Rigado LLC
3950 Fairview Industrial Dr. SE Suite 100
Salem, Oregon 97302
866-6-RIGADO ◆ modules@rigado.com
www.rigado.com/modules

RigDFU

Secure Over-the-Air & UART Bootloader

1. Introduction

An embedded bootloader is a small piece of software that enables an embedded system to update its main application firmware. This allows for the ability to add features and fix defects found after manufacture. Rigado provides a Secure Bluetooth Low Energy and UART bootloader to all customers who purchase nRF5-based BMD series modules, allowing them to keep their products in the field updated while protecting their valuable Intellectual Property (IP). RigDFU is only available on our nRF5-base BMD series modules. This will be referred to as BMD series modules in the rest of this document.

1.1 Feature List

- Over-the-Air and UART firmware updates
- Main application, bootloader, and SoftDevice are updateable
- End-to-end firmware encryption using AES-128 in EAX Mode
- Protects the device from imposter OTA firmware
- Fail-safe update process
- Customer configurable encryption key
- Easy to use with full suite of tools (Windows, Linux, OSX)
- Compatible with Rigablue libraries for iOS and Android

2. Secure Bootloader

Rigado's RigDFU bootloader uses 128-bit AES-EAX encryption to secure the transfer of firmware images to BMD Series modules. Security is achieved by encrypting the binary firmware image in a controlled environment prior to transfer. After encryption the encrypted image is sent to the BMD module using either available transfer method. Once the encrypted image is received by the BMD module, either over-the-air via Bluetooth Low Energy or the UART interface, it will decrypt and verify the integrity of the image. Once decryption and verification is successful, the firmware image is programmed to flash memory and the bootloader will start the application. With the secure bootloader in place, your firmware can be protected before it leaves your servers.

3. SoftDevice Support

The Rigado secure bootloader currently supports applications running on Nordic Semiconductor S110 SoftDevice version 8.0.0, S130 SoftDevice version 2.0.0, and S132 SoftDevice version 2.0.0. A bootloader binary is provided for each SoftDevice version.

Table of Contents

4. BMD-200 Pin Functions

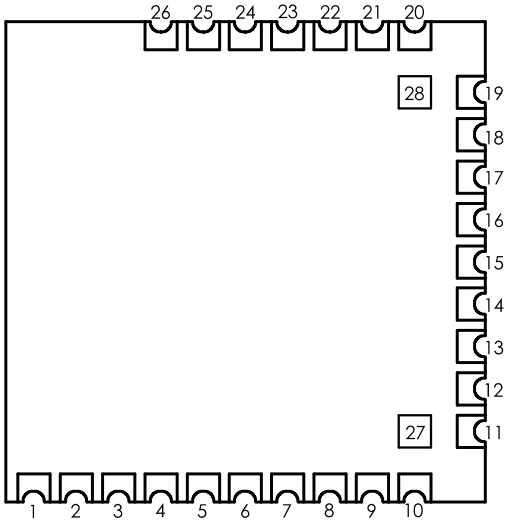

Figure 1- BMD-200 Pin out (Top View)

Pin	Name	Direction	BMD-200 RigDFU Pin Functions
11	P0.00	In/Out	Not Used by RigDFU / BMDware DTM UART RX
12	P0.01	In/Out	Not Used by RigDFU / BMDware DTM UART TX
21	P0.09	In	**RigDFU Serial Bootloader UART RX -** Disabled when BLE is in-use
22	P0.10	Out	**RigDFU Serial Bootloader UART TX** - Disabled when BLE is in-use
24	SWDIO	In/Out	**Debug I/O / RESET**
25	SWDCLK	In	**Debug Clock**
18	V_{CC}	Power	**+1.8V to +3.6V DC -** 1µF - 4.7µF ceramic capacitor is recommended between V_{CC} and GND
1, 2, 3, 4, 7, 10, 19, 26, (27, 28 opt.)	GND	Power	**Electrical Ground**
15, 16, 17, 20, 23, 5, 6, 8, 9, 13, 14	--	N/A	Not Used by RigDFU

Table 1 - BMD-200 RigDFU Pin Functions

5. BMD-300/301 Pin Descriptions

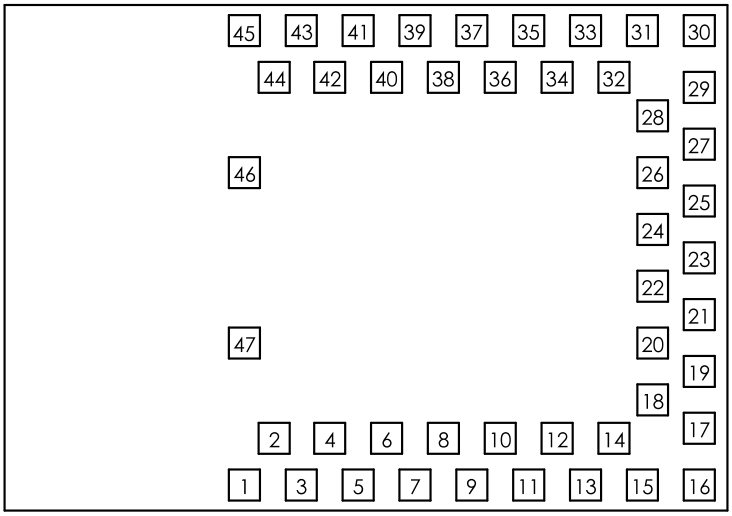

Figure 2 – BMD-300 Pin out (Top View)

