Rigado LLC
2601 25th ST SE Suite 200
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 BMD-200 modules, allowing them to keep their products in the field updated while protecting their valuable Intellectual Property (IP).

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
- 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 the BMD-200. Security is achieved by the encryption of the binary firmware image in a controlled environment prior, after which is can be sent to the BMD-200 using a method of the customer's choosing. Once the encrypted image is received by the BMD-200, 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 S110 SoftDevice versions 7.1.0 and 8.0.0. A bootloader binary is provided for each SoftDevice version. Additional SoftDevice support may be available upon request.

4. Revision History

Date	Revision	Author	Notes
06/11/2015	1.0	EPS	Initial Release

Table of Contents

1. INTRODUCTION 1
1.1 FEATURE LIST 1
2. SECURE BOOTLOADER 1
3. SOFTDEVICE SUPPORT 1
4. REVISION HISTORY 2
5. PIN DESCRIPTIONS 4
6. MEMORY LAYOUT 5
6.1 SOFTDEVICE S110 V7.1.0 5
6.2 SOFTDEVICE S110 V8.0.0 5
7. DEVELOPMENT SETUP 6
7.1 INSTALL REQUIRED TOOLS 6
7.1.1 Linux Setup 6
7.1.2 OS X Setup 6
7.1.3 Windows Setup 7
7.2 SECURE BOOTLOADER MODES 7
8. BOOTLOADER TOOLS 8
8.1 BOOTLOADER TOOLS FOLDER STRUCTURE 8
9. PROGRAMMING TOOLS 9
9.1 A NOTE ABOUT INVOKING PYTHON 9
9.2 MODULE MAC ADDRESS 9
9.2.1 Bootloader MAC Address 9
9.3 INSTALLATION OF THE BOOTLOADER 10
9.4 APPLICATION INSTALLATION WHEN PROGRAMMING THE BOOTLOADER 10
10. IMAGE TOOLS 11
10.1 UPDATE BINARY OVERVIEW 11
10.1.1 Generating Unsigned Application Binaries 11
10.1.2 Generating Encrypted Application Binaries 11
11. UPDATE TOOLS 12
11.1 PERFORMING AN OTA UPDATE FROM LINUX AND OS X 12
11.1.1 Unencrypted Update 12
11.1.2 Encrypted Update 12
11.1.3 Performing Serial Updates 13
11.1.4 Serial Unencrypted Update 13
11.1.5 Serial Encrypted Update 13
12. STARTING THE BOOTLOADER FROM YOUR APPLICATION 15
12.1 BOOTLOADER STARTUP AND TIMEOUT 15
12.1.1 Other Bootloader Timeouts 15
12.2 STARTING THE BOOTLOADER OVER BLE 15
12.2.1 Example code 15

5. Pin Descriptions

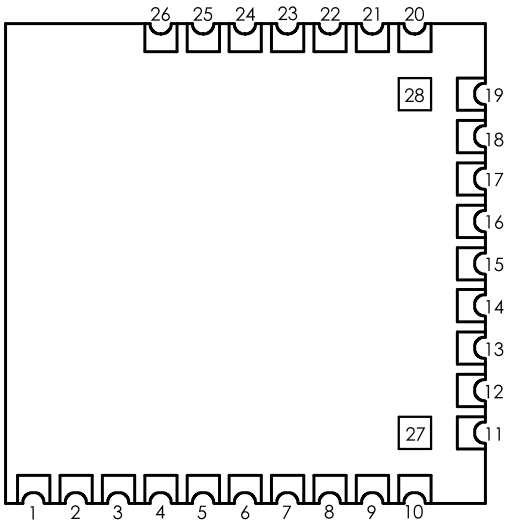

Figure 1- Pin out (Top View)

Pin description

Pin	Name	Direction	Description
21	P0.09	In	Bootloader RX
22	P0.10	Out	Bootloader TX
24	SWDIO	In/Out	SWDIO/ $\overline{\text{RESET}}$
25	SWDCLK	In	SWDCLK
5, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 20, 23	Not Used	N/A	Not Used
18	VCC	Power	+1.8V to +3.6V[1]
1, 2, 3, 4, 7, 10, 19, 26, (27, 28 opt.)	GND	Power	Electrical Ground

Table 1 – Pin Descriptions

Note 1: An external capacitor for V_{CC} is not strictly required, however using a 1µF - 4.7µF ceramic capacitor is recommended.

6. Memory Layout

The memory layout of the bootloader is as follows:

6.1 SoftDevice S110 v7.1.0

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)

6.2 SoftDevice S110 v8.0.0

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*

7. Development Setup

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

7.1 Install Required Tools

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

7.1.1 Linux Setup

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. Install Python 3.x:
    a. `sudo apt-get install python3`
    b. `python3 -m pip install pySerial`
2. Install Segger JLink tools:
    a. https://www.segger.com/jlink-software.html
    b. **Tip:** Download the TGZ archive
    c. Copy JLinkEXE to <sd version>/programming
    d. Copy liglinkarm.so.4.xx.x to <sd version>/programming and rename to libjlinkarm.so.4
3. Install the following tools for OTA Updates:
    a. `sudo apt-get install nodejs node-gyp npm bluetooth bluez-utils libbluetooth-dev`
4. On Ubuntu 12.04, a new Node.js may be needed:
    a. `sudo apt-get install python-software-properties`
    b. `sudo add-apt-repository ppa:chris-lea/node.js`
    c. `sudo apt-get install nodejs`
5. Install Crypto++ library
    a. `sudo apt-get install libcrypto++`
6. Attach a JLink programmer or Rigado BMD-200 Evaluation board to the machine

7.1.2 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-200 Evaluation board to the machine

7.1.3 Windows Setup

Note: The use of OTA Update tools is not currently supported on Windows due to the lack of built-in Bluetooth 4.0 support. Future releases may include this ability.

