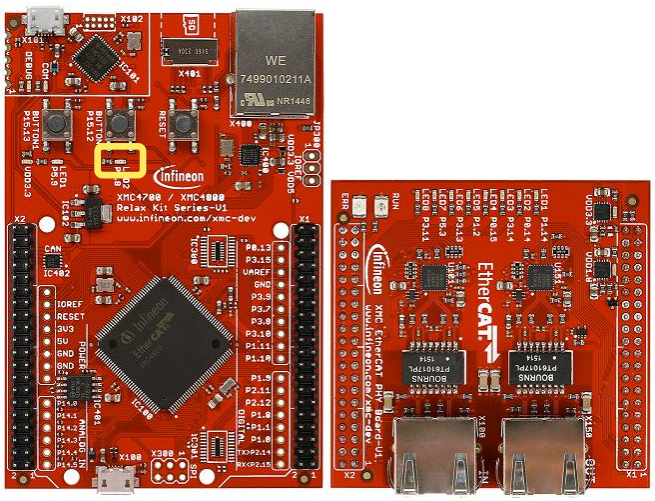
XMC4800 EtherCAT APP SSC Firmware Update Slave Example

Getting Started
Version 3.0

1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

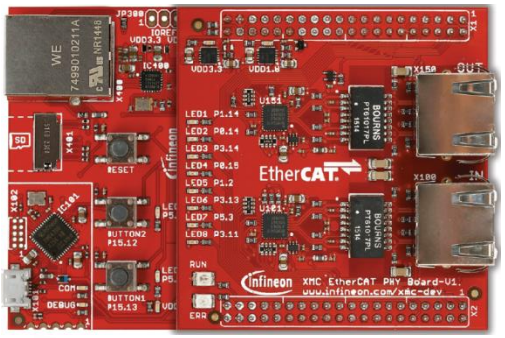
1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

Overview



This example demonstrates the implementation of a EtherCAT slave node which is capable doing firmware updates via FoE (File over EtherCAT).
This example here is based on the basic example described inside „Getting Started - XMC4800_Relax_EtherCat _APP_Slave_SSC Example_V2.2.pdf“.
Before you proceed with this example, make sure to have a solid understanding of the basic documentation first.
Here you will learn, to configure XMC4800 having two independent flash executables installed and the flash organized in four independent partitions. You will see how to generate a firmware update binary within DAVE4, add the firmware update functionality to the slave stack code and implement CRC32 checking on the binary. Finally you will see the firmware update functionality in action using TwinCAT.

Requirements



XMC4800 Relax EtherCAT Kit

RJ45 Ethernet Cable

Windows Laptop installed

- DAVE v4 (Version4.3.2 or higher)
- TwinCAT2 or TwinCAT3 Master PLC
- Slave Stack Code Tool **Version 5.12**

Micro USB Cable (Debugger connector)

Requirements - free downloads

TwinCAT2 (30 day trial; 32bit Windows only)
Link: Download TwinCAT2

or

TwinCAT3 (no trial period; usability limited; 32bit and 64bit Windows)
Link: Download TwinCAT3

ATTENTION: According to our experience TwinCAT is best compatible with Intel™ ethernet chipset.
For details on compatibility with your hardware, additional driver and general installation support please get into contact with your local BECKHOFF support.

Requirements - free downloads

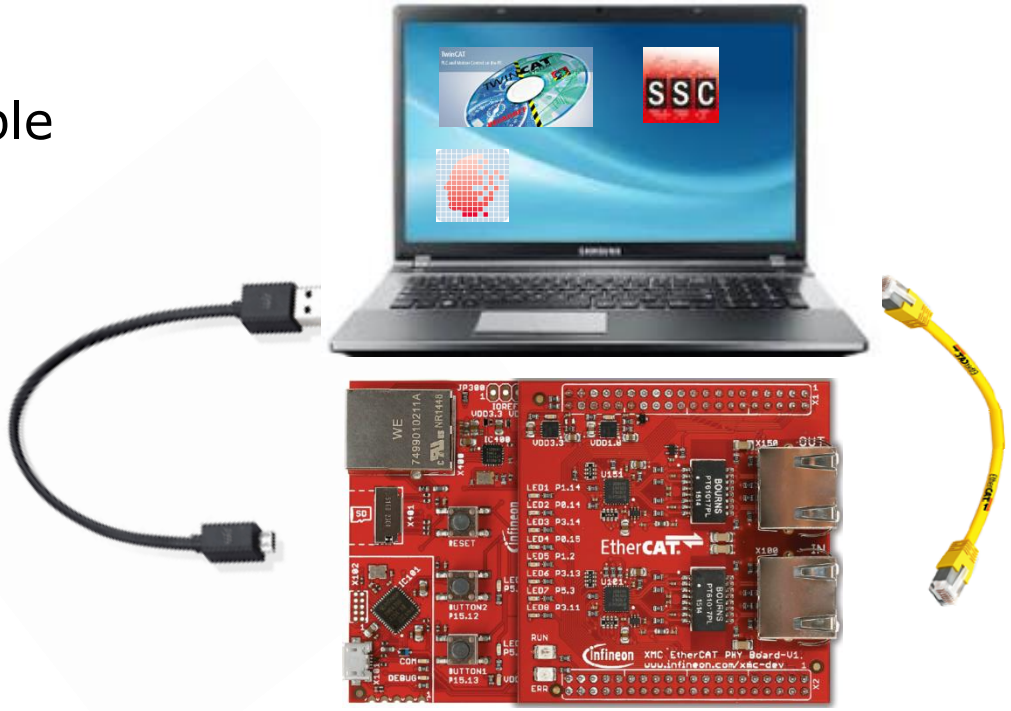
DAVE (v4.3.2 or higher)
Link: Download DAVE (Version 4)

EtherCAT Slave Stack Code Tool
Version 5.12
(ETG membership obligatory)
Link: Slave Stack Code Tool

1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

Setup – Hardware

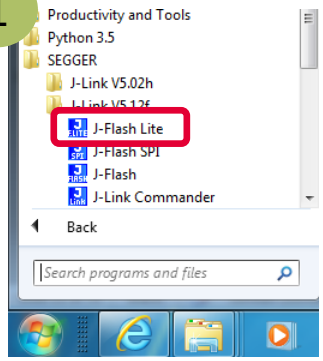
Micro USB cable
Debugger
connected to
X101 debug
connector



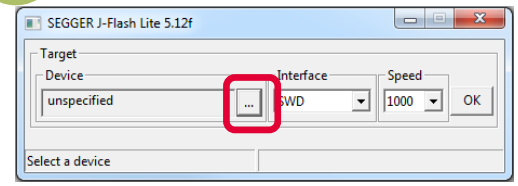
Ethernet Cable
connected to
IN-port

Setup – Cleanup flash of XMC4800

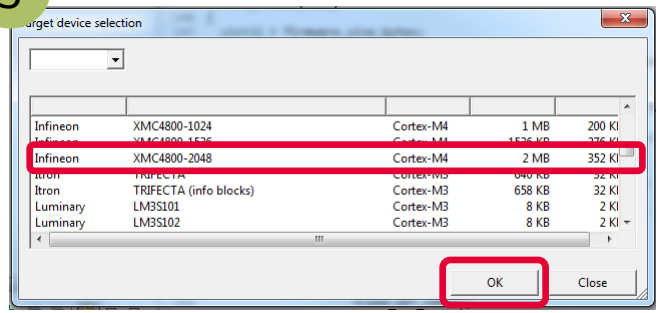
1



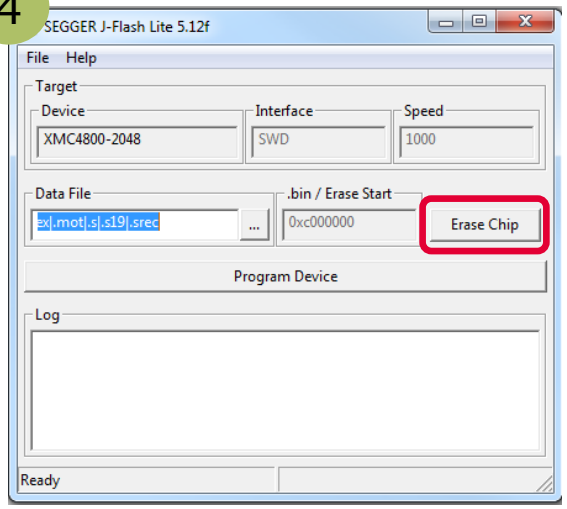
2



3



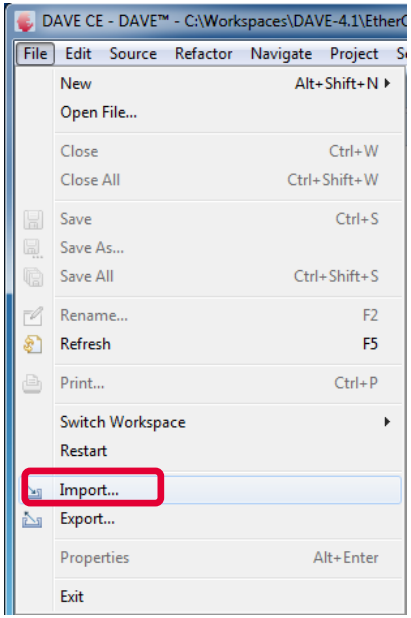
4



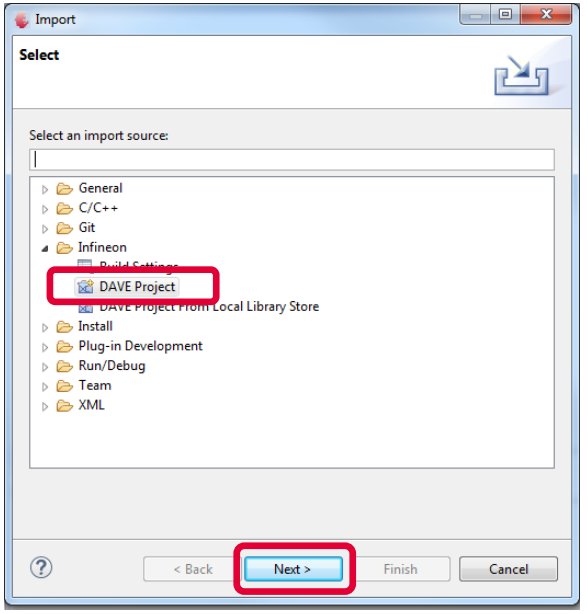
Make sure the XMC flash on your XMC4800 Relax EtherCAT Kit is cleaned up

Setup – Import the two example projects into DAVE

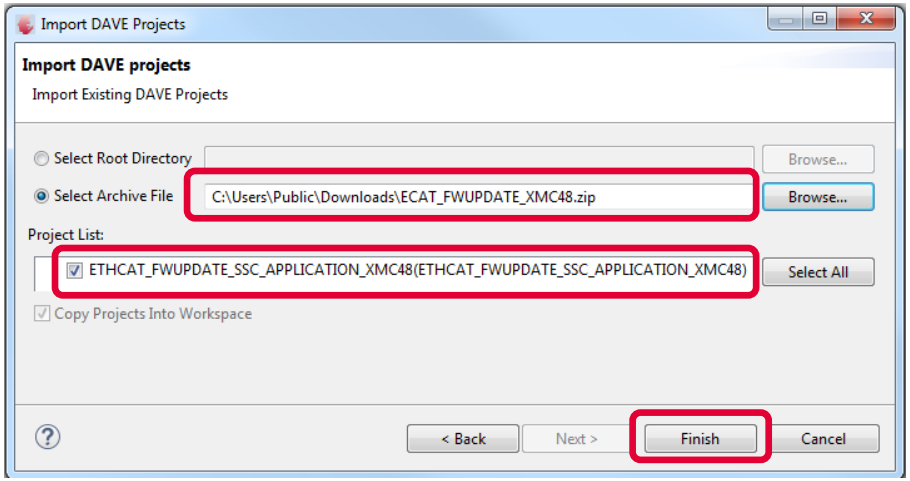
1



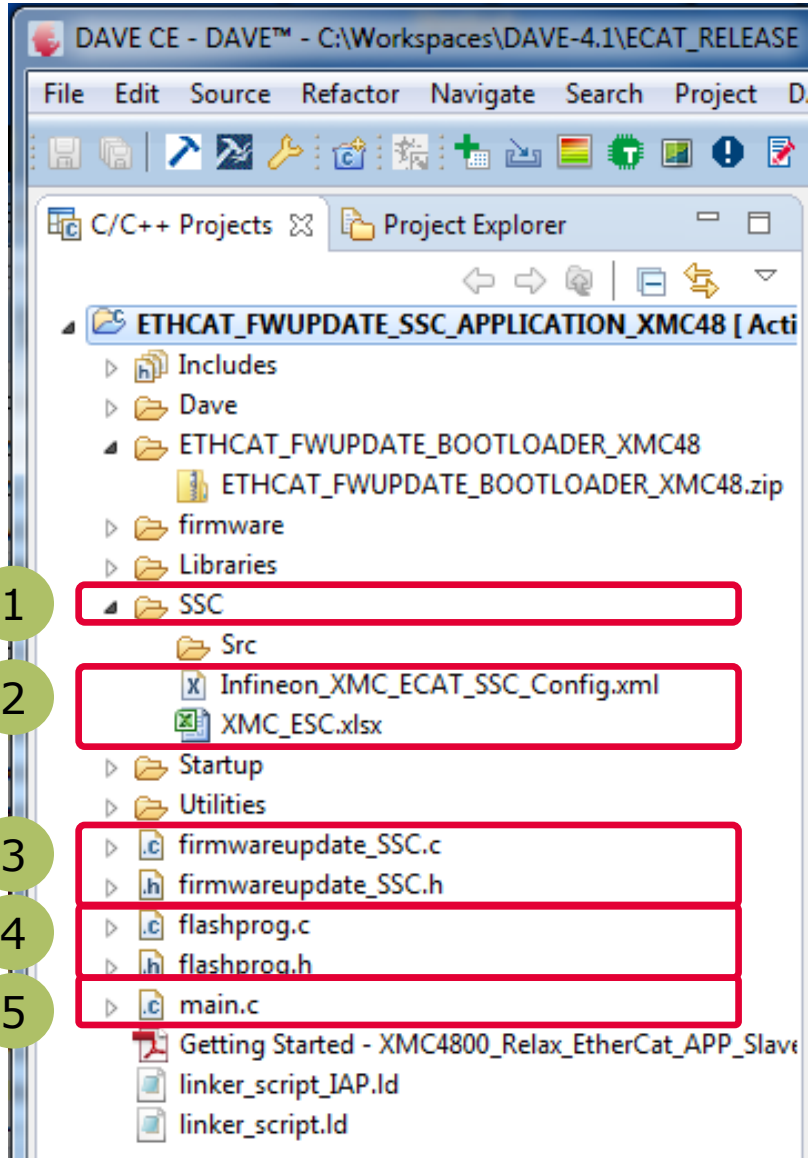
2



3



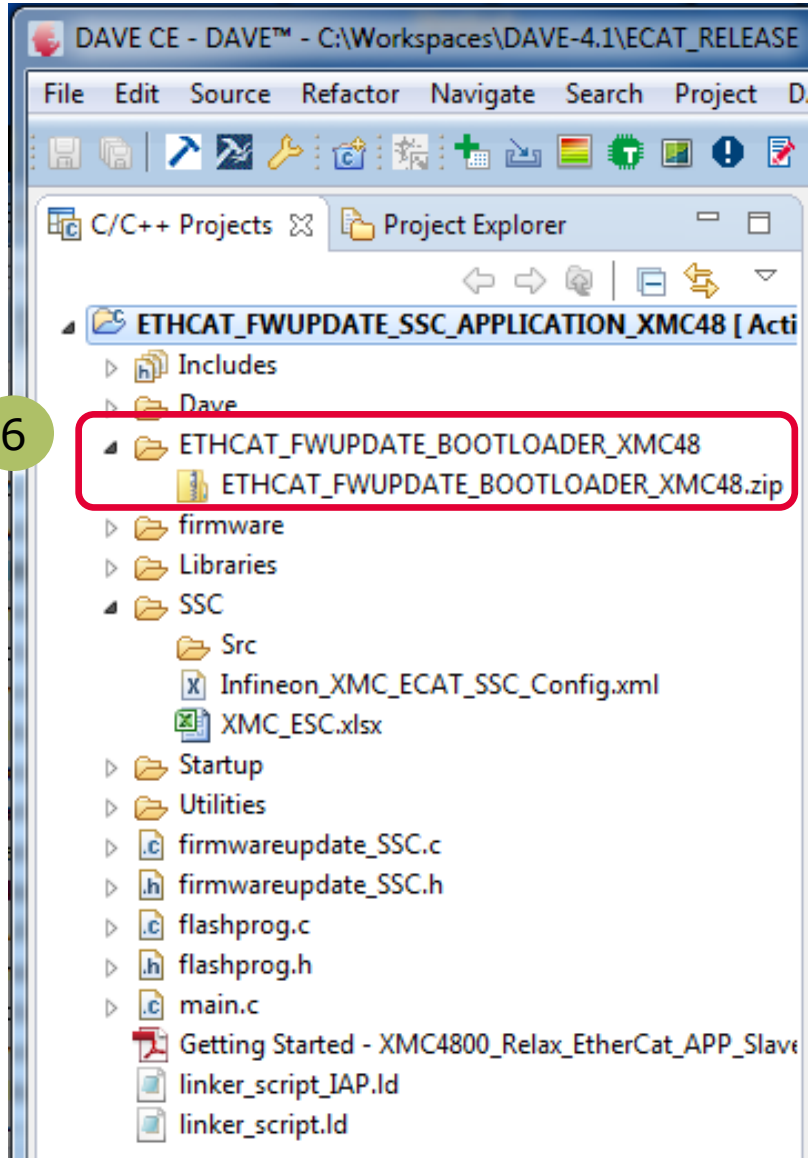
Setup – Imported application example



After the project import, you will find the project for application executable:

1. The project is nearly complete for build. It only misses the EtherCAT slave stack code. For these files, the Src folder has been already prepared.
2. The EtherCAT slave stack code for the XMC4800 is generated by configuration files. These configuration files are included in the project already.
3. Firmware update example implementation; called from SSC
4. Utility files to support the flash programming
5. Main.c of EtherCAT™ application

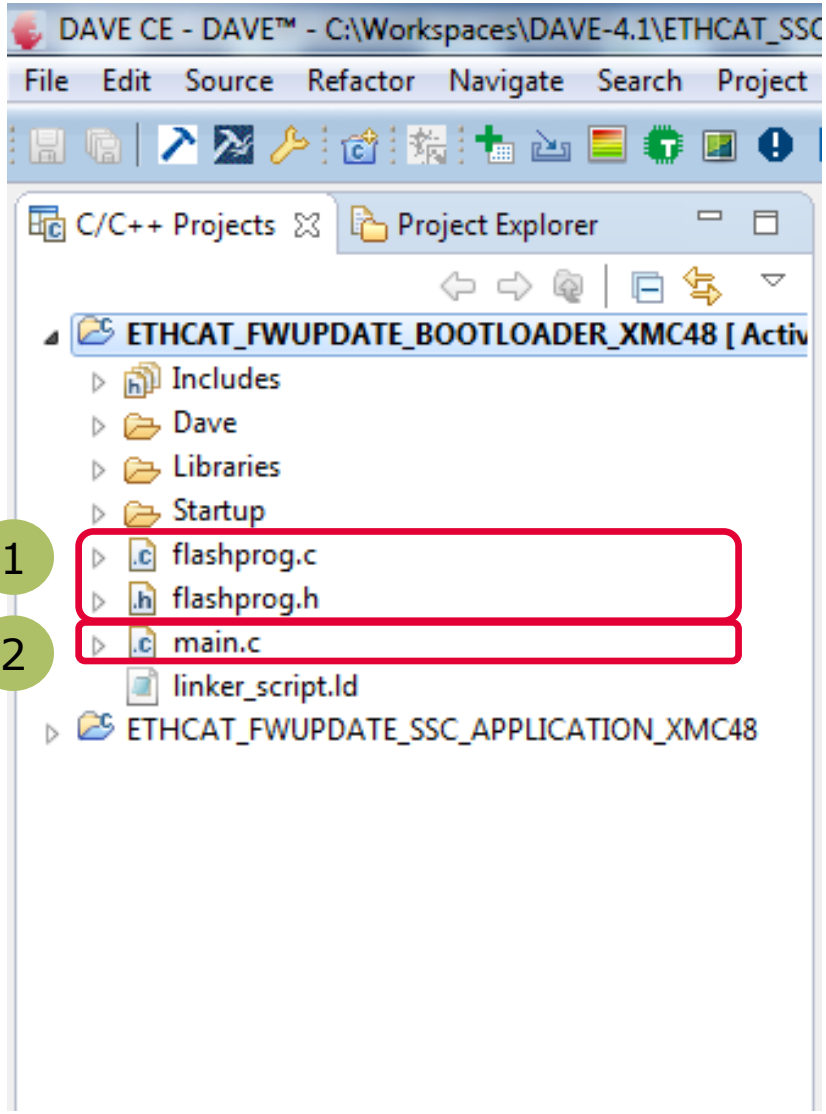
Setup – Imported application example



6 The zip-file to import the 2nd project you find in the application project you just have imported.

Import this project in the same way you just have imported the application project.

Setup – Imported bootloader example



After the project import you will find the project for bootloader executable:

1. Utility files to support the flash programming
2. Main.c of bootloader application

The following slides show the architecture, how the applications interact and the essential implementation details behind bootloader and application executable.

1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

Boot modes – Overview

Inside the setup section of this documentation, you already imported two independent projects into DAVE. The executables resulting from these projects have to be stored inside flash and are interacting with each other:

1. **Application executable**
   (ETHCAT_FWUPDATE_SSC_APPLICATION_XMC48)

   Executes EtherCAT™ slave stack

   Downloads new firmware into backup partition via FoE

2. **Bootloader executable**
   (ETHCAT_FWUPDATE_BOOTLOADER_XMC48)

   Checks the availability of new firmware inside backup partition

   If available, updates application executable with new firmware

→ XMC supports a variety of boot modes for execution of independent executables. See the following slides to get a short overview on the boot modes which are used inside this example.

Boot modes – Overview

Table 27-2 System reset boot modes

SWCON[3:0]	Boot mode
0000_B	Normal
0001_B	ASC BSL
0010_B	BMI
0011_B	CAN BSL
0100_B	PSRAM boot
1000_B	ABM-0
1100_B	ABM-1
1110_B	Fallback ABM

1

2

XMC4800 supports different boot modes.

1 These boot modes can be selected by driving the boot mode pins (JTAG TCK TMS) with appropriate logic levels and issuing a system or power on reset: Normal, ASC BSL (UART bootstrap), CAN BSL (CAN bootstrap), BMI

2 By preparing a header information inside flash and issuing a system reset (via software) these boot modes can be used in addition. These boot modes cannot be entered with power on reset:
Alternative Boot modes (ABM-0/ABM-1), PSRAM boot, FallbackABM

Boot modes – normal and alternative boot mode 0

Table 27-2 System reset boot modes

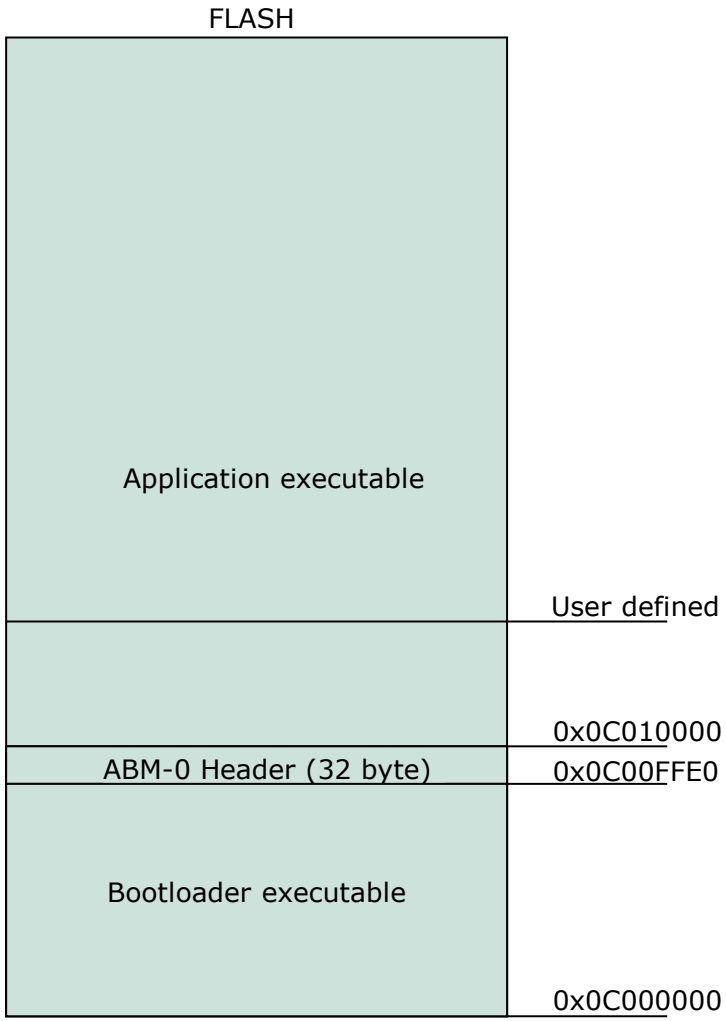
SWCON[3:0]	Boot mode
0000_B	Normal
0001_B	ASC BSL
0010_B	BMI
0011_B	CAN BSL
0100_B	PSRAM boot
1000_B	ABM-0
1100_B	ABM-1
1110_B	Fallback ABM

For this example these boot modes are used:

Normal boot mode (after system or power on reset)
An application located at start of the flash is executed.

Alternative boot mode 0 (after system reset)
An application located at a user defined address inside flash is executed. The address is defined inside the „ABM Header".
ABM Header must be located at last 32bytes (0x0C00FEE0) of the first 64kB physical sector inside flash.

Boot modes – exemplary memory outline



The application executable is located at a flash address defined inside ABM-0 Header(@0x0C00FFE0):

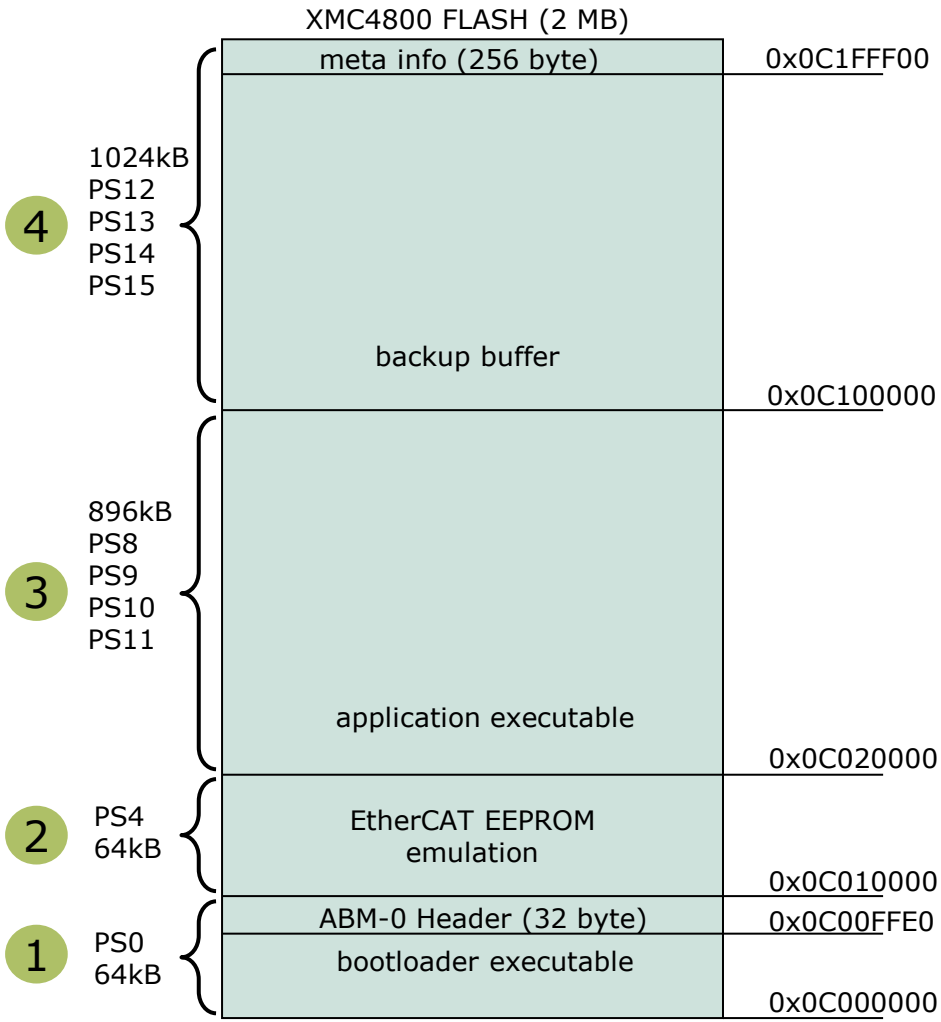
- Application is never executed directly after power on reset
- Application execution can only be triggered by a system reset from bootloader executable

The bootloader executable is located at flash address 0x0C000000:

- Bootloader execution is triggered by power on or system reset .
- Bootloader starts the application executable or proceeds with bootloader functionality.

