# EtherCAT Slave Controller Software

# USER’S GUIDE

# Copyright



Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

# Revision Information

This is version v2.01.00.00 of this document, last updated on August 31, 2020.

# Table of Contents


**Copyright** . . . 2

**Revision Information** . . . 2

**1 Introduction** . . . 4
1.1 Terms and Abbreviations . . . 5
1.2 EtherCAT References and Resources . . . 5

**2 EtherCAT Software Development Overview** . . . 6
2.1 EtherCAT Application Software Stack . . . 7
2.2 EtherCAT Software States and Application APIs . . . 8
2.3 EtherCAT Development Overview . . . 10

**3 Getting Started Using Examples** . . . 11
3.1 CPU1 PDI HAL Test Example . . . 11
3.2 CPU1 Echoback Demo Example . . . 12
3.3 CPU1 Echoback Solution Example . . . 15
3.4 CPU1 Allocate ECAT to CM Example . . . 18
3.5 CM PDI HAL Test Example . . . 18
3.6 CM Echoback Demo Example . . . 19
3.7 CM Echoback Solution Example . . . 22
3.8 CM CiA402 Solution Example . . . 25

**4 How-To Procedures** . . . 28
4.1 Example External Connections . . . 28
4.2 Setup TwinCAT . . . 28
4.3 Scanning for EtherCAT Devices via TwinCAT . . . 30
4.4 Program ControlCard EEPROM . . . 31
4.5 Use TwinCAT Memory Window . . . 32
4.6 Generate Slave Stack Code . . . 33

**5 Troubleshooting** . . . 35

**6 Using Acontis EtherCAT Master** . . . 37
6.1 EC-Engineer: Installation and Setup . . . 37
6.2 EC-Engineer: Add ESI File . . . 40
6.3 EC-Engineer: CPU1 Echoback Demo Example . . . 41
6.4 EC-Engineer: Program ControlCard EEPROM . . . 44
6.5 EC-Engineer: Viewing ESC Registers . . . 46

**7 ESC HAL APIs** . . . 47
7.1 CPU1 HAL APIs . . . 47
7.1.1 Function Documentation . . . 48
7.2 CM HAL APIs . . . 59
7.2.1 Function Documentation . . . 60

**8 EtherCAT Performance Data** . . . 74
8.1 EtherCAT Example Software Analysis . . . 74
8.2 EtherCAT Network Analysis . . . 77

**9 Revision History** . . . 78

**IMPORTANT NOTICE** . . . 79

# 1 Introduction

The EtherCAT Slave Controller (ESC) hardware abstraction layer (HAL) drivers and slave examples are designed to operate on the EtherCAT hardware peripheral on F2838x devices. The F2838x devices support EtherCAT on either CPU1 or the Connectivity Manager (CM). Either core can be setup to be an ESC.

This user guide details information on EtherCAT software development, how to setup the EtherCAT master software (TwinCAT or EC-Engineer) on your computer, provides details on the HAL driver APIs, and steps to run the EtherCAT Slave example applications.

**IMPORTANT:** F2838x software, including EtherCAT software, is now designed for use with XTAL clock source of 25MHz (previously 20MHz)

- The latest F2838x controlCARDs (Rev.B and later) have been updated to use 25MHz by default and require no action from the developer.
- IF you have an older 20MHz XTAL controlCARD (E1, E2, or Rev.A), refer to the controlCARD documentation on steps to reconfigure the controlCARD from 20MHz to 25MHz. The EtherCAT examples are configured to not run on 20MHz controlCARD and will halt/enter infinite loop to blink LEDs if ran.
- Note that if you have a custom board, to meet EtherCAT standards, the EtherCAT IP and EtherCAT PHYs need to share the same input clock. This is further detailed in the F2838x Technical Reference Manual as well as Beckhoff EtherCAT documentation.

**Minimum Requirements:**
Code Composer Studio v9, C2000 Compiler v18.12.1.LTS, ARM Compiler v18.12.1.LTS

**Chapter Overview:**

**Chapter 2** - **EtherCAT Software Development Overview**
An overview of EtherCAT software application development
**Chapter 3** - **Getting Started Using Examples**
Summary of the EtherCAT examples and the necessary steps for running the examples on the device using TwinCAT Master
**Chapter 4** - **How-To Procedures**
Provides various step-by-step instructions on how-to setup, use, and configure the ESC via the EtherCAT TwinCAT Master
**Chapter 5** - **Troubleshooting**
Details common usage problems and their solutions
**Chapter 6** - **Using Acontis EtherCAT Master**
Details how to use the Acontis EtherCAT Master instead of TwinCAT Master
**Chapter 7** - **ESC HAL APIs**
Describes the CPU1 and CM HAL driver APIs
**Chapter 8** - **EtherCAT Performance Data**
Profiled EtherCAT example application and network performance data
**Chapter 9** - **Revision History**
The revision history of the software

## 1.1 Terms and Abbreviations

| Term | Definition |
|---|---|
| CCS | Code Composer Studio |
| EtherCAT | Ethernet for Control Automation Technology |
| CM | Connectivity Manager |
| HAL | Hardware Abstraction Layer |
| PDI | Processor Data Interface |
| TwinCAT | EtherCAT master software from Beckhoff |
| EC-Engineer | EtherCAT master software from Acontis |
| ESC | EtherCAT Slave Controller |
| ESCSS | EtherCAT Slave Controller Subsystem |
| ENI | EtherCAT Network Information |
| ESI | EtherCAT Slave Information |
| API | Application Programming Interface |
| SSC | Beckhoff Slave Stack Code Tool |

Table 1.1: Terms and Abbreviations

## 1.2 EtherCAT References and Resources

- **Texas Instruments:**
  - TMS320F2838x Microcontrollers Technical Reference Manual: Link
  - EtherCAT Protocol Training for F2838x MCUs: Link
- **EtherCAT Forum:**
  - ETG EtherCAT Developers Forum: Link
- **EtherCAT Masters:**
  - Beckhoff TwinCAT: Link
  - Acontis EC-Engineer: Link
- **EtherCAT Tools:**
  - Slave Stack Code Tool: Link
  - Conformance Test Tool: Link
- **Highlighted EtherCAT Documents:**
  - ESC Slave Implementation Guide: Link
  - ESC Datasheet Section I - Technology (IP components, EEPROM, network cycle): Link
  - ESC Datasheet Section II - Register Description: Link
  - ESC Application Note - PHY Selection Guide: Link

# 2 EtherCAT Software Development Overview

EtherCAT Slave Controller software development involves understanding the data flow of a complete system application, familiarizing with the EtherCAT application stack, and knowing how the EtherCAT master/slave interact.

This chapter includes the following:

**2.1 EtherCAT Application Software Stack** - Details on the various layers in the EtherCAT software application stack

**2.2 EtherCAT Software States and Application APIs** - Details on the EtherCAT slave software states and main EtherCAT application APIs

**2.3 EtherCAT Development Overview** - Details on the high level steps to setup an EtherCAT slave application

Figure 2.1 below represents an example EtherCAT slave controller system running on the F2838x. This example uses CPU1 to run the motor control algorithms and uses the CM to handle running the EtherCAT software. The flow is as follows:

1. CPU1 - Initializes the device, allocates the EtherCAT IP to the CM core, and begins the motor algorithms
2. CM - Completes the EtherCAT hardware initialization, performs the EtherCAT software initialization, and begins running the EtherCAT slave state machine
3. EtherCAT Master - Locates the F2838x slave on the network and begins providing the motor commands and data
4. From here, CM receives the EtherCAT master data, transfers it to CPU1 via IPC, CPU1 responds to the CM via IPC with any feedback data (ex: motor status), and CM provides this feedback data to the EtherCAT Master

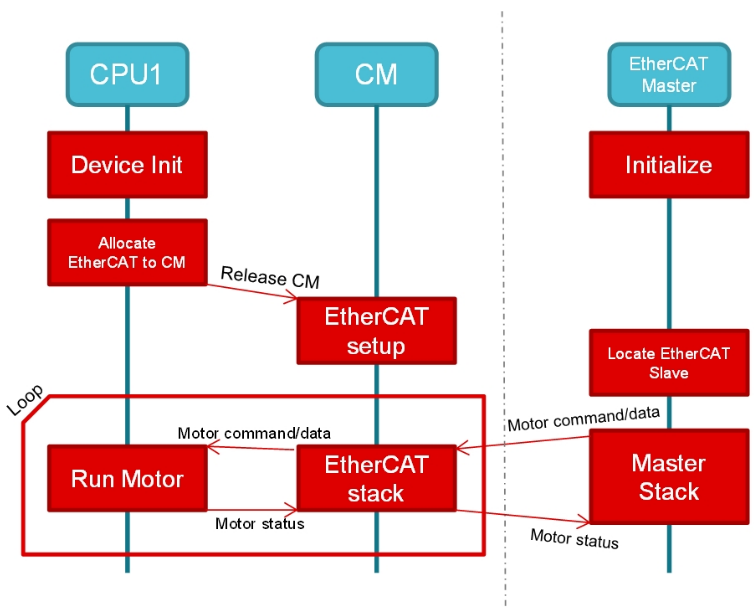


Figure 2.1: EtherCAT Slave Example

## 2.1 EtherCAT Application Software Stack

This section details the various EtherCAT slave application stack layers. The figure below displays the layers from MCU to application.

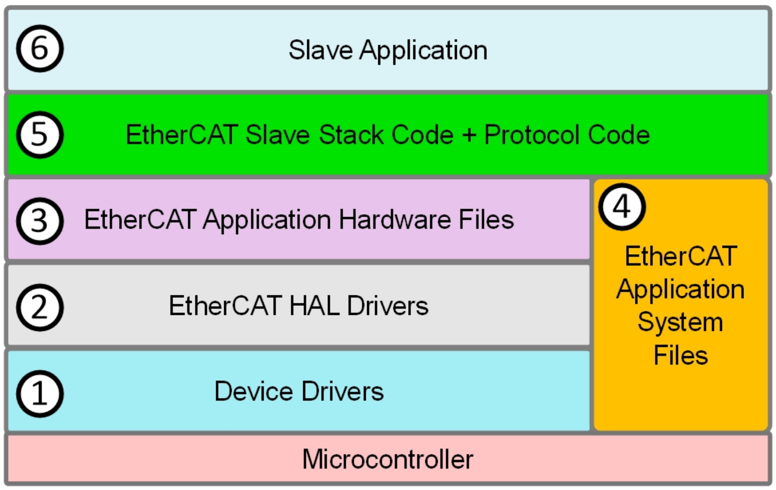


Figure 2.2: EtherCAT Application Stack

**Layer 1**:

- There are the specific microcontroller drivers
- Such drivers include the system control, EtherCAT subsystem, etc

**Layer 2**:

- EtherCAT device-specific slave hardware abstraction layer (HAL) drivers
- These API requirements are defined by the Beckhoff Slave Implementation specification
- API functions include EtherCAT hardware initialization, reading/writing to EtherCAT registers/memory, etc

**Layer 3**:

- Application device-specific hardware header files to map HAL driver APIs to Beckhoff slave stack API naming and usage scheme

**Layer 4**:

- Device-specific application system wrapper APIs
- Includes wrapper functions for APIs such as memcpy, memset, etc
- For CPU1, these are primarily necessary to handle word to byte conversations

**Layer 5**:

- The Beckhoff EtherCAT slave stack code developed and provided by Beckhoff
- The EtherCAT slave stack code includes the EtherCAT slave state machine

- Additionally, the stack code includes the supported EtherCAT protocol code (CAN over EtherCAT, Ethernet over EtherCAT, etc)
- These files are generated from the Beckhoff Slave Stack Code (SSC) tool

**Layer 6**:

- The user slave application
- Includes the main application loop and application required APIs to handle EtherCAT slave state changes. Refer to this section for more info on the states.
- Additionally, depending on the EtherCAT protocol, includes defines for that protocol. For example, it defines the object dictionary for the CAN over EtherCAT protocol.

## 2.2 EtherCAT Software States and Application APIs

The EtherCAT slave has 4 main states: Initialization(Init), PreOperational(PreOP), SafeOperational(SafeOP), and Operational(OP). Any EtherCAT slave application requires a set of function handlers that are called by the Beckhoff slave stack state machine. The main EtherCAT application code goes in `APPL_Application()` which is called by the main EtherCAT loop or SYNC interrupt. Figure 2.3 below details the various slave software states and the handlers that are called by the stack during those transitions.

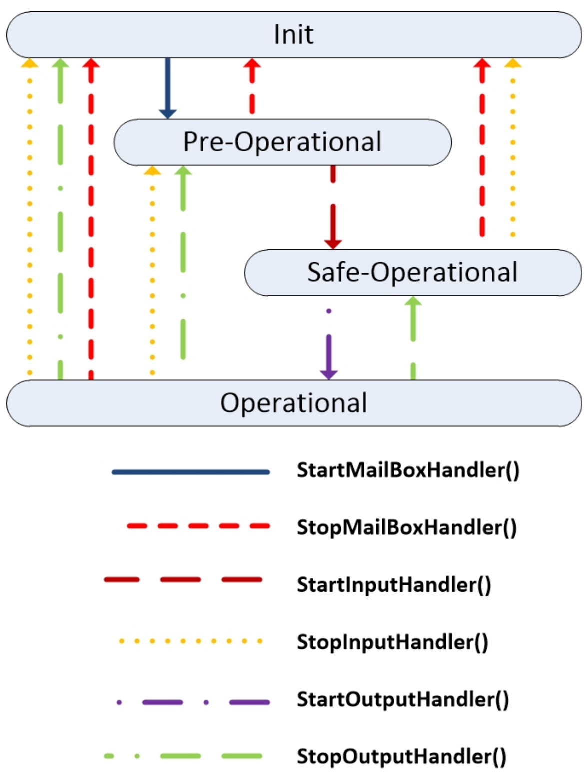


Figure 2.3: EtherCAT Software States

The details of each state and transition actions:

1. **Init State**:
   - First state upon EtherCAT initialization
   - No communicaton on the applications layer
   - EtherCAT master has access to the datalink information registers
2. **Transition to PreOP**:
   - Master requested PreOP state
   - SyncManager (SM0, SM1) mailbox settings are checked
   - Mailbox SyncManager enabled
   - `APPL_StartMailBoxHandler()` called
3. **PreOP State**:
   - Mailbox communicaton on the application layer
   - No process data communication
4. **Transition to SafeOP**:
   - Master requested SafeOP state
   - Master configures application parameters using the mailbox (ex: calculate process data size, setup process data mapping, application specific settings)
   - Master configures DL register (process data syncManagers, FMMUs)
   - `APPL_StartInputHandler()` called
5. **SafeOP State**:
   - Mailbox communication on the application layer
   - Process data communication for inputs only (outputs kept in "safe" state)
6. **Transition to OP**:
   - Master requested OP state
   - Master sends valid Outputs
   - `APPL_StartOutputHandler()` called
7. **OP State**:
   - Process data communication for inputs and outputs

## 2.3 EtherCAT Development Overview

This provides a look at the high level development flow of the steps described in later chapters in this guide. For more specific, step-by-step details, see Getting Started Using Examples Chapter.

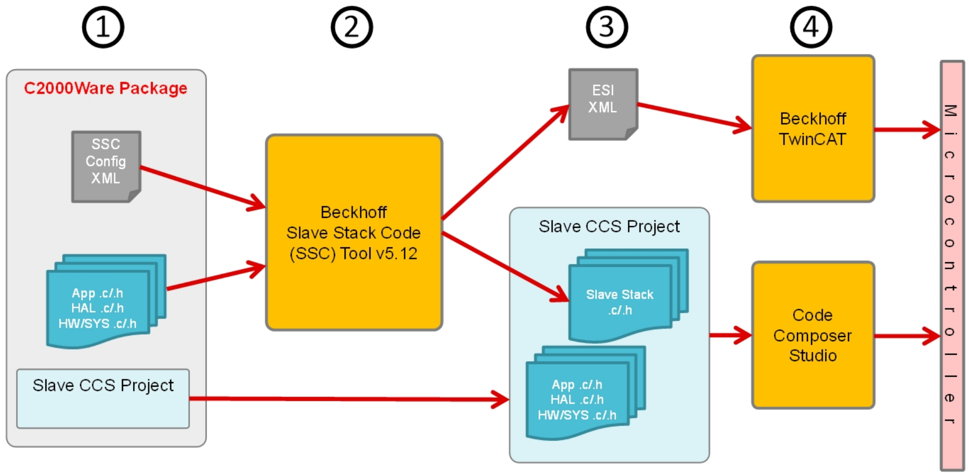


Figure 2.4: EtherCAT Development Flow

1. Download C2000Ware and familiarize with the EtherCAT collateral provided
   - SSC Configuration XML (f2838x_ssc_config.xml): A file to import into the SSC tool to be able to generate F2838x CPU1 or F2838x CM specific slave stack files
   - Application files (main example code, EtherCAT HAL drivers, EtherCAT device system files): Files required for the examples as well as interfacing with the slave stack code files
   - Slave CCS Projects: CCS example projects for F2838x CPU1 and F2838x CM
2. Using the slave stack code (SSC) tool
   - Beckhoff EtherCAT slave stack code is not provided in C2000Ware, requires generation from the SSC tool
   - The SSC tool configurations for F2838x are setup via importing the SSC configuration XML
   - Upon importing, users can select between generating slave stack code for F2838x CPU1 or CM
3. Understanding the SSC generated files
   - The SSC generates the device configured slave stack code files which get imported as part of the slave CCS example projects
   - The SSC generates the ESI (EtherCAT Slave Information) XML file. This file is provided to the EtherCAT master to be used to program the EtherCAT slave EEPROM as well as for the master to understand the capabilities of that particular EtherCAT slave.
4. Running EtherCAT master and programming the MCU
   - Once the EtherCAT slave example project (which now includes the EtherCAT slave stack files) is imported and built in CCS, this gets loaded and ran on the F2838x device
   - After providing the ESI file to the EtherCAT master, such as Beckhoff TwinCAT, the EtherCAT master is used to program the EtherCAT slave EEPROM. Once EEPROM is valid, EtherCAT master can begin the process of requesting the EtherCAT slave to reach operational mode.

# 3 Getting Started Using Examples

This chapter details how to get setup and begin using the EtherCAT examples. These instructions use **Beckhoff's TwinCAT** as the EtherCAT master.

For supplemental instructions on how to use **Acontis' EC-Engineer master** to perform equivalent actions, refer to Chapter 6.

The EtherCAT software includes the following:

**3.1 CPU1 PDI HAL Test Example** Sets up EtherCAT for CPU1 and performs a test of the PDI HAL APIs. Additionally, the example can be used to get started with basic communication between the EtherCAT master and ESC

**3.2 CPU1 Echoback Demo Example** A precompiled demo example that demonstrates usage of the EtherCAT slave stack code and loops back data to the master from the slave.

