AN12948

Using LIN or CAN Bus to Upgrade Image on KW36/38

Rev. 0.9 — October 2020

Application Note

Contents

1 Introduction

Both LIN and CAN bus are international standard serial communication protocols and used widely in the automotive, industry, medical fields and so on. As a wireless microcontroller with Bluetooth LE 5.0 and Generic FSK support, KW36/38 family of devices have rich peripheral resources including 2 Low Power UART (LPUART) modules with LIN support and 1 FlexCAN module with CAN FD support.

One LIN or CAN bus always carries several nodes. For image upgrading, some nodes have the OTAP (Over-the-Air Programming) capability to get the new image from the OTAP server. But the other nodes without OTAP capability may only obtain the image through LIN or CAN serial communication method from the OTAP capable nodes.

This document describes how to use the LIN or CAN nodes, which have OTAP capability to upgrade the nodes, which have no OTAP capability, by LIN or CAN bus.

2 Driver enablement

2.1 Run the driver codes

With KW36 as an example, follow the steps below to run the driver code:

1. Download the latest SDK of KW36 from https://mcuxpresso.nxp.com/.
2. Prepare 2 FRDM-KW36 DK boards to act as LIN/CAN nodes. 12 V DC source and some electric wires are needed for the power supply and LIN/CAN communication.
3. Go to the path, *SDK\boards\frdmkw36\driver_examples\lin*, to find the driver examples of LIN master and LIN slave, and program the generated firmware to the 2 boards to demonstrate how to use LIN bus to transfer the data between master and slave node.
4. Go to the path, *SDK\boards\frdmkw36\driver_examples\flexcan\interrupt_transfer*, to find the driver example of FlexCAN non-blocking interrupt transfer to demonstrate how CAN nodes communicate.

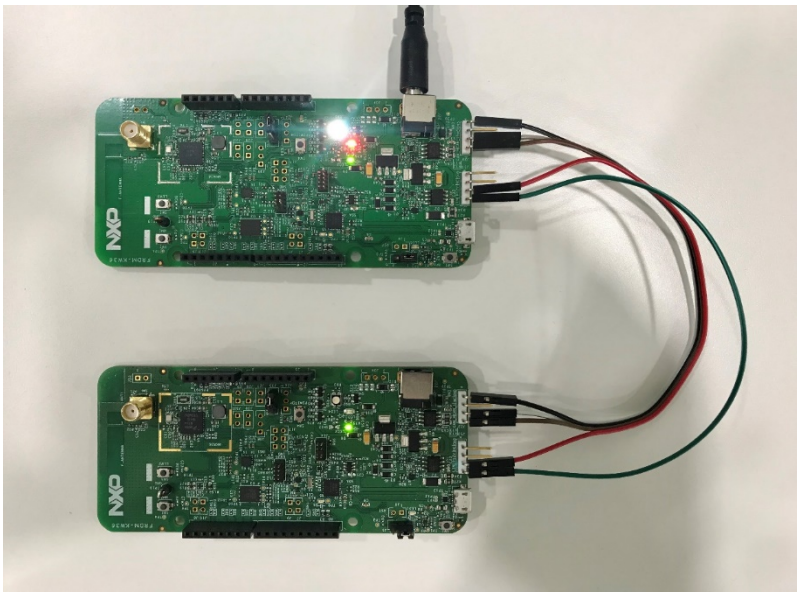


Figure 1. Using FRDM-KW36 DK boards to debug LIN/CAN communication

2.2 Port the driver codes to Bluetooth LE application

As mentioned above, some nodes having the OTAP capability get the image from the remote server and then transfer the image to the nodes without OTAP capability via LIN or CAN bus. NXP has one Bluetooth LE OTAP solution to perform image transmission from Bluetooth LE OTAP server to OTAP client via OTAP profile. For more details about Bluetooth LE OTAP solution, refer to chapter “OTAP” of the *BLE Application Developer’s Guide*.

NOTE

Bluetooth LE OTAP is just one solution to get the new image from remote server; the developer can select other available solutions of wireless or serial protocol.

On current NXP Bluetooth LE OTAP solution, when the Bluetooth LE OTAP client application finishes downloading the firmware from the Bluetooth LE OTAP server, it sets the value, indicating new image available, to BootFlags in flash and resets. The bootloader copies the new image data from selected storage area to internal flash when it identifies a new image. The client runs the new image at next reset once data copy finishes

One LIN network includes one master and one or multiple slaves. The KW36 LIN master should have the Bluetooth LE OTAP capability working as client and get the new image from the Bluetooth LE OTAP server for the LIN slave or itself. The LIN slaves might not have the Bluetooth LE OTAP feature to load the image from OTAP server directly. So, they can only be upgraded by LIN master through LIN bus.

One CAN network includes several nodes without the master-slave distinction. One KW36 CAN node has the Bluetooth LE OTAP capability to get the image from the OTAP server. Suppose this is Node A. And the others can only be upgraded by Node A through CAN bus. These nodes are called Node B (1, 2...N). The figure below shows the data flow of the whole image upgrading system.

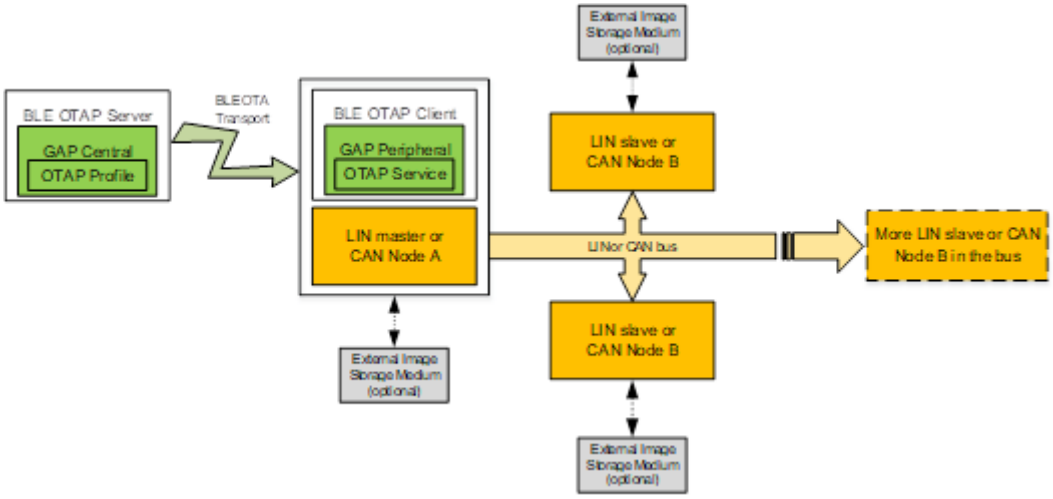


Figure 2. Data flow of image upgrading system

To enable the Bluetooth LE OTAP feature, copy the LIN master driver example code to the location, *SDK\boards\frdmkw36\wireless_examples\bluetooth\otac_att*, and name it as *lin_master* project. LIN slave does not need Bluetooth LE OTAP capability, but it needs the EEPROM operating and image switching features. For convenience, copy the LIN slave driver example code to the *SDK\boards\frdmkw36\wireless_examples\bluetooth\otac_att* project and name it as *lin_slave* project but only use the necessary features.

Similarly, you can port the CAN driver codes to the *SDK\boards\frdmkw36\wireless_examples\bluetooth\otac_att* project to enable the Bluetooth LE OTAP capability or the necessary EEPROM operating and image switching features. Name the projects as *can_a* and *can_b*.

NOTE

If you want to use other wireless examples available in the *SDK\boards\frdmkw36\wireless_examples\bluetooth* directory, for example, *w_uart*, you can add the Bluetooth LE OTAP profile and functions in this example. And then copy the LIN/CAN driver example code to this selected project. The released examples are based on the *w_uart* project.

To integrate the CAN and LIN driver examples into Bluetooth LE projects, you can follow the *Chapter 5 Adding FlexCAN and LIN demo examples into a Bluetooth LE project* of *AN12273 Using MCUXpresso SDK CAN and LIN Drivers to Create a Bluetooth LE-CAN and Bluetooth LE-LIN Bridges on KW36/KW35*.

NOTE

The example projects released with this application note are built in IAR Embedded Workbench for Arm 8.50.4 and MCUXpresso IDE v11.0.1_2563. Same or higher IDE version should be used in this development.

3 Image obtaining

When the LIN master/CAN Node A enables the Bluetooth LE OTAP capability, it can get the image from the Bluetooth LE OTAP server through Bluetooth LE OTAP profile. The OTA file needs to add OTA header and some tail information to include its OTA File Identifier, Image Version, Image Identifier, Image Size and so on based on the application binary file. The Image Identifier field can be used to indicate if the OTA file is for LIN slave/CAN Node B or LIN master/CAN Node A itself.

By default, the Image Identifier is 0x0001 for the LIN master/CAN Node A itself. You can define another value for LIN slave/CAN Node B in *otap_interface.h*, for example, 0x000A:

#define gBleOtaImageIdForLinCanNode_c (0x000AU)

To enable the creation of a binary file for your application in IAR IDE, open project **Options > Output Converter**, activate the **Generate additional output** checkbox and choose the **Raw binary** option from the **Output format** drop-down menu. You can also override the name of the output file. A screenshot of the described configuration is shown in figure below.

