



Secondary Boot Loader (SBL) Software Design

Version 0.1

TABLE OF CONTENTS

1. PURPOSE	3
2. DEFINITIONS, ABBREVIATIONS, ACRONYMS	3
3. DESIGN.....	4
3.1 SYSTEM ARCHITECTURE	5
3.1.1 <i>Image Flasher</i>	5
3.1.2 <i>Image Loader</i>	6
4. DETAILED FLOW	8
4.1 PRE INITIALIZATION	8
4.2 MPU CONFIGURATION	8
4.3 PINMUX	8
4.4 SOC	9
4.5 USER INPUT.....	9
4.6 IMAGE FLASHER	9
4.7 IMAGE LOADER	10
4.8 DECISION ANALYSIS AND RESOLUTION (DAR).....	10
4.8.1 <i>TCMA vs TCMB</i>	10
4.9 CONFIGURABLE OPTIONS	11
4.10 RUNNING THE APPLICATION	11
5. STANDARDS, CONVENTIONS AND PROCEDURES	15
5.1 DOCUMENTATION STANDARDS	15
5.2 SOFTWARE DEVELOPMENT TOOLS	15
5.3 SAFETY STANDARDS	16
6. APPENDIX	16

TABLE OF FIGURES

Figure 1: Flash layout	5
Figure 2: Image Updater	6
Figure 3: Image Loader	7

Revision Control

Author Name	Description	Version	Date
Jyothi Pandit	Initial version	0.1	7/9/2018

1. Purpose

This document describes the design and implementation considerations for secondary bootloader software.

2. Definitions, Abbreviations, Acronyms

Term	Definition
SBL	Secondary BootLoader
TCM	Tightly Coupled Memory
CAN	Controller Area Network
MPU	Memory Protection Unit

3. Design

The secondary bootloader primarily is responsible for updating the application meta image in the SFLASH by receiving the image over a serial interface. It then loads and runs the updated application meta image.

The ROM(primary) bootloader always loads the SBL. Application can choose to either update or load and run the application meta image.

The SBL will ensure the factory default backup image is never erased or updated by the image updater.

The upgrade is done be for the entire meta image.

The SBL performs the checksum verification to verify the validity of the meta image it is trying to load from SFLASH. In the case where the image is corrupted or download is interrupted, it will provide a failsafe mechanism to reload the image. This is done automatically by the SBL by resetting the board.

If the primary meta image fails to load, the factory default backup image will be loaded.

If both fail, SBL will reset the board so the user can re-attempt the download of the meta image.

Secondary bootloader application aims at providing a reference design to fulfill the above requirement.

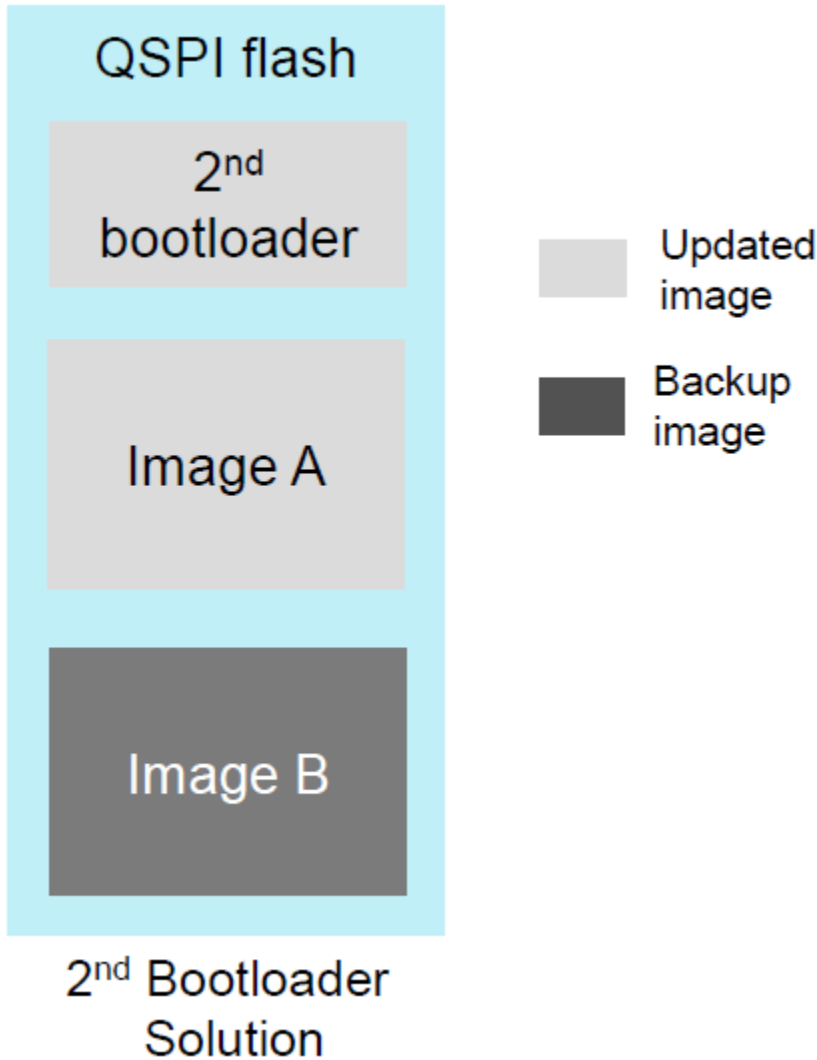


Figure 1: Flash layout

3.1 System Architecture

SBL doesn't use any environment variables. It will wait for a predetermined number of cycles waiting for user input else it will auto boot Image A.