**3.3 CPU1 Echoback Solution Example** An example solution framework, which requires EtherCAT slave stack code to be integrated, that performs a loop back of data to the master from the slave.

**3.4 CPU1 Allocate ECAT to CM Example** Configures the EtherCAT clock and GPIOs before allocating EtherCAT ownership to the CM

**3.5 CM PDI HAL Test Example** Sets up EtherCAT for CM and performs a test of the PDI HAL APIs. Additionally, the example can be used to get started with basic communication between the EtherCAT master and ESC

**3.6 CM Echoback Demo Example** A precompiled demo example that demonstrates usage of the EtherCAT slave stack code and loops back data to the master from the slave.

**3.7 CM Echoback Solution Example** An example solution framework, which requires EtherCAT slave stack code to be integrated, that performs a loop back of data to the master from the slave.

**3.8 CM CiA402 Solution Example** An example solution framework, which requires EtherCAT slave stack code to be integrated, that includes a sample CiA402 application.

## 3.1 CPU1 PDI HAL Test Example

- This example sets up EtherCAT to be allocated to CPU1 and configures the required EtherCAT GPIOs and clocking. Additionally, the example performs a series of reads and writes to the full range of EtherCAT RAM using the HAL APIs. These can be observed from the CCS memory browser or TwinCAT ESC memory browser.
- This example is self-checking when performing the reads and writes. The following details the pass and fail signals:
  - Pass Signal - Both controlCARD LEDs (D1,D2) are on (not flashing)
  - Fail Signal - Both controlCARD LEDs (D1,D2) are flashing

**Note:** The intent of this project is to demonstrate the usage of the PDI. Therefore, no EtherCAT stack is included in this demo.

1. First, TwinCAT must be installed and setup. Refer to Section 4.2.
2. Check your external connections: Section 4.1
3. Open CCS and import the example `f2838x_cpu1_pdi_hal_test_app`

4. Select the RAM build configuration and build the example
5. Load the example to the controlCARD and run the code
   (a) **Important**: If the EtherCAT HAL example for the CM was loaded previously and the controlCARD hasn't been power cycled since, make sure to power cycle the controlCARD before running this example
6. If this your first time running any EtherCAT code (CPU1 or CM) on the controlCARD, the LEDs should be indicating a fail signal.
   (a) This failure is occurring because the minimum required EtherCAT EEPROM locations aren't programmed yet.
   (b) Refer to Section 4.4 on how to program the EEPROM
7. Once EEPROM is programmed or re-programmed for the correct core, reset the CPU and restart the example.
8. Set a breakpoint on `ESC_debugUpdateESCRegLogs()` in `pdi_test_app.c` and the CPU should hit the breakpoint. The pass signal should be indicated by the controlCARD LEDs. If not, pause the execution and investigate further.
9. The `ESC_debugUpdateESCRegLogs()` will continually update the `ESC_escRegs` data structure with the EtherCAT register and RAM values added for monitoring as part of `ESC_setupPDITestInterface()`. This data structure can be viewed using the CCS Expressions window.
10. You can now restart the example and set various breakpoints within `ESC_setupPDITestInterface()` to observe the reads/writes from CCS as well as the TwinCAT Master memory window. Additionally, you can change values via either interface to introduce failures in the PDI test. Refer to Section 4.5 for information on using the TwinCAT Master memory window.

## 3.2 CPU1 Echoback Demo Example

- Fail Signal (if running on controlCARD with 20MHz XTAL) - Both controlCARD LEDs (D1,D2) are flashing
- This demo example is a precompiled demonstration of the EtherCAT slave stack code.
- This demo example emulates a bank of switches (inputs) and LEDs (outputs). The EtherCAT master controls the LEDs' states and the EtherCAT slave loops back the virtual LED signals into the virtual switches so that the master can read back the LED output state.

**Note:** To view the source code and/or debug this project using CCS, refer to the CPU1 Echoback Solution Example 3.3.

1. First, TwinCAT must be installed and setup. Refer to Section 4.2.
2. Check your external connections: Section 4.1
3. Run `ethercat_slave_ssc_and_demo_setup.exe` installer to extract the demo files into the EtherCAT examples directory.
4. Within CCS, verify to be in the CCS Debug view and connect to CPU1.
5. Once connected to CPU1, go to `Run -> Load -> Load Program` and select `f2838x_cpu1_echoback_demo_FLASH.out`. Then click `Resume`.
6. Copy the ESI file (`F2838x CPU1 EtherCAT Slave.xml`) into the TwinCAT directory (Default location: `C:/TwinCAT/3.1/Config/Io/EtherCAT`). If the TwinCAT application is already opened, it must first be closed and re-opened for the ESI file to be discovered.

7. Refer to Section 4.4 on how to program the EEPROM.
8. Once EEPROM is programmed, do the following:
   (a) Disconnect and power cycle the board
   (b) Reload CPU1 application
   (c) Reconnect to board to TwinCAT, rescan for devices, and restart TwinCAT in `config mode`

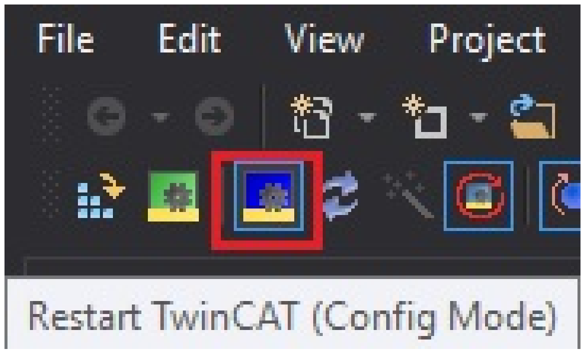


Figure 3.1: TwinCAT Restart in Config Mode Button

9. If you want to observe the variables in CCS, right-click within the CCS Expressions window and choose `Import`. Then select the `expressions_window_input_output_varaibles.txt` file.
10. Within TwinCAT, double-click on the discovered EtherCAT box and observe that the EtherCAT slave is running in OP mode.

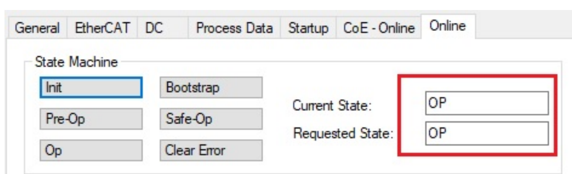


Figure 3.2: EtherCAT Slave in OP Mode

11. Within TwinCAT, expand the explorer to the EtherCAT box and find the various output/input mappings.

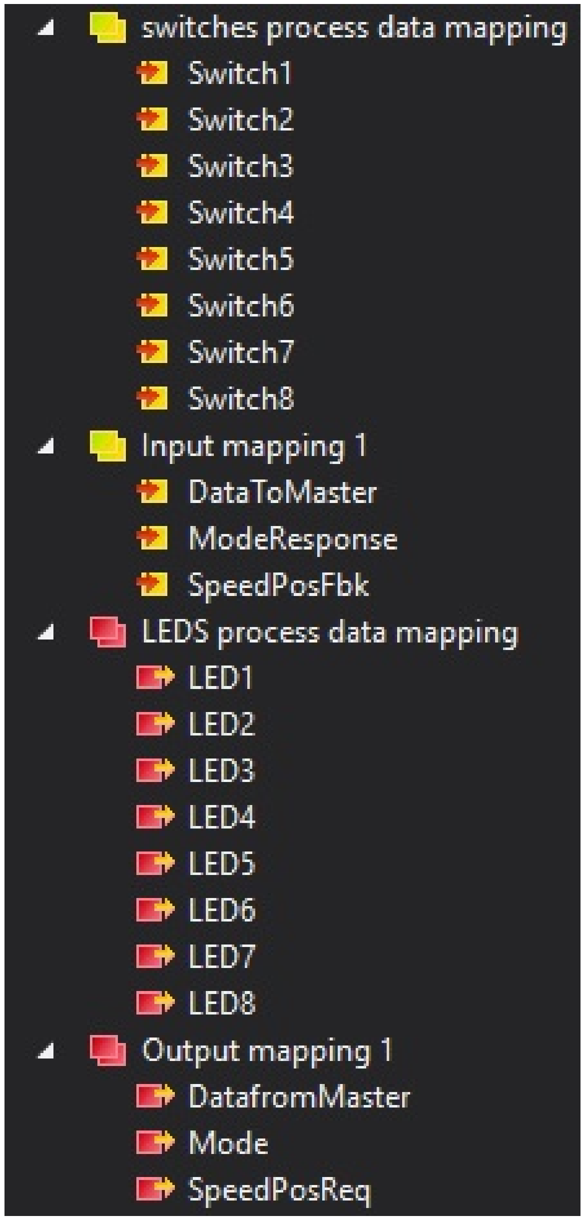


Figure 3.3: TwinCAT Solution Explorer Inputs and Outputs

(a) Select the `LEDS process data mapping` in the solution explorer and in the window on the right, you can change the value of any of the virtual LEDs. Switch to the `switches process data mapping` to see the looped back values. For example, if LED1 is set to 1, then Switch1 should also be 1.

(b) Select the `Output mapping 1` in the solution explorer and in the window on the right, you can change the values of the 3 data variables. Once set, the looped back value can be observed from the `Input mapping 1` variables.

## 3.3 CPU1 Echoback Solution Example

- This example requires application and slave stack files to be generated via the SSC tool before building/running.
- This example emulates a bank of switches (inputs) and LEDs (outputs). The EtherCAT master controls the LEDs' states and the EtherCAT slave loops back the virtual LED signals into the virtual switches so that the master can read back the LED output state.

1. First, TwinCAT must be installed and setup. Refer to Section 4.2.
2. Install the SSC tool V5.12
   (a) **Important**: Only V5.12 is supported. Only download this version.
   (b) Download at ETG SSC ET9300
3. Check your external connections: Section 4.1
4. Run `ethercat_slave_ssc_and_demo_setup.exe` installer to extract the F2838x SSC configuration and echoback application files required by the SSC tool. These will be located in the newly created `ssc_configuration` directory.
5. Open the SSC tool and a `New Project` dialog box will open. Select `Import` and locate the `f2838x_ssc_config.xml`. Then click `Open`.
6. Use the Custom drop-down menu to select `TI F2838x CPU1 Sample (Includes Sample Application)` and click `OK`.

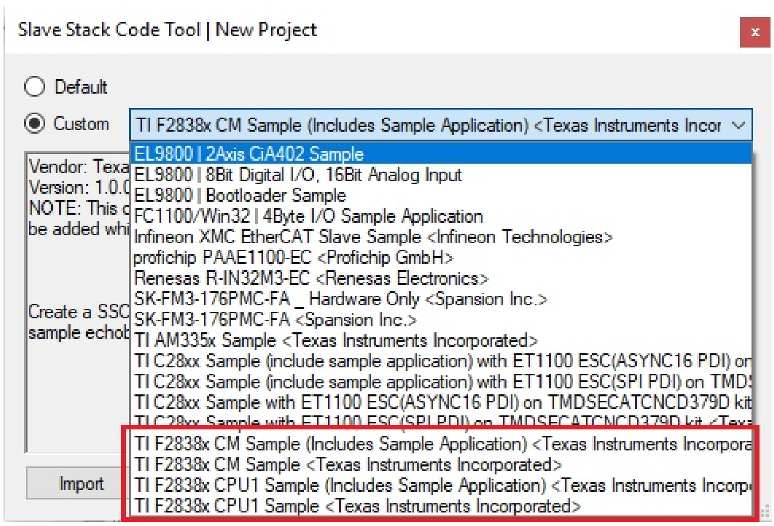


Figure 3.4: SSC Configuration Window

7. Click `Yes` when the pop up window asks about requiring external files to proceed.
8. Save the SSC project.
9. Within SSC tool, go to `Project -> Create new Slave Files`
   (a) Change the Source Folder directory to the `/examples/f2838x_cpu1_echoback_solution` directory.
   (b) Leave the ESI file directory location as is.
   (c) Click `Start` and then `OK`.

10. Import the example from `/examples/f2838x_cpu1_echoback_solution` into CCS and build it for `RAM` or `FLASH`.
11. Within CCS, verify to be in the CCS Debug view and connect to CPU1.
12. Once connected to CPU1, go to `Run -> Load -> Load Program` and select `f2838x_cpu1_echoback_solution.out`. Then click `Resume`.
13. Copy the ESI file (`F2838x CPU1 EtherCAT Slave.xml`) generated by the SSC tool into the TwinCAT directory (Default location: `C:/TwinCAT/3.1/Config/Io/EtherCAT`) If the TwinCAT application is already opened, it must first be closed and re-opened for the ESI file to be discovered.
14. Refer to Section 4.4 on how to program the EEPROM.
15. Once EEPROM is programmed, do the following:
    (a) Disconnect and power cycle the board
    (b) Reload CPU1 application
    (c) Reconnect to board to TwinCAT, rescan for devices, and restart TwinCAT in `config mode`

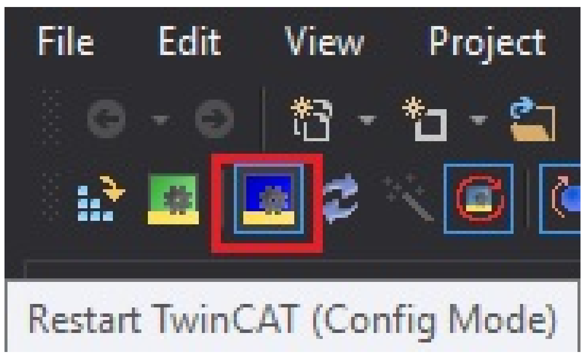


Figure 3.5: TwinCAT Restart in Config Mode Button

16. Within TwinCAT, double-click on the discovered EtherCAT box and observe that the EtherCAT slave is running in OP mode.

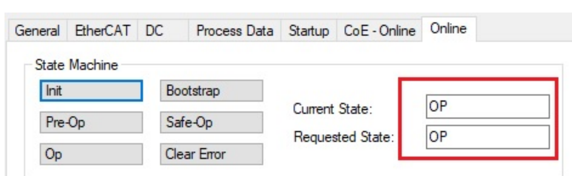


Figure 3.6: EtherCAT Slave in OP Mode

17. Within TwinCAT, expand the explorer to the EtherCAT box and find the various output/input mappings.

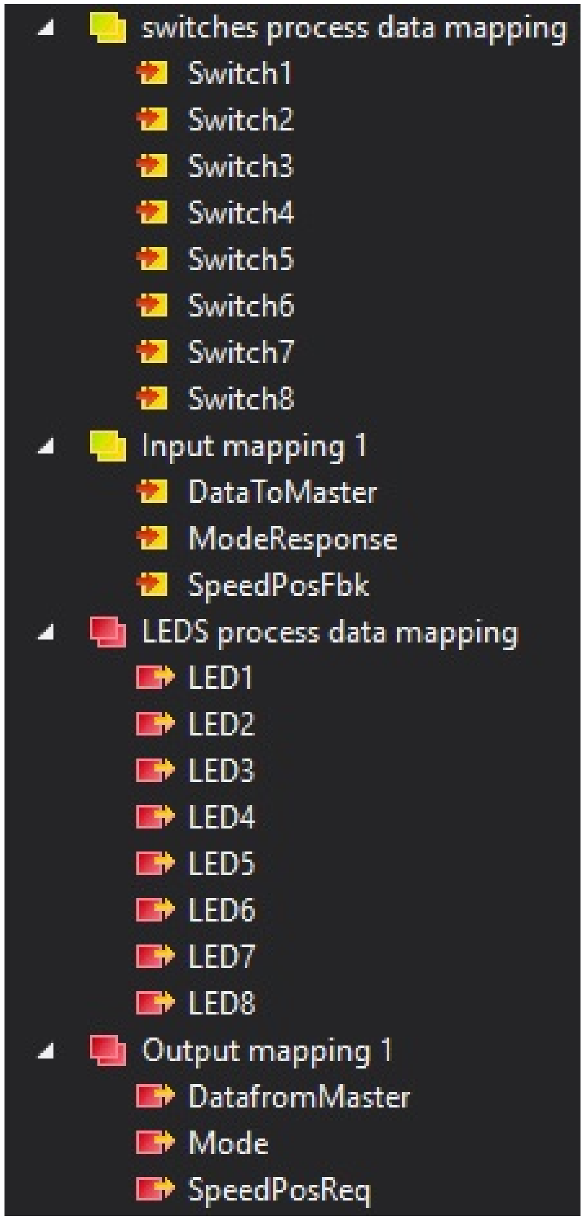


Figure 3.7: TwinCAT Solution Explorer Inputs and Outputs

(a) Select the `LEDS process data mapping` in the solution explorer and in the window on the right, you can change the value of any of the virtual LEDs. Switch to the `switches process data mapping` to see the looped back values. For example, if LED1 is set to 1, then Switch1 should also be 1.
(b) Select the `Output mapping 1` in the solution explorer and in the window on the right, you can change the values of the 3 data variables. Once set, the looped back value can be observed from the `Input mapping 1` variables.

## 3.4 CPU1 Allocate ECAT to CM Example

- This example sets up the EtherCAT required GPIOs and clocking then allocates the EtherCAT ownership to the Connectivity Manager (CM) core. Additionally, the example configures the CM clocks and releases the core to boot.

1. Check your external connections: Section 4.1
2. Open CCS and import the example `f2838x_cpu1_allocate_ecat_to_cm`
3. Select the RAM build configuration and build the example
4. Load the example to the controlCARD and run the code
   (a) **Important**: If the EtherCAT HAL example for CPU1 was loaded previously and the controlCARD hasn't been power cycled since, make sure to power cycle the controlCARD before running the CM HAL example
5. The example will perform the necessary setup and then enter an infinite loop.
6. The CM PDI HAL test example can now be run (if not already loaded) on the CM.

## 3.5 CM PDI HAL Test Example

- **Important**: The CPU1 example to allocate ECAT to the CM must be running before running this test example. (See Section 3.4)
- This example performs a series of reads and writes to full range of EtherCAT RAM using the HAL APIs. These can be observed from the CCS memory browser or TwinCAT ESC memory browser.
- This example is self-checking when performing the reads and writes. The following details the pass and fail signals:
  - Pass Signal - Both controlCARD LEDs (D1,D2) are on (not flashing)
  - Fail Signal - Both controlCARD LEDs (D1,D2) are flashing

**Note:** The intent of this project is to demonstrate the usage of the PDI. Therefore, no EtherCAT stack is included in this demo.