Pin	Name	Direction	BMD-300 RigDFU Pin Descriptions
22	P0.06	Out	**Serial Bootloader UART TX -** Disabled when BLE is in-use
24	P0.08	In	**Serial Bootloader UART RX** - Disabled when BLE is in-use
27	P0.11	Out	Not Used by RigDFU / BMDware DTM UART RX
28	P0.12	In	Not Used by RigDFU / BMDware DTM UART RX
17	V_{CC}	Power	**+1.7V to +3.6V DC -** 1µF - 4.7µF ceramic capacitor is recommended between V_{CC} and GND
1,2,3,4, 5,16,18, 29,30,45, 46, 47	GND	Power	**Electrical Ground**
6,7,8,9, 10,11,12, 13,14,15, 19,20,21, 23,25,26, 31,32,33, 34,35,36, 37, 38,39, 40,41,42	--	N/A	Not Used by RigDFU

6. Memory Layout

The memory layout of the bootloader versions is as follows:

6.1 SoftDevice S110 v7.1.0 (Deprecated; No Longer Available)

Application	Start Address	End Address	Size (Bytes)
SoftDevice 7.1.0	0x00000	0x15FFF	90112 (0x16000)
User Application	0x16000	0x27BFF	72704 (0x11C00)
User Application Data[1]	0x27C00	0x28BFF	4096 (0x1000)
Bootloader Swap Space	0x28C00	0x3A7FF	72704 (0x11C00)
Bootloader	0x3A800	0x3F7FF	20408 (0x5000)
Bootloader Settings Data	0x3F800	0x3FBFF	1024 (0x400)
Rigado Bootloader Data	0x3FC00	0x3FFFF	1024 (0x400)

[1]*User Application Data is maintained through application updates*

6.2 SoftDevice S110 v8.0.0 (BMD-200-A and BMD-200-B only)

Application	Start Address	End Address	Size (Bytes)
SoftDevice 8.0.0	0x00000	0x17FFF	98304 (0x18000)
User Application	0x18000	0x28BFF	68606 (0x10C00)
User Application Data[1]	0x28C00	0x29BFF	4096 (0x1000)
Bootloader Swap Space	0x29C00	0x3A7FF	68606 (0x10C00)
Bootloader	0x3A800	0x3F7FF	20408 (0x5000)
Bootloader Settings Data	0x3F800	0x3FBFF	1024 (0x400)
Rigado Bootloader Data	0x3FC00	0x3FFFF	1024 (0x400)

[1]*User Application Data is maintained through application updates*

6.3 User Data

The User Application data space has moved for RigDFU on the S130 and S132. To maintain backwards compatibility, the location of this space remains unchanged for the S110 version of RigDFU. The User Data area is now just after the Swap space and allows applications to work with a default pstorage configuration. Many of the examples from Nordic use pstorage and this change allows for its use without modification.

6.4 SoftDevice S130 v2.0.0 (BMD-200-B Only)

Application	Start Address	End Address	Size (Bytes)
SoftDevice S132 2.0.0	0x00000	0x1AFFF	110592 (0x1B000)
User Application	0x1B000	0x2A3FF	62464 (0xF400)
Bootloader Swap Space	0x2A400	0x397FF	62464 (0xF400)
User Application Data[1]	**0x39800**	**0x3A7FF**	**4096 (0x1000)**
Bootloader	0x3A800	0x3F7FF	20408 (0x5000)
Bootloader Settings Data	0x3F800	0x3FBFF	1024 (0x400)
Rigado Bootloader Data	0x3FC00	0x3FFFF	1024 (0x400)

[1]*User Application Data is maintained through application updates*

6.5 SoftDevice S132 v2.0.0 (BMD-300 Series Only)

Application	Start Address	End Address	Size (Bytes)
SoftDevice S132 2.0.0	0x00000	0x1BFFF	114688 (0x1C000)
User Application	0x1C000	0x46FFF	176128 (0x2B000)
Bootloader Swap Space	0x47000	0x71FFF	176128 (0x2B000)
User Application Data[1]	**0x72000**	**0x74FFF**	**12288 (0x3000)**
Bootloader	0x75000	0x7DFFF	36864 (0x9000)
Bootloader Settings Data	0x7E000	0x7EFFF	4096 (0x1000)
Rigado Bootloader Data	0x7F000	0x7FFFF	4096 (0x1000)

[1]*User Application Data is maintained through application updates*

7. Development Setup

Before the secure bootloader can be used, some setup is needed on the development machine.

7.1 Rigado GitHub repositories

Rigado maintains software repositories at https://github.com. Request access at the Rigado.com website. The "bootloader-tools" repository, described in Section 9, below, is necessary for RigDFU. Other repositories contain example BMD module firmware and applications for iOS and Android mobile devices.

7.2 Install Required Tools

The bootloader scripts for performing firmware updates require a number of tools to work properly in addition to the bootloader-tools repository. The following sections denote these required tools and provide additional information where necessary.

7.2.1 Linux Setup

Installation commands listed below are based on Ubuntu Linux.

Note: *To use the Over-the-air (OTA) Update tools on Linux, built-in Bluetooth 4.0 hardware or a Bluetooth Low Energy USB dongle are necessary.*

1. Update the Ubuntu package lists
   a. `sudo apt-get update`
2. Install Python 3.x and Python Serial libraries:
   a. `sudo apt-get install python3 python3-pip`
   b. `python3 -m pip install pySerial`
3. Install the following tools for OTA Updates (Ubuntu commands shown):
   a. `sudo apt-get install bluetooth bluez libbluetooth-dev`
4. Install node.js
   a. `curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -`
   b. `sudo apt-get install -y nodejs`
5. Install git
   a. `sudo apt-get install git`
6. Obtain the Rigado GitHub bootloader-tools repository
   a. Change directory to somewhere you want to store the repository
   b. `git clone https://github.com/rigado/bootloader-tools`
7. Install noble and other required modules
   a. `cd bootloader-tools/update-tools/ble`
   b. `npm install`
8. Attach a JLink programmer or Rigado BMD Evaluation board to the machine

7.2.2 OS X Setup

This guide requires the use of the OS X package manager, Brew, and the OS X Terminal app. Other package managers may be used with similar results so long as they include the necessary packages. Use of other package managers will not be covered at this time.