The design of the secondary bootloader is divided into 2 parts.

3.1.1 Image Flasher

Update a new or existing application meta image in SFLASH. UART is used to download the meta image over a serial connection. Refer to the appendix to download the image using CANFD.

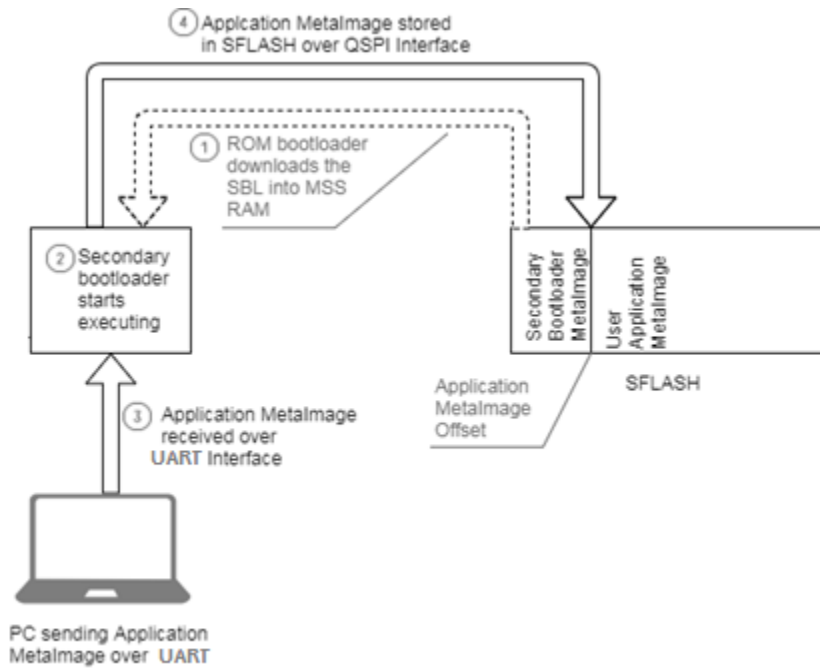


Figure 2: Image Updater

3.1.2 Image Loader

Downloads the existing application meta image from SFLASH into the RAMs of all subsystems after verifying the validity of the image.

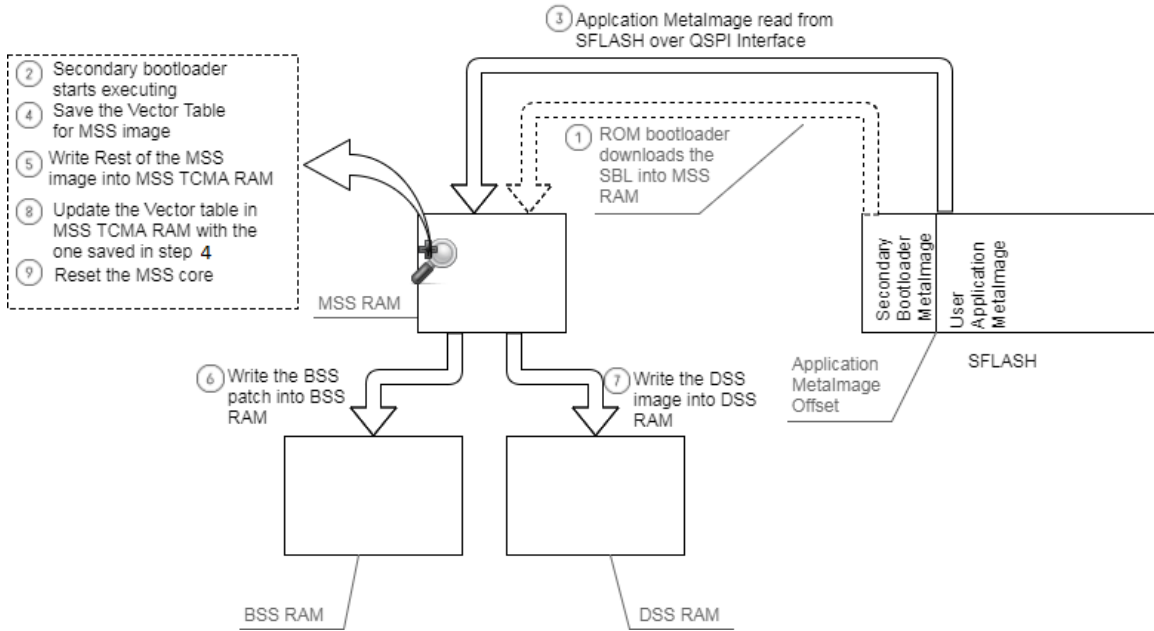


Figure 3: Image Loader

4. Detailed Flow

The following sections explain the SBL design in detail.

4.1 Pre Initialization

As most of the SBL code would be executing from the MSS TCMB memory, it needs to be copied from TCMA to TCMB as part of the pre-initialization routine.

The copy routine is coded in main.c. The copy routine is the only SBL function that will run in TCMA memory. The rest of the SBL routines run from TCMB memory.

The linker command file sbl_linker.cmd locates the main.c routine/variables in TCMA memory.

```
SECTIONS
{
  tcmlibs: > PROG_RAM ALIGN(8)
  {
    main.oer4f
    -l boot.aer4ft(.text)
  }
  systemHeap : {} > DATA_RAM
}
```

The following variables defined in the linker command file xwr_r4f.cmd will be used to copy the code to TCMB.

```
.text : {} > PROG_RAM ALIGN(8) run = DATA_RAM, LOAD_START(_libLoadStart),
LOAD_END(_libLoadEnd), RUN_START(_libRunAddr)

.const : {} > PROG_RAM ALIGN(8) run = DATA_RAM, LOAD_START(_constLoadStart),
LOAD_END(_constLoadEnd), RUN_START(_constRunAddr)
```

4.2 MPU Configuration

The following memory section settings are used in this application.