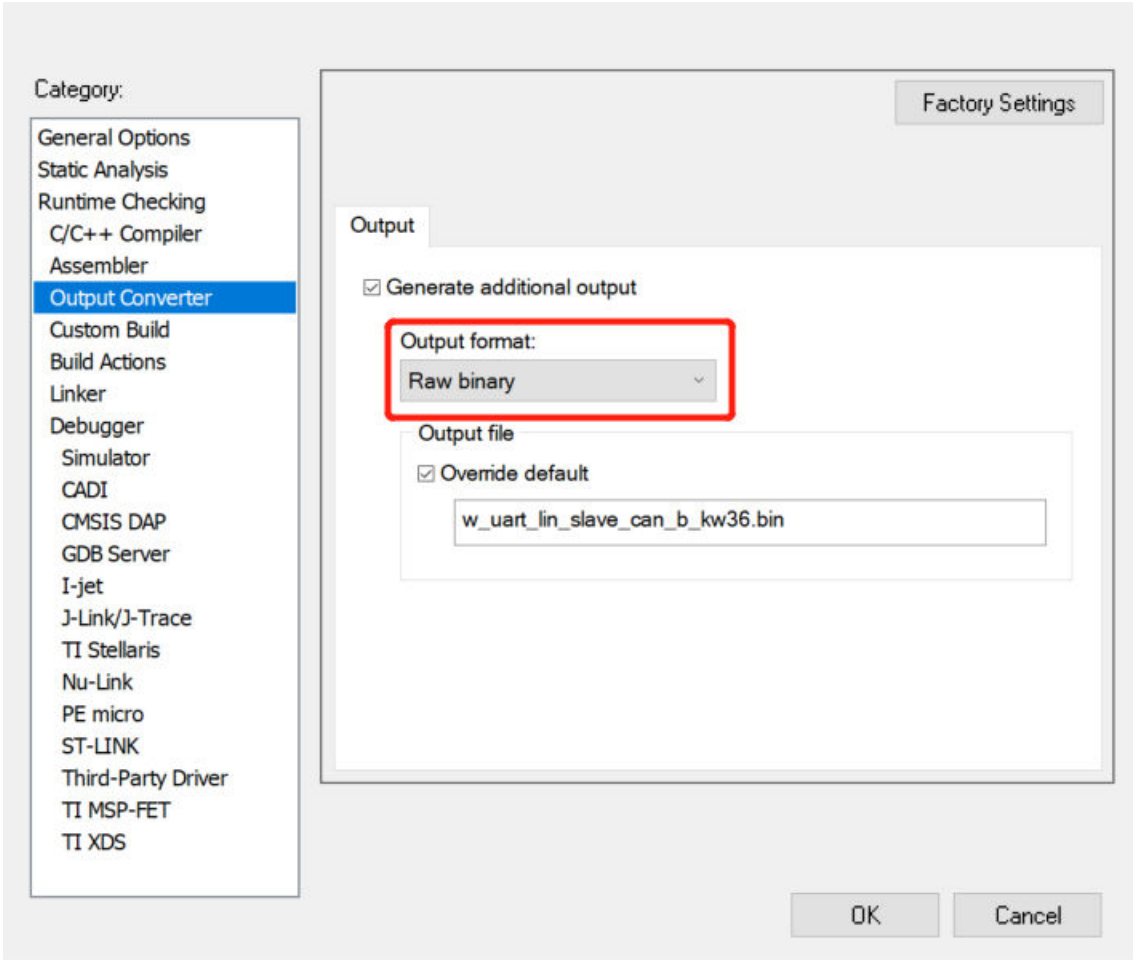


Figure 3. Enabling binary output in IAR IDE

To enable the creation of a binary file for your application in MCUXpresso IDE, open project **Properties > C/C++ Build > Settings > Build steps**, press the Edit button for the Post-build steps then the Post-build steps window will show up. Add the following command or uncommented it (by removing '#' character at the beginning) if it is already there.

arm-none-eabi-objcopy -v -O binary "${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin"

A screenshot of this window is shown in figure below.

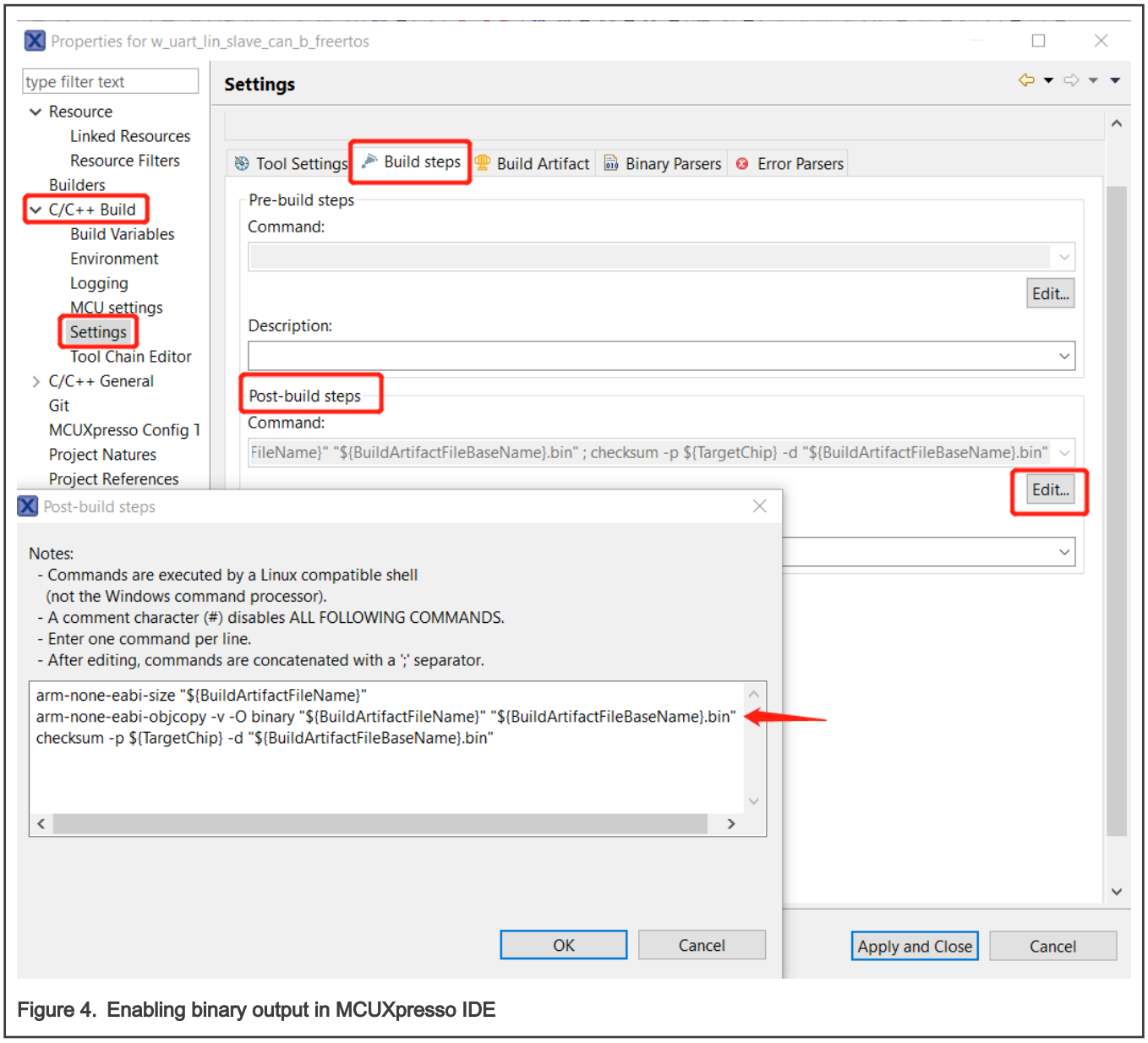


Figure 4. Enabling binary output in MCUXpresso IDE

The NXP Connectivity Test Tool can generate the Bluetooth LE OTA file for the binary built in MCUXpresso or IAR by following the below steps:

1. Click **OTA Updates>OTAP Bluetooth LE** in the top menu.
2. Set the custom value, 0x000A, in the **Image Id** of OTA header.
3. Click **Browse** to load the binary that need to be upgraded.
4. Select the chip model, KW36/KW38 and click **OK**.
5. Click **Save** to get the generated OTA file.

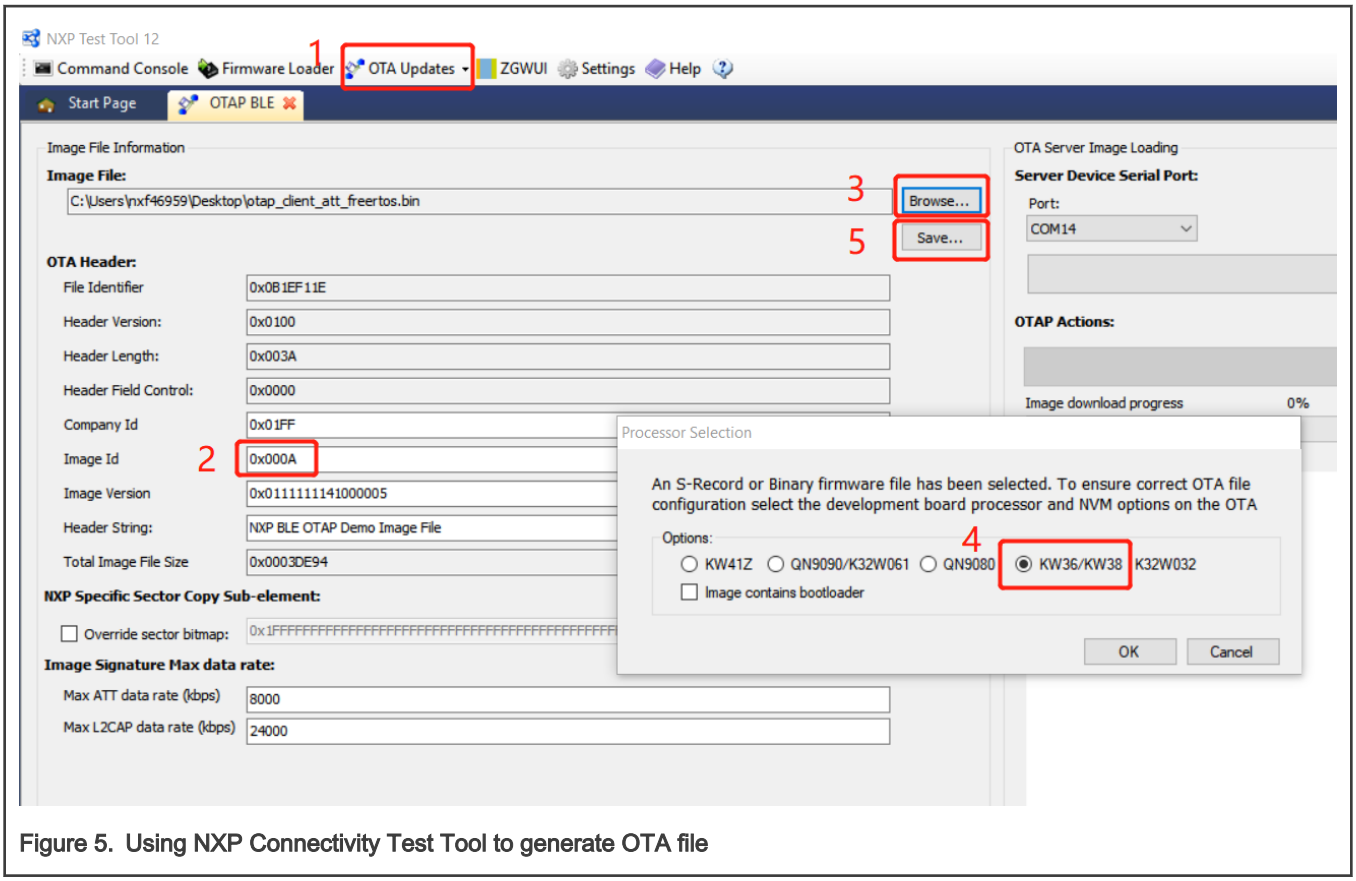


Figure 5. Using NXP Connectivity Test Tool to generate OTA file

If the LIN master/CAN Node A detects that the OTA file is for itself, it sets the value of available new image to TRUE to BootFlags in the specific position of internal flash in *OTA_SetNewImageFlag()* when data downloading completes. After reset, the bootloader finds a new available image and copies the data from selected storage area to internal flash. The LIN master/CAN Node A runs new image when bootloader finishes copying and resets again.

If the LIN master/CAN Node A detects that the OTA file is for LIN slave/CAN Node B, it sets another available new image flag in *OtaSupport.h*, for example, 0xAA, to BootFlags. This is required to indicate there is a new image for LIN slave/CAN Node B and to prevent the bootloader to copy the LIN slave/CAN Node B image to itself if the device resets unexpectedly.