Check out Brew at http://brew.sh. If Brew is not currently installed, it can be installed from the Brew website.

1. Install Node.js and Node.js Package Manager:
    a. `brew install node`
    b. `brew install npm`
2. Install Python 3.x:
    a. `brew install python3`
    b. `python3 -m pip install pySerial`
3. Install Segger JLink tools
    a. https://www.segger.com/jlink-software.html
4. Attach a JLink programmer or Rigado BMD Evaluation board to the machine

Alternatively, all of the above tools can be installed from source or their respective install packages.

7.2.3 Windows Setup

Note: *The use of OTA Update tools is not currently supported on Windows. Future releases of bootloader-tools may include this ability. Only UART updates may be used on Windows.*

1. Install Python 3.x:
    a. https://www.python.org/downloads
    b. Add the path to python.exe to your system path (Ex: C:\Python34)
    c. From console: `python -m pip install pySerial`
2. Install Segger JLink tools:
    a. https://www.segger.com/jlink-software.html
3. Attach JLink programmer or Rigado BMD Evaluation board to the machine

7.3 A Note about Invoking Python

When running the below scripts, all python commands may instead be either python3 on Linux and OS X or <Path to Python 3.4>\python.exe on Windows depending on the PATH variable for the shell program in use. Use “python --version” to check which version of python is invoked. All programming should be performed using Python 3.x. If the shell is pointing at Python 2.x, then either adjust the PATH for the shell or run the commands using the absolute path to the Python 3.x installation.

8. Module MAC Address

At the factory, Rigado programs all modules with a unique MAC address. The nRF5 series parts do not have permanent storage for an assigned MAC address. Rigado stores the MAC address for the BMD series modules in the UICR. However, the UICR is erased anytime the debugging interface on the nRF5 is used to perform a full chip erase (such as programming the bootloader). In order to preserve the Rigado MAC address, the program.py script provides a save MAC (-sm) option. This option instructs the script to read out the current MAC address from the UICR and re-write it during programming. If another MAC address is preferred, the MAC (-m) option can be used to program any MAC address. Note that if the MAC address is programmed to FF:FF:FF:FF:FF:FF, then the MAC address will be considered empty and the random public address read from the FICR will be used instead.

For more information regarding the UICR and FICR, refer to the nRF5 Series Reference Manuals at infocenter.nordicsemi.com.

Note: *Due to the nature of readback protection on the nRF52 series parts, it is impossible for our toolset to save the MAC address. If `-sm` is specified during programming of the BMD-300 Series modules, this option will be ignored and the MAC address will be cleared to FF:FF:FF:FF:FF:FF. The address preprogrammed into the FICR will be used.*

8.1 Bootloader MAC Address

The bootloader MAC address is slightly different than the normal module MAC address. Some systems cache all Bluetooth Low Energy service and characteristic data upon the initial first connection. When resetting into the bootloader, it is important that the central device can appropriately discover the new services and characteristics that are available on the bootloader rather than those available on the application firmware. In order to facilitate proper discovery, the bootloader's MAC address is inverted from the original value.

8.2 Inversion Formula

To determine the true MAC address of the bootloader, invert all of the bits of the true MAC address and then `or` the Most Significant Byte with C0.

Example:
Original MAC - 94549300A57D
Bootloader MAC - BEABC6FFA82

9. Secure Bootloader Modes

The Rigado bootloader has two main modes of operation: `Secure` and `Unsecure`.

9.1 Unsecure Mode

The unsecure bootloader performs a simple checksum on the whole image and then copies it into application flash after the firmware update process has completed. The entire firmware image is sent unencrypted. A device sniffing Bluetooth packets or capturing the UART data can acquire the firmware during transfer. This mode allows anyone to update the unit and is typically used for development.

9.2 Secure Mode

Secure mode uses encrypted images that are signed and encrypted prior to OTA or serial-wire transfer. Since the data sent is encrypted, sniffing the data will not yield any useful results unless the private key for the encrypted is known to the attacker.

- To increase security, Rigado recommends each device have a unique private key.

The firmware is not considered decrypted until the entire image transfer is complete. Only signed images can be loaded once encryption is enabled. Read-back protection is enabled on the BMD series modules to prevent any SWD programmer from reading flash memory. However, the device is erasable using a full chip erase.

10. Bootloader Tools

Rigado provides various programming and image generation tools for use with the secure bootloader. The tools are broken into three different categories:

Programming - Python and Segger JLink scripts which install the bootloader, appropriate SoftDevice version, and if requested, application firmware. A MAC address and/or encryption key can also be specified.

Binary Generation Tools - Python scripts and software for generating unencrypted and encrypted binaries for the purpose of firmware updates only.

Firmware Update Tools - Node.js and Python scripts providing update capabilities both over the wire via a Serial port and OTA via Bluetooth Low Energy[1].