- MSS TCMA : Read, Write and Executable
- MSS TCMB : Read, Write and Executable
- BSS TCMA : Read, Write and Executable
- BSS TCMB : Read and Write

4.3 PinMux

The following peripherals are used for this application. Hence the pin muxing for these peripherals is required.

- QSPI : CLK, CS, D0(IN/OUT), D1(IN), D2(IN), D3(IN)
- UART : UART-Tx, UART-Rx

The pinmux setting will be left in the state configured by SBL. It will not be reset during SBL cleanup.

4.4 SOC

The existing SOC driver in MMWAVE SDK is modified to allow the SBL to:

- Configure the MPU as required by the SBL.
- Hold the DSP in reset during the SBL execution.

4.5 User Input

The UART is initialized with following configuration:

Baudrate: 115200
Data: 8 bit
Parity: None
Stop bit: 1 bit
Flow control: None

User can choose from one of the following option:

- a. Update meta image via UART.
- b. Download existing meta image from Flash.

4.6 Image Flasher

The following steps are performed in this phase:

- Initialize QSPI Flash interface.
- Erase the SFLASH for SBL_MAX_METAIMAGE_SIZE at an offset of SBL_METAIMAGE_OFFSET.

- Initialize the peripheral used to download the image.
- Wait to receive the image over UART for a pre-determined number of cycles.
 - User can use any standard serial protocol to receive the application meta image. Xmodem is used as an example to showcase the download over UART.
- Close the QSPI Flash interface.
- Close the UART interface.

4.7 Image Loader

The following steps are performed in this phase:

- Initialize the DMA.
- Initialize QSPI Flash interface.
- Parse the application meta image present in the SFLASH.
- Load each Sub-System image in their respective RAMs.
 - If no Error: Close the QSPI Flash interface.
 - If Error: Parse and load the factory default backup image.
- If load of both images fail, reset the board to re-attempt an application meta image download.

4.8 Decision Analysis and Resolution (DAR)

4.8.1 TCMA vs TCMB

Whether to execute the Secondary Bootloader code from TCMA or TCMB.

- **TCMA advantages over TCMB**

As the primary bootloader already load the code into the TCMA, no additional effort is required in the SBL application code. Thus, the execution time will be relatively less and the SBL code would be simpler.

- **TCMB advantages over TCMA**

If the SBL code is executing from the TCMA, the user application code has to be loaded at an offset larger or equal to the SBL code size. The available TCMA memory available for the user application will be reduced. Whereas, executing the code from TCMB allows the user application code to use the whole of TCMA

memory. Also, the TCMB memory is initialized by the application startup code ensuring there would be no reduction in the TCMB memory for user application.

- **Decision**

Apart from the startup code responsible for copying the code into the TCMB memory, the SBL code will be kept in TCMB memory.

4.9 Configurable options

Certain fields that can be modified by the application are discussed below:

#define SBL_METAIMAGE_OFFSET	(256 * 1024)
------------------------------	--------------

The user application meta image offset from the start of flash memory. This value shall be larger than the size of SBL meta image.

#define SBL_AUTOBOOT_COUNT	(20)
----------------------------	------

Number of seconds SBL will wait before autobooting the existing image. This allows the user to press a key to stop the autoboot and flash a new image.

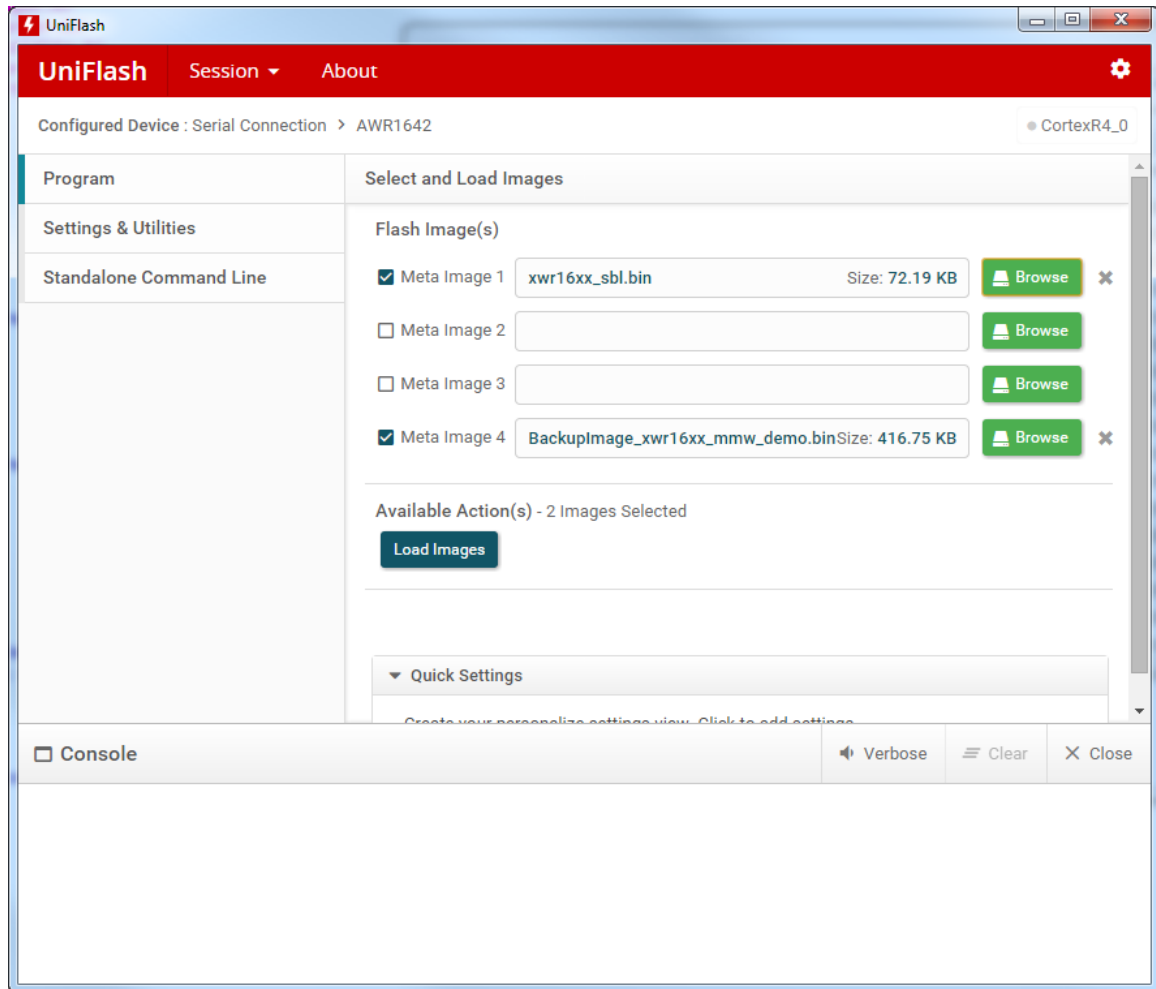
Refer to the top level header file for all the configurable fields.