#define gBootValueForLinCanNode_c (0xAA)

You need to define a bool variable, *g_ota_for_lin_or_can_node* in *OtaSupport.h*, for *OtapClient_IsImageFileHeaderValid()* to identify if the new image from Bluetooth LE OTAP server is for LIN slave/CAN Node B or not. If yes, set g_ota_for_lin_or_can_node to TRUE.

```
bool_t g_ota_for_lin_or_can_node = FALSE;

static otapStatus_t OtapClient_IsImageFileHeaderValid (bleOtaImageFileHeader_t* imgFileHeader)
{
    otapStatus_t headerStatus;
    …
    if (gOtapStatusSuccess_c == headerStatus)
    {
        if (gBleOtaImageIdForLinCanNode_c == ((otapClientData.imgId[1] << 8)
            + otapClientData.imgId[0]))
        {
            g_ota_for_lin_or_can_node = TRUE;
        }
        else
        {
            g_ota_for_lin_or_can_node = FALSE;
        }
    }
    return headerStatus;
}
```

Figure 6. Code for identifying node for the image

```
void OTA_SetNewImageFlag(void)
{
…
    union{
        uint32_t value;
        uint8_t aValue[FSL_FEATURE_FLASH_PFLASH_BLOCK_WRITE_UNIT_SIZE];
    }bootFlag;
    uint32_t status;

    if( mNewImageReady )
    {
        NV_Init();

        if (gOtaForLinOrCanNode)
        {
            bootFlag.value = gBootValueForLinCanNode_c;
        }
        else
        {
            bootFlag.value = gBootValueForTRUE_c;
        }

        status = NV_FlashProgramUnaligned((uint32_t)&gBootFlags.newBootImageAvailable,
                                          sizeof(bootFlag),
                                          bootFlag.aValue);
    }
…
}
```

Figure 7. Code for setting new image flag

Generally, LIN upgrading and CAN upgrading are mutually exclusive. Define one macro *gOtaUseBusSelection_d* in *app_preinclude.h* to select LIN or CAN for the image upgrading; *gOtaUseBus_LIN_c* is used for LIN option while *gOtaUseBus_CAN_c* is used for CAN option.

```
#define gOtaUseBus_LIN_c        1
#define gOtaUseBus_CAN_c        2
#define gOtaUseBusSelection_d   gOtaUseBus_CAN_c
#ifndef gOtaUseBusSelection_d
#define gOtaUseBusSelection_d   gOtaUseBus_LIN_c
#endif
```

Figure 8. Code for selecting the bus to do image upgrading

By default, the application calls *ResetMCU()* to reset the device after setting the new image flags to make the bootloader copy image data. But instead of reset, it should start the LIN/CAN transfer if the image is for LIN slave/CAN Node B. For this, add the following code after *OTA_SetNewImageFlag()* and before *ResetMCU()* every time these two functions are called one after another in *otap_client.c*.

```
OTA_SetNewImageFlag();
if (gOtaForLinOrCanNode)
{
    gOtaForLinOrCanNode = FALSE;
    /* start to LIN or CAN upgrade transfer */
    LinCanOtaStartCallback();
}
else
{
    ResetMCU();
}
```

Figure 9. Code for starting LIN/CAN upgrading process

Include an additional condition in the *OtapBootloader.c* of *SDK\boards\frdmkw36\wireless_examples\framework\bootloader_otap* project to prevent incorrect loading and switching; follow the code below.

```
void Boot_CheckOtapFlags(void)
{
    gpBootInfo = (bootFlags_t*)gBootImageFlagsAddress_c;

    if (( !FLib_MemCmpToVal((const void*)gpBootInfo->u0.aNewBootImageAvailable, 0xFF,
        sizeof(gpBootInfo->u0.aNewBootImageAvailable))
            && (gpBootInfo->u0.aNewBootImageAvailable[0] != 0xAA)) ||
        FLib_MemCmpToVal((const void*)gpBootInfo->u1.aBootProcessCompleted, 0xFF,
        sizeof(gpBootInfo->u1.aBootProcessCompleted)))
    {
        /* Write the new image */
        Boot_LoadImage();
    }
}
```

Figure 10. Code for identifying if bootloader needs to load the new image

4 Image storage

The image received from Bluetooth LE OTAP server can be saved in the internal flash or external EEPROM in the LIN master/CAN Node A site. Similarly, the image received from LIN master/CAN Node A can be saved in the internal flash or external EEPROM in LIN slave/CAN Node B. The storage area selection of LIN master/CAN Node A and LIN slave/CAN Node B is independent.

4.1 Set the image storage in IAR IDE

Set the configurations as below to select internal flash for upgrading:

- In the *app_preinclude.h* of the application project:

  #define gEepromType_d gEepromDevice_InternalFlash_c

- In the **Options > Linker > Config** of the application project:

  gUseInternalStorageLink_d=1

  gEraseNVMLink_d=0

- In the **Options > C/C++ Compiler > Preprocessor** of the bootloader project:

  gEepromType_d=gEepromDevice_InternalFlash_c

The image is saved in defined OTAP internal storage space as shown in figure below.

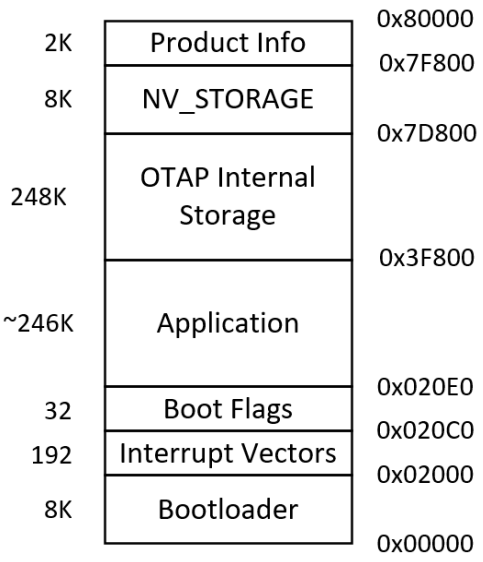


Figure 11. Memory layout of internal flash

To select external EEPROM, for example, AT45DB041E, as designed in the DK board, set the configurations as below:

- In the *app_preinclude.h* of the application project:

  #define gEepromType_d gEepromDevice_AT45DB041E_c

- In the **Options > Linker > Config** of the application project:

  gUseInternalStorageLink_d=0

  gEraseNVMLink_d=0

- In the **Options > C/C++ Compiler > Preprocessor** of the bootloader project:

  gEepromType_d=gEepromDevice_AT45DB041E_c

The image is saved in the AT45DB041E from 0 offset via SPI communication.

4.2 Set the image storage in MCUXpresso IDE

Set the configurations as below to select internal flash for upgrading:

- In the *app_preinclude.h* of the application project:

  #define gEepromType_d gEepromDevice_InternalFlash_c

- In the **Properties > C/C++ Build > Settings > Tool Settings > MCU Linker > Miscellaneous** of the application project:

  --defsym=gUseInternalStorageLink_d=1

- In the **Properties > C/C++ Build > MCU Settings** of the application project, select the **PROGRAM_FLASH** line and click **Split** to split this space to two parts, one for current application and another one for the OTAP storage. Follow Figure 12 to set the **Name**, **Location** and **Size** for the internal OTAP storage.

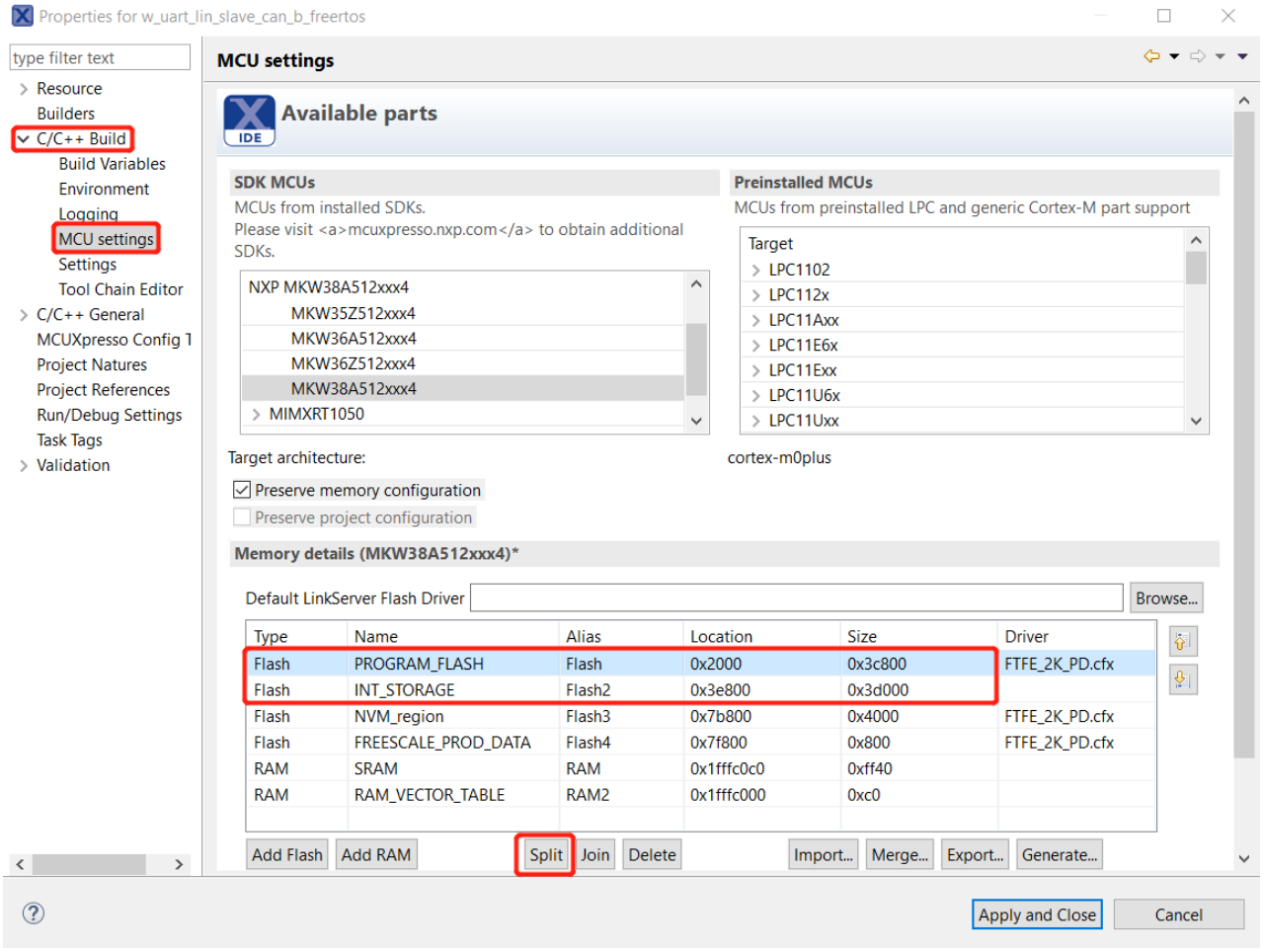


Figure 12. Flash settings for internal flash upgrading in MCUXpresso IDE

- Open *project/linkscripts/end_text.ldt*, remove the *FILL(0xFFFFFFFF)* and *BYTE(0xFF)* lines shown in Figure 13.

```
_etext = .;

.NVM_region :
{
    FILL(0xFFFFFFFF);
    . = ORIGIN(NVM_region) + LENGTH(NVM_region) - 1;
    BYTE(0xFF)
} > NVM_region
```

Figure 13. Remove the codes in end_text.ldt file

- In the **Properties > C/C++ Build > Settings > MCU C Compiler > Preprocessor** of bootloader project:

  gEepromType_d=gEepromDevice_InternalFlash_c

To select external EEPROM, for example, AT45DB041E, as designed in the DK board, set the configurations as below:

- In the *app_preinclude.h* of the application project:

  #define gEepromType_d gEepromDevice_AT45DB041E_c

- In the **Properties > C/C++ Build > Settings > Tool Settings > MCU Linker > Miscellaneous** of the application project:

--defsym=gUseInternalStorageLink_d=0

- In the **Properties > C/C++ Build > MCU Settings** of the application project, follow Figure 14 to set the **Location** and **Size** of **PROGRAM_FLASH** for application storage.