[1]*OTA Update tools via Node.js are currently only supported on OS X and Linux.*

10.1 Bootloader Tools Folder Structure

The folders for the bootloader tools are organized into three folders:

- **programming**
  - binaries - Contains binaries for bootloader, SoftDevice, and other required binaries
  - program.py - Python script for programming the bootloader and all necessary binaries
- **image-tools**
  - genimage - Contains genimage.py which is used to generate unencrypted application firmware update binaries
  - signimage - Contains the signimage application which encrypts the output from genimage based on the supplied input key
- **update-tools**
  - ble - Contains dfu.js which performs an OTA update on Linux and OS X systems using Bluetooth Low Energy hardware
  - serial - Contains dfu.py which performs an update over a serial port

11. Programming Tools

The programming tools are used to install the bootloader to BMD series modules. Additionally, the programming tools can install an application binary.

11.1 Installation of the Bootloader

The bootloader can be installed either unsecured or secured depending on the need. The bootloader can be secured at a later time if required. In addition, the main application binary can be installed at the same time. The following sections describe how to install the bootloader for various scenarios. The bootloader is not secured at Rigado's factory.

11.2 Unsecure Installation

To install the bootloader without security:

```
python program.py -m 945493ffffff
```

11.3 Secure Installation

To install the bootloader and enable the encryption features, use the following:

```
python program.py -m 945493ffffff -k <16 hex pair key>
```

Examples:

```
python program.py -m 945493ffffff -k 00112233445566778899aabbccddeeff
python program.py -m 945493ffffff -k d6460dd794904bca992ada885310cfa1
```

If the key is programmed to all Fs, then the key is considered empty and the bootloader will behave as unsecure.

11.4 Application Installation when Programming the Bootloader

In addition to installing the bootloader, the programming script can also be used to install an application binary to the application memory space in flash. The application binary may be in either Intel HEX or binary format.

Steps

1. Add the -a option to any of the above programming script recipes
    a. Examples
        1. `python3 program.py -m 945493ffffff -a <path to app>.<hex or bin>`
        2. `python3 program.py -m 945493ffffff -a <path to app>.<hex or bin> -k 00112233445566778899aabbccddeeff`

12. Image Tools

This section describes how to generate binaries for performing Over-the-air (OTA) and Serial based updates. OTA and Serial updates use the same binary output from these tools. The only difference is the transport layer. This is not the same binary as used by the programming tool, program.py.

12.1 Update Binary Overview

The Firmware Update process involves sending a binary image to the bootloader which has additional information embedded into the beginning of the image. This information informs the bootloader of how much data to expect and what data it is being sent. The data sent to the bootloader will either be encrypted or unencrypted. The bootloader determines whether the data is encrypted or unencrypted based on whether or not a key has been set during programming of the bootloader. If encrypted, only encrypted application binaries will be accepted by the bootloader.

Steps

1. Generate unencrypted firmware update binary
2. Optionally, generate encrypted update binary
3. Perform update via OTA dfu.js script, Serial dfu.py script, or via a mobile device

12.2 Generating Unsigned Application Binaries

A binary application from a tool such as `fromelf.exe` cannot directly be used for a firmware updates. Instead, an Intel Hex file is required. Intel Hex files can be easily generated using Keil, IAR, or GNU make. The genimage.py script provided by Rigado will generate a firmware update binary from the Intel Hex application binary.

```
python genimage.py -a <input.hex> -o <output.bin>
```

The output of genimage.py is used to perform a firmware update to an unsecured bootloader.

12.3 Generating Encrypted Application Binaries

If encryption is being used, then the unsigned application binary needs to be signed and encrypted using the signimage tool. The signimage tool uses the unencrypted firmware update binary and the device private key as inputs and outputs the encrypted application image.

Linux
```
./signimage-linux <input.bin> <output.bin> <key>
```

OS X
```
./signimage-osx <input.bin> <output.bin> <key>
```

Windows 32-bit
```
signimage-32 <input.bin> <output.bin> <key>
```

Windows 64-bit
```
signimage-64 <input.bin> <output.bin> <key>
```

13. Update Tools

The update tools provide the ability to update the application firmware either OTA or via a wired serial port. The following sections explain how to use both of these tools.

13.1 Performing An OTA Update from Linux and OS X

An OTA update can be performed on Linux and OS X using the Node.js script dfu.js located in <sd version>/update-tools/ble. This script is capable of programming both encrypted and unencrypted firmware images as well as setting the private key for a device[1].

13.1.1 Unencrypted Update

Before running an update, ensure that the binary for the update has been generated using the genimage.py tool. A binary output directly from tools such as fromelf.exe will not work as it will not have the appropriate information to inform the bootloader of the type of update.

To perform an unencrypted OTA update:

```
node dfu.js <binary file>
```

Example:

```
node dfu.js blinky.bin
```

Note: *The device must already be in the bootloader prior to running dfu.js. Dfu.js does not know how to find a device that is running an application. Another way is to start dfu.js, and then reset the device manually.*

13.1.2 Encrypted Update

Before an encrypted update is possible, a private key must first be programmed to the device. Typically, the key is programmed to the device when programming the bootloader. However, it is possible to use dfu.js to set the key only. Once set, the key cannot be changed or erased by dfu.js[1].

