

# Secure Bootloader Plus And FLASH File Guardian

ST Microelectronics STM32 Edition

V2.0.0

November 02, 2020

## 1.0 Overview

The Secure Bootloader exists as 2 separate pieces that work together providing security for your intellectual property (IP) and your embedded system products. These 2 parts are the Secure Bootloader Plus that is programmed into your product's FLASH memory and FLASH File Guardian, a Windows GUI that encrypts your application binary. The Secure Bootloader and FLASH File Guardian work as a team protecting your company's IP from theft and protecting your product from being programmed with unauthorized executable code. This 2 pronged approach provides a level of security a step above other bootloader products.

The device application (your device application code) is built and output to a binary file. Then FLASH File Guardian is used to encrypt the binary locking away its contents from outside tampering or theft. This secured file (.ffg) is then safe to distribute via email, FTP or any other means you choose to your customers and technical service providers for updating your product devices in the field.

MCU Flasher is used to perform the actual update. MCU Flasher will communicate with Secure Bootloader pre-installed on your product's board to erase any previous application code and to then program your device with the new application (.ffg file). All decryption is performed inside the MCU prior to being programmed into FLASH. MCU Flasher does not perform the decryption on the PC. This ensures the security of your IP. Another added benefit of MCU Flasher is that upon connection MCU Flasher will request and display data from the current application. These data items include the board's serial number, the date it was commissioned with its number, the current firmware part number, the firmware version, the firmware version date and the date the device was updated with that version. It will also set the device's RTC if equipped to the date and time retrieved from the PC's OS saving the operator the time of entering date and time manually.

## 2.0 High level steps for building your application for use with Secure Bootloader plus

Specific instructions for these steps are found later in this manual

1. Build the Secure Bootloader Plus project and program your board with it
2. Prepare your application for use with Secure Bootloader Plus
  - a. Copy the folder "SBLp" from the example application provided with the bootloader into your application's project folder
  - b. Include the .c and .s files found in SBLp into your project build
  - c. Add the folder SBLp to your project's include paths
  - d. Use the supplied linker script file (.ld) supplied in SBLp or edit your existing .ld file using the device specific instructions
  - e. Use your tool's project settings to output a binary file. The elf file will still get created
  - f. If and ONLY if you are using a Cortex M0 device. Copy the code found in file "M0 Code.txt" and paste it into your project's main function inside the first **USER CODE** section under the function main's opening curly brace. This code is also found in main.c of the supplied example application for all Cortex M0 devices.
3. Build your application.
  - a. Your application may now be loaded to your board and debugged with your debugging tools or you may use FLASH File Guardian to secure your app then use MCU Flasher to upload the resulting .ffg file to your board

### 3.0 Building an application for use in a system with Secure Bootloader Plus

Normally a Cortex M application is linked with its vector table starting at the default vector table location for the intended micro and the .text code section following very close behind. Secure Bootloader Plus now occupies this position so that the bootloader always first gains control of the processor coming out of reset. Because the bootloader is never erased or re-FLASHed the board is assured to always boot with known good code (Secure Bootloader Plus) preventing your system from being “bricked” in the field. Even if the new application crashes the board may be reset to bring up Secure Bootloader Plus! Your application needs to be located in FLASH memory above the bootloader and the linker script is where we make that happen.

#### 3.1 The linker script and ID Data Block in your application

Driven 2 Design supplies an example application for use with the Secure Bootloader product. In the project folder for this example application (SBLptests) is a folder named “SBLp”. This folder must be included in the build of your application for use with Secure Bootloader and contains the following, A linker script file, a .s file that creates the ID Data Block, and a SblUtility.c/.h file pair that provides functions to your application code for reading the data items from the ID Data Block if required.