4.10 Running the application

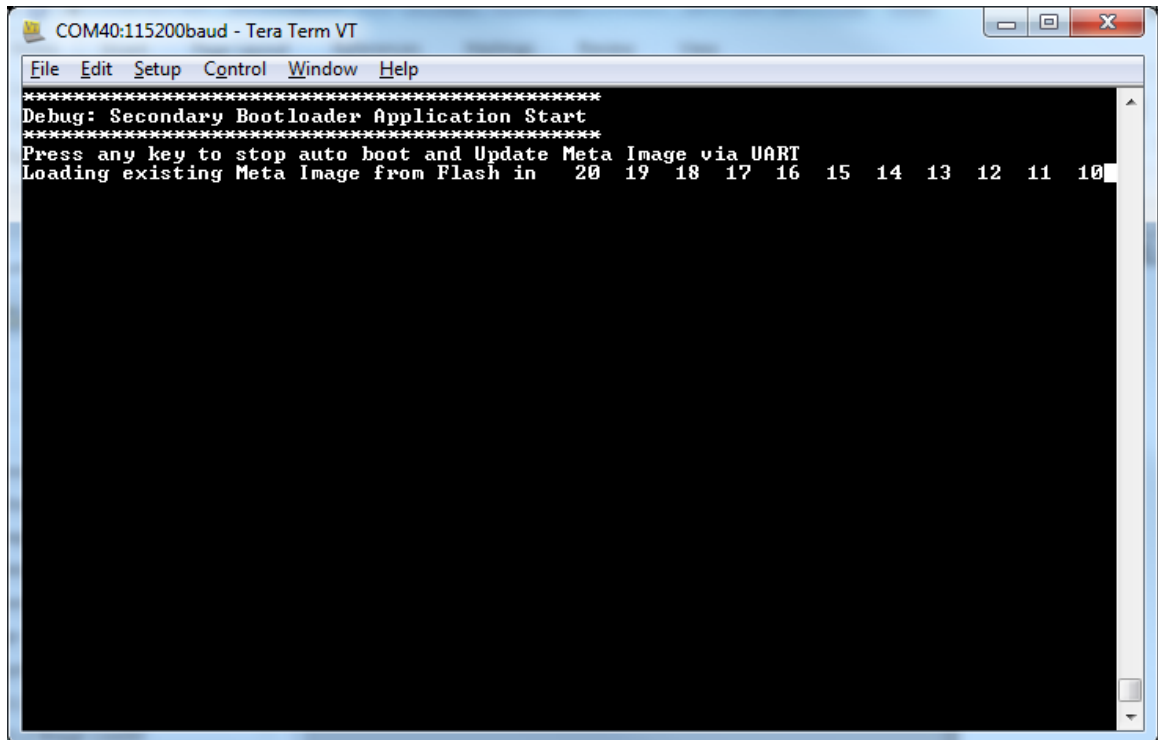
The meta image for SBL application will be created automatically along with the out file upon compiling this application. Note that this image will contain only the MSS binary (no BSS or DSS binary). This meta image needs to be flashed once using the conventional method (uniflash). From this point onwards, this image will reside in the SFLASH as the primary meta image (until a new image is flashed using the uniflash) and will be loaded each time after a cold or warm reset. Follow the below steps to run the application.

Note: Since the factory default backup image is not modified by the SBL, it also needs to be flashed once using the conventional method (uniflash).