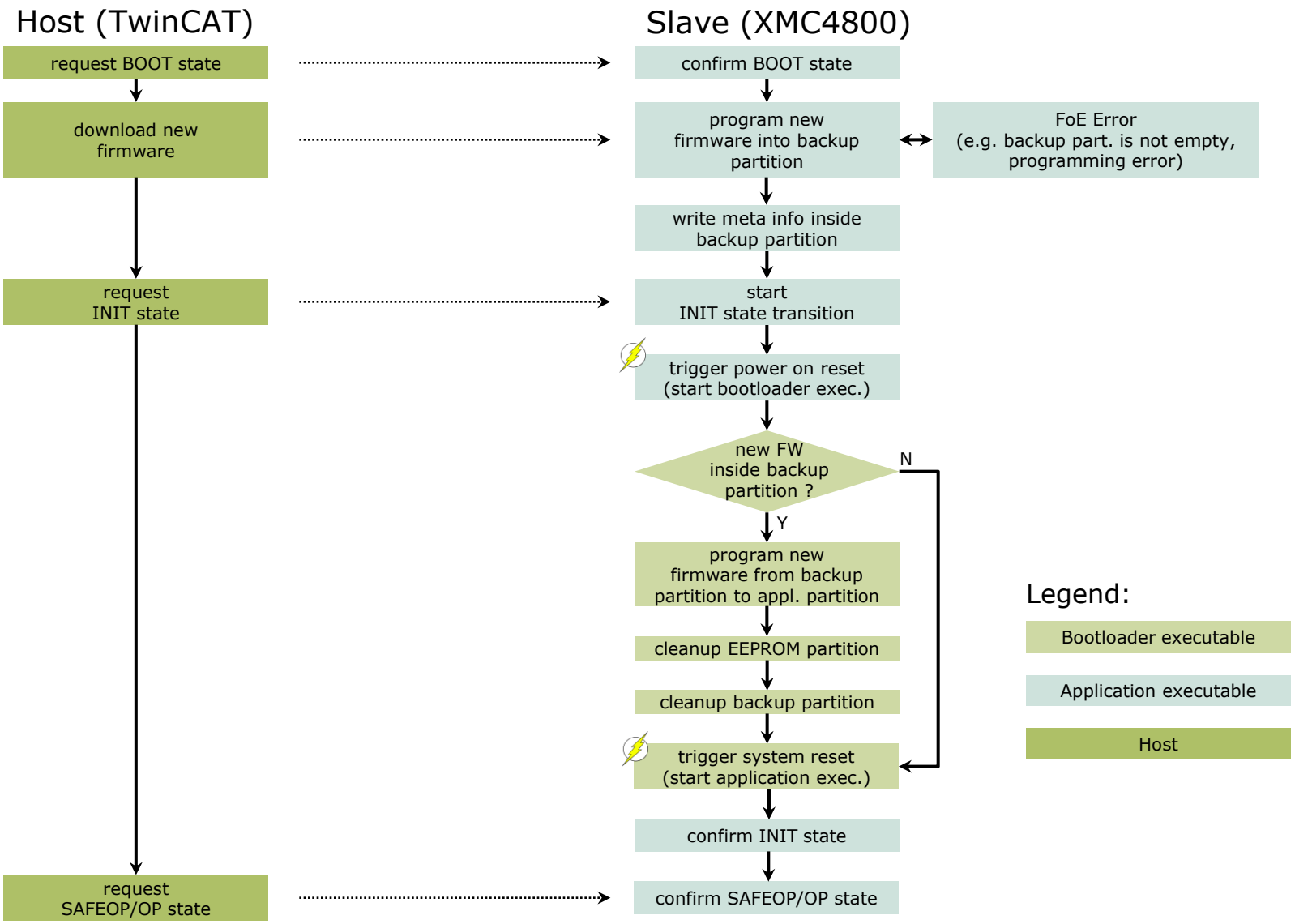
1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

Architecture – Flash partitioning



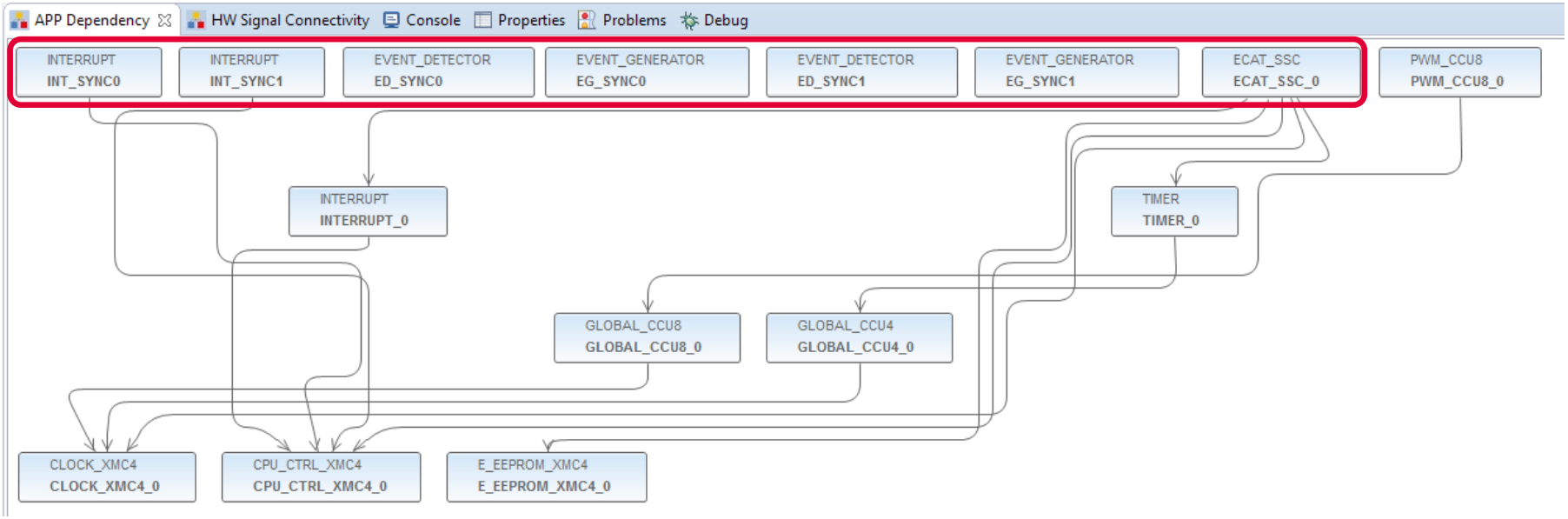
1. Bootloader executable is always executed first after each power on reset. Checks availability of new firmware inside backup buffer. If new firmware is available, it is copied from backup buffer to application executable partition incl. error checks. After cleaning of backup and EEPROM emulation partition, restarts the device with application executable.
2. EtherCAT EEPROM emulation. If empty, is initialized by ECAT_SSC APP with the slaves default settings during first init.
3. Application executable implements EtherCAT™ SSC stack. Downloads new firmware to backup partition via FoE incl. error checks. If success, issues power on reset to trigger reprogramming of application by bootloader executable.
4. Backup buffer used to store new firmware by application executable. Last flash page is reserved for meta info. Today only size of new FW is stored inside meta info. Backup buffer is checked by bootloader to reprogram application executable after integrity of firmware is proven.

Architecture – Host/Slave interaction states for firmware update



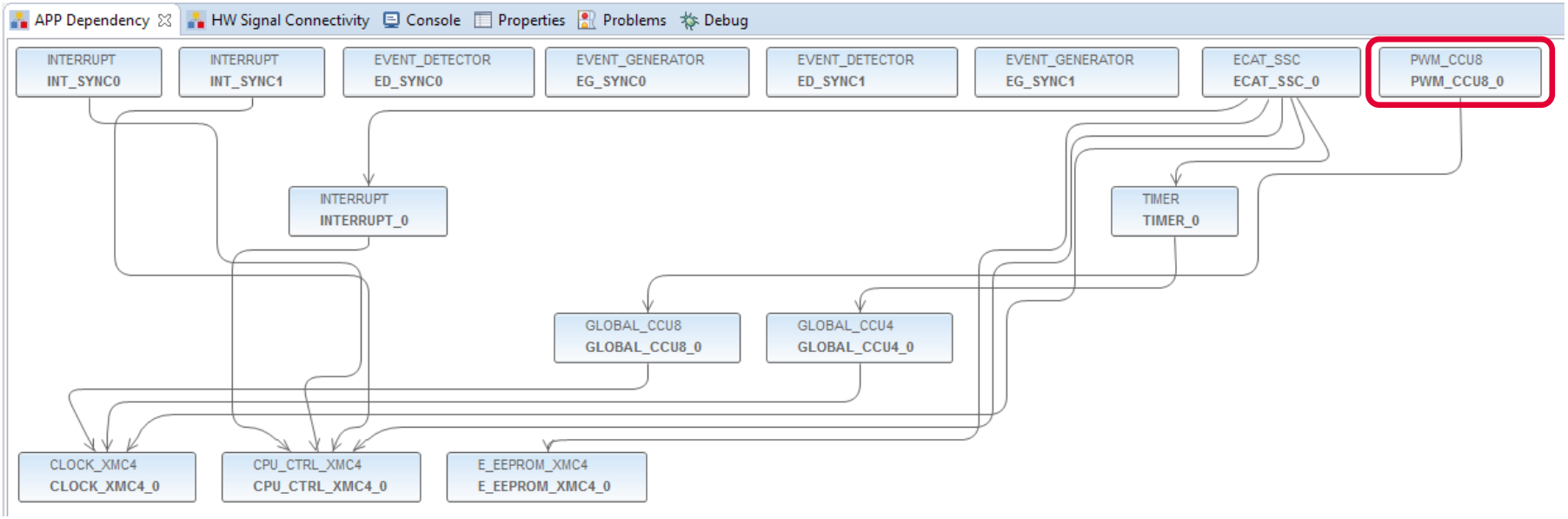
1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

Application – Overview on used APPs



Same like for the basic example, the ECAT_SSC APP assigns the system resources (automatically done by DAVE using the respective lower level apps) and pins (by manual configuration) to setup a proper EtherCAT communication. The EVENT_DETECTOR, EVENT_GENERATOR and INTERRUPT APPs are used to connect the sync_out_0 and sync_out_1 of the ECAT_SSC APP to the interrupt service routines of the SSC-stack.

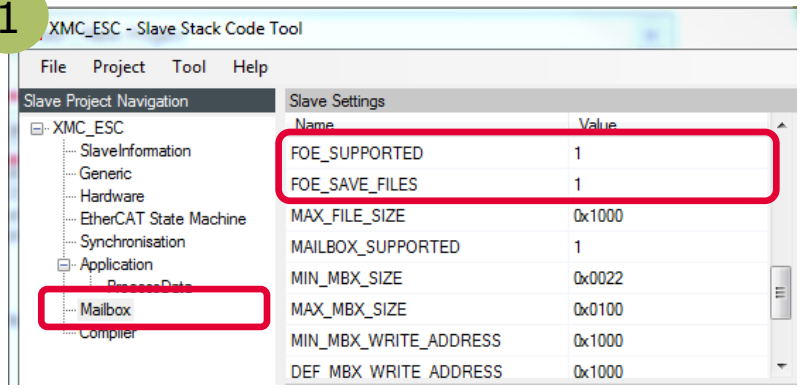
Application – Overview on used APPs



The basic example used PWM_CCU8 APP to control the dimming level of the LED2 on your Relax Kit by TwinCAT host. This example uses PWM_CCU8 APP to set a constant flashing rate of either 2Hz or 20Hz to LED2. The two different application binaries are then used to test firmware update functionality. This allows to visualize the successful firmware update by just checking the flash rate of LED2.

Application – SSC Tool
EtherCAT™ stack configuration

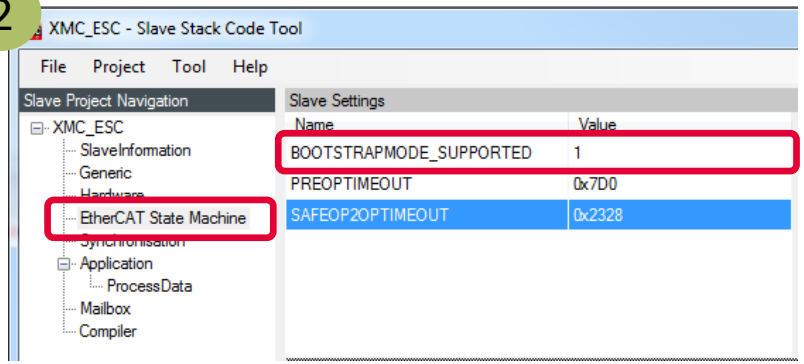
1



1 Enable FoE support including saving of files

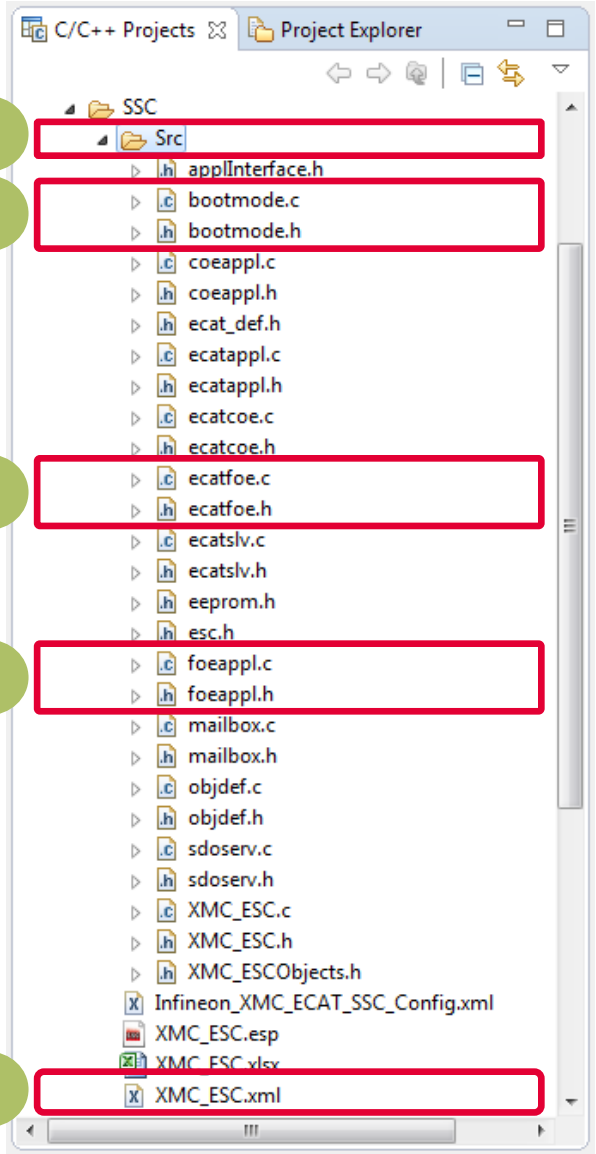
2 Enable bootstrap mode support

2



→ To generate the slave stack code and ESI-file with this configuration, please see the documentation of the basic example.

Application – SSC Tool
Find and use your result



After the code generation you find the respective files inside the project:

3 Check the availability of the generated slave stack code inside Src folder

4 Because you configured FoE and bootstrap mode, these files are added to slave stack code.

5 Check the availability of the ESI-file and download to the host by these 3 steps:

1. Stop TwinCAT System Manager
2. Copy the ESI file to resp. destination
   for TwinCAT2:
   C:\TwinCAT\Io\EtherCAT
   for TwinCAT3:
   C:\TwinCAT\3.1\Config\Io\EtherCAT
3. Restart TwinCAT System Manager to start re-work of the device description cache.

Application – Extend slave stack code with calls into your application

Inside the generated file *XMC_ESC.c* the function APPL_Application is implemented. This function is executed cyclic and implements the application specific code
A) … from mainloop or
B) … if synchronisation is active from ISR

Inside main.c of the example, the function
*void process_app(TOBJ7000 *OUT_GENERIC, TOBJ6000 *IN_GENERIC);*
implements the mapping of the input/output data to buttons and LEDs. In addition it checks if a firmware update has finished. If yes, it triggers a system reset to start bootloader executable. Modify the function APPL_Application to call process_app in the following way:

Originally generated code:

```
/////////////////////////////////////////////////////////////////////////////////////////
/**
\brief    This function will called from the synchronisation ISR
          or from the mainloop if no synchronisation is supported
*/////////////////////////////////////////////////////////////////////////////////////////
void APPL_Application(void)
{
#if _WIN32
   #pragma message ("Warning: Implement the slave application")
#else
    #warning "Implement the slave application"
#endif
}
```

Modified code:

```
/////////////////////////////////////////////////////////////////////////////////////////
/**
\brief    This function will called from the synchronisation ISR
          or from the mainloop if no synchronisation is supported
*/////////////////////////////////////////////////////////////////////////////////////////
void process_app(TOBJ7000 *OUT_GENERIC, TOBJ6000 *IN_GENERIC);
void APPL_Application(void)
{
   process_app(&OUT_GENERIC0x7000, &IN_GENERIC0x6000);
}
```

Application – Extend slave stack code with calls into your application

Inside the generated file *XMC_ESC.c* file the function APPL_StopMailboxHandler is implemented. This function is called when slave is requested to INIT state.

The function *void FWUPDATE_StateTransitionInit(void)*
of the example implementation is called to indicate the slave state changes from BOOT to INIT.

