations from the user space interface (for example E^2PROM writing). On device disconnection or realtime request, the idle mode is stopped by cancelling the work and flushing the workqueue.

1.3.2.2 Operation Mode

Operation mode is entered when a realtime module requests the master. The idle mode is stopped and the bus is scanned by getting the number of slaves and executing the “Slave scan state machine” (see section 1.6.5) for each slave. The master is now ready to create domains and accept PDO registrations and slave configurations. After that, cyclic communication can be done by the realtime module.

Master Phases Every realtime module should use the master in three phases:

1. *Startup* - The master is requested and the bus is validated. Domains are created and PDOs are registered. Slave configurations are applied.

2. *Operation* - Cyclic code is run, process data is exchanged and the master state machine is executed.

3. *Shutdown* - Cyclic code is stopped and the master is released.

1.4 Device Modules

Device modules are network device driver modules that handle Ethernet devices, which the master can use to connect to an EtherCAT bus.

Section 1.4.1 offers an overview of general Linux network driver modules, while section 1.4.2 will show the requirements to an EtherCAT-enabled network driver. Finally, sections 1.4.3 to 1.4.5 show how to fulfill these requirements and implement such a driver module.

1.4.1 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 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 transmittion, 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 user space or from the network stack) via serveral 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:

int (*open)(struct net_device *) This function is called when network communication has to be started, for example after a command *ifconfig ethX up* from user space. Frame reception has to be enabled by the driver.

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

int (*hard_start_xmit)(struct sk_buff *, struct net_device *) 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.

struct net_device_stats *(*get_stats)(struct net_device *) 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 everytime 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()*[2]: It passes a frame to the network stack, that was just received by the device. Frame data has to be packed into 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 serveral 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.

1.4.2 EtherCAT Network Drivers

There are a few requirements for Ethernet network devices to function as EtherCAT devices, when connected to an EtherCAT bus.

Dedicated Interfaces 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. Therefore, there is no need to notify the driver about frame reception: The master can instead query the hardware for received frames.

Figure 1.4 shows two workflows for cyclic frame transmittion 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,

[2]This function is part of the NAPI ("New API"), that replaces the "old" 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

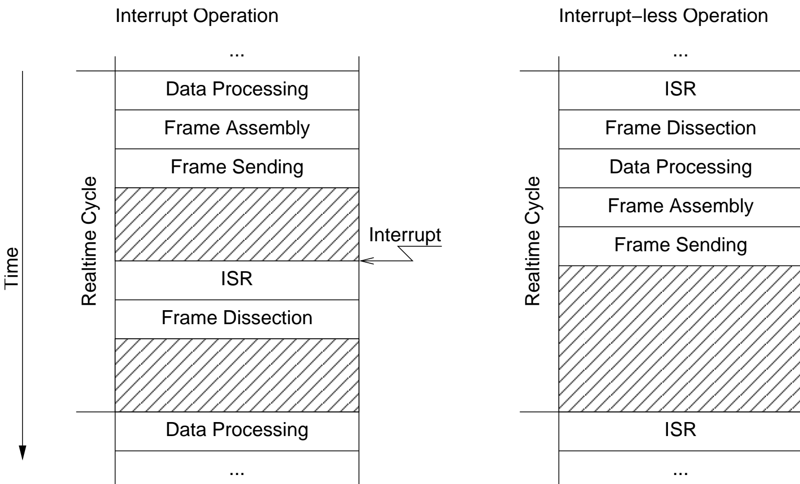

Figure 1.4: Interrupt Operation versus Interrupt-less Operation

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 there is simply no need for an interrupt. Moreover hardware interrupts are not conducive in improving the driver's realtime behaviour: Their undeterministic incidences contribute to increasing the jitter. Besides, if a realtime extension (like RTAI) is used, some additional effort would have to be made to priorize 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 has 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 bugfixes on the standard drivers have to be ported to the Ether-CAT-capable versions from time to time.

1.4.3 Device Selection

After loading the master module, at least one EtherCAT-capable network driver module has to be loaded, that connects one of its devices to the master. To specify an EtherCAT device and the master to connect to, all EtherCAT-capable network driver modules should provide two module parameters:

ec_device_index PCI device index of the device that is connected to the EtherCAT bus. If this parameter is left away, all devices found are treated as ordinary Ethernet devices. Default: −1

ec_master_index Index of the master to connect to. Default: 0

The following command loads the EtherCAT-capable RTL8139 device driver, telling it to handle the second device as an EtherCAT device and connecting it to the first master:

```
host# modprobe ec_8139too ec_device_index=1
```

Usually, this command does not have to be entered manually, but is called by the EtherCAT init script. See section 1.8.3.1 for more information.

1.4.4 The Device Interface

An anticipation to the section about the master module (section 1.5) 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 which is explained below. All functions of the device interface are named with the prefix *ecdev*.

Device Registration A network device driver can connect a physical device to an EtherCAT master with the *ecdev_register()* function.

```
ec_device_t *ecdev_register(unsigned int master_index,
                            struct net_device *net_dev,
                            ec_isr_t isr,
                            struct module *module);
```

The first parameter *master_index* must be the index of the EtherCAT master to connect to (see section 1.5), followed by *net_dev*, the pointer to the corresponding net_device structure, which represents the network device to connect. The third parameter *isr* must be a pointer to the interrupt service routine (ISR) handling the device. The master will later execute the ISR in order to receive frames and to update the device status. The last parameter *module* must be the pointer to the device driver module, which is usually accessible via the macro *THIS_MODULE* (see next paragraph). On success, the function returns a pointer to an *ec_device_t* object, which has to be specified when calling further functions of the device interface. Therefore the device module has to store this pointer for future use. In error case, the *ecdev_register()* returns *NULL*, which means that the device could not be registered. The reason for this is printed to *syslog*. In this case, the device module is supposed to abort the module initialisation and let the *insmod* command fail.

Implicit Dependencies The reason for the module pointer has to be specified at device registration is a non-trivial one: The master has to know about the module, because there will be an implicit dependency between the device module and a later connected realtime module: When a realtime module connects to the master, the use count of the master module will be increased, so that the master module can not be unloaded for the time of the connection. This is reasonable, and so automatically done by the kernel. The kernel knows about this dependency, because the realtime module uses kernel symbols provided by the master module. Moreover it is mandatory, that the device module can be unloaded neither, because it is implicitely used by the realtime module, too. Unloading it would lead to a fatal situation, because the master would have no device to send and receive frames for the realtime module. This dependency can not be detected automatically, because the realtime module does not use any symbols of the device module. Therefore the master explicitly increments the use counter of the connected device module upon connection of a realtime module and decrements it, if the realtime module disconnects. In this manner, it is impossible to unload a device module while the master is in use. This is done with the kernel function pair *try_module_get()* and *module_put()* . The first one increases the use count of a module and only fails, if the module is currenly being unloaded. The last one decreases the use count again and never fails. Both functions take a pointer to the module as their argument, which the device module therefore has to specify upon device registration.

Device Unregistering The unregistration of a device is usually done in the device module's cleanup function, by calling the *ecdev_unregister()* function and specifying the master index and a pointer to the device object again.

```
void ecdev_unregister(unsigned int master_index,
                      ec_device_t *device);
```

This function can fail too (if the master index is invalid, or the given device was not registered), but due to the fact, that this failure can not be dealt with appropriately, because the device module is unloading anyway, the failure code would not be of any interest. So the function has a void return value.

Starting the Master When a device has been initialized completely and is ready to send and receive frames, the master has to be notified about this by calling the *ecdev_start()* function.

```
int ecdev_start(unsigned int master_index);
```

The master will then enter "Idle Mode" and start scanning the bus (and possibly handling EoE slaves). Moreover it will make the bus accessible via Sysfs interface and react to user interactions. The function takes one parameter *master_index*, which has to be the same as at the call to *ecdev_register()*. The return value will be non-zero if the starting process failed. In this case the device module is supposed to abort the init sequence and make the init function return an error code.

Stopping the Master Before a device can be unregistered, the master has to be stopped by calling the *ecdev_stop()* function. It will stop processing messages of EoE slaves and leave "Idle Mode". The only parameter is *master_index*. This function can not fail.

```
void ecdev_stop(unsigned int master_index);
```

A subsequent call to *ecdev_unregister()* will now unregister the device savely.

Receiving Frames The interrupt service routine handling device events usually has a section where new frames are fetched from the hardware and forwarded to the kernel network stack via *netif_receive_skb()*. For an EtherCAT-capable device, this has to be replaced by calling the *ecdev_receive()* function to forward the received data to the connected EtherCAT master instead.

```
void ecdev_receive(ec_device_t *device,
                   const void *data,
                   size_t size);
```

This function takes 3 arguments, a pointer to the device object (*device*), a pointer to the received data, and the size of the received data. The data range has to include the Ethernet headers starting with the destination address and reach up to the last octet of EtherCAT data, excluding the FCS. Most network devices handle the FCS in hardware, so it is not seen by the driver code and therefore doesn't have to be cut off manually.

Handling the Link Status Information about the link status (i. e. if there is a carrier signal detected on the physical port) is also important to the master. This information is usually gathered by the ISR and should be forwarded to the master by calling the *ecdev_link_state()* function. The master then can react on this and warn the realtime module of a lost link.

```
void ecdev_link_state(ec_device_t *device,
                      uint8_t new_state);
```

The parameter *device* has to be a pointer to the device object returned by *ecdev_-register()*. With the second parameter *new_state*, the new link state is passed: 1, if the link went up, and 0, if it went down.

1.4.5 Patching Network Drivers

This section will demonstrate, how to make a standard Ethernet driver EtherCAT-capable. The below code examples are taken out of the modified RealTek RTL8139 driver coming with the EtherCAT master (*devices/8139too.c*). The driver was originally developed by Donald Becker, and is currently maintained by Jeff Garzik.

Unfortunately, there is no standard procedure to enable an Ethernet driver for use with the EtherCAT master, but there are a few common techniques, that are described in this section.

1. A first simple rule is, that *netif_*()*-calls must be strictly 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.

As mentioned before, the driver will handle both EtherCAT and ordinary Ethernet devices. This implies, that for each device-dependent operation, it has to be checked if an EtherCAT device is involved, or just an Ethernet device. For means of simplicity, this example driver will only handle one EtherCAT device. This makes the case differentiations easier.

Global Variables First of all, there have to be additional global variables declared, as shown in the listing:

```
1 static int ec_device_index = -1;
2 static int ec_device_master_index = 0;
3 static ec_device_t *rtl_ec_dev;
4 struct net_device *rtl_ec_net_dev = NULL;
```

(1) – (2) To comply to the requirements for parameters of EtherCAT device modules described in section 1.4.3, there have to be additional parameter variables: *ec_device_index* holds the index of the EtherCAT device and defaults to −1 (no EtherCAT device), while *ec_device_master_index* stores index of the master, the single device will be connected to. Default: 0

(3) *rtl_ec_dev* will be the pointer to the later registered RealTek EtherCAT device, which can be used as a parameter for device methods.

(4) *rtl_ec_net_dev* is a pointer to the *net_device* structure of the dedicated device and is set while scanning the PCI bus and finding the device with the specified index. This is done inside the *pci_module_init()* function executed as the first thing on module loading.

Module Initialization Below is the (shortened) coding of the device driver's module init function:

```
1  static int __init rtl8139_init_module(void)
2  {
3          if (pci_module_init(&rtl8139_pci_driver) < 0) {
4                  printk(KERN_ERR "Failed␣to␣init␣PCI␣mod.\n");
5                  goto out_return;
6          }
7
8          if (rtl_ec_net_dev) {
9                  printk(KERN_INFO "Registering"
10                        "␣EtherCAT␣device...\n");
11                 if (!(rtl_ec_dev =
12                       ecdev_register(ec_device_master_index,
13                                      rtl_ec_net_dev,
14                                      rtl8139_interrupt,
```

```
15                                     THIS_MODULE))) {
16                         printk(KERN_ERR "Failed␣to␣reg."
17                                        "␣EtherCAT␣device!\n");
18                         goto out_unreg_pci;
19                 }
20
21                 printk(KERN_INFO "Starting␣EtherCAT"
22                        "␣device...\n");
23                 if (ecdev_start(ec_device_master_index)) {
24                         printk(KERN_ERR "Failed␣to␣start"
25                                "␣EtherCAT␣device!\n");
26                         goto out_unreg_ec;
27                 }
28         } else {
29                 printk(KERN_WARNING "No␣EtherCAT␣device"
30                        "␣registered!\n");
31         }
32
33         return 0;
34
35     out_unreg_ec:
36         ecdev_unregister(ec_device_master_index, rtl_ec_dev);
37     out_unreg_pci:
38         pci_unregister_driver(&rtl8139_pci_driver);
39     out_return:
40         return -1;
41 }
```

(3) This call initializes all RTL8139-compatible devices found on the pci bus. If a device with index *ec_device_index* is found, a pointer to its *net_device* structure is stored in *rtl_ec_net_dev* for later use (see next listings).

(8) If the specified device was found, *rtl_ec_net_dev* is non-zero.

(11) The device is connected to the specified master with a call to *ecdev_register()*. If this fails, module loading is aborted.

(23) The device registration was successful and the master is started. This can fail, which aborts module loading.

(29) If no EtherCAT device was found, a warning is output.

Device Searching During the PCI initialization phase, a variable *board_idx* is increased for each RTL8139-compatible device found. The code below is executed for each device:

```
1 if (board_idx == ec_device_index) {
2         rtl_ec_net_dev = dev;
3         strcpy(dev->name, "ec0");
4 }
```

(1) The device with the specified index will be the EtherCAT device.

Avoiding Device Registration Later in the PCI initialization phase, the net_devices get registered. This has to be avoided for EtherCAT devices and so this is a typical example for an EtherCAT case differentiation:

```
1 if (dev != rtl_ec_net_dev) {
2         i = register_netdev(dev);
3         if (i) goto err_out;
4 }
```

(1) If the current net_device is not the EtherCAT device, it is registered at the network stack.

Avoiding Interrupt Registration In the next two listings, there is an interrupt requested and the device's interrupts are enabled. This also has to be encapsulated by if-clauses, because interrupt operation is not wanted for EtherCAT devices.

```
1 if (dev != rtl_ec_net_dev) {
2         retval = request_irq(dev->irq, rtl8139_interrupt,
3                              SA_SHIRQ, dev->name, dev);
4         if (retval) return retval;
5 }
```

```
1 if (dev != rtl_ec_net_dev) {
2         /* Enable all known interrupts by setting
3            the interrupt mask. */
4         RTL_W16(IntrMask, rtl8139_intr_mask);
5 }
```

Frame Sending The listing below shows an exerpt of the function representing the *hard_start_xmit()* callback of the net_device.

```
1 /* Note: the chip doesn't have auto-pad! */
2 if (likely(len < TX_BUF_SIZE)) {
3         if (len < ETH_ZLEN)
4                 memset(tp->tx_buf[entry], 0, ETH_ZLEN);
5         skb_copy_and_csum_dev(skb, tp->tx_buf[entry]);
```

```
6         if (dev != rtl_ec_net_dev) {
7                 dev_kfree_skb(skb);
8         }
9 } else {
10        if (dev != rtl_ec_net_dev) {
11                dev_kfree_skb(skb);
12        }
13        tp->stats.tx_dropped++;
14        return 0;
15 }
```

(6) + (10) The master uses a fixed socket buffer for transmission, which is reused and may not be freed.

Frame Receiving During ordinary frame reception, a socket buffer is created and filled with the received data. This is not necessary for an EtherCAT device:

```
1 if (dev != rtl_ec_net_dev) {
2         /* Malloc up new buffer, compatible with net-2e. */
3         /* Omit the four octet CRC from the length. */
4
5         skb = dev_alloc_skb (pkt_size + 2);
6         if (likely(skb)) {
7                 skb->dev = dev;
8                 skb_reserve(skb, 2); /* 16 byte align
9                                         the IP fields. */
10                eth_copy_and_sum(skb, &rx_ring[ring_off + 4],
11                                 pkt_size, 0);
12                skb_put(skb, pkt_size);
13                skb->protocol = eth_type_trans(skb, dev);
14
15                dev->last_rx = jiffies;
16                tp->stats.rx_bytes += pkt_size;
17                tp->stats.rx_packets++;
18
19                netif_receive_skb (skb);
20        } else {
21                if (net_ratelimit())
22                        printk(KERN_WARNING
23                               "%s:␣Memory␣squeeze,␣dropping"
24                               "␣packet.\n", dev->name);
25                tp->stats.rx_dropped++;
26        }
27 } else {
28        ecdev_receive(rtl_ec_dev,
```

```
29                          &rx_ring[ring_offset + 4], pkt_size);
30          dev->last_rx = jiffies;
31          tp->stats.rx_bytes += pkt_size;
32          tp->stats.rx_packets++;
33  }
```

(28) If the device is an EtherCAT device, no socket buffer is allocated. Instead a pointer to the data (which is still in the device's receive ring) is passed to the EtherCAT master. Unnecessary copy operations are avoided.

(30) – (32) The device's statistics are updated as usual.

Link State The link state (i. e. if there is a carrier signal detected on the receive port) is determined during execution of the ISR. The listing below shows the different processing for Ethernet and EtherCAT devices:

```
1  if (dev != rtl_ec_net_dev) {
2          if (tp->phys[0] >= 0) {
3                  mii_check_media(&tp->mii, netif_msg_link(tp),
4                                  init_media);
5          }
6  } else {
7          void __iomem *ioaddr = tp->mmio_addr;
8          uint16_t link = RTL_R16(BasicModeStatus)
9                          & BMSR_LSTATUS;
10         ecdev_link_state(rtl_ec_dev, link ? 1 : 0);
11 }
```

(3) The "media check" is done via the media independent interface (MII), a standard interface for Fast Ethernet devices.

