EtherCAT SDK User Manual

rt:labs

EtherCAT SDK User Manual

rt-labs AB

http://www.rt-labs.com

Revision: 497

DISCLAIMER

RT-LABS AB MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, rt-labs AB, reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of rt-labs AB, to notify any person of such changes.

December 22, 2017

Contents

Chapter 1

EtherCAT SDK Introduction

EtherCAT SDK is a complete toolset for developing and maintaining EtherCAT slaves. It includes the following components.

- **EtherCAT Slave Editor**: This component is used to specify the properties of EtherCAT slaves and to generate C source code for them.
- **EtherCAT Explorer**: This component is used to connect to running EtherCAT slaves, to do network exploration, logging and diagnostics.

Together with SOES EtherCAT Slave Stack the developer has an all-in-one tool for developing EtherCAT slaves in an efficient way, which makes it easy to support and maintain them throughout their life cycle.

EtherCAT SDK is availible in the following two different packages:

- EtherCAT SDK Eclipse plug-in
- EtherCAT SDK stand-alone application

1 Download and Installation

1.1 EtherCAT SDK Eclipse Plug-in

For EtherCAT SDK eclipse plug-in, download and install as a normal Eclipse plug-in.

Go to **Help** > **Install new Software...**, enter the rt-labs download site hosting the plug-in:

`http://download.rt-labs.com/ethercat/sdk/updates`

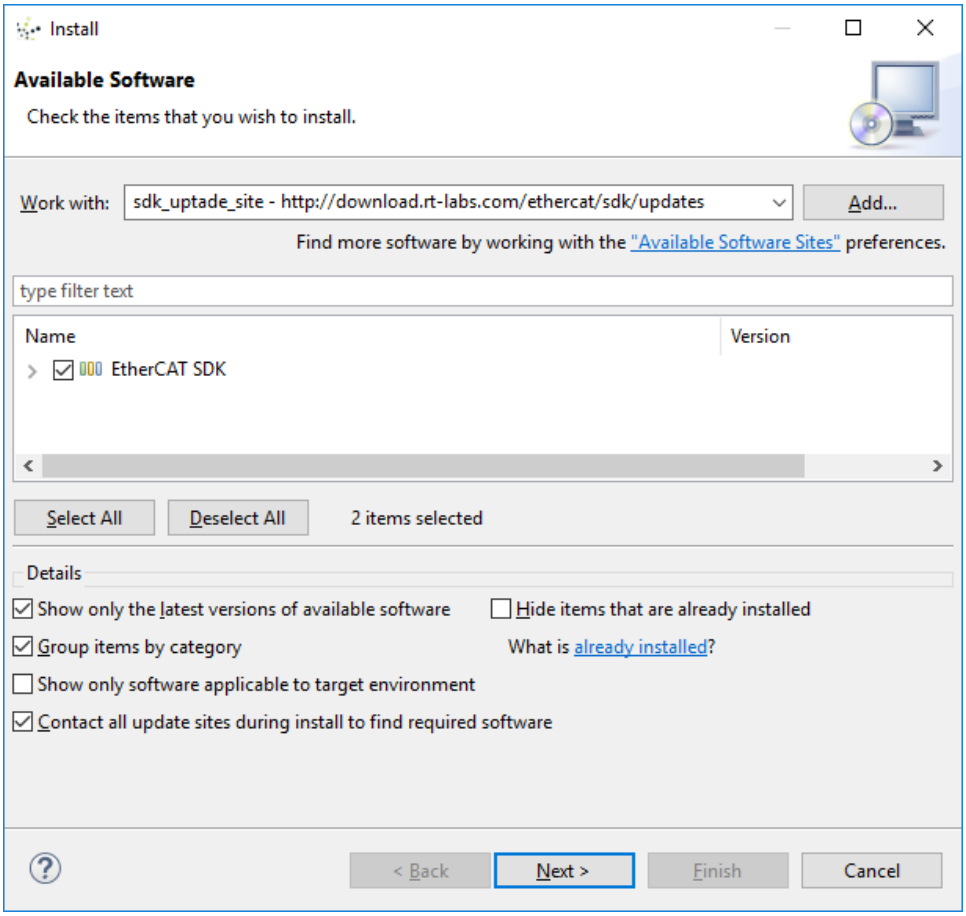

Figure 1.1: Enter Help > Install new software...

A license is required for the application to work. See section EtherCAT SDK License for instructions on how to acquire and install the license.

1.2 EtherCAT SDK Stand-Alone Application

Download the EtherCAT SDK from rt-labs download site in one of the following versions:

- `http://download.rt-labs.com/ethercat/sdk/ethercat-sdk-win32.-win32.x86.zip`
- `http://download.rt-labs.com/ethercat/sdk/ethercat-sdk-win32.-win32.x86_64.zip`
- `http://download.rt-labs.com/ethercat/sdk/ethercat-sdk-linux.gtk.-x86_64.zip`

Place the selected EtherCAT SDK zip file in the destination folder, unzip it and run the file `ethercat-sdk.exe`.

A license is required for the application to work. See section EtherCAT SDK License for instructions on how to acquire and install the license.

1.3 EtherCAT SDK License

A license is required to be able to generate slave code from the EtherCAT Slave Editor and to connect to an EtherCAT network with the EtherCAT Explorer.

To install a license perform the following steps:

Go to **Windows** > **Preferences** > **Licenses**. There are two options:

1. **Add License...**, browse and select an existing license file.
2. **Request** an evaluation license by providing name and e-mail address. An e-mail with a license will be sent to given e-mail address.

Use *Option 1* to add a license received by e-mail.

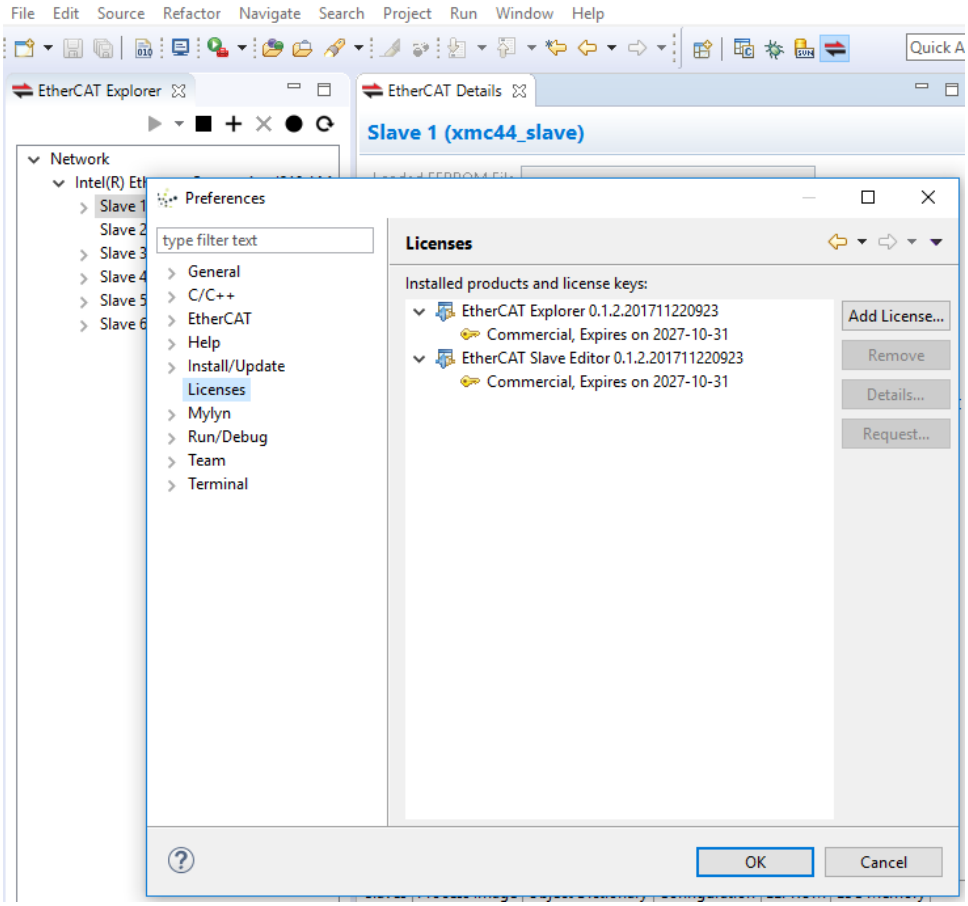

Figure 1.2: Go to Windows > Preferences > Licenses

2 Release Notes

2.1 Release 1.0.1

This release contains minor adjustments of the software.

1. Add the EtherCAT Explorer component to the stand-alone application. #186

2. Remove validation of item indices from the **Application** tab in the slave editor. #185
3. Compute the generated **Profile number** and **Add info** from **Device type** #187
4. Remove start modes that are only relevant for EAP networks. #188

2.2 Release 1.0.0

This is the first official release of the software.

Chapter 2

Slave Editor

1 Overview

The Slave Editor is an EtherCAT slave design tool supporting the developer in creating EtherCAT conformant slaves by providing all necessary outputs in an information aligned way and by providing a SOES EtherCAT slave stack application API for the application to use.

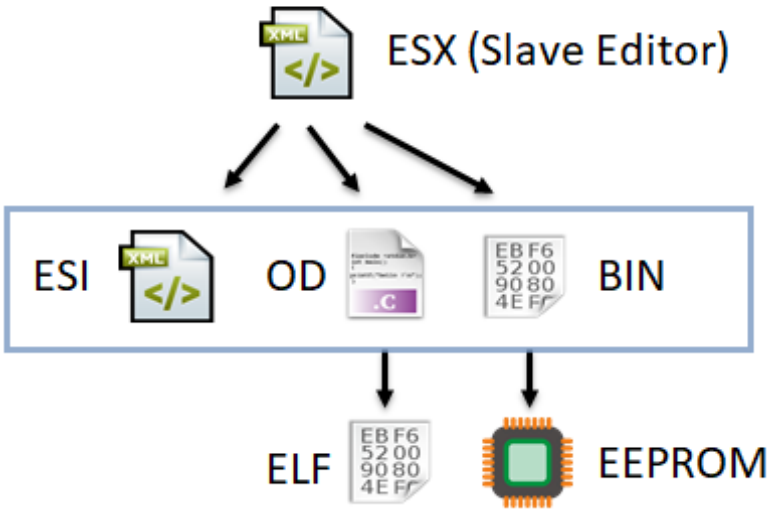

Figure 2.1: EtherCAT Slave Editor output objects ESI-file, C Object Dictionary and EEPROM image.