1. First, TwinCAT must be installed and setup. Refer to Section 4.2.
2. Check your external connections: Section 4.1
3. Open CCS and import the example `f2838x_cm_pdi_hal_test_app`
4. Select the RAM build configuration and build the example
5. Load the example to the controlCARD and run the code
6. If this your first time running any EtherCAT code (CPU1 or CM) on the controlCARD, the LEDs should be indicating a fail signal.
   (a) This failure is occurring because the minimum required EtherCAT EEPROM locations aren't programmed yet.
   (b) Refer to Section 4.4 on how to program the EEPROM
7. Once EEPROM is programmed or re-programmed for the correct core, reset the CPU and restart the example.
8. Set a breakpoint on `ESC_debugUpdateESCRegLogs()` in `pdi_test_app.c` and the CPU should hit the breakpoint. The pass signal should be indicated by the controlCARD LEDs. If not, pause the execution and investigate further.

9. The `ESC_debugUpdateESCRegLogs()` will continually update the `ESC_escRegs` data structure with the EtherCAT register and RAM values added for monitoring as part of `ESC_setupPDITestInterface()`. This data structure can be viewed using the CCS Expressions window.
10. You can now restart the example and set various breakpoints within `ESC_setupPDITestInterface()` to observe the reads/writes from CCS as well as the TwinCAT Master memory window. Additionally, you can change values via either interface to introduce failures in the PDI test. Refer to Section 4.5 for information on using the TwinCAT Master memory window.

## 3.6 CM Echoback Demo Example

- **Important**: The CPU1 example to allocate ECAT to the CM must be running before running this test example. (See Section 3.4)
- Fail Signal (if running on controlCARD with 20MHz XTAL) - Both controlCARD LEDs (D1,D2) are flashing
- This demo example is a precompiled demonstration of the EtherCAT slave stack code.
- This demo example emulates a bank of switches (inputs) and LEDs (outputs). The EtherCAT master controls the LEDs' states and the EtherCAT slave loops back the virtual LED signals into the virtual switches so that the master can read back the LED output state.

**Note:** To view the source code and/or debug this project using CCS, refer to the CM Echoback Solution Example 3.7.

1. First, TwinCAT must be installed and setup. Refer to Section 4.2.
2. Check your external connections: Section 4.1
3. Run `ethercat_slave_ssc_and_demo_setup.exe` installer to extract the demo files into the EtherCAT examples directory.
4. Within CCS, verify to be in the CCS Debug view and connect to the CM core.
5. Once connected to the CM core, go to `Run -> Load -> Load Program` and select `f2838x_cm_echoback_demo_FLASH.out`. Then click `Resume`.
6. Copy the ESI file (`F2838x CM EtherCAT Slave.xml`) into the TwinCAT directory (Default location: `C:/TwinCAT/3.1/Config/Io/EtherCAT`) If the TwinCAT application is already opened, it must first be closed and re-opened for the ESI file to be discovered.
7. Refer to Section 4.4 on how to program the EEPROM.
8. Once EEPROM is programmed, do the following:
   (a) Disconnect and power cycle the board
   (b) Reload CPU1 and CM applications
   (c) Reconnect to board to TwinCAT, rescan for devices, and restart TwinCAT in `config mode`

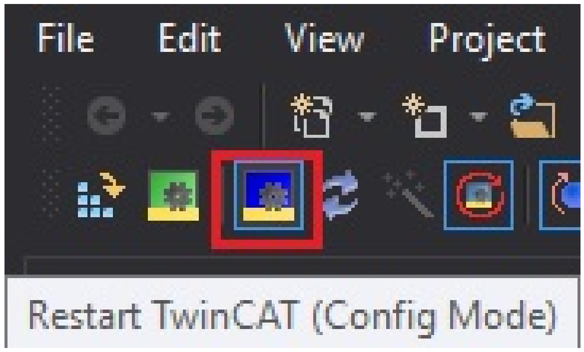


Figure 3.8: TwinCAT Restart in Config Mode Button

9. If you want to observe the variables in CCS, right-click within the CCS Expressions window and choose `Import`. Then select the `expressions_window_input_output_varaibles.txt` file.
10. Within TwinCAT, double-click on the discovered EtherCAT box and observe that the EtherCAT slave is running in OP mode.

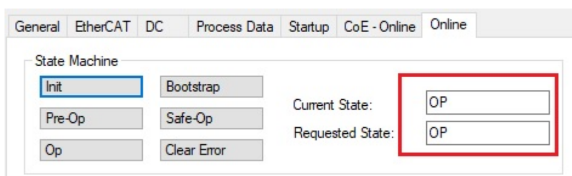


Figure 3.9: EtherCAT Slave in OP Mode

11. Within TwinCAT, expand the explorer to the EtherCAT box and find the various output/input mappings.

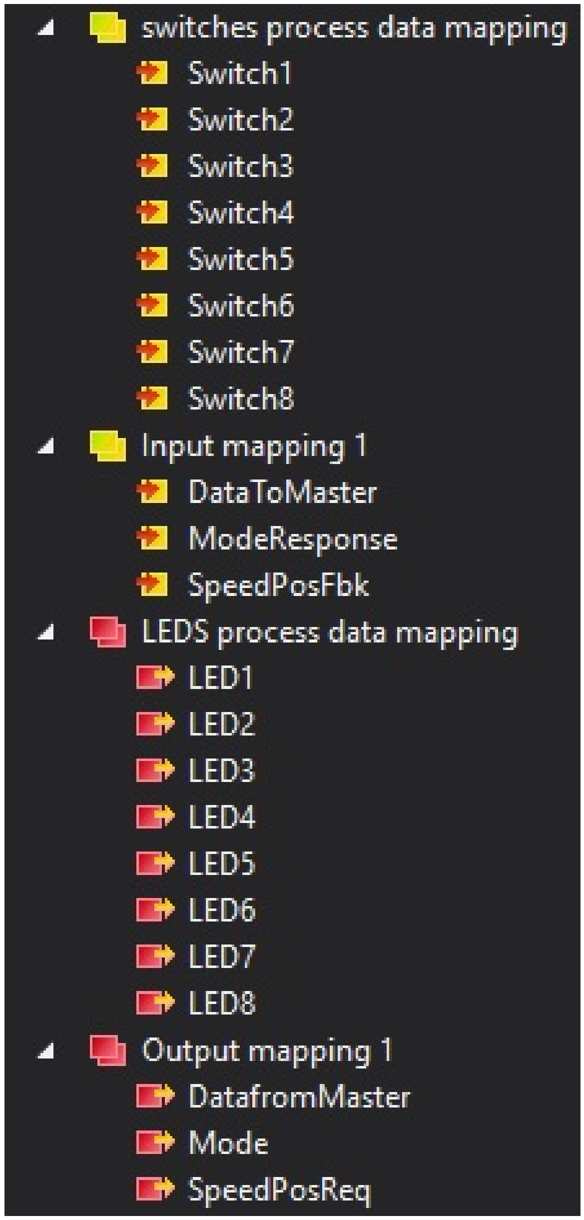


Figure 3.10: TwinCAT Solution Explorer Inputs and Outputs

(a) Select the `LEDS process data mapping` in the solution explorer and in the window on the right, you can change the value of any of the virtual LEDs. Switch to the `switches process data mapping` to see the looped back values. For example, if LED1 is set to 1, then Switch1 should also be 1.

(b) Select the `Output mapping 1` in the solution explorer and in the window on the right, you can change the values of the 3 data variables. Once set, the looped back value can be observed from the `Input mapping 1` variables.

## 3.7 CM Echoback Solution Example

- **Important**: The CPU1 example to allocate ECAT to the CM must be running before running this test example. (See Section 3.4)
- This example requires application and slave stack files to be generated via the SSC tool before building/running.
- This example emulates a bank of switches (inputs) and LEDs (outputs). The EtherCAT master controls the LEDs' states and the EtherCAT slave loops back the virtual LED signals into the virtual switches so that the master can read back the LED output state.

1. First, TwinCAT must be installed and setup. Refer to Section 4.2.
2. Install the SSC tool V5.12
   (a) **Important**: Only V5.12 is supported. Only download this version.
   (b) Download at ETG SSC ET9300
3. Check your external connections: Section 4.1
4. Run `ethercat_slave_ssc_and_demo_setup.exe` installer to extract the F2838x SSC configuration and echoback application files required by the SSC tool. These will be located in the newly created `ssc_configuration` directory.
5. Open the SSC tool and a `New Project` dialog box will open. Select `Import` and locate the `f2838x_ssc_config.xml`. Then click `Open`.
6. Use the Custom drop-down menu to select `TI F2838x CM Sample (Includes Sample Application)` and click `OK`.

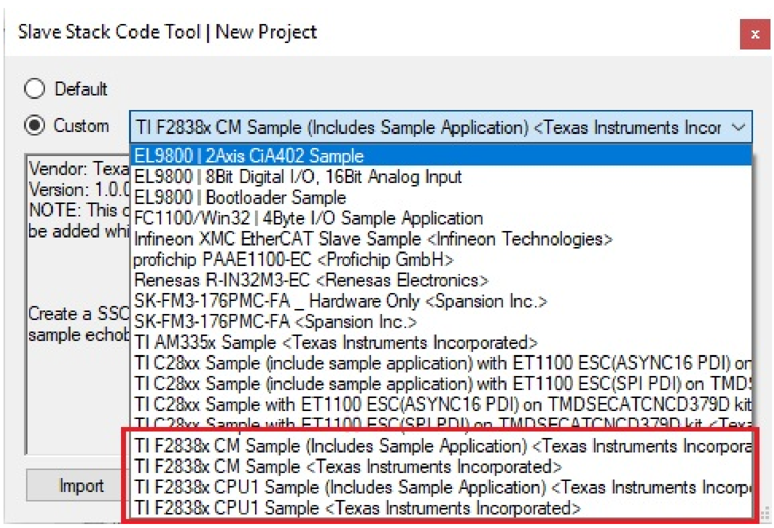


Figure 3.11: SSC Configuration Window

7. Click `Yes` when the pop up window asks about requiring external files to proceed.
8. Save the SSC project.
9. Within SSC tool, go to `Project -> Create new Slave Files`
   (a) Change the Source Folder directory to the `/examples/f2838x_cm_echoback_solution` directory.

(b) Leave the ESI file directory location as is.

(c) Click `Start` and then `OK`.

10. Import the example from `/examples/f2838x_cm_echoback_solution` into CCS and build it for `RAM` or `FLASH`.
11. Within CCS, verify to be in the CCS Debug view and connect to the CM core.
12. Once connected to the CM core, go to `Run -> Load -> Load Program` and select `f2838x_cm_echoback_solution.out`. Then click `Resume`.
13. Copy the ESI file (`F2838x CM EtherCAT Slave.xml`) generated by the SSC tool into the TwinCAT directory (Default location: `C:/TwinCAT/3.1/Config/Io/EtherCAT`) If the TwinCAT application is already opened, it must first be closed and re-opened for the ESI file to be discovered.
14. Refer to Section 4.4 on how to program the EEPROM.
15. Once EEPROM is programmed, do the following:

    (a) Disconnect and power cycle the board

    (b) Reload CPU1 and CM applications

    (c) Reconnect to board to TwinCAT, rescan for devices, and restart TwinCAT in `config mode`

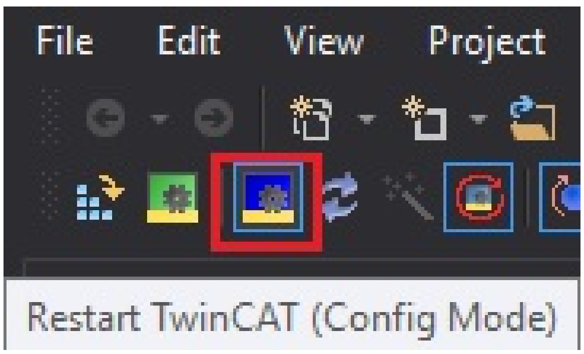


Figure 3.12: TwinCAT Restart in Config Mode Button

16. Within TwinCAT, double-click on the discovered EtherCAT box and observe that the EtherCAT slave is running in OP mode.

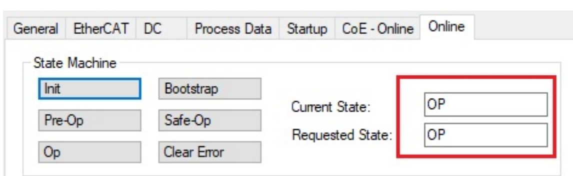


Figure 3.13: EtherCAT Slave in OP Mode

17. Within TwinCAT, expand the explorer to the EtherCAT box and find the various output/input mappings.

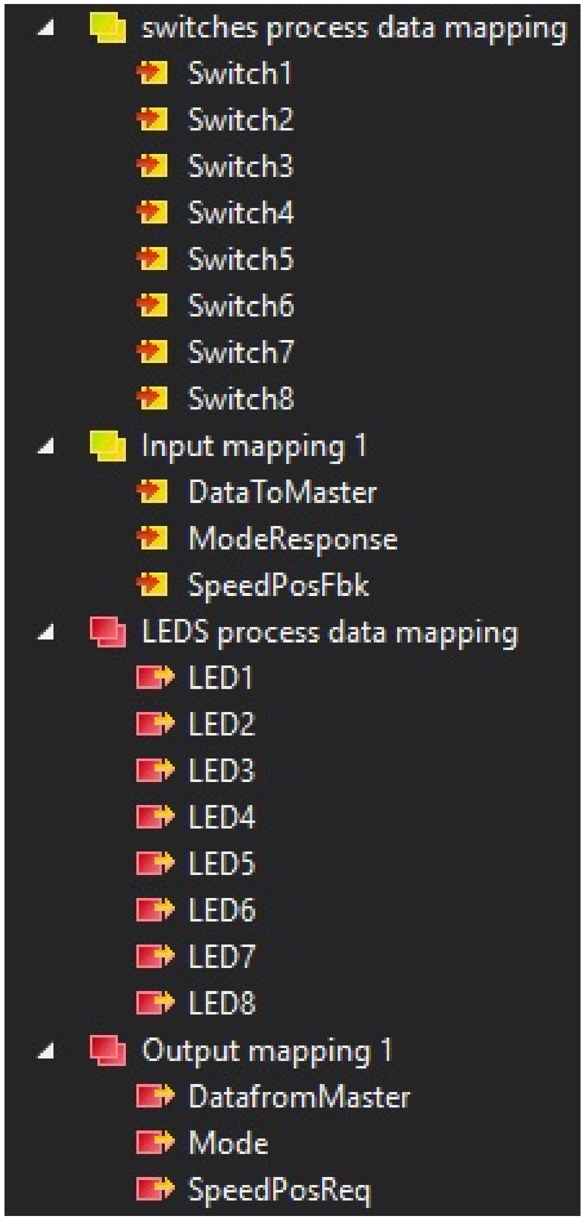


Figure 3.14: TwinCAT Solution Explorer Inputs and Outputs

(a) Select the `LEDS process data mapping` in the solution explorer and in the window on the right, you can change the value of any of the virtual LEDs. Switch to the `switches process data mapping` to see the looped back values. For example, if LED1 is set to 1, then Switch1 should also be 1.
(b) Select the `Output mapping 1` in the solution explorer and in the window on the right, you can change the values of the 3 data variables. Once set, the looped back value can be observed from the `Input mapping 1` variables.

## 3.8 CM CiA402 Solution Example

- **Important**: The CPU1 example to allocate ECAT to the CM must be running before running this test example. (See Section 3.4)
- This example requires application and slave stack files to be generated via the SSC tool before building/running.
- This example integrates the sample CiA402 application from Beckhoff.

1. First, TwinCAT must be installed and setup. Refer to Section 4.2.
2. Install the SSC tool V5.12
   (a) **Important**: Only V5.12 is supported. Only download this version.
   (b) Download at ETG SSC ET9300
3. Check your external connections: Section 4.1
4. Run `ethercat_slave_ssc_and_demo_setup.exe` installer to extract the F2838x SSC configuration and echoback application files required by the SSC tool. These will be located in the newly created `ssc_configuration` directory.
5. Open the SSC tool and a `New Project` dialog box will open. Select `Import` and locate the `f2838x_ssc_config.xml`. Then click `Open`.
6. Use the Custom drop-down menu to select `TI F2838x CM Sample` (the one WITHOUT the sample application) and click `OK`.

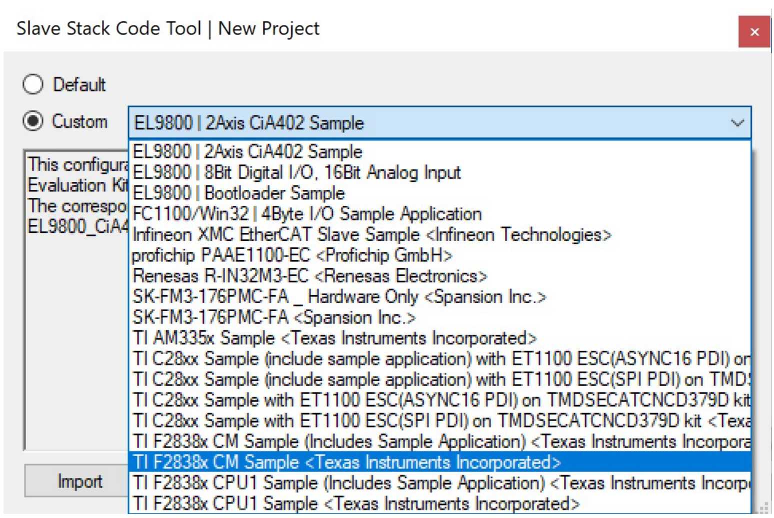


Figure 3.15: SSC Configuration Window

7. Click `Yes` when the pop up window asks about requiring external files to proceed.
8. In the Slave Project Navigation, select `Application` and for the `CiA402_DEVICE` slave setting, set the value to `1`.