To set the key on a device with no key:

```
sudo node dfu.js --newkey <key>
```

Example:

```
sudo node dfu.js --newkey 00112233445566778899aabbccddeeff
```

Next, an encrypted image must be generated with the key for the device. Check out the Generating Encrypted Binaries section for information on how to generate an encrypted binary. Once the encrypted binary is ready:

```
sudo node dfu.js <encrypted-binary>
```

In the previous example, the binary blinky.bin was sent over as an unencrypted binary. This time, the encrypted version, blinky-secure.bin is sent:

```
sudo node dfu.js blinky-secure.bin
```

[1]*The private key for a device is sent without encryption. A private key should only be assigned when the device is in a securely controlled area. Once the private key is assigned, dfu.js cannot change it.*

13.2 Performing Serial Updates from Linux, OS X, and Windows

Rigado provides a Python script for updating the device application firmware via a connected serial port. The script, dfu.py, is found in update-tools/serial. The bootloader UART uses the following settings:

- 115200 baud
- 8-bits
- 1 stop bit
- No Parity

The serial bootloader can perform both encrypted and unencrypted updates via the same image generation tools previously described. In addition to performing application firmware updates, the device key can be programmed and changed via dfu.py.

All rules applying to OTA updates also apply to the serial update interface.

Note: *The device must already be in the bootloader prior to running dfu.py. Dfu.py does not know how to find a device that is running an application. Another way is to start dfu.py, and then reset the device manually.*

13.2.1 Input Parameters for dfu.py

-s: Specifies the serial port to use for communications
-i: Specifies the input binary file
--newkey: Specifies a new key that should be programmed to the device
--oldkey: Specifies the old key that will be replaced by new key. If no key is present, then --oldkey must be specified as ffffffffffffffffffffffffffffffff.

To erase the key from a device fully (i.e. unsecure the device), the current key must be provided and --oldkey must be specified as ffffffffffffffffffffffffffffffff.

13.2.2 Serial Unencrypted Update

To perform an unencrypted Serial update:
python dfu.py -s <serial port> -i <application binary>

Examples:

```
python dfu.py -s /dev/tty.usbserial-DN002U6U -i blinky_ota.bin
python dfu.py -s COM26 -i blinky_ota.bin
```

13.2.3 Serial Encrypted Update

To perform an encrypted update, the device must first have a private key assigned. If it does not, then encrypted updates are unavailable.

To set a private key on the device when no key is programmed:

```
python dfu.py -s <serial_port> --oldkey ffffffffffffffffffffffffffffffff --
newkey <key>
```

Examples:

```
python3 dfu.py -s /dev/tty.usbserial-DN002U6U --oldkey
ffffffffffffffffffffffffffffffff --newkey 0123456789abcdef0123456789abcdef

C:\Python34\python.exe dfu.py -s COM26 --oldkey
ffffffffffffffffffffffffffffffff --newkey 0123456789abcdef0123456789abcdef
```

To change the private key for a device when a key is already present:

```
python dfu.py -s <serial_port> --oldkey <old_key> --newkey <key>
```

Examples:

```
python3 dfu.py -s /dev/tty.usbserial-DN002U6U --oldkey
0123456789abcdef0123456789abcdef --newkey 00112233445566778899aabbccddeeff

C:\Python34\python.exe dfu.py -s COM26 --oldkey
0123456789abcdef0123456789abcdef --newkey 00112233445566778899aabbccddeeff
```

To unsecure the bootloader:

```
python dfu.py -s <serial_port> --oldkey <old_key> --newkey
ffffffffffffffffffffffffffffffff
```

To perform an encrypted update:

```
python dfu.py -s <serial_port> -i <encrypted_binary>
```

Examples:

```
python dfu.py -s /dev/tty.usbserial-DN002U6U -i blinky_ota_secure.bin
python dfu.py -s COM26 -i blinky_ota_secure.bin
```

14. Getting the Bootloader version from your Application

The version information for the bootloader is programmed at a static offset in the bootloader image. This information is offset 0x1000 bytes from the beginning of the bootloader image. All source listed below is contained in the src folder of the repository.

14.1 Start Addresses

BMD-200	BMD-300
0x3B800	0x76000

14.2 Retrieving the version information

The bootloader version information is described by the following typedefs:

```
typedef enum version_type_e
{
    VERSION_TYPE_RELEASE = 1,
    VERSION_TYPE_DEBUG
} version_type_t;

typedef enum softdevice_support_e
{
    SOFTDEVICE_SUPPORT_S110 = 1,
    SOFTDEVICE_SUPPORT_S120,     //Not used
    SOFTDEVICE_SUPPORT_S130,
    SOFTDEVICE_SUPPORT_RESERVED1,
    SOFTDEVICE_SUPPORT_RESERVED2,
    SOFTDEVICE_SUPPORT_RESERVED3,
    SOFTDEVICE_SUPPORT_RESERVED4,
    SOFTDEVICE_SUPPORT_RESERVED5
} softdevice_support_t;

typedef enum hardware_support_e
{
    HARDWARE_SUPPORT_NRF51 = 1,
    HARDWARE_SUPPORT_NRF52,
    HARDWARE_SUPPORT_RESERVED1,
    HARDWARE_SUPPORT_RESERVED2,
    HARDWARE_SUPPORT_RESERVED3,
    HARDWARE_SUPPORT_RESERVED4,
    HARDWARE_SUPPORT_RESERVED5
} hardware_support_t;
```

```
typedef struct rig_firmware_info_s
{
    uint32_t magic_number_a;        //Always 0x465325D4
    uint32_t info_size;             //Size of this structure
    uint8_t version_major;
    uint8_t version_minor;
    uint8_t version_rev;
    uint32_t build_number;
    version_type_t version_type;
    softdevice_support_t sd_support;
    hardware_support_t hw_support;
    uint16_t protocol_version;
    uint32_t magic_number_b;        //Always 0x49B0784C
} rig_firmware_info_t;
```