Originally generated code:

```
80  ////////////////////////////////////////////////////////////////////////////////////////
81  /**
82   \return     0, NOERROR_INWORK
83
84   \brief    The function is called in the state transition from PREEOP to INIT
85             to stop the mailbox handler. This functions informs the application
86             about the state transition, the application cannot refuse
87             the state transition.
88
89  *////////////////////////////////////////////////////////////////////////////////////////
90
91  UINT16 APPL_StopMailboxHandler(void)
92  {
93      return ALSTATUSCODE_NOERROR;
94  }
```

Modified code:

```
81  ////////////////////////////////////////////////////////////////////////////////////////
82  /**
83   \return     0, NOERROR_INWORK
84
85   \brief    The function is called in the state transition from PREEOP to INIT
86             to stop the mailbox handler. This functions informs the application
87             about the state transition, the application cannot refuse
88             the state transition.
89
90  ////////////////////////////////////////////////////////////////////////////////////////
91  void FWUPDATE_StateTransitionInit(void);
92  UINT16 APPL_StopMailboxHandler(void)
93  {
94      FWUPDATE_StateTransitionInit();
95      return ALSTATUSCODE_NOERROR;
96  }
```

Application – Extend slave stack code with calls into your application

1

foeappl.c

```
172 UINT16 FOE_Write(UINT16 MBXMEM * pName, UINT16 nameSize, UINT32 password)
173 {
174   /* firmware update only in boot mode
175    * boot mode active ? */
176   if ( bBootMode )
177   {
178     /* password valid ? */
179     if (password == 0xBEEFBEEF)
180     {
181       /* start firmware update */
182       FWUPDATE_StartDownload();
183       return 0;
184     }
185     else
186     {
187       /* wrong password */
188       return ECAT_FOE_ERRCODE_ILLEGAL;
189     }
190   }
191   else
192     /* no files are stored/accepted */
193     return ECAT_FOE_ERRCODE_DISKFULL;
194 }
```

2

foeappl.c

```
219 UINT16 FOE_Data(UINT16 MBXMEM * pData, UINT16 Size)
220 {
221   /* boot mode active ?
222    * -> proceed with firmware update
223    */
224   if ( bBootMode )
225   {
226     /* forward data to firmwareupdate handler */
227     return FWUPDATE_Data(pData, Size);
228   }
229   else
230   /* no other files than firmware data are stored/accepted */
231     return ECAT_FOE_ERRCODE_DISKFULL;
232 }
```

3

foeappl.c

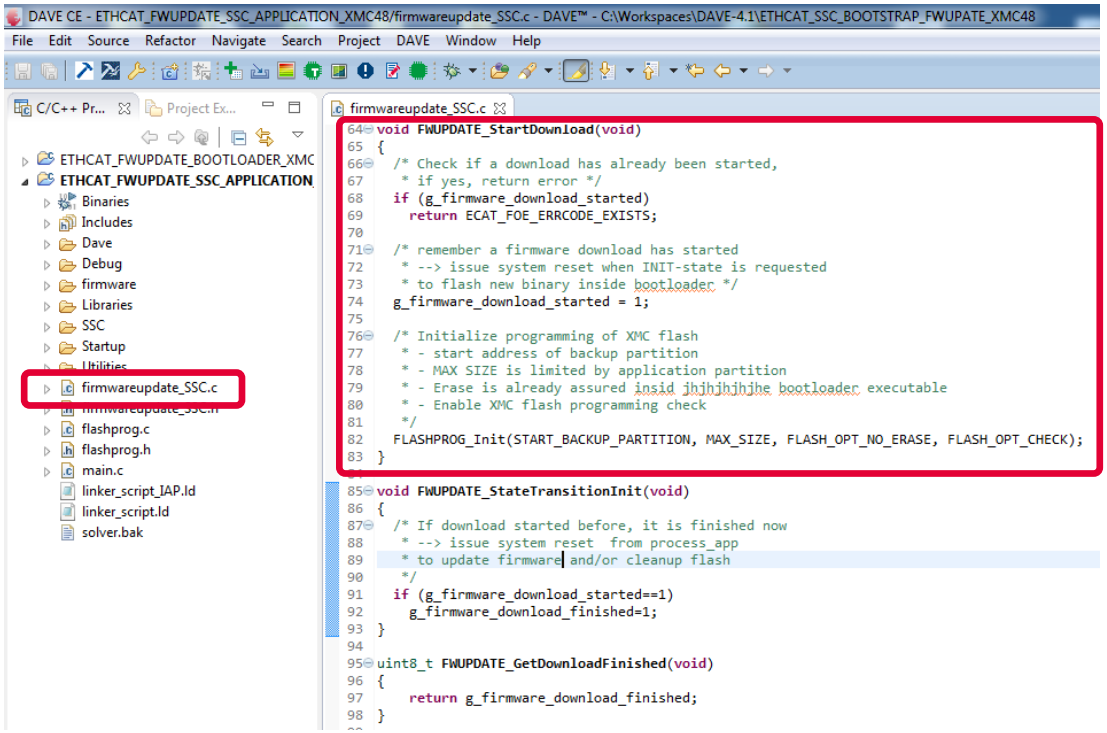
```
46 #define _FOEAPPL_ 1
47     #include "foeappl.h"
48 #undef _FOEAPPL_
49 #define _FOEAPPL_ 0
50
51 #include "bootmode.h"
52 #include "firmwareupdate_SSC.h"
53
```

Inside the generated file foeappl.c your application specific code for the FoE functionality is implemented.

Every FoE write transaction starts with a call to FOE_Write and is continued with calls to FOE_Data. Here you add the calls to the firmware update example implementation:

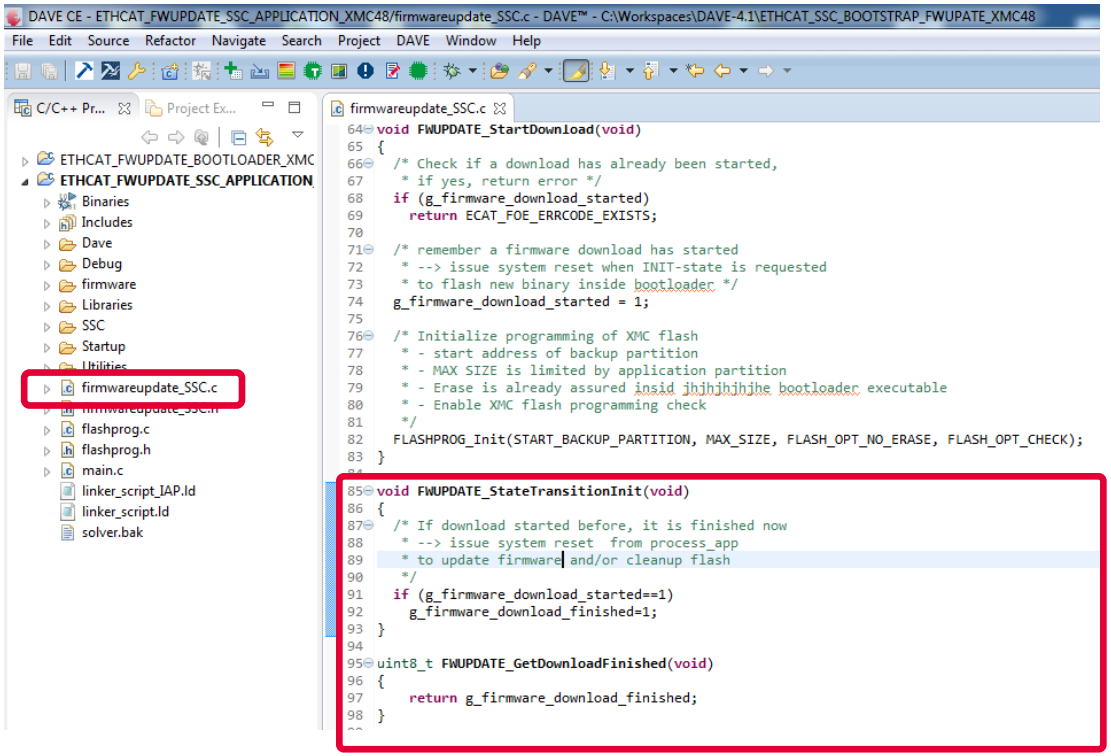
1. Check if BOOT state is active and password matches(optional). If yes, start download by calling example implementation, otherwise return error.
2. Check if BOOT state is still active. If yes, forward data to example implementation, otherwise return error.
3. Include header file to firmware update example implementation.

Application – Firmware update example implementation



FWUPDATE_StartDownload is called from SSC stack when a new firmware update in BOOT state is initiated. Check if update was already started in the past. In this case return error, because flash is no more clean. If flash is clean, initialize flash programming to store data inside backup partition.

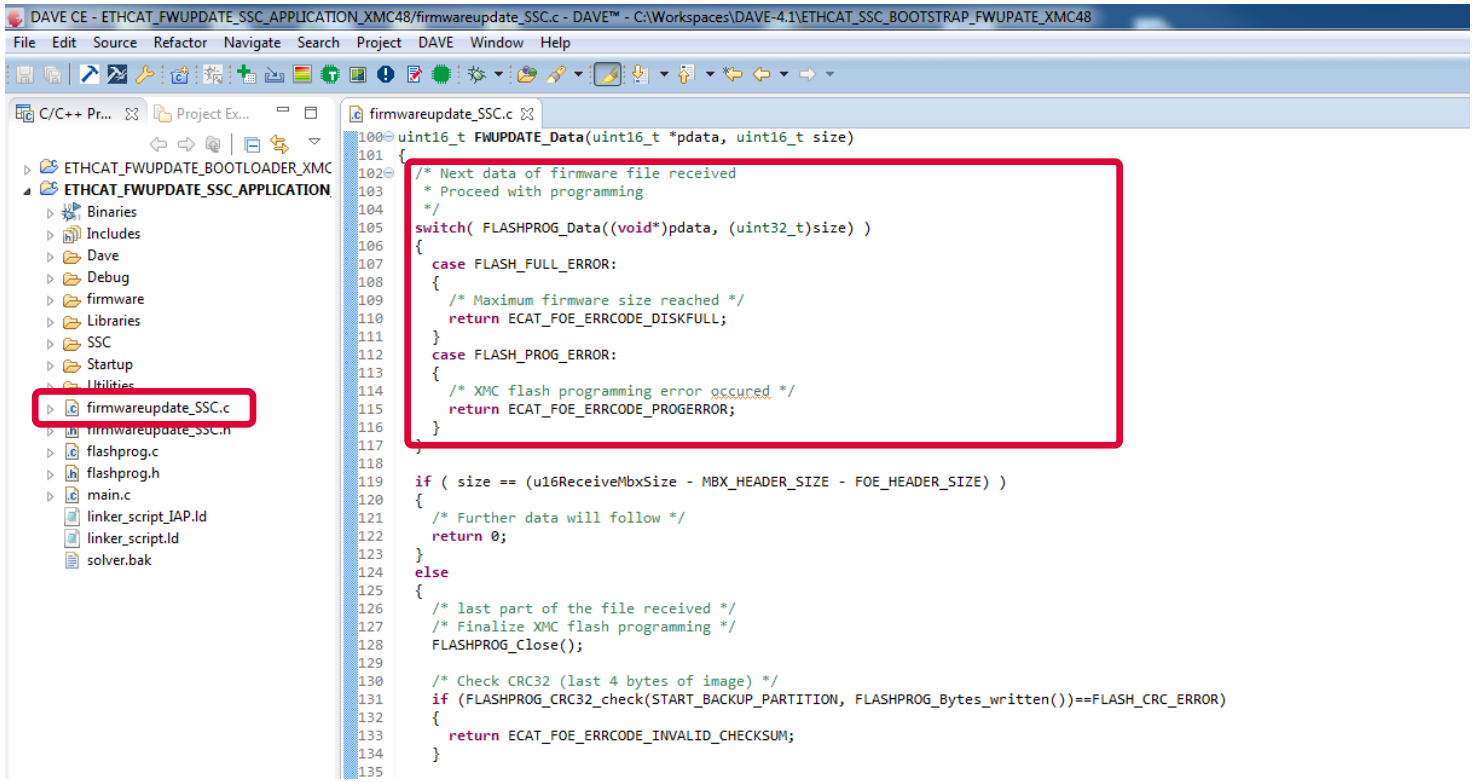
Application – Firmware update example implementation



FWUPDATE_StateTransitionInit is called from SSC stack when state changed to INIT. If a download was started before, it has finished now.

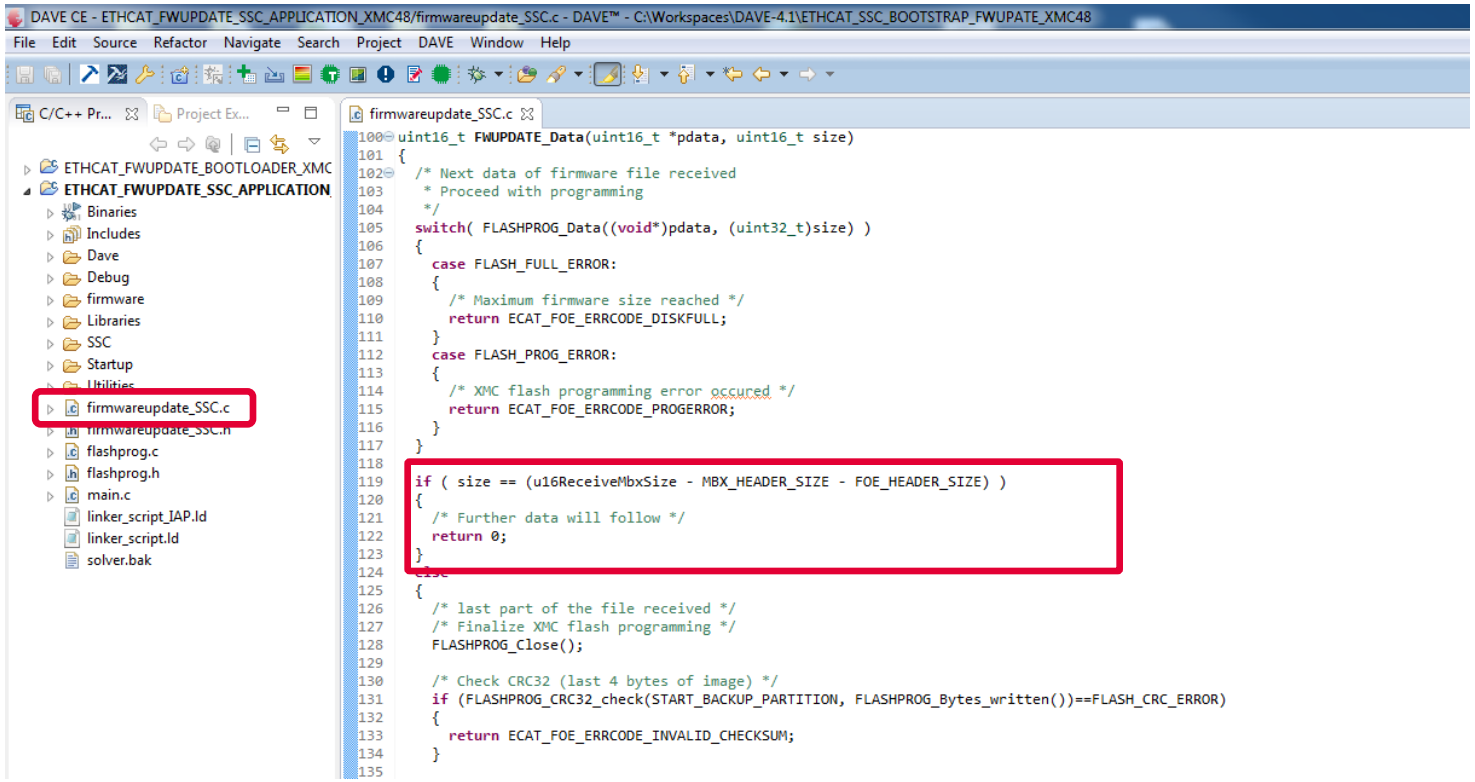
FWUPDATE_GetDownloadFinished is used to check download state inside process_app. Used to trigger a system reset to start bootloader for updating firmware (copy backup partition to application partition).

Application – Firmware update example implementation



FWUPDATE_Data is called from SSC stack sequentially for all the file data. Forward data to flash programming. Return error if flash is full or programming error occured.