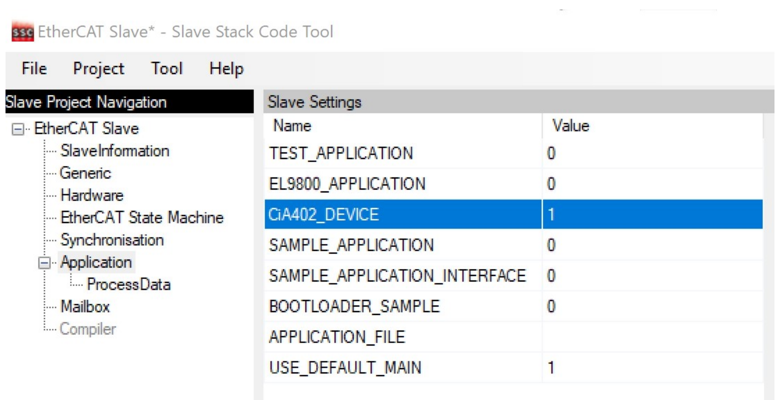


Figure 3.16: SSC Application Window

9. Save the SSC project.
10. Within SSC tool, go to `Project -> Create new Slave Files`
    (a) Change the Source Folder directory to the `/examples/f2838x_cm_cia402_solution` directory.
    (b) Leave the ESI file directory location as is.
    (c) Click `Start` and then `OK`.
11. **Important**: This example assumes structs are packed, open the generated `ecat_def.h` stack file and locate the `#define STRUCT_PACKED_END`. Set this define to `__attribute__((packed))`.
12. Import the example from `/examples/f2838x_cm_cia402_solution` into CCS and build it for `RAM` or `FLASH`.
13. Within CCS, verify to be in the CCS Debug view and connect to the CM core.
14. Once connected to the CM core, go to `Run -> Load -> Load Program` and select `f2838x_cm_cia402_solution.out`. Then click `Resume`.
15. Copy the ESI file (`F2838x CM EtherCAT Slave.xml`) generated by the SSC tool into the TwinCAT directory (Default location: `C:/TwinCAT/3.1/Config/Io/EtherCAT`) If the TwinCAT application is already opened, it must first be closed and re-opened for the ESI file to be discovered.
16. Refer to Section 4.4 on how to program the EEPROM (can follow the echoback example steps).
17. Once EEPROM is programmed, do the following:
    (a) Disconnect and power cycle the board
    (b) Reload CPU1 and CM applications
    (c) Reconnect to board to TwinCAT and rescan for devices

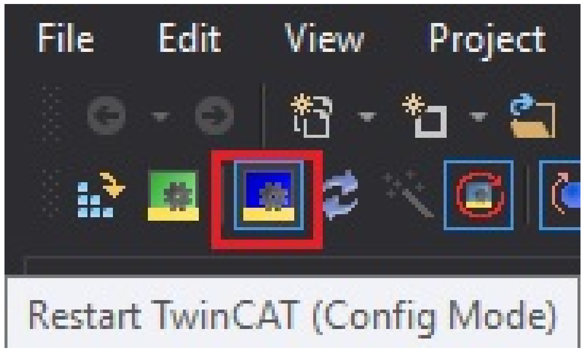


Figure 3.17: TwinCAT Restart in Config Mode Button

18. With the ESI indicating CiA402 support, TwinCAT will ask about how to append the linked axis. Select whichever makes sense for your axis.

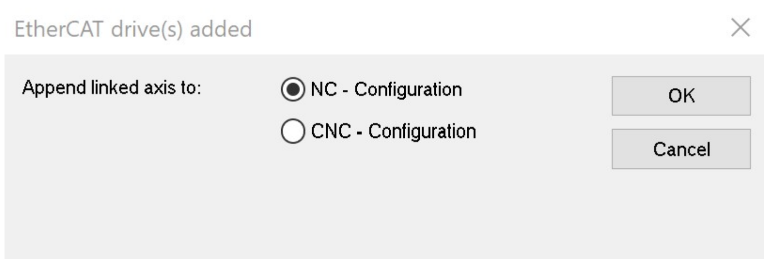


Figure 3.18: TwinCAT Drives Configuration

19. Within TwinCAT, double-click on the discovered EtherCAT box and observe that the EtherCAT slave is running in OP mode.

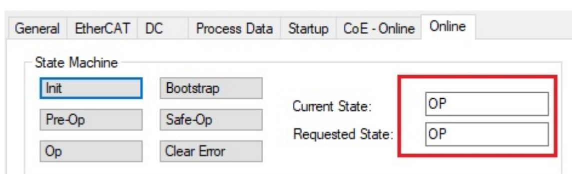


Figure 3.19: EtherCAT Slave in OP Mode

20. For further details on configuring and running this example as well as details on the CiA402 drive profile objects refer to `Chapter 10` in the following Beckhoff document: Application Note ET9300 (EtherCAT Slave Stack Code)
    - Any questions regarding the CiA402 sample implementation, post on the ETG developer forums: Link

# 4 How-To Procedures

## 4.1 Example External Connections

| Example | External Connections |
|---|---|
| CPU1 PDI HAL Test | Mini USB connection between controlCARD and computer<br>Ethernet cable connected to controlCARD RJ45 Port 0 and to computer |
| CPU1 Echoback Demo | Mini USB connection between controlCARD and computer<br>Ethernet cable connected to controlCARD RJ45 Port 0 and to computer |
| CPU1 Echoback Solution | Mini USB connection between controlCARD and computer<br>Ethernet cable connected to controlCARD RJ45 Port 0 and to computer |
| CPU1 Allocate to CM | Mini USB connection between controlCARD and computer |
| CM PDI HAL Test | Mini USB connection between controlCARD and computer<br>Ethernet cable connected to controlCARD RJ45 Port 0 and to computer |
| CM Echoback Demo | Mini USB connection between controlCARD and computer<br>Ethernet cable connected to controlCARD RJ45 Port 0 and to computer |
| CM Echoback Solution | Mini USB connection between controlCARD and computer<br>Ethernet cable connected to controlCARD RJ45 Port 0 and to computer |
| CM CiA402 Solution | Mini USB connection between controlCARD and computer<br>Ethernet cable connected to controlCARD RJ45 Port 0 and to computer |

Table 4.1: Example External Connections

## 4.2 Setup TwinCAT

1. Optional: Install Microsoft Visual Studio. This isn't required since TwinCAT will install a Visual Studio shell if no Visual Studio installation is found.
   (a) Download and install Microsoft Visual Studio
   (b) TwinCAT supports integration into Visual Studio 2010/2012/2013/2015/2017
2. Download and install TwinCAT3 from the Beckhoff
   (a) Follow the left sidebar to `Download->Software->TwinCAT 3->TE1xxx | Engineering` and select the software product `TwinCAT 3.1 eXtended Automation Engineering (XAE)`
3. Once installation is complete, verify that the TwinCAT Runtime is active
   (a) Check that the TwinCAT Config Mode icon is shown in the Windows notification panel. Right click on this icon and select `Tools->TwinCAT Switch Runtime`. From the TcSwitchRuntime window, verify that it is `active`. When active, it will only provide the option to `Deactivate`. Don't Deactivate!

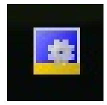

Figure 4.1: TwinCAT Config Mode Icon

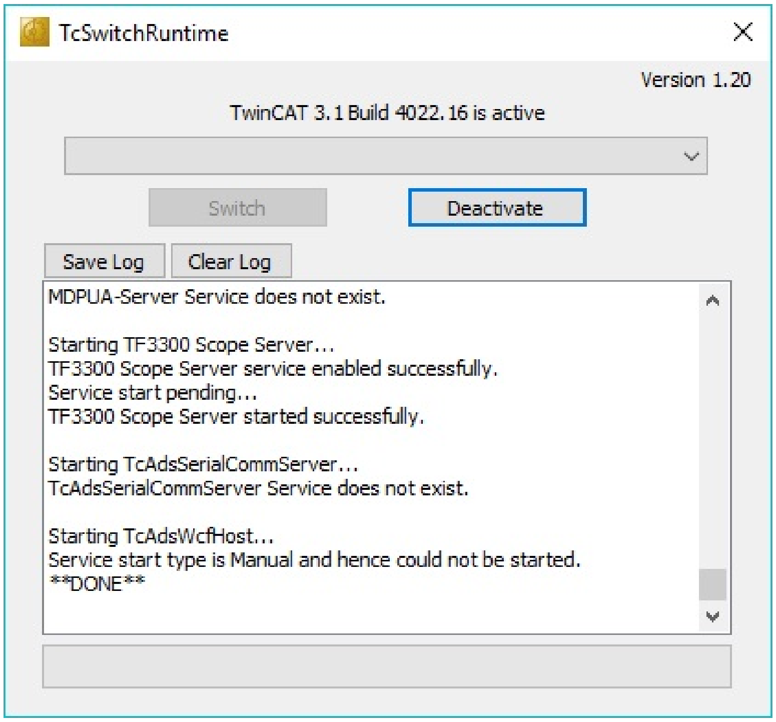


Figure 4.2: TcSwitchRuntime Window Activated

   (b) If the icon isn't present, then locate the TwinCAT Runtime executable from the file system. (Default installation location is typically: `C:/TwinCAT/TcSwitchRuntime`)

4. Start up Visual Studio with TwinCAT using one of the following methods:
   (a) Recommended: Right click the TwinCAT Config Mode icon from the Windows notification panel and select `TwinCAT XAE`
   (b) Use installed desktop icon: `TwinCAT XAE`
   (c) Use installed Start Menu icon under Beckhoff folder: `TwinCAT XAE`
5. Once Visual Studio running, verify that the main toolbar has options `TwinCAT` and `PLC` shown. If these aren't present, then the TwinCAT Switch Runtime isn't active.
6. Within Visual Studio, create a new EtherCAT project. Select `File -> New -> Project` and under templates select `TwinCAT Projects` then `TwinCAT XAE Project (XML format)`. Fill in a name and click `OK`.
7. Now that the project is created, verify that a realtime Ethernet adapter is installed.
   (a) In Visual Studio, select the `TwinCAT` menu from the main toolbar and select `Show Realtime Ethernet Compatible Devices`
   (b) In the popup window, under `Installed and ready to use devices(realtime capable)` category, if no connections are shown, select one from the list of Compatible devices and click `Install`.

8. TwinCAT setup is complete.

## 4.3 Scanning for EtherCAT Devices via TwinCAT

1. Open the TwinCAT project created via Section 4.2
2. Verify that the controlCARD is running the HAL example code and that the development computer (running TwinCAT) is connected via an Ethernet cable to the port 0 connection on the controlCARD.
   (a) Port 0 is the top Ethernet port on the side of the controlCARD with two Ethernet connections.
3. In Visual Studio on the left side solution explorer, expand the `Project`, then expand `I/O`
4. Right click on `Devices` and select `Scan`
   (a) A dialog will popup stating that `Not all types of devices can be found automatically`. Click `OK`.
5. Once scanning is complete, a popup window will appear. The following options may appear:
   (a) A popup stating that `1 new I/O devices found` where the device is `Device 2 (EtherCAT Automation Protocol)`. This or any other device numbers besides `Device 1` is correct, click `OK`.
   (b) A popup stating that no devices have been found or stating that `1 new I/O devices found` where the device is `Device 1 (EtherCAT Automation Protocol)`. This means some setup is incorrect. Verify that the example is running on the device (or at least has gone through the GPIO setup and reset of the EtherCAT IP). Then check the Section 5 for troubleshooting.
6. After clicking `OK`, another popup will ask to `Scan for boxes`. Click `Yes`.
7. After clicking `YES`, another popup will ask to `Activate Free Run`. Click `Yes`.
8. In the solution explorer on the left, under devices you should see `Device 2 (EtherCAT)`. Under that, there will be a `Box #`. This `Box` is the controlCARD ESC.

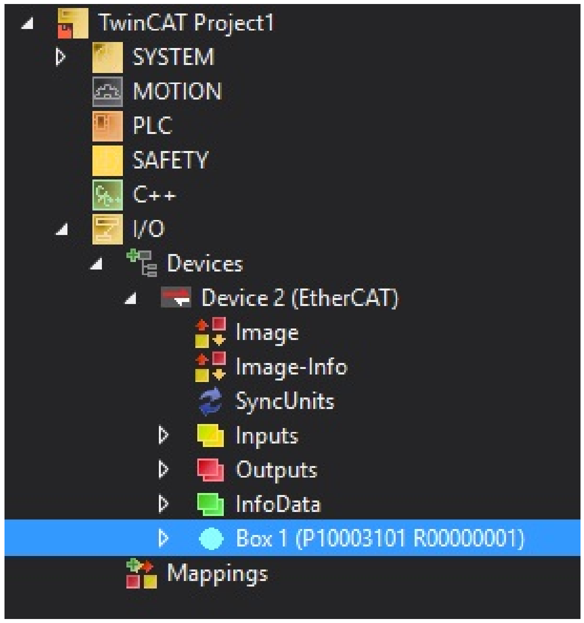


Figure 4.3: TwinCAT Solution Explorer

9. The EtherCAT master communication is now setup with the slave device.

## 4.4 Program ControlCard EEPROM

Verify first that TwinCAT has discovered the ESC (See Section 4.3 for steps)

1. In the Visual Studio solution explorer, double click on `Box #` under `Device 2 (EtherCAT).`
2. The TwinCAT project window should be open to the right of the solution explorer and have some tabs such as `General`, `EtherCAT`, etc
3. Select the `EtherCAT` tab and then click on `Advanced Settings`
4. In the new window, expand the `ESC Access` menu, then expand the `E2PROM` menu. Click on `Smart View`
5. If running a HAL PDI test Example, follow these steps to begin programming the EEPROM. Otherwise, skip to the next step.
   (a) Click on `Write E2PROM` and select `Browse`. Browse to `/C2000ware_X_XX_XX_XX/libraries/communications/ethercat/f2838x/eeprom` and select `f2838x_cpu1_pdi_test_app.bin` if running the CPU1 example or select `f2838x_cm_pdi_test_app.bin` if running the CM example. Click `OK`.
      i. Note that these BIN files only program the required first 15 bytes of EEPROM and should only be used with the HAL examples.
      ii. Additionally, either CPU1 or CM HAL example will work with either of the EEPROM files provided but for identification purposes two are provided so from the TwinCAT master, the user can identify which core is controlling the EtherCAT IP.
6. If running an Echoback example, follow these steps to begin programming the EEPROM.
   (a) Click on `Write E2PROM` and expand the `Texas Instruments Incorporated` menu within the `Available EEPROM Descriptions` window.

   (b) Expand `TI C28xx Slave Devices` and select `F2838x CM EtherCAT Slave`. Click `OK`.

7. Visual Studio will indicate that the EEPROM is being programmed. When it completes, if the Smart View doesn't automatically update with the new contents, you can select `Read E2PROM` to read back the newly programmed values.
8. The Product Code for CPU1 is `0x10003201` and the Product Code for CM is `0x10003101`

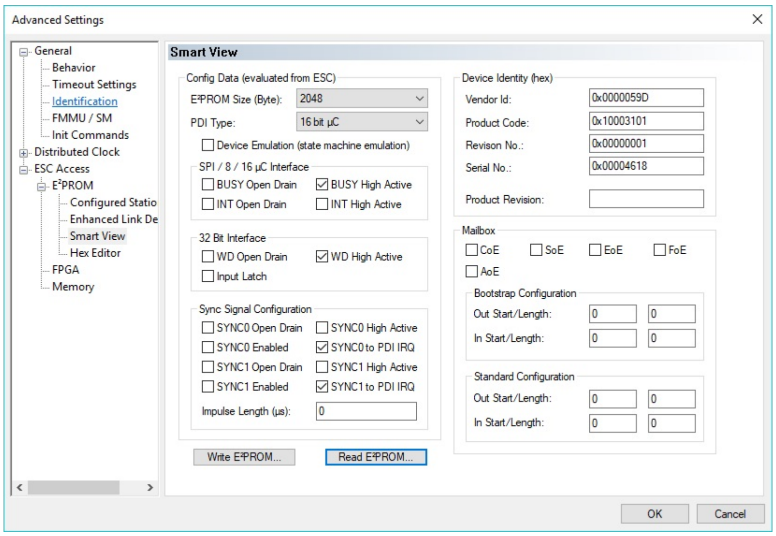


Figure 4.4: TwinCAT EEPROM Window

## 4.5 Use TwinCAT Memory Window

Verify first that TwinCAT has discovered the ESC (See Section 4.3 for steps)

1. In the Visual Studio solution explorer, double click on `Box #`.
2. The TwinCAT project window should be open and have some tabs such as `General`, `EtherCAT`, etc
3. Select the `EtherCAT` tab and then click on `Advanced Settings`
4. In the new window, expand the `ESC Access` menu, then select `Memory`. This is the connected ESC memory.

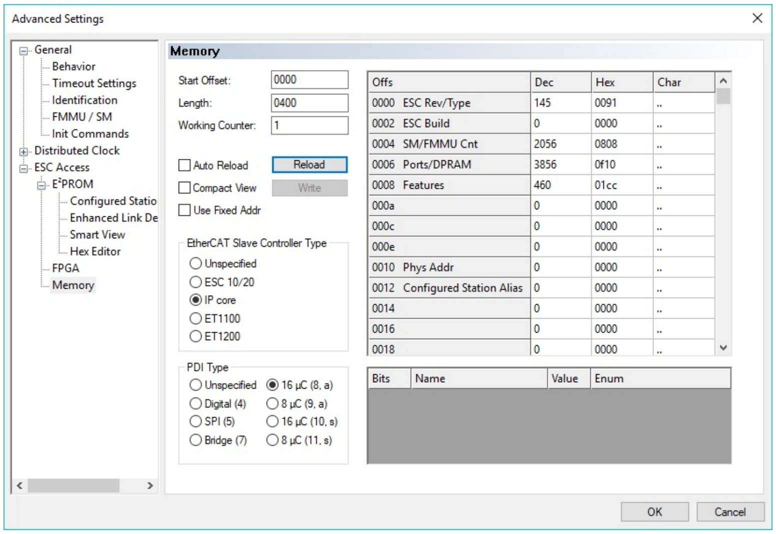


Figure 4.5: TwinCAT Memory Window

5. Adjust the `Start Offset` and `Length` as necessary to view the ESC registers or RAM. Note that these are byte offsets.
   (a) ESC registers are `0x0` to `0xFFF`
   (b) ESC RAM is `0x1000` to `0x4FFF`
6. You can select `Reload` once the offsets are changed or if the ESC is changing memory that needs to be reflected here on the Master side.
7. Additionally, the memory values can be manipulated through this window and can be applied once the `Write` button is selected. Such changes can be confirmed by viewing the same memory through the CCS memory browser.

## 4.6 Generate Slave Stack Code

These are steps to generate slave stack code without an application. If you are looking for instructions on generating for a specific F2838x EtherCAT example, refer to Getting Started Chapter 3.

1. Install the SSC tool V5.12
   (a) **Important**: Only V5.12 is supported. Only download this version.
   (b) Download at ETG SSC ET9300
2. Run `ethercat_slave_ssc_and_demo_setup.exe` installer to extract the F2838x SSC configuration and device system files required by the SSC tool. These will be located in the newly created `ssc_configuration` directory.
3. Open the SSC tool and a `New Project` dialog box will open. Select `Import` and locate the `f2838x_ssc_config.xml`. Then click `Open`.
4. Use the Custom drop-down menu to select `TI F2838x CPU1 Sample` or `TI F2838x CM Sample` and click `OK`.