(7) – (10) For EtherCAT devices, the link state is fetched manually from the appropriate device register, and passed to the EtherCAT master by calling *ecdev_link_state()*.

Module Cleanup Below is the module's cleanup function:

```
1  static void __exit rtl8139_cleanup_module (void)
2  {
3          printk(KERN_INFO "Cleaning␣up␣RTL8139-EtherCAT"
4                  "␣module...\n");
5
6          if (rtl_ec_net_dev) {
7                  printk(KERN_INFO "Stopping␣device...\n");
```

```
8                     ecdev_stop(ec_device_master_index);
9                     printk(KERN_INFO "Unregistering␣device...\n");
10                    ecdev_unregister(ec_device_master_index,
11                                     rtl_ec_dev);
12                    rtl_ec_dev = NULL;
13          }
14
15          pci_unregister_driver(&rtl8139_pci_driver);
16
17          printk(KERN_INFO "RTL8139-EtherCAT␣module"
18                 "␣cleaned␣up.\n");
19  }
```

⑥ Stopping and unregistration is only done, if a device was registered before.

⑧ The master is first stopped, so it does not access the device any more.

⑩ After this, the device is unregistered. The master is now “ophaned”.

1.5 The Master Module

The EtherCAT master is designed to run as a kernel module. Moreover the master kernel module *ec_master* can handle multiple masters at the same time: The number of masters has to be passed to the module with the parameter *ec_master_count*, that defaults to 1. A certain master can later be addressed by its index. For example, if the master module has been loaded with the command

```
host# modprobe ec_master ec_master_count=2
```

the two masters can be addressed by their indices 0 and 1 respectively (see figure 1.5). This master index mandatory for certain functions of the master interfaces.

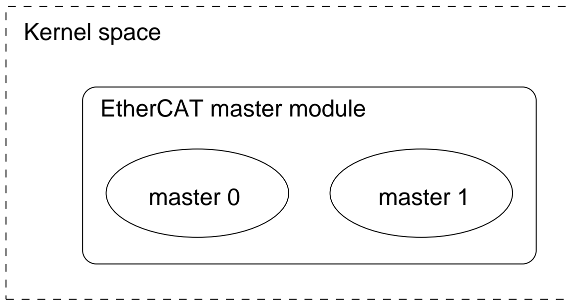

Figure 1.5: Multiple masters in one module

Master Log Messages The master module gives information about it's state and events via the syslog interface. The module loading command above should result in the following syslog messages:

```
EtherCAT: Master driver, 1.1 (stable) - rev. 513,
          compiled by fp at Aug  09 2006 09:43:50
EtherCAT: Initializing 2 EtherCAT master(s)...
EtherCAT: Initializing master 0.
EtherCAT: Initializing master 1.
EtherCAT: Master driver initialized.
```

The master provides information about it's version number, subversion revision number and compile information, like the date of compilation and the user, who compiled. All messages are prefixed either with `EtherCAT:`, `EtherCAT WARNING:` or `EtherCAT ERROR:`, which makes searching the logs easier.

1.5.1 Class Reference

This section is not intended to be a complete reference of master classes and functions[3], but will give a general survey of the master's classes, and how they interact.

Figure 1.6 shows an UML class diagram of the master classes.

The following subsections introduce serveral classes with their attributes and methods.

1.5.1.1 The Master Class

Figure 1.7 shows an UML class diagram of the master class. There is a short explanation of the attributes and methods below.

Master Attributes

list is a listhead structure that is needed to manage the list of masters in the master module (see section 1.5).

reserved is a flag, that marks the master as reserved for a realtime module, so that a call to *ecrt_request_master()* fails, if another module is already using the master.

index contains the number of the master. The first master will get index 0, the second index 1, and so on.

kobj In order to make the master object available via Sysfs (see section 1.8.1), this structure is needed inside the master object (see section 1.8.1).

[3]The comprehensive master reference can be obtained at http://etherlab.org/download/download-en.html

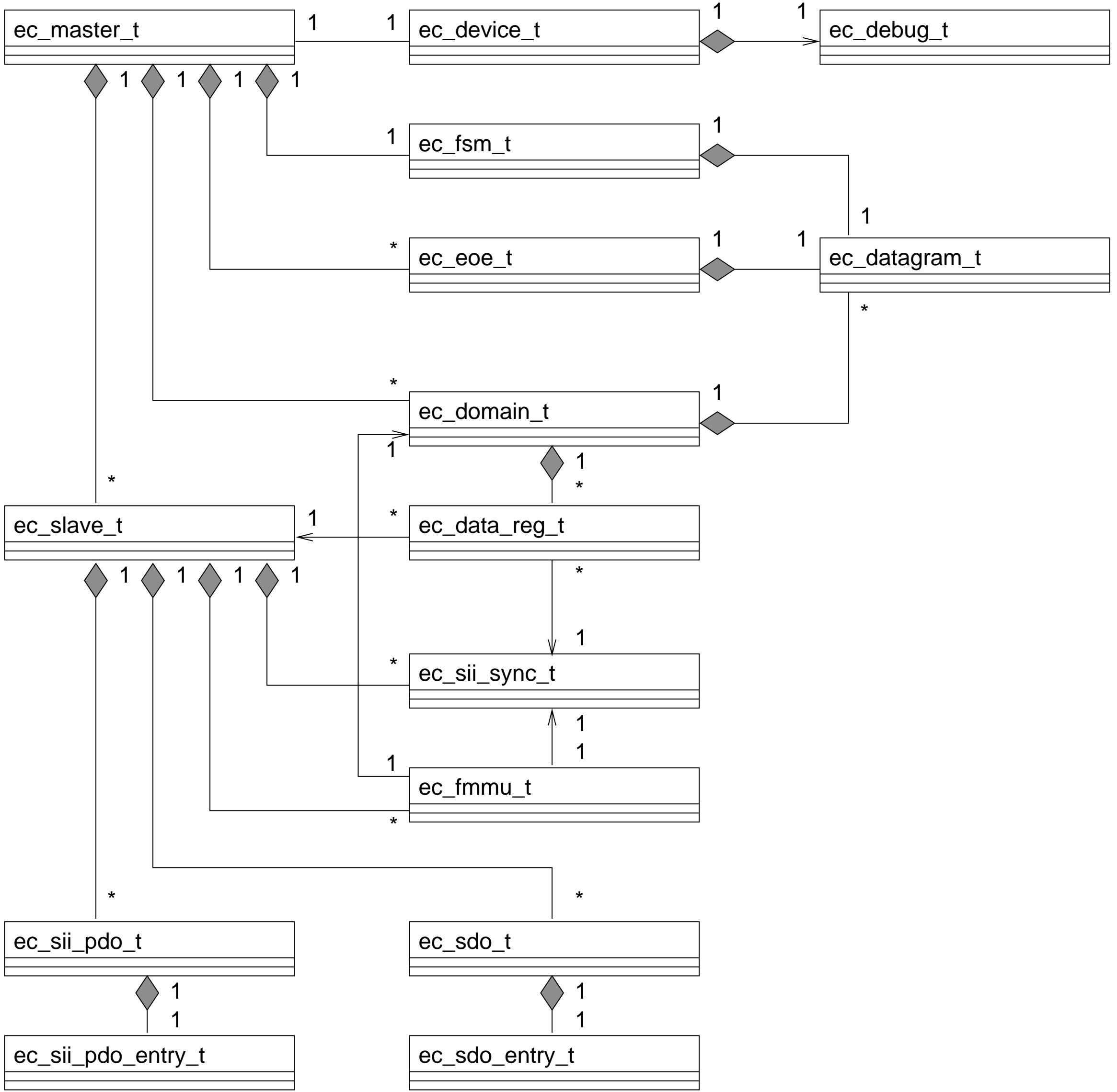

Figure 1.6: UML class diagram with associations

```
ec_master_t
-----------------------------------------
list: struct list_head
reserved: unsigned int
index: unsigned int
kobj: struct kobj
slaves: struct list_head
slave_count: unsigned int
device: ec_device_t *
datagram_queue: struct list_head
datagram_index: uint8_t
domains: struct list_head
debug_level: unsigned int
stats: ec_stats_t
workqueue: struct workqueue_struct *
idle_work: struct work_struct
fsm: ec_fsm_t
mode: ec_master_mode_t
eoe_timer: struct timer_list
eoe_running: unsigned int
eoe_handlers: struct list_head
internal_lock: spinlock_t
request_cb: int (*)(void *)
release_cb: void (*)(void *)
cb_data: void *
eeprom_write_enable: uint8_t
-----------------------------------------
+ ec_master_init()
+ ec_master_clear()
+ ec_master_reset()
+ ec_master_idle_start()
+ ec_master_idle_stop()
+ ec_master_eoe_start()
+ ec_master_eoe_stop()
+ ec_master_receive_datagrams()
+ ec_master_queue_datagram()
+ ec_master_output_stats()
+ ec_master_clear_slaves()
– ec_master_send_datagrams()
– ec_master_idle()
– ec_master_eoe_run()
+ ecrt_master_callbacks()
+ ecrt_master_create_domain()
+ ecrt_master_activate()
+ ecrt_master_deactivate()
+ ecrt_master_send()
+ ecrt_master_receive()
+ ecrt_master_prepare()
+ ecrt_master_run()
+ ecrt_master_start_eoe()
+ ecrt_master_debug()
+ ecrt_master_print()
+ ecrt_master_get_slave()
```

Figure 1.7: Master UML class diagram

slaves is the list of slaves. It consists of objects of the *ec_slave_t* class (see section 1.5.1.2).

slave_count is the number of slaves in the list.

device points to the network device that is used by the master to send and receive frames (see section 1.5.1.3). It is *NULL*, if no device is connected.

datagram_queue is a list of datagrams (see section 1.5.1.4) that have to be sent by the master, or have already been sent and wait to be received again. Upon reception or error, the datagrams are dequeued.

datagram_index contains the index value for the next datagram. The master stores this incrementing index into every datagram, to make it easier to assign a received datagram to the one sent before.

domains contains the list of domains created by the realtime module (section 1.5.1.5).

debug_level controls, how much debugging output is printed by the master: 0 means no debugging output, 1 means to output certain executing marks and actions, and 2 means to output frame contents in addition. This value can be changed at runtime via the Sysfs interface (see section 1.8.1).

stats is a statistics object that contains certain counters (like the number of missed frames). These statistics are output on demand, but at most once a second.

workqueue is the kernel workqueue used for idle mode.

idle_work is the work object, that is queued.

fsm The attribute *fsm* represents the master's finite state machine, that does all the slave processing. See sections 1.5.1.6 and 1.6 for further details.

mode contains the current master mode, if it is orphaned, idle, or in operation mode.

eoe_timer is the kernel timer used for EoE processing.

eoe_running marks the state of EoE processing.

eoe_handlers is the list of EoE handlers (see section 1.5.1.7).

internal_lock is a spinlock used in idle mode, that controls the concurrency of the idle and EoE processes.

request_cb The "request lock" callback function, the master has to provide for foreign instances, which want to access the master (see section 1.5.4).

release_cb The callback function that will release the master lock.

cb_data This value will be passed as an argument to each callback.

eeprom_write_enable flag can be written via Sysfs to enable the general writing of E^2PROM contents.

Public Master Methods

ec_master_init() is the master's constructor. It initializes all attributes, creates the workqueue, creates EoE handlers and the state machine object, and adds the kernel object to the Sysfs hierarchy.

ec_master_clear() is the destructor and undoes all these actions.

ec_master_reset() clears the master, but initializes it again. This is needed, when a realtime module disconnects: Slaves and other attributes are cleared and are later rebuilt by the idle process.

ec_master_idle_start/stop() These methods enable or disable the idle process.

ec_master_eoe_start/stop() These methods do the same for the EoE timer.

ec_master_receive_datagrams() This method is called by the device, which uses it to pass received frames to the master. The frame is dissected and the contained datagrams are assigned to the datagram objects in the datagram queue, which are dequeued on reception or error.

ec_master_queue_datagram() This method is used to queue a new datagram for sending and receiving.

ec_master_output_stats() This method is cyclically called to output a summary of the *stats* attribute at most once a second.

ec_master_clear_slaves() clears the list of slaves. This is needed on connection/disconnection of a realtime module or at a topology change in idle mode, when all slaves objects are rebuilt.

Private Master Methods A few of a master's methods are private, meaning, that they can only be called from other master methods:

ec_master_send_datagrams() searches the datagram queue for unsent datagrams, allocates frames to send them, does the actual sending and marks the datagrams as sent.

ec_master_idle_run() is the work function for the idle mode. It executes the idle state machine, described in section 1.6.4.

ec_master_eoe_run() is called by the EoE timer and is responsible for communicating with EoE-capable slaves. See section 1.7.1 for more information.

Master Methods (Realtime Interface) The master methods belonging to the EtherCAT realtime interface begin with the prefix *ecrt* instead of *ec*. The functions of the realtime interface are explained in section 1.5.2.2.

1.5.1.2 The Slave Class

Figure 1.8 shows an UML class diagram of the slave class. There is a short explanation of the attributes and methods below.

Slave Attributes

list The master holds a slave list, therefore the slave class must contain this structure used as an anchor for the linked list.

kobj This pointer serves as base object for the slave's Sysfs representation.

master is the pointer to the master owning this slave object.

ring_position is the logical position in the logical ring topology.

station_address is the configured station address. This is always the ring position + 1).

ec_slave_t
list: struct list_head kobj: struct kobj master: ec_master_t * ring_position: uint16_t station_address: uint16_t coupler_index: uint16_t coupler_subindex: uint16_t base_type: uint8_t base_revision: uint8_t base_build: uint16_t base_fmmu_count: uint16_t base_sync_count: uint16_t dl_link: uint8_t[4] dl_loop: uint8_t[4] dl_signal: uint8_t[4] sii_alias: uint16_t sii_vendor_id: uint32_t sii_product_code: uint32_t sii_revision_number: uint32_t sii_serial_number: uint32_t sii_rx_mailbox_offset: uint16_t sii_rx_mailbox_size: uint16_t sii_tx_mailbox_offset: uint16_t sii_tx_mailbox_size: uint16_t sii_mailbox_protocols: uint16_t sii_physical_layer: uint8_t[4] registered: uint8_t fmmus: ec_fmmu_t[EC_MAX_FMMUS] fmmu_count: uint8_t eeprom_data: uint8_t * eeprom_size: uint16_t eeprom_strings: struct list_head eeprom_syncs: struct list_head eeprom_pdos: struct list_head eeprom_group: char * eeprom_image: char * eeprom_order: char * eeprom_name: char * new_eeprom_data: uint8_t * new_eeprom_size: uint16_t sdo_dictionary: struct list_head requested_state: ec_slave_state_t current_state: ec_slave_state_t error_flag: unsigned int online: unsigned int varsize_fields: struct list_head
+ ec_slave_init() + ec_slave_clear() + ec_slave_prepare_fmmu() + ec_slave_fetch_strings() + ec_slave_fetch_general() + ec_slave_fetch_sync() + ec_slave_fetch_pdo() + ec_slave_locate_string() + ec_slave_calc_sync_size() + ec_slave_print() + ec_slave_mbox_prepare_send() + ec_slave_mbox_prepare_check() + ec_slave_mbox_check() + ec_slave_mbox_prepare_fetch() + ec_slave_mbox_fetch() + ec_slave_mbox_simple_io() + ec_slave_mbox_simple_receive() – ec_slave_write_eeprom() + ecrt_slave_conf_sdo8() + ecrt_slave_conf_sdo16() + ecrt_slave_conf_sdo32() + ecrt_slave_field_size()

Figure 1.8: Slave UML class diagram

coupler_index is the index of the last bus coupler.

coupler_subindex is the slave's position, counted from the last bus coupler. See section 1.5.3 for more information.

base_* These attributes contain base information about the slave, that are read from the "DL Information" attribute.

dl_* These fields store information of the "DL Status" attribute, for example states of the the communication ports.

sii_* These attributes contain values from the "Slave Information Interface" [2, section 6.4], mostly identity and mailbox information, but also the list of sync manager configurations and PDOs.

registered This flag is set, if one or more PDOs of the slave have been registered for process data exchange. Otherwise a warning is output, because the slave is unused.

fmmus Is an array of FMMU configurations, that have to be applied to the slave.

fmmu_count contains number of FMMUs used.

eeprom_* These fields contain E^2PROM contents and the extracted category information [3, section 5.4].

new_eeprom_data If this pointer is not *NULL*, it points to new E^2PROM contents, that have to be written to the slave.

new_eeprom_size This field represents the size of the new E^2PROM data.

requested_state is the requested slave state.

current_state is the current slave state.

error_flag is used by the operation and idle state machines to indicate, that a state transisition has failed and should not be tried again until an external event happens.

online This flag contains the online state of the slave (i. e. if it currently responds to the masters commands). Changes of the online state are always reported.

varsize_fields is only suitable for slaves that provide PDOs of variable size (like slaves that manage a sub-fieldbus) and contains information about what size this fields actually should have.

Public Slave Methods

ec_slave_init() The slave's constructor.

ec_slave_clear() The slave's destructor.

ec_prepare_fmmu() prepares an FMMU configuration. The FMMU is configured for a certain sync manager and domain.

ec_fetch_*() Serveral methods to extract information of the E^{2}PROM category contents.

ec_slave_locate_string() extracts a string out of a STRING category and allocates string memory.

ec_slave_calc_sync_size() calculates the size of sync manager contents, because they can be variable due to variable-sized PDOs.

ec_slave_info() This method prints all slave information into a buffer for Sysfs reading.

ec_slave_mbox_*() These functions prepare datagrams for mailbox communication, or process mailbox responses, respectively.