1. Add the .s and .c files from folder “SBLp” to your application build from inside your firmware tools and include Sblutility.h in your c files where you wish to use the SblUtility.c functions. The .s file creates the ID Data Block that will reside in your application and edits to your linker script will cause the ID Data Block to be created and linked in.
2. Edit the linker script file (.ld file) for your project reassigning the addresses for the memory regions. Use the Driven 2 Design supplied device specific linker script instructions provided for your specific device and tool set. Below are examples of linker scripts taken from a typical project (*figure 1*) and a project being built for use with Secure Bootloader Plus (*figure 2*). For this example both projects target the STM32L496xx using ST Micro’s CubeIDE development tools.

```

/* Specify the memory areas */
MEMORY
{
RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 320K
FLASH (rx)     : ORIGIN = 0x80000000, LENGTH = 1024K
}

/* Define output sections */
SECTIONS
{
/* The startup code goes first into FLASH */
.isr_vector :
{
. = ALIGN(4);
KEEP(*(.isr_vector)) /* Startup code */
. = ALIGN(4);
} >FLASH

/* The program code and other data goes into FLASH */
.text :
{
. = ALIGN(4);
*(.text)          /* .text sections (code) */
*(.text*)         /* .text* sections (code) */
*(.glue_7)        /* glue arm to thumb code */
*(.glue_7t)       /* glue thumb to arm code */
*(.eh_frame)

KEEP (*(.init))
KEEP (*(.fini))

. = ALIGN(4);
_etext = .;      /* define a global symbols at end of code */
} >FLASH

/* Constant data goes into FLASH */
.rodata :
{
. = ALIGN(4);
*(.rodata)       /* .rodata sections (constants, strings, etc.) */
*(.rodata*)      /* .rodata* sections (constants, strings, etc.) */
. = ALIGN(4);
} >FLASH

```

Figure 1 - Memory region definitions and section assignments for a typical application build

```

/* Memories location and size for a 1MB FLASH device (0x100000) */
MEMORY
{
    RAM      (xrw)  : ORIGIN = 0x20000000, LENGTH = 320K
    VECTAB   (rx)   : ORIGIN = 0x8006800,  LENGTH = 0x1B0
    ID_DATA  (rx)   : ORIGIN = 0x80069B0,  LENGTH = 0x60
    FLASH    (rx)   : ORIGIN = 0x8006A10,  LENGTH = 0xF95F0
}

/* Place sections */
SECTIONS
{
    /* Place the startup code into "FLASH" Rom type memory */
    .isr_vector :
    {
        . = ALIGN(4);
        KEEP(*(.isr_vector)) /* Vector Table */
        . = ALIGN(4);
    } >VECTAB

    /* Place the ID Data block into "FLASH" Rom type memory */
    .iddata :
    {
        KEEP(*(.iddata)) /* ID Data Block */
    } >ID_DATA

    /* Place the startup and program code with other data into "FLASH" Rom type memory */
    .text :
    {
        . = ALIGN(4);
        *(.text)           /* .text sections (code) */
        *(.text*)          /* .text* sections (code) */
        *(.glue_7)         /* glue arm to thumb code */
        *(.glue_7t)        /* glue thumb to arm code */
        *(.eh_frame)

        KEEP (*(.init))
        KEEP (*(.fini))

        . = ALIGN(4);
        _etext = .;        /* define a global symbols at end of code */
    } >FLASH

    /* Constant data into "FLASH" Rom type memory */
    .rodata :
    {
        . = ALIGN(4);
        *(.rodata)         /* .rodata sections (constants, strings, etc.) */
        *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
        . = ALIGN(4);
    } >FLASH
}

```

Figure 2 - Memory region definitions and section assignments for an application build intended for use with Secure Bootloader Plus

A typical CubeIDE project (*figure 1*) defines just 2 memory regions, RAM and FLASH. In these examples the FLASH region is the one we are concerned with. In the typical example just below the region definitions come the section assignments. The section assignments tell the linker where to put each build section and in the typical example there are several such sections that are placed in FLASH. The 2 we care about are the `.isr_vector` and the `.text` sections. These sections are placed into FLASH memory in the order they appear in the linker file.

The first section to be placed is the interrupt vector table (.isr\_vector). This section is defined and coded in the startup.s file for the project. In this example's case the file is *startup\_stm32l496xx.s* created by the tools. The next section to be placed is the .text section. The .text section is the entirety of your application code and also includes what is called "startup" code that is defined in the startup.s file.

In the second example (*figure 2*) are the region definitions and section assignments for an application build intended for use with Secure Bootloader Plus (SBLp). Note that there are 2 additional regions defined and also 2 additional section assignments that both belong in FLASH memory. Also notice that the ORIGIN value for the first FLASH section (VECTAB) is not 0x08000000. This is because the Bootloader occupies that space so your application must be located higher. The L496's FLASH memory is erasable in 2 KB pages so your application is located on the first 2 KB boundary past the end of the bootloader code. This allows for erasing the application while leaving the bootloader intact. These additions allow for correctly locating your app and provide for the inclusion of the 96 (0x60) byte ID Data Block that must be included in your application build.