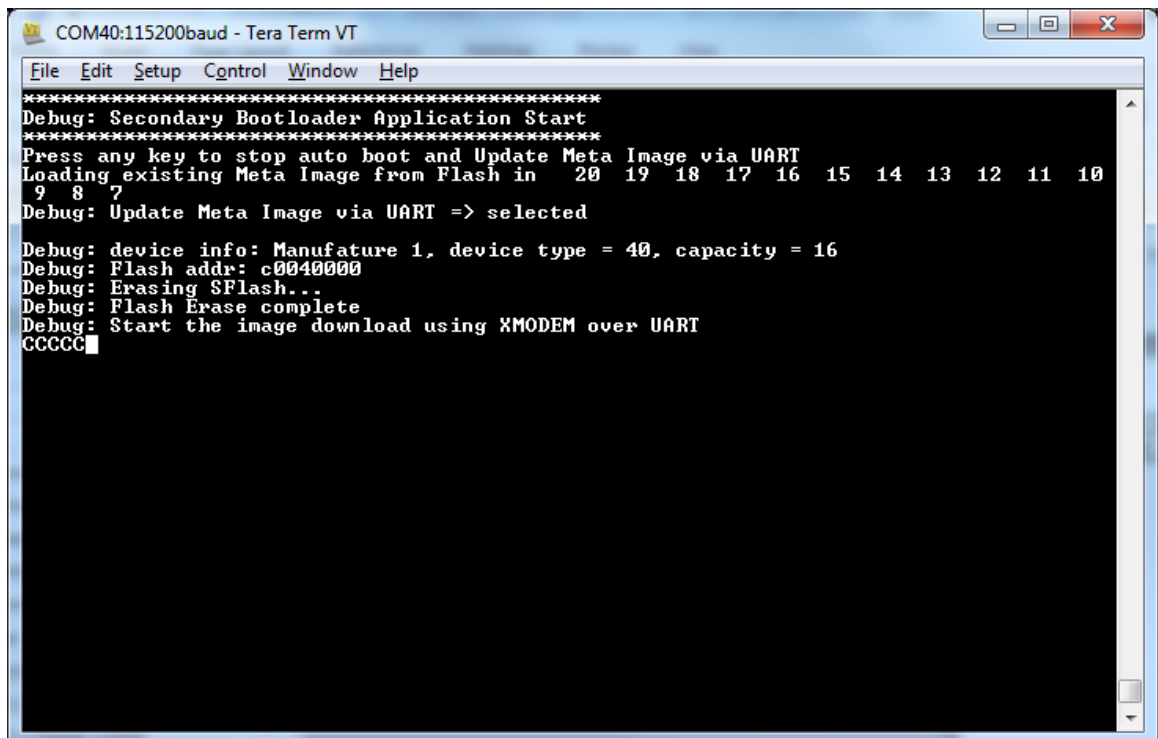
1. Load the SBL image and a factory default backup image as shown below.



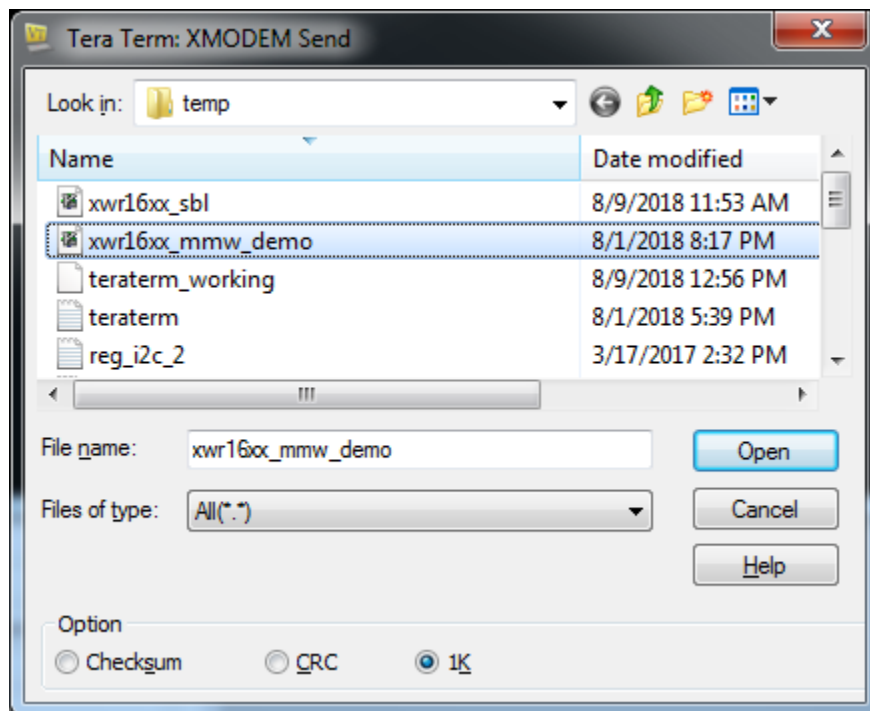
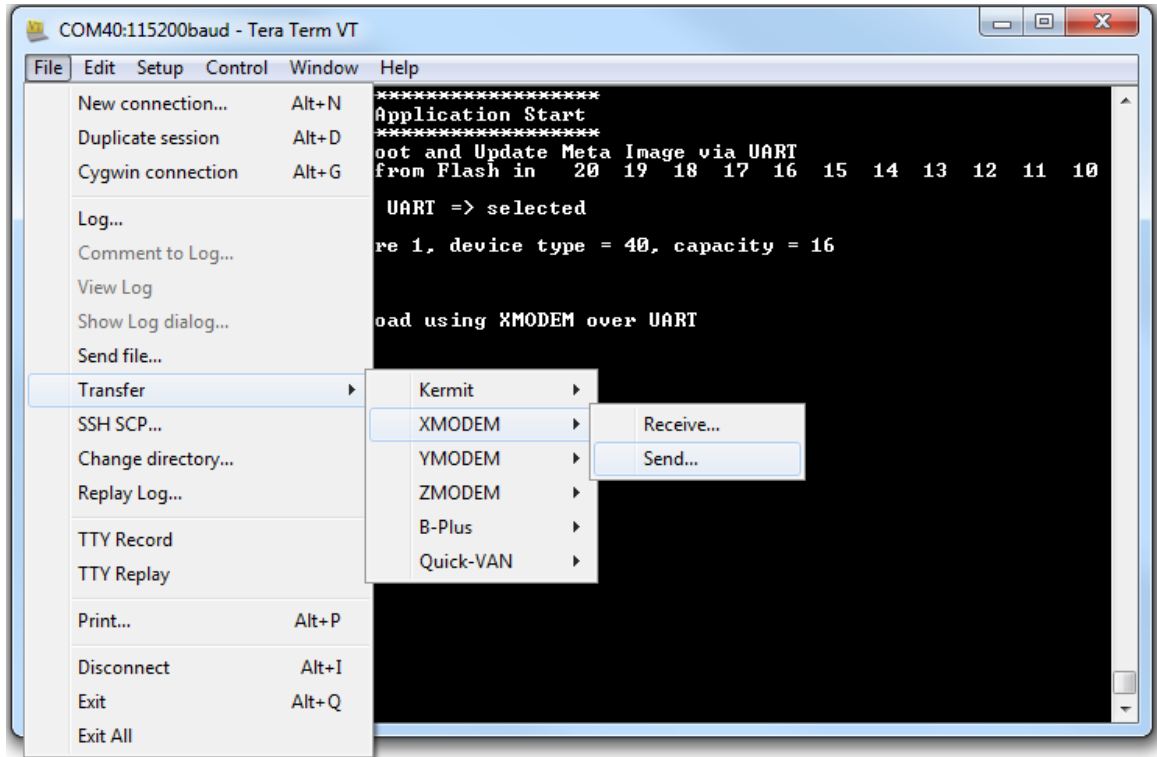
2. Find the COM port which shows as “*XDS110 Class Application/User UART*” under device manager.
3. Connect to the COM port found in previous step (using any serial terminal application like Teraterm, Hyperterminal etc.) and configure the port with the configuration specified above.
4. SBL will print the following message prompting the user to press any key to stop the autoboot process and load a new application meta image over UART.