Private Slave Methods

ec_slave_write_eeprom() This function accepts E^{2}PROM data from user space, does a quick validation of the contents and schedules them for writing through the idle state machine.

Slave Methods (Realtime Interface)

ecrt_slave_conf_sdo*() These methods accept SDO configurations, that are applied on slave activation (i. e. everytime the slave is configured). The methods differ only in the data size of the SDO (8, 16 or 32 bit).

ecrt_slave_pdo_size() This method specifies the size of a variable-sized PDO.

1.5.1.3 The Device Class

The device class is responsible for communicating with the connected EtherCAT-enabled network driver. Figure 1.9 shows its UML class diagram.

```
ec_device_t
---
master: ec_master_t *
dev: struct net_device *
open: uint8_t
tx_skb: struct sk_buff *
isr: ec_isr_t *
module: struct module *
link_state: uint8_t
dbg: ec_debug_t
---
+ ec_device_init(): int
+ ec_device_clear()
+ ec_device_open(): int
+ ec_device_close(): int
+ ec_device_call_isr()
+ ec_device_tx_data(): uint8_t *
+ ec_device_send()
+ ecdev_receive()
+ ecdev_link_state()
```

Figure 1.9: Device UML class diagram

Device Attributes

master A pointer to the master, which owns the device object.

dev This is the pointer to the *net_device* structure of the connected network device driver.

open This flag stores, if the network device is "opened" and ready for transmitting and receiving frames.

tx_skb The transmittion socket buffer. Instead of allocating a new socket buffer for each frame, the same socket buffer is recycled and used for every frame.

isr The pointer to the network device's interrupt service routine. *ec_isr_t* is a type definition in the device interface, which looks like below:

```
typedef irqreturn_t (*ec_isr_t)(int, void *,
                                struct pt_regs *);
```

module A pointer to the network driver module, to increase and decrease the use counter (see paragraph "Implicit Dependencies" in section 1.4.4).

link_state The current link state. It can be 0 "down" or 1 "up".

dbg Every device objects contains a debugging interface (see sectios 1.5.1.8 and 1.8.4).

Public Device Methods

ec_device_init() The device constructor.

ec_device_clear() The device destructor.

ec_device_open() "Opens" the device for transmittion and reception of frames. This is equivalent to the *ifconfig up* command for ordinary Ethernet devices.

ec_device_close() Stops frame transmittion and reception. This is equivalent to the *ifconfig down* command for ordinary Ethernet devices.

ec_device_call_isr() Calls the interrupt service routine of the device.

ec_device_tx_data() Returns a pointer into the memory of the transmittion socket buffer *tx_skb*. This is used by the master to assemble a new EtherCAT frame.

ec_device_send() Sends an assembled frame by passing it to the device's *hard_start_-xmit()* callback.

Device Methods (Device Interface) The device methods belonging to the device interface are explained in section 1.4.4.

1.5.1.4 The Datagram Class

So send and receive a datagram, an object of the *ec_datagram_t* class is needed. It can be initialized with a datagram type [2, section 5.4] and length (optionally filled with data) and appended to the master's datagram queue. Figure 1.10 shows its UML class diagram.

Datagram Attributes

list This attribute is needed to make a list of datagrams, as used in the domain class (see section 1.5.1.5).

queue This attribute is the anchor to the master's datagram queue, which is implemented as a linked list.

type The datagram type. *ec_datagram_type_t* is an enumeration, which can have the values *EC_DATAGRAM_APRD*, *EC_DATAGRAM_APWR*, *EC_DATAGRAM_-NPRD*, *EC_DATAGRAM_NPWR*, *EC_DATAGRAM_BRD*, *EC_DATAGRAM_-BWR* or *EC_DATAGRAM_LRW*.

address The slave address. For all addressing schemes take 4 bytes, *ec_address_t* is a union type:

```
ec_datagram_t
---------------------------------
list: struct list_head
queue: struct list_head
type: ec_datagram_type_t
address: ec_address_t
data: uint8_t *
mem_size: size_t
data_size: size_t
index: uint8_t
working_counter: uint16_t
state: ec_datagram_state_t
t_sent: cylces_t
---------------------------------
+ ec_datagram_init()
+ ec_datagram_clear()
+ ec_datagram_prealloc(): int
+ ec_datagram_nprd():int
+ ec_datagram_npwr(): int
+ ec_datagram_aprd(): int
+ ec_datagram_apwr(): int
+ ec_datagram_brd(): int
+ ec_datagram_bwr(): int
+ ec_datagram_lrw(): int
```

Figure 1.10: Datagram UML class diagram

```
typedef union {
        struct {
                uint16_t slave; /**< configured or
                                     autoincrement
                                     address */
                uint16_t mem; /**< physical memory
                                   address */
        } physical; /**< physical address */
        uint32_t logical; /**< logical address */
} ec_address_t;
```

data The actual data of the datagram. These are either filled in before sending (at writing access) or are inserted by the adressed slave(s) (at reading access). In any case, the data memory must be dynamically allocated. Besides, this can be done before cyclic processing with the *ec_datagram_prealloc()* method (see below).

mem_size The size of the allocated memory, *data* points to.

data_size The size of the actual data in the *data* memory.

index The sequential EtherCAT datagram index. This value is set by the master on sending, to easier assign a received datagram to a queued datagram object.

working_counter The working counter of the datagram. This is set to zero on sending and filled with the real value of the working counter on datagram reception.

state The state of the datagram. *ec_datagram_state_t* is an enumeration and can be *EC_DATAGRAM_INIT*, *EC_DATAGRAM_QUEUED*, *EC_DATAGRAM_SENT*, *EC_DATAGRAM_RECEIVED*, *EC_DATAGRAM_TIMED_OUT* or *EC_DATAGRAM_ERROR*.

t_sent This attribute is set to the timestamp, when the datagram was sent, to later detect a timeout.

Public Datagram Methods

ec_datagram_init() The datagram's constructor.

ec_datagram_clear() The datagram's destructor.

ec_datagram_prealloc() Allocates memory for the datagram data. This is especially needed, if the datagram structure will later be used in a context, where no dynamic memory allocation is allowed.

ec_datagram_nprd() Initializes a "Node-Addressed Physical Read" datagram [2, section 5.4.1.2].

ec_datagram_npwr() Initializes a "Node-Addressed Physical Write" datagram [2, section 5.4.2.2].

ec_datagram_aprd() Initializes a "Auto-Increment Physical Read" datagram [2, section 5.4.1.1].

ec_datagram_apwr() Initializes a "Auto-Increment Physical Write" datagram [2, section 5.4.2.1].

ec_datagram_brd() Initializes a "Broadcast Read" datagram [2, section 5.4.1.3].

ec_datagram_bwr() Initializes a "Broadcast Write" datagram [2, section 5.4.2.3].

ec_datagram_lrw() Initializes a "Logical ReadWrite" datagram [2, section 5.4.3.4].

1.5.1.5 The Domain Class

The domain class encapsules PDO registration and management of the process data image and its exchange. The UML class diagram can be seen in figure 1.11.

```
ec_domain_t
-----------------------------------------------
kobj: struct kobject
list: struct list_head
index: unsigned int
master: ec_master_t *
data_size: size_t
datagrams: struct list_head
base_address: uint32_t
response_count: unsigned int
data_regs: struct list_head
working_counter_changes: unsigned int
t_last: cycles_t
-----------------------------------------------
+ ec_domain_init(): int
+ ec_domain_clear()
+ ec_domain_alloc()
+ ec_domain_queue()
– ec_domain_reg_pdo_entry(): int
– ec_domain_clear_data_regs()
– ec_domain_add_datagram(): int
+ ecrt_domain_register_pdo(): ec_slave_t *
+ ecrt_domain_register_pdo_list(): int
+ ecrt_domain_process()
+ ecrt_domain_state(): int
```

Figure 1.11: Domain UML class diagram

Domain Attributes

kobj This *kobject* structure is needed for the Sysfs representation of the domain.

list The master manages a list of domains, so this list anchor is needed.

index The domain's index. The first domain will get index 0, the second index 1, and so on.

master A pointer to the master owning the domain.

data_size The size of the domain's process data image.

datagram A linked list with the datagram objects, the domain needs for process data exchange (see section 1.5.1.4).

base_address This attribute stores the logical offset, to which the domain's process data are mapped.

response_count The sum of the datagrams' working counters at the last process data exchange. Changes are always reported.

data_regs The (linked) list of PDO registrations. The realtime module requests the exchange of certain PDOs and supplies the address of process data pointers, that will later point to the respective locations in the process data image. These "data registrations" are saved in the *data_regs* list.

working_counter_changes This field stores the number of working counter changes since the last notification. This helps to reduce syslog output in case of frequent changes.

t_last The timestamp of the last working counter change notification.

Public Domain Methods

ec_domain_init() The domain's constructor.

ec_domain_clear() The domain's destructor.

ec_domain_alloc() Allocates the process data image and the respective datagrams based on the process data registrations.

ec_domain_queue() Queues the domain's datagrams for exchange via the master.

Private Domain Methods

ec_domain_reg_pdo_entry() This method is used to do a PDO registration. It finds the appropriate sync manager covering the PDO data, calculates its offset in the sync-manager-protected memory and prepares the FMMU configurations for the related slave. Then the PDO registration is appended to the list.

ec_domain_clear_data_regs() Clearing all process data registrations is needed in serveral places and therefore has been sourced out to an own method.

ec_domain_add_datagram() This methods allocates a datagram and appends it to the list. This is done during domain allocation.

Domain Methods (Realtime Interface) The domain methods belonging to the realtime interface are introduced in section 1.5.2.3.

1.5.1.6 The Finite State Machine Class

This class encapsules all state machines, except the EoE state machine. Its UML class diagram can be seen in figure 1.12.

ec_fsm_t
master: ec_master_t * slave: ec_slave_t * datagram: ec_datagram_t master_state: void (*)(ec_fsm_t *) master_slaves_responding: unsigned int master_slave_states: ec_slave_state_t master_validation: unsigned int slave_state: void (*)(ec_fsm_t *) sii_state: void (*)(ec_fsm_t *) sii_offset: uint16_t sii_mode: unsigned int sii_value: uint8_t[4] sii_start: cycles_t change_state: void (*)(ec_fsm_t *) change_new: ec_sllave_state_t change_start: cycles_t coe_state: void (*)(ec_fsm_t *) sdodata: ec_sdo_data_t * coe_start: cycles_t
+ ec_fsm_init(): int + ec_fsm_clear() + ec_fsm_reset() + ec_fsm_execute() + ec_fsm_startup() + ec_fsm_startup_running(): int + ec_fsm_startup_success(): int + ec_fsm_configuration() + ec_fsm_configuration_running(): int + ec_fsm_configuration_success(): int – ec_fsm_master_start() – ec_fsm_master_broadcast() – ec_fsm_master_read_states() – ec_fsm_master_validate_vendor() – ec_fsm_master_validate_product() – ec_fsm_master_rewrite_addresses() – ec_fsm_master_configure_slaves() – ec_fsm_master_write_eeprom() – ec_fsm_startup_start() – ec_fsm_startup_broadcast() – ec_fsm_startup_scan() – ec_fsm_configuration_start() – ec_fsm_configuration_conf() – ec_fsm_slavescan_start() – ec_fsm_slavescan_address() – ec_fsm_slavescan_state() – ec_fsm_slavescan_base() – ec_fsm_slavescan_datalink() – ec_fsm_slavescan_eeprom_size() – ec_fsm_slavescan_eeprom_data() – ec_fsm_slaveconf_init() – ec_fsm_slaveconf_sync() – ec_fsm_slaveconf_preop() – ec_fsm_slaveconf_fmmu() – ec_fsm_slaveconf_sdoconf() – ec_fsm_slaveconf_saveop() – ec_fsm_slaveconf_op() – ec_fsm_sii_start_reading() – ec_fsm_sii_read_check() – ec_fsm_sii_read_fetch() – ec_fsm_sii_start_writing() – ec_fsm_sii_write_check() – ec_fsm_sii_write_check2() – ec_fsm_change_start() – ec_fsm_change_check() – ec_fsm_change_status() – ec_fsm_change_code() – ec_fsm_change_ack() – ec_fsm_change_check_ack() – ec_fsm_coe_down_start() – ec_fsm_coe_down_request() – ec_fsm_coe_down_check() – ec_fsm_coe_down_response() – ec_fsm_end() – ec_fsm_error()

Figure 1.12: Finite State Machine UML class diagram

FSM Attributes

master A pointer to the master owning the FSM object.

slave Serveral sub state machines are executed for single slaves. This pointer stores the current slave for these FSMs.

datagram The FSM class has its own datagram, which is used in every state and sub-state.

master_state This function pointer stores the current state function for one of the master's state machines.

master_slaves_responding This attribute is used in the operation state machine (see section 1.6.3) to store the number of slaves, that responded to the last broadcast command.

master_slave_states This attribute stores the slave states, that were determined by the last broadcast command.

master_validation This flag is used by the operation state machine and is non-zero, if a bus validation has to be done.

slave_state This function pointer stores the current state of the slave scan state machine (see section 1.6.5) or the slave configuration state machine (see section 1.6.6).

sii_state This function pointer stores the current state of the SII state machine (see section 1.6.8).

sii_offset This attribute is used by the SII state machine to store the word address for the current read or write cycle.

sii_mode If this attribute is zero, the SII access is done with "auto-increment" datagrams [2, section 5.4]. If it is non-zero, "station-address" datagrams are used.

sii_value This attribute stores the value to write, or the read value, respectively.

sii_start A timestamp attribute, that stores the beginning time of an SII operation to detect a timeout.

change_state This function pointer stores the current state of the state change state machine.

change_new This attribute stores the requested state for the state change state machine.

change_start A timestamp attribute to detect a timeout while changing slave states.

coe_state This function pointer stores the current state of the CoE state machines.

sdodata This is an SDO data object that stores information about the current SDO to write.

coe_start A timestamp attribute to detect timeouts during CoE configuration.

Public FSM Methods

ec_fsm_init() Constructor of the FSM class.

ec_fsm_clear() Destructor of the FSM class.

ec_fsm_reset() Resets the whole FSM object. This is needed to restart the master state machines.

ec_fsm_execute() Executes one state of the current state machine and then returns.

ec_fsm_startup() Initializes the master startup state machine, which determines the number of slaves and executes the slave scan state machine for each slave.

ec_fsm_startup_running() Returns non-zero, if the startup state machine did not terminate yet.

ec_fsm_startup_success() Returns non-zero, if the startup state machine terminated with success.

ec_fsm_configuration() Initializes the master configuration state machine, which executes the slave configuration state machine for each slave.

ec_fsm_configuration_running() Returns non-zero, if the configuration state machine did not terminate yet.

ec_fsm_configuration_success() Returns non-zero, if the configuration state machine terminated with success.

FSM State Methods The rest of the methods showed in the UML class diagram are state methods of the state machines. These states are described in section 1.6.

1.5.1.7 The EoE Class

Objects of the *ec_eoe_t* class are called EoE handlers. Each EoE handler represents a virtual network interface and can be coupled to a EoE-capable slave on demand. The UML class diagram can be seen in figure 1.13.

ec_eoe_t
list: struct list_head slave: ec_slave_t * datagram: ec_datagram_t state: void (*)(ec_eoe_t *) dev: struct net_device * stats: struct net_device_stats opened: unsigned int t_last: cycles_t rx_skb: struct sk_buff * rx_skb_offset: off_t rx_skb_size: size_t rx_expected_fragment: uint8_t rx_counter: uint32_t rx_rate: uint32_t tx_queue: struct list_head tx_queue_active: unsigned int tx_queued_frames: unsigned int tx_queue_lock: spinlock_t tx_frame: ec_eoe_frame_t tx_frame_number: uint8_t tx_fragment_number: uint8_t tx_offset: size_t tx_counter: uint32_t tx_rate: uint32_t
+ ec_eoe_init(): int + ec_eoe_clear() + ec_eoe_run() + ec_eoe_active(): int – ec_eoe_flush() – ec_eoe_send() – ec_eoe_state_rx_start() – ec_eoe_state_rx_check() – ec_eoe_state_rx_fetch() – ec_eoe_state_tx_start() – ec_eoe_state_tx_sent()

Figure 1.13: EoE UML class diagram

EoE Attributes

list The master class maintains a list of EoE handlers. Therefore this list anchor is needed.

slave If an EoE handler is coupled to a slave, this pointer points to the related slave object. Otherwise it is *NULL*.

datagram Every EoE handler owns a datagram object to exchange data with the coupled slave via its state machine.

state This function pointer points to the current state of the EoE state machine (see section 1.7.1).

dev A pointer to the *net_device* structure that represents the network interface to the kernel.

stats The statistics object for the network interface.

opened This flag stores, if the network interface was opened. No EoE processing will be done, if the device is not opened.

t_last This timestamp attribute stores the time of the last bit rate measurement.

rx_skb A pointer to the current receive socket buffer. On every first fragment of a received frame, a new receive socket buffer is allocated. On every last fragment, this buffer will be passed to the network stack.

rx_skb_offset This attribute stores the offset for the next fragment data in the receive socket buffer.

rx_skb_size This attribute stores the current data size of the receive socket buffer.

rx_expected_fragment The expected number of the next fragment. If a fragment with an invalid number is received, the whole frame will be dropped.

rx_counter This is the sum of the octets received since the last bit rate measurement.

rx_rate This attribute stores the receive bit rate in bps.

tx_queue Each EoE handler maintains a transmittion queue for frames, that come in via the network interface. This queue is implemented with a linked list and protected by a spinlock.

tx_queue_active This flag stores, if the transmittion queue is currently accepting frames from the network stack. If the queue gets filled up, frame transmittion is suspended with a call to *netif_stop_queue()*. If the fill state decreases below the half capacity, frame transmittion is restarted with *netif_wake_queue()*.

tx_queued_frames The number of frames in the transmittion queue.

tx_queue_lock The spinlock used to protect the transmittion queue. This is needed, because the queue is accessed both from network stack context and from the master's EoE timer.

tx_frame The frame that is currently sent. The *ec_eoe_frame_t* structure combines the socket buffer structure with a list head to append it to the transmittion queue.

tx_frame_number The EoE protocol demands to maintain a sequencial frame number, that must be increased with every frame sent.

tx_fragment_number The sequencial number of the next fragment to transmit.

tx_offset Current frame data offset for the next fragment to transmit.

tx_counter The number of octets transferred since the last bit rate measurement.

tx_rate The recent transmittion bit rate in bps.