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
    b. Add the path to JLink.exe to your system path (Ex: C:\Program Files (x86)\SEGGER\JLink_V500a)
3. Attach JLink programmer or Rigado BMD-200 Evaluation board to the machine

7.2 Secure Bootloader Modes

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

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 image is sent in the clear such that a device sniffing Bluetooth packets or capturing the UART data could acquire the firmware during transfer. This mode allows anyone to update the unit and is typically used for development.

The secure bootloader 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 image is not decrypted until the entire image transfer is complete. Only signed images can be loaded once encryption is enabled. Readback protection is enabled on the BMD-200 to prevent an SWD programmer from reading flash memory, however the device can still be erased using a full-chip erase.

8. 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 S110 version, and if requested, an application binary. A MAC address and/or encryption key can also be loaded.

Binary Generation Tools - Python scripts and software which generate 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.*

8.1 Bootloader Tools Folder Structure

The folders for the bootloader tools are organized into two trees of the same structure. One for Nordic SoftDevice S110 v7.1.0 (sd71) and one for SoftDevice S110 v8.0.0 (sd8).

- programming
  - binaries - Contains binaries for bootloader, SoftDevice, and other required binaries
  - jlink_scripts - Contains scripts for interacting with the Segger JLink
  - 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 instead of via Bluetooth

9. Programming Tools

The programming tools are used to install the bootloader to BMD-200 modules. Additionally, the programming tools can install an application binary.

9.1 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. To check which version of python is invoked, use python --version. 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.

9.2 Module MAC Address

At the factory, Rigado programs all modules with a unique MAC address. The nRF series parts do not have permanent storage for an assigned MAC address. Rigado has chosen to store the MAC address for its BMD-200 module in the UICR. However, the UICR is erased anytime the debugging interface on the nRF is used to perform a full chip erase (such as programming the bootloader). In order to preserve the Rigado MAC address, the program.py -sm option can be used. 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 -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 in the FICR will be used instead.

For more information regarding the UICR and FICR, refer to the nRF 51xxx Series Reference Manual at www.nordicsemi.com.

9.2.1 Bootloader MAC Address

The bootloader MAC address is slight 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 system 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.

9.2.1.1 Inversion Formula

To determine the true MAC address of the bootloader, invert all of the bits of the true MAC address and then set the Most Significant byte to C0.

For example:
Original MAC - 94549300A57D
Bootloader MAC - C0ABC6FFA82

9.3 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 necessary. In addition, the main application binary can also be installed at the same time. The following sections describe how to install the bootloader for the various scenarios.

9.3.1.1 Unsecure Installation

To install the bootloader without security:

```
python program.py -sm
```

9.3.1.2 Secure Installation

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

```
python program.py -sm -k <16 hex pair key>
```

Examples:

```
python program.py -sm -k 00112233445566778899aabbccddeeff
python program.py -sm -k d6460dd794904bca992ada885310cfa1
```

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

9.4 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 must be generated from the build tools used to build the application. For example, if using the Keil toolchain from ARM, the fromelf.exe tool can be used to output a binary from the AXF file produced during build:

```
fromelf.exe --bin --output <binary_output_file> <path_to_axf_file>
```

Steps:

1. Copy application in binary for to folder containing program.py
2. Rename to application.bin
3. Add the -a option to any of the above programming script recipes
    a. Examples
        1. `python3 program.py -sm -a`
        2. `python3 program.py -sm -a -k 00112233445566778899aabbccddeeff`

10. 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.

10.1 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. Generate encrypted update binary (only if using encryption)
3. Perform update via OTA dfu.js script, Serial dfu.py script, or via a mobile device

10.1.1 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 or GCC 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 can be used to perform a firmware update to an unsecured bootloader.

10.1.2 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 ues 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>
```

11. 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.

11.1 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].

11.1.1 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:

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

Example:

```
sudo 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.

11.1.2 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.*

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.

11.1.3 Performing Serial Updates

Rigado provides a Python script for updating the device application firmware via a connected serial port. The script can be found in <sd version>/update-tools/serial/dfu.py. 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.

11.1.3.1 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.

11.1.4 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.bin
python dfu.py -s COM26 -i blinky.bin
```

11.1.5 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_secure.bin
python dfu.py -s COM26 -i blinky_secure.bin
```

12. 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.

12.1 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.

12.1.1 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.

12.2 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.

12.2.1 Example code

```
#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 );

   nrf_delay_us(500 * 1000);

   sd_softdevice_disable();

   //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);
       }
   }
}
```