2 Start the Slave Editor

1. Start the stand-alone EtherCAT SDK or Workbench hosting the EtherCAT SDK plug-in.
2. Create or use an existing workspace
3. Create a new or use an existing project for Slave Editor output

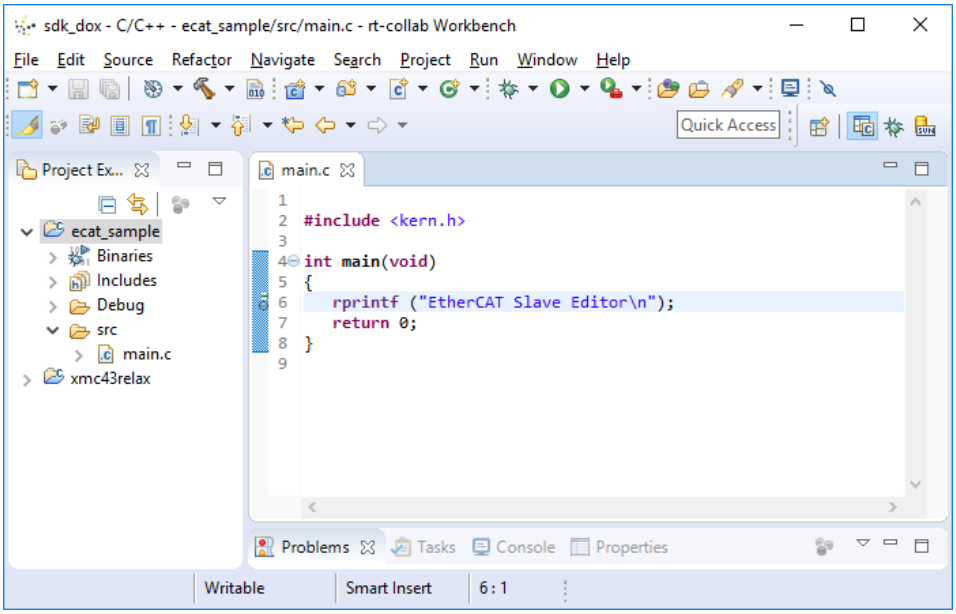

Figure 2.2: Example with simple Hello World project

3 Create an EtherCAT Slave Editor Project

1. Go to the create new project wizard **File** > **New** > **Other...**
2. Select wizard **EtherCAT** > **EtherCAT Slave Description**

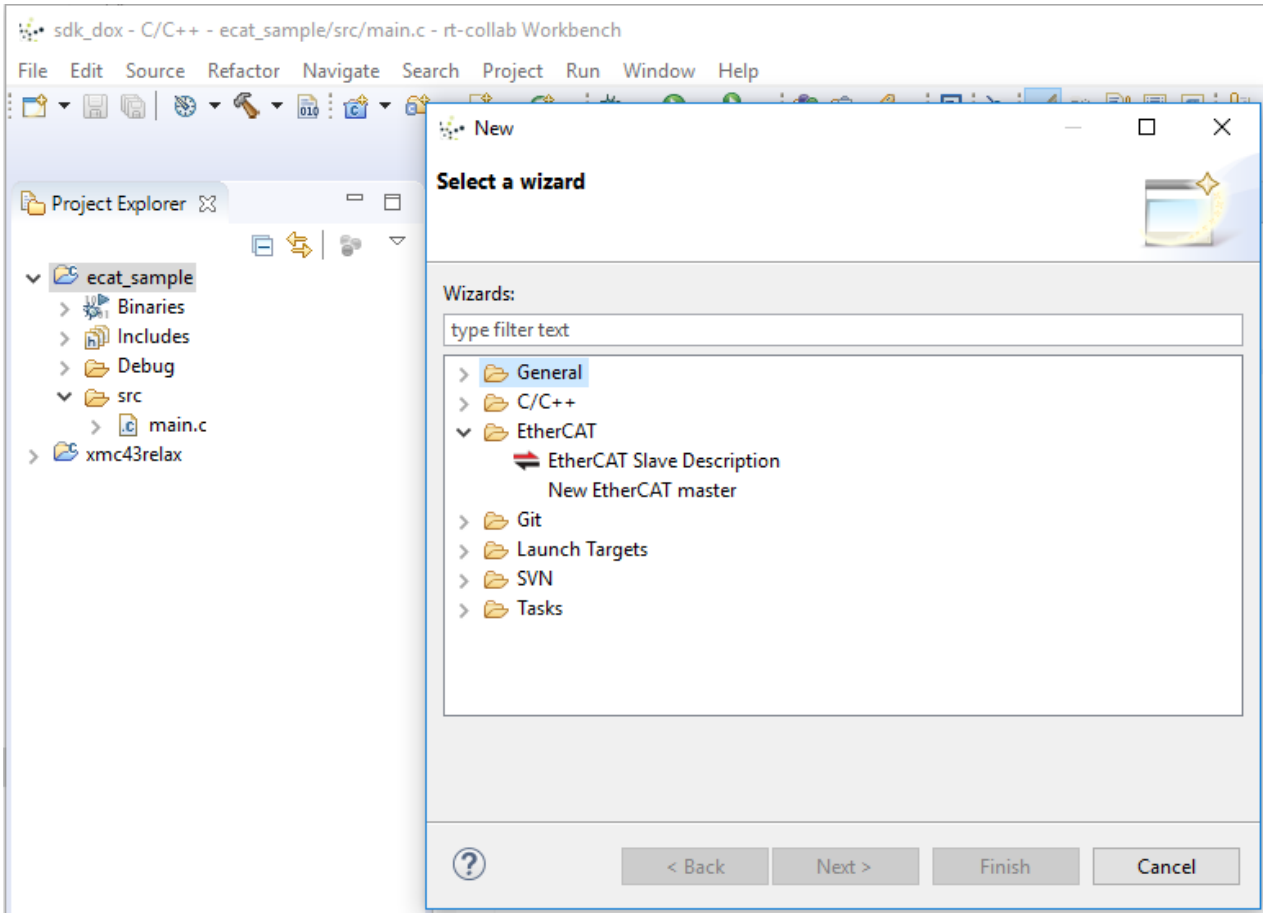

Figure 2.3: Select Wizard

3. Press Next and choose location for generated output.
4. Enter the EtherCAT slave information details, it is used to identify and present details of the slave device.

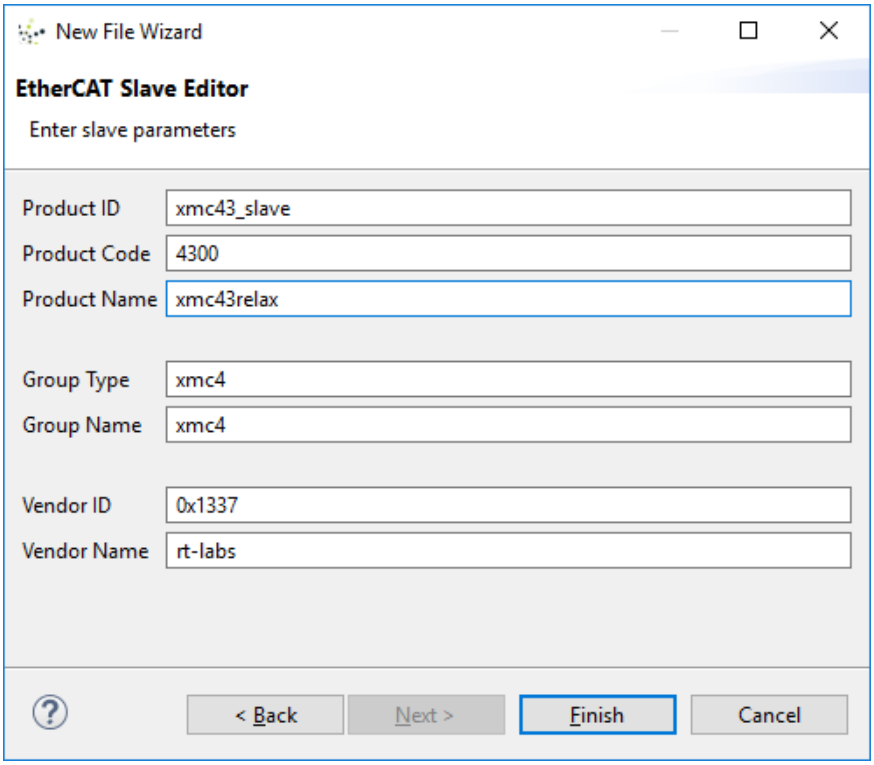

Figure 2.4: Add slave information

(a) Product ID, Device type identity, not used for identification

(b) Product Code, Vendor specific product code, used for identification in conjunction with vendor id

(c) Product Name, Detailed name of device, not used for identification

(d) Group Type, Group for similar devices with slightly different features (The Slave Editor treat it as one device in one group)

(e) Group Name, Name for this group (The Slave Editor treat it as one device in one group)

(f) Vendor ID, assigned by the EtherCAT Technology Group, used for identification in conjunction with product code

(g) Vendor Name, Vendor Name

5. Press finish to complete creating the EtherCAT Slave Editor project

4 Edit EtherCAT Slave Identity Information

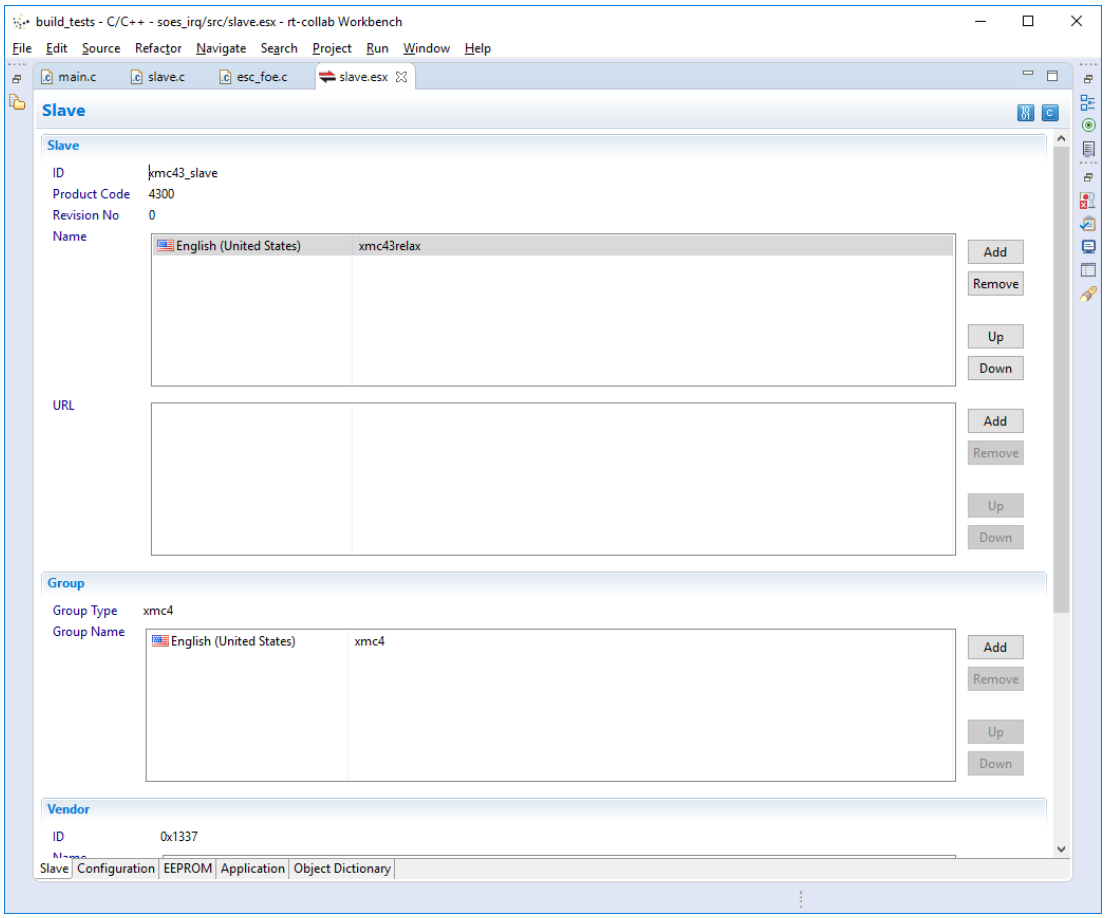

Figure 2.5: Slave Tab

Contains general slave information given when creating the EtherCAT Slave Editor project, here it is possible to add, update and remove given information and also add local language support where applicable. In addition, you can also add

- URL, for further information on the device. Usually pointing to the vendors homepage where up to date ESI files can be downloaded
  - Company, Company URL
  - Description, URL to ESI files