Public EoE Methods

ec_eoe_init() The EoE handler's constructor. The network interface is allocated and registered.

ec_eoe_clear() The EoE handler's destructor. The network interface is unregistered and all allocated memory is freed.

ec_eoe_run() Executes the EoE state machine (see section 1.7.1) for this handler.

ec_eoe_active() Returns true, if the handler has a slave coupled and the network interface is opened.

Private EoE Methods

ec_eoe_flush() Clears the transmittion queue and drops all frames queued for sending.

ec_eoe_send() Sends one fragment of the current frame.

EoE State Methods The rest of the private methods are state functions for the EoE state machine, which is discussed in section 1.7.1.

1.5.1.8 The Debug Class

The debug class maintains a virtual network interface. All frames that are sent and received by the master will be forwarded to this network interface, so that bus monitoring can be done with third party tools (see section 1.8.4). Figure 1.14 shows the UML class diagram.

ec_debug_t
dev: struct net_device * stats: struct net_device_stats opened: uint8_t
+ ec_debug_init(): int + ec_debug_clear() + ec_debug_send()

Figure 1.14: Debug UML class diagram

Debug Attributes

dev A pointer to the allocated *net_device* structure that represents the network interface in the kernel.

stats An object for interface statistics.

opened Stores the state of the device. Frames will only be forwarded, if the device was opened with the *ifconfig up* command (or something similar).

Public Debug Methods

ec_debug_init() The constructor.

ec_debug_clear() The destructor.

ec_debug_send() This method forwards a frame to the virtual network interface. It dynamically allocates a new socket buffer and passes it to the network stack.

1.5.2 The Realtime Interface

The realtime interface provides functions and data structures for realtime modules to access and use an EtherCAT master.

1.5.2.1 Master Requesting and Releasing

Before a realtime module can access am EtherCAT master provided by the master module, it has to reserve one for exclusive use. After use, it has to release the requested master and make it available for other modules. This is done with the following functions:

```
ec_master_t *ecrt_request_master(unsigned int master_index);
void ecrt_release_master(ec_master_t *master);
```

The *ecrt_request_master()* function has to be the first function a module has to call, when using EtherCAT. The function takes the index of the master as its argument. The first master has index 0, the nth master has index $n-1$. The number of existent masters has to be specified when loading the master module (see section 1.5). The function tries to reserve the specified master and scans for slaves. It returns a pointer to the reserved master object upon success, or *NULL* if an error occured.

The *ecrt_release_master()* function releases a reserved master after use. It takes the pointer to the master object returned by *ecrt_request_master()* as its argument and can never fail.

1.5.2.2 Master Methods

Domain Creation For process data exchange, at least one process data domain is needed (see section 1.3.1).

```
ec_domain_t *ecrt_master_create_domain(ec_master_t *master);
```

The *ecrt_master_create_domain()* method creates a new process data domain and returns a pointer to the new domain object. This object can be used for registering process data objects and exchange process data in cyclic operation. On failure, the function returns *NULL*.

Slave Handlers To access a certain slave, there is a method to get a slave handler:

```
ec_slave_t *ecrt_master_get_slave(const ec_master_t *,
                                  const char *);
```

The *ecrt_master_get_slave()* method returns a pointer to a certain slave object, specified by its ASCII address (see section 1.5.3). If the address is invalid, *NULL* is returned.

Master Activation When all domains are created, and all process data objects are registered, the master can be activated:

```
int ecrt_master_activate(ec_master_t *master);
void ecrt_master_deactivate(ec_master_t *master);
```

By calling the *ecrt_master_activate()* method, all slaves are configured according to the prior method calls and are brought into *OP* state. In this case, the method has a return value of 0. Otherwise (wrong configuration or bus failure) the method returns non-zero.

The *ecrt_master_deactivate()* method is the counterpart to the activate call: It brings all slaves back into *INIT* state again. This method should be called prior to *ecrt_master_release()*.

Locking Callbacks For concurrent master access, the realtime module has to provide a locking mechanism (see section 1.5.4):

```
void ecrt_master_callbacks(ec_master_t *master,
                           int (*request_cb)(void *),
                           void (*release_cb)(void *),
                           void *cb_data);
```

The “request lock” and “release lock” callbacks can be set with the *ecrt_master_callbacks()* method. It takes two function pointers and a data value as additional arguments. The arbitrary data value will be passed as argument on every callback. Asynchronous master access (like EoE processing) is only possible if these callbacks have been set.

Preparation of Cyclic Data Exchange Cyclic operation mostly consists of the three steps input, processing and output. In EtherCAT terms this would mean: Receive datagrams, evaluate process data and send datagrams. The first cycle differs from this principle, because no datagrams have been sent yet, so there is nothing to receive. To avoid having a case differantiation (in terms of an *if* clause), the following method exists:

```
void ecrt_master_prepare(ec_master_t *master);
```

As a last thing before cyclic operation, a call to the *ecrt_master_prepare()* method should be issued. It makes all process data domains queue their datagrams and issues a send command, so that the first receive call in cyclic operation will not fail.

Frame Sending and Receiving To send all queued datagrams and to later receive the sent datagrams there are two methods:

```
void ecrt_master_send(ec_master_t *master);
void ecrt_master_receive(ec_master_t *master);
```

The *ecrt_master_send()* method takes all datagrams, that have been queued for transmission, packs them into frames, and passes them to the network device for sending.

The *ecrt_master_receive()* queries the network device for received frames (by calling the ISR), 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 then dequeued.

Running the Operation State Machine The master's operation state machine (see section 1.6.3) monitors the bus in cyclic operation and reconfigures slaves, if necessary. Therefore, the following method should be called cyclically:

```
void ecrt_master_run(ec_master_t *master);
```

The *ecrt_master_run()* method executes the master's operation state machine step by step. It returns after processing one state and queuing a datagram. Calling this function is not mandatory, but highly recommended.

Master Monitoring It is also highly recommended to evaluate the master's error state. In this way it is possible to notice lost network links, failed bus segments, and other issues:

```
int ecrt_master_state(const ec_master_t *master);
```

The *ecrt_master_state()* method returns the master's error state. The following states are defined as part of the realtime interface:

EC_MASTER_OK means, that no error has occurred.

EC_MASTER_LINK_ERROR means, that the network link is currently down.

EC_MASTER_BUS_ERROR means, that one or more slaves do not respond.

1.5.2.3 Domain Methods

PDO Registration To access data of a slave's PDO in cyclic operation, it is necessary to make it part of a process data domain:

```
ec_slave_t *ecrt_domain_register_pdo(ec_domain_t *domain,
                                     const char *address,
                                     uint32_t vendor_id,
                                     uint32_t product_code,
                                     const char *pdo_name
                                     void **data_ptr);
int ecrt_domain_register_pdo_list(ec_domain_t *domain,
                                  const ec_pdo_reg_t *pdos);
```

The *ecrt_domain_register_pdo()* method registers a certain PDO as part of the domain and takes the address of the process data pointer. This pointer will be set on master activation and then can be parameter to the *EC_READ_** and *EC_WRITE_** macros described below.

A perhaps easier way to register multiple PDOs at the same time is to fill an array of *ec_pdo_reg_t* and hand it to the *ecrt_domain_register_pdo_list()* method. Attention: This array has to be terminated by an empty structure ({})!

Evaluating Domain Data To evaluate domain data, the following method has to be used:

```
void ecrt_domain_process(ec_domain_t *domain);
```

The *ecrt_domain_process()* method sets the domains state and requeues its datagram for sending.

Domain State Similar to the master state, a domain has an own error state:

```
int ecrt_domain_state(const ec_domain_t *domain);
```

The *ecrt_domain_state()* method returns the domain's error state. It is non-zero if not all process data values could be exchanged, and zero otherwise.

1.5.2.4 Slave Methods

SDO Configuration To configure slave SDOs, the function interface below can be used:

```
int ecrt_slave_conf_sdo8(ec_slave_t *slave,
                         uint16_t sdo_index,
                         uint8_t sdo_subindex,
                         uint8_t value);
int ecrt_slave_conf_sdo16(ec_slave_t *slave,
                          uint16_t sdo_index,
                          uint8_t sdo_subindex,
                          uint16_t value);
int ecrt_slave_conf_sdo32(ec_slave_t *slave,
                          uint16_t sdo_index,
                          uint8_t sdo_subindex,
                          uint32_t value);
```

The *ecrt_slave_conf_sdo*()* methods prepare the configuration of a certain SDO. The index and subindex of the SDO, and the value have to be specified. The configuration is done each time, the slave is reconfigured. The methods only differ in the SDO's data type. If the configuration could be prepared, zero is returned. If an error occured, non-zero is returned.

Variable-sized PDOs For specifying the size of variable-sized PDOs, the following method can be used:

```
int ecrt_slave_pdo_size(ec_slave_t *slave,
                        const char *pdo_name,
                        size_t size);
```

The *ecrt_slave_pdo_size()* method takes the name of the PDO and the size. It returns zero on success, otherwise non-zero.

1.5.2.5 Process Data Access

The endianess of the process data could differ from that of the CPU. Therefore, process data access has to be done by the macros below, that are also provided by the realtime interface:

```
#define EC_READ_BIT(DATA, POS)
#define EC_WRITE_BIT(DATA, POS, VAL)

#define EC_READ_U8(DATA)
#define EC_READ_S8(DATA)
#define EC_READ_U16(DATA)
#define EC_READ_S16(DATA)
#define EC_READ_U32(DATA)
#define EC_READ_S32(DATA)

#define EC_WRITE_U8(DATA, VAL)
#define EC_WRITE_S8(DATA, VAL)
#define EC_WRITE_U16(DATA, VAL)
#define EC_WRITE_S16(DATA, VAL)
#define EC_WRITE_U32(DATA, VAL)
#define EC_WRITE_S32(DATA, VAL)
```

There are macros for bitwise access (*EC_READ_BIT()*, *EC_WRITE_BIT()*), and bytewise access (*EC_READ_*()*, *EC_WRITE_*()*). The bytewise macros carry the data type in their name. Example: *EC_WRITE_S16()* writes a 16 bit signed value to EtherCAT data. The *DATA* parameter is supposed to be a process data pointer, as provided at PDO registration.

The macros use the kernel's endianess conversion macros, that are preprocessed to empty macros in case of equal endianess. This is the definition for the *EC_READ_U16()* macro:

```
#define EC_READ_U16(DATA) \
        ((uint16_t) le16_to_cpup((void *) (DATA)))
```

The *le16_to_cpup()* macro converts a little-endian, 16 bit value to the CPU's architecture and takes a pointer to the input value as its argument. If the CPU's architecture

is little-endian, too (for example on X86 and compatible), nothing has to be converted. In this case, the macro is replaced with an empty macro by the preprocessor and so there is no unneeded function call or case differentiation in the code.

For keeping it portable, it is highly recommended to make use of these macros.

1.5.3 Slave Addressing

The master offers the serveral slave addressing schemes (for PDO registration or configuration) via the realtime interface. For this reason, slave addresses are ASCII-coded and passed as a string. The addressing schemes are independent of the EtherCAT protocol and represent an additional feature of the master.

Below, the allowed addressing schemes are described. The descriptions are followed by a regular expression formally defining the addressing scheme, and one or more examples.

Position Addressing This is the normal addressing scheme, where each slave is addressed by its ring position. The first slave has address 0, and the nth slave has address $n - 1$. This addressing scheme is useful for small busses that have a fixed number of slaves.
RegEx: `[0-9]+` — Example: `"42"`

Advanced Position Addressing Bus couplers segment the bus into (physical) blocks. Though the logical ring positions keep being the same, it is easier to address a slave with its block number and the relative position inside the block. This addressing is done by passing the (zero-based) index of the bus coupler (not the coupler's ring position), followed by a colon and the relative position of the actual slave starting at the bus coupler.
RegEx: `[0-9]+:[0-9]+` — Examples: `"0:42"`, `"2:7"`

Alias Addressing Each slave can have a "secondary slave address" or "alias address"[4] stored in its E^2PROM. The alias is evaluated by the master and can be used to address the slave, which is useful when a clearly defined slave has to be addressed and the ring position is not known or can change over time. This scheme is used by starting the address string with a mesh (#) followed by the alias address. The latter can also be provided as hexadecimal value, prefixed with *0x*.
RegEx: `#(0x[0-9A-F]+|[0-9]+)` — Examples: `"#6622"`, `"#0xBEEF"`

Advanced Alias Addressing This is a mixture of the "Alias Addressing" and "Advanced Position Addressing" schemes. A certain slave is addressed by specifying its relative position after an aliased slave. This is very useful, if a complete block of slaves can vary its position in the bus. The bus coupler preceeding the block should get an alias. The block slaves can then be addressed by specifying this

[4]Information about how to set the alias can be found in section 1.8.1.4

alias and their position inside the block. This scheme is used by starting the address string with a mesh (#) followed by the alias address (which can be hexadecimal), then a colon and the relative posision of the slave to address.
RegEx: `#(0x[0-9A-F]+|[0-9]+):[0-9]+` — Examples: `"#0xBEEF:7"`, `"#6:2"`

In anticipation of section 1.5.2, the functions accepting these address strings are *ecrt_-master_get_slave()*, *ecrt_domain_register_pdo()* and *ecrt_domain_register_pdo_list()* (the latter through the *ec_pdo_reg_t* structure).

1.5.4 Concurrent Master Access

In some cases, one master is used by serveral instances, for example when a realtime module does cyclic process data exchange, and there are EoE-capable slaves that require to exchange Ethernet data with the kernel (see section 1.7.1). For this reason, the master is a shared resource, and access to it has to be sequenctialized. 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. Imagine, the realtime module uses RTAI functionality, then ordinary kernel semaphores would not be sufficient. For that, an important design decision was made: The realtime module 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 master lock by callbacks, that have to be set by the realtime module. Moreover the realtime module can deny access to the master if it consideres it to be awkward at the moment.

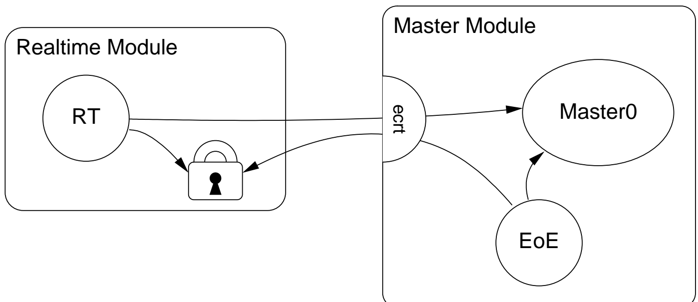

Figure 1.15: Concurrent master access

Figure 1.15 exemplary shows, how two processes share one master: The realtime module's cyclic thread uses the master for process data exchange, while the master-internal EoE process uses it to communicate with EoE-capable slaves. Both have to aquire the master lock before access: The realtime thread can access the lock natively, while the EoE process has to use the master callbacks. Section 2.4 gives an example, of how to implement this.

1.6 The Master's 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[5]. 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[6]) 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 1.6.1 will introduce some of the theory used, while the 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
```

[5]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.

[6]At this time, synchronous master access will be adequate to show the advantages of an FSM. The asynchronous approach will be discussed in section 1.7.1

```
4 next_state = state_2;
5 // 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:

```
1 // state 2
2 if (datagram->state != EC_DGRAM_STATE_RECEIVED) {
3         next_state = state_error;
4         return; // state processing finished
5 }
6 slave_states = EC_READ_U8(datagram->data); // process datagram
7 // state processing finished.
```

See section 1.6.2 for an introduction to the state machine programming concept used in the master code.

1.6.1 State Machine Theory

A finite state machine [7] 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 1.1). 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.

The state diagram for the same example looks like the one in figure 1.16. 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.

Table 1.1: 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

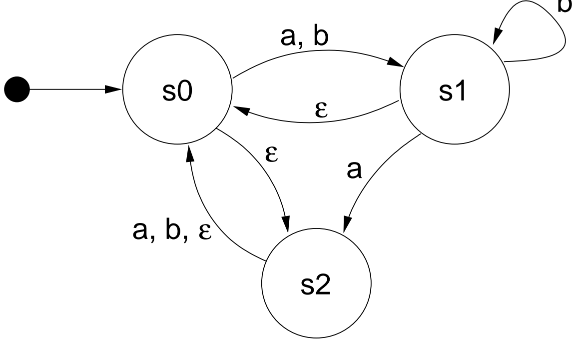

Figure 1.16: A typical state transition diagram

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 oftenly taken as a counter-argument against general use of state machines in complex environments. It has to be mentioned, that this point is misleading [8]. 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 serveral state machines, that are executed hierarchically and so serve as sub state machines. These are also described below.