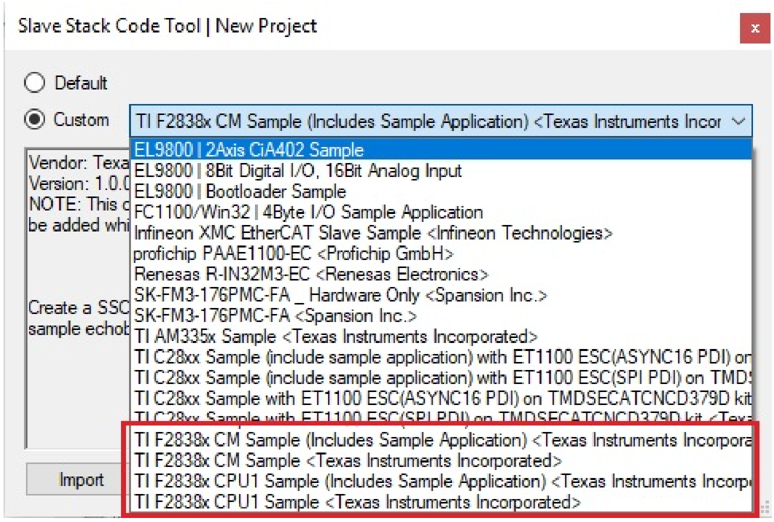


Figure 4.6: SSC Configuration Window

5. Click `Yes` when the pop up window asks about requiring external files to proceed.
6. Save the SSC project.
7. Within SSC tool, go to `Project -> Create new Slave Files`
   (a) You can leave both source and ESI directory paths as default
   (b) Click `Start` and then `OK`.

# 5 Troubleshooting

This chapter details some common issues that can cause the user trouble when using TwinCAT or the examples.

**Problem: "Example won't import into Code Composer Studio (CCS)"**
**Solutions:**

- Verify that you have the latest `C2000 Device Support Package` installed within CCS. In CCS go `Help` -> `Check for Updates`.
- Verify that you have the minimum required compiler versions installed for both C2000 and ARM. In CCS go `Help` -> `Install Code Generation Compiler Tools`.
- If you've previously imported an example into CCS, deleted it, and can't import it again, verify that the example is completely deleted from the CCS workspace (not just the CCS Project Explorer).

**Problem: "EtherCAT network fails to initialize when running TwinCAT"**
(This can include: "Reload Devices" fails, "Scan" for devices fails, "Restart EtherCAT in config mode" fails)
**Solutions:**

- Power-cycle the controlCARD
- Confirm that the EtherCAT example is loaded and running
- Verify that a realtime Ethernet adapter is installed
  - In Visual Studio, select the `TwinCAT` menu from the main toolbar and select `Show Realtime Ethernet Compatible Devices`
  - In the popup window, under `Installed and ready to use devices(realtime capable)` category, if no connections are shown, select one from the list of Compatible devices and click `Install`.

**Problem: "Example is getting stuck when attempting to enable ESCSS debug access"**
**Solutions:**

- Power-cycle the controlCARD. This problem will occur when previously running EtherCAT from one core and then trying to run EtherCAT from another core without power-cycle.

**Problem: "The EEPROM and slave stack examples are loaded but device won't go to OP mode"**
**Solutions:**

- Restart TwinCAT in config mode

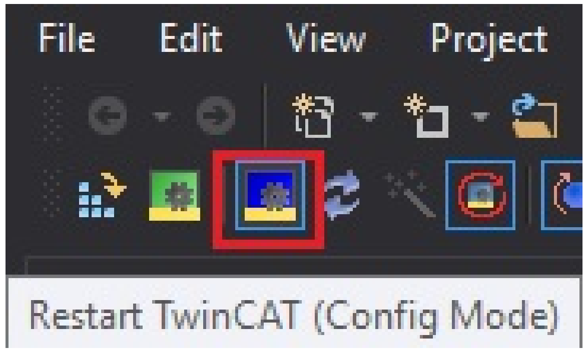


Figure 5.1: TwinCAT Restart in Config Mode Button

- Power-cycle the controlCARD, restart TwinCAT, and re-scan

# 6 Using Acontis EtherCAT Master

This chapter details how to use Acontis EC-Engineer as the EtherCAT master.

The chapter includes the following:

**6.1 EC-Engineer: Installation and Setup**
**6.2 EC-Engineer: Add ESI File**
**6.3 EC-Engineer: CPU1 Echoback Demo Example**
**6.4 EC-Engineer: Program ControlCard EEPROM**
**6.5 EC-Engineer: Viewing ESC Registers**

## 6.1 EC-Engineer: Installation and Setup

Important: If you have installed Beckhoff TwinCAT master, make sure to disable the Ethernet port TwinCAT drivers via the network adapter properties before using EC-Engineer master.

1. Download and install EC-Engineer (Acontis EtherCAT master) from Acontis Technologies.
2. Additionally, EC-Engineer requires Npcap.
   (a) Download from Npcap.
   (b) Make sure to install Npcap with WinPcap compatibility mode

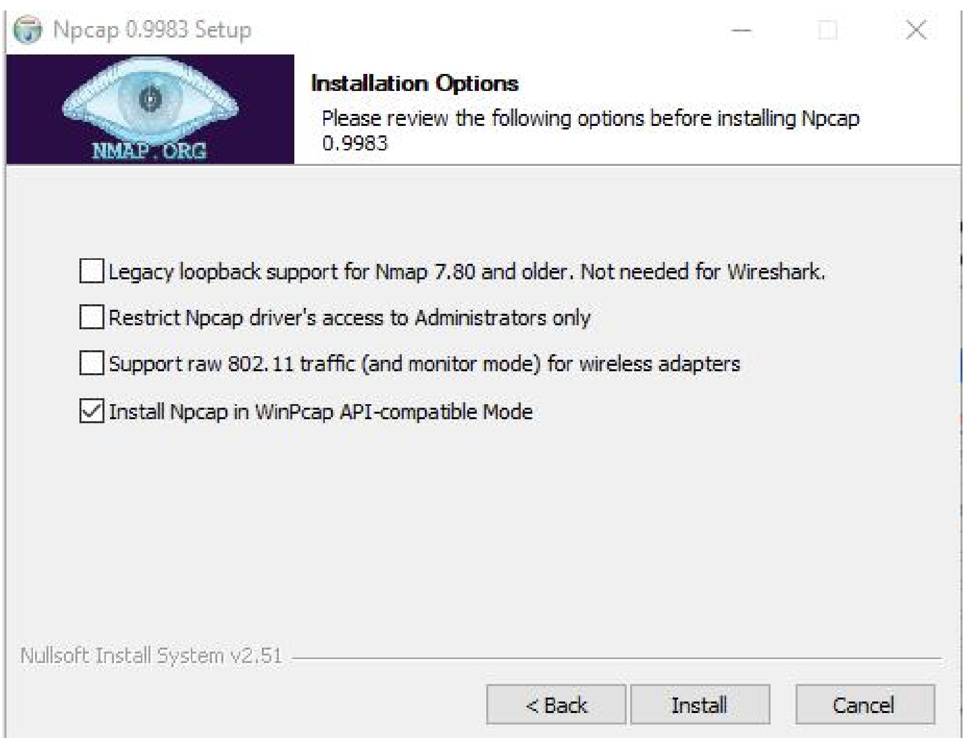


Figure 6.1: EC-Engineer Npcap installation

3. Before continuing, make sure the controlCARD running one of the EtherCAT examples is connected to the computer.
4. Start EC-Engineer and select `Online Configuration` from the `Start Page`.
5. In the `Select Master Unit Dialog`, select `EtherCAT Master Unit (Class A)`.
6. In the `Master` tab, change the `Cycle Time` to 10000, and use the `Network Adapter` drop-down to select the appropriate network adapter that is connected to the EtherCAT network.

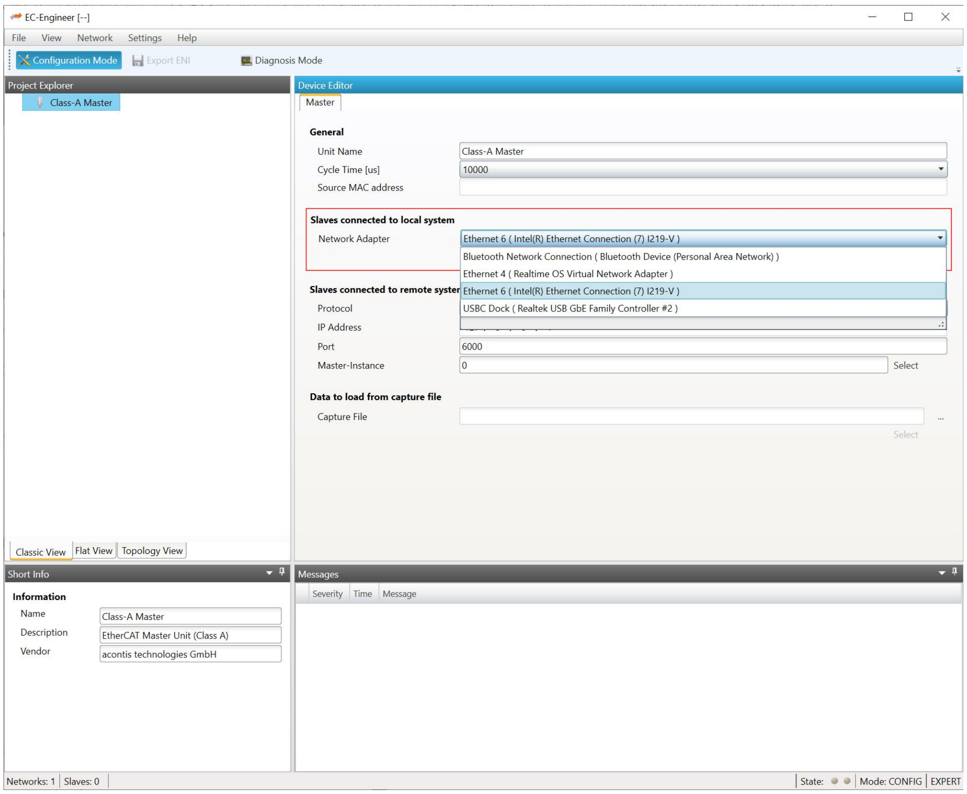


Figure 6.2: EC-Engineer Master Network Setup

7. EC-Engineer will now automatically perform a scan and find any connected slave devices.
8. In the `Project Explorer` window, the slave node should show up under `Class-A Master`.
9. Select the top `Diagnosis Mode` button to start up the online communications.
10. If the ESI file isn't added to EC-Engineer yet, then a pop-up notification will appear about a pending error state. This can be ignored until ESI is loaded. This is because only certain communications are possible until an ESI is loaded.
    (a) See Section 6.2 for steps on adding ESI file
11. The EtherCAT master communications is now setup with the slave device.

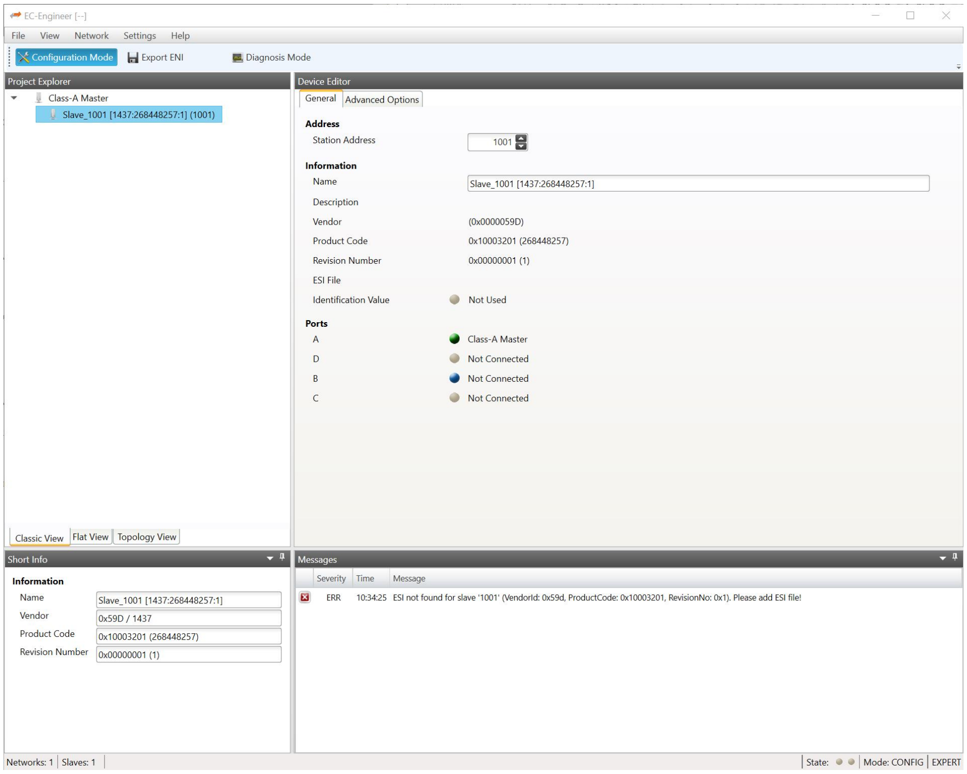


Figure 6.3: EC-Engineer Slave Identified

## 6.2 EC-Engineer: Add ESI File

1. Open EC-Engineer and select `File -> ESI Manager`.

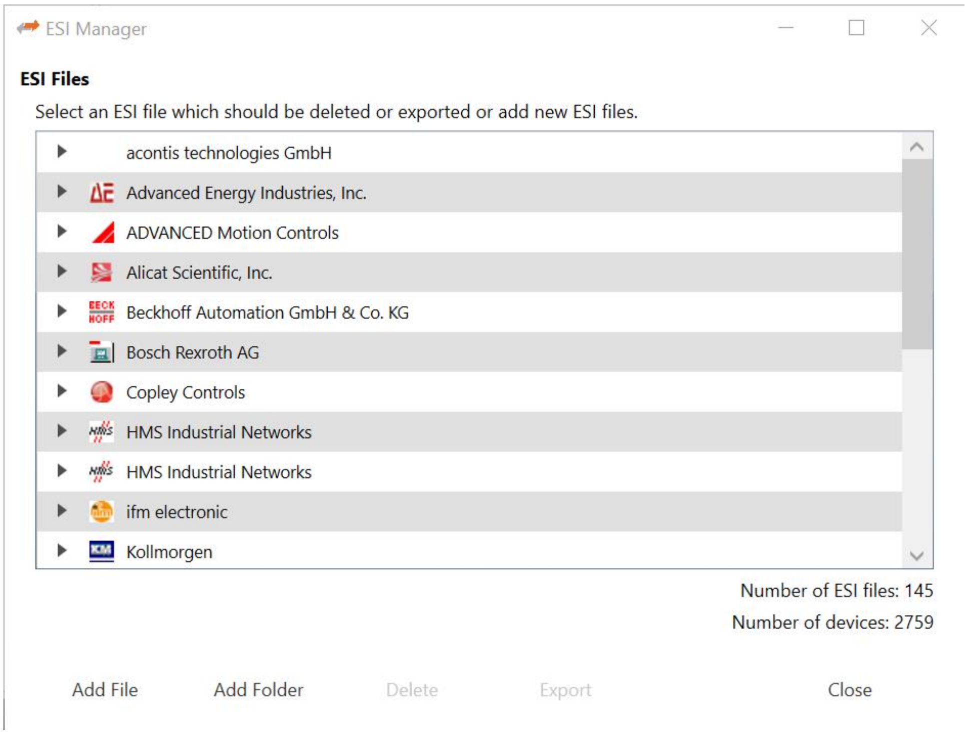


Figure 6.4: EC-Engineer EEPROM Tab

2. Check on `Add File` and browse to the ESI XML file.
3. The ESI file has now been added to the EC-Engineer ESI database.

## 6.3 EC-Engineer: CPU1 Echoback Demo Example

This details how to use the EtherCAT slave CPU1 Echoback example with EC-Engineer master. Similar steps can be followed for using the EtherCAT slave CM Echoback example.

- This demo example is a precompiled demonstration of the EtherCAT slave stack code.
- This demo example emulates a bank of switches (inputs) and LEDs (outputs). The EtherCAT master controls the LEDs' states and the EtherCAT slave loops back the virtual LED signals into the virtual switches so that the master can read back the LED output state.

1. First, EC-Engineer must be installed and setup. Refer to Section 6.1.
2. Check your external connections: Section 4.1
3. Run `ethercat_slave_ssc_and_demo_setup.exe` installer to extract the demo files into the EtherCAT examples directory.
4. Within CCS, verify to be in the CCS Debug view and connect to CPU1.
5. Once connected to CPU1, go to `Run -> Load -> Load Program` and select `f2838x_cpu1_echoback_demo_FLASH.out`. Then click `Resume`.
6. Follow these steps to add the ESI file to EC-Engineer. Refer to Section 6.2.
7. Refer to Section 6.4 on how to program the EEPROM.
8. Once EEPROM is programmed, do the following:
   (a) Disconnect and power cycle the board
   (b) Reload CPU1 application
   (c) Reconnect to board to EC-Engineer, start a new Online Configuration, and rescan the EtherCAT Network.

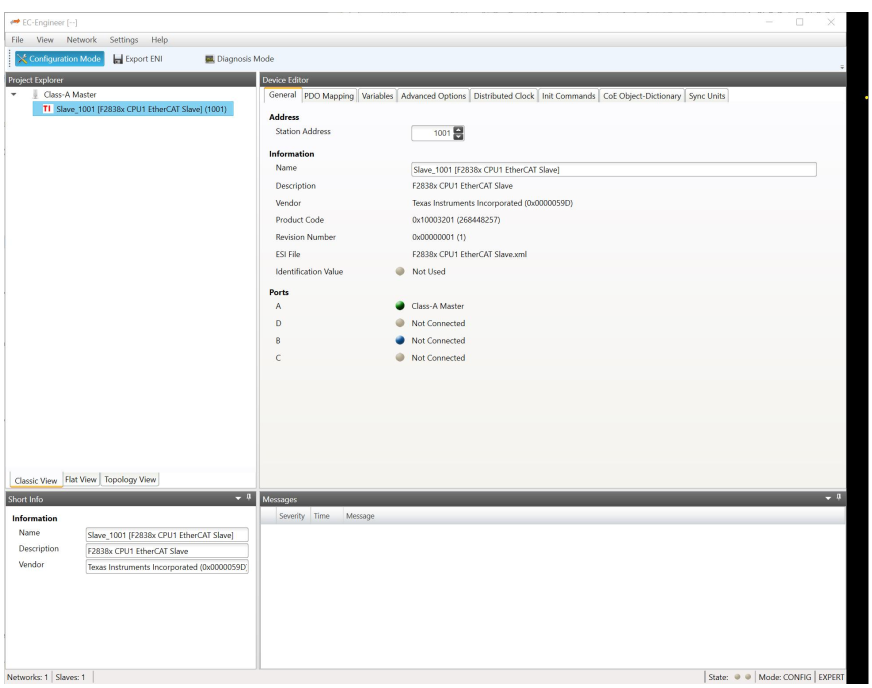


Figure 6.5: EC-Engineer Discover Slave

9. If you want to observe the variables in CCS, right-click within the CCS Expressions window and choose `Import`. Then select the `expressions_window_input_output_varaibles.txt` file.
10. Select the `Diagnosis Mode` button to start communications with the slave.
11. Select `Yes` to set the state to `OPERATIONAL`.