5 Enter EtherCAT slave Data Link Layer Configuration

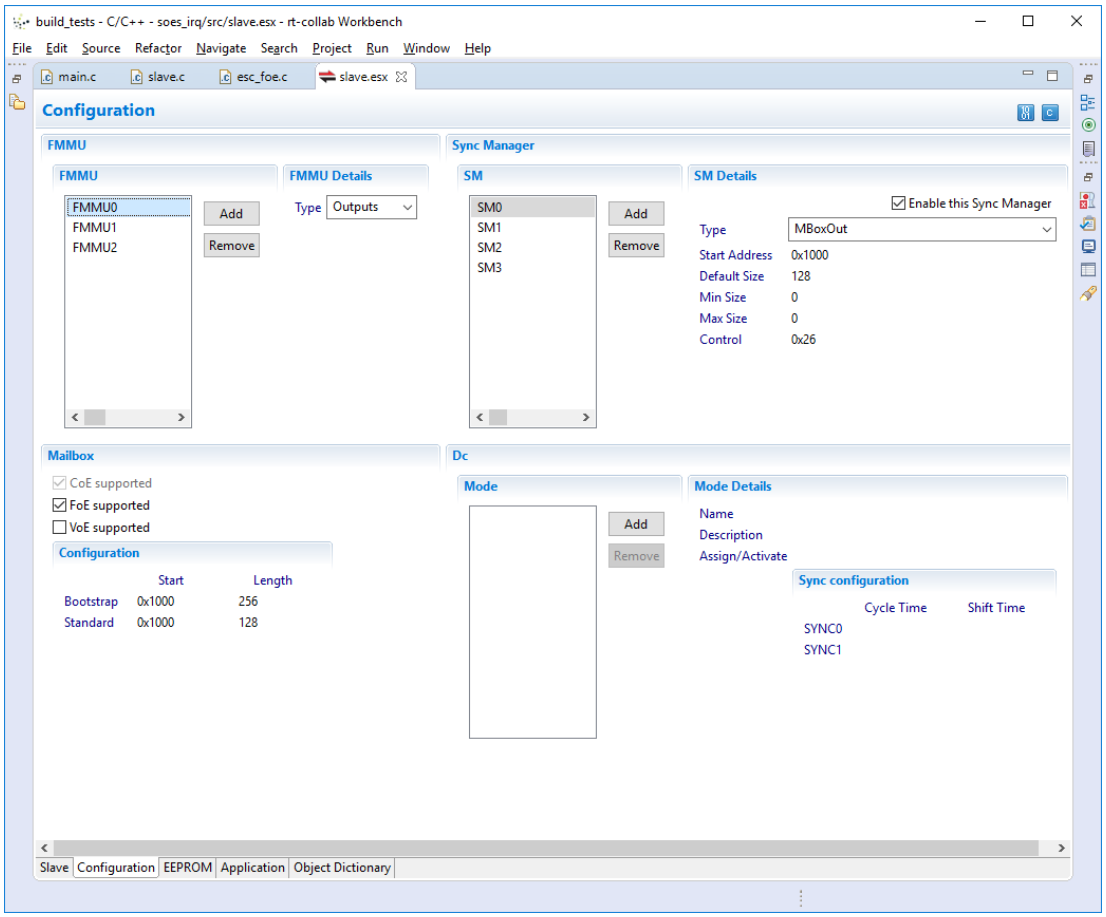

Figure 2.6: Configuration TAB

Basic configuration of the EtherCAT slave Data Link Layer including device Mailbox, FMMU and SM. Settings here are shared among EEPROM and ESI-file.

- Mailbox, description of available mailbox protocols
  - Bootstrap, address and length of bootstrap mailboxes, activate in bootstrap mode only.
  - Standard, address and length of standard mailboxes, standard mailbox settings are used during normal operations
- FMMU, definition of FMMU usage
  - Outputs, used for RxPDO
  - Inputs, used for TxPDO
  - MBoxState, FMMU used to poll Input MailboxState
- SM description of SyncManager including start address and direction
  - MBoxOut, Mailbox Data Master $>$ Slave
  - MBoxIn, Mailbox Data Slave $>$ Master
  - Outputs, Process Data Master $>$ Slave

  - Inputs, Process Data Slave > Master

- DC, description of synchronization mode for offline configuration
  - Name, unique identifier of operation mode for configuration tool
  - Description, vendor specific description of operation mode, recommended 'Free Run - no sync', 'SM Synchronous - synchronized on SyncManager event when process data is written' and 'DC-Synchronous - synchronized on DC event '
  - Assign / Activate - value of latch and sync control registers
  - Cycle Time - Cycle Time
  - Shift Time - Shift offset

6 Enter EtherCAT slave EEPROM PDI Settings

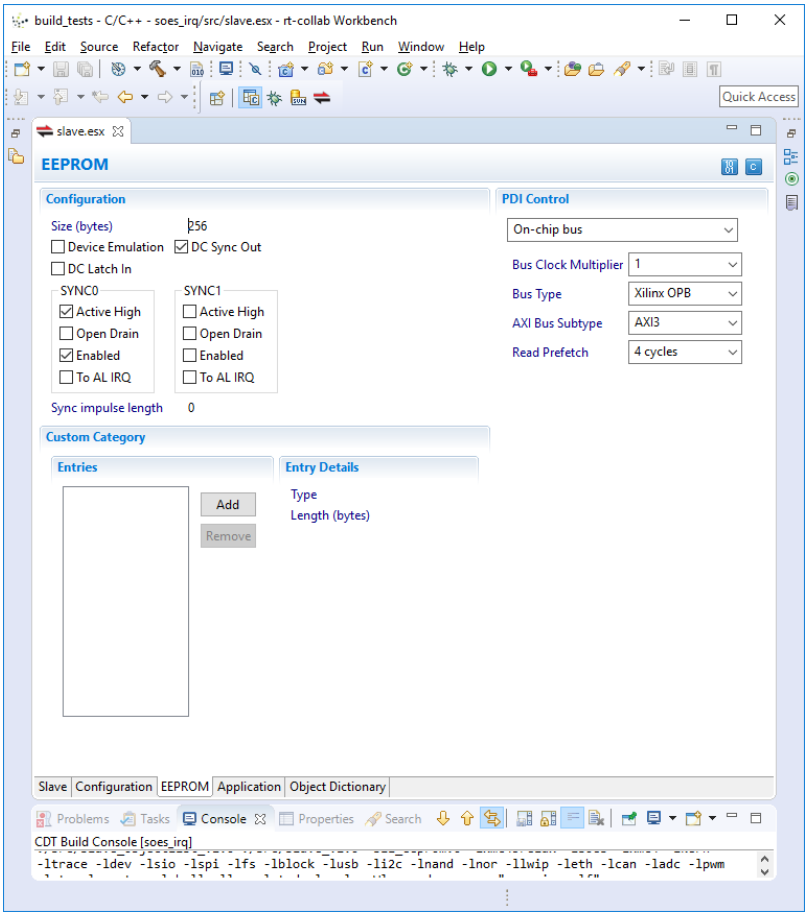

Figure 2.7: EEPROM Tab

EEPROM only settings for the EtherCAT Slave Controller Configuration Area, required to boot the ESC properly (this is ESC dependent). Most EEPROM data are taken from already given information and not presented here.

- Configuration, initialization values for ESC PDI configuration registers. Consult the ESC manual for proper settings.

- PDI Control, initialization values for ESC PDI control registers. Consult the ESC manual for proper settings.
- Custom Category, add custom category header and allocate room for custom category data in the EEPROM. Data will not be populated.

7 Add EtherCAT Slave PDO and Configuration Parameters

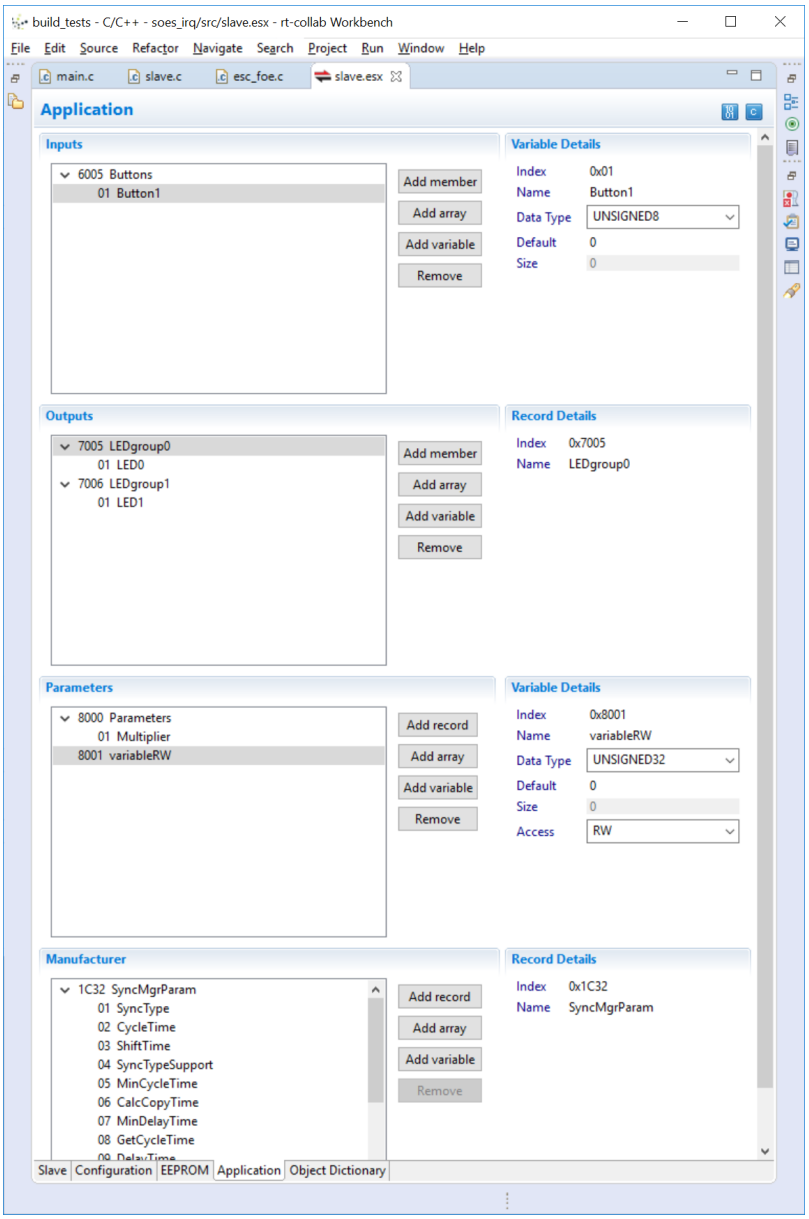

Figure 2.8: Application TAB

Here the application defines process data, configuration parameters and manufacture specific data. Objects entered in the different areas will get indexes according to ranges specified by the MDP (Modular Device Profile).

- 0x2000-0x5FFF, Manufacture Specific Area

- 0x6000-0x6FFF, Inputs Area, Objects that will be mapped to TxPDO
- 0x7000-0x7FFF, Outputs Area, Objects that will be mapped to RxPDO
- 0x8000-0x8FFF, Configuration Area, Configuration and settings objects. **TIP:** The EtherCAT master should write settings variables, the 0x8000 range, every time before making the state transition from PRE-OP to SAFE-OP.

The application can specify objects of 3 types.

1. Record - a group of variables of same or different datatypes that belong together, a struct in C terms
2. Array - an array of variables of same datatype with a size describing number of elements, different datatypes may be used for the array.
3. Variable - a single variable of different datatypes

The GUI provide assistance on allowed data types for the different types.