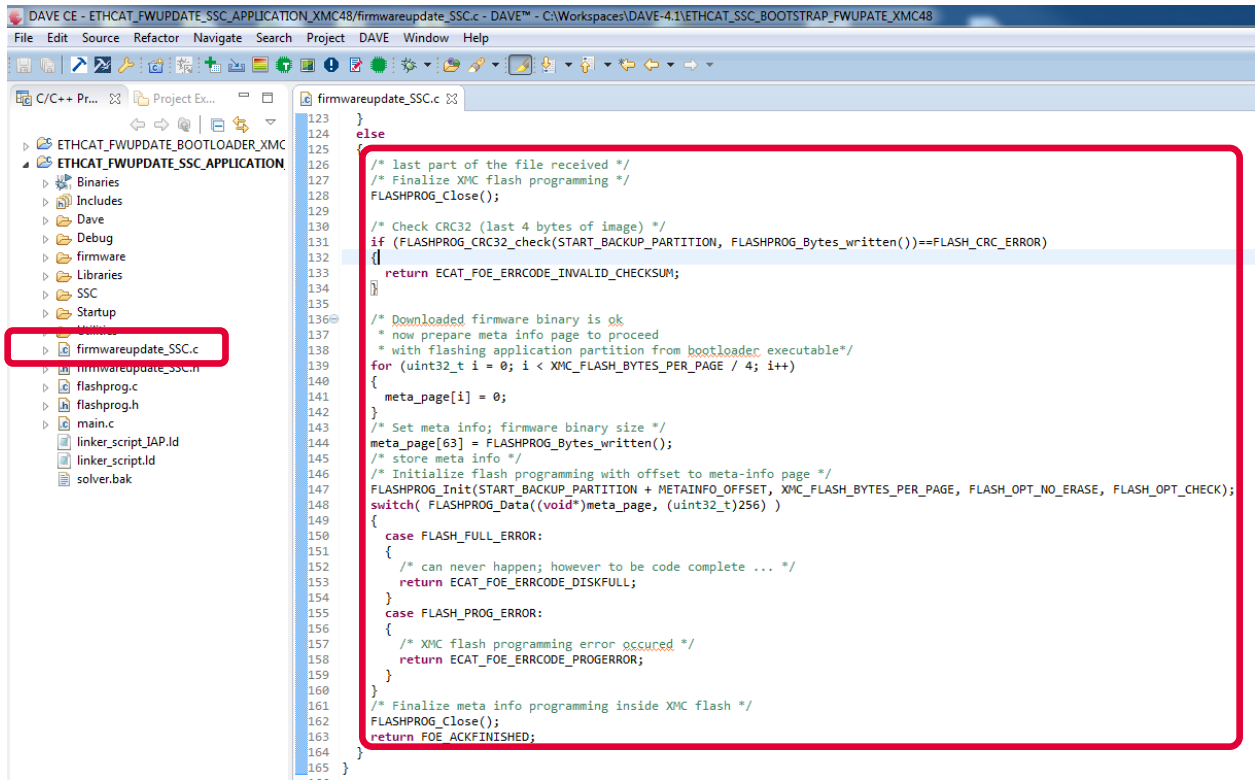
Application – Firmware update example implementation



FWUPDATE_Data (cont'd)

Use fill status of mailbox to check, if file transfer is complete.
If file transfer is not complete, return 0 to receive further data.

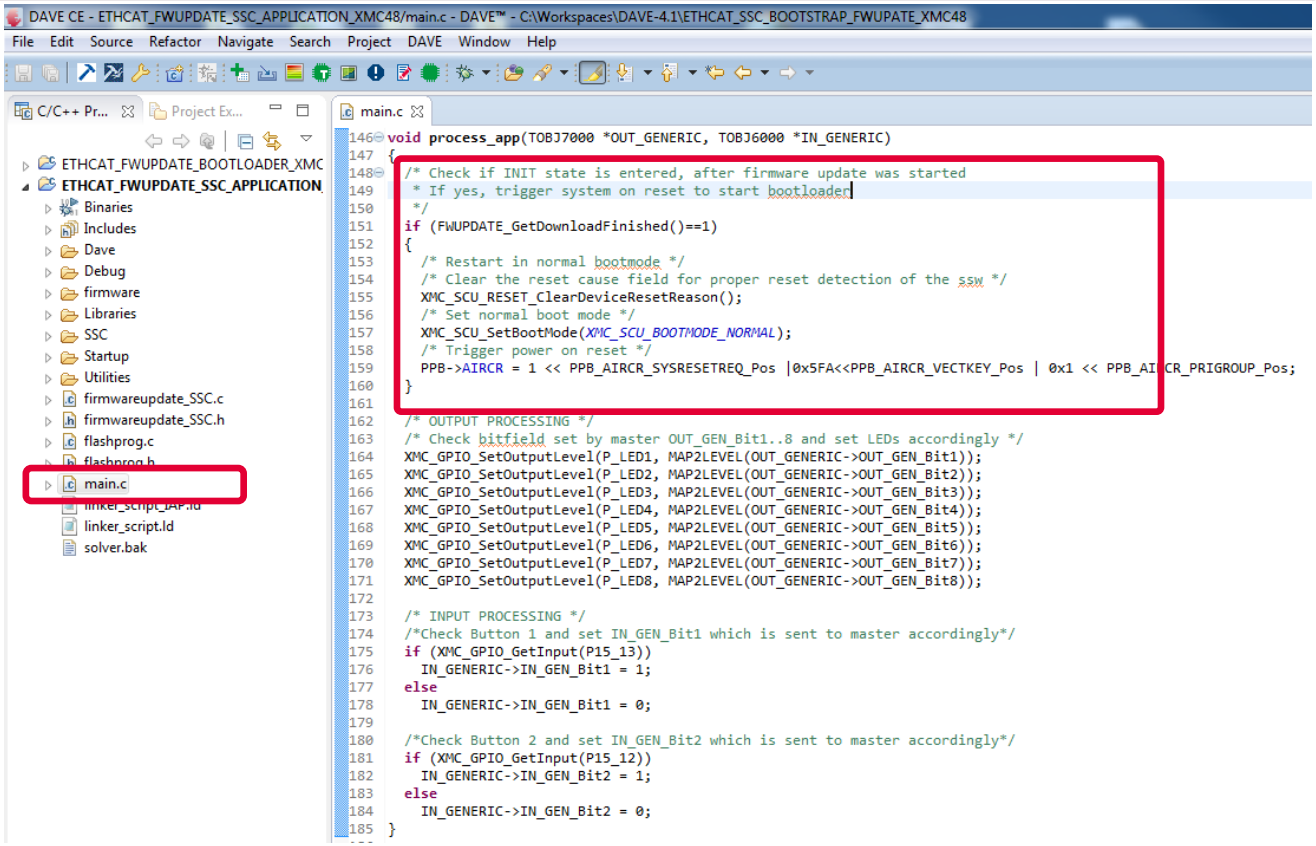
Application – Firmware update example implementation



FWUPDATE_Data (cont'd)

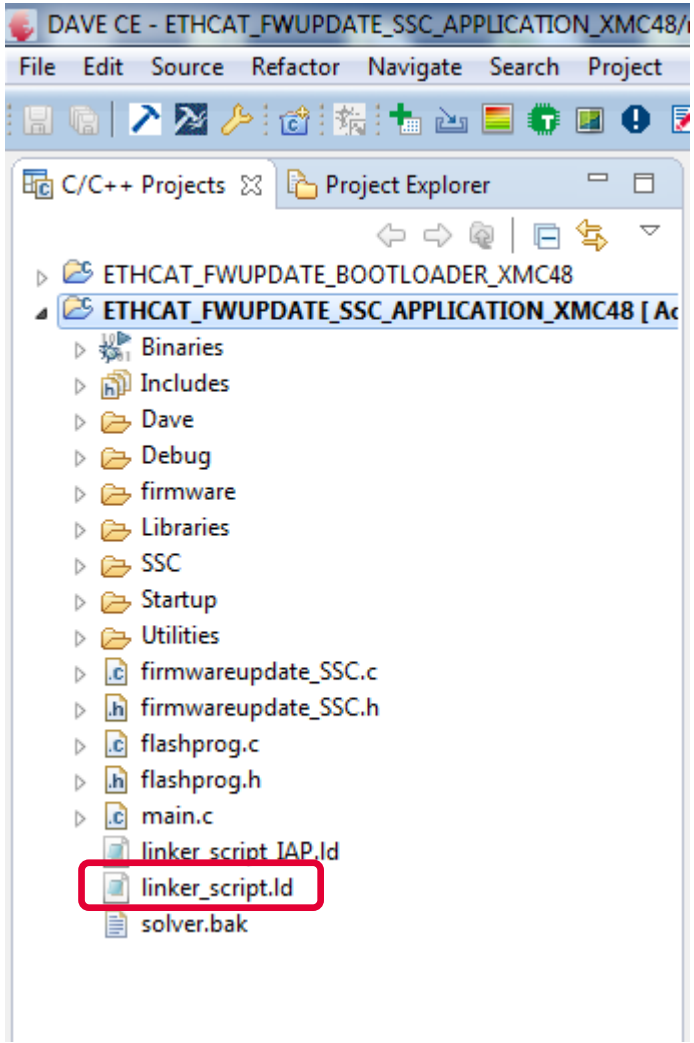
Use fill status of mailbox to check if file transfer is complete.
If file transfer is complete, close current programming. Check CRC32 of flash content (last 4 bytes of binary must carry CRC) and return error if needed. Prepare page data (256 byte) for meta info. Write meta info to flash. Return code for successful file transfer FOE_ACKFINSIHED.

Application – Trigger system reset to restart with bootloader executable



process_app is cyclic called from SSC stack to implement the application specific behaviour (e.g. I/Os). For the firmware update example here it is checked, if backup partition was modified and INIT state entered. If yes, trigger system reset to start bootloader for programming new firmware and/or cleanup backup partition.

Application – Modifying the linker file



Inside the linker file the start address of your application is defined.

The default linker file of DAVE projects defines the location of the vector table and program/data to the flash start address:

0x0C000000

To reserve space for EEPROM emulation the ECAT_SSC APP used inside your project, overwrites this default linker file with every code generation to remap the program/data start address:

0x0C000000 vector table
0x0C020000 program/data

This setting does not match to flash partitioning used for this example, because the vector table overlaps with the bootloader partition.

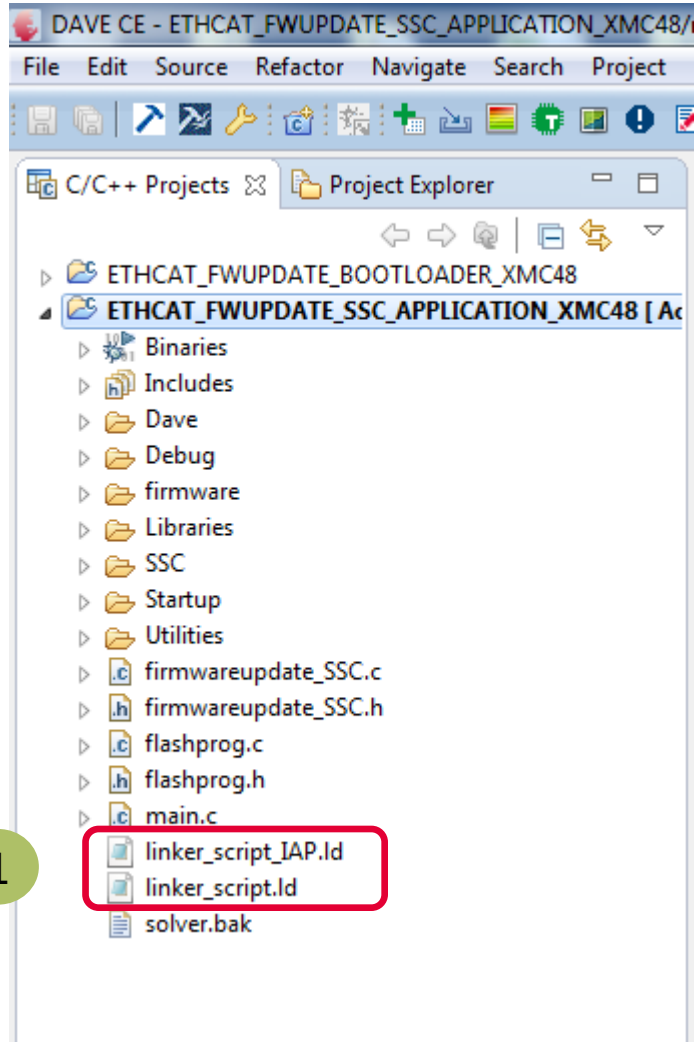
Instead, the following setting is needed:

0x0C020000 vector table + program/data

→ **The linker file must be modified and overwriting with every APP code generation must be avoided.**

See the following slides to do this

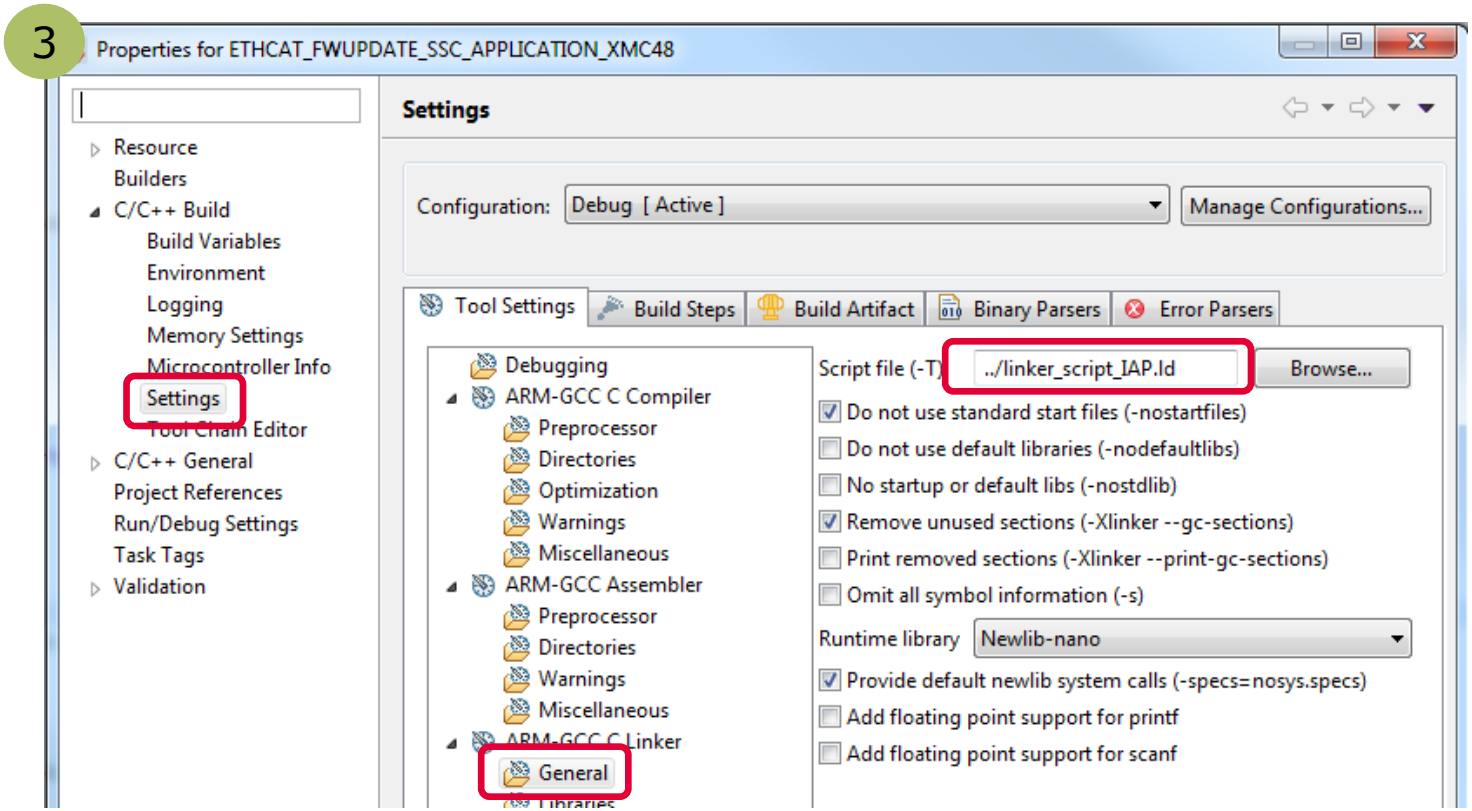
Application – Modifying the linker file



To avoid regular overwriting of the linker file by code generation of the ECAT_SSC APP, follow these steps:

1. Create a copy of the linker file and rename the copy:
   „linker_script_IAP.ld“
2. Open project properties:

Application – Modifying the linker file



3 Define name of linker file: „linker_script_IAP.ld“

Application – Modifying the linker file

linker_script_IAP.ld | linker_script.ld

```
63 MEMORY
64 {
65   FLASH_0_cached(RX) : ORIGIN = 0x08000000, LENGTH = 0x00010000
66   FLASH_0_uncached(RX) : ORIGIN = 0x0C000000, LENGTH = 0x00010000
67   FLASH_1_cached(RX) : ORIGIN = 0x08020000, LENGTH = 0x001E0000
68   FLASH_1_uncached(RX) : ORIGIN = 0x0C020000, LENGTH = 0x001E0000
69   PSRAM_1(!RX) : ORIGIN = 0x1FFE8000, LENGTH = 0x18000
70   DSRAM_1_system(!RX) : ORIGIN = 0x20000000, LENGTH = 0x20000
71   DSRAM_2_comm(!RX) : ORIGIN = 0x20020000, LENGTH = 0x20000
72   SRAM_combined(!RX) : ORIGIN = 0x1FFE8000, LENGTH = 0x00058000
73 }
74
75 SECTIONS
76 {
77   /* TEXT section */
78
79   .reset :
80   {
81     KEEP(*(.reset));
82   } > FLASH_0_cached AT > FLASH_0_uncached
83
84   .text :
85   {
86     sText = .;
87     *(.text .text.* .gnu.linkonce.t.*);
88
89     /* C++ Support */
90     KEEP(*(.init))
91     KEEP(*(.fini))
92
93     /* .ctors */
```

4

linker_script_IAP.ld | linker_script.ld

```
63 MEMORY
64 {
65   FLASH_0_cached(RX) : ORIGIN = 0x08000000, LENGTH = 0x00010000
66   FLASH_0_uncached(RX) : ORIGIN = 0x0C000000, LENGTH = 0x00010000
67   FLASH_1_cached(RX) : ORIGIN = 0x08020000, LENGTH = 0x001E0000
68   FLASH_1_uncached(RX) : ORIGIN = 0x0C020000, LENGTH = 0x001E0000
69   PSRAM_1(!RX) : ORIGIN = 0x1FFE8000, LENGTH = 0x18000
70   DSRAM_1_system(!RX) : ORIGIN = 0x20000000, LENGTH = 0x20000
71   DSRAM_2_comm(!RX) : ORIGIN = 0x20020000, LENGTH = 0x20000
72   SRAM_combined(!RX) : ORIGIN = 0x1FFE8000, LENGTH = 0x00058000
73 }
74
75 SECTIONS
76 {
77   /* TEXT section */
78
79   .text : ALIGN (4)
80   {
81     sText = .;
82     KEEP(*(.reset));
83     *(.text .text.* .gnu.linkonce.t.*);
84
85     /* C++ Support */
86     KEEP(*(.init))
87     KEEP(*(.fini))
88
89     /* .ctors */
90     *crtbegin.o(.ctors)
91     *crtbegin?.o(.ctors)
```

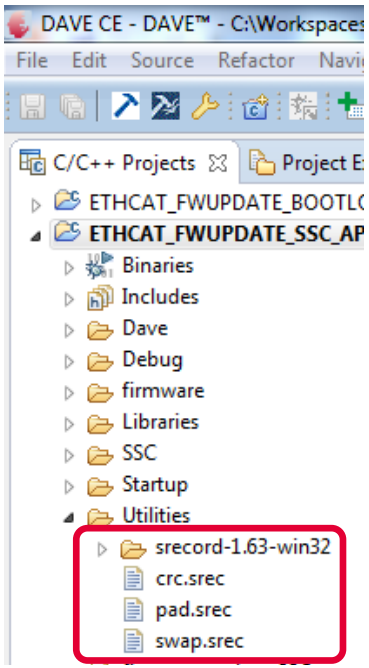
4 Modify the new linker file. Move assignment of vector table to text section inside „linker_script_IAP.ld“

Application – Creating the binary output including CRC32

The firmware update files used inside this example have the following format:

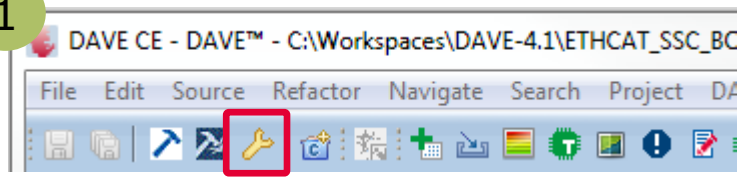
1. Binary output format
2. The size of binary data must be a multiple of 4 bytes (word aligned)
3. The last word should represent the CRC32 of the previous binary data

→ SRecord is used to automate the formatting of the binary output.
The windows executable of SRecord and prepared scripts are provided inside the example to get the job done.
See the following slides how the tool is integrated into the output linking stage of DAVE™.

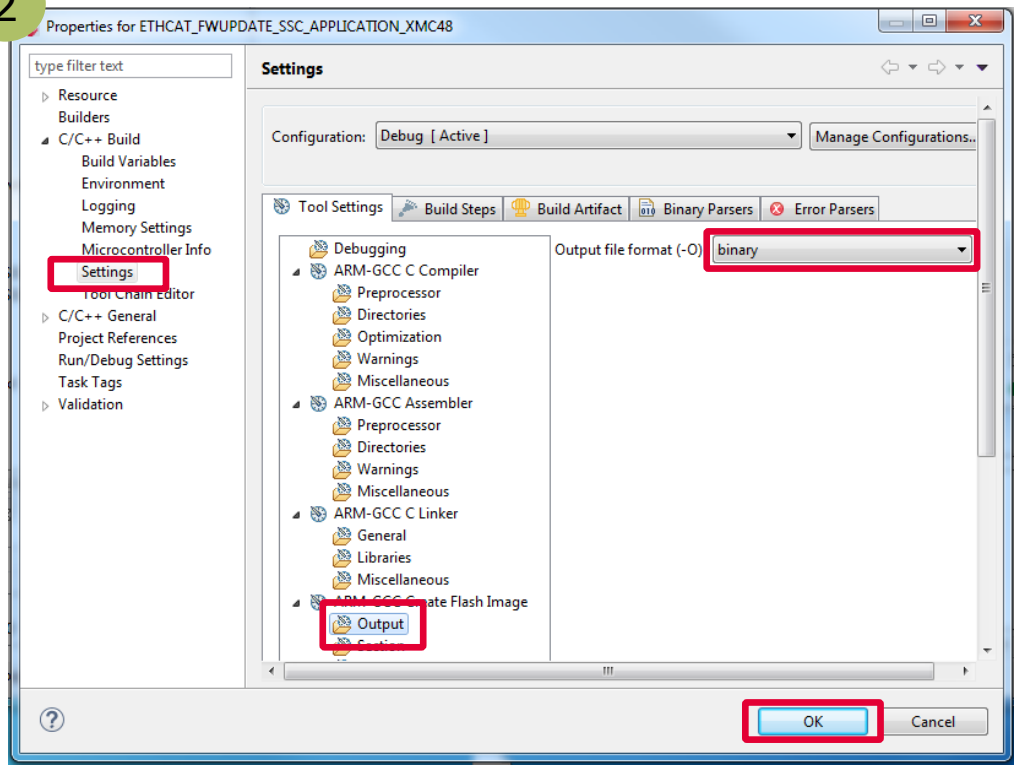


Application – Creating the binary output including CRC32

1



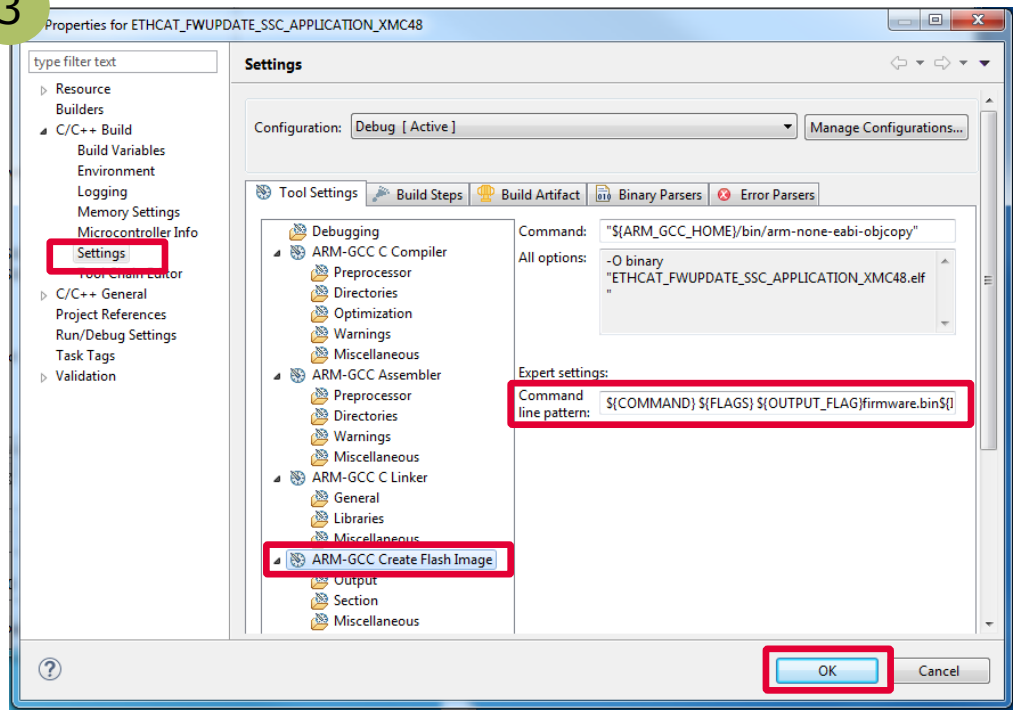
2



1. Open project properties dialog
2. Set output format to binary format

Application – Creating the binary output including CRC32

3



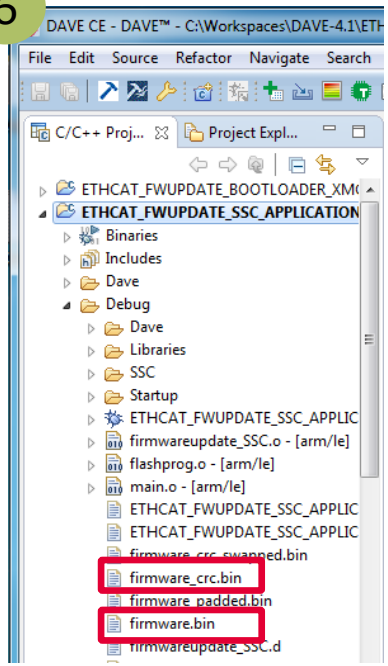
3 Modify the command line pattern for post processing:
${COMMAND} ${FLAGS} ${OUTPUT_FLAG}firmware.bin${INPUTS} &
"../Utilities/srecord-1.63-win32/srec_cat" "@../Utilities/pad.srec" &
"../Utilities/srecord-1.63-win32/srec_cat" "@../Utilities/crc.srec" &
"../Utilities/srecord-1.63-win32/srec_cat" "@../Utilities/swap.srec"

Application – Creating the binary output including CRC32

4

5

6



4 Regenerate the source code for DAVE™ APPs

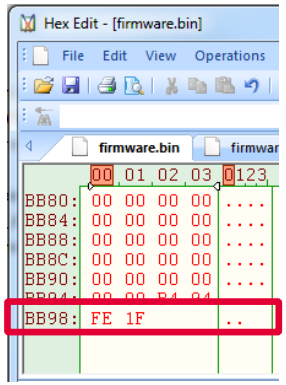
5 Rebuild the project

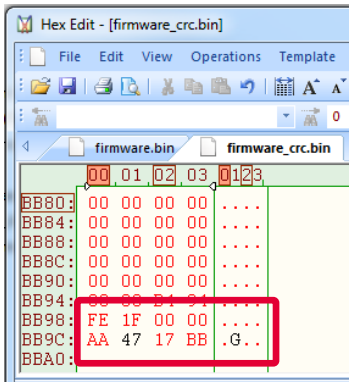
6 Find the result inside debug folder:

firmware.bin
- binary output unmodified

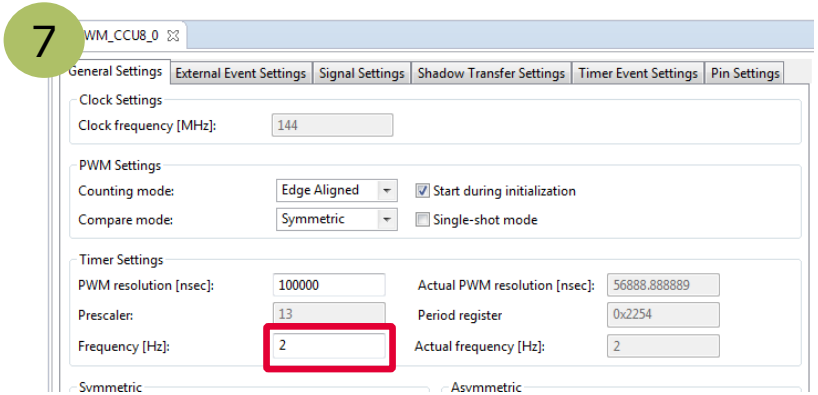
firmware_CRC.bin
- binary output padded to word alignment
- CRC32 added.

