**OTA 2.0 of BDSLE**

**1 Wristband MCU Flash Memory management**

We suggest dividing MCU Code Memory to 3 parts. Boot loader, Image A, and Image B. Take EFM32G210F128 for example. This MCU has 128K flash memory. If the boot loader uses 4K bytes flash memory. Then the Image A and Image B both have the memory 62K.

|  |  |
| --- | --- |
| 0x0000F800 | Image B (62K) |
| 0x00001000 | Image A (62K) |
| 0x00000000 | 4K Boot Loader |

Boot loader is placed at the beginning of the flash memory. Per each MCU reset, the boot loader is the first program to run. It is responsible for checking which image has the newer version. Once checked, it will go to the newer image to run.

Image A and image B are the firmware with different versions. Both them have SPP (Serial Port Profile developed by BDE) implemented. If current running image is image A, it can receive the newer firmware data through SPP from smart phone or other central device and write the firmware data to the region of Image B. And after done the receiving and writing the newer firmware data, MCU will be reset by Image A. And the image B will get the right to run after the boot loader checked the newer firmware.

To implement this, MCU must have more double space than before. For example, if your project uses EFM32G200F64 and the firmware without OTA has 40K bytes, you must change the MCU to EFM32G210F128. Image A and Image B will occupy 80K more bytes.

**2 Steps of Updating Firmware through BLE**

***Step 1***:

Mobile reads the current wristband firmware version through the Air command **FWVERSION** defined by this document. The wristband sends response packet ***FWVERSIONVM.N***. M and N indicate a signal number. Fox example, if current running image A’s version is v2.5. Then response is ***FWVERSIONV2.5***.

Usually, the current running image is the newer version. The mobile checks whether the firmware to be updated has newer version than running image in the wristband. If newer, go to *step 2*, otherwise aborted.

***Step 2:***

Mobile determines which image of wristband is running and which image should be written by sending **FWSELECT** command. Wristband sends response **FWSELECT APPA** or **FWSELECT APPB**.

For example if current running image is B, mobile should select image A data to update. Note, on mobile side, each updating, the new firmware should include two images (image A and image B). And both the images have the same version number. Go to *step 3*.

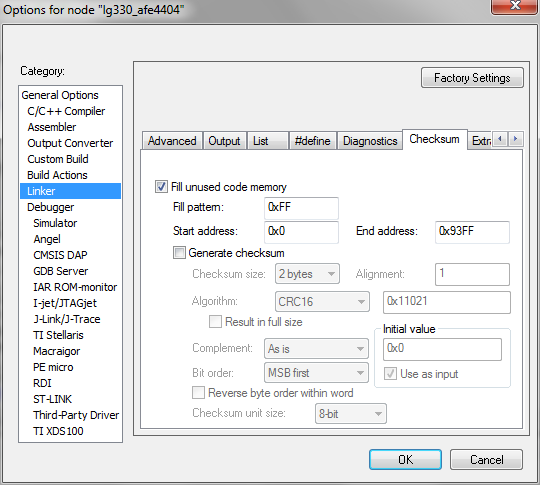
***Step 3:***

Mobile send command **UPDATE START** to the wristband. And then wristband sends **READY** as its response. Then the wristband will disconnect the BT connection and enter Programming mode after its advertising on again. NOTE: when the wristband enters the programming mode, it must disable internal RC oscillator by using API **HAL\_SetCtlClockSource(HAL\_CTL\_RC\_OSC\_DISABLE)**;. And for the highest updating speed, the wristband should update connection parameter to the minimum interval supported by Mobile when connection is just connected again (For iPhone, the supported minimum connect interval is 18.75ms).

Meanwhile, mobile start scanning the wristband advertising again. If there is no advertising wristband after 10 second elapsed. Mobile can prompt the user the updating is failed. If found advertising wristband, go to *Step 4*.

***Step 4:***

The firmware data is a binary file generated by EWARM. It has the extension name ‘.bin’. Note, not ‘.hex’ file. Each binary firmware is aligned with 512 bytes. This can be done easily be EWARM setting (Linker -> Checksum Setting):



Mobile divides the firmware data to N blocks. Each block has 512 bytes in length. If the firmware has M bytes size, the block number will be M/512.

Mobile encapsulates each block to the following packet format (520 bytes).

0x55 | BLOCK\_ID | 512 BYTES Block Firmware | CRCH | CRCL | CHECKSUMH | CHECKSUML | TAIL.

0x55 --- Header.

BLOCK\_ID– One byte block ID. BLOCK\_ID ranges 0~255. So the maximum firmware data is 256 \* 512 = 128K.