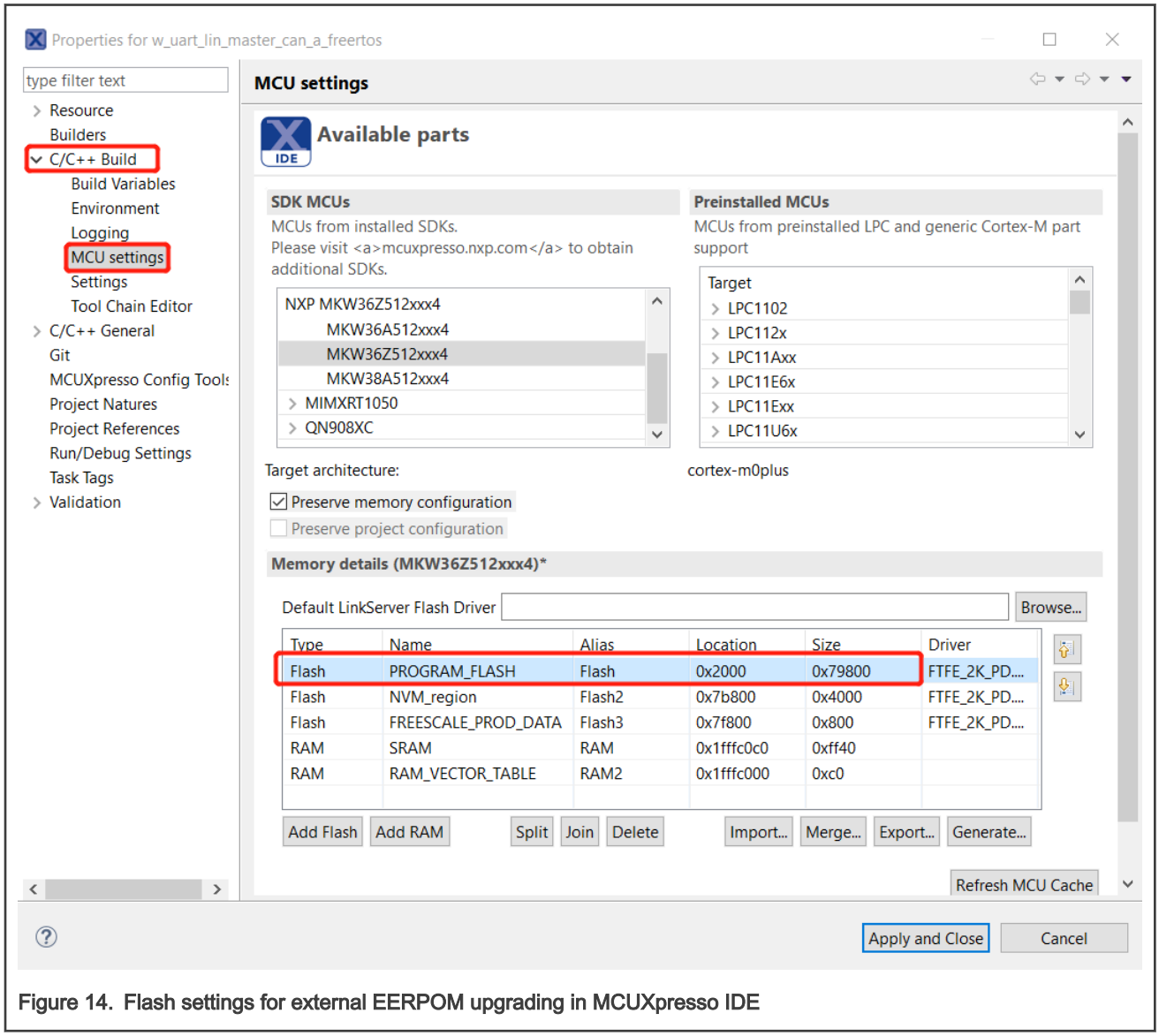


Figure 14. Flash settings for external EERPOM upgrading in MCUXpresso IDE

- Open *project/linkscripts/end_text.ldt*, remove the *FILL(0xFFFFFFFF)* and *BYTE(0xFF)* lines shown in Figure 13
- In the **Properties > C/C++ Build > Settings > MCU C Complier > Preprocessor** of the bootloader project:

  gEepromType_d=gEepromDevice_AT45DB041E_c

4.3 Image size optimization

Figure 11 indicates that if the image needs to be upgraded is larger than the size of OTAP internal storage space, the internal flash upgrading method will not be available. But you can try to select the appropriate optimization level in IDE to decrease the compiling image size then the internal flash upgrading may become available again. On the other hand, smaller image size can take less upgrading time.

For IAR IDE, open the **Options > C/C++ Compiler > Optimizations**, select the **Level** of **High**, **Medium** or **Low** then you can get the smaller image for upgrading.

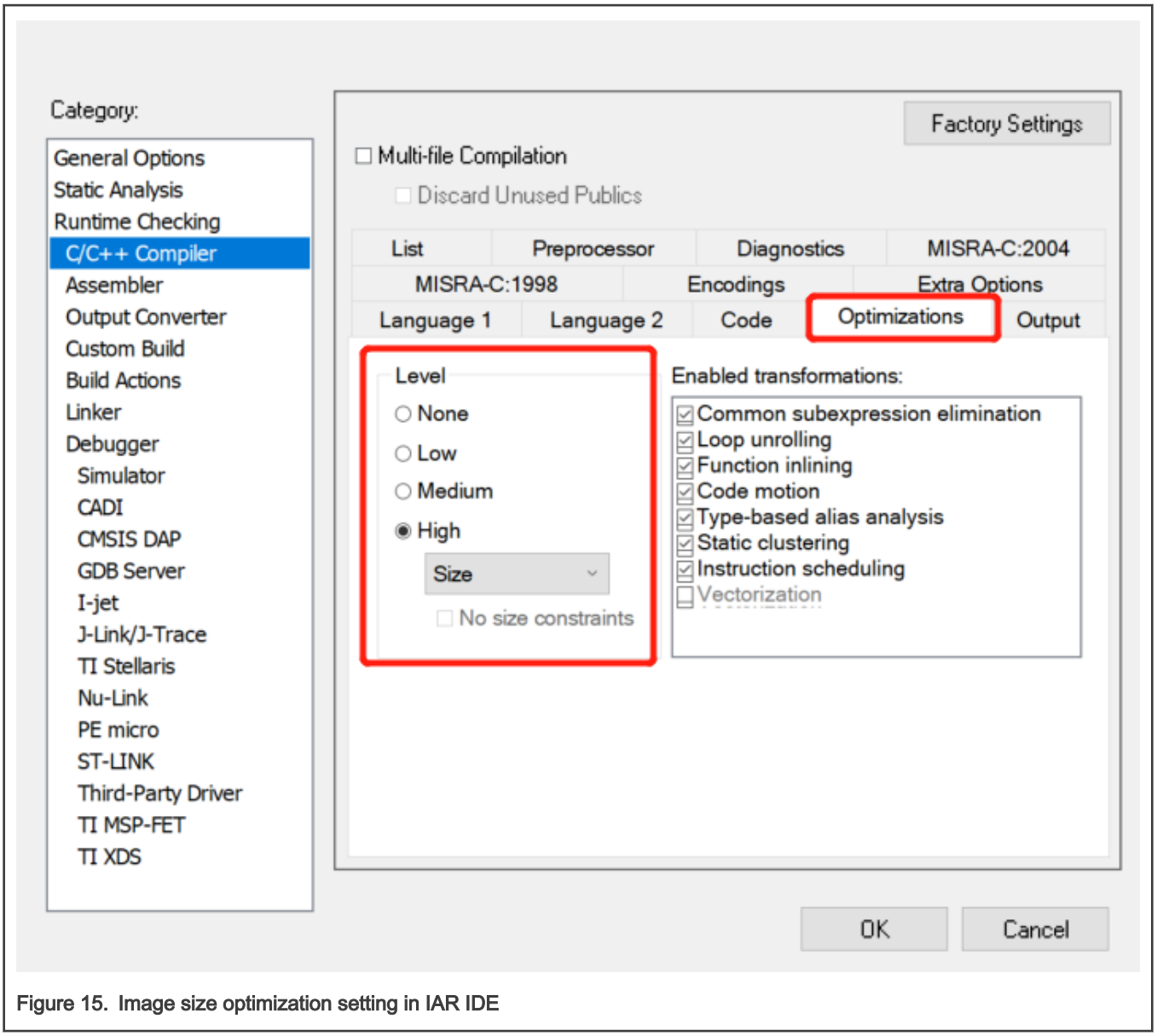


Figure 15. Image size optimization setting in IAR IDE

For MCUXpresso IDE, open the **Properties > C/C++ Build > Settings > Tool Settings > MCU C Compiler > Optimization**, select the **Optimization Level** of **Optimize for size (-Os)**, **Optimize (-O1)** or other level then you may get the smaller image for upgrading.