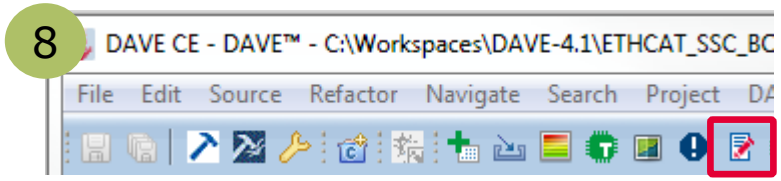
Use a hex editor to visualize the diff:

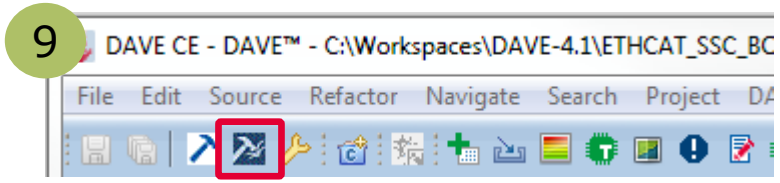


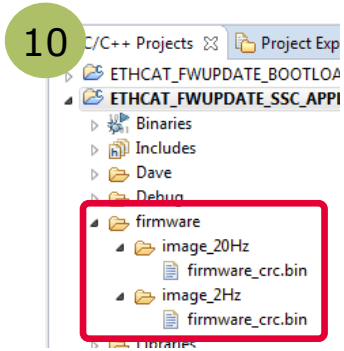


Application – Creating the binary output including CRC32 for testing









Inside the test section of this documentation two firmware update binaries are needed:

- LED2 flashing with 2Hz
- LED2 flashing with 20Hz

To prepare the two binaries set the flash frequency inside PWM_CCU8 APP to 20Hz/2Hz and create the binaries:

7 Set frequency

8 Generate source code for APPs

9 Rebuild the project

10 For your convinience the two firmware update binaries are already prepared and are delivered within the project.

1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

Bootloader – implementing main()

main.c

```
uint32_t main(void)
{
  uint32_t firmware_size_bytes;
  uint32_t* ptr_backupdata = 0;
firmware_size_bytes = (START_ADDRESS_BACKUP_PARTITION + METAINFO_OFFSET / 4)[63];

/* check metainfo if firmware for update is available inside backup partition */
if ( (firmware_size_bytes > 0) && (firmware_size_bytes < APP_PARTITION_MAX_SIZE) )
{
  /* metainfo indicates a new firmware is available inside backup partition */
  /* check CRC32 of firmware inside backup partition */
  if (FLASHPROG_CRC32_check(START_ADDRESS_BACKUP_PARTITION,
                            firmware_size_bytes) == FLASH_OK)
  {
    /* CRC32 of backup partition is OK - start programming */

    /* program new firmware into application partition */
    FLASHPROG_Init(START_ADDRESS_APP_PARTITION,
                   APP_PARTITION_MAX_SIZE,
                   FLASH_OPT_ERASE,
                   FLASH_OPT_CHECK);
    FLASHPROG_Data(START_ADDRESS_BACKUP_PARTITION, firmware_size_bytes);
    FLASHPROG_Close();

    /* delete EEPROM content (2nd 64k sector) */
    FLASHPROG_Delete_physical_sector(XMC_FLASH_PHY_SECTOR_4);

    /* Check CRC32 of firmware inside application partition*/
    if (FLASHPROG_CRC32_check(START_ADDRESS_APP_PARTITION,
                              firmware_size_bytes) != FLASH_OK)
    {
      /* Restart to retry programming */
      BL_Normal_Restart();
    }
    /* OK - new firmware was successfully programmed into application partition */
  }
}

/* CRC32 checked software from backup partition was just successfully programmed
 * into application partition.
 * In any case, make sure backup partition is erased before restarting firmware inside
 * application partition.
 */
ptr_backupdata = START_ADDRESS_BACKUP_PARTITION;
while(ptr_backupdata<START_ADDRESS_BACKUP_PARTITION + BACKUP_PARTITION_MAX_SIZE / 4)
{
  /* is erase needed? */
  if (*ptr_backupdata!=0)
  {
    /* erase backup partition */
    FLASHPROG_Delete_physical_sectors(ptr_backupdata, 1);
  }
  ptr_backupdata++;
}

/* Restart inside application partition */
BL_FlashABM0_Restart();

  return 0;
}
```

1. Check meta info if new firmware binary is available inside backup partition
2. If CRC32 check of backup partition is OK, start programming new software.
   Delete EEPROM content of old firmware. If CRC32 check of new programmed application partition is not OK (which is most unlikely to happen) restart bootloader to retry.
3. Check if backup partition is polluted for any reason. If yes, clean it before starting application executable. By this, backup partition is always clean when application executable is started.
4. Restart device with application executable.

Bootloader – triggering restart in alternative boot mode 0 and normal boot mode

main.c

```
/*****************************************************************
 * API IMPLEMENTATION
 *****************************************************************/
void BL_FlashABM0_Restart(void)
{
  /* Restart in alternative bootmode 0 */
  /* Clear the reset cause field for proper reset detection of the ssw */
  XMC_SCU_RESET_ClearDeviceResetReason();
  /* Set ABM0 as boot mode in SWCON field of STCON register */
  XMC_SCU_SetBootMode(XMC_SCU_BOOTMODE_ABM0);
  /* Trigger power on reset */
  PPB->AIRCR = 1 << PPB_AIRCR_SYSRESETREQ_Pos
                    |0x5FA<<PPB_AIRCR_VECTKEY_Pos
                    | 0x1 << PPB_AIRCR_PRIGROUP_Pos;
}

void BL_Normal_Restart(void)
{
  /* Restart in alternative bootmode 0 */
  /* Clear the reset cause field for proper reset detection of the ssw */
  XMC_SCU_RESET_ClearDeviceResetReason();
  /* Set ABM0 as boot mode in SWCON field of STCON register */
  XMC_SCU_SetBootMode(XMC_SCU_BOOTMODE_NORMAL);
  /* Trigger power on reset */
  PPB->AIRCR = 1 << PPB_AIRCR_SYSRESETREQ_Pos
                    |0x5FA<<PPB_AIRCR_VECTKEY_Pos
                    | 0x1 << PPB_AIRCR_PRIGROUP_Pos;
}
```

1 To restart in alternative boot mode 0, the following actions are needed:

Clear the reset source of the last reset

Set boot mode to alternative boot mode

Issue system reset (software reset)

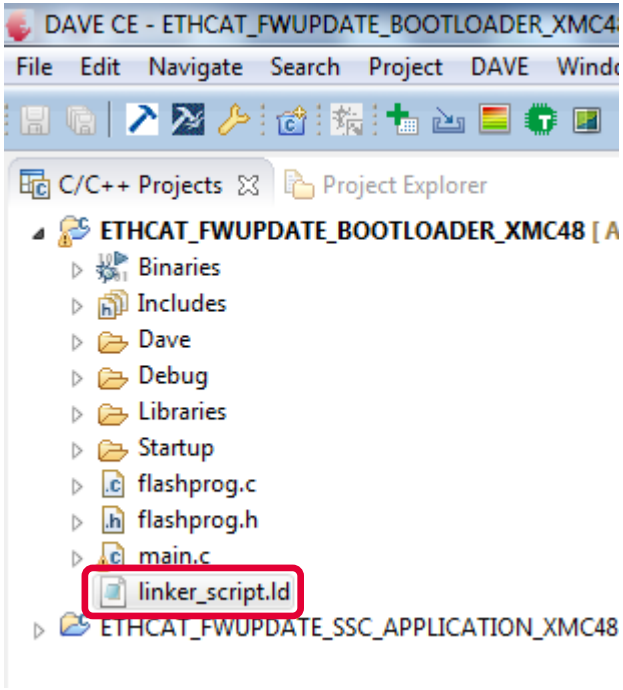
2 To restart in normal boot mode, the following actions are needed:

Clear the reset source of the last reset

Set boot mode to normal boot mode

Issue system reset (software reset)

Bootloader – Limit bootloader flash size inside linker file



The bootloader executable is limited to the first 64KB inside flash.

To make build aware of this limitation, the linker file has to be adapted:

linker_script.ld

```
63 MEMORY
64 {
65   FLASH_1_cached(RX) : ORIGIN = 0x08000000, LENGTH = 0x00010000
66   FLASH_1_uncached(RX) : ORIGIN = 0x0C000000, LENGTH = 0x00010000
67   PSRAM_1(!RX) : ORIGIN = 0x1FFE8000, LENGTH = 0x18000
68   DSRAM_1_system(!RX) : ORIGIN = 0x20000000, LENGTH = 0x20000
69   DSRAM_2_comm(!RX) : ORIGIN = 0x20020000, LENGTH = 0x20000
70   SRAM_combined(!RX) : ORIGIN = 0x1FFE8000, LENGTH = 0x00058000
71 }
```

Bootloader – Defining ABM0 header

main.c

```
/*********************************************************************
 * MACROS AND DEFINES
 ********************************************************************/
/** MagicKey value for \ref ABM_Header_t. see */
#define ABM_HEADER_MAGIC_KEY 0xA5C3E10F
/* start address of backup buffer (@1MByte)*/
#define START_ADDRESS_BACKUP_PARTITION       (uint32_t*)0x0C100000
/* start address of application buffer
 * (@128kByte after bootloader(64k) and EEPROM(64k)*/
#define START_ADDRESS_APP_PARTITION          (uint32_t*)0x0C020000
/* first 128KByte is occupied by bootlader */
#define APP_PARTITION_MAX_SIZE               (1024 * 1024 - 128 * 1024)
/* backup space including meta-info space */
#define BACKUP_PARTITION_MAX_SIZE            (1024 * 1024)
/* offset to backup start address
 * of page to store file size inside backup */
#define METAINFO_OFFSET                      (1024 * 1024 - 256)

/**
 * \brief Alternative Boot Mode structure
 */
typedef struct ABM_Header {
  uint32_t MagicKey; /**< Magic key. Always 0xA5C3E10F */
  uint32_t StartAddress; /**< Start address of the programm to load */
  uint32_t Length; /**< Length of the programm to load. */
  uint32_t ApplicationCRC32; /**< CRC32 Sum of the complete application code */
  uint32_t HeaderCRC32; /**< CRC32 Sum of the four fields before */
} ABM_Header_t;

/*********************************************************************
 * LOCAL DATA
 ********************************************************************/
static const ABM_Header_t  __attribute__((section(".flash_abm")))
ABM0_Header = {
  .MagicKey = ABM_HEADER_MAGIC_KEY,
  .StartAddress = 0x08020000,  /* Start Flash Physical Sector 1 */
  .Length = 0xFFFFFFFF,
  .ApplicationCRC32 = 0xFFFFFFFF,
  .HeaderCRC32 =  0xEF423163
};

/*********************************************************************
 * API PROTOTYPES
 ********************************************************************/
```

linker_script.ld

```
  /* http://mcuoneclipse.com/2012/11/01/defining-variables-
  .abm ABSOLUTE(0x0800FFE0): AT(0x0800FFE0 | 0x04000000)
  {
     KEEP(*(.flash_abm))
  } > FLASH_1_cached

  /* DSRAM layout (Lowest to highest)*/
  Stack (NOLOAD) :
  {
```

Inside bootloader partition the ABM0 header must be located:

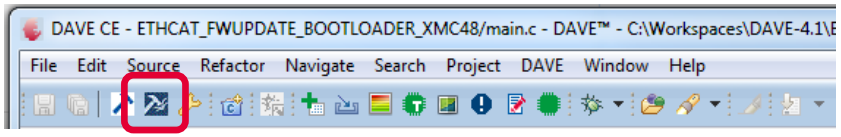
1. Define magic key of ABM header
2. Define ABM header structure
3. Define content of ABM header and location inside flash. Beside magic key and CRC, the start adress of the flash is the only mandatory information needed.
4. Define location of ABM header inside linker script to the last 32byte inside the first 64KB of flash: 0x0C00FFE0

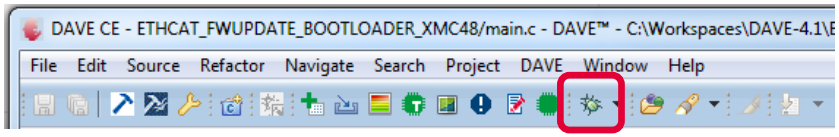
1. Overview and requirements
2. Setup
3. Short overview – boot modes
4. Architecture
5. Implementation of the application
6. Implementation of the bootloader
7. How to test – using TwinCAT3 as host

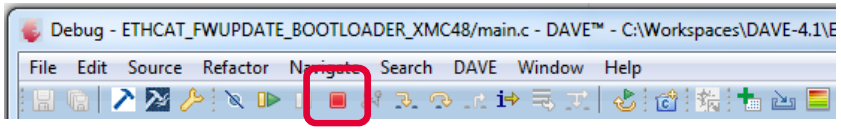
How to test – Downloading the executables and start application executable via bootloader

ACTIONS

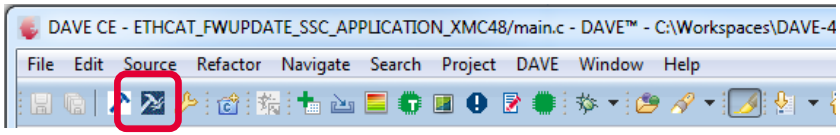
1. Set bootloader as active project, build and download the bootloader executable software to the XMC4800 and stop debugger

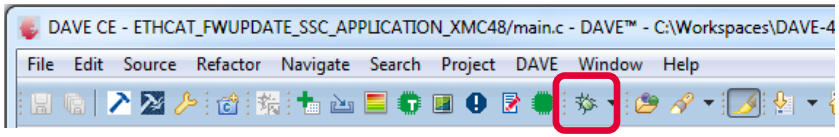


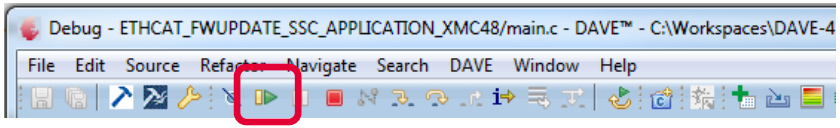




2. Set application as active project, build and download the application executable software to the XMC4800 and start the debugger







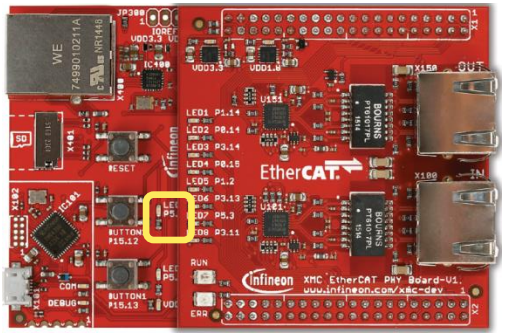
Note: Please be aware the application executable has reached the breakpoint inside main() already via bootloader executable

How to test – Application executable is running

OBSERVATIONS

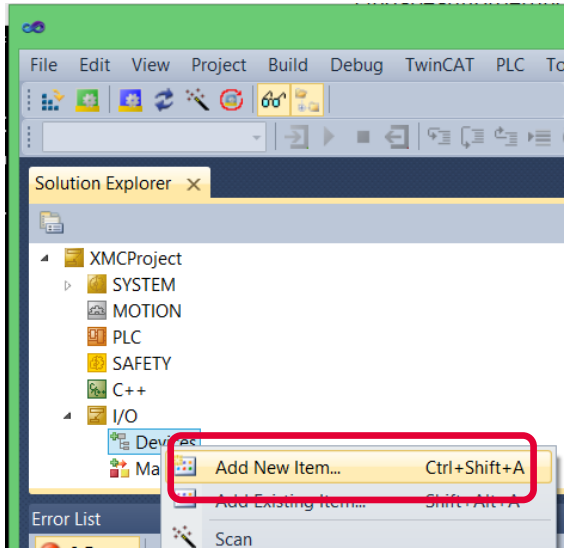
Application executable is running.