5. Press any key to stop the autoboot process. You will see the following prints. The flash at SBL_METAIMAGE_OFFSET address will be erased.

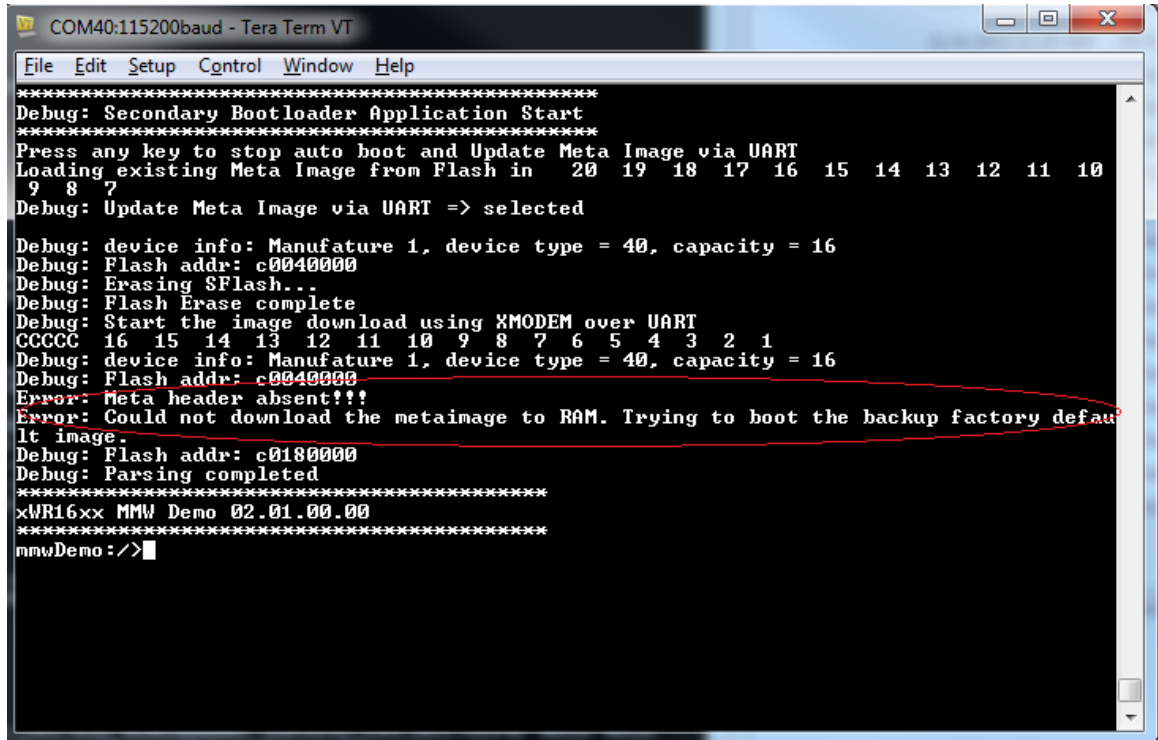


- Start the file download using XMODEM via the Teraterm.