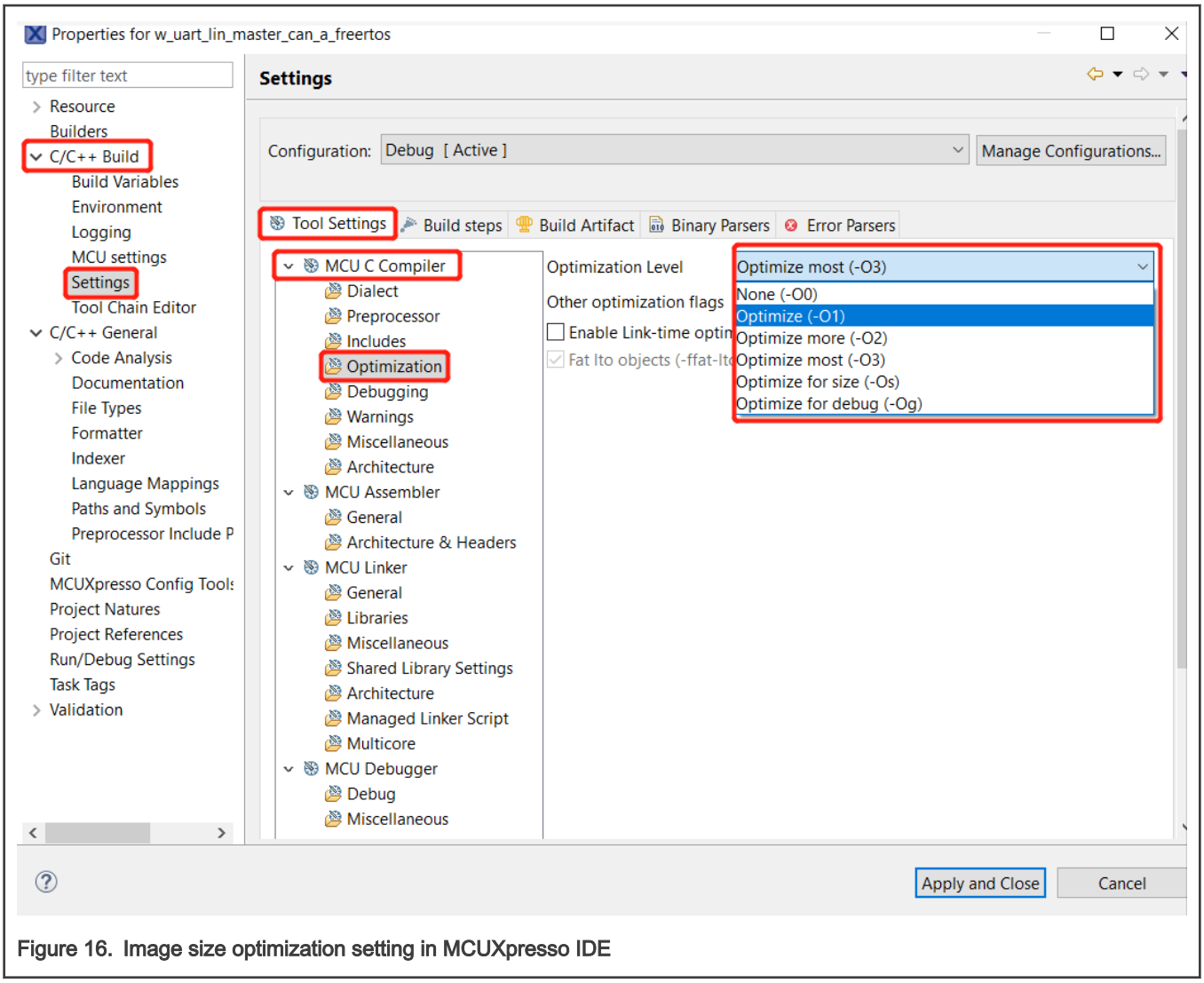


Figure 16. Image size optimization setting in MCUXpresso IDE

5 Image transfer

When the image loading from the OTAP server finishes, the LIN master/CAN Node A calls *LinOtaStartCallback()*/ *CanOtaStartCallback()* to start the LIN/CAN transfer.

5.1 Transfer via LIN bus

To define 3 identifiers of LIN unconditional frame type in the LIN task schedule table, write the following code in *lin_cfg.h*:

```
typedef enum
{
    gID_OtapCmd_c = 0x31,
    gID_OtapGetStatus_c,
    gID_OtapData_c
} lin_id_t;

#define LIN_FRAME_BUF_SIZE       (11U)
#define LIN_NUM_OF_FRMS           (3U)
```

Figure 17. Code for defining LIN frame identifiers and sizes

Also, include the following code in *lin_cfg.c*:

```
uint8_t g_lin_frame_data_buffer[LIN_FRAME_BUF_SIZE] = {0};

static const lin_frame_struct lin_frame_tbl[LIN_NUM_OF_FRMS] = {
    {  LIN_FRM_UNCD, 1U, LIN_RES_PUB, 0U, 0U, 1U, 10U, 0U }
    ,{ LIN_FRM_UNCD, 2U, LIN_RES_SUB, 1U, 1U, 1U, 4U, 0U }
    ,{ LIN_FRM_UNCD, 8U, LIN_RES_PUB, 3U, 2U, 1U, 2U, 0U }};

static uint8_t LI0_lin_configuration_RAM[LI0_LIN_SIZE_OF_CFG]= {
    0x00, gID_OtapCmd_c, gID_OtapGetStatus_c, gID_OtapData_c};
const uint16_t  LI0_lin_configuration_ROM[LI0_LIN_SIZE_OF_CFG]= {
    0x0000, gID_OtapCmd_c, gID_OtapGetStatus_c, gID_OtapData_c};
```

Figure 18. Code for defining LIN frame schedule table

- *gID_OtapCmd_c*: notifies LIN slave to start or end the image data transfer, carrying the *lin_ota_cmd_c* variable defined in *lin_cfg.h* in Byte0 as the payload.

```
typedef enum
{
    LIN_OTA_CMD_NONE = 0x00,
    LIN_OTA_CMD_START,
    LIN_OTA_CMD_END,
    LIN_OTA_CMD_CONTINUE
} lin_ota_cmd_c;
```

Figure 19. Code for defining lin_ota_cmd_c enumeration

- *gID_OtapGetStatus_c*: gets the status of LIN slave, carrying the *lin_ota_status_t* variable defined in *lin_cfg.h* in Byte0 and sequence number (0~255) in Byte1 as the payload.

```
typedef enum
{
    LIN_OTA_STATUS_IDLE = 0x00,
    LIN_OTA_STATUS_READY,
    LIN_OTA_STATUS_RUNNING,
    LIN_OTA_STATUS_FINISH,
    LIN_OTA_STATUS_ABORT
} lin_ota_status_t;
```

Figure 20. Code for defining lin_ota_status_c enumeration

- *gID_OtapData_c*: sends the data by frame to LIN slave, carrying maximum 8-byte image data as the payload.

To consider the payload size and data rate limitation of LIN bus, you may not get the status of LIN slave by each data frame since it will cause lower upgrading speed. LIN master reads one block data, for example, 1 KB, from the selected storage area and saves to RAM buffer. It then sends them via *gID_OtapData_c* frames to the LIN slave continuously. Define the sizes in *lin_cfg.h* and related reading variables in *lin_cfg.c* of the *lin_master* project:

```
#define LIN_OTA_BLOCK_SIZE        (1024U)
#define LIN_OTA_FRAME_SIZE        (8U)
#define LIN_OTA_FRAMES_OF_BLOCK   (LIN_OTA_BLOCK_SIZE / LIN_OTA_FRAME_SIZE)

static uint8_t eeprom_read_buffer[LIN_OTA_BLOCK_SIZE] = {0};
uint32_t g_lin_read_flash_offset = gBootData_ImageLength_Offset_c;
```

Figure 21. Code for defining LIN transmission sizes and reading variables

LIN slave saves the received data to RAM buffer. When one block transfer finishes, the LIN slave writes the received block to the specific offset of image storage area. It then responds the status and next block sequence number via *gID_OtapGetStatus_c* to the LIN master. The LIN master continues the reading and transfer of next block. Define the sizes in *lin_cfg.h* and related writing variables in *lin_cfg.c* of the *lin_slave* project:

```
#define LIN_OTA_BLOCK_SIZE        (1024U)
#define LIN_OTA_FRAME_SIZE        (8U)
#define LIN_OTA_FRAMES_OF_BLOCK   (LIN_OTA_BLOCK_SIZE / LIN_OTA_FRAME_SIZE)

static uint8_t eeprom_write_buffer[LIN_OTA_BLOCK_SIZE] = {0};
uint32_t g_lin_write_flash_offset = gBootData_ImageLength_Offset_c;
```

Figure 22. Code for defining LIN reception sizes and writing variables

To construct the state machine of upgrading, define the *lin_ota_stage_t* enumeration in *lin_cfg.h* and variables in *lin_cfg.c*:

```
typedef enum
{
    LIN_OTA_STAGE_IDLE = 0x00,
    LIN_OTA_STAGE_TX_DATA,
    LIN_OTA_STAGE_END
} lin_ota_stage_t;

lin_ota_status_t g_lin_ota_status = LIN_OTA_STATUS_IDLE;
lin_ota_stage_t g_lin_ota_stage = LIN_OTA_STAGE_IDLE;
```

Figure 23. Code for defining lin_ota_stage_t enumeration and variables

It is possible to support multiple slave nodes upgrading at the same time, if the nodes have the same function and no bad communication in the bus. The image upgrading flow diagrams of LIN master and LIN slave are shown in Figure 24 and Figure 25.