Properties for objects entered in the different areas

- General,
  - added object will automatically get assigned next free index in specified range.
  - index and sub index can be changed in the GUI by placing the cursor on the number and edited.
  - callbacks for inputs, outputs and RW parameters in generated C API must be implemented in the application.
- Inputs,
  - object will automatically get included in the TxPDO map.
  - object will automatically get a callback created in generated C API code for reading local data to the TxPDO object, cb_get_<object name>
- Outputs,
  - object will automatically get included in the RxPDO map.
  - object will automatically get a callback created in generated C API code for setting local data read from the RxPDO object, cb_set_<object name>
  - object will automatically get added to the default APP_safeoutput in the C API, there outputs are set to their default
- Parameters,
  - record, array or variable objects with access Read-Write will automatically get a post SDO download callback created in the generated C API code, cb_post_write_<object name>
- Manufacture,
  - no restrictions

8 Generate C Source Code, EEPROM and ESI-file

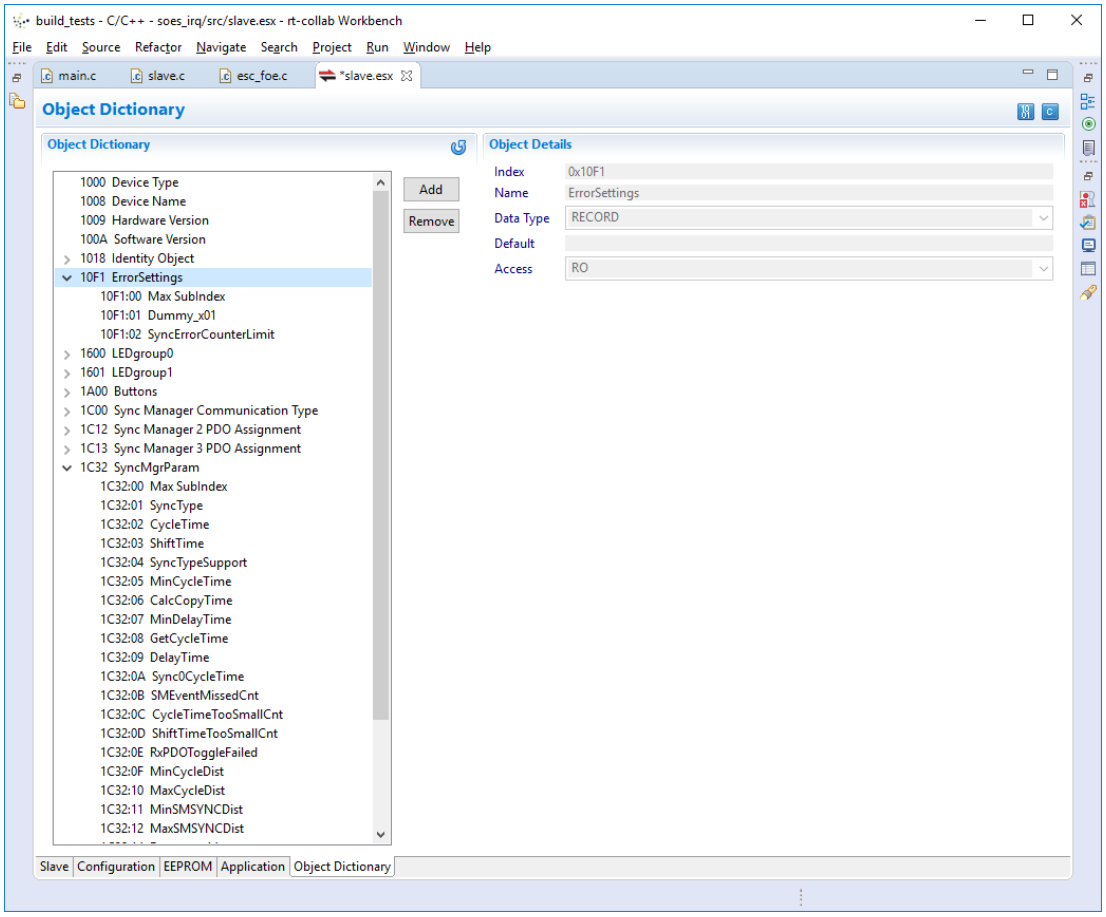

Figure 2.9: Object Dictionary Tab

Overview of the resulting Object Dictionary. The buttons in the right upper corner icon "IOIO" and icon "C" are used to generate the output. Code can be generated from any view.

- "IOIO"
  - ∗.bin, the EEPROM image. Targets using emulated EEPROM use objcopy to create a linkable object included in the target executable
  - ∗.xml, ESI file.
- "C" generate C source code.
  - utypes.h - user types created for the objects defined for the application
  - config.h - stack configuration parameters to setup hardware objects
  - <project name>.c/.h generated and fixed stack C API
  - <project name>_objectlist.c generated C object dictionary

Chapter 3

EtherCAT Explorer User Manual

1 Overview

EtherCAT Explorer is an EtherCAT network exploration, logging and diagnostics tool. Together with EtherCAT SDK Slave Editor and SOES EtherCAT Slave Stack the developer have an all-in-one tool for developing slaves in an efficient way easy to support and maintain throughout the lifecycle.

2 Start the EtherCAT Explorer

1. Install WinPcap to use EtherCAT Explorer (WinPcap is part of Wireshark. If it is already installed there is no need to install again.)
2. Start the stand-alone EtherCAT SDK or workbench hosting the EtherCAT SDK plug-in.
3. Open the **EtherCAT Explorer** perspective, goto **Windows** $>$ **Perspective** $>$ **Open Perspective** $>$ **Other...**.
4. Choose **EtherCAT Explorer**

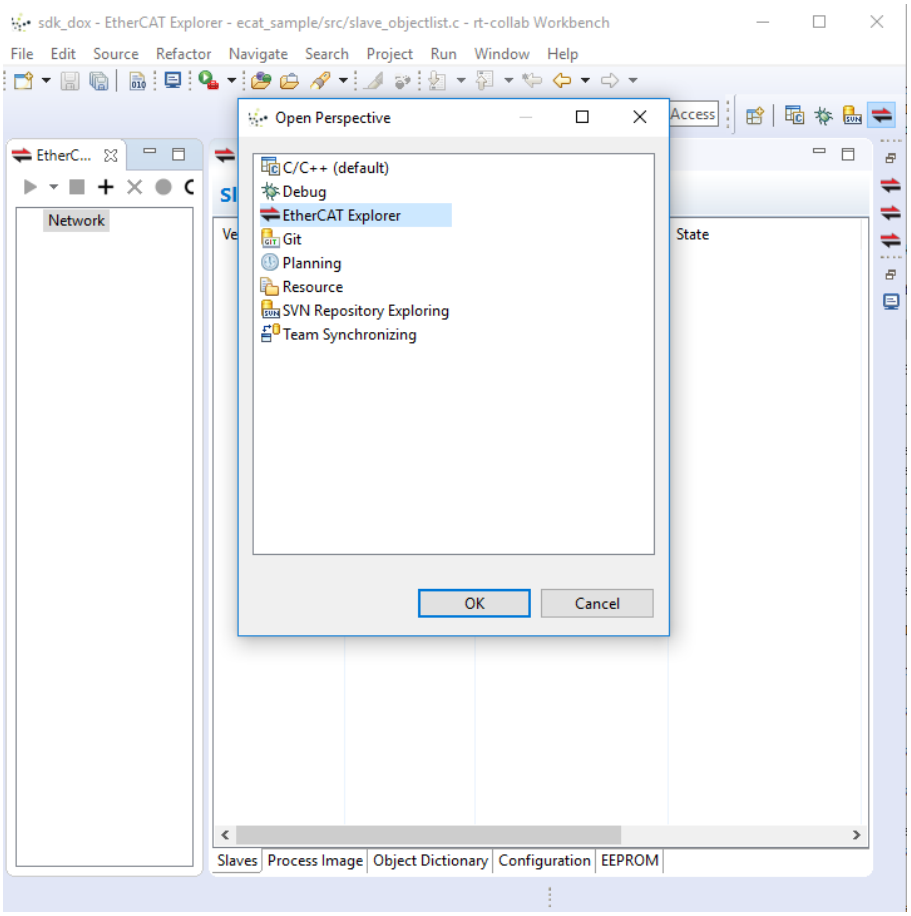

Figure 3.1: Open the EtherCAT Explorer perspective

3 Connect to an EtherCAT Network

Follow these instructions to configure and connect to an EtherCAT network.

1. To add a new EtherCAT master press the **+** button in the network tree-view toolbar
2. Select network interface and **Finish**
3. Start the network by pressing the **Play** arrow button
4. The slave network start is logged in the **Console** view
5. Individual slave information is presented in the columns. The **State** column can be edited, but the rest of the columns are read-only. Via state you can change individual slaves state from the drop-down list. No state change is validated and it have to be done in the correct sequence according to the EtherCAT state machine.
6. Starting, stopping, adding and removing masters in the network tree-view is handled by the buttons in the tree-view toolbar.

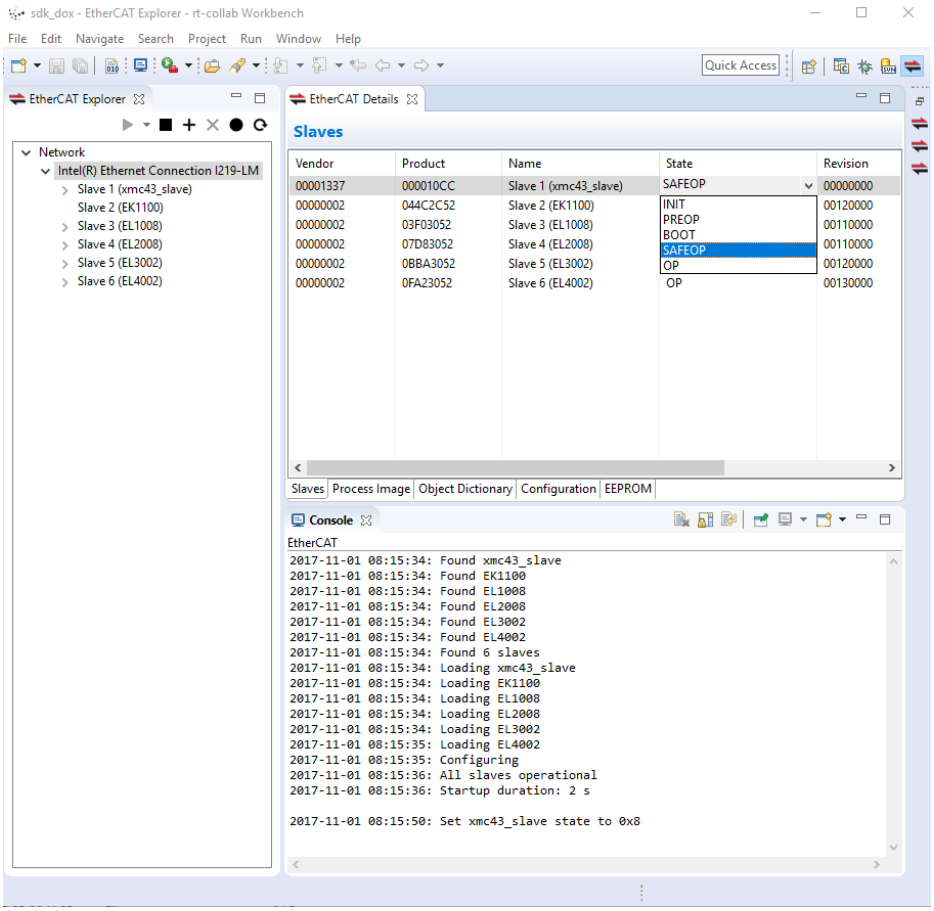

Figure 3.2: EtherCAT Network Overview

4 Explore EtherCAT Network Process Data

Select the **Process Image** tab to monitor and control the EtherCAT network process data image, all PDO objects are listed by name, type, bit length and value. For outputs it is possible to enter and change values. PDO:s can be drag-and-dropped to views for logging, EtherCAT Trace, EtherCAT Watch and EtherCAT SDO Watch.