512 BYTES Block Firmware: 512 bytes block data from the binary file.

CRCH and CRCL : are high and low byte of 2-byte CRC code. The CRC can calculated by following alorigthm. It just includes the 512 bytes block data.

crc = (crc >> 8) | (crc << 8);

crc ^= data;

crc ^= (crc & 0xff) >> 4;

crc ^= crc << 12;

crc ^= (crc & 0xff) << 5;

CHECKSUMH | CHECKSUML: are the high and low byte of the 2-byte checksum code. The checksum is simply defined the sum of each byte, and discard the overflow part. See the following algorithm:

uint16 checksum = 0;

uint16 idx = 0;

for (idx = 0; idx < size; i++)

checksum += fw[idx];

TAIL: It fixes as 0x0000 or 0xFFFF. 0x0000 indicates this is not the last block. 0xFFFF indicates this the last block.

Because BLE ATT payload limitation, the above packet should be divided into 26 short ATT fragments to send. Each fragment can fill with 20 bytes. Each fragment can be sent without any response. And after done the 26 fragments the wristband will send:

0x55 | BLOCK\_ID **OK** if both the CRC and Checksum are right checked by wristband. **OK** are ASCII characters. And 0x55 and BLOCK\_ID are number value.

0x55 | BLOCK\_ID **NG** if something wrong. **NG** are characters.

If wrong, mobile waits for 5 second and resend this block. If repeat time counts up to 5, mobile sends **EXIT UPDATE** and wristband responses **EXITOK**.

If right, mobile continues to send next block.

If all blocks send complete, go to step 5.

***Step 5:***

Mobile sends **VERIFY** CRCH | CRCL | CHECKSUMH | CHECKSUML command to request the entire file CRC code and Checksum code.

Wristband responses: **VERIFYOK** or **VERIFYNG**. Wristband check whether CRC and checksum are matched. If matched **VERIFYOK** is the response, otherwise **VERIFYNG**.

When received **VERIFYOK** Mobile sends **RESTART** and wrist sends response **RESTART READY** and reset MCU immediately. If mobile received **VERIFYNG**, mobile send **EXIT UPDATE** and wristband send **EXITOK**. If wrong, mobile should prompt user to retry.

**3 Command Detail**

Each command contains 20 bytes. Some of them are ASCII characters and HEX byte parameters. If there ASCII characters and parameters are not enough. Then 0x00 should be filled.

*3.1 Firmware Version Query Command:*

uint8 PDU\_CMD[20] = {‘**F’, ‘W’, ‘V’, ‘E’, ‘R’, ‘S’, ‘I’, ‘O’, ‘N’,** 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

uint8 PDU\_RSP[20] = {‘**F’, ‘W’, ‘V’, ‘E’, ‘R’, ‘S’, ‘I’, ‘O’, ‘N’, ‘V’, ‘2’, ‘.’,‘5’,** 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0}; // This is an example v2.5.

*3.2 Firmware Image Select Command*

uint8 PDU\_CMD[20] = {‘**F’, ‘W’, ‘S’, ‘E’, ‘L’, ‘E’, ‘C’, ‘T’,** 0,0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

uint8 PDU\_RSP[20] = {‘**F’, ‘W’, ‘S’, ‘E’, ‘L’, ‘E’, ‘C’, ‘T’, ‘ ’, ‘A’, ‘P’, ‘P’,‘A’,** 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0}; // This is an example mobile should send image A to //wristband

NOTE: There is a space character between “FWSELECT” and “APPA” in the PDU\_RSP.

*3.3 Update Start Command*

uint8 PDU\_CMD[20] = {‘**U’, ‘P’, ‘D’, ‘A’, ‘T’, ‘E’, ‘ ’, ‘S’, ‘T’**, **‘A’**, **‘R’**, **‘T’**, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

uint8 PDU\_RSP[20] = {‘**R’, ‘E’, ‘A’, ‘D’, ‘Y’,** 0, 0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

NOTE: There is a space character between “UPDATE” and “START” in the PDU\_CMD.