The magic number values are randomly chosen 32-bit values to help ensure the data retrieved is actually the version information. The following function can be used to retrieve the bootloader information.

```
/** @file bootloader_info.c
*
* @brief This module provides functions to retrieve the installed
bootloader
*        version information.
*
* @par
* COPYRIGHT NOTICE: (c) Rigado
* All rights reserved.
* Source code licensed under Software License Agreement in
license.txt.
* You should have received a copy with purchase of BMD series product
* and with this repository.  If not, contact modules@rigado.com.
*/

#if defined(NRF52)
#define BOOTLOADER_START_ADDR       0x75000
#elif defined(NRF51)
#define BOOTLOADER_START_ADDR       0x3A800
#endif
#define BOOTLOADER_INFO_OFFSET      0x1000
#define BOOTLOADER_INFO_ADDR        (BOOTLOADER_START_ADDR +
BOOTLOADER_INFO_OFFSET)
#define MAGIC_HEADER                0x465325D4
#define MAGIC_FOOTER                0x49B0784C

#include <stdint.h>
#include <string.h>

#include "nrf_error.h"
```

```
#include "rig_firmware_info.h"

#include "bootloader_info.h"

uint32_t bootloader_info_read(rig_firmware_info_t * p_info)
{
    const uint8_t * bl_info = (uint8_t*)BOOTLOADER_INFO_ADDR;
    uint32_t err = NRF_SUCCESS;

    if(!p_info)
    {
        return NRF_ERROR_INVALID_PARAM;
    }

    memcpy(p_info, bl_info, sizeof(rig_firmware_info_t));

    if(MAGIC_HEADER != p_info->magic_number_a ||
       MAGIC_FOOTER != p_info->magic_number_b)
    {
        err = NRF_ERROR_INVALID_DATA;
    }

    return err;
}
```

15. Starting the Bootloader from your Application

When installed, the Rigado Secure bootloader runs every time the device powers on or resets. This allows for recovery of devices with a failed application update and to ensure the application can invoke the bootloader over BLE or the UART.

15.1 Bootloader Startup and Timeout

When the bootloader starts, it first checks the bootloader settings to determine if a valid application exists in the application storage bank. If a valid application is present, then the bootloader runs for a short period of time, roughly 2 seconds, and then starts the application. If a connection is made to the bootloader before this timeout, then, the bootloader will continue running.

If no application is available when the bootloader starts, then the bootloader will run indefinitely.

15.1.1 Bootloader Startup Options

In RigDFU version 3.1+, the retained register of the nRF5x parts can be used to instruct RigDFU to run for a longer period of time, 3 minutes, or to run the application immediately. The options are:

- 0xB1 - Run the bootloader for 3 minutes
- 0xC5 - Immediately restart the application

15.1.2 Other Bootloader Timeouts

The bootloader has activity timeouts built-in. These timeouts help with cases where a disruption to the BLE connection occurs during a firmware update.

OTA Update Start Timeout (10 seconds)

Upon connection to the bootloader, the device performing the update has 10 seconds to start the update. If no command is received during this time, then the bootloader will reset. If the BLE connection is active, the bootloader will first disconnect.

OTA Command Timeout (15 seconds)

Once an OTA update has started, the bootloader will timeout if no commands are received for 15 seconds.

15.2 Starting the Bootloader Over BLE

To start the bootloader over BLE, the device must have at least one writable characteristic. The device can be set up to accept any arbitrary command to start the bootloader. Preferably, the command should be more than one byte long, but this is not necessary.

15.2.1 Example: Reset and Run RigDFU for 2 seconds

```
#define BOOTLOADER_COMMAND 01020304 //example command
//Rigado suggests using a long 128-bit reset key
#define DISCONNET_REASON BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION
static void bootloader_start(uint16_t conn_handle)
{
   uint32_t err_code;

   /* Force disconnect, disable softdevice, and then reset */
   sd_ble_gap_disconnect( conn_handle, DISCONNET_REASON );

   sd_softdevice_disable();

   nrf_delay_us(500 * 1000);

   //reset system to start the bootloader
   NVIC_SystemReset();
}

//in characteristic write handler
{
   if(length == 4)
   {
       uint32_t command = (data[0] +
                           (data[1] << 8) +
                           (data[2] << 16) +
                           (data[3] << 24));
       if(command == BOOTLOADER_RESET_COMMAND)
       {
           bootloader_start(p_beacon_config->conn_handle);
       }
   }
}
```

15.2.2 Example: Reset and Run RigDFU for 3 minutes

```
#define BOOTLOADER_COMMAND 01020304 //example command
#define BOOTLOADER_DFU_START 0xB1
//Rigado suggests using a long 128-bit reset key
#define DISCONNET_REASON BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION
static void bootloader_start(uint16_t conn_handle)
{
   uint32_t err_code;

   /* Force disconnect, disable softdevice, and then reset */
   sd_ble_gap_disconnect( conn_handle, DISCONNET_REASON );
   //The below requires at least bootloader 3.1
   err_code = sd_power_gpregret_set(BOOTLOADER_DFU_START);
   APP_ERROR_CHECK(err_code);

   sd_softdevice_disable();

   nrf_delay_us(500 * 1000);

   //reset system to start the bootloader
   NVIC_SystemReset();
}
```

```
//in characteristic write handler
{
   if(length == 4)
   {
       uint32_t command = (data[0] +
                           (data[1] << 8) +
                           (data[2] << 16) +
                           (data[3] << 24));
       if(command == BOOTLOADER_RESET_COMMAND)
       {
           bootloader_start(p_beacon_config->conn_handle);
       }
   }
}
```