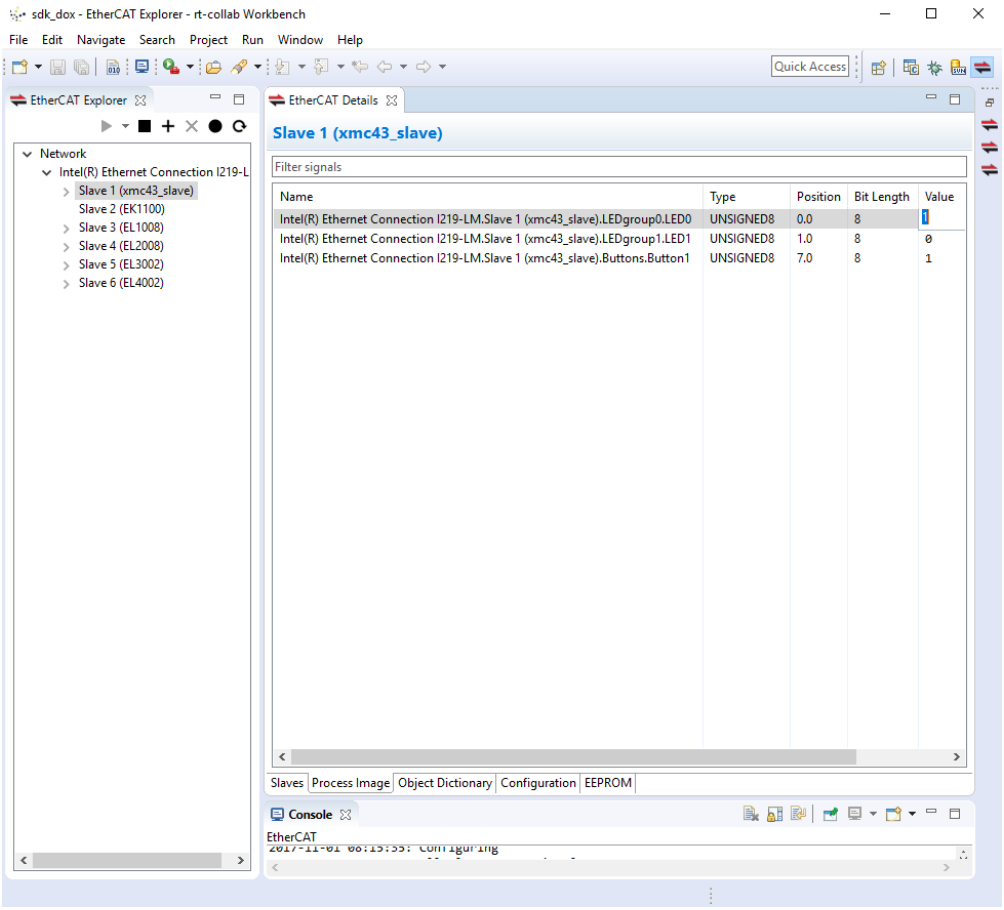

Figure 3.3: Process Image view

5 Explore EtherCAT Slave Object Dictionary

Select **Object Dictionary** view to browse slaves Object Dictionary, select a slave in the network tree-view to populate the view, data is only read on-demand by pressing the '**Refresh**'-button in the top right corner.

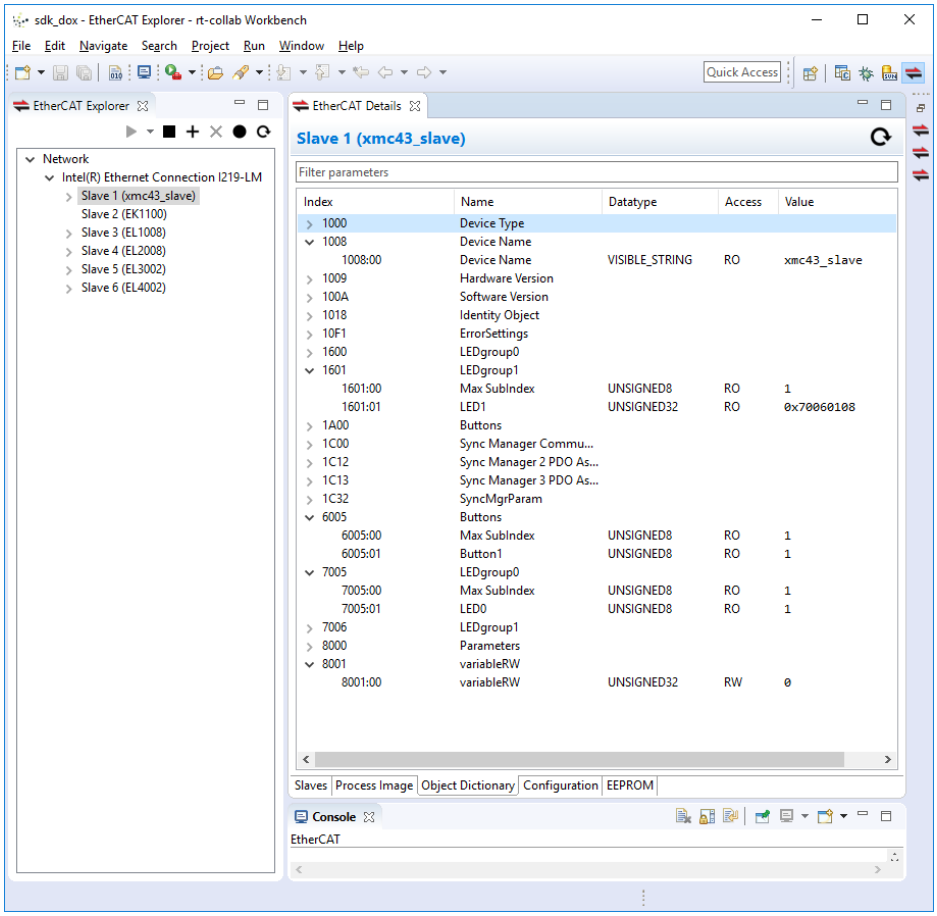

Figure 3.4: Object Dictionary view

6 Explore EtherCAT Slave Data Link Layer Configuration

Select Configuration for view of slaves' physical/logic mappings for the SyncManager and FMMU objects.

6.1 Write a file to EtherCAT slave

Press the **'Download File'**-button to transfer files to the slave over FoE (File Over EtherCAT). It will open a file browser dialog to select the file to transfer. The result of the transfer will be presented in the Console.

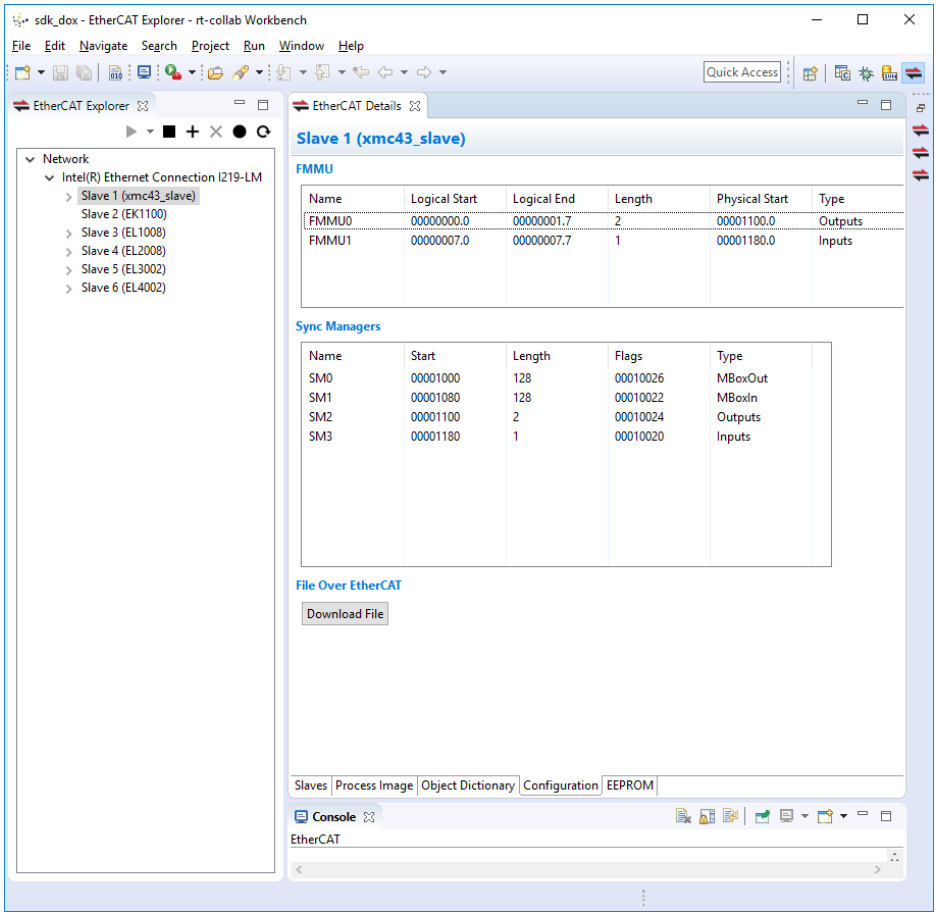

Figure 3.5: Configuration view

7 Manage EtherCAT Slave EEPROM

Select **EEPROM** view for slave EEPROM management. It supports Read and Write EEPROM online, Save and Load EEPROM images offline. It is possible to modify a read or loaded image by editing or adding EEPROM entries via the tree-view fields. However, the proper way to modify the EEPROM content, to keep slave identity information aligned, is by making the necessary changes in the EtherCAT Slave Editor.

7.1 Read online EEPROM

Select EtherCAT Slave in the tree-view. Press the **'Read EEPROM'**-button to populate the data view with EEPROM data read online.

7.2 Write Online EEPROM

Select EtherCAT Slave in the tree-view. Press the **'Write EEPROM'**-button to write the date to the slave.

NOTE! A faulty EEPROM can cause problems starting the EtherCAT slave.

7.3 Save an EEPROM Image File

Press the **'Save File'**-button to write present data in data view to an image file on disk.

7.4 Load EEPROM image file

Press the **'Load File'**-button to load data to data view, the data view is write to the slave by Write EEPROM. A loaded file can be opened in the external linked editor if associated.

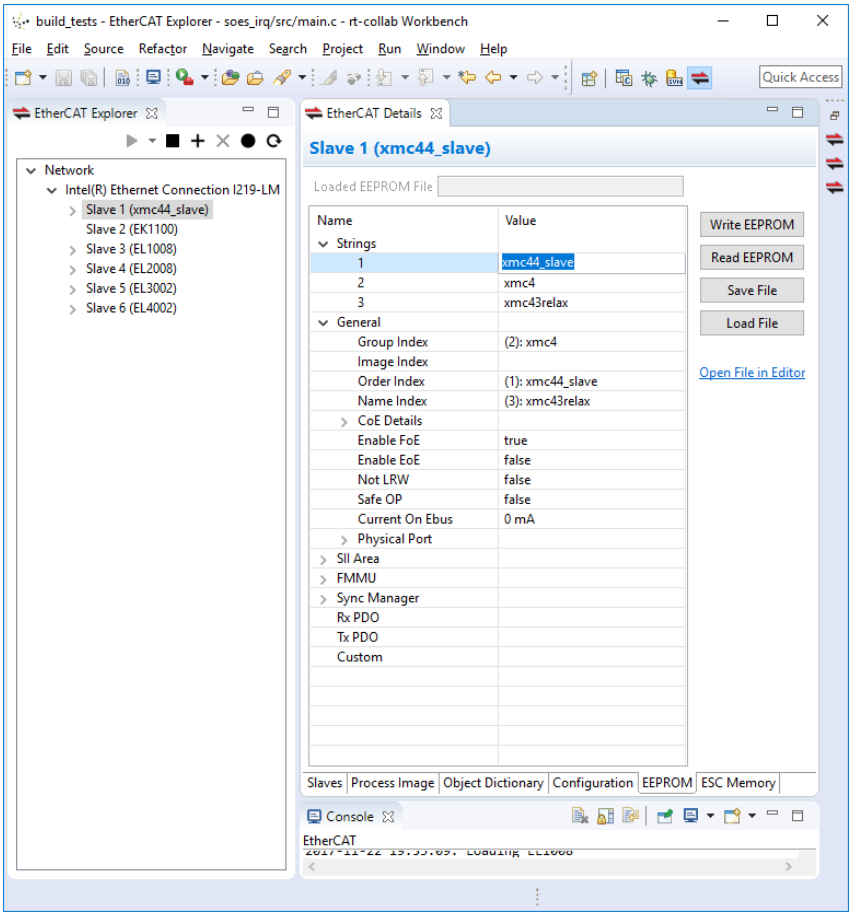

Figure 3.6: EEPROM view

8 Explore EtherCAT Slave ESC Registers and RAM

Select **ESC Memory** for EtherCAT Slave Controller Register and RAM view, select a slave in the network tree-view to populate the view, data for expanded sections are read on-demand by pressing the **'Refresh'**-button in the top right corner.