7. Once the image is downloaded, SBL verifies the image and loads from SFLASH to RAM.

Note: If the application meta image boot fails for any reason, the backup image is loaded from SFLASH to RAM.



```

COM40:115200baud - Tera Term VT
File Edit Setup Control Window Help
*****
Debug: Secondary Bootloader Application Start
*****
Press any key to stop auto boot and Update Meta Image via UART
Loading existing Meta Image from Flash in 20 19 18 17 16 15 14 13 12 11 10
9 8 7
Debug: Update Meta Image via UART => selected

Debug: device info: Manufacture 1, device type = 40, capacity = 16
Debug: Flash addr: c0040000
Debug: Erasing SFlash...
Debug: Flash Erase complete
Debug: Start the image download using XMODEM over UART
CCCCC 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Debug: device info: Manufacture 1, device type = 40, capacity = 16
Debug: Flash addr: c0040000
Error: Meta header absent!!!
Error: Could not download the metaimage to RAM. Trying to boot the backup factory default image.
Debug: Flash addr: c0180000
Debug: Parsing completed
*****
xWR16xx MMW Demo 02.01.00.00
*****
mmwDemo : />

```

If both the images fail to load, SBL resets the board and goes back to step 4.

5. Standards, Conventions and Procedures

5.1 Documentation Standards

The driver's software is documented using doxygen. Test code is not documented with doxygen because of future restructuring for MCPI test framework compliance.

5.2 Software development tools

Refer to the mmWave SDK release notes for details on the software needed for compilation and debugging of this driver.

5.3 Safety Standards

MISRA-C compliance is under study and will be implemented in future as part of the mmWave SDK requirement.

6. Appendix

The current implementation of SBL uses UART as a serial interface to download the application meta image. The following section outlines the changes required to be made to download the file using CANFD instead of UART.

1. Initialization: The peripheral initialization code is abstracted in the SBL_transport_init() API. Modify the API to initialize the CANFD interface by calling CANFD_init() and set the bit times by calling CANFD_configBitTime(). Create a 3 receive message objects to:
 - a. Receive the data packets – All data packets will be sent using this message object.
 - b. Receive the terminate packet – Indicate the end of transmission.
 - c. Handshake message object – Special message Id used to indicate to SBL to initialize the above 2 message objects and wait for specific CAN message to open path for flashing the application meta image. E.g., “SBL_CANFD_UPDATE_MESSAGE_ID: 0x0CFD” This will make sure the flash update happens only when this particular message is exchanged.

The following pseudo code shows the above steps.

```

/*Intialize CANFD configuration parameters. */
memset(mcanCfgParams, sizeof(CANFD_MCANInitParams), 0);

mcanCfgParams->fdMode = 0x1U;
mcanCfgParams->brsEnable = 0x1U;
mcanCfgParams->txpEnable = 0x0U;
mcanCfgParams->efbi = 0x0U;
mcanCfgParams->pxhddisable = 0x0U;
mcanCfgParams->darEnable = 0x1U;
mcanCfgParams->wkupReqEnable = 0x1U;
mcanCfgParams->autoWkupEnable = 0x1U;
mcanCfgParams->emulationEnable = 0x0U;
mcanCfgParams->emulationFAck = 0x0U;
mcanCfgParams->clkStopFAck = 0x0U;
mcanCfgParams->wdcPreload = 0x0U;
mcanCfgParams->tdcEnable = 0x1U;
mcanCfgParams->tdcConfig.tdcf = 0U;
mcanCfgParams->tdcConfig.tdco = 8U;
mcanCfgParams->monEnable = 0x0U;
mcanCfgParams->asmEnable = 0x0U;
mcanCfgParams->tsPrescalar = 0x0U;
mcanCfgParams->tsSelect = 0x0U;
mcanCfgParams->timeoutSelect = CANFD_MCANTimeOutSelect_CONT;
mcanCfgParams->timeoutPreload = 0x0U;
mcanCfgParams->timeoutCntEnable = 0x0U;

```

```

mcanCfgParams->filterConfig.rrfe = 0x1U;
mcanCfgParams->filterConfig.rrfs = 0x1U;
mcanCfgParams->filterConfig.anfe = 0x1U;
mcanCfgParams->filterConfig.anfs = 0x1U;
mcanCfgParams->msgRAMConfig.lss = 127U;
mcanCfgParams->msgRAMConfig.lse = 64U;
mcanCfgParams->msgRAMConfig.txBufNum = 32U;
mcanCfgParams->msgRAMConfig.txFIFOSize = 0U;
mcanCfgParams->msgRAMConfig.txBufMode = 0U;
mcanCfgParams->msgRAMConfig.txEventFIFOSize = 0U;
mcanCfgParams->msgRAMConfig.txEventFIFOWaterMark = 0U;
mcanCfgParams->msgRAMConfig.rxFIFO0size = 0U;
mcanCfgParams->msgRAMConfig.rxFIFO0OpMode = 0U;
mcanCfgParams->msgRAMConfig.rxFIFO0waterMark = 0U;
mcanCfgParams->msgRAMConfig.rxFIFO1size = 64U;
mcanCfgParams->msgRAMConfig.rxFIFO1waterMark = 64U;
mcanCfgParams->msgRAMConfig.rxFIFO1OpMode = 64U;

mcanCfgParams->eccConfig.enable = 1;
mcanCfgParams->eccConfig.enableChk = 1;
mcanCfgParams->eccConfig.enableRdModWr = 1;

mcanCfgParams->errInterruptEnable = 1U;
mcanCfgParams->dataInterruptEnable = 1U;
mcanCfgParams->appErrCallBack = sblErrStatusCallback;
mcanCfgParams->appDataCallBack = sblDataCallback;

/* Initialize the CANFD driver. */
canHandle = CANFD_init(&mcanCfgParams, &errCode);
if (canHandle == NULL)
{
    System_printf("Error: CANFD Module Initialization failed [Error code %d]\n",
        errCode);
    return -1;
}
/* Configure the bit timing parameters. */
mcanBitTimingParams.nomBrp = 0x2U;
mcanBitTimingParams.nomPropSeg = 0x8U;
mcanBitTimingParams.nomPseg1 = 0x6U;
mcanBitTimingParams.nomPseg2 = 0x5U;
mcanBitTimingParams.nomSjw = 0x1U;

mcanBitTimingParams.dataBrp = 0x1U;
mcanBitTimingParams.dataPropSeg = 0x2U;
mcanBitTimingParams.dataPseg1 = 0x2U;
mcanBitTimingParams.dataPseg2 = 0x3U;
mcanBitTimingParams.dataSjw = 0x1U;

retVal = CANFD_configBitTime(canHandle, &mcanBitTimingParams, &errCode);
if (retVal < 0)
{
    System_printf("Error: CANFD Module configure bit time failed [Error code %d]\n",
        errCode);
    return -1;
}