15.2.3 Example: Reset application and skip bootloader

```
#define DEVICE_RESET_COMMAND 11223344 //example command
#define DFU_RESET_APP 0xC5
//Rigado suggests using a long 128-bit reset key
#define DISCONNET_REASON BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION
static void device_reset(uint16_t conn_handle)
{
   uint32_t err_code;

   /* Force disconnect, disable softdevice, and then reset */
   sd_ble_gap_disconnect( conn_handle, DISCONNET_REASON );
   //The below requires at least bootloader 3.1
   err_code = sd_power_gpregret_set(DFU_RESET_APP);
   APP_ERROR_CHECK(err_code);

   sd_softdevice_disable();

   nrf_delay_us(500 * 1000);

   //reset system to start the bootloader
   NVIC_SystemReset();
}

//in characteristic write handler
{
   if(length == 4)
   {
       uint32_t command = (data[0] +
                           (data[1] << 8) +
                           (data[2] << 16) +
                           (data[3] << 24));
       if(command == DEVICE_RESET_COMMAND)
       {
           device_reset(p_beacon_config->conn_handle);
       }
   }
}
```

16. Appendix A - Serial Protocol

This section describes the serial protocol for interfacing to RigDFU via a connected UART.

16.1 UART Configuration

The RigDFU serial port is configured as follows:

- RX pin: 9
- TX pin: 10
- RTS pin: 3
- CTS pin: 2
- Settings: 115200 Baud, 8 bits, No Parity, 1 Stop bit

16.2 Serial Bootloader Activation

When RigDFU is started, the serial bootloader is not enabled by default. To activate the serial bootloader, the 'serial bootloader activation message' must be sent via a connected UART.

The activation message is a 32-bit number constructed as an array of bytes. The bytes are as follows:

Serial Bootloader Activation Message: { 0xCA, 0x9D, 0xC6, 0xA4}

If RigDFU received the 'serial bootloader activation message' before it's command timeout (2000 ms), the serial bootloader will start and relay with the following identification string:

RigDFU Serial Identification String: RigDFU/V.V.V (VV)/AA:AA:AA:AA:AA:AA\r\n

- 'V.V.V (RR)' = version string, (e.g. "3.2.0 (42)")
- 'AA:AA:AA:AA:AA:AA' = MAC address string read from nRF UICR (e.g. 95:54:93:00:A5:7D)

Once the 'RigDFU Serial Identification String' has been received, the serial bootloader is active and ready to accept serial DFU commands.

16.3 Serial Frames

All UART data sent to and received from the Serial Bootloader must be packed in a 'Serial Frame'. The 'Serial Frame' is defined as follows:

- Byte 0: Frame Start Marker (0xAA)
- Byte 1: Frame Length: Total Length including length byte but excluding the Frame Start Marker
- Byte 2: Frame OpCode
- Bytes 3 - 255: N-bytes Frame Data, 0-253 bytes, not all frames require data

16.3.1 Serial Frame Data Escaping

Since the byte '0xAA' is used to mark the beginning of a frame, any 0xAA data bytes, including the length, must be escaped. Data bytes with a value of '0xAA' or '0xAB' are sent in a special 2 byte manner:

- 0xAA becomes 0xAB 0xAC
- 0xAB becomes 0xAB 0xAB

The extra bytes from escaping are not counted as part of the total length for a serial frame. For example:
Suppose opcode 0xAA is to be sent (0xAA is not a valid opcode, but is used here for illustrative purposes):

- Byte 0: Frame Start Marker (0xAA)
- Byte 1: Frame Length (0x04)
- Byte 2: Frame OpCode (0xAA)
- Bytes 3+: 2 bytes Frame Data (0xAB, 0xFF)

The above 'Serial Frame' is transmitted as follows over the UART:
{ 0xAA, 0x04, 0xAB, 0xAC, 0xAB, 0xAB, 0xFF }

16.4 Timeouts

RigDFU has 3 different timeouts. When the timeout is reached, RigDFU will reset. The timeouts are as follows:

- 2s/120s timeout from RigDFU start after a system reset to receiving the 'Serial Bootloader Activation Message'. If no valid application is found for RigDFU to start, then the 120 second timeout will apply. Otherwise, the 2 second timeout applies
- 15 second timeout from activating the serial bootloader to receiving a DFU OpCode
- 10 second timeout between subsequent DFU packet transmissions

16.5 DFU OpCode Definitions

16.5.1 Host -> RigDFU OpCodes

- 0x01: Start DFU
- 0x02: Initialize DFU
- 0x03: Send DFU Image Data
- 0x04: Validate DFU Image Data
- 0x05: Activate and Reset
- 0x06: Reset Device
- 0x09: Configure DFU

16.5.2 RigDFU -> Host OpCodes

- 0x10: DFU Response Data

16.6 DFU OpCode Details

16.6.1 DFU Response Data (0x10)

The DFU Response Data OpCode is used to communicate the status of an operation received. Only RigDFU should send this OpCode to the host. For each OpCode received from the host, RigDFU will acknowledge and notify the host with a DFU Response Data OpCode.

Required Frame Data: 1 byte

- Data Byte 0: Error Code (see below)

16.6.1.1 DFU Response Data Error Codes

- 0x01: Success - Operation completed with no errors
- 0x02: Invalid State Error - Invalid state for the received OpCode
- 0x03: OpCode Not Supported Error - Unsupported parameters for received OpCode (typically an alignment issue)
- 0x04: Data Size Error - Data size exceeds limits
- 0x05: CRC Error - Data failed CRC verification
- 0x06: Operation Failed Error - Undefined error occurred
- 0x07: Success but waiting for more data - Operation completed with no errors and data transfer should continue

Note: *The Host does not need to respond to this OpCode*

16.6.2 Start DFU (0x01)

The 'Start DFU' OpCode is used to start the DFU operation, notify RigDFU of what regions will be updated, and the size of the image that will be transferred. The 'Start DFU' OpCode must be the first OpCode sent to RigDFU after activation of the Serial Bootloader.