Check LED2 is flashing with a rate of 2Hz

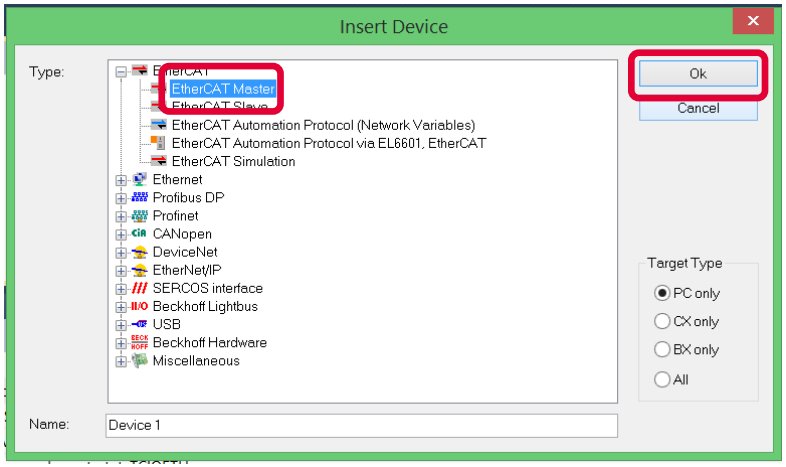


How to test – start the TwinCAT 3 master to run (1/4)

1



2



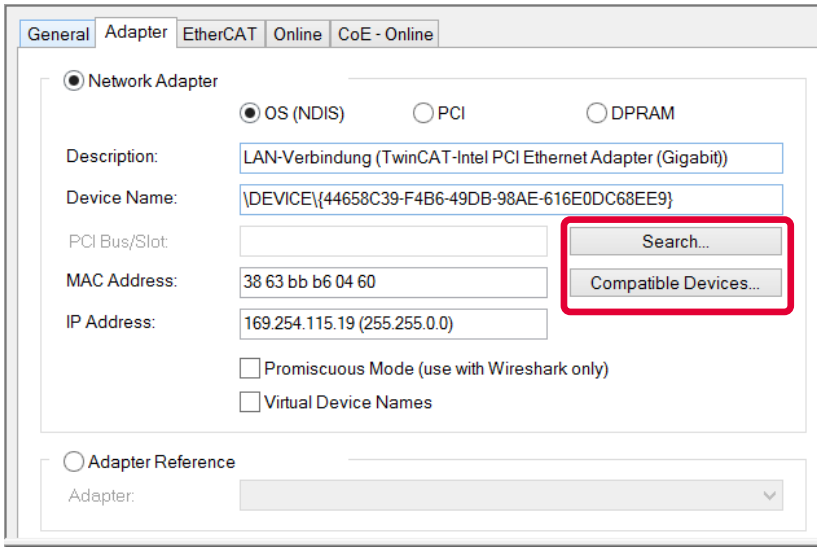
ACTIONS

After starting the TwinCAT System Manager from windows start menu:

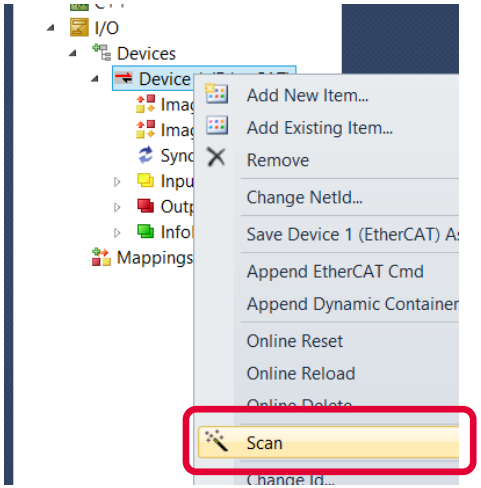
1. Right Click I/O-Devices and select „Add New Item…"
2. Create an EtherCAT master device by double click

How to test – start the TwinCAT 3 master to run (2/4)

3



4



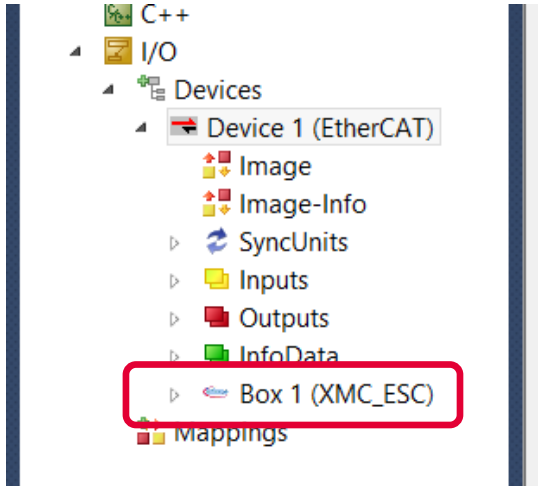
ACTIONS

3 Select the network adapter you want to use (search and select).
Application hint:
In case the device is not found please install the respective device driver by following the instructions given by TwinCAT through the „Compatible Devices…“ button.

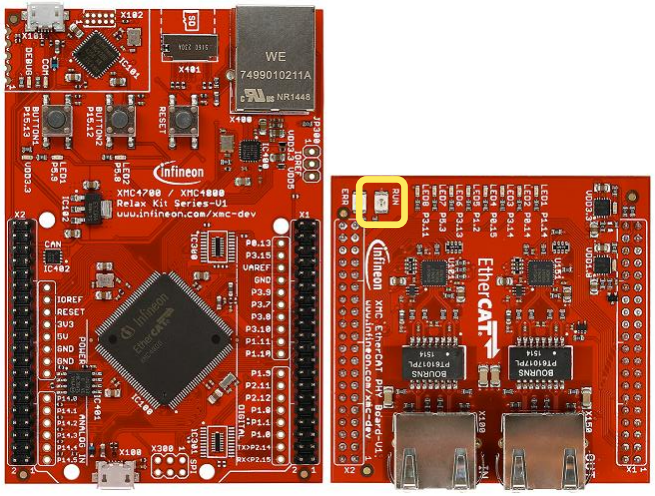
4 Right Click EtherCAT master and select „Scan Boxes…“

How to test – start the TwinCAT 3 master to run (3/4)

1



2



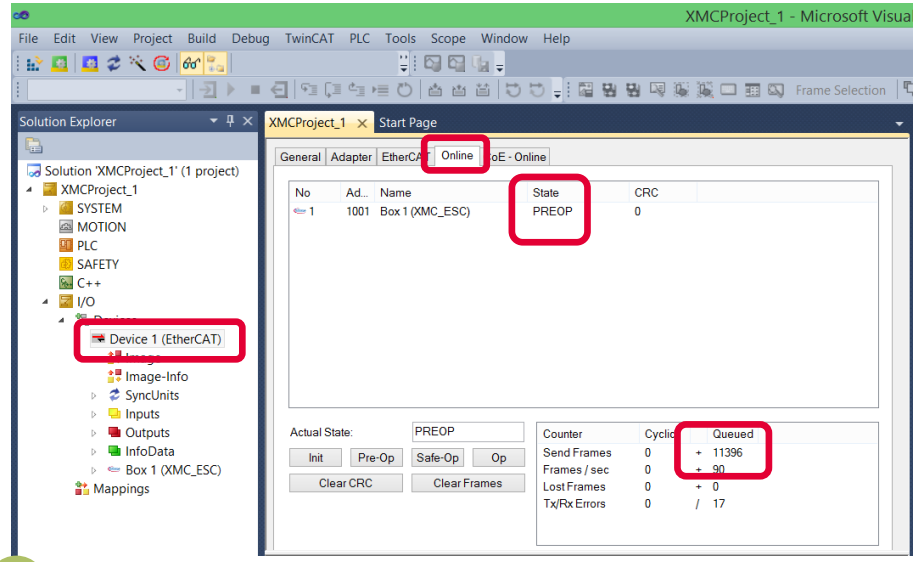
OBSERVATIONS

1 The slave appears as a node on the EtherCAT master bus

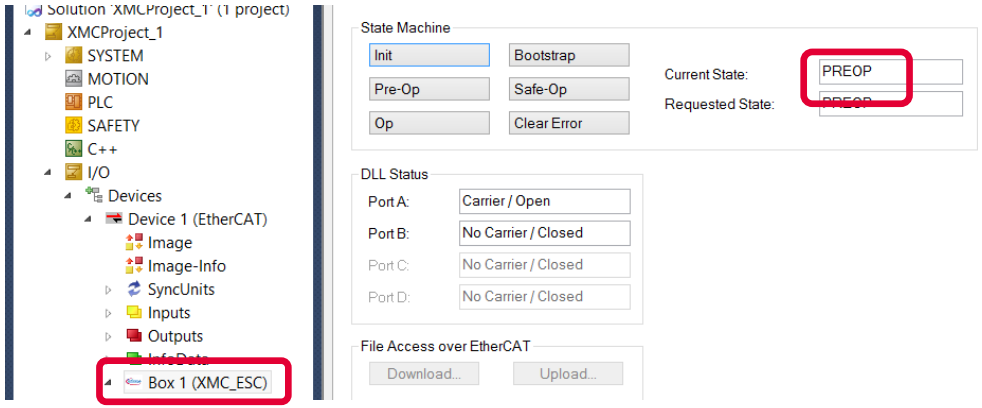
2 The RUN-LED is flashing indicating PREOP-state

How to test – start the TwinCAT 3 master to run (4/4)

3



4



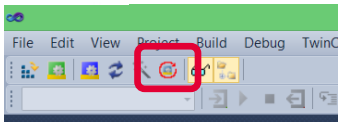
OBSERVATIONS

3 EtherCAT master view: Inside the EtherCAT master online state you see the queued frames counting up, the connected slave and its PREOP state.

4 EtherCAT slave view: The PREOP-state of the slave is indicated within the TwinCAT system manager .

How to test – Setting slave to operational mode

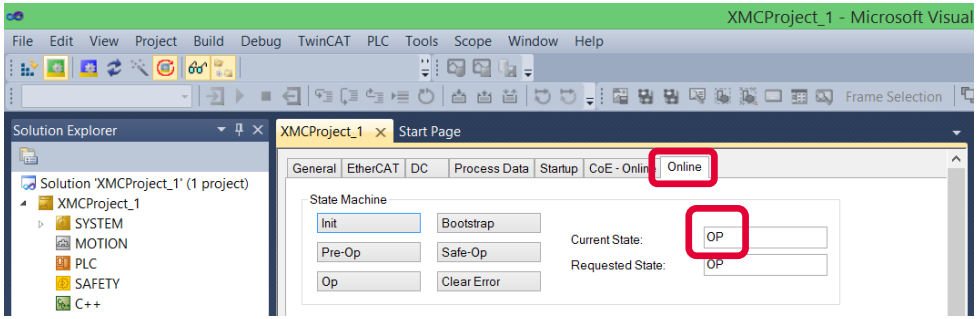
ACTION



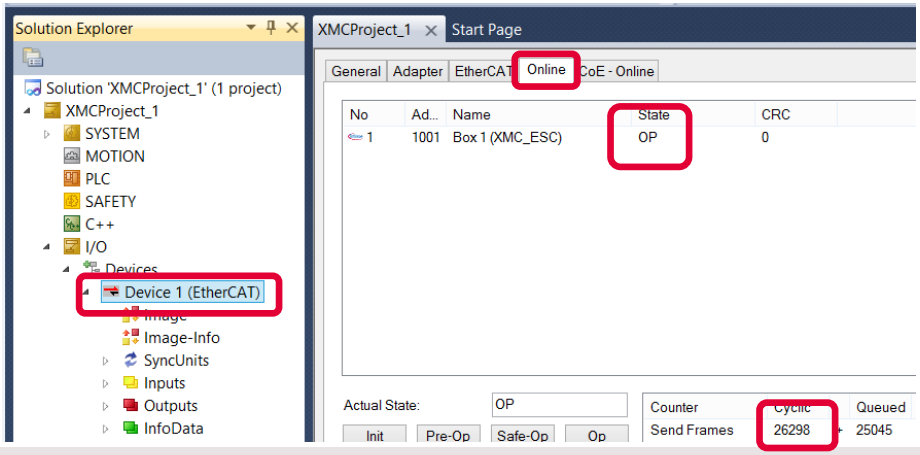
Set master device to free run mode

OBSERVATIONS

1



2

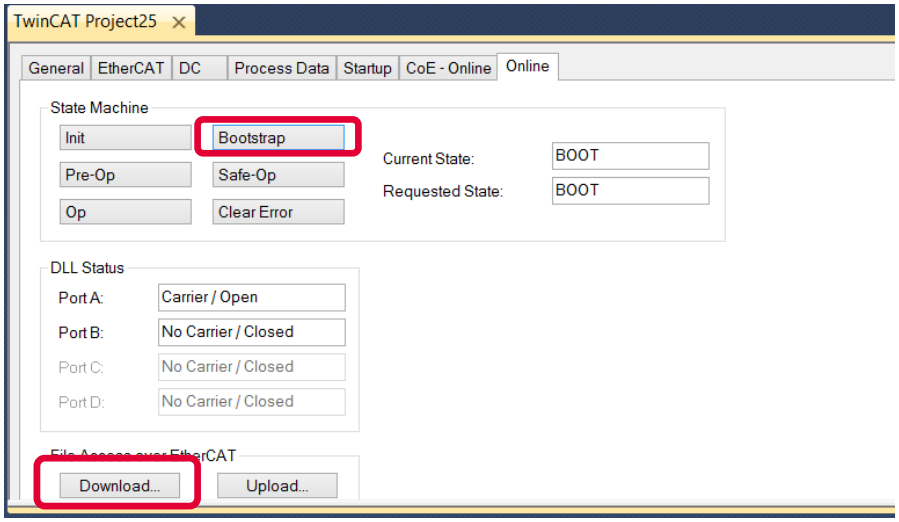


1 EtherCAT slave view: Online status of slave shows the slave in OP state

2 EtherCAT master view: Online status of master shows the slave in OP state. Frames are no more queued. Cyclic counter is incrementing.

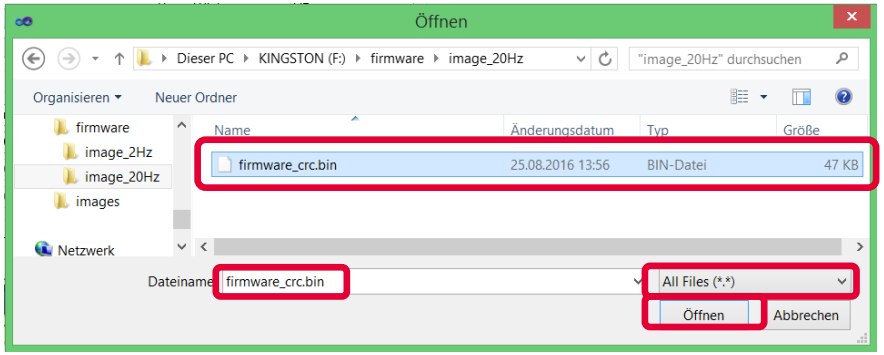
3 „XMC EtherCAT PHY Board“: RUN-LED is static turned on indicating OP-state.

How to test – Update firmware via FoE

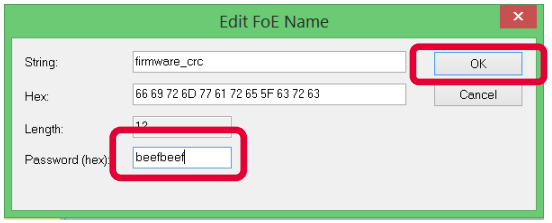


1

2



3



4

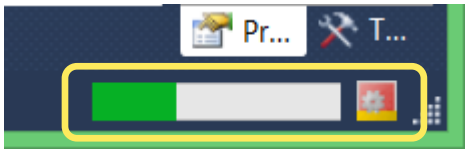
ACTIONS

1. Enter bootstrap state
2. Start download dialog
3. Select 20Hz firmware binary
4. Enter 32bit passkey „beefbeef“ and confirm with „OK“

How to test – Check the progress of firmware download via FoE

OBSERVATIONS

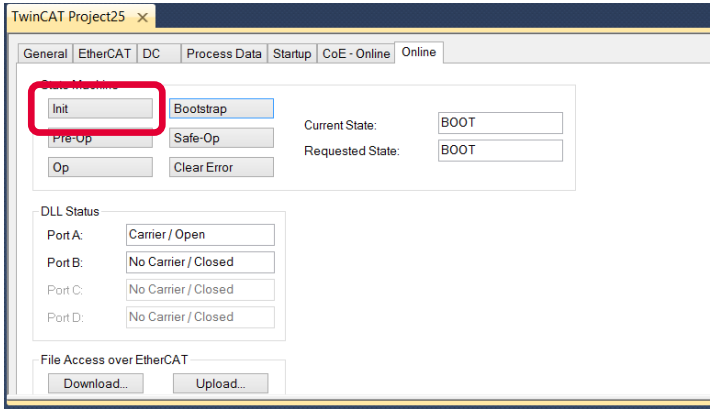
While the firmware is downloaded to the backup partition, a progress bar in the very right corner of the TwinCAT window indicates the upload status to you.



How to test – Update firmware via FoE

ACTIONS

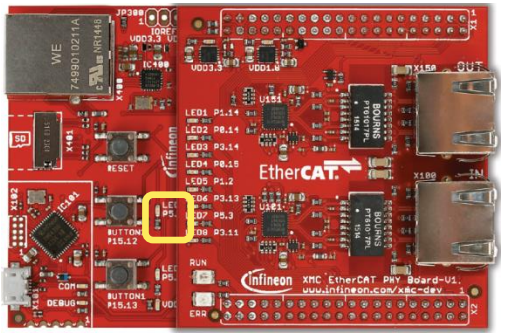
Set slave to INIT state. This triggers the restart of the device with bootloader executable. Firmware inside application partition will be updated.



OBSERVATIONS

The slave is restarted and after firmware was updated inside flash becomes operational again. Now LED2 is flashing with a rate of 20Hz.

→ Firmware update succeeded



Part of your life. Part of tomorrow.

Infineon