```

```

/* Setup the handshake message object and wait for a message to check if the meta image has to
be updated or auto boot should resume. */
rxMsgObjectParams.direction = CANFD_Direction_RX;
rxMsgObjectParams.msgIdType = CANFD_MCANXidType_11_BIT;
rxMsgObjectParams.msgIdentifier = "SBL_CANFD_UPDATE_MESSAGE_ID;

rxMsgObjHandle = CANFD_createMsgObject(canHandle, &rxMsgObjectParams,
                                     &errCode);
if (rxMsgObjHandle == NULL)
{
    System_printf("Error: CANFD create Rx message object failed [Error code %d]\n",
                 errCode);
    return -1;
}

/*****Update meta image message object has been received. Proceed with downloading
the image and writing to flash. *****/

/* Setup the receive message object for receiving data packets. */
rxMsgObjectParams.direction = CANFD_Direction_RX;
rxMsgObjectParams.msgIdType = CANFD_MCANXidType_11_BIT;
rxMsgObjectParams.msgIdentifier = SBL_CANFD_DATA_MSG_ID;

rxMsgObjHandle = CANFD_createMsgObject(canHandle, &rxMsgObjectParams,
                                     &errCode);
if (rxMsgObjHandle == NULL)
{
    System_printf("Error: CANFD create Rx message object failed [Error code %d]\n",
                 errCode);
    return -1;
}

/* Setup the receive message object for receiving terminate packets. */
rxMsgObjectParams.direction = CANFD_Direction_RX;
rxMsgObjectParams.msgIdType = CANFD_MCANXidType_11_BIT;
rxMsgObjectParams.msgIdentifier = SBL_CANFD_TERMINATE_MSG_ID;

rxMsgObj2Handle = CANFD_createMsgObject(canHandle, &rxMsgObjectParams,
                                     &errCode);
if (rxMsgObj2Handle == NULL)
{
    System_printf("Error: CANFD create Rx message object failed [Error code %d]\n",
                 errCode);
    return -1;
}

```

2. Download: The application meta image download is abstracted in the SBL_transportGetFile () API. Modify the API to receive packets over CANFD interface and write them to SFLASH.

The following pseudo code shows the above steps.

```

/* Registered callback function to receive data. */
static void sblDataCallback(CANFD_MsgObjHandle handle, CANFD_Reason reason)
{

```

```

uint32_t buffOffset = (gRxPkts%NUM_PKT_IN_BUFF);

if (reason == CANFD_Reason_RX)
{
    retVal = CANFD_getData(handle, &id, &rxFrameType, &rxIdType,
        &rxDataLength, &dataBuff[buffOffset*64], &errCode);

    if (retVal < 0)
    {
        System_printf("Error: CAN receive data failed [Error code %d]\n", errCode);
        return;
    }

    /* Message ID for terminate message? */
    if (id == TERMINATE_MSG_ID)
    {
        gLastMsgFlag = 1;
    }

    gRxPkts++;
}
return;
}

/* SBL_transportGetFile () function. */
{
while (1)
{
    /* Wait till a new data packet is received. */
    while (gPktsWrt == gRxPkts);

    /* Check if the terminate Message Id is received? */
    if (gLastMsgFlag != 0)
    {
        break;
    }
    buffOffset = (gPktsWrt%NUM_PKT_IN_BUFF);
    QSPIFlash_singleWrite(QSPIFlashHandle, flashAddr, 64,
        (uint8_t*)&dataBuff[buffOffset*64]);

    /* Increment the Flash pointer. */
    flashAddr = flashAddr + 64;

    /* Increment the number of packets written. */
    gPktsWrt++;
    totDataLen = totDataLen + 64;
}
}

```

Run the CANFD PC application and send the application meta image.

3. Pinmux: Replace the pinmux setting for UART with CANFD.

```

/* Setup the PINMUX to bring out the XWR16xx CANFD pins. */
Pinmux_Set_OverrideCtrl(SOC_XWR16XX_PINE14_PADAE,

```

```
PINMUX_OUTEN_RETAIN_HW_CTRL, PINMUX_INPEN_RETAIN_HW_CTRL);  
    Pinmux_Set_FuncSel(SOC_XWR16XX_PINE14_PADAE,  
SOC_XWR16XX_PINE14_PADAE_CANFD_TX);  
  
    Pinmux_Set_OverrideCtrl(SOC_XWR16XX_PIND13_PADAD,  
PINMUX_OUTEN_RETAIN_HW_CTRL, PINMUX_INPEN_RETAIN_HW_CTRL);  
    Pinmux_Set_FuncSel(SOC_XWR16XX_PIND13_PADAD,  
SOC_XWR16XX_PIND13_PADAD_CANFD_RX);
```

4. SBL reads user input or prints debug, error messages on the console using UART peripheral. This functionality is abstracted via the SBL_printf() API. Modify the API to use suitable mechanism to achieve the above if required.