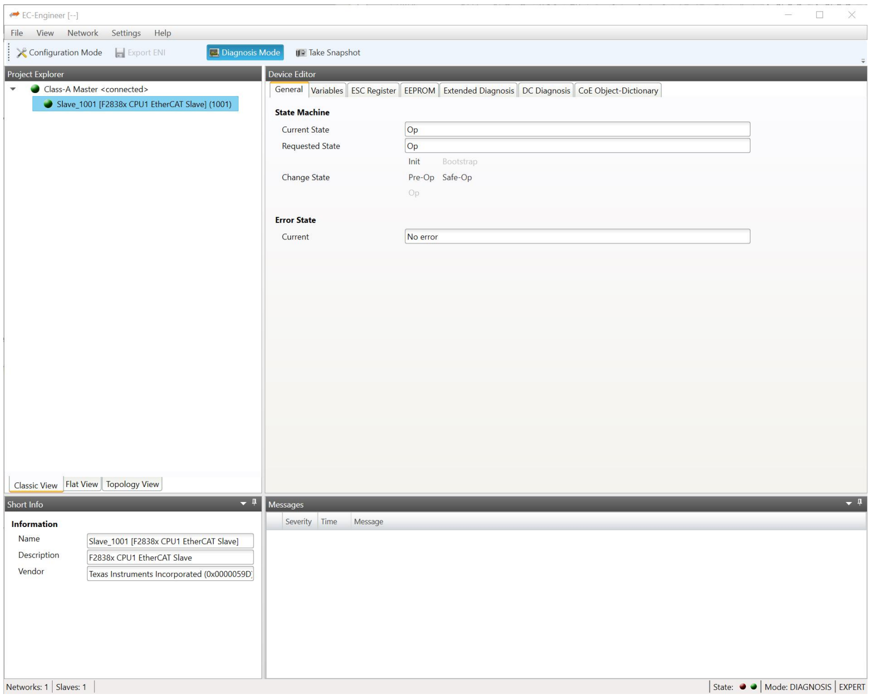

Figure 6.6: EtherCAT Slave in OP Mode

12. Within the `Device Editor` on the `General` tab, verify that the icons for the Master and Slave are green for `OPERATIONAL`.

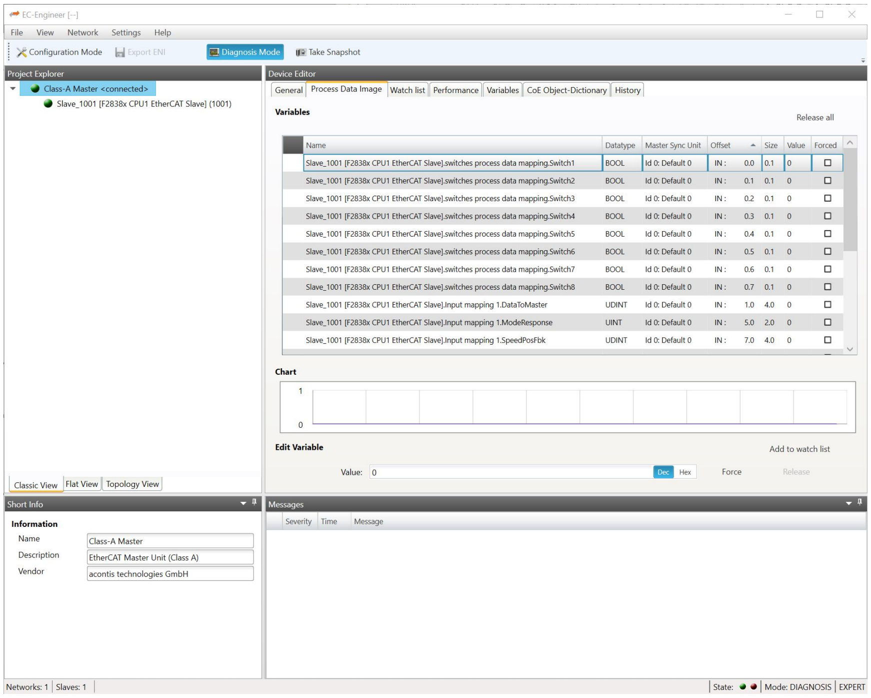

Figure 6.7: EC-Engineer Process Data Inputs and Outputs

13. Select the Master device from the `Project Explorer` window and then select the `Process Data Image` tab in the `Device Editor` window. Here you can see all of the process data object mappings for input and output data.
    (a) You can change the value of any of the virtual LED variables. Select one of the LEDS in the output variable data, change the value, and then select the `Force` button. You can observe the change that occurred in the switches process data mapping. For example, if LED1 is set to 1, then Switch1 should also be 1.
    (b) You can also change the value of the Output mapping 1 process data. Once set, the looped back value can be observed from the Input mapping 1 variables.

## 6.4 EC-Engineer: Program ControlCard EEPROM

Verify first that Acontis EC-Engineer has discovered the ESC (See Section 6.1 for steps) and the ESI file has been added (See Section 6.2 for steps).

1. In the EC-Engineer window, select the `Diagnosis Mode` button on the top bar.
2. Select the slave device in the `Project Explorer`, then click on the `EEPROM` tab within the `Device Editor` window and select the `Hex View`.

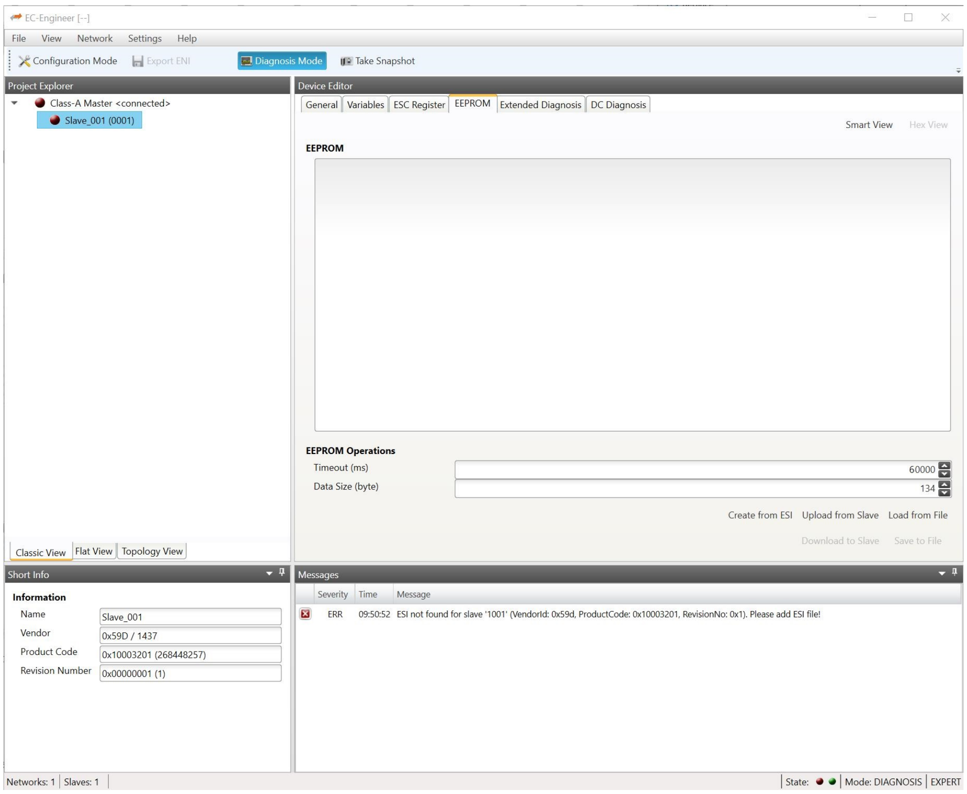


Figure 6.8: EC-Engineer EEPROM Tab

3. Select `Create from ESI` button in bottom right of `Device Editor` window and select the ESI file.
4. Select `Download to Slave`. Note: If a timeout error occurs, change the Timeout value to `60000`. Click `Upload from Slave` to confirm data was written.
5. Select `Smart View` and observe the programmed EEPROM.

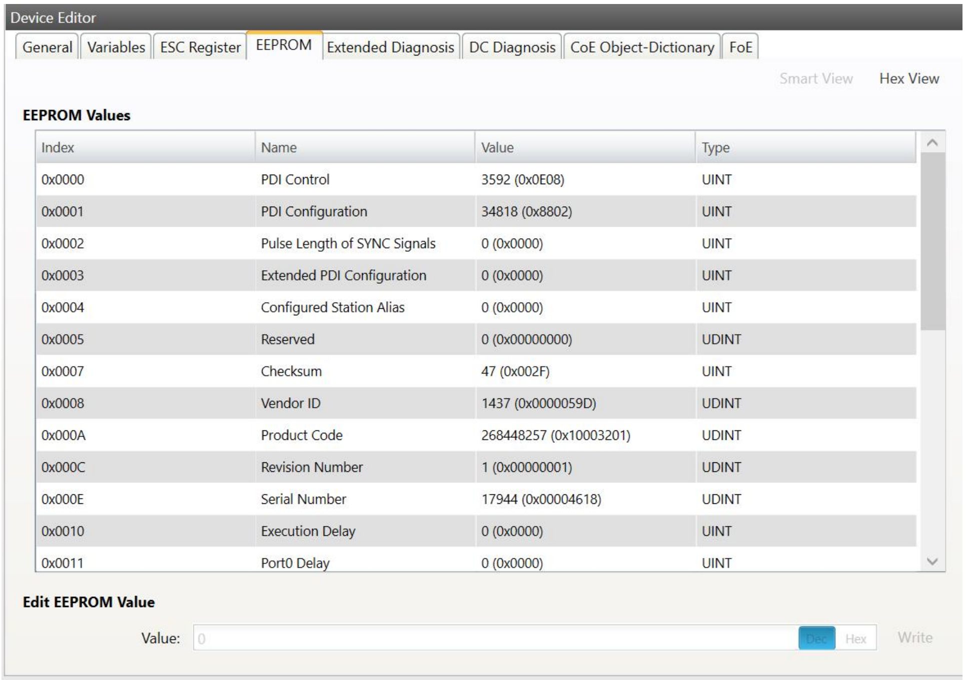


Figure 6.9: EC-Engineer EEPROM Data

## 6.5 EC-Engineer: Viewing ESC Registers

1. Once EC-Engineer is open and slave node is discovered, select the `Diagnosis Mode` button.
2. Select the slave node in the `Project Explorer` window and select the `ESC Register` tab in the `Device Editor` window.

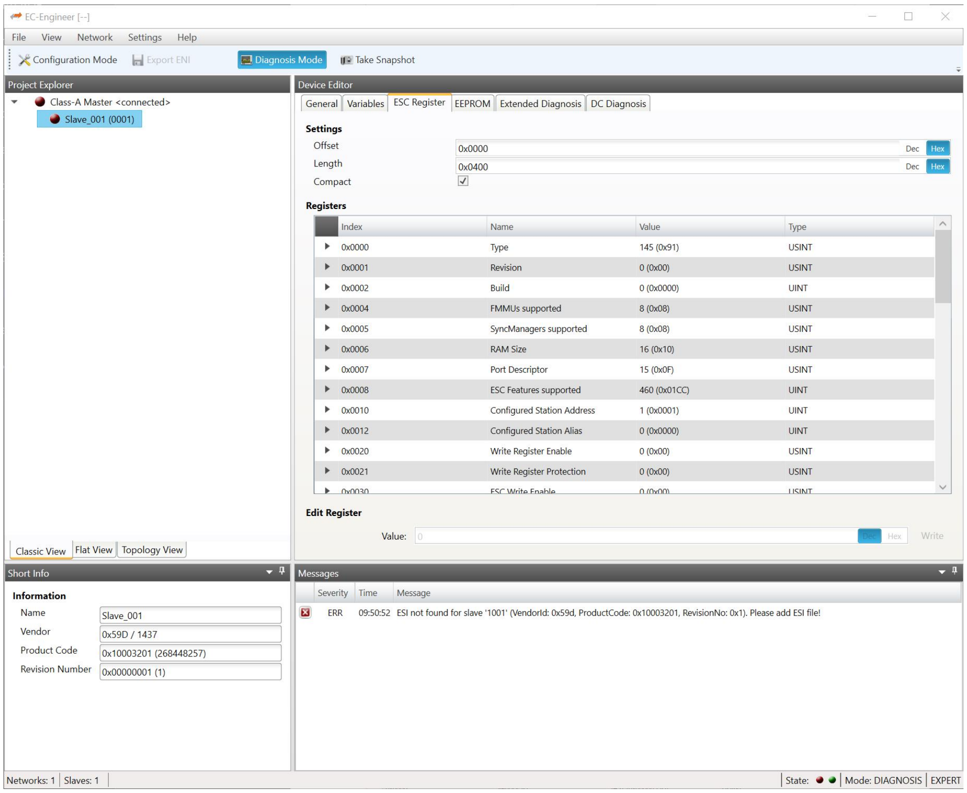

Figure 6.10: EC-Engineer EEPROM Data

3. Here you can view all the ESC registers on the slave node. Note that the offsets are in bytes.
   (a) ESC registers are 0x0 to 0xFFF
   (b) ESC RAM is 0x1000 to 0x4FFF
4. The ESC memory will reload every 30 seconds or when you change the offset or length.
5. Additionally, the memory values can be manipulated through this window and will be applied once the `Write` button is selected.

# 7 ESC HAL APIs

This chapter details the CPU1 and CM ESC HAL APIs.

## 7.1 CPU1 HAL APIs

### Functions

- __interrupt void ESC_applicationLayerHandler (void)
- __interrupt void ESC_applicationSync0Handler (void)
- __interrupt void ESC_applicationSync1Handler (void)
- void ESC_clearTimer (void)
- void ESC_debugAddESCRegsAddress (uint16_t address)
- void ESC_debugInitESCRegLogs (void)
- void ESC_debugUpdateESCRegLogs (void)
- uint32_t ESC_getTimer (void)
- void ESC_holdESCInReset (void)
- uint16_t ESC_initHW (void)
- uint16_t ESC_loadedCheckEEPROM (void)
- void ESC_passFailSignalSetup (void)
- void ESC_readBlock (ESCMEM_ADDR ∗pData, uint16_t address, uint16_t len)
- void ESC_readBlockISR (ESCMEM_ADDR ∗pData, uint16_t address, uint16_t len)
- uint32_t ESC_readDWord (uint16_t address)
- uint32_t ESC_readDWordISR (uint16_t address)
- uint16_t ESC_readWord (uint16_t address)
- uint16_t ESC_readWordISR (uint16_t address)
- void ESC_releaseESCReset (void)
- void ESC_releaseHW (void)
- void ESC_resetESC (void)
- void ESC_setLed (uint16_t runLed, uint16_t errLed)
- void ESC_setupPDITestInterface (void)
- void ESC_signalFail (void)
- void ESC_signalPass (void)
- uint32_t ESC_timerIncPerMilliSec (void)
- void ESC_writeBlock (ESCMEM_ADDR ∗pData, uint16_t address, uint16_t len)
- void ESC_writeBlockISR (ESCMEM_ADDR ∗pData, uint16_t address, uint16_t len)
- void ESC_writeDWord (uint32_t dWordValue, uint16_t address)
- void ESC_writeDWordISR (uint32_t dWordValue, uint16_t address)
- void ESC_writeWord (uint16_t wordValue, uint16_t address)
- void ESC_writeWordISR (uint16_t wordValue, uint16_t address)

## 7.1.1 Function Documentation

### 7.1.1.1 ESC_applicationLayerHandler

Application Layer Handler

**Prototype:**

```
__interrupt void
ESC_applicationLayerHandler(void)
```

**Description:**
This function is the interrupt handler for EtherCAT application/PDI interrupts.

**Returns:**
None

### 7.1.1.2 ESC_applicationSync0Handler

Application Sync 0 Handler

**Prototype:**

```
__interrupt void
ESC_applicationSync0Handler(void)
```

**Description:**
This function is the interrupt handler for EtherCAT SYNC0 interrupts.

**Returns:**
None

### 7.1.1.3 ESC_applicationSync1Handler

Application Sync 1 Handler

**Prototype:**

```
__interrupt void
ESC_applicationSync1Handler(void)
```

**Description:**
This function is the interrupt handler for EtherCAT SYNC1 interrupts.

**Returns:**
None

### 7.1.1.4 ESC_clearTimer

Clears the Timer Value

**Prototype:**

```
void
ESC_clearTimer(void)
```

**Description:**
This function resets the timer counter.

**Returns:**
None

#### 7.1.1.5 ESC_debugAddESCRegsAddress

Adds ESC Register Address to be read to RAM for Debug

**Prototype:**
```
void
ESC_debugAddESCRegsAddress(uint16_t address)
```

**Parameters:**
***address*** is the ESC register or memory byte address that needs to be read

**Description:**
This function is optional for the non-HAL API Test application use cases and is available for applications or users to add a register to the pre-set array of registers that are ready by the ESC_debugUpdateESCRegLogs() function using the PDI interface.

**Note:**
Only 16-bit reads are provided by the reference code.

This function is only relevant for the HAL API Test application.

**Returns:**
None.

#### 7.1.1.6 ESC_debugInitESCRegLogs

Initializes ESC Register Read Log Array

**Prototype:**
```
void
ESC_debugInitESCRegLogs(void)
```

**Description:**
This function is optional for the non-HAL API Test application use cases and is available for applications or users to and initializes the registers read log array to default 0xFFFF. This is called once during init time or user can call it after every update to reset the previous read values in the array.

**Note:**
This function is only relevant for the HAL API Test application.

**Returns:**
None.

### 7.1.1.7 ESC_debugUpdateESCRegLogs

Reloads local RAM with ESC register Values

**Prototype:**

```
void
ESC_debugUpdateESCRegLogs(void)
```

**Description:**
This function is optional for the non-HAL API Test application use cases and is available for applications or users to perform a load of pre-set ESC registers in a loop by using PDI interface.