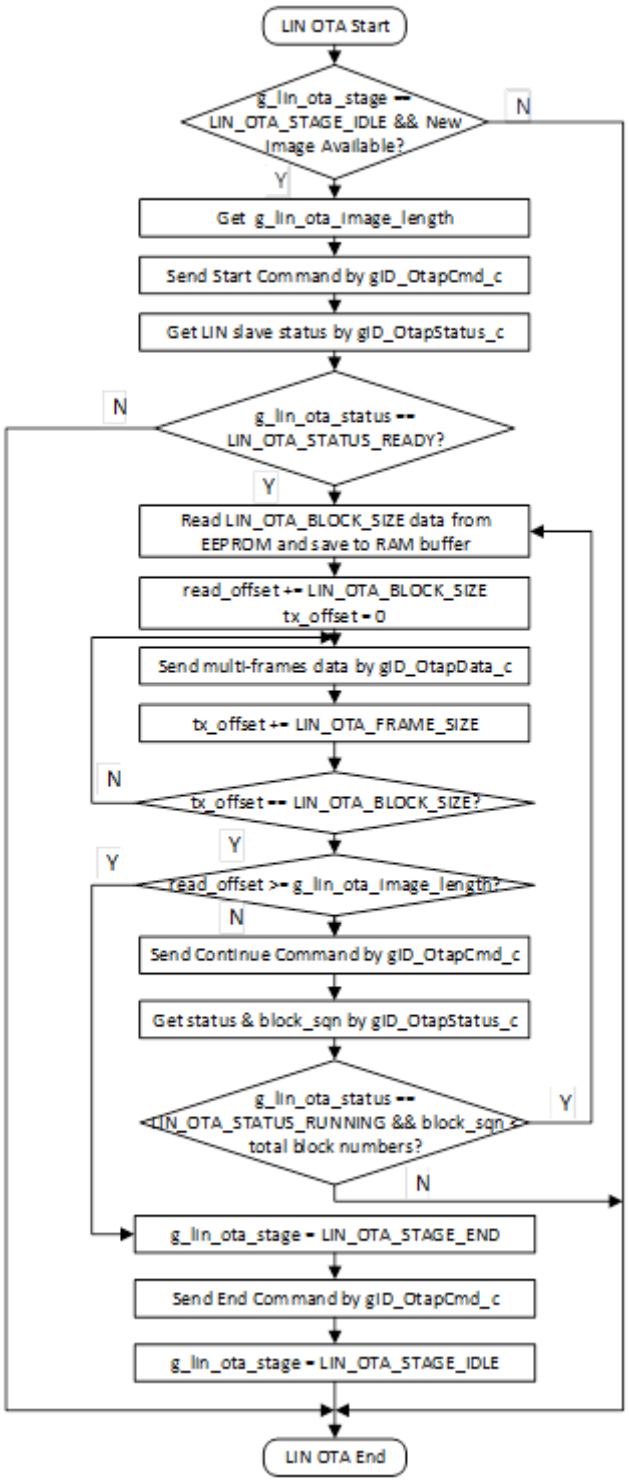


Figure 24. Image upgrading flow diagram of LIN master

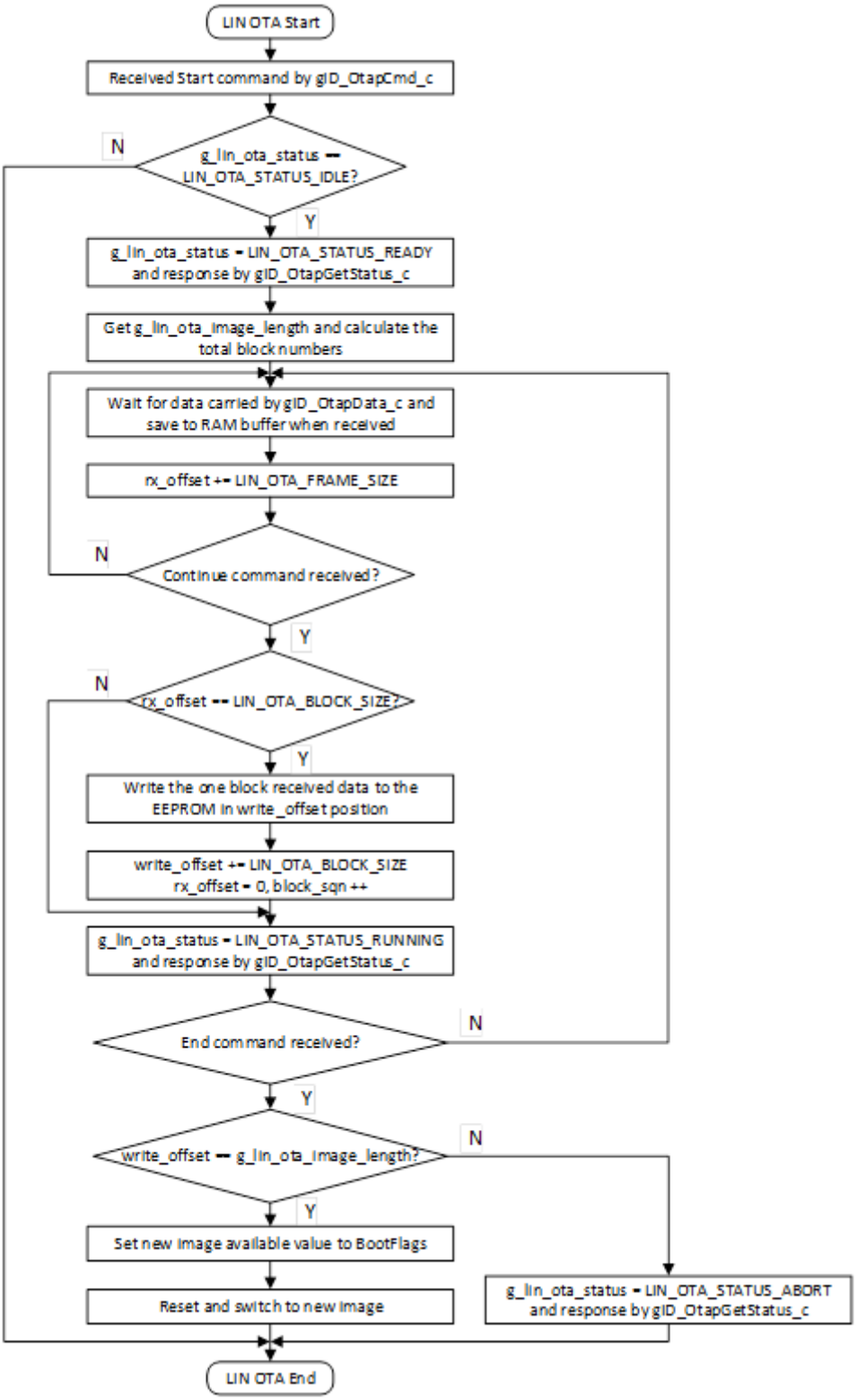


Figure 25. Image upgrading flow diagram of LIN slave

5.2 Transfer via CAN bus

You need to define 11-bit standard CAN identifiers for TX and RX identification in *flexcan_cfg.h*. CAN Node B should have the same RX identifier as the TX identifier of Node A and have the same TX identifier as the RX identifier of Node A.

Node A:

#define CAN_TX_IDENTIFIER (0x123)

#define CAN_RX_IDENTIFIER (0x321)

Node B:

Both CAN Node A and CAN Node B should be set to RX mode to listen the data of bus when idle.

#define CAN_TX_IDENTIFIER (0x321)

#define CAN_RX_IDENTIFIER (0x123)

Generally, CAN nodes broadcast data to other nodes on the bus, such as periodic sensor data. But for the image data, the transmitter should know the status of each frame/block sent, since any missed or incorrect data may cause the receiver upgrading failure. Considering the reliability and fast CAN baud rate, expect to receive a response of each frame transmitted from the receiver to confirm that the data was received correctly.

Define the commands in *flexcan_cfg.h* for the image upgrading as below:

```
typedef enum
{
    CAN_GEN_CMD_NONE = 0x00,
    CAN_GEN_CMD_OTA_CMD,
    CAN_GEN_CMD_OTA_DATA,
    CAN_GEN_CMD_OTA_STATUS
} can_general_cmd_t;
```

Figure 26. Code for defining can_general_cmd_t enumeration

- *CAN_GEN_CMD_OTA_CMD*: notifies Node B to start or end the image data transfer. *CAN_GEN_CMD_OTA_CMD* is put in Byte0, followed by *can_ota_cmd_c* in Byte1 in the data payload.

```
typedef enum
{
    CAN_OTA_CMD_NONE = 0x00,
    CAN_OTA_CMD_START,
    CAN_OTA_CMD_END
} can_ota_cmd_c;
```

Figure 27. Code for defining can_ota_cmd_c enumeration

- *CAN_GEN_CMD_OTA_DATA*: carries the image data for Node A and the received status of each frame for Node B. CAN supports maximum 8-byte data payload, while CAN FD supports maximum 64 bytes. Byte0 is used to carry the *CAN_GEN_CMD_OTA_DATA* value. For Node A, Byte1-2 is used for frame sequence number (0~65535). You can transfer a frame by 8 bytes image data for EEPROM alignment. So, CAN FD mode should be enabled in *flexcan_interrupt_transfer.c*to support (1+2+8=)11-byte length.

  #define USE_CANFD (1)

  You can carry more image data each frame, for example, 16 bytes, 32 bytes, to speed up the upgrading process.

  For Node B, Byte1 is used to put the ACK/NAK of the received frame.

- *CAN_GEN_CMD_OTA_STATUS*: lets Node A know the current status of Node B. Byte0 carries the *can_ota_status_t* byte defined in *flexcan_cfg.h*.

```
typedef enum
{
    CAN_OTA_STATUS_IDLE = 0x00,
    CAN_OTA_STATUS_READY,
    CAN_OTA_STATUS_RUNNING,
    CAN_OTA_STATUS_FINISH,
    CAN_OTA_STATUS_ABORT
} can_ota_status_t;
```

Figure 28. Code for defining can_ota_status_c enumeration

When upgrading starts, the Node A reads one block data from EEPROM and saves to RAM buffer firstly. It then sends them to Node B by frames. Define the sizes in *flexcan_cfg.h* and related reading variables in *flexcan_interrupt_transfer.c* in *can_a* project:

```
#define CAN_OTA_BLOCK_SIZE        (1024U)
#define CAN_OTA_FRAME_SIZE        (8U)
#define CAN_OTA_FRAMES_OF_BLOCK   (CAN_OTA_BLOCK_SIZE / CAN_OTA_FRAME_SIZE)

static uint8_t eeprom_read_buffer[CAN_OTA_BLOCK_SIZE] = {0};
uint32_t g_can_read_flash_offset = gBootData_ImageLength_Offset_c;
```

Figure 29. Code for defining CAN transmission sizes and reading variables

Node B keeps waiting for the data continuously. It sends response to Node A after receiving a frame of data and then saves them to the RAM buffer. It writes the data of RAM buffer to EEPROM when it receives a complete block. Define the sizes in *flexcan_cfg.h* and related writing variables in *flexcan_interrupt_transfer.c* in the *can_b* project:

```
#define CAN_OTA_BLOCK_SIZE        (1024U)
#define CAN_OTA_FRAME_SIZE        (8U)
#define CAN_OTA_FRAMES_OF_BLOCK   (CAN_OTA_BLOCK_SIZE / CAN_OTA_FRAME_SIZE)

static uint8_t eeprom_write_buffer[CAN_OTA_BLOCK_SIZE] = {0};
uint32_t g_can_write_flash_offset = gBootData_ImageLength_Offset_c;
```

Figure 30. Code for defining CAN reception sizes and writing variables

To construct the state machine of upgrading, define the *can_ota_stage_t* enumeration in *flexcan_cfg.h* and variables in *flexcan_interrupt_transfer.c* as follows:

```
typedef enum
{
    CAN_OTA_STAGE_IDLE = 0x00,
    CAN_OTA_STAGE_TX_DATA,
    CAN_OTA_STAGE_END
} can_ota_stage_t;

can_ota_status_t g_can_ota_status = CAN_OTA_STATUS_IDLE;
can_ota_stage_t g_can_ota_stage = CAN_OTA_STAGE_IDLE;
```

Figure 31. Code for defining can_ota_stage_t enumeration and variables

The image upgrading flow diagrams of CAN Node A and CAN Node B are shown in Figure 32 and Figure 33.