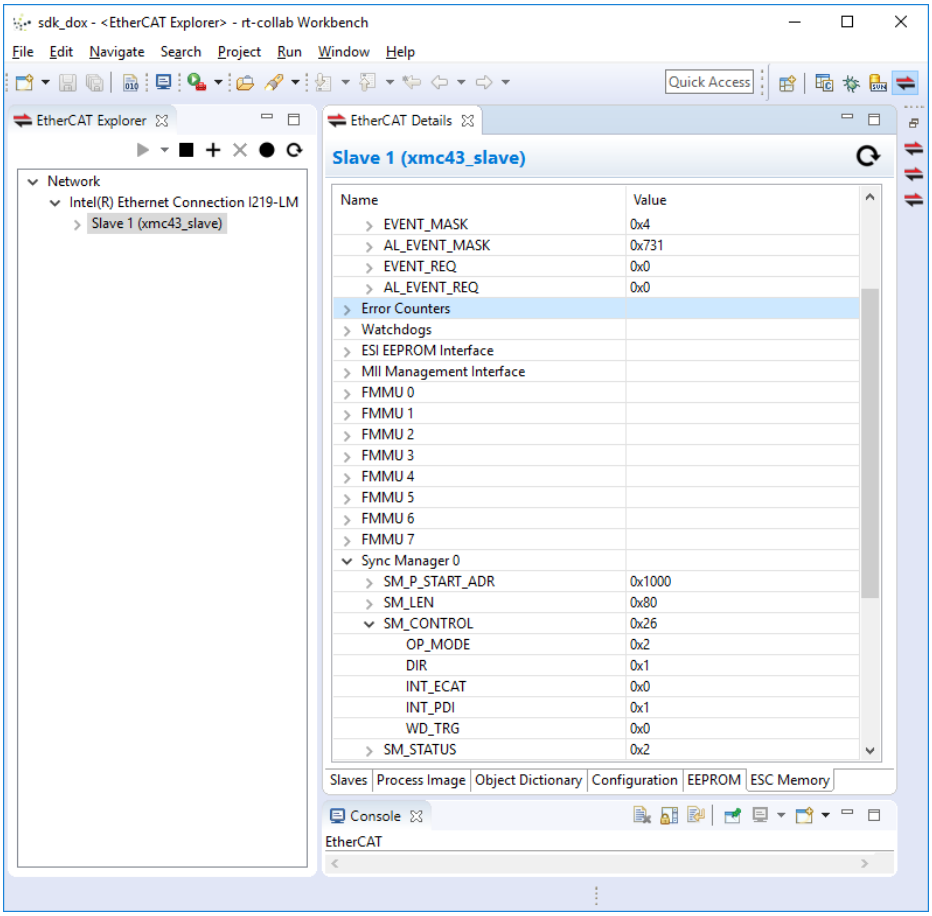

Figure 3.7: ESC (EtherCAT Slave Controller) Register and RAM view

Chapter 4

EtherCAT Explorer Reference Manual

This document lists and describes the features of the EtherCAT Explorer application. It can be used as a specification document or a brief but detailed and technical user documentation.

1 Network Model

All functionality of the application is centred around an EtherCAT network model.

The model consists of a list of *masters*. Each master has information about how to start or connect to an EtherCAT master. It also contains a list of connected *slaves*.

Each slave has information about one embedded unit. The information included the following: Name, object dictionary data, process image data and slave state.

The model structure looks like this:

- Master 1
  - Slave 1
  - Slave 2
  - ...
- Master 2
- ...

A master can either be in the state *stopped* or in the state *started*. Commands can only be executed on a started master, and information can only be read from a started master. (Information that has been read from a started master is however still availible in the GUI when the master has been stopped.)

When a master is started it scans the network for connected slaves. When a slave is found the master reads information about it into the model.

When the user exists the application the information about masters and slaves is saved to disk to be availible the next time the application is started. This included the information about the current process image and object dictionary values.

1.1 Master Types

The application has support for two types of EtherCAT masters.

- **Local SOEM Masters:** These are masters that run locally on the PC, as part of the EtherCAT Explorer application itself. They use the SOEM EtherCAT master implementation.
- **Remote EAP Masters:** These are masters that run on a different computer. EtherCAT Explorer communicates with them using EAP (EtherCAT Automation Protocol) protocol.

2 Common Functionality in GUI Views

This section describes functionality which is common to many of the views in the application.

2.1 Display Number Base

Most values in lists and trees can be displayed in decimal, hexadecimal or binary base. This is select in a context pop-up menu which is accessed by right-clicking on the object values.

2.2 Copy Text from Tables and Trees

Most values in table and tree vies can be copied to the clipboard. This action is triggered in a context pop-up menu which is accessed by right-clicking on the object values.

3 SDO Data Reuse

Normally when EtherCAT Explorer starts a master and connect to the network it reads information from the slaves in the network. This information includes the structure and contents objects dictionaries of the slaves. But when a master has read this information once it can re-use it to make this startup procedure quicker.

Note

> This functionality is currently implemented only for EAP (Remote) EtherCAT masters. The reason for this is that the data transfer rate is generally lower from an EAP network making data reuse more useful.

The following start alternatives are availible:

Clean Start (Read all data): This reads both the structure and values of the objects dictionary. This is the slowest start alternative, but it must be used if the structure of the object dictionary has changed. **Normal Start (Read only SDO values):** This assumes that the structure of the object dictionary is unchanged but refreshes all SDO values. **Quick Start (Skip reading SDO values):** This skips both reading the structure and the values of the object dictionary.

Warning

> If the **Normal Start** alternative is used when the object dictionary has changed since last update it can lead to serious problems! If the **Quick Start** alternative is used all SDO values will be stale. That is, they will have the values they had at the last refresh instead of the values they have when the master is started.

Slave matching$**$: This data is reused if a slave on the network matches a saved slave. The following properties of the slaves must be identical for two slaves to match:

- Name
- Serial Number
- Revision Number
- Vendor ID

Measurements on some networks show that SDO data reuse can reduce the start-up time of an EAP master with 50 %.

4 EtherCAT Explorer View

This view consists of a list of the configured EtherCAT masters.

- The view has a toolbar with the following buttons:
  - Create new master
  - Delete selected master or slave. This can also be triggered by the **Del** key.
  - Start traffic logging for selected master
  - Refresh traffic logging for selected master
  - Start selected master and connect to the network. This button has a menu with three sub-buttons. The difference start modes that they trigger is described in section SDO Data Reuse.
- **F2** key opens a **Rename Master** dialog.

Note

When a master is renamed in loses its connection to all PDO:s and SDO:s that have been added to the watch views. The PDO:s and SDO:s have to be re-added.

- Masters can be expanded to show their connected slaves
- Slaves can be expanded to show the names of the slaves' PDO:s

5 EtherCAT Details View

This view consists of a number of tabs which each provides some information about and send commands to the slaves in the network.

This view has the following properties:

- It contains detailed information about one slave, or – in some tabs – information about all slaves of master.
- The active slave (or master) is selected in the EtherCAT Explorer View.
- It consists of a number of tabs that each displays some information about the selected slave.

5.1 Slaves Tab

This tab lists *all* slaves for the selected master. It displays basic properties that is read from each slave.

The user can change the slave state in a drop-down menu in the table. When the slave state is changed in the GUI a message is sent on the network to the slave instructing it to set the new state.

Note

> It is possible to instruct a slave to make an invalid state transitions. It this happens the slave might ignore the instruction or go to an error state.

If a slave reports that it is in an error state then that state is displayed in red.

5.2 Process Image Tab

This tab lists the PDO:s (Process Data Objects) of the selected slave, or for all slaves of a selected master.

The information in the view is periodically re-read from the slaves. More about information about data refresh can be found in section network_com.

PDO:s values that are not read-only can be edited by clicking on their value or selecting them and pressing **F2**. When a value has been edited a message is immediately sent to the slave to update its PDO value.

5.3 Object Dictionary Tab

This tab lists the SDO:s (Service Data Objects) of the selected slave.

The information in the view is read during master startup (unless a quick start mode is used). It can be re-read by using the **Refresh** button in the header of the tab.

SDO:s values that are not read-only can be edited by clicking on their value or selecting them and pressing **F2**. When a value has been edited a message is immediately sent to the slave to update its SDO value.

5.4 Configuration Tab

This tab displays data about the FMMU:s (Fieldbus Memory Management Unit) and the sync managers of the selected slave. This information is read during master startup.

It also has a button to initiate a File Over EtherCAT procedure. This lets the user select a file on the PC which will be transfered to the selected slave.

5.5 EEPROM Tab

This tab displays EEPROM memory. The EEPROM data is parsed and displayed in a tree view.

EEPROM can be read from and written to slaves in the network. EEPROM that is loaded into the tab can also be saved to a file in the file system.

If the CRC checksum of the loaded EEPROM file is not correct a warning message is displayed in the tab header.

When EEPROM has been saved to or loaded from a file the file name is displayed in the top of the view.

5.6 ESC Memory Tab

This tab displays ESC (EtherCAT Slave Controller) memory for the selected slave. The ESC data is parsed and displayed in a tree view.

Data in the view is read from the slave when the top-level categories in the view are expanded. If the read procedure takes a too long time (for example due to network problems) a progress information dialog is displayed.

The **Refresh** button in the tab header makes the application reload data for all expanded sections from the selected slave.

6 Watch Views

There are two kinds of watch views:

- **EtherCAT Watch** view for for PDO values
- **EtherCAT SDO Watch** view for SDO values.

Both work in the same way. The views contain a list of the names the PDO and SDO values together with their current value.

PDO values are refreshed continually. SDO values are refreshed only when the Object Dictionary Tab is refreshed.

The values in the views can be edited. This immediately sends write commands to the slave on the network.

The values in the views can be displayed in different number bases and they can be copied. This is done using the pop-up context menu.

6.1 Adding and Removing Values

The values are added to the view by dragging them with the mouse from the Process Image Tab or Object Dictionary Tab in the EtherCAT Details View.

Values are removed using the pop-up context menu in of views. Values can be removed either one by one or all at once

7 Graph Views

The graph views visualises how PDO values changes with some variable.

8 EEPROM Editor

This components can be used to view and edit EEPROM data files, in the same way as in the EEPROM Tab of the EtherCAT Details View. The difference is that the EEPROM tab requires and connected slave to work, while the EEPROM Editor is separate from the Details view and the configured masters and slaves.

The editor also has full undo-redo functionality, which the EEPROM tag does not.

Chapter 5

SOES EtherCAT Slave Stack

1 Overview

The Simple Open EtherCAT Slave (SOES) is a library that provides user applications with means to access the EtherCAT fieldbus communication environment:

- EtherCAT State Machine
- Mailbox Interfaces
- Protocols
  - CoE
  - FoE

Required support for mailbox and protocols are examples when you need a slave stack to control the Application layer of EtherCAT. The PDI (Process Data Interface) used for such applications are either SPI (or similar) or an internal/etxernal CPU interface

2 Getting Started

Goto `OpenEtherCATsociety/SOES on GitHub`, there you can find releases or current version.