**Note:**
This function is only relevant for the HAL API Test application.

**Returns:**
None.

### 7.1.1.8 ESC_getTimer

Gets the Current Timer Value

**Prototype:**

```
uint32_t
ESC_getTimer(void)
```

**Description:**
This function returns the current timer counter value from the CPU timer.

**Returns:**
Returns the 1's compliment of the timer counter register value

### 7.1.1.9 ESC_holdESCInReset

Hold ESC in Reset

**Prototype:**

```
void
ESC_holdESCInReset(void)
```

**Description:**
This function holds the ESC peripheral in reset.

**Returns:**
None.

### 7.1.1.10 ESC_initHW

Initializes the Device for EtherCAT

**Prototype:**

```
uint16_t
ESC_initHW(void)
```

**Description:**

This function initializes the host controller, interrupts, SYNC signals, PDI, and other necessary peripherals.

**Returns:**

Returns **ESC_HW_INIT_SUCCESS** if initialization was successful and **ESC_HW_INIT_FAIL** if an error occurred during initialization

#### 7.1.1.11 ESC_loadedCheckEEPROM

Checks if EEPROM was Loaded

**Prototype:**

```
uint16_t
ESC_loadedCheckEEPROM(void)
```

**Description:**

This function checks if the EEPROM load happened properly or not. The function reads the EEPROM LOADED register bit in the DL register as no proper EEPROM loaded IO signal is available. Recommended to be called by applications during start up and after an EEPROM reload happens.

**Note:**

ESC RAM access via PDI is blocked until EEPROM happens correctly.

**Returns:**

Returns **ESC_EEPROM_SUCCESS** if EEPROM loaded successfully, **ESC_EEPROM_NOT_LOADED** if EEPROM not loaded as per the ESC DL register status, and **ESC_EEPROM_LOAD_ERROR** if EEPROM ESC control status register indicates that EEPROM is not loaded and device information not available.

#### 7.1.1.12 ESC_passFailSignalSetup

Sets up the ControlCARD GPIOs for LEDs

**Prototype:**

```
void
ESC_passFailSignalSetup(void)
```

**Description:**

This function sets up the LED GPIOs that are used to signal the PASS/FAIL conditions.

**Note:**

This function is tied to the controlCARD as in the GPIOs used as per the controlCARD hardware design.

This function is only relevant for the HAL API Test application.

**Returns:**

None.

#### 7.1.1.13 ESC_readBlock

Reads the ESC Data into Local Buffer with Interrupts Disabled

**Prototype:**

```
void
ESC_readBlock(ESCMEM_ADDR *pData,
              uint16_t address,
              uint16_t len)
```

**Parameters:**

***pData*** is the pointer to the local destination buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**

This function is used to access the ESC registers and the DPRAM area with interrupts disabled. The function disables interrupts, reads the requested number of bytes from the ESC address, copies the data into the data buffer specified, and re-enables interrupts.

**Returns:**

None

#### 7.1.1.14 ESC_readBlockISR

Reads the ESC Data into Local Buffer

**Prototype:**

```
void
ESC_readBlockISR(ESCMEM_ADDR *pData,
                 uint16_t address,
                 uint16_t len)
```

**Parameters:**

***pData*** is the pointer to the local destination buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**

This function is used to access the ESC registers and the DPRAM area. The function reads the requested number of bytes from the ESC address and copies the data into the data buffer specified.

**Returns:**

None

#### 7.1.1.15 ESC_readDWord

Reads two 16-bit words from ESC Memory with interrupts disabled

**Prototype:**
```
uint32_t
ESC_readDWord(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**
This function disables interrupts, then reads two 16-bit words from the specified ESC address, and re-enables interrupts.

**Returns:**
Returns two 16-bit words

#### 7.1.1.16 ESC_readDWordISR

Reads two 16-bit words from ESC Memory

**Prototype:**
```
uint32_t
ESC_readDWordISR(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**
This function reads two 16-bit words from the specified ESC address.

**Returns:**
Returns two 16-bit words

#### 7.1.1.17 ESC_readWord

Reads one 16-bit word from ESC Memory with interrupts disabled

**Prototype:**
```
uint16_t
ESC_readWord(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**
This function disables interrupts, reads one 16-bit word from the specified ESC address, and re-enables interrupts.

**Returns:**
Returns 16-bit word value

#### 7.1.1.18 ESC_readWordISR

Reads one 16-bit word from ESC Memory

**Prototype:**
```
uint16_t
ESC_readWordISR(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**
This function reads one 16-bit word from the specified ESC address..

**Returns:**
Returns 16-bit word value

#### 7.1.1.19 ESC_releaseESCReset

Release ESC from Reset

**Prototype:**
```
void
ESC_releaseESCReset(void)
```

**Description:**
This function de-activates the ESC peripheral reset signal and brings ESC out of reset.

**Returns:**
None.

#### 7.1.1.20 ESC_releaseHW

Releases the Device Resources

**Prototype:**
```
void
ESC_releaseHW(void)
```

**Description:**
This function releases the allocated device resources.

**Note:**
Implementation of this function is left to the end user and currently performs no action.

**Returns:**
None

#### 7.1.1.21 ESC_resetESC

Reset the ESC

**Prototype:**

```
void
ESC_resetESC(void)
```

**Description:**
This function resets the ESC peripheral.

**Returns:**
None.

#### 7.1.1.22 ESC_setLed

Updates the EtherCAT Run and Error LEDs

**Prototype:**

```
void
ESC_setLed(uint16_t runLed,
           uint16_t errLed)
```

**Parameters:**
***runLed*** is the EtherCAT run LED state
***errLed*** is the EtherCAT error LED state

**Description:**
This function updates the EtherCAT run and error LEDS (or EtherCAT status LED).

**Note:**
This is configured to use the LED GPIOs for the controlCARD.

**Returns:**
None

#### 7.1.1.23 ESC_setupPDITestInterface

Setup and run tests on the PDI Interface

**Prototype:**

```
void
ESC_setupPDITestInterface(void)
```

**Description:**
This function is optional for the non-HAL API Test application use cases and is available for applications or users to perform a test of the PDI interface. The function reads the PDI control registers, initializes an array of registers that needs to be read from ESC and also performs read write tests on all of the RAM in ESC using the HAL API.

**Note:**
This function is only relevant for the HAL API Test application.

**Returns:**
None.

#### 7.1.1.24 ESC_signalFail

Signal Fail Status on ControlCARD LEDs

**Prototype:**
```
void
ESC_signalFail(void)
```

**Description:**
This function provides a FAIL signature on the LED GPIOs when the tests complete successfully.

**Note:**
This function is tied to the controlCARD as in the GPIOs used as per the controlCARD hardware design.

This function is only relevant for the HAL API Test application.

**Returns:**
None.

#### 7.1.1.25 ESC_signalPass

Signal Pass Status on ControlCARD LEDs

**Prototype:**
```
void
ESC_signalPass(void)
```

**Description:**
This function provides a PASS signature on the LED GPIOs when the tests complete successfully.

**Note:**
This function is tied to the controlCARD as in the GPIOs used as per the controlCARD hardware design.

This function is only relevant for the HAL API Test application.

**Returns:**
None.

#### 7.1.1.26 ESC_timerIncPerMilliSec

Get the Timer Increment Value

**Prototype:**
```
uint32_t
ESC_timerIncPerMilliSec(void)
```

**Description:**
This function returns a constant value of 200000UL for the timer increment value as the CPU timer is configured to run at 200MHz.

**Returns:**
Returns a constant value depending on the max frequency of the CPU timer

#### 7.1.1.27 ESC_writeBlock

Writes the Local Buffer Data into the ESC Memory with interrupts disabled

**Prototype:**
```
void
ESC_writeBlock(ESCMEM_ADDR *pData,
               uint16_t address,
               uint16_t len)
```

**Parameters:**
***pData*** is the pointer to the local source buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**
This function disables interrupts, writes the requested number of bytes from the data buffer into the specified ESC addresses, and re-enables interrupts.

**Returns:**
None

#### 7.1.1.28 ESC_writeBlockISR

Writes the Local Buffer Data into the ESC Memory

**Prototype:**
```
void
ESC_writeBlockISR(ESCMEM_ADDR *pData,
                  uint16_t address,
                  uint16_t len)
```

**Parameters:**
***pData*** is the pointer to the local source buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**
This function writes the requested number of bytes from the data buffer and into the specified ESC addresses.

**Returns:**
None

#### 7.1.1.29 ESC_writeDWord

Writes two 16-bit words into ESC Memory with interrupts disabled

**Prototype:**
```
void
ESC_writeDWord(uint32_t dWordValue,
               uint16_t address)
```

**Parameters:**
***dWordValue*** is the local 32-bit variable which contains the value that needs to be written.
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**
This function disables interrupts, writes two 16-bit words from *DWordValue* to the ESC address, and re-enables interrupts.

**Returns:**
None

#### 7.1.1.30 ESC_writeDWordISR

Writes two 16-bit words into ESC Memory

**Prototype:**
```
void
ESC_writeDWordISR(uint32_t dWordValue,
                  uint16_t address)
```

**Parameters:**
***dWordValue*** is the local 32-bit variable which contains the value that needs to be written.
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**
This function writes two 16-bit words from *DWordValue* to the ESC address.

**Returns:**
None

### 7.1.1.31 ESC_writeWord

Writes one 16-bit word into ESC Memory with interrupts disabled

**Prototype:**
```
void
ESC_writeWord(uint16_t wordValue,
              uint16_t address)
```

**Parameters:**

***wordValue*** is the local 16-bit variable which contains the value to be written.

***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**

This function disables interrupts, writes one 16-bit word from *WordValue* into the specified ESC address, and re-enables interrupts.

**Returns:**

None

### 7.1.1.32 ESC_writeWordISR

Writes one 16-bit word into ESC Memory

**Prototype:**
```
void
ESC_writeWordISR(uint16_t wordValue,
                 uint16_t address)
```

**Parameters:**

***wordValue*** is the local 16-bit variable which contains the value to be written.

***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**

This function writes one 16-bit word from *WordValue* into the specified ESC address.

**Returns:**

None

## 7.2 CM HAL APIs

## Functions

- __interrupt void ESC_applicationLayerHandler (void)
- __interrupt void ESC_applicationSync0Handler (void)
- __interrupt void ESC_applicationSync1Handler (void)
- void ESC_clearTimer (void)
- void ESC_debugAddESCRegsAddress (uint16_t address)

- void ESC_debugInitESCRegLogs (void)
- void ESC_debugUpdateESCRegLogs (void)
- uint32_t ESC_getTimer (void)
- void ESC_holdESCInReset (void)
- uint16_t ESC_initHW (void)
- uint16_t ESC_loadedCheckEEPROM (void)
- void ESC_passFailSignalSetup (void)
- void ESC_readBlock (ESCMEM_ADDR *pData, uint16_t address, uint16_t len)
- void ESC_readBlockISR (ESCMEM_ADDR *pData, uint16_t address, uint16_t len)
- uint8_t ESC_readByte (uint16_t address)
- uint8_t ESC_readByteISR (uint16_t address)
- uint32_t ESC_readDWord (uint16_t address)
- uint32_t ESC_readDWordISR (uint16_t address)
- uint16_t ESC_readWord (uint16_t address)
- uint16_t ESC_readWordISR (uint16_t address)
- void ESC_releaseESCReset (void)
- void ESC_releaseHW (void)
- void ESC_resetESC (void)
- void ESC_setLed (uint8_t runLed, uint8_t errLed)
- void ESC_setupPDITestInterface (void)
- void ESC_signalFail (void)
- void ESC_signalPass (void)
- uint32_t ESC_timerIncPerMilliSec (void)
- void ESC_writeBlock (ESCMEM_ADDR *pData, uint16_t address, uint16_t len)
- void ESC_writeBlockISR (ESCMEM_ADDR *pData, uint16_t address, uint16_t len)
- void ESC_writeByte (uint8_t byteValue, uint16_t address)
- void ESC_writeByteISR (uint8_t byteValue, uint16_t address)
- void ESC_writeDWord (uint32_t dWordValue, uint16_t address)
- void ESC_writeDWordISR (uint32_t dWordValue, uint16_t address)
- void ESC_writeWord (uint16_t wordValue, uint16_t address)
- void ESC_writeWordISR (uint16_t wordValue, uint16_t address)

## 7.2.1 Function Documentation

### 7.2.1.1 ESC_applicationLayerHandler

Application Layer Handler

**Prototype:**

```
__interrupt void
ESC_applicationLayerHandler(void)
```

**Description:**
This function is the interrupt handler for EtherCAT application/PDI interrupts.

**Returns:**
None

#### 7.2.1.2 ESC_applicationSync0Handler

Application Sync 0 Handler

**Prototype:**

```
__interrupt void
ESC_applicationSync0Handler(void)
```

**Description:**
This function is the interrupt handler for EtherCAT SYNC0 interrupts.

**Returns:**
None

#### 7.2.1.3 ESC_applicationSync1Handler

Application Sync 1 Handler

**Prototype:**

```
__interrupt void
ESC_applicationSync1Handler(void)
```

**Description:**
This function is the interrupt handler for EtherCAT SYNC1 interrupts.

**Returns:**
None

#### 7.2.1.4 ESC_clearTimer

Clears the Timer Value

**Prototype:**

```
void
ESC_clearTimer(void)
```

**Description:**
This function resets the timer counter.

**Returns:**
None

#### 7.2.1.5 ESC_debugAddESCRegsAddress

Adds ESC Register Address to be read to RAM for Debug

**Prototype:**

```
void
ESC_debugAddESCRegsAddress(uint16_t address)
```

**Parameters:**

***address*** is the ESC register or memory address that needs to be read

**Description:**

This function is optional for the non-HAL API Test application use cases and is available for applications or users to add a register to the pre-set array of registers that are ready by the ESC_debugUpdateESCRegLogs() function using the PDI interface.

**Note:**

Only 16-bit reads are provided by the reference code.

This function is only relevant for the HAL API Test application.

**Returns:**

None.

#### 7.2.1.6 ESC_debugInitESCRegLogs

Initializes ESC Register Read Log Array

**Prototype:**

```
void
ESC_debugInitESCRegLogs(void)
```

**Description:**

This function is optional for the non-HAL API Test application use cases and is available for applications or users to and initializes the registers read log array to default 0xFFFF. This is called once during init time or user can call it after every update to reset the previous read values in the array.

**Note:**

This function is only relevant for the HAL API Test application.

**Returns:**

None.

#### 7.2.1.7 ESC_debugUpdateESCRegLogs

Reloads local RAM with ESC register Values

**Prototype:**

```
void
ESC_debugUpdateESCRegLogs(void)
```

**Description:**

This function is optional for the non-HAL API Test application use cases and is available for applications or users to perform a load of pre-set ESC registers in a loop by using PDI interface.

**Note:**

This function is only relevant for the HAL API Test application.

**Returns:**

None.

#### 7.2.1.8 ESC_getTimer

Gets the Current Timer Value

**Prototype:**
```
uint32_t
ESC_getTimer(void)
```

**Description:**
This function returns the current timer counter value from the CPU timer.

**Returns:**
Returns the 1's compliment of the timer counter register value

#### 7.2.1.9 ESC_holdESCInReset

Hold ESC in Reset

**Prototype:**
```
void
ESC_holdESCInReset(void)
```

**Description:**
This function holds the ESC peripheral in reset.

**Returns:**
None.

#### 7.2.1.10 ESC_initHW

Initializes the Device for EtherCAT

**Prototype:**
```
uint16_t
ESC_initHW(void)
```

**Description:**
This function initializes the host controller, interrupts, SYNC signals, PDI, and other necessary peripherals.

**Returns:**
Returns **ESC_HW_INIT_SUCCESS** if initialization was successful and **ESC_HW_INIT_FAIL** if an error occurred during initialization

#### 7.2.1.11 ESC_loadedCheckEEPROM

Checks if EEPROM was Loaded

**Prototype:**
```
uint16_t
ESC_loadedCheckEEPROM(void)
```

**Description:**
This function checks if the EEPROM load happened properly or not. The function reads the EEPROM LOADED register bit in the DL register as no proper EEPROM loaded IO signal is available. Recommended to be called by applications during start up and after an EEPROM reload happens.

**Note:**
ESC RAM access via PDI is blocked until EEPROM happens correctly.

**Returns:**
Returns **ESC_EEPROM_SUCCESS** if EEPROM loaded successfully, **ESC_EEPROM_NOT_LOADED** if EEPROM not loaded as per the ESC DL register status, and **ESC_EEPROM_LOAD_ERROR** if EEPROM ESC control status register indicates that EEPROM is not loaded and device information not available.

### 7.2.1.12 ESC_passFailSignalSetup

Sets up the ControlCARD GPIOs for LEDs

**Prototype:**
```
void
ESC_passFailSignalSetup(void)
```

**Description:**
This function sets up the LED GPIOs that are used to signal the PASS/FAIL conditions.

**Note:**
This function is tied to the controlCARD as in the GPIOs used as per the controlCARD hardware design.

This function is only relevant for the HAL API Test application.

**Returns:**
None.

### 7.2.1.13 ESC_readBlock

Reads the ESC Data into Local Buffer with Interrupts Disabled

**Prototype:**
```
void
ESC_readBlock(ESCMEM_ADDR *pData,
              uint16_t address,
              uint16_t len)
```

**Parameters:**
***pData*** is the pointer to the local destination buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**
This function is used to access the ESC registers and the DPRAM area with interrupts disabled. The function disables interrupts, reads the requested number of bytes from the ESC address, copies the data into the data buffer specified, and re-enables interrupts.

**Returns:**
None

#### 7.2.1.14 ESC_readBlockISR

Reads the ESC Data into Local Buffer

**Prototype:**
```
void
ESC_readBlockISR(ESCMEM_ADDR *pData,
                 uint16_t address,
                 uint16_t len)
```

**Parameters:**
***pData*** is the pointer to the local destination buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**
This function is used to access the ESC registers and the DPRAM area. The function reads the requested number of bytes from the ESC address and copies the data into the data buffer specified.

**Returns:**
None

#### 7.2.1.15 ESC_readByte

Reads one byte from ESC Memory with interrupts disabled

**Prototype:**
```
uint8_t
ESC_readByte(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes.

**Description:**
This function disables interrupts, reads one byte from the specified ESC address, and re-enables interrupts.

**Returns:**
Returns byte value

#### 7.2.1.16 ESC_readByteISR

Reads one byte from ESC Memory

**Prototype:**
```
uint8_t
ESC_readByteISR(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes.