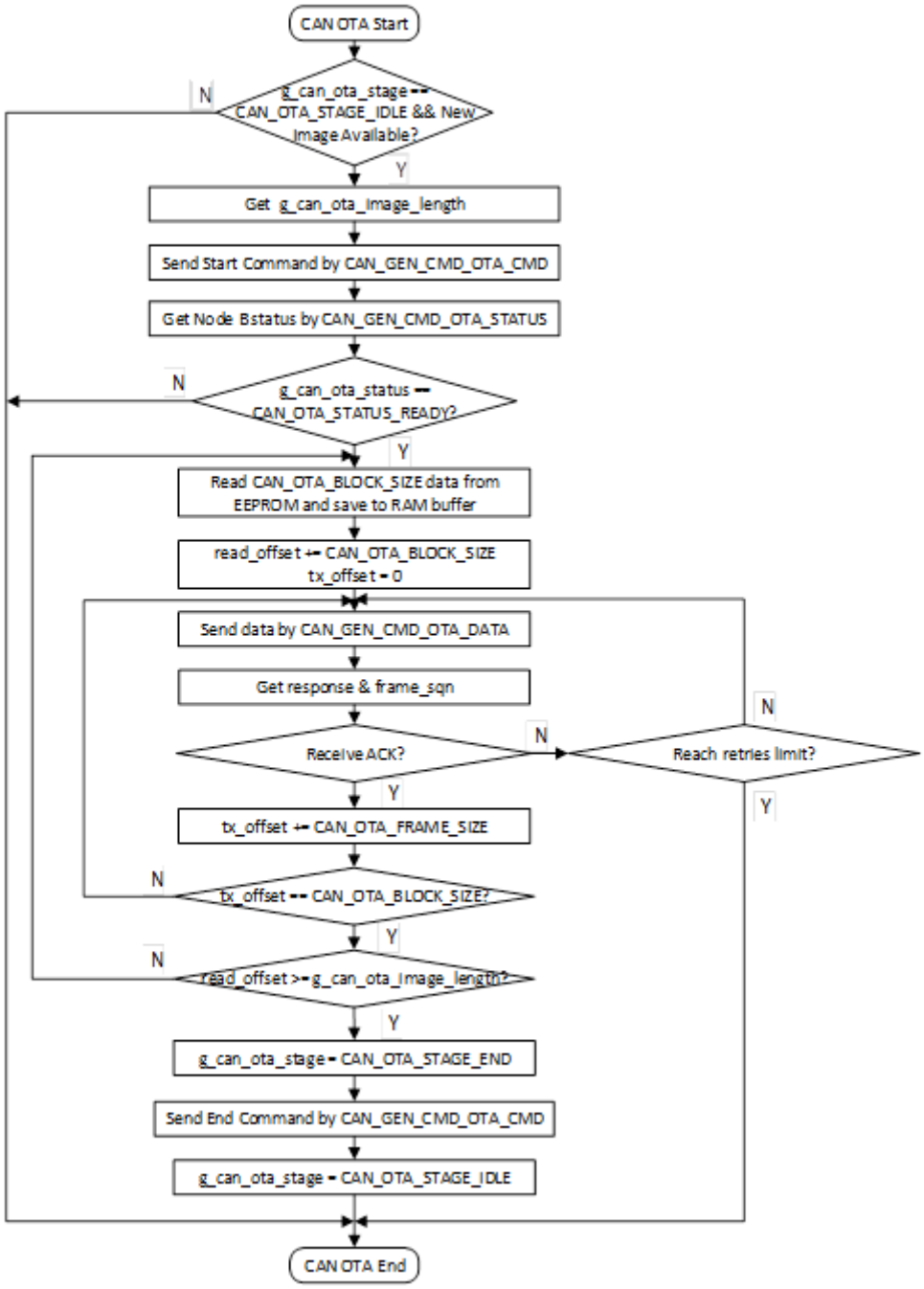


Figure 32. Image upgrading flow diagram of CAN Node A

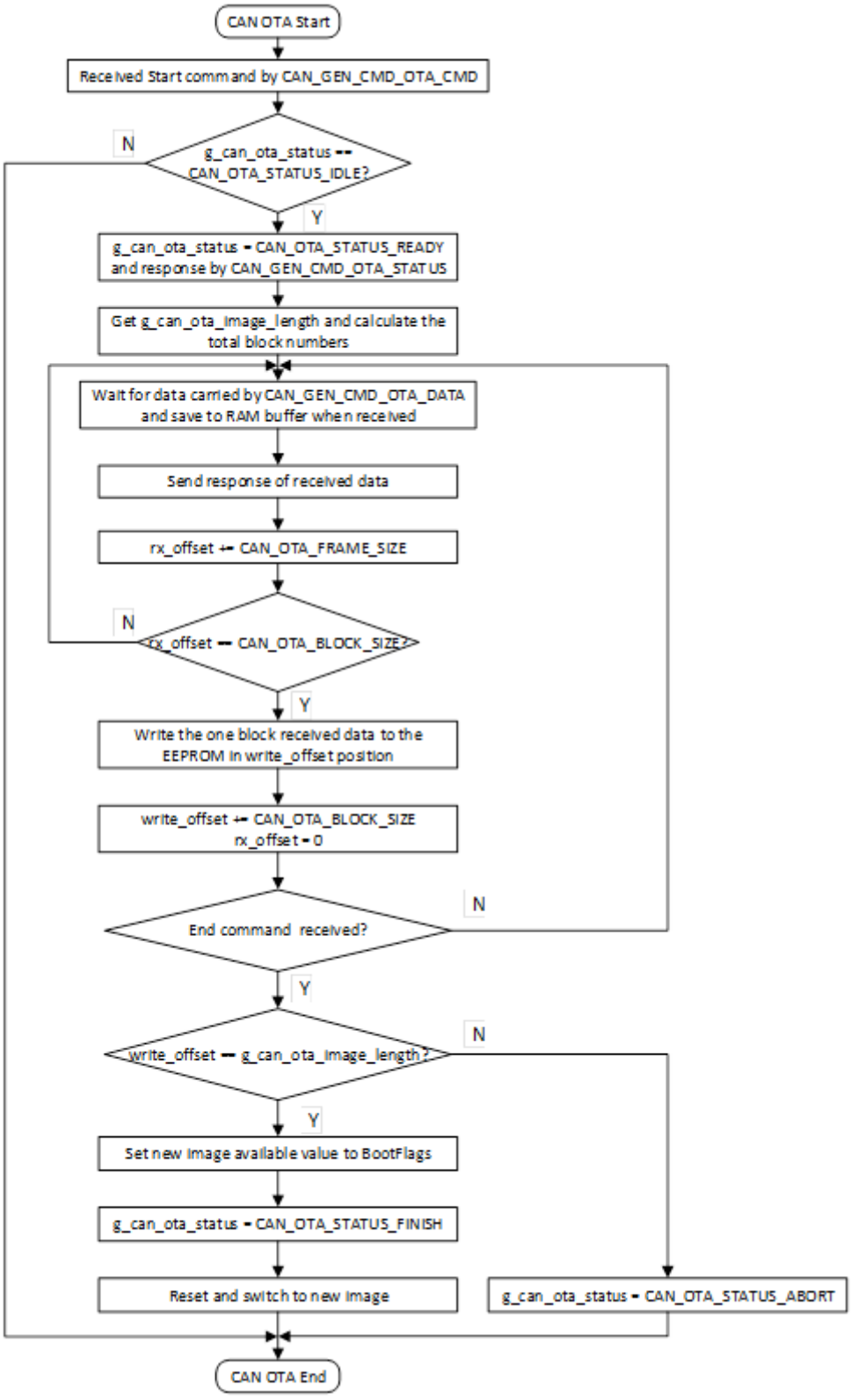


Figure 33. Image upgrading flow diagram of CAN Node B

The CAN baud rate is up to 1 Mbps. To consider the stability of upgrading process and support multi-nodes upgrading, you need to upgrade the nodes one by one if there are multiple identical Node B in the same bus. All Node B have different device identifiers.

CAN Node A requests the device identifiers of all Node B firstly, if it has new available image for them. Then it upgrades them serially according to the device identifiers.

Add one specific command *CAN_GEN_CMD_GET_DEV_ID* to get device identifiers of Node B for Node A, as shown below.

```
typedef enum
{
    CAN_GEN_CMD_NONE = 0x00,
    CAN_GEN_CMD_OTA_CMD,
    CAN_GEN_CMD_OTA_DATA,
    CAN_GEN_CMD_OTA_STATUS,
    CAN_GEN_CMD_GET_DEV_ID
} can_general_cmd_t;
```

Figure 34. Code for adding CAN_GEN_CMD_GET_DEV_ID in can_general_cmd_t enumeration

Define the low 16 bits of Bluetooth LE MAC address as the device identifier of Node B.

```
uint16_t g_can_device_id;

void BleApp_GenericCallback(gapGenericEvent_t *pGenericEvent)
{
    …
    switch (pGenericEvent->eventType)
    {
        case gPublicAddressRead_c:
        {
            uint8_t addr[2];
            FLib_MemCpy(addr, pGenericEvent->eventData.aAddress, sizeof(uint8_t) * 2);
            g_can_device_id = addr[0] + (addr[1] << 8);
        }
        break;
    …
    }
}
```

Figure 35. Code for getting device identifier in CAN Node B

CAN Node A requests the device identifiers of CAN Node B via *CAN_GEN_CMD_GET_DEV_ID* and opens the waiting window of 2 seconds. CAN Node B responds with its device identifier after a delay of random (0~1020) milliseconds on receiving the *CAN_GEN_CMD_GET_DEV_ID* request. CAN Node A saves the received identifiers to the buffer and starts upgrading them one by one after the waiting window closes.

6 Image switching

Once LIN slave/CAN Node B receives End command from LIN master/ CAN Node A, it indicates that the image data transfer has been finished. After writing the Image Length and Sector Bitmap to the header of selected storage area, LIN slave/CAN Node B should set the new image available value, *gBootValueForTRUE_c*, to the BootFlags of internal flash to let bootloader know that image switching is required.

Note: The Start Marker (0xDE, 0xAD, 0xAC, 0xE5) needs to be written before the Image Length field if the application selects the internal flash as the image storage area.

```
#if (gEepromType_d == gEepromDevice_InternalFlash_c)
    uint8_t start_marker[gBootData_Marker_Size_c] = {gBootData_StartMarker_Value_c};
    /* Write the Start marker at the beginning of the internal storage. */
    EEPROM_WriteData(gBootData_Marker_Size_c, gBootData_StartMarker_Offset_c, start_marker);
#endif
```

Figure 36. Code for adding Start Marker if internal flash is selected for storage

7 Testing

7.1 Hardware setup

1. Prepare two FRDM-KW36/38 DK boards, where one acts as LIN master/CAN Node A and the other one acts as LIN slave/CAN Node B. You also need two mini/micro USB cables, five Dupont female-to-female wires, and 12 V power.
2. For LIN testing, unmount R34 and R27 resistors of the LIN slave board.
3. Connect the pins between two FRDM-KW46/38 DK boards.
   - LIN connector: J13: pin 1 (LIN)
   - CAN connector: J23: pin 1, 2 (CAN_H, CAN_L)
   - Power: J13: pin 2, 4 (12V, GND)
4. Plug 12 V power adapter to J32 of one of the boards.

   See Figure 1 for the connections between boards.

   NOTE

   When using the auto baud rate feature in LIN testing, one additional Dupont male-to-male wire is needed to connect J1-5 and J2-9 on LIN slave board.
5. Use USB cables to connect J11 of boards with your personal computer. Download the generated OTAP bootloader (SDK\boards\frdmkw36\wireless_examples\framework|bootloader_otap) first and then download the LIN/CAN OTAP application firmware to the boards. Open the serial terminal (115200 bps, 8 data bits, no parity, 1 stop bit, no flow control) on your PC to watch the testing process.
6. Press SW1 to reset the boards. You can see the initial serial log on the PC terminal.