*3.4 Exit Update Command*

uint8 PDU\_CMD[20] = {‘**E’, ‘X’, ‘I’, ‘T’, ‘ ’, ‘U’, ‘P’, ‘D’, ‘A’**, **‘T’**, **‘E’**, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

uint8 PDU\_RSP[20] = {‘**E’, ‘X’, ‘I’, ‘T’, ‘O’, ‘K’**, 0, 0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

*3.5 Verify Entire File Command*

uint8 PDU\_CMD[20] = {‘**V’, ‘E’, ‘R’, ‘I’, ‘F’, ‘Y’, 0xAA, 0xBB, 0xCC, 0xDD**, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

0xAA and 0xBB are CRCH and CRCL respectively. 0xCC and 0xDD are CHECKSUMH and CHECKSUUML respectively. After received the command, the wristband should check whether the CRC and Check sum are equal to the value send by mobile.

uint8 PDU\_RSP\_OK[20] = {‘**V’, ‘E’, ‘R’, ‘I’, ‘F’, ‘Y’, ‘O’, ‘K’,** 0, 0, 0, 0, 0, 0 ,0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

uint8 PDU\_RSP\_FAIL[20] = {‘**V’, ‘E’, ‘R’, ‘I’, ‘F’, ‘Y’, ‘N’, ‘G’,** 0, 0, 0, 0, 0, 0 ,0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

*3.5 Reset Wristband MCU*

uint8 PDU\_CMD[20] = {‘**R’, ‘E’, ‘S’, ‘T’, ‘A’, ‘R’, ‘T’**, 0, 0,0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

uint8 PDU\_RSP[20] = {‘**R’, ‘E’, ‘S’, ‘T’, ‘A’, ‘R’, ‘T’, ‘ ’, ‘R’, ‘E’, ‘A’, ‘D’, ‘Y’,** 0 ,0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0, 0 , 0, 0, 0};

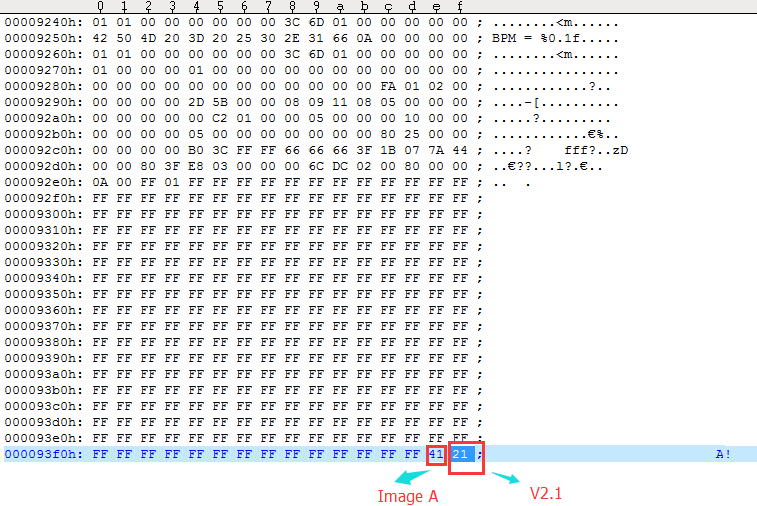
NOTE: There is a space character between RESTART and READY.

**4 Firmware Binary File Format**

Firmware shall align with 512 bytes. This can be done by setting IAR option in its linker configuring interface (this is mentioned above).

And each updating, the IAR engineer should provide two images to Mobile App builder. Both the two images indeed have the same version number. The distinction is just they have different offset and different last two bytes.

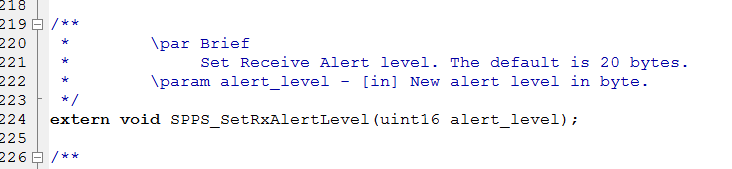
The last byte of one image is version number. It is BCD format. The last second byte is image (A or B) indicator which is an ASCII character. The following is screenshot of Image A.



**5 Other Consideration**

When wristband entering programming mode, it should disable EM9301 RC oscillator and update the connection interval to a minimum vale supported by mobile. The purpose for this is to improve the stability when the large data is transferring. However, after exit the programming mode, wristband shall enable EM9301 internal RC oscillator and update the connection interval to a normal value to minimize the power consumption.

By default, image A gets the right to run if the two images have the same version.

When entering the programming mode, the SPP Service Receiving Alert Levels should set to 20 bytes. Because each packet in this mode is 20 bytes aligned. This can be down by call API: 

If possible, we suggest set SPPS Rx Buffer size larger than or equal to 20 bytes and Rx Alert Level to 20 bytes.

If your SPPS does not include this API, please update SPPS.h header file. The latest BDSLE stack library version (**BDSLE\_IAR\_M3\_Lib\_1\_3\_1\_r12\_nolimit.a**) includes this API.

**6 Revision**

R0.0 2015-03-20: Initial version.