**Description:**
This function reads one byte from the specified ESC address.

**Returns:**
Returns byte value

#### 7.2.1.17 ESC_readDWord

Reads two 16-bit words from ESC Memory with interrupts disabled

**Prototype:**
```
uint32_t
ESC_readDWord(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**
This function disables interrupts, then reads two 16-bit words from the specified ESC address, and re-enables interrupts.

**Returns:**
Returns two 16-bit words

#### 7.2.1.18 ESC_readDWordISR

Reads two 16-bit words from ESC Memory

**Prototype:**
```
uint32_t
ESC_readDWordISR(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**
This function reads two 16-bit words from the specified ESC address.

**Returns:**
Returns two 16-bit words

#### 7.2.1.19 ESC_readWord

Reads one 16-bit word from ESC Memory with interrupts disabled

**Prototype:**
```
uint16_t
ESC_readWord(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**
This function disables interrupts, reads one 16-bit word from the specified ESC address, and re-enables interrupts.

**Returns:**
Returns 16-bit word value

#### 7.2.1.20 ESC_readWordISR

Reads one 16-bit word from ESC Memory

**Prototype:**
```
uint16_t
ESC_readWordISR(uint16_t address)
```

**Parameters:**
***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**
This function reads one 16-bit word from the specified ESC address.

**Returns:**
Returns 16-bit word value

#### 7.2.1.21 ESC_releaseESCReset

Release ESC from Reset

**Prototype:**
```
void
ESC_releaseESCReset(void)
```

**Description:**
This function de-activates the ESC peripheral reset signal and brings ESC out of reset.

**Returns:**
None.

### 7.2.1.22 ESC_releaseHW

Releases the Device Resources

**Prototype:**
```
void
ESC_releaseHW(void)
```

**Description:**
This function releases the allocated device resources.

**Note:**
Implementation of this function is left to the end user and currently performs no action.

**Returns:**
None

### 7.2.1.23 ESC_resetESC

Reset the ESC

**Prototype:**
```
void
ESC_resetESC(void)
```

**Description:**
This function resets the ESC peripheral.

**Returns:**
None.

### 7.2.1.24 ESC_setLed

Updates the EtherCAT Run and Error LEDs

**Prototype:**
```
void
ESC_setLed(uint8_t runLed,
           uint8_t errLed)
```

**Parameters:**
***runLed*** is the EtherCAT run LED state
***errLed*** is the EtherCAT error LED state

**Description:**
This function updates the EtherCAT run and error LEDS (or EtherCAT status LED).

**Note:**
This is configured to use the LED GPIOs for the controlCARD.

**Returns:**
None

#### 7.2.1.25 ESC_setupPDITestInterface

Setup and run tests on the PDI Interface

**Prototype:**

```
void
ESC_setupPDITestInterface(void)
```

**Description:**
This function is optional for the non-HAL API Test application use cases and is available for applications or users to perform a test of the PDI interface. The function reads the PDI control registers, initializes an array of registers that needs to be read from ESC and also performs read write tests on all of the RAM in ESC using the HAL API.

**Note:**
This function is only relevant for the HAL API Test application.

**Returns:**
None.

#### 7.2.1.26 ESC_signalFail

Signal Fail Status on ControlCARD LEDs

**Prototype:**

```
void
ESC_signalFail(void)
```

**Description:**
This function provides a FAIL signature on the LED GPIOs when the tests complete successfully.

**Note:**
This function is tied to the controlCARD as in the GPIOs used as per the controlCARD hardware design.

This function is only relevant for the HAL API Test application.

**Returns:**
None.

#### 7.2.1.27 ESC_signalPass

Signal Pass Status on ControlCARD LEDs

**Prototype:**

```
void
ESC_signalPass(void)
```

**Description:**
This function provides a PASS signature on the LED GPIOs when the tests complete successfully.

**Note:**
This function is tied to the controlCARD as in the GPIOs used as per the controlCARD hardware design.

This function is only relevant for the HAL API Test application.

**Returns:**
None.

#### 7.2.1.28 ESC_timerIncPerMilliSec

Get the Timer Increment Value

**Prototype:**
```
uint32_t
ESC_timerIncPerMilliSec(void)
```

**Description:**
This function returns a constant value of 125000UL for the timer increment value as the CPU timer is configured to run at 125MHz.

**Returns:**
Returns a constant value depending on the max frequency of the CPU timer

#### 7.2.1.29 ESC_writeBlock

Writes the Local Buffer Data into the ESC Memory with interrupts disabled

**Prototype:**
```
void
ESC_writeBlock(ESCMEM_ADDR *pData,
               uint16_t address,
               uint16_t len)
```

**Parameters:**
***pData*** is the pointer to the local source buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**
This function disables interrupts, writes the requested number of bytes from the data buffer into the specified ESC addresses, and re-enables interrupts.

**Returns:**
None

### 7.2.1.30 ESC_writeBlockISR

Writes the Local Buffer Data into the ESC Memory

**Prototype:**

```
void
ESC_writeBlockISR(ESCMEM_ADDR *pData,
                  uint16_t address,
                  uint16_t len)
```

**Parameters:**

***pData*** is the pointer to the local source buffer. (Type of pointer depends on the host controller architecture, detailed in ecat_def.h or the Slave Stack Code Tool)

***address*** is the EtherCAT slave controller offset address which specifies the offset within the ESC memory area in bytes. (Only valid addresses are used depending on ESC 8 bit, 16 bit, or 32 bit access specified in ecat_def.h or the Slave Stack Code Tool)

***len*** is the access size in bytes

**Description:**

This function writes the requested number of bytes from the data buffer and into the specified ESC addresses.

**Returns:**

None

### 7.2.1.31 ESC_writeByte

Write one byte into ESC Memory with interrupts disabled

**Prototype:**

```
void
ESC_writeByte(uint8_t byteValue,
              uint16_t address)
```

**Parameters:**

***byteValue*** is the local 8-bit variable which contains the value to be written.

***address*** is the EtherCAT slave controller offset address in bytes.

**Description:**

This function disables interrupts, writes one byte from *byteValue* into the specified ESC address, and re-enables interrupts.

**Returns:**

None

### 7.2.1.32 ESC_writeByteISR

Write one byte into ESC Memory

**Prototype:**

```
void
ESC_writeByteISR(uint8_t byteValue,
                 uint16_t address)
```

**Parameters:**

***byteValue*** is the local 8-bit variable which contains the value to be written.

***address*** is the EtherCAT slave controller offset address in bytes.

**Description:**

This function writes one byte from *byteValue* into the specified ESC address.

**Returns:**

None

### 7.2.1.33 ESC_writeDWord

Writes two 16-bit words into ESC Memory with interrupts disabled

**Prototype:**

```
void
ESC_writeDWord(uint32_t dWordValue,
               uint16_t address)
```

**Parameters:**

***dWordValue*** is the local 32-bit variable which contains the value that needs to be written.

***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**

This function disables interrupts, writes two 16-bit words from *dWordValue* to the ESC address, and re-enables interrupts.

**Returns:**

None

### 7.2.1.34 ESC_writeDWordISR

Writes two 16-bit words into ESC Memory

**Prototype:**

```
void
ESC_writeDWordISR(uint32_t dWordValue,
                  uint16_t address)
```

**Parameters:**

***dWordValue*** is the local 32-bit variable which contains the value that needs to be written.

***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 32-bit aligned address boundary.

**Description:**

This function writes two 16-bit words from *dWordValue* to the ESC address.

**Returns:**

None

### 7.2.1.35 ESC_writeWord

Writes one 16-bit word into ESC Memory with interrupts disabled

**Prototype:**

```
void
ESC_writeWord(uint16_t wordValue,
              uint16_t address)
```

**Parameters:**

***wordValue*** is the local 16-bit variable which contains the value to be written.

***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**

This function disables interrupts, writes one 16-bit word from *wordValue* into the specified ESC address, and re-enables interrupts.

**Returns:**

None

### 7.2.1.36 ESC_writeWordISR

Writes one 16-bit word into ESC Memory

**Prototype:**

```
void
ESC_writeWordISR(uint16_t wordValue,
                 uint16_t address)
```

**Parameters:**

***wordValue*** is the local 16-bit variable which contains the value to be written.

***address*** is the EtherCAT slave controller offset address in bytes. This must be a valid 16-bit aligned address boundary.

**Description:**

This function writes one 16-bit word from *wordValue* into the specified ESC address.

**Returns:**

None

# 8 EtherCAT Performance Data

## 8.1 EtherCAT Example Software Analysis

**IMPORTANT:** All the following analysis should be taken as reference since CPU cycles and memory usage will vary based on the specifics of the EtherCAT application implementation.

**Profiling 1: Application Loop and ISR Cycles**

- **Analysis Details:** Example solution analysis of the average CPU cycles for the EtherCAT main loop (excluding ISRs), application (PDI) ISR, and SYNC0 ISR.
- **Application Details:**
  - TwinCAT Master + Distributed Clocks Mode
  - Project: F2838x echoback solution (CPU1 at 200MHz, CM at 125MHz)
  - Beckhoff Slave Stack Code v5.12
  - EtherCAT Profile: CAN over EtherCAT
  - EtherCAT Output Variables: 1 8-bit, 1 16-bit, and 2 32-bit
  - EtherCAT Input Variables: 1 8-bit, 1 16-bit, and 2 32-bit
  - Compiler v18.12.4.LTS at optimization level 2

| Execution Memory | Unit | EtherCAT Owner: CPU1 | EtherCAT Owner: CM |
|---|---|---|---|
| RAM | CPU Cycles | Main Loop: 698<br>Application (PDI) ISR: 994<br>SYNC0 ISR: 765 | Main Loop: 614<br>Application (PDI) ISR: 798<br>SYNC0 ISR: 785 |
| FLASH | CPU Cycles | Main Loop: 899<br>Application (PDI) ISR: 1120<br>SYNC0 ISR: 919 | Main Loop: 622<br>Application (PDI) ISR: 797<br>SYNC0 ISR: 745 |

Table 8.1: Application Loop and ISR Cycles Analysis

**Profiling 2: Application Memory Usage**

- **Analysis Details:** Example solution analysis of the RAM and FLASH memory usage by the application code as well as the slave stack code.
- **Application Details:**
  - Project: F2838x echoback solution (CPU1 at 200MHz, CM at 125MHz)
  - Beckhoff Slave Stack Code v5.12
  - EtherCAT Profile: CAN over EtherCAT
  - EtherCAT Output Variables: 1 8-bit, 1 16-bit, and 2 32-bit
  - EtherCAT Input Variables: 1 8-bit, 1 16-bit, and 2 32-bit
  - Compiler v18.12.4.LTS at optimization level 2

| Application Code | Unit | EtherCAT Owner: CPU1 | EtherCAT Owner: CM |
|---|---|---|---|
| Main App<br>(all non-stack code) | Bytes | RAM Usage: 1,820<br>FLASH Usage: 13,922 | RAM Usage: 1,020<br>FLASH Usage: 5,036 |
| Stack Only | Bytes | RAM Usage: 2,032<br>FLASH Usage: 18,844 | RAM Usage: 1,810<br>FLASH Usage: 18,798 |
| Main App + Stack | Bytes | RAM Usage: 3,852<br>FLASH Usage: 32,766 | RAM Usage: 2,830<br>FLASH Usage: 23,834 |

Table 8.2: Application Memory Usage Analysis

### Profiling 3: Application CPU Loading

- **Analysis Details:**
  - Example solution analysis of the EtherCAT application's CPU loading and how much estimated available CPU bandwidth (in CPU cycles) there is for non-EtherCAT related uninterruptible tasks.
  - The analysis was performed seperately at two locations within the EtherCAT application: **(1)** At the end of the EtherCAT main loop and **(2)** Within a 10kHz timer ISR.
  - The profiling increased the CPU cycle workload at one of the specified application locations while monitoring two statuses: **(1)** The Application/PDI ISR and SYNC0 ISR interrupt frequencies and **(2)** The EtherCAT slave stack PDI watchdog.
  - The number of uinterruptible CPU cycles that were able to be processed before the ISR frequencies dropped more than 5% were captured and then CPU cycles were captured again once the EtherCAT PDI watchdog began to timeout. These captured CPU cycles provide an estimate about available CPU cycle bandwidth while running an EtherCAT slave stack application.
- **Application Details:**
  - TwinCAT Master + Distributed Clocks Mode
  - Project: F2838x echoback solution (CPU1 at 200MHz, CM at 125MHz)
  - Beckhoff Slave Stack Code v5.12
  - EtherCAT Profile: CAN over EtherCAT
  - EtherCAT Output Variables: 1 8-bit, 1 16-bit, and 2 32-bit
  - EtherCAT Input Variables: 1 8-bit, 1 16-bit, and 2 32-bit
  - 100Hz DC SYNC0 Task
  - 100ms EtherCAT PDI Watchdog Timeout
  - Compiler v18.12.4.LTS at optimization level 2

| **Execution Memory** | **Unit** | **CPU Task Location** | **EtherCAT Owner: CPU1** | **EtherCAT Owner: CM** |
|---|---|---|---|---|
| RAM | CPU Cycles Available | Main Loop | ISR Timing: 2,198,005<br>PDI Watchdog: 20,020,006 | ISR Timing: 1,374,012<br>PDI Watchdog: 12,510,012 |
| FLASH | CPU Cycles Available | Main Loop | ISR Timing: 2,223,000<br>PDI Watchdog: 20,026,000 | ISR Timing: 1,372,013<br>PDI Watchdog: 12,508,013 |
| RAM | CPU Cycles Available | 10kHz Timer ISR | ISR Timing: 2,198,005<br>PDI Watchdog: 20,020,005 | ISR Timing: 1,374,011<br>PDI Watchdog: 12,510,011 |
| FLASH | CPU Cycles Available | 10kHz Timer ISR | ISR Timing: 2,227,002<br>PDI Watchdog: 20,026,002 | ISR Timing: 1,372,008<br>PDI Watchdog: 12,504,008 |

Table 8.3: Application CPU Loading Analysis

## 8.2 EtherCAT Network Analysis

Network Details for All Analysis Items

- **EtherCAT Master:** TwinCAT
- **EtherCAT Slaves:** F28388D using DP83822 PHYs
- **Number of Slaves:** 6
- **DC Cyclical Task Time:** 1ms
- **DC Cyclical Task Data:** 32 16-bit inputs and 32 16-bit outputs
- **Network Cable Length (1st to last slave):** 102 feet

Network Analysis Items' Descriptions

- **DC Sync0 Signal Delay Time:** The average time between the triggering of slave 1's SYNC0 signal and slave 6's SYNC0 signal.
- **DC Sync0 Signal Jitter Variance Time:** The jitter variance time of the slave 1's SYNC0 signal and slave 6's SYNC0 signal that varies as EtherCAT adjusts synchronization on the network.
- **DC Sync0 Delay to PWM Signal:** The time between when the SYNC0 event on the slave occurs and when the corresponding PWM output signal occurs (which is triggered by the SYNC0 signal).
- **Slave Communication Cycle:** The time for an EtherCAT frame to exit Slave 1, tranverse the network, and return to Slave 1.
- **Frame communication time from PHY0 to PHY1:** The time for an EtherCAT frame to enter F2838x's PHY0, go through the EtherCAT processing unit, and exit PHY1.

| Analysis Item | Unit | EtherCAT Owner: CPU1 | EtherCAT Owner: CM |
|---|---|---|---|
| DC Sync0 Signal Delay Time | Nanoseconds | 15 - 20 | 15 - 20 |
| DC Sync0 Signal Jitter Variance Time | Nanoseconds | 20 | 20 |
| DC Sync0 Delay to PWM Signal | Nanoseconds | 25 | 25 |
| Frame communication time from PHY0 to PHY1 | Nanoseconds | 405 | 405 |
| Slave Communication Cycle | Microseconds | 7 | 7 |

Table 8.4: EtherCAT Network Analysis

# 9 Revision History

**v2.01.00.00: New example and Enhancement Updates**
- New CM CiA402 Solution Example
- Updated user guide to include EtherCAT software and network performance data
- Clarified SYNC and LATCH GPIO configuration in HAL drivers
- Updated examples to default to using optimization level 2
- Fixed CM example FPU settings
- Updated CM linker command files to prioritize using CxRAMs

**v2.00.03.00: New chapter and 25MHz controlCARD XTAL Changes**
- Updated user guide to include details regarding controlCARD update to 25MHz XTAL
- Updated example code to warn about 25MHz XTAL controlCARD change
- Updated user guide to include new chapter on EtherCAT software development

**v2.00.02.00: Bug and Enhancement Updates**
- Updated user guide to include instructions on using Acontis EC-Engineer as EtherCAT Master
- Updated EtherCAT allocation to CM operation order (f2838x cpu1 allocate ecat to cm example)
- Updated SSC config XML to include updated patch file via Beckhoff for CPU1 and CM to fix sSyncmanagertype issue in 5.12 slave stack
- Fixed CPU1 examples to have entry point at code_start
- Fixed HAL SetLED() API to align with ControlCARD hardware changes

**v2.00.01.00: Bug and Enhancement Updates**
- Fixed C2000Ware path typo in user guide
- Add link to device target configuration file in example projects
- Fix memset function for CPU1 to handle input as number of bytes
- Update SSC config XML and associated files for CPU1 memset changes

**v2.00.00.00: Slave Stack Examples**
- CPU1 and CM HAL Driver API Updates to support Stack
- CPU1 Allocate ECAT to CM Example Updated
- CPU1 Echoback Demo Example
- CPU1 Echoback Solution Example
- CM Echoback Demo Example
- CM Echoback Solution Example
- SSC Configuration and application stack files

**v1.00.00.00: First Release**
- CPU1 and CM HAL Driver APIs
- CPU1 PDI HAL Test Example
- CPU1 Allocate ECAT to CM Example
- CM PDI HAL Test Example
- Master mode initialization is supported

# IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

| **Products** | | **Applications** | |
|---|---|---|---|
| Amplifiers | amplifier.ti.com | Audio | www.ti.com/audio |
| Data Converters | dataconverter.ti.com | Automotive | www.ti.com/automotive |
| DLP® Products | www.dlp.com | Broadband | www.ti.com/broadband |
| DSP | dsp.ti.com | Digital Control | www.ti.com/digitalcontrol |
| Clocks and Timers | www.ti.com/clocks | Medical | www.ti.com/medical |
| Interface | interface.ti.com | Military | www.ti.com/military |
| Logic | logic.ti.com | Optical Networking | www.ti.com/opticalnetwork |
| Power Mgmt | power.ti.com | Security | www.ti.com/security |
| Microcontrollers | microcontroller.ti.com | Telephony | www.ti.com/telephony |
| RFID | www.ti-rfid.com | Video & Imaging | www.ti.com/video |
| RF/IF and ZigBee® Solutions | www.ti.com/lprf | Wireless | www.ti.com/wireless |

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265