1.6.2 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:

```
1  enum {STATE_1, STATE_2, STATE_3};
2  int state = STATE_1;
3
4  void state_machine_run(void *priv_data) {
5          switch (state) {
6                  case STATE_1:
7                          action_1();
8                          state = STATE_2;
9                          break;
10                 case STATE_2:
11                         action_2()
12                         if (some_condition) state = STATE_1;
13                         else state = STATE_3;
14                         break;
15                 case STATE_3:
16                         action_3();
17                         state = STATE_1;
18                         break;
19         }
20 }
```

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:

```
1  void (*state)(void *) = state1;
2
3  void state_machine_run(void *priv_data) {
4          state(priv_data);
5  }
6
7  void state1(void *priv_data) {
8          action_1();
9          state = state2;
10 }
```

```
11
12 void state2(void *priv_data) {
13         action_2();
14         if (some_condition) state = state1;
15         else state = state2;
16 }
17
18 void state3(void *priv_data) {
19         action_3();
20         state = state1;
21 }
```

In the master code, state pointers of all state machines[7] are gathered in a single object of the *ec_fsm_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 1.6.1. As mentioned, the "Mealy" model offers a higher flexibility, which can be seen in the listing below:

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

```
1  void state9(void *priv_data) {
2          action_9();
3          if (some_condition) {
4                  action_9a();
5                  state = state7;
```

[7] 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.

```
6           }
7           else {
8                   action_9b();
9                   state = state10;
10          }
11  }
```

This model is oftenly 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 encapsule the related workflows and moreover avoids the "state explosion" phenomenon described in section 1.6.1. 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:

```
1   void ec_fsm_slaveconf_saveop(ec_fsm_t *fsm)
2   {
3           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.

1.6.3 The Operation State Machine

The Operation state machine is executed by calling the *ecrt_master_run()* method in cyclic realtime code. Its purpose is to monitor the bus and to reconfigure slaves after a bus failure or power failure. Figure 1.17 shows its transition diagram.

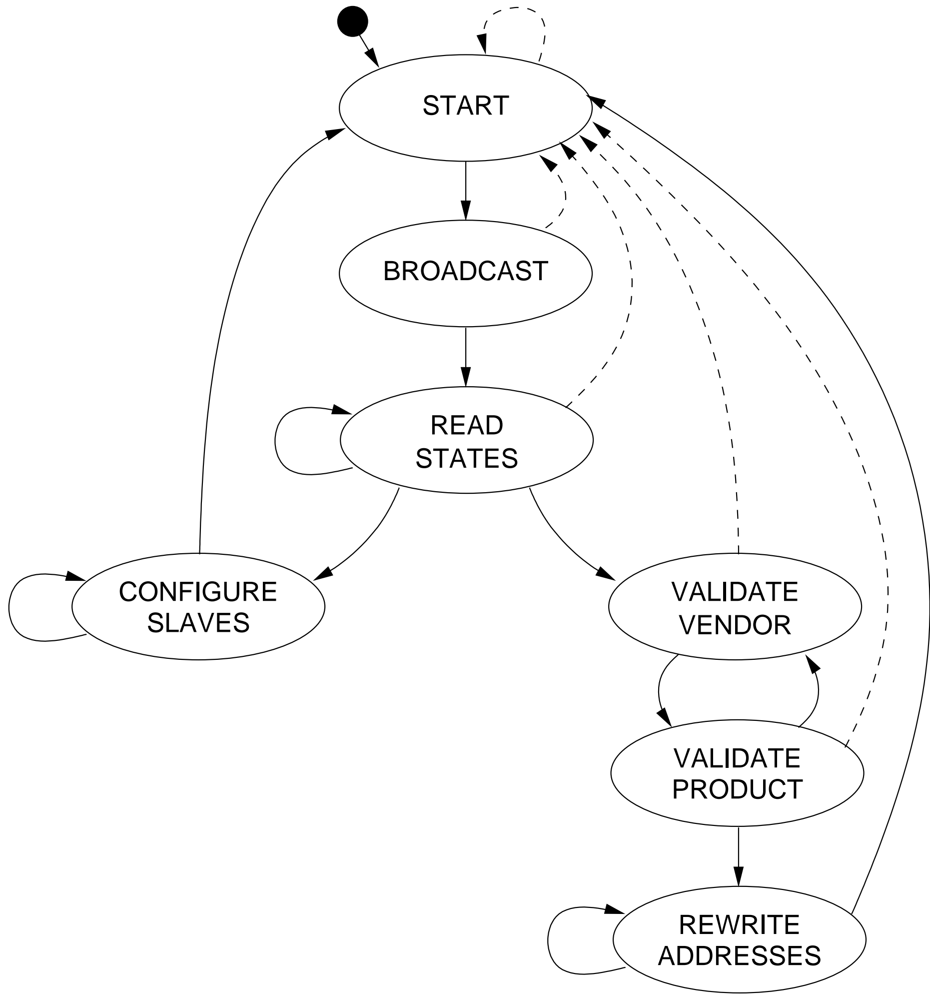

Figure 1.17: Transition diagram of the operation state machine

START This is the beginning state of the operation state machine. There is a datagram issued, that queries the “AL Control Response” attribute [3, section 5.3.2]

of all slaves via broadcast. In this way, all slave states and the number of slaves responding can be determined. → *BROADCAST*

BROADCAST The broadcast datagram is evaluated. A change in the number of responding slaves is treates as a topology change. If the number of slaves is not as expected, the bus is marked as "tainted". In this state, no slave reconfiguration is possible, because the assignment of known slaves and those present on the bus is ambiguous. If the number of slaves is considered as right, the bus is marked for validation, because it turned from tainted to normal state and it has to be checked, if all slaves are valid. Now, the state of every single slave has to be determined. For that, a (unicast) datagram is issued, that queries the first slave's "AL Control Response" attribute. → *READ STATES*

READ STATES If the current slave did not respond to its configured station address, it is marked as offline, and the next slave is queried. → *READ STATES*

If the slave responded, it is marked as online and its current state is stored. The next slave is queried. → *READ STATES*

If all slaves have been queried, and the bus is marked for validation, the validation is started by checking the first slaves vendor ID. → *VALIDATE VENDOR*

If no validation has to be done, it is checked, if all slaves are in the state they are supposed to be. If not, the first of slave with the wrong state is reconfigured and brought in the required state. → *CONFIGURE SLAVES*

If all slaves are in the correct state, the state machine is restarted. → *START*

CONFIGURE SLAVES The slave configuration state machine is executed until termination. → *CONFIGURE SLAVES*

If there are still slaves in the wrong state after another check, the first of these slaves is configured and brought into the correct state again. → *CONFIGURE SLAVES*

If all slaves are in the correct state, the state machine is restarted. → *START*

VALIDATE VENDOR The SII state machine is executed until termination. If the slave has the wrong vendor ID, the state machine is restarted. → *START*

If the slave has the correct vendor ID, its product ID is queried. → *VALIDATE PRODUCT*

VALIDATE PRODUCT The SII state machine is executed until termination. If the slave has the wrong product ID, the state machine is restarted. → *START*

If the slave has the correct product ID, the next slave's vendor ID is queried. → *VALIDATE VENDOR*

If all slaves have the correct vendor IDs and product codes, the configured station addresses can be safely rewritten. This is done for the first slave marked as offline. → *REWRITE ADDRESSES*

REWRITE ADDRESSES If the station address was successfully written, it is searched for the next slave marked as offline. If there is one, its address is reconfigured, too. → *REWRITE ADDRESSES*

If there are no more slaves marked as offline, the state machine is restarted. → *START*

1.6.4 The Idle State Machine

The Idle state machine is executed by a kernel workqueue, if no realtime module is connected. Its purpose is to make slave information available to user space, operate EoE-capable slaves, read and write E^2PROM contents and test slave functionality. Figure 1.18 shows its transition diagram.

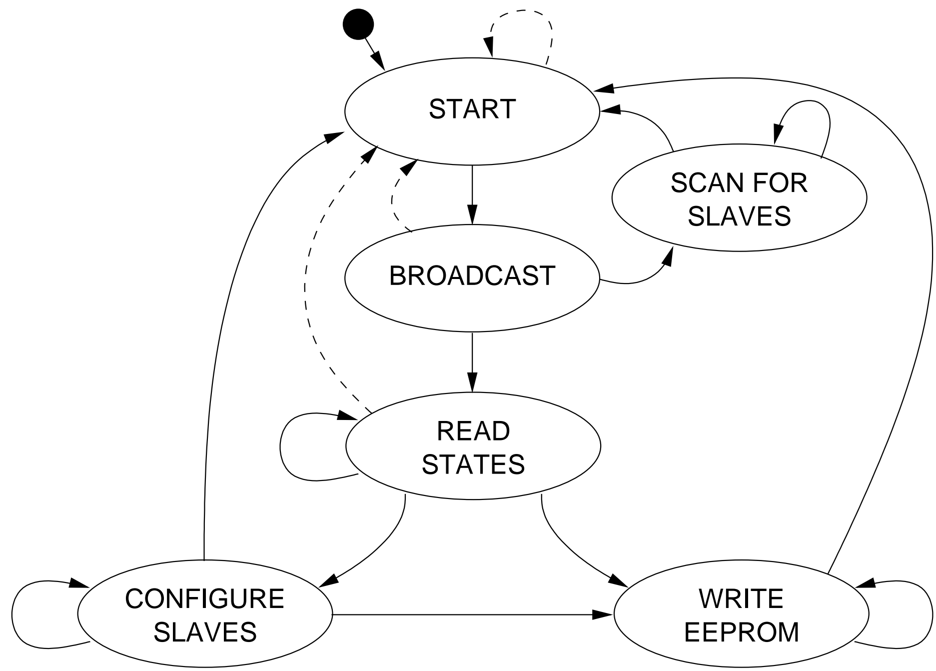

Figure 1.18: Transition diagram of the idle state machine

START The beginning state of the idle state machine. Similar to the operation state machine, a broadcast datagram is issued, to query all slave states and the number of slaves. → *BROADCAST*

BROADCAST The number of responding slaves is evaluated. If it has changed since the last time, this is treated as a topology change and the internal list of slaves is cleared and rebuild completely. The slave scan state machine is started for the first slave. → *SCAN FOR SLAVES*

If no topology change happened, every single slave state is fetched. → *READ STATES*

SCAN FOR SLAVES The slave scan state machine is executed until termination. → *SCAN FOR SLAVES*

If there is another slave to scan, the slave scan state machine is started again. → *SCAN FOR SLAVES*

If all slave information has been fetched, slave addresses are calculated and EoE processing is started. Then, the state machine is restarted. → *START*

READ STATES If the slave did not respond to the query, it is marked as offline. The next slave is queried. → *READ STATES*

If the slave responded, it is marked as online. And the next slave is queried. → *READ STATES*

If all slave states have been determined, it is checked, if any slaves are not in the state they supposed to be. If this is true, the slave configuration state machine is started for the first of them. → *CONFIGURE SLAVES*

If all slaves are in the correct state, it is checked, if any E^2PROM write operations are pending. If this is true, the first pending operation is executed by starting the SII state machine for writing access. → *WRITE EEPROM*

If all these conditions are false, there is nothing to do and the state machine is restarted. → *START*

CONFIGURE SLAVES The slave configuration state machine is executed until termination. → *CONFIGURE SLAVES*

After this, it is checked, if another slave needs a state change. If this is true, the slave state change state machine is started for this slave. → *CONFIGURE SLAVES*

If all slaves are in the correct state, it is determined, if any E^2PROM write operations are pending. If this is true, the first pending operation is executed by starting the SII state machine for writing access. → *WRITE EEPROM*

If all prior conditions are false, the state machine is restarted. → *START*

WRITE EEPROM The SII state machine is executed until termination. → *WRITE EEPROM*

If the current word has been written successfully, and there are still word to write, the SII state machine is started for the next word. → *WRITE EEPROM*

If all words have been written successfully, the new E^2PROM contents are evaluated and the state machine is restarted. → *START*

1.6.5 The Slave Scan State Machine

The slave scan state machine, which can be seen in figure 1.19, leads through the process of fetching all slave information.

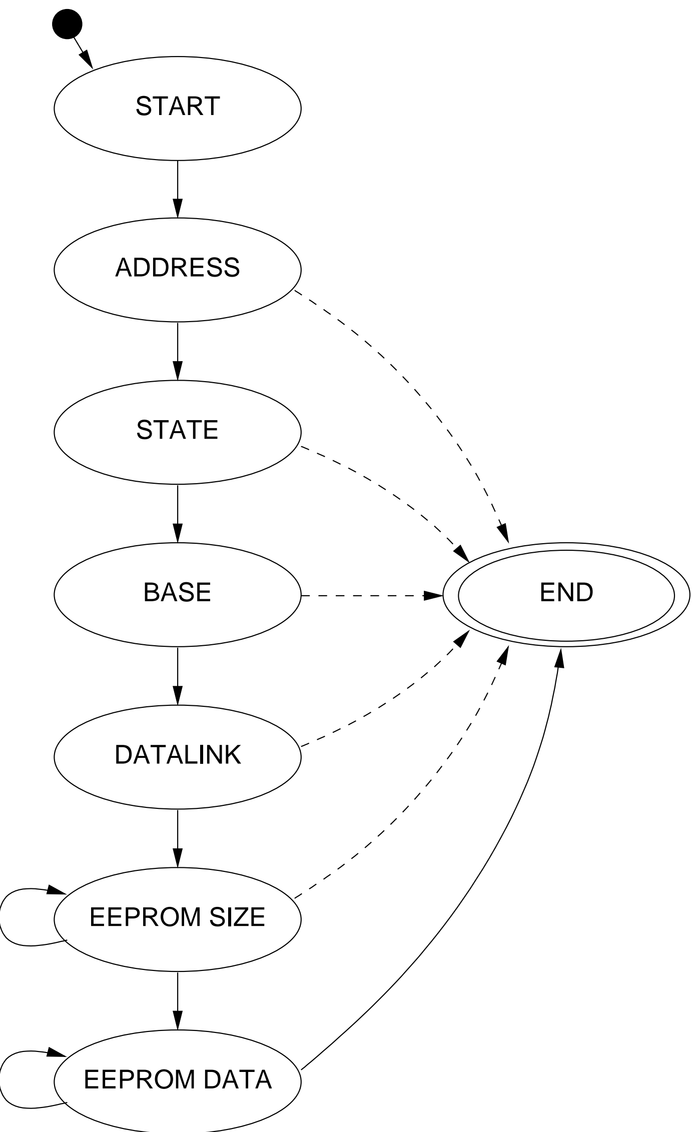

Figure 1.19: Transition diagram of the slave scan state machine

START In the beginning state of the slave scan state machine, the station address is written to the slave, which is always the ring position + 1. In this way, the address 0x0000 (default address) is not used, which makes it easy to detect unconfigured slaves. → *ADDRESS*

ADDRESS The writing of the station address is verified. After that, the slave's "AL Control Response" attribute is queried. → *STATE*

STATE The AL state is evaluated. A warning is output, if the slave has still the *Change* bit set. After that, the slave's "DL Information" attribute is queried. → *BASE*

BASE The queried base data are evaluated: Slave type, revision and build number, and even more important, the number of supported sync managers and FMMUs are stored. After that, the slave's data link layer information is read from the "DL Status" attribute at address 0x0110. → *DATALINK*

DATALINK In this state, the DL information is evaluated: This information about the communication ports contains, if the link is up, if the loop has been closed and if there is a carrier detected on the RX side of each port.

Then, the state machine starts measuring the size of the slave's E^2PROM contents. This is done by subsequently reading out each category header, until the last category is reached (type 0xFFFF). This procedure is started by querying the first category header at word address 0x0040 via the SII state machine. → *EEPROM SIZE*

EEPROM SIZE The SII state machine is executed until termination. → *EEPROM SIZE*

If the category type does not mark the end of the categories, the position of the next category header is determined via the length of the current category, and the SII state machine is started again. → *EEPROM SIZE*

If the size of the E^2PROM contents has been determined, memory is allocated, to read all the contents. The SII state machine is started to read the first word. → *EEPROM DATA*

EEPROM DATA The SII state machine is executed until termination. → *EEPROM DATA*

Two words have been read. If more than one word is needed, the two words are written in the allocated memory. Otherwise only one word (the last word) is copied. If more words are to read, the SII state machine is started again to read the next two words. → *EEPROM DATA*

The complete E^2PROM contents have been read. The slave's identity object and mailbox information are evaluated. Moreover the category types STRINGS, GENERAL, SYNC and PDO are evaluated. The slave scanning has been completed. → *END*

END Slave scanning has been finished.

1.6.6 The Slave Configuration State Machine

The slave configuration state machine, which can be seen in figure 1.20, leads through the process of configuring a slave and bringing it to a certain state.

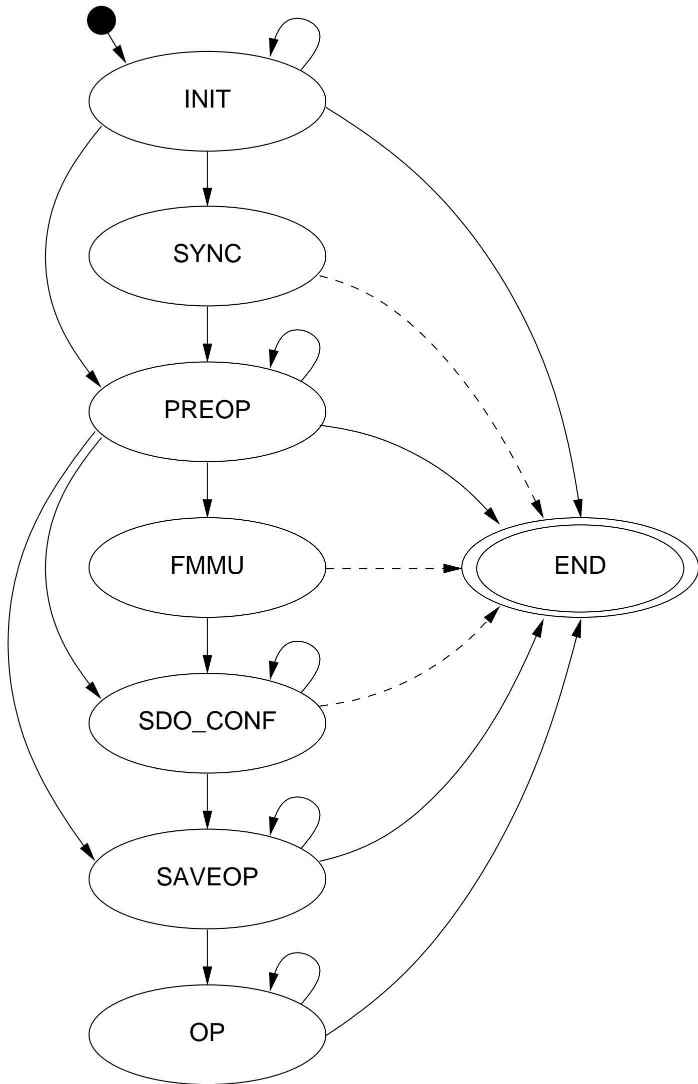

Figure 1.20: Transition diagram of the slave configuration state machine

INIT The state change state machine has been initialized to bring the slave into the *INIT* state. Now, the slave state change state machine is executed until termination. → *INIT*

If the slave state change failed, the configuration has to be aborted. → *END*

The slave state change succeeded and the slave is now in *INIT* state. If this is the target state, the configuration is finished. → *END*

If the slave does not support any sync managers, the sync manager configuration can be skipped. The state change state machine is started to bring the slave into *PREOP* state. → *PREOP*

Sync managers are configured conforming to the sync manager category information provided in the slave's E^2PROM. The corresponding datagram is issued. → *SYNC*

SYNC If the sync manager configuration datagram is accepted, the sync manager configuration was successful. The slave may now enter the *PREOP* state, and the state change state machine is started. → *PREOP*

PREOP The state change state machine is executed until termination. → *PREOP*

If the state change failed, the configuration has to be aborted. → *END*

If the *PREOP* state was the target state, the configuration is finished. → *END*

If the slave supports no FMMUs, the FMMU configuration can be skipped. If the slave has SDOs to configure, it is begun with sending the first SDO. → *SDO_CONF*

If no SDO configurations are provided, the slave can now directly be brought into the *SAVEOP* state and the state change state machine is started again. → *SAVEOP*

Otherwise, all supported FMMUs are configured according to the PDOs requested via the master's realtime interface. The appropriate datagram is issued. → *FMMU*

FMMU The FMMU configuration datagram was accepted. If the slave has SDOs to configure, it is begun with sending the first SDO. → *SDO_CONF*

Otherwise, the slave can now be brought into the *SAVEOP* state. The state change state machine is started. → *SAVEOP*

SDO_CONF The CoE state machine is executed until termination. → *SDO_CONF*

If another SDO has to be configured, a new SDO download sequence is begun. → *SDO_CONF*

Otherwise, the slave can now be brought into the *SAVEOP* state. The state change state machine is started. → *SAVEOP*

SAVEOP The state change state machine is executed until termination. → *SAVEOP*

If the state change failed, the configuration has to be aborted. → *END*

If the *SAVEOP* state was the target state, the configuration is finished. → *END*

The slave can now directly be brought into the *OP* state and the state change state machine is started a last time. → *OP*

OP The state change state machine is executed until termination. → *OP*

If the state change state machine terminates, the slave configuration is finished, regardless of its success. → *END*

END The termination state.

1.6.7 The State Change State Machine

The state change state machine, which can be seen in figure 1.21, leads through the process of changing a slave's state. This implements the states and transitions described in [3, section 6.4.1].

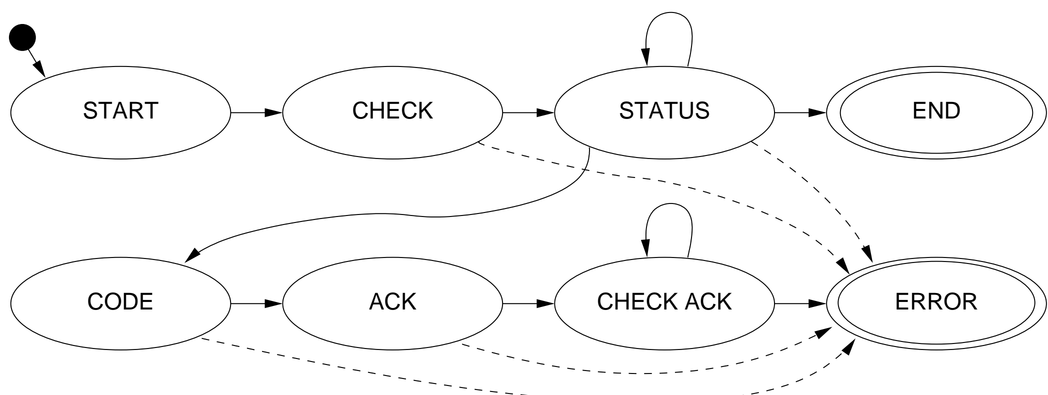

Figure 1.21: Transition diagram of the state change state machine

START The beginning state, where a datagram with the state change command is written to the slave's "AL Control Request" attribute. Nothing can fail. → *CHECK*

CHECK After the state change datagram has been sent, the "AL Control Response" attribute is queried with a second datagram. → *STATUS*

STATUS The read memory contents are evaluated: While the parameter *State* still contains the old slave state, the slave is busy with reacting on the state change command. In this case, the attribute has to be queried again. → *STATUS*

In case of success, the *State* parameter contains the new state and the *Change* bit is cleared. The slave is in the requested state. → *END*

If the slave can not process the state change, the *Change* bit is set: Now the master tries to get the reason for this by querying the *AL Status Code* parameter. → *CODE*

END If the state machine ends in this state, the slaves's state change has been successful.

CODE The status code query has been sent. Reading the *AL Status Code* might fail, because not all slaves support this parameter. Anyway, the master has to acknowledge the state change error by writing the current slave state to the "AL Control Request" attribute with the *Acknowledge* bit set. → *ACK*

ACK After that, the "AL Control Response" attribute is queried for the state of the acknowledgement. → *CHECK ACK*

CHECK ACK If the acknowledgement has been accepted by the slave, the old state is kept. Still, the state change was unsuccessful. → *ERROR*

If the acknowledgement is ignored by the slave, a timeout happens. In any case, the overall state change was unsuccessful. → *ERROR*

If there is still now response from the slave, but the timer did not run out yet, the slave's "AL Control Response" attribute is queried again. → *CHECK ACK*

ERROR If the state machine ends in this state, the slave's state change was unsuccessful.

1.6.8 The SII State Machine

The SII state machine (shown in figure 1.22) implements the process of reading or writing E^2PROM data via the Slave Information Interface described in [3, section 5.4].

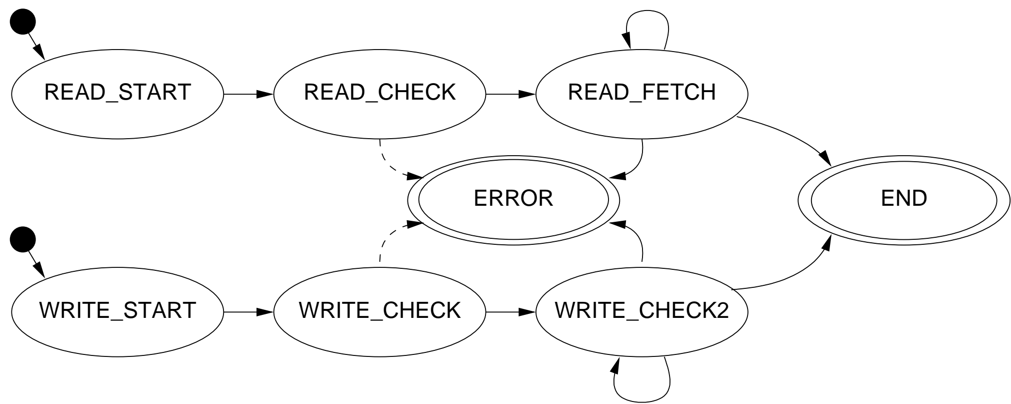

Figure 1.22: Transition diagram of the SII state machine

READ_START The beginning state for reading access, where the read request and the requested address are written to the SII attribute. Nothing can fail up to now. → *READ_CHECK*

READ_CHECK When the SII read request has been sent successfully, a timer is started. A check/fetch datagram is issued, that reads out the SII attribute for state and data. → *READ_FETCH*

READ_FETCH Upon reception of the check/fetch datagram, the *Read Operation* and *Busy* parameters are checked:

- If the slave is still busy with fetching E^2PROM data into the interface, the timer is checked. If it timed out, the reading is aborted (→ *ERROR*), if not, the check/fetch datagram is issued again. → *READ_FETCH*
- If the slave is ready with reading data, these are copied from the datagram and the read cycle is completed. → *END*

The write access states behave nearly the same:

WRITE_START The beginning state for writing access, respectively. A write request, the target address and the data word are written to the SII attribute. Nothing can fail. → *WRITE_CHECK*

WRITE_CHECK When the SII write request has been sent successfully, the timer is started. A check datagram is issued, that reads out the SII attribute for the state of the write operation. → *WRITE_CHECK2*

WRITE_CHECK2 Upon reception of the check datagram, the *Write Operation* and *Busy* parameters are checked:

- If the slave is still busy with writing E^2PROM data, the timer is checked. If it timed out, the operation is aborted (→ *ERROR*), if not, the check datagram is issued again. → *WRITE_CHECK2*
- If the slave is ready with writing data, the write cycle is completed. → *END*

1.7 Mailbox Protocol Implementations

The EtherCAT master implements the EoE and the CoE mailbox protocols. See the below section for details.

1.7.1 Ethernet-over-EtherCAT (EoE)

The EtherCAT master implements the Ethernet-over-EtherCAT mailbox protocol 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 *eoeX*, where X is a number provided by the kernel on interface registration. 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 encapsuled in the *ec_eoe_t* class (see section 1.5.1.7). So the master does not create the network interfaces directly: This is done inside the constructor of the *ec_eoe_t* class. An object of this class is called "EoE handler" below. An EoE handler additionaly 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 transmittion 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()*.

Static Handler Creation The master creates a pool of EoE handlers at startup, that are coupled to EoE-capable slaves on demand. The lifetime of the corresponding network interfaces is equal to the lifetime of the master module. This approach is opposed to creating the virtual network interfaces on demand (i. e. on running across a new EoE-capable slave). The latter approach was considered as difficult, because of serveral reasons:

- The *alloc_netdev()* function can sleep and must be called from a non-interrupt context. This reduces the flexibility of choosing an appropriate method for cyclic EoE processing.
- Unregistering network interfaces requires them to be "down", which can not be guaranteed upon sudden disappearing of an EoE-capable slave.

- The connection to the EoE-capable slaves must be as continuous as possible. Especially the transition from idle to operation mode (and vice versa) causes the rebuilding of the internal data structures. These transitions must be as transparent as possible for the instances using the network interfaces.

Number of Handlers The master module has a parameter *ec_eoeif_count* to specify the number of EoE interfaces (and handlers) per master to create. This parameter can either be specified when manually loading the master module, or (when using the init script) by setting the *$EOE_INTERFACES* variable in the sysconfig file (see section 1.8.3.2). Upon loading of the master module, the virtual interfaces become available:

```
host# ifconfig -a
eoe0      Link encap:Ethernet  HWaddr 00:11:22:33:44:06
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

eoe1      Link encap:Ethernet  HWaddr 00:11:22:33:44:07
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
...
```

Coupling of EoE Slaves During execution of the slave scan state machine (see section 1.6.5), the master determines the supported mailbox protocols. 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. After slave scanning, the master runs through all slaves again and couples each EoE-capable slave to a free EoE handler. It can happen, that there are not enough EoE handlers to cover all EoE-capable slaves. In this case, the number of EoE handlers must be increased accordingly.

EoE State Machine Every EoE handler owns an EoE state machine, that is used to send frames to the coupled slave and receive frames from the it via the EoE communication primitives. This state machine is showed in figure 1.23.

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*

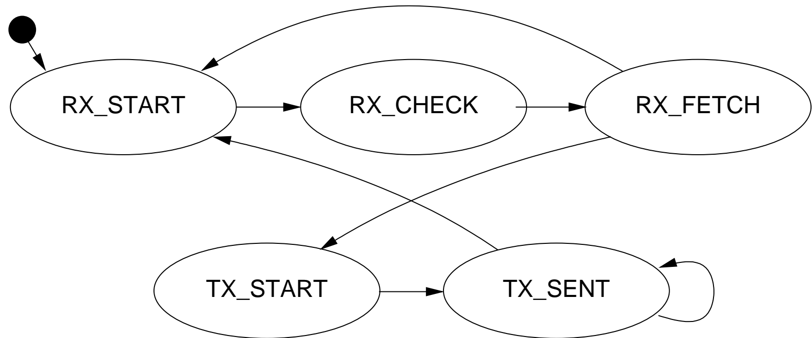

Figure 1.23: Transition diagram of the EoE state machine

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 transmittion 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*

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 thing would be to execute the EoE state machines synchronously to the operation state machine (see section 1.6.3) with every realtime cycle. This approach has the following disadvantages:

- Only one EoE fragment can 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 realtime cycles. Moreover, the data rate would be dependent on the frequency of the realtime process.

- The receiving and forwarding of frames to the kernel requires the dynamic allocation of frames. Some realtime extensions do not support calling memory allocation functions in realtime context, so the EoE state machine may not be executed with each realtime cycle.

To overcome these problems, 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 1.5.4. Section 2.4 gives practical implementation examples.

Idle Mode EoE data must also be exchanged idle mode, to guarantee the continuous availability of the connection to the EoE-capable slaves. Although there is no realtime module connected in this case, the master is still accessed by the idle state machine (see section 1.6.4), that is executed by the master's workqueue. With the EoE timer running in addition, there is still concurrency, that has to be protected by a lock. Therefore the master owns an internal spinlock that is used protect master access during idle mode.

Automatic Configuration By default, slaves are left in *INIT* state during idle mode. If an EoE interface is set to running state (i. e. with the *ifconfig up* command), the requested slave state of the related slave is automatically set to *OP*, whereupon the idle state machine will attempt to configure the slave and put it into operation.

1.7.2 CANopen-over-EtherCAT (CoE)

The CANopen-over-EtherCAT protocol [3, section 5.6] is used to configure slaves on application level. Each CoE-capable slave provides a list of SDOs for this reason.

SDO Configuration The SDO configurations have to be provided by the realtime module. This is done via the *ecrt_slave_conf_sdo*()* methods (see section 1.5.2.4), that are part of the realtime interface. The slave stores the SDO configurations in a linked list, but does not apply them at once.

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 *SAVEOP* state. Therefore the SDO configuration has to be part of the slave configuration state machine (see section 1.6.6): It is implemented via an SDO download state machine, that is executed just before entering the slave's *SAVEOP* 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 1.24.

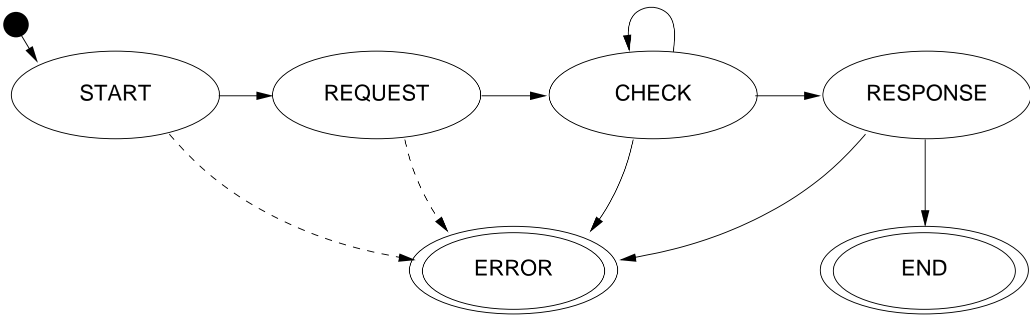

Figure 1.24: Transition diagram of the CoE download state machine

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*

END The SDO download was successful.

ERROR The SDO download was aborted due to an error.

1.8 User Space

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

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

Bus visualization is a second point: For development and debugging purposes it would be nice, if one could show the connected slaves with a single command.

Another aspect is automatic startup and configuration. If the master is to be integrated into a running system, it must be able to automatically start with a persistent configuration.

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 [6] (the former Ethereal) or others.

This section covers all those points and introduces the interfaces and tools to make all that possible.

1.8.1 The Sysfs Interface

The system filesystem (Sysfs) was introduced with Linux kernel 2.5 and is a well-defined interface for drivers to export information to user space. It serves also as an relief for the process filesystem (Procfs), where over the years much non-process information was concentrated.

Sysfs exports information about devices, classes and busses via a virtual filesystem, usually mounted to */sys*. The EtherCAT master slightly differs from this concept, because the only physical device is the network adapter it uses for bus communication, which is already represented in Sysfs. For the EtherCAT bus is no system bus like PCI (with device and driver structures), it would not make any sense to represent it as bus structure in Sysfs.

Therefore, the EtherCAT master is represented as a new directory directly unter the Sysfs root. Every master gets its own Sysfs entry named *ethercatX*, where X is the index of the master. Two masters would result in the directories */sys/ethercat0* and */sys/ethercat1*, respectively.

The Sysfs base class in the kernel code is the *kobject* structure. Each object structure, that is to be represented in Sysfs, has to contain such a structure, because due to the concurrent access (through “normal” kernel code and Sysfs code) the object deletion gets a little more complicated: The object may not be freed until no instance uses it any more. Therefore, each kobject maintains a reference counter. If the reference counter gets zero, the object is finally freed. A kobject can be registered to appear as a directory in Sysfs with a call to *kobject_add()*. Each kobject type can define attributes, that appear as files in the kobject's directory. Callback functions have to be provided for reading (and perhaps writing) access.

1.8.1.1 Master Attributes

Below is a typical listing of the masters Sysfs directory (that is a file system representation of the master's kobject):