7.2 APP test

The NXP APP IoT Toolbox can be used to test the LIN/CAN image upgrading. Install it to your smart phone and switch on the Bluetooth of the handset. Test procedures are shown as below:

1. Click the "**OTAP**" icon in the IoT Toolbox main page, and start the scanning mode.
2. Put your LIN master/CAN Node A into advertising mode. Ensure to select the Bluetooth LE Peripheral GAP role using SW3 on the DK board if you use *w_uart* based examples. Press SW2 to start advertising and connect it with the APP.
3. **Open** the OTA file you have already generated with the steps described in Chapter 3 and loaded into your phone. The file size and valid file status displays if the file is available.
4. Click **Upload** to start the Bluetooth LE OTAP transfer from the APP to the LIN master/CAN Node A. The LIN master/CAN Node A starts the LIN/CAN transfer automatically after receiving the whole image.

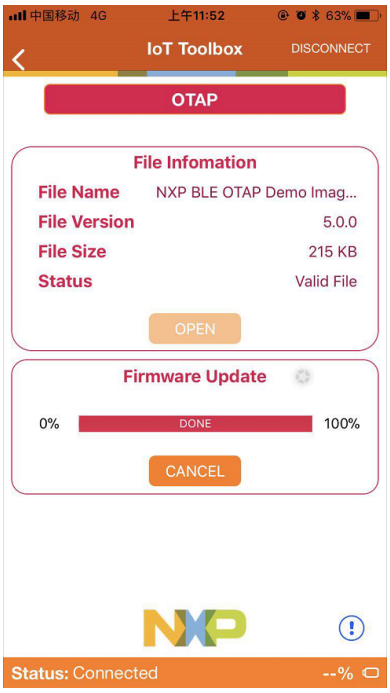
Figure 37 shows the transmission of "LIN Slave v2" image to the LIN slave from the LIN master. Once completed, the LIN slave switches to v2 from v1. At 19200 bps baud rate that set by calling *LIN_GetMasterDefaultConfig()*| *LIN_GetSlaveDefaultConfig()*, it will take about 6.5 minutes to complete the LIN upgrading for ~200 kB image. When external flash is selected, the reboot will add approximately 20 seconds more to the upgrading process.

Figure 38 shows the transmission of “CAN Node B v2” image to the CAN Node B from CAN Node A. Once completed, the CAN Node B switches to v2 from v1. At 1 Mbps baud rate that set by calling *FLEXCAN_GetDefaultConfig()*, it will take about 13 seconds to complete the CAN upgrading for ~200 kB image. When external flash is selected, the reboot will add approximately 20 seconds more to the upgrading process.

Table 1 shows the image transfer performance of LIN and CAN buses.

Table 1. Image Transfer Performance of LIN and CAN

Bus type	Baud Rate setting	Transfer time for ~200 kB image
LIN	19200 bps	6.5 minutes
CAN	1 Mbps	13 seconds

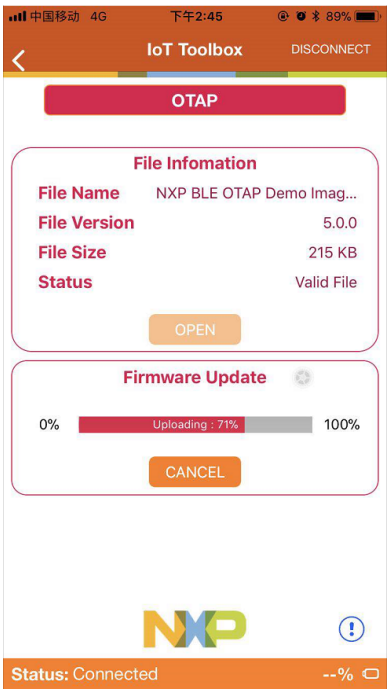


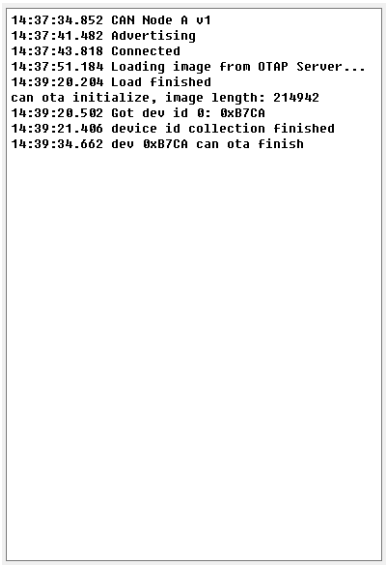

```
11:36:52.785 LIN Master v1
11:36:56.081 Advertising
11:36:57.294 Connected
11:37:11.783 Loading image from OTAP Server...
11:38:40.876 Load finished
11:38:40.890 lin ota init, image length: 215109
11:38:40.962 read block 0 from eeprom
11:38:42.977 read block 1 from eeprom
11:38:44.991 read block 2 from eeprom
11:38:47.006 read block 3 from eeprom
11:38:49.021 read block 4 from eeprom
11:38:51.037 read block 5 from eeprom
11:38:53.051 read block 6 from eeprom
11:38:55.066 read block 7 from eeprom
11:38:57.081 read block 8 from eeprom
11:38:59.096 read block 9 from eeprom
11:39:01.111 read block 10 from eeprom
11:39:03.128 read block 11 from eeprom
11:39:05.141 read block 12 from eeprom
11:39:07.156 read block 13 from eeprom
11:39:09.172 read block 14 from eeprom
11:39:11.186 read block 15 from eeprom
11:39:13.200 read block 16 from eeprom
11:39:15.216 read block 17 from eeprom
11:39:17.231 read block 18 from eeprom
11:39:19.246 read block 19 from eeprom
11:39:21.260 read block 20 from eeprom
11:39:23.275 read block 21 from eeprom
11:39:25.290 read block 22 from eeprom
11:39:27.305 read block 23 from eeprom
11:39:29.320 read block 24 from eeprom
11:39:31.334 read block 25 from eeprom
11:39:33.350 read block 26 from eeprom
11:39:35.365 read block 27 from eeprom
11:39:37.381 read block 28 from eeprom
11:39:39.395 read block 29 from eeprom
11:39:41.411 read block 30 from eeprom
11:39:43.426 read block 31 from eeprom
11:39:45.440 read block 32 from eeprom
```

```
11:36:54.120 LIN Slave v1
11:38:40.896 lin ota start...
11:38:40.996 length: 215109 bytes, total 211
blocks
11:38:42.952 block 0 written finished
11:38:44.934 block 1 written finished
11:38:46.965 block 2 written finished
11:38:48.964 block 3 written finished
11:38:50.995 block 4 written finished
11:38:52.993 block 5 written finished
11:38:55.025 block 6 written finished
11:38:57.023 block 7 written finished
11:38:59.056 block 8 written finished
11:39:01.054 block 9 written finished
11:39:03.085 block 10 written finished
11:39:05.083 block 11 written finished
11:39:07.114 block 12 written finished
11:39:09.114 block 13 written finished
11:39:11.144 block 14 written finished
11:39:13.143 block 15 written finished
11:39:15.173 block 16 written finished
11:39:17.173 block 17 written finished
11:39:19.205 block 18 written finished
11:39:21.203 block 19 written finished
11:39:23.234 block 20 written finished
11:39:25.233 block 21 written finished
11:39:27.263 block 22 written finished
11:39:29.262 block 23 written finished
11:39:31.294 block 24 written finished
11:39:33.292 block 25 written finished
11:39:35.324 block 26 written finished
11:39:37.323 block 27 written finished
11:39:39.355 block 28 written finished
11:39:41.353 block 29 written finished
11:39:41.365 block 30 written finished
11:39:45.383 block 31 written finished
11:39:47.413 block 32 written finished
11:39:49.413 block 33 written finished
11:39:51.444 block 34 written finished
```

```
11:44:37.613 block 176 written finished
11:44:39.612 block 177 written finished
11:44:41.643 block 178 written finished
11:44:43.643 block 179 written finished
11:44:45.674 block 180 written finished
11:44:47.674 block 181 written finished
11:44:49.706 block 182 written finished
11:44:51.706 block 183 written finished
11:44:53.737 block 184 written finished
11:44:55.736 block 185 written finished
11:44:57.768 block 186 written finished
11:44:59.767 block 187 written finished
11:45:01.798 block 188 written finished
11:45:03.798 block 189 written finished
11:45:05.829 block 190 written finished
11:45:07.828 block 191 written finished
11:45:09.860 block 192 written finished
11:45:11.860 block 193 written finished
11:45:13.892 block 194 written finished
11:45:15.891 block 195 written finished
11:45:17.921 block 196 written finished
11:45:19.921 block 197 written finished
11:45:21.953 block 198 written finished
11:45:23.952 block 199 written finished
11:45:25.983 block 200 written finished
11:45:27.983 block 201 written finished
11:45:30.014 block 202 written finished
11:45:32.015 block 203 written finished
11:45:34.047 block 204 written finished
11:45:36.046 block 205 written finished
11:45:38.076 block 206 written finished
11:45:40.077 block 207 written finished
11:45:42.110 block 208 written finished
11:45:44.108 block 209 written finished
11:45:44.342 write the last block, 210
11:45:44.363 lin ota end
ready to reset and switch
11:46:03.288 LIN Slave v2
```

Figure 37. Using IoT Toolbox to test LIN image upgrading





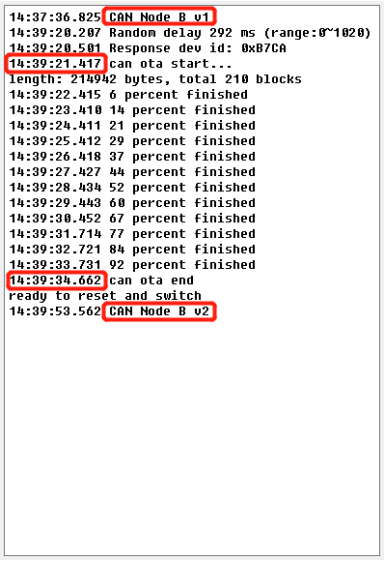


Figure 38. Using IoT Toolbox to test CAN image upgrading

8 Revision history

Table 2. Revision history

Rev	Date	Description
0.1	25 May 2020	Initial draft

Table continues on the next page...

Table 2. Revision history (continued)

Rev	Date	Description
0.2	27 May 2020	Add chapter “Image Storage” and Tab 1 - Image Transfer Performance of LIN and CAN.
0.3	2 June 2020	Add Figure - Data Flow of Image Upgrading System.
0.4	4 June 2020	Add KW38 support.
0.5	5 June 2020	Replace NXP Connectivity Test Tool capture to latest version.
0.6	7 July 2020	1. Reformat all the codes in this document 2. Add “Hardware setup” in “Testing” chapter 3. Mention the specific project files that add the codes
0.7	8 July 2020	Add captions for the figures displaying long code and change single/double line code to text form
0.8	16 July 2020	1. Updates in the description of topics, such as code porting procedure, hardware setup and so on, for clarity. 2. Add LIN/CAN OTA bus selection macro description.
0.9	16 October 2020	1. Add MCUXpresso IDE support 2. Add the NVM setting step in both MCUXpresso and IAR to reduce the size of the image sent over the air 3. Add the note for the additional hardware setting when using LIN auto baud rate feature 4. Add the binary output enablement in IDE

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: October 2020
Document identifier: AN12948