Clone or download to the destination folder. SOES is split in 3 parts.

- soes - the generic part configured from the application
- application - the application API that configure the stack and control the execution
- soes HAL - hardware abstraction layer for the slave stack, not generic and complements the application configuration of the stack by providing necessary hardware configuration and functions

For a project, use generic SOES and pick one application and soes/hal as template that fits selected hardware and software platform. If the EtherCAT Slave Editor is used it will generate parts of the application, otherwise they have to be provied by the developer.

Project structure

- soes archive

  - SOES/soes
  - SOES/soes/include/sys/gcc/cc.h

- application binary

  - SOES/application
  - SOES/soes/hal

Under SOES/application a set of sample applications can be found to be used as reference designs for different platforms.

3 EtherCAT Slave Stack Initialization

The EtherCAT slave stack is setup with an ESC (EtherCAT Slave Controller) generic initialization function , ecat_slv_init. The esc_cfg_t argument passed to the stack is copied, except the user_arg, to stack internal variables and therefore can go out-of-scope.

There are 3 modes of operations for the stack

- Polling (Free-Run)

  - PDI interrupt is not enabled
  - SyncX interrupt are not enabled
  - AL Event are polled for events every ESC Read/Write to be handled by the stack
  - ecat_slv() get called regular to handle stack operations

- Mixed Polling/Interrupt (SM or DC Synchronous)

  - PDI interrupt is enabled
  - Only SM2 should be masked to generate PDI interrupt.
  - Sync0 interrupt is enabled if DC Synchronous
  - AL Event are polled for events every ESC Read/Write to be handled by the stack
  - ecat_slv_poll() get called regular to handle stack operations
  - DIG_process(DIG_PROCESS_WD_FLAG) get called regular to kick the watchdog counter

- Interrupt (SM or DC Synchronous)

  - PDI interrupt is enabled
  - In addition to SM2 add SMCHANGE, EEP(if EEPROM emulated), ALCONTROL, SM0 and SM1 to the mask to generate PDI interrupt.
  - Sync0 interrupt is enabled if DC Synchronous
  - AL Event are not polled for events
  - ecat_slv_worker() get called when an interrupt occur, preferably from a background task
  - DIG_process(DIG_PROCESS_WD_FLAG) get called regular to kick the watchdog counter

Structure configuration parameter

- User Input - Optional

- Stack parameter configuration - Mandatory
- Stack and Application interaction functions - Optional and Mandatory depending on mode of operations

```
static esc_cfg_t config =
{
   /* User input to stack */
   .user_arg = NULL, /* passed along to ESC_config and ESC_init */

   /* Manadatory input to stack */
   .use_interrupt = 1, /* flag telling the stack if the user application will use
                          interrupts, 0= Polling, 1 = Mixed Polling/Interrupt
                          and Interrupt */

   .watchdog_cnt = 100, /* non UNIT watchdog counter, for the application
                           developer to decide UNIT. This example set 100
                           cnt and by calling ecat_slv or
                           DIG_process(DIG_PROCESS_WD_FLAG) every 1ms,
                           it creates a watchdog running at ~100ms. */

   /* Values taken from config.h to configure the stack mailboxes and SyncManagers */
   .mbxsize = MBXSIZE,
   .mbxsizeboot = MBXSIZEBOOT,
   .mbxbuffers = MBXBUFFERS,
   .mb[0] = {MBX0_sma, MBX0_sml, MBX0_sme, MBX0_smc, 0},
   .mb[1] = {MBX1_sma, MBX1_sml, MBX1_sme, MBX1_smc, 0},
   .mb_boot[0] = {MBX0_sma_b, MBX0_sml_b, MBX0_sme_b, MBX0_smc_b, 0},
   .mb_boot[1] = {MBX1_sma_b, MBX1_sml_b, MBX1_sme_b, MBX1_smc_b, 0},
   .pdosm[0] = {SM2_sma, 0, 0, SM2_smc, SM2_act},
   .pdosm[1] = {SM3_sma, 0, 0, SM3_smc, SM3_act},

   /* Optional input to stack for user application interaction with the stack
    * all functions given must be implemented in the application.
    */
   .pre_state_change_hook = NULL, /* hook called before state transition */
   .post_state_change_hook = NULL, /* hook called after state transition */

   .application_hook = NULL, /* hook in application loop called when
                                DIG_process(DIG_PROCESS_APP_HOOK_FLAG) */
   .safeoutput_override = NULL, /* user override of default safeoutput when stack
                                   stop outputs */

   .pre_object_download_hook = NULL, /* hook called before object download,
                                        if hook return 0 the download will not
                                        take place */
   .post_object_download_hook = NULL, /* hook called after object download */

   .rxpdo_override = NULL, /* user override of default rxpdo */
   .txpdo_override = NULL, /* user override of default txpdo */

   /* Mandatory input to stack for SM and DC synchronous applications */
   .esc_hw_interrupt_enable = NULL, /* callback to function that enable IRQ
                                       based on the Event MASK */
   .esc_hw_interrupt_disable = NULL, /* callback to function that disable IRQ
                                        based on the Event MASK */

   /* Mandatory input for emulated eeprom */
   .esc_hw_eep_handler = NULL /* callback to function that handle an emulated eeprom */
};
```

The stack is setup but not running.

```
void main_run(void * arg)
{
...
   ecat_slv_init(&config);
```

4 EtharCAT Slave Stack API

4.1 DIG_process - Process Data Handler

Implements the watch-dog counter to count if the slave should make a state change to SAFEOP due to missing incoming SM2 events. Updates local I/O and run the application in the following order, call read EtherCAT outputs, execute user provided application hook and call write EtherCAT inputs.

- `#define DIG_PROCESS_INPUTS_FLAG 0x01`
- `#define DIG_PROCESS_OUTPUTS_FLAG 0x02`
- `#define DIG_PROCESS_WD_FLAG 0x04`
- `#define DIG_PROCESS_APP_HOOK_FLAG 0x08`

Parameters

in	*flags*	= User input what to execute

```
void DIG_process (uint8_t flags);
```

4.2 ecat_slv_worker - Non-synchronous Interrupt Handler

Handler for SM change, SM0/1, AL CONTROL and EEPROM events, the application control what interrupts that should be served and re-activated with event mask argument. Interrupts served here are not part of synchronization and is handle preferably by a background-task or similar. NOTE: No locking of ESC_read/ESC_write is done for slaves that rely on consecutive address/data beeing provided on ESC_read/ESC_write.

Parameters

in	*event_mask*	= Event mask for interrupts to serve and re-activate after served

```
void ecat_slv_worker (uint32_t event_mask);
```

4.3 ecat_slv_poll - Interrupt Polling Routine for Non-Synchronous Interrupt

Poll SM0/1, EEPROM and AL CONTROL events in a SM/DC synchronization application NOTE: No locking of ESC_read/WSC_write is done for slaves that rely on consecutive address/data beeing provided on ESC_read/ESC_write.

```
void ecat_slv_poll (void);
```

4.4 ecat_slv - All Polling Routine

Poll all events in a free-run application

```
void ecat_slv (void);
```

4.5 ecat_slv_init - Stack Initialization

Parameters

in	*config*	= User input how to configure the stack

```
void ecat_slv_init (esc_cfg_t * config);
```

5 EtherCAT Slave Stack HW Layer Implementation

5.1 EtherCAT HW Layer Polling Interrupts

When running in polling or mixed polling/interrupt mode AL Event must be polled. Best done in ESC_read and ESC_write

```
void ESC_read (uint16_t address, void *buf, uint16_t len)
{
   ESCvar.ALevent = <HW read ALevent>;
   memcpy (buf, ESCADDR(address), len);
}

void ESC_write (uint16_t address, void *buf, uint16_t len)
{
   ESCvar.ALevent = <HW read ALevent>;
   memcpy (ESCADDR(address), buf, len);
}
```

5.2 EtherCAT HW Layer Enable/Disable Interrupts

When running in mixed polling/interrupt or interrupt mode the EtherCAT HW layer should provide with function to enable and disable interrupts, those functions will be called by the EtherCAT slave stack with mask to enable/disable SM2 and DC events, DC if DC is active. The code below acts as pseudo code since access to ESC registers varies.

Example SM- or DC synchrounous application only PDI interrupt is used

```
void ESC_interrupt_enable (uint32_t mask)
{
   AL_EVENT_MASK = mask;
}

void ESC_interrupt_disable (uint32_t mask)
{
   AL_EVENT_MASK &= ~mask;
}
```

Example DC synchrounous application PDI and separate sync0 is used.

```
void ESC_interrupt_enable (uint32_t mask)
{
   AL_EVENT_MASK = mask;
   if(mask & dc_mask)
   {
       int_enable(SYNC0);
   ...
}
void ESC_interrupt_disable (uint32_t mask)
{
   AL_EVENT_MASK &= ~mask;
   if(mask & dc_mask)
   {
       int_disable(SYNC0);
   ...
}
```

5.3 EtherCAT HW Layer SYNC0 Interrupt

Sync0 used in DC synchronous is generated from a ESC internal block for Distributed Clocks, sync0 interrupts can be setup as a seperate interrupt source or part of the PDI interrupt. In most cases it is used together with SM2 interrupts, therefor DIG_process allow to flag what parts it should execute. In this example sync0 is expected to happen after PDI isr SM2, SM2 handle DIG_process(DIG_PROCESS_OUTPUTS_FLAG) copying the RxPDO to local variables. Sync0 finish with executing the application and writing the TxPDO for next frame, DIG_process(DIG_PROCESS_APP_HOOK_FLAG | DIG_PROCESS_INPUTS_FLAG).

Example DC synchronous application with separate sync0 interrupt

```
void sync0_isr (void * arg)
{
    DIG_process(DIG_PROCESS_APP_HOOK_FLAG | DIG_PROCESS_INPUTS_FLAG);
}
```

5.4 EtherCAT HW Layer PDI Interrupt

PDI interrupt is a collection of interrupt sources, consult the ESC reference manual for details. Example, sync0 is handled by a separate interrupt, SM2 is mandatory when running an interrupt mode. Depending on software support the application can choose to poll AL event for non-synchronous interrupts when running mixed polling/interrupt mode Or run interrupt mode where no polling of AL event is used. The code below acts as pseudo code since access to ESC registers varies.