```
host> ls /sys/ethercat0
debug_level          slave000  slave003  slave006
eeprom_write_enable  slave001  slave004  slave007
info                 slave002  slave005  slave008
```

The following attributes exist in the master directory:

debug_level (read/write) This attribute contains the master's debug level, which controls, how much information is printed into syslog. The values 0 (no additional debug messages), 1 (a few additional debug messages) and 2 (all additional debug messages) are defined. Writing is done with command like

```
host# echo 1 > /sys/ethercat0/debug_level
```

and is receipted with a syslog message by the master:

```
EtherCAT: Master debug level set to 1.
```

enable_eeprom_writing (read/write) See section 1.8.1.4 for how to use this attribute.

info (read only) This attribute contains information about the master. Example contents are below:

```
host> cat /sys/ethercat0/info

Mode: IDLE
Slaves: 9

Timing (min/avg/max) [us]:
  Idle cycle: 4 / 4.38 / 34
  EoE cycle: 9 / 11.91 / 23

EoE statistics (RX/TX) [bps]:
  eoe0: 0 / 3184
```

The mode can be *ORPHANED*, *IDLE* or *OPERATION*. The other parameters are self-explanatory.

1.8.1.2 Domain Attributes

In operation mode, each created domain is represented as a directory *domainX*, where X is the domain index. Below is a listing of the domain directory contents:

```
host> ls /sys/ethercat0/domain0
image_size
```

The domain directories currently only export the domain's image size. It is planned to export the whole process data mapping for debugging purposes.

1.8.1.3 Slave Attributes

Each slave on the bus is represented in its own directory *slaveXXX*, where XXX is the slave's 3-digit ring position in the EtherCAT bus. Below is a listing of a slave directory:

```
host> ls /sys/ethercat0/slave003
eeprom  info  state
```

eeprom (read/write) See section 1.8.1.4 for how to use this attribute.

info (read only) This attribute contains a bunch of information about the slave. Below is an example output:

```
host> cat /sys/ethercat0/slave003/info

Name: EL4132 2K. Ana. Ausgang +/-10V
Vendor ID: 0x00000002
Product code: 0x10243052

State: INIT
Ring position: 3
Advanced position: 1:3

Data link status:
  Port 0 (EBUS) Link down, Loop open, Signal detected
  Port 1 (EBUS) Link down, Loop open, Signal detected
  Port 2 (EBUS) Link down, Loop closed, No signal
  Port 3 (EBUS) Link down, Loop closed, No signal

Mailboxes:
  RX mailbox: 0x1800/246, TX mailbox: 0x18F6/246
  Supported protocols: CoE, FoE

SII data:
  Group: AnaOut
  Image: TERM_AO
  Order#: EL4132

Sync-Managers:
```

```
  0: 0x1800, length 246, control 0x26, enable
  1: 0x18F6, length 246, control 0x22, enable
  2: 0x1000, length 0, control 0x24, enable
  3: 0x1100, length 0, control 0x20, enable

PDOs:
  RXPDO "Channel 1" (0x1600), Sync-Manager 2
    "Output" 0x6411:1, 16 bit
  RXPDO "Channel 2" (0x1601), Sync-Manager 2
    "Output" 0x6411:2, 16 bit
```

This is nearly all of the SII category information needed to configure the slave, supplemented with state and addressing information.

state (read/write) This attribute contains the slave's state. It can be read or written:

```
host# cat /sys/ethercat0/slave003/state
OP
host# echo SAVEOP > /sys/ethercat0/slave003/state
```

This command should also be receipted with a syslog message:

```
EtherCAT: Accepted new state SAVEOP for slave 3.
EtherCAT: Changing state of slave 3 from OP to SAVEOP.
EtherCAT: Slave states: INIT, SAVEOP, OP.
```

After the new requested state was accepted from user space, the operation state machine (see section 1.6.3) or the idle state machine (section 1.6.4) notices, that the requested slave state differs from the current one, and therefore executes the slave configuration state machine, until the slave has reached the requested state.

1.8.1.4 E^{2}PROM Access

It is possible to directly read or write the complete E^{2}PROM contents of the slaves. This was introduced for the reasons below:

- The format of the E^{2}PROM 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 E^{2}PROM data fields have to be altered (like the alias address). A quick writing must be possible for that.
- Through read access, analyzing category data is possible from user space.

Reading out E^2PROM data is as easy as reading other attributes. Though the data are in binary format, analyzation is easier with a tool like *hexdump*:

```
host> cat /sys/ethercat0/slave003/eeprom | 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 E^2PROM contents gets as easy as copying a file:

```
host> cp /sys/ethercat0/slave003/eeprom slave003.eep
```

Writing access is only possible as *root*. Moreover writing has to be explicitly enabled and is only allowed in idle mode. This is a safety measure, because without the correct memory contents, a slave is unusable. Writing E^2PROM contents in operation mode is not provided yet.

E^2PROM writing is enabled with the command below:

```
host# echo 1 > /sys/ethercat0/eeprom_write_enable
```

The success can be seen in the syslog messages again:

```
EtherCAT: Slave EEPROM writing enabled.
```

Now, it is possible to write E^2PROM contents to a slave. The master will accept data through the *eeprom* file and will perform a short validation of the contents, before starting the write operation. This validation checks the complete size and the category headers.

```
host# cat slave003.eep > /sys/ethercat0/slave003/eeprom
```

The write operation can take a few seconds.

```
EtherCAT: EEPROM writing scheduled for slave 3, 88 words.
EtherCAT: Writing EEPROM of slave 3...
EtherCAT: Finished writing EEPROM of slave 3.
```

1.8.2 User Space Tools

There is a user space tool called *lsec* ("List EtherCAT") to visualize the EtherCAT bus. Running it usually results in an output like this:

```
host> lsec
EtherCAT bus listing for master 0:
   0  1:0  OP      EK1100 Ethernet Kopplerklemme (2A E-Bus)
   1  1:1  INIT    EL4132 2K. Ana. Ausgang +/-10V
```

```
2  1:2  INIT    EL4132 2K. Ana. Ausgang +/-10V
3  1:3  SAVEOP  EL4132 2K. Ana. Ausgang +/-10V
4  1:4  INIT    EL5101 Incremental Encoder Interface
5  1:5  INIT    EL1014 4K. Dig. Eingang 24V, 10s
6  1:6  OP      EL6601 1 Port Switch (Ethernet, CoE)
7  1:7  INIT    EL5101 Incremental Encoder Interface
8  1:8  INIT    EL5001 1K. SSI Encoder
```

Every slave is displayed as one text row. The first column shows its ring position, the second displays the "advanced position address" (see section 1.5.3) and the third column displays the current slave state. The last column is the slave's name, as it appears in the "general" E^2PROM category.

The lsec program is a Perl script, that evaluates the Sysfs *info* attributes of the slaves (see section 1.8.1.3). This is done for master 0 by default, but the master index can be specified via command line:

```
host> lsec -h
Usage: ec_list [OPTIONS]
        -m <IDX>    Query master <IDX>.
        -h          Show this help.
```

This script has proved as useful for troubleshooting: If it displays slaves, the master is up and running, and the bus connection is present, too. It is also useful when building up a bus: It can verify the list of slaves and help to create a process data image (see chapter 2).

1.8.3 System Integration

To integrate the EtherCAT master into a running system, it has to be guaranteed, that it is started on system startup. In addition, there has to be a persistent configuration, that is also applied on startup.

1.8.3.1 The EtherCAT Init Script

The EtherCAT master provides an "init script", that conforms to the requirements of the "Linux Standard Base" (LSB, [5]). The script is installed to */etc/init.d/ethercat*, so that the master can be easily inserted as a service. The different Linux distributions offer different ways to mark the service for starting and stopping in certain runlevels (for example, SUSE provides the *insserv* command).

To provide service dependencies (i. e. which services have to be started before) right 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:

```
### BEGIN INIT INFO
# Provides:          ethercat
# Required-Start:    $local_fs $syslog $network
# Should-Start:      $time
# Required-Stop:     $local_fs $syslog $network
# Should-Stop:       $time
# Default-Start:     3 5
# Default-Stop:      0 1 2 6
# Short-Description: EtherCAT master modules
# Description:
### END INIT INFO
```

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*. Besides, a link to the script is placed at */usr/sbin/rcethercat* for easier access.

```
host# /etc/init.d/ethercat restart
Shutting down EtherCAT master                    done
Starting EtherCAT master                         done
```

1.8.3.2 The EtherCAT Sysconfig File

For persistent configuration, the init script uses a sysconfig file installed to */etc/sysconfig/ethercat*, that is mandatory for the init script. The sysconfig file contains all configuration variables needed to operate a master:

DEVICE_INDEX This variable must contain the PCI index of the EtherCAT device. Setting this is mandatory for the EtherCAT init script. Default: −1

EOE_INTERFACES The number of virtual Ethernet-over-EtherCAT interfaces, every master creates on startup. See section 1.7.1. Default: 0

EOE_BRIDGE If this variable is set, all EoE interfaces will be added to a network bridge according to IEEE 802.1D after master startup. The variable must contain the name of the bridge. To use this functionality, the kernel must be configured with the *CONFIG_BRIDGE* option and the *bridge-utils* package must be installed (i. e. the *brctl* command is needed).

EOE_IP_ADDRESS The IP address of the EoE bridge. Setting this together with *EOE_IP_NETMASK* will let the local host communicate with devices on the EoE bridge.

EOE_IP_NETMASK IP netmask of the EoE bridge.

EOE_EXTRA_INTERFACES The list of extra interfaces to include in the EoE bridge. Set this to interconnect the EoE bridge with other local interfaces. If

EOE_BRIDGE is empty or undefined, setting this variable has no effect. Important: The IP address of the listed interfaces will be cleared. Setting *EOE_IP_ADDRESS* and *EOE_IP_NETMASK* will re-enable them for IP traffic.

EOE_GATEWAY The IP address of the default gateway. If this variable is set, the gateway will be renewed after bridge installation. This is necessary, if the default gateway's interface is one of the *EOE_EXTRA_INTERFACES*.

1.8.4 Monitoring and Debugging

For debugging purposes, every EtherCAT master registeres a read-only network interface *ecX*, where X is a number, provided by the kernel on device registration. While it is "up", the master forwards every frame sent and received to this interface.

This makes it possible to connect an network monitor (like Wireshark or tcpdump) to the debug interface and monitor the EtherCAT frames.

It has to be considered, that can be frame rate can be very high. The idle state machine usually runs every kernel timer interrupt (up to 1 kHz) and with a connected realtime module, the rate can be even higher.

Attention: The socket buffers needed for the operation of the debugging interface have to be allocated dynamically. Some Linux realtime extensions do not allow this in realtime context!

1.9 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.

1.9.1 Realtime Interface Profiling

One of the most important timing aspects are the runtimes of the realtime interface functions, that are called in cyclic context. These functions make up an important part of the overall timing of the realtime module. 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 microseconds with help of the *cpu_khz* variable, that contains the number of increments per millisecond.

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 cycle time of 100 µs. The measuring was repeated $n = 100$ times and the results were averaged. These can be seen in table 1.2.

Table 1.2: Profiling of a Realtime Cycle on a 2.0 GHz Processor

Element	Mean Duration [µs]	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 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 1.9.2.

1.9.2 Bus Cycle Measuring

For measuring the time, a frame is "on the wire", two timestamps must be 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 part of the realtime module. 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[8]. In this case, the process data keeps being the 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.

[8] To limit syslog output, a mechanism has been implementet, that outputs a summarized notification at maximum once a second.

2 Using the EtherCAT Master

This chapter will give practical examples of how to use the EtherCAT master via the realtime interface by writing a realtime module.

Section 2.1 shows how to compile and install the master, while the sections 2.2 to 2.4 give examples for different realtime modules.

2.1 Compiling and Installing

The current EtherCAT master code is available at [1]. After downloading the *tar.bz2* file, it has to be unpacked with the command below (or similar):

```
host> tar xjf ethercat-stable-1.1-r513-src.tar.bz2
host> cd ethercat-stable-1.1-r513-src
```

For compilation, the kernel sources are needed. Basically any kernel sources are appropriate[1], that are configured with networking. If the host kernel is not the running kernel, a copy of the configuration template has to be made:

```
host> cp ethercat.conf.tmpl ethercat.conf
```

Now the *$KERNEL* variable inside the *ethercat.conf* file can be adjusted to reflect the appropriate kernel version. If everything is correct now, the sucessive call to

```
host> make
```

will result in no errors.

The following commands have to be entered as *root*: To install the kernel modules, the init script, the sysconfig file and the user space tools, the below command has to be executed:

```
host# make install
```

If the sysconfig file did not exist yet, the user is notified to edit it, before running the master. For the contents of the file, see section 1.8.3.2. To give a short summary: The most important thing is to adjust the *$DEVICE_INDEX* variable. It has to be set to the index of the compatible network device to use with EtherCAT, where the

[1] If a realtime extension is to be used, the kernel has to be patched before that.

order of devices is dependent on their position in the PCI bus. If this is not known, the index can be determinded with trial and error, but it has to be considered that a wrong value can cause a loss of network connection.

After the basic configuration is done, the master can be started with the below command:

```
host# /etc/init.d/ethercat start
```

The operation of the master can be observed by looking at the syslog messages, which should look like the ones below:

```
1  EtherCAT: Master driver, 1.1 (stable) - rev. 513,
2      compiled by fp at Aug 09 2006 10:23:20
3  EtherCAT: Initializing 1 EtherCAT master(s)...
4  EtherCAT: Initializing master 0.
5  EtherCAT: Master driver initialized.
6  ec_8139too Fast Ethernet driver 0.9.27 Revision 513,
7      compiled by fp at Aug 09 2006 10:23:20
8  ec_device_index is 0
9  ACPI: PCI Interrupt 0000:01:00.0[A] -> Link [LNKC]
10     -> GSI 11 (level, low) -> IRQ 11
11 ec0: RealTek RTL8139 at 0xd047c000, 00:c0:26:00:c6:aa, IRQ 11
12 ec0:  Identified 8139 chip type 'RTL-8100B/8139D'
13 Registering EtherCAT device...
14 Starting EtherCAT device...
15 EtherCAT: Link state changed to UP.
16 EtherCAT: Starting Idle mode.
17 EtherCAT: 11 slaves responding.
18 EtherCAT: Slave states: INIT, OP.
19 EtherCAT: Scanning bus.
20 EtherCAT: Bus scanning completed.
21 EtherCAT: No EoE handlers coupled.
```

(1) The master module is loaded, and one master is initialized.

(6) The EtherCAT-capable RTL8139 device driver is loaded. It connects its first network device to the master.

(16) The master starts idle mode and begins scanning the bus for slaves.

2.2 A Minimal Example Module

This section will explain the usage of the EtherCAT master from a minimal kernel module. The complete module code is obtainable as a part of the EtherCAT master code release (see [1], file *examples/mini/mini.c*).

The minimal example uses a kernel timer (software interrupt) to handle cyclic code. After the timer function is executed, it re-adds itself with a delay of one *jiffy*, which results in a timer frequency of *HZ*

The module-global variables, needed to operate the master can be seen in listing 2.1.

Listing 2.1: Minimal variables

```
1  struct timer_list timer;
2
3  ec_master_t *master = NULL;
4  ec_domain_t *domain1 = NULL;
5
6  void *r_dig_in, *r_ana_out;
7
8  ec_pdo_reg_t domain1_pdos[] = {
9          {"1", Beckhoff_EL1014_Inputs, &r_dig_in},
10         {"2", Beckhoff_EL4132_Ouput1, &r_ana_out},
11         {}
12 };
```

(1) There is a timer object declared, that is needed to tell the kernel to install a timer and execute a certain function, if it runs out. This is done by a variable of the *timer_list* structure.

(3) – (4) There is a pointer declared, that will later point to a requested EtherCAT master. Additionally there is a pointer to a domain object needed, that will manage process data IO.

(6) The pointers *r_** will later point to the raw process data values inside the domain memory. The addresses they point to will be set during a call to *ec_master_-activate()*, that will create the domain memory and configure the mapped process data image.

(8) – (12) The configuration of the mapping of certain PDOs in a domain can easily be done with the help of an initialization array of the *ec_pdo_reg_t* type, defined as part of the realtime interface. Each record must contain the ASCII bus-address of the slave (see section 1.5.3), the slave's vendor ID and product code, and the index and subindex of the PDO to map (these four fields can be specified in junction, by using one of the defines out of the *include/ecdb.h* header). The last field has to be the address of the process data pointer, so it can later be redirected appropriately. Attention: The initialization array must end with an empty record ({})!