Required Frame Data: 12 bytes

- Bytes 0 - 3: Softdevice length to be transferred (bytes, uint32_t, little endian)
- Bytes 4 - 7: Bootloader length to be transferred (bytes, uint32_t, little endian)
- Bytes 8 - 11: Application length to be transferred (bytes, uint32_t little endian)

Note about Updating the Bootloader and/or Softdevice:
The Bootloader and Application may be updated independently. However, updating the Bootloader should be performed with extreme caution. If the Softdevice is to be updated, the Bootloader MUST be updated at the same time. This ensures the integrity of the Bootloader by ensuring the programmed Bootloader is built for the correct Softdevice version. RigDFU does not validate this however. Instead, it simply enforces this restriction.

Typical DFU Response Data:

- Success - Proceed to Initialize DFU OpCode

16.6.3 Initialize DFU (0x02)

The Initialize DFU OpCode is used to initialize the cryptography unit for decryption and validation of the transferred DFU image. The Initialize DFU OpCode must be sent after a successful Start DFU OpCode, otherwise it will fail. This is true regardless of whether or not the image is encrypted.

Required Frame Data: 32 bytes

- Bytes 0 - 15: Image Crypto IV (Initialization Vector)
- Bytes 16 - 31: Image Crypto Tag

Typical DFU Response Data:

- Success - Proceed to Send DFU Image Data OpCode

16.6.4 Send DFU Image Data (0x03)

The Send DFU Image Data OpCode is used to send the firmware image data to RigDFU. The Send DFU Image Data OpCode must be sent after a successful Initialize DFU OpCode, otherwise it will fail. The total amount of frame data must be a multiple of 4. This OpCode is repeatedly sent until all image data is transferred.

Required Frame Data: 4 - 248 bytes

- Bytes 0 - N: The next N-bytes of image data (**N must be a multiple of 4, 0 - 248)**

Typical DFU Response Data:

- Success, waiting for more data (0x07) - Continue transmission of 'Send DFU Image Data' frames as more data is expected
- Success (0x01) - Proceed to Validate DFU Image Data OpCode, DFU Image Data transfer complete

16.6.5 Validate DFU Image Data (0x04)

The Validate DFU Image Data OpCode is used to validate the transferred DFU image. The OpCode must be sent after a successful 'Send DFU Image Data' session, otherwise it will fail.

Required Frame Data: 0 bytes

- N/A

Typical DFU Response Data:

- Success (0x01) - Image decrypted/validated successfully
- CRC Error (0x05) - Image failed validation, restart the DFU and try again

16.6.6 Activate and Reset (0x05)

The Activate and Reset OpCode is used to activate the new image and reset the device. After reset, the new image will execute. This OpCode must be sent only after a Validate DFU Image Data OpCode returns success, otherwise it will fail.

Required Frame Data: 0 bytes

- N/A

Typical DFU Response Data:

- Success (0x01) - Image Activated, RigDFU will now reset device
- Invalid State Error (0x02) - Invalid state for this OpCode, RigDFU will reset device

16.6.7 Reset Device (0x06)

The Reset Device OpCode will reset the RigDFU microcontroller. This is a legal OpCode at any time while the Serial Bootloader is activated. Sending this OpCode at any time is safe because RigDFU will not mark an application as valid and active until after completing the Activate and Reset state.

Required Frame Data: 0 bytes

- N/A

Typical Response Data: None since the device will reset

16.6.8 Configure DFU (0x09)

The configure DFU OpCode is separate from the other DFU operations. It is used to "provision" RigDFU. This OpCode can only be sent while no other DFU operations are in progress. The Serial Bootloader must first be activated and then the Configure DFU OpCode can be sent. Using the Configure DFU OpCode, the host can update the device's encryption key.

Note: *If an encryption key is already programmed, the data contained in Configure DFU data frame must be encrypted using the previous key. Otherwise the key update will fail.*

Required Frame Data: 92 bytes

- Bytes 0 - 92: See below section 'Building an Encrypted Configuration Packet'

Expected Response Data:

- Success (0x01) - RigDFU provisioning data updated successfully
- CRC Error (0x05) - The config packet did not validate correctly (most likely the wrong previous key was supplied)

16.6.8.1 Building an Encrypted Configuration Packet

- Build a plain text binary to encrypt using signimage. The signimage tool expected the following data in little endian format:
  - uint32_t[3] - metadata lengths (set to 0x00000030, 0x00000000, 0x00000000)
  - uint8_t[16] - crypto iv placeholder (fill with 0x00)
  - uint8_t[16] - crypto tag placeholder (fill with 0x00)
  - uint8_t[16] - current encryption key (fill with 0x00 if no key is set)

    - uint8_t[16] - new encryption key (fill with 0xFF to disable encryption)
    - uint8_t[16] - reserved, must be filled with 0x00
- Save this plain text configuration packet binary to a file such as 'config_plaintext.bin'
- Encrypt the plain text configuration packet using signimage.
    - signimage <plain_text>.bin <output>.bin <device_key>, key is LSB first and if no key is set, this parameter should NOT be supplied
    - If encryption is not currently enabled, this step should be skipped.
- The configuration packet can now be used to secure or update the security for the attached device.

17. Revision History

Date	Revision	Notes
06/11/2015	1.0	Initial Release
07/30/2015	1.1	Add Serial Bootloader section
04/05/2016	1.2	Add S130 and S132 support information, add note about pstorage configuration, other updates and changes