Example SM- or DC Synchronous application running mixed polling/interrupt mode

```
void pdi_isr (void * arg)
{
   /* High prio interrupt used for synchronization */
   if(ESCvar.ALevent & ESCREG_ALEVENT_SM2)
   {
      /* If DC sync is not active, run the application, all except for the Watchdog */
      if(ESCvar.dcsync == 0)
      {
         DIG_process(DIG_PROCESS_OUTPUTS_FLAG | DIG_PROCESS_APP_HOOK_FLAG |
               DIG_PROCESS_INPUTS_FLAG);
      }
      /* If DC sync is active, call output handler only */
      else
      {
         DIG_process(DIG_PROCESS_OUTPUTS_FLAG);
      }
   }
}
```

Example SM- or DC Synchronous application running interrupt mode, add separate handler for non-synchronous interrupts to minimize execution time of the interrupt. In this example a semaphore is used to start a low prio task.

```
void pdi_isr (void * arg)
{
...
   ESCvar.ALevent = AL_EVENT_MASK;
   /* High prio interrupt used for synchronization */
   if(ESCvar.ALevent & ESCREG_ALEVENT_SM2)
   {
      /* If DC sync is not active, run the application, all except for the Watchdog */
      if(ESCvar.dcsync == 0)
      {
         DIG_process(DIG_PROCESS_OUTPUTS_FLAG | DIG_PROCESS_APP_HOOK_FLAG |
```

```
                DIG_PROCESS_INPUTS_FLAG);
      }
      /* If DC sync is active, call output handler only */
      else
      {
         DIG_process(DIG_PROCESS_OUTPUTS_FLAG);
      }
   }
   /* Assume there is task support, handle low prio interrupts from back-ground
    * task not blocking for coming SM2 interrupts
    */
   if(ESCvar.ALevent & (ESCREG_ALEVENT_CONTROL | ESCREG_ALEVENT_SMCHANGE
         | ESCREG_ALEVENT_SM0 | ESCREG_ALEVENT_SM1 | ESCREG_ALEVENT_EEP))
   {
      /* Mask interrupts while servicing them */
      AL_EVENT_MASK &= ~(ESCREG_ALEVENT_CONTROL | ESCREG_ALEVENT_SMCHANGE
            | ESCREG_ALEVENT_SM0 | ESCREG_ALEVENT_SM1 | ESCREG_ALEVENT_EEP);
      /* Signal back-ground task */
      sem_signal(ecat_isr_sem);
   }
}
```

5.5 EtherCAT HW Layer Emulated EEPROM

Using emulated eeprom require a handler called by application code. The handler should call the generic EEP handler for incoming EEP interrupt and implement the HW parts for EEP write/EEP read. Example can be found for XMC4 targets under <SOES>/application

```
void ESC_eep_handler(void)
{
   /* Handle incoming EEP interrupt  */
   EEP_process ();
   /* Implements read/write for volatile and non-volatile EEPROM data */
   EEP_hw_process();
}

static esc_cfg_t config =
{
..
   .esc_hw_eep_handler = ESC_eep_handler
..
};
```

5.6 EtherCAT HW Layer Initialization

The ESC hardware layer is possible to place in the SOES library or Application project, to handle both scenarios the EtherCAT slave stack pass the ESC configuration as parameter via ecat_slv_init to ESC_init. ESC config has an user_arg that can be used for miscellaneous information passed from the application to ESC hardware initialization. ESC_init should setup the hardware to serve the application with necessary functions. The code below acts as pseudo code since access to ESC registers varies.

Example SM- or DC Synchronous application running mixed polling/interrupt mode

```
void ESC_init (const esc_cfg_t * config)
{
    /* Setup PDI interrupt */
    int_connect (IRQ_PDI, pdi_isr, NULL);
    int_enable (IRQ_PDI);

    /* Set mask to disable all interrupts, the stack enables SM2 interrupt */
    AL_EVENT_MASK = 0;

    /* Setup sync0 interrupt if DC synchronous */
```

```
    int_connect (IRQ_SYNC0, sync0_isr, NULL);
    /* Let the stack enable the DC interrupt */
    int_disable(IRQ_SYNC0);
}
```

Example SM- or DC Synchronous application running interrupt mode, example implement an emulated EEPROM and require ESCREG_ALEVENT_EEP, no need to pass that event mask if EEPROM is connected via I2C.

```
/* Non-synchronous interrupt handler function */
static void isr_run(void * arg)
{
   while(1)
   {
      sem_wait(ecat_isr_sem);
      ecat_slv_worker(ESCREG_ALEVENT_CONTROL | ESCREG_ALEVENT_SMCHANGE
            | ESCREG_ALEVENT_SM0 | ESCREG_ALEVENT_SM1 | ESCREG_ALEVENT_EEP);
   }
}

void ESC_init (const esc_cfg_t * config)
{
    /* Create non-synchronous interrupt handler task and use a
     * semaphore for signaling
     */
    ecat_isr_sem = sem_create(0);
    task_spawn ("soes_isr", isr_run, 9, 2048, NULL);

    /* Setup PDI interrupt */
    int_connect (IRQ_PDI, pdi_isr, NULL);
    int_enable (IRQ_PDI);

    /* Set mask to enable non-synchronous interrupts to be able to operate,
     * let the stack enable SM2 interrupt.
     */
    AL_EVENT_MASK = (ESCREG_ALEVENT_SMCHANGE |
                     ESCREG_ALEVENT_EEP |
                     ESCREG_ALEVENT_CONTROL |
                     ESCREG_ALEVENT_SM0 |
                     ESCREG_ALEVENT_SM1);

    /* Setup sync0 interrupt if DC synchronous */
    int_connect (IRQ_SYNC0, sync0_isr, NULL);
    /* Let the stack enable the DC interrupt */
    int_disable(IRQ_SYNC0);
}
```

6 Implement the Application

If the EtherCAT Slave Editor is used it will generate parts of the application that needs to be implemented, otherwise the final linking of the application will fail with undefined reference errors. If no code generating tool is used one of the reference designs can be modified with desired data.

There are two types of stack interaction funtions

- Overrides - are mutually exclusive, if set only the function provided by the applcation will execute.
- Hooks - are a complement to the code provided by SOES.

6.1 Handle Application Configuration Parameters

The EtherCAT slave stack give the developer possibility to verify, validate and take actions on SDO downloads, it is done with pre- and post SDO download hooks.

Pre SDO download handler:

It is possible to add a pre SDO download hook that will give the application a chance to validate and prevent the download from taking place.

Post SDO download handler:

The EtherCAT Slave Editor will create post SDO download hooks for all SDO parameter objects with access type ReadWrite, it is the most common use-case but to cover all use-cases it is possible to add a custom post_object_download_hook that will be called for all other SDO objects not matching the previous criterias.

```
int user_pre_dl_objecthandler (uint16_t index, uint8_t subindex)
{
   /* Prevent object from beeing written */
   if (index == 0x1c12)
   {
      SDO_abort (index, subindex, ABORT_READONLY);
      return 0;
   }
   return 1;
}

void user_post_dl_objecthandler (uint16_t index, uint8_t subindex)
{
   /* Re-calculate PDO size on change */
   if (index == 0x1c12)act
      RXPDOsize = ESC_SM2_sml = sizeRXPDO();
   }
}

static esc_cfg_t config =
{
...
   .pre_object_download_hook = user_pre_dl_objecthandler,
   .post_object_download_hook = user_post_dl_objecthandler,
...
};

/* EtherCAT Slave Editor generated code */
void ESC_objecthandler (uint16_t index, uint8_t subindex)
{
 ...
   /* Genrated post download call back, must be implemented by the application */
   case 0x8001:
   {
      cb_post_write_variableRW(subindex);
      break;
   }
...
}
```

6.2 Handle Application IO

The EtherCAT Slave Editor will create hooks for all RxPDO/TxPDO objects, the hook will be called accordingly when required. If no code generating tool is used one of the reference designs can be modified with desired callbacks.

```
void cb_get_Buttons()
{
   Rb.Buttons.Button1 = gpio_get(GPIO_BUTTON1);
}
void cb_set_LEDgroup0()
{
   gpio_set(GPIO_LED1, Wb.LEDgroup0.LED0);
}
```

```
void DIG_process (uint8_t flags)
{
...
   /* Handle Outputs */
   if ((flags & DIG_PROCESS_OUTPUTS_FLAG) > 0)
   {
...
         RXPDO_update();
...
         /* Set outputs */
         cb_set_LEDgroup0();
         cb_set_LEDgroup1();
...
...
   /* Handle Inputs */
   if ((flags & DIG_PROCESS_INPUTS_FLAG) > 0)
   {
...
         /* Update inputs */
         cb_get_Buttons();
         TXPDO_update();
...
}
```

SOES include a mandatory function for safe outputs, it is called by the stack when it stop outputs, the EtherCAT Slave Editor generate code setting all RxPDOs to their default values. It's possible for the user to override this function providing a custom function and possible to add local IO handling.

```
void APP_safeoutput (void)
{
...
   if(ESCvar.safeoutput_override != NULL)
   {
      (ESCvar.safeoutput_override)();
   }
   else
   {
      // Set safe values for Wb.LEDgroup0
      Wb.LEDgroup0.LED0 = 0;
...
   }
}
```

6.3 Application State Machine Interactions

The EtherCAT slave stack give the developer possibility to verify, validate and take actions on state changes, it is done with pre- and post state change hooks.

Parameters

in	$*as$	= Combined info for requested and current state ((ALCONTROL $<<$ 4)| ALSTATUS)
in	$*an$	= Local value of AL Status in the stack state machine, will become the EtherCAT Slave AL Status when the state machine function finsh.

```
void post_state_change_hook (uint8_t * as, uint8_t * an);
void post_state_change_hook (uint8_t * as, uint8_t * an);

\
void post_state_change_hook (uint8_t * as, uint8_t * an)
{
   /* Add specific step change hooks here */
   if ((*as == BOOT_TO_INIT) && (*an == ESCinit))
```

```
   {
      /* On upgrade finished verify the image and reset, else return and fail
       * the state change
       */
      upgrade_finished();
      /* If we return here */
      ESC_ALerror (ALERR_NOVALIDFIRMWARE);
      /* Upgrade failed, enter init with error */
       an  = (ESCinit | ESCerror);
   }
   else if((*as == PREOP_TO_SAFEOP))
   {
      rprintf("boot PREOP_TO_SAFEOP\n");
      ESC_ALerror (ALERR_NOVALIDFIRMWARE);
      /* Stay in preop with error bit set */
       an = (ESCpreop | ESCerror);
   }
#endif
}
static esc_cfg_t config =
{
...
   .application_hook = user_application,
...
};
```

6.4 Execute the Application

To excute the application accordingly when required, add an application hook serving as the user application entry point.

```
void DIG_process (uint8_t flags)
{
   /* Call application */
   if ((flags & DIG_PROCESS_APP_HOOK_FLAG) > 0)
   {
...
      /* Call application callback if set */
      if (ESCvar.application_hook != NULL)
      {
         (ESCvar.application_hook)();
      }
   }
...
}
static esc_cfg_t config =
{
...
   .application_hook = user_application,
...
};
```

7 Run the Application

For examples and references goto <SOES>/applications and <SOES>/soes/hal.

7.1 EtherCAT Slave Stack Polling

Configure not to use interrupts, call the EtherCAT stack handler periodically. This correspond to legacy use of SOES, soes_init , while(1) { soes(); }.

```
static esc_cfg_t config =
```

```
{
..
   .use_interrupt = 0,
..
};
void main_run(void * arg)
{
    ecat_slv_init(&config);
    while(1)
    {
        ecat_slv();
    }
}
```

7.2 EtherCAT Slave Stack Mixed Polling/Interrupt Mode

Configure to use interrupts, set interrupt enable/disable handlers, call the EtherCAT slave stack handler for polling non-synchronous interrupts periodically. Kick the watchdog supervising for incoming data from the EtherCAT Master

```
static esc_cfg_t config =
{
..
   .use_interrupt = 1,
   .esc_hw_interrupt_enable = ESC_interrupt_enable,
   .esc_hw_interrupt_disable = ESC_interrupt_disable,
..
};
void main_run(void * arg)
{
    ecat_slv_init(&config);
    while(1)
    {
        /* Kick watchdog with watchdog count intervals */
        DIG_process(DIG_PROCESS_WD_FLAG);
        ecat_slv_poll();
    }
}
```

7.3 EtherCAT Slave Stack Interrupt Mode

Configure to use interrupts, set interrupt enable/disable handlers, kick the watchdog supervising for incoming data from the EtherCAT Master

```
static esc_cfg_t config =
{
..
   .use_interrupt = 1,
   .esc_hw_interrupt_enable = ESC_interrupt_enable,
   .esc_hw_interrupt_disable = ESC_interrupt_disable,
..
};
void main_run(void * arg)
{
    ecat_slv_init(&config);
    while(1)
    {
        /* Kick watchdog with watchdog count intervals */
        DIG_process(DIG_PROCESS_WD_FLAG);
        task_delay(1)
    }
}
```