The initialization of the minimal realtime module is done by the "Minimal init function" in listing 2.2.

Listing 2.2: Minimal init function

```
int __init init_mini_module(void)
{
        if (!(master = ecrt_request_master(0))) {
                goto out_return;
        }

        if (!(domain1 = ecrt_master_create_domain(master))) {
                goto out_release_master;
        }

        if (ecrt_domain_register_pdo_list(domain1,
                                          domain1_pdos)) {
                goto out_release_master;
        }

        if (ecrt_master_activate(master)) {
                goto out_release_master;
        }

        ecrt_master_prepare(master);

        init_timer(&timer);
        timer.function = run;
        timer.expires = jiffies + 10;
        add_timer(&timer);

        return 0;

      out_release_master:
        ecrt_release_master(master);
      out_return:
        return -1;
}
```

(3) It is tried to request the first EtherCAT master (index 0). On success, the *ecrt_request_master()* function returns a pointer to the reserved master, that can be used as an object to following functions calls. On failure, the function returns *NULL*.

(7) In order to exchange process data, a domain object has to be created. The *ecrt_master_create_domain()* function also returns a pointer to the created domain, or *NULL* in error case.

(11) The registration of domain PDOs with an initialization array results in a single function call. Alternatively the data fields could be registered with individual

calls of *ecrt_domain_register_pdo()*.

(16) After the configuration of process data mapping, the master can be activated for cyclic operation. This will configure all slaves and bring them into *OP* state.

(20) This call is needed to avoid a case differentiation in cyclic operation: The first operation in cyclic mode is a receive call. Due to the fact, that there is nothing to receive during the first cycle, there had to be an *if*-statement to avoid a warning. A call to *ec_master_prepare()* sends a first datagram containing a process data exchange datagram, so that the first receive call will not fail.

(22) – (25) The master is now ready for cyclic operation. The kernel timer that cyclically executes the *run()* function is initialized and started.

The coding of a cleanup function fo the minimal module can be seen in listing 2.3.

Listing 2.3: Minimal cleanup function

```
1 void __exit cleanup_mini_module(void)
2 {
3         del_timer_sync(&timer);
4         ecrt_master_deactivate(master);
5         ecrt_release_master(master);
6 }
```

(3) To cleanup the module, it it necessary to stop the cyclic processing. This is done by a call to *del_timer_sync()* which safely removes a queued timer object. It is assured, that no cyclic work will be done after this call returns.

(4) This call deactivates the master, which results in all slaves being brought to their *INIT* state again.

(5) This call releases the master, removes any existing configuration and silently starts the idle mode. The value of the master pointer is invalid after this call and the module can be safely unloaded.

The final part of the minimal module is that for the cyclic work. Its coding can be seen in listing 2.4.

Listing 2.4: Minimal cyclic function

```
1 void run(unsigned long data)
2 {
3         static uint8_t dig_in_0;
4 
5         ecrt_master_receive(master);
6         ecrt_domain_process(domain1);
```

```
7
8           dig_in_0 = EC_READ_BIT(r_dig_in, 0);
9           EC_WRITE_S16(r_ana_out, dig_in_0 * 0x3FFF);
10
11          ecrt_master_run(master);
12          ecrt_master_send(master);
13
14          timer.expires += 1; // frequency = HZ
15          add_timer(&timer);
16 }
```

(5) The cyclic processing starts with receiving datagrams, that were sent in the last cycle. The frames containing these datagrams have to be received by the network interface card prior to this call.

(6) The process data of domain 1 has been automatically copied into domain memory while datagram reception. This call checks the working counter for changes and re-queues the domain's datagram for sending.

(8) This is an example for reading out a bit-oriented process data value (i. e. bit 0) via the *EC_READ_BIT()* macro. See section 1.5.2.5 for more information about those macros.

(9) This line shows how to write a signed, 16-bit process data value. In this case, the slave is able to output voltages of −10 V to +10 V with a resolution of 16 bit. This write command outputs either 0 V or +5 V, depending of the value of *dig_in_0*.

(11) This call runs the master's operation state machine (see section 1.6.3). A single state is processed, and datagrams are queued. Mainly bus observation is done: The bus state is determined and in case of slaves that lost their configuration, reconfiguration is tried.

(12) This method sends all queued datagrams, in this case the domain's datagram and one of the master state machine. In best case, all datagrams fit into one frame.

(14) – (15) Kernel timers are implemented as "one-shot" timers, so they have to be re-added after each execution. The time of the next execution is specified in *jiffies* and will happen at the time of the next system timer interrupt. This results in the *run()* function being executed with a frequency of *HZ*.

2.3 An RTAI Example Module

The whole code can be seen in the EtherCAT master code release (see [1], file *examples/rtai/rtai_sample.c*).

Listing 2.5 shows the defines and global variables needed for a minimal RTAI module with EtherCAT processing.

Listing 2.5: RTAI task declaration

```
1 #define FREQUENCY 10000
2 #define TIMERTICKS (1000000000 / FREQUENCY)
3
4 RT_TASK task;
```

(1) – (2) RTAI takes the cycle period as nanoseconds, so the easiest way is to define a frequency and convert it to a cycle time in nanoseconds.

(4) The *task* variable later contains information about the running RTAI task.

Listing 2.6 shows the module init function for the RTAI module. Most lines are the same as in listing 2.2, differences come up when starting the cyclic code.

Listing 2.6: RTAI module init function

```
1  int __init init_mod(void)
2  {
3          RTIME requested_ticks, tick_period, now;
4
5          if (!(master = ecrt_request_master(0))) {
6                  goto out_return;
7          }
8
9          if (!(domain1 = ecrt_master_create_domain(master))) {
10                 goto out_release_master;
11         }
12
13         if (ecrt_domain_register_pdo_list(domain1,
14                                           domain1_pdos)) {
15                 goto out_release_master;
16         }
17
18         if (ecrt_master_activate(master)) {
19                 goto out_release_master;
20         }
21
22         ecrt_master_prepare(master);
23
24         requested_ticks = nano2count(TIMERTICKS);
25         tick_period = start_rt_timer(requested_ticks);
26
27         if (rt_task_init(&task, run, 0, 2000, 0, 1, NULL)) {
```

```
28                  goto out_stop_timer;
29          }
30
31          now = rt_get_time();
32          if (rt_task_make_periodic(&task, now + tick_period,
33                                    tick_period)) {
34                  goto out_stop_task;
35          }
36
37          return 0;
38
39      out_stop_task:
40          rt_task_delete(&task);
41      out_stop_timer:
42          stop_rt_timer();
43      out_deactivate:
44          ecrt_master_deactivate(master);
45      out_release_master:
46          ecrt_release_master(master);
47      out_return:
48          return -1;
49  }
```

(24) – (25) The nanoseconds are converted to RTAI timer ticks and an RTAI timer is started. *tick_period* will be the "real" number of ticks used for the timer period (which can be different to the requested one).

(27) The RTAI task is initialized by specifying the cyclic function, the parameter to hand over, the stack size, priority, a flag that tells, if the function will use floating point operations and a signal handler.

(32) The task is made periodic by specifying a start time and a period.

The cleanup function of the RTAI module in listing 2.7 is nearly as simple as that of the minimal module.

Listing 2.7: RTAI module cleanup function

```
1  void __exit cleanup_mod(void)
2  {
3          rt_task_delete(&task);
4          stop_rt_timer();
5          ecrt_master_deactivate(master);
6          ecrt_release_master(master);
7          rt_sem_delete(&master_sem);
8  }
```

(2) The RTAI task will be stopped and deleted.

(3) After that, the RTAI timer can be stopped.

The rest is the same as for the minimal module.

Worth to mention is, that the cyclic function of the RTAI module (listing 2.8) has a slightly different architecture. The function is not executed until returning for every cycle, but has an infinite loop in it, that is placed in a waiting state for the rest of each cycle.

Listing 2.8: RTAI module cyclic function

```
1  void run(long data)
2  {
3          while (1) {
4                  ecrt_master_receive(master);
5                  ecrt_domain_process(domain1);
6
7                  k_pos = EC_READ_U32(r_ssi_input);
8
9                  ecrt_master_run(master);
10                 ecrt_master_send(master);
11
12                 rt_task_wait_period();
13         }
14 }
```

(3) The *while (1)* loop executes for the lifetime of the RTAI task.

(12) The *rt_task_wait_period()* function sets the process into a sleeping state until the beginning of the next cycle. It also checks, if the cyclic function has to be terminated.

2.4 Concurrency Example

As mentioned before, there can be concurrent access to the EtherCAT master. The realtime module and a EoE process can compete for master access, for example. In this case, the module has to provide the locking mechanism, because it depends on the module's architecture which lock has to be used. The module makes this locking mechanism available to the master through the master's locking callbacks.

In case of RTAI, the lock can be an RTAI semaphore, as shown in listing 2.9. A normal linux semaphore would not be appropriate, because it could not block the RTAI task due to RTAI running in a higher domain than the linux kernel (see [9]).

Listing 2.9: RTAI semaphore for concurrent access

```
1 SEM master_sem;
```

The module has to implement the two callbacks for requesting and releasing the master lock. An exemplary coding can be seen in listing 2.10.

Listing 2.10: RTAI locking callbacks for concurrent access

```
1  int request_lock(void *data)
2  {
3          rt_sem_wait(&master_sem);
4          return 0;
5  }
6
7  void release_lock(void *data)
8  {
9          rt_sem_signal(&master_sem);
10 }
```

(1) The *request_lock()* function has a data parameter. The master always passes the value, that was specified when registering the callback function. This can be used for handing the master pointer. Notice, that it has an integer return value (see line 4).

(3) The call to *rt_sem_wait()* either returns at once, when the semaphore was free, or blocks until the semaphore is freed again. In any case, the semaphore finally is reserved for the process calling the request function.

(4) When the lock was requested successfully, the function should return 0. The module can prohibit requesting the lock by returning non-zero (see paragraph "Tuning the jitter" below).

(7) The *release_lock()* function gets the same argument passed, but has a void return value, because is always succeeds.

(9) The *rt_sem_signal()* function frees the semaphore, that was prior reserved with *rt_sem_wait()*.

In the module's init function, the semaphore must be initialized, and the callbacks must be passed to the EtherCAT master:

Listing 2.11: Module init function for concurrent access

```
1  int __init init_mod(void)
2  {
3          RTIME tick_period, requested_ticks, now;
4
```

```
5         rt_sem_init(&master_sem, 1);
6
7         if (!(master = ecrt_request_master(0))) {
8                 goto out_return;
9         }
10
11        ecrt_master_callbacks(master, request_lock,
12                              release_lock, NULL);
13        // ...
```

(5) The call to *rt_sem_init()* initializes the semaphore and sets its value to 1, meaning that only one process can reserve the semaphore without blocking.

(11) The callbacks are passed to the master with a call to *ecrt_master_callbacks()*. The last parameter is the argument, that the master should pass with each call to a callback function. Here it is not used and set to *NULL*.

For the cyclic function being only one competitor for master access, it has to request the lock like any other process. There is no need to use the callbacks (which are meant for processes of lower priority), so it can access the semaphore directly:

Listing 2.12: RTAI cyclic function for concurrent access

```
1  void run(long data)
2  {
3          while (1) {
4                  rt_sem_wait(&master_sem);
5
6                  ecrt_master_receive(master);
7                  ecrt_domain_process(domain1);
8
9                  k_pos = EC_READ_U32(r_ssi_input);
10
11                 ecrt_master_run(master);
12                 ecrt_master_send(master);
13
14                 rt_sem_signal(&master_sem);
15                 rt_task_wait_period();
16         }
17 }
```

(4) Every access to the master has to be preceeded by a call to *rt_sem_wait()*, because another instance might currently access the master.

(14) When cyclic processing finished, the semaphore has to be freed again, so that other processes have the possibility to access the master.

A little change has to be made to the cleanup function in case of concurrent master access.

Listing 2.13: RTAI module cleanup function for concurrent access

```
1 void __exit cleanup_mod(void)
2 {
3         rt_task_delete(&task);
4         stop_rt_timer();
5         ecrt_master_deactivate(master);
6         ecrt_release_master(master);
7         rt_sem_delete(&master_sem);
8 }
```

(7) Upon module cleanup, the semaphore has to be deleted, so that memory can be freed.

Tuning the Jitter Concurrent access leads to higher jitter of the realtime process, because there are situations, in which the realtime process has to wait for a process of lower priority to finish accessing the master. In most cases this is acceptable, because a master access cycle (receive/process/send) only takes 10 µs to 20 µs on recent systems, what would be the maximum additional jitter. However some applications demand a minimum jitter. For this reason the master access can be prohibited by the realtime module: If the time, another process wants to access the master, is to close to the beginning of the next realtime cycle, the module can disallow, that the lock is taken. In this case, the request callback has to return 1, meaning that the lock has not been taken. The foreign process must abort its master access and try again next time.

This measure helps to significantly reducing the jitter produced by concurrent master access. Below are exerpts of an example coding:

Listing 2.14: Variables for jitter reduction

```
1 #define FREQUENCY 10000 // RTAI task frequency in Hz
2 // ...
3 cycles_t t_last_cycle = 0;
4 const cycles_t t_critical = cpu_khz * 1000 / FREQUENCY
5                           - cpu_khz * 30 / 1000;
```

(3) The variable *t_last_cycle* holds the timer ticks at the beginning of the last realtime cycle.

(4) *t_critical* contains the number of ticks, that may have passed since the beginning of the last cycle, until there is no more foreign access possible. It is calculated by substracting the ticks for 30 µs from the ticks for a complete cycle.

Listing 2.15: Cyclic function with reduced jitter

```
1  void run(long data)
2  {
3          while (1) {
4                  t_last_cycle = get_cycles();
5                  rt_sem_wait(&master_sem);
6                  // ...
```

(4) The ticks of the beginning of the current realtime cycle are taken before reserving the semaphore.

Listing 2.16: Request callback for reduced jitter

```
1  int request_lock(void *data)
2  {
3          // too close to the next RT cycle: deny access.
4          if (get_cycles() - t_last_cycle > t_critical)
5                  return -1;
6
7          // allow access
8          rt_sem_wait(&master_sem);
9          return 0;
10 }
```

(4) If the time of request is too close to the next realtime cycle (here: < 30 µs before the estimated beginning), the locking is denied. The requesting process must abort its cycle.

Bibliography

[1] Ingenieurgemeinschaft IgH: EtherLab – Open Source Toolkit for rapid realtime code generation under Linux with Simulink/RTW and EtherCAT technology. URL: http://etherlab.org, July 31, 2006.

[2] IEC 61158-4-12: Data-link Protocol Specification. International Electrotechnical Comission (IEC), 2005.

[3] IEC 61158-6-12: Application Layer Protocol Specification. International Electrotechnical Comission (IEC), 2005.

[4] GNU General Public License, Version 2. URL: http://www.gnu.org/licenses/gpl.txt. August 9, 2006.

[5] Linux Standard Base. URL: http://www.freestandards.org/en/LSB. August 9, 2006.

[6] Wireshark. URL: http://www.wireshark.org. August 9, 2006.

[7] *Hopcroft, J. E. / Ullman, J. D.*: Introduction to Automata Theory, Languages and Computation. Adison-Wesley, Reading, Mass. 1979.

[8] *Wagner, F. / Wolstenholme, P.*: State machine misunderstandings. In: IEE journal "Computing and Control Engineering", 2004.

[9] RTAI. The RealTime Application Interface for Linux from DIAPM. URL: http://www.rtai.org, 2006.

Glossary

ADEOS Adaptive Domain Environment for Operating Systems, page 1

ASCII American Standard Code for Information Interchange, page 48

ecdev EtherCAT Device, page 12

ecrt EtherCAT Realtime Interface, page 26

FSM Finite State Machine, page 50

HZ Kernel macro containing the timer interrupt frequency, page 85

ISR Interrupt Service Routine, page 9

LSB Linux Standard Base, page 2

MII Media Independent Interface, page 20

PCI Peripheral Component Interconnect, Computer Bus, page 11

RTAI RealTime Application Interface, page 1

Sysfs System Filesystem, page 2

Index

Bus cycle, 80

CoE, 70
Concurrency, 49

Datagram
    Class, 31
Device
    Class, 29
Device interface, 12
Device modules, 3, 8
Domain, 5
    Class, 33

E^2PROM
    Access, 75
ecrt, 26
EoE, 25, 66, 91
    Class, 38
Examples
    Concurrency, 91
    Minimal, 84
    RTAI, 88

FMMU
    Configuration, 5
FSM, 50
    Class, 35
    EoE, 68
    Idle, 58
    Operation, 56
    SII, 65
    Slave Configuration, 62
    Slave Scan, 59
    State Change, 64
    Theory, 51

GPL, 2

Idle mode, 7
Init script, 77
Interrupt, 9, 10
ISR, 9, 13, 45

jiffies, 85
Jitter, 94

LSB, 77
lsec, 76

Mailbox, 66
Master
    Architecture, 3
    Class, 22
    Compilation, 83
    Features, 1
    Usage, 83
Master modes, 6
Master module, 3, 21
module_put(), 13
Monitoring, 79

net_device, 9
netif, 9
Network drivers, 8, 15

Operation mode, 7

PDO, 5
Process data, 5

Realtime
    Profiling, 79
Realtime interface, 42
Realtime module, 3

SII, 65, 68

Slave
    Addressing, 48
    Class, 26
Socket buffer, 9, 10
Sysconfig file, 78
Sysfs, 72
syslog, 13, 84

try_module_get(), 13

User space, 72
    Tools, 76