The memory region VECTAB was added to separate the startup code from the vector table. In a typical build this is not necessary because the startup code is added on top of the vector table but in a build for SBLp it becomes necessary. The region ID\_DATA was added to provide a region in FLASH memory for the ID Data Block. The ID data Block section of the example given is defined in the file *ID\_DataBlock\_L496x.s* and the definition does an initial fill of this section with the value 0xff for all bytes in the section. This is the default blank state of FLASH memory. The GUI used to secure your application binary, Flash File Guardian, checks for this section. If it is not there your file will be rejected and not used.

As in the typical example the first section to be placed into FLASH is the .isr\_vector section. This is the application's location of the interrupt vector table. The next section to be placed is the .idata section that is defined in the included .s file *"/SBLp/ ID\_DataBlock\_L496x.s"*. After this placement follows the .text section and all others. From this you can see that the ID Data Block is placed just above the interrupt vector table and after it the application code section.

### 3.2 A look at the binary output

On the next page is an example (*figure 3*) of an application binary that was built per the typical case to occupy memory starting at the default location of 0x08000000. Note here that the addresses shown in the hex editor are absolute file offsets so the first address is shown as zero. When burned into FLASH memory zero becomes 0x08000000, the FLASH region address. This is also true of Figure 4 except the ORIGIN value becomes 0x08006800 not 0x08000000. This is because the bootloader occupies the space of 0x08000000 to 0x080067FF.

As can be seen, the area from 0x00000000 through 0x000001CF are filled with ISR vectors and beginning at 0x000001B0 is the .text section (application code). This is not the picture you want in a binary intended for use with Secure Bootloader Plus.

Have a look at Figure 4 on the page after. This is how your binary should appear in the hex editor when properly built for use with SBLp. The vector table ends at 0x000001CF and the area from 0x000001B0 to 0x0000020F is filled with 0xFF. This section filled with 0xFF is the section .idata (region ID\_DATA) that is added to the linker script file. This section is defined in the file

"ID\_DataBlock\_L496x.s" supplied by Driven 2 Design and is placed in the binary output by the instructions added to the linker script file in the example (*figure 2*) above. The memory following and beginning at 0x00000210 is the .text section.

Cygnus FREE EDITION - [Shell.bin]

File Edit View Window Help

00000000 00 00 05 20 99 05 00 08-F1 04 00 08 F3 04 00 08 .....  
 00000010 F5 04 00 08 F7 04 00 08-F9 04 00 08 00 00 00 00 .....  
 00000020 00 00 00 00 00 00 00 00-00 00 00 00 FB 04 00 08 .....  
 00000030 FD 04 00 08 00 00 00 00-FF 04 00 08 01 05 00 08 .....  
 00000040 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000050 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000060 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000070 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000080 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000090 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 000000A0 E9 05 00 08 E9 05 00 08-05 05 00 08 E9 05 00 08 .....  
 000000B0 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 000000C0 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 000000D0 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 000000E0 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 000000F0 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000100 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000110 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000120 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000130 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000140 E9 05 00 08 E9 05 00 08-E9 05 00 08 11 05 00 08 .....  
 00000150 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000160 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000170 E9 05 00 08 E9 05 00 08-E9 05 00 08 00 00 00 00 .....  
 00000180 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 00000190 E9 05 00 08 E9 05 00 08-E9 05 00 08 E9 05 00 08 .....  
 000001A0 E9 05 00 08 E9 05 00 08-00 00 00 00 00 00 00 00 .....  
 000001B0 10 B5 05 4C 23 78 33 B9-04 4B 13 B1 04 48 AF F3 ...L#x3 K H..  
 000001C0 00 80 01 23 23 70 10 BD-F0 01 00 20 00 00 00 00 ...##p.....  
 000001D0 6C 59 00 08 08 B5 03 4B-1B B1 03 49 03 48 AF F3 1Y...K I H..  
 000001E0 00 80 08 BD 00 00 00 00-F4 01 00 20 6C 59 00 08 ...1Y.....  
 000001F0 03 46 13 F8 01 2B 00 2A-FB D1 18 1A 01 38 70 47 ...F...+\*...8pG  
 00000200 01 F0 FF 01 10 2A 2B DB-10 F0 07 0F 08 D0 10 F8 .....\*+.....  
 00000210 01 3B 01 3A 8B 42 2D D0-10 F0 07 0F 42 B3 F6 D1 ...B-...B..  
 00000220 F0 B4 41 EA 01 21 41 EA-01 41 22 F0 07 04 7F F0 ...A !A A".....  
 00000230 00 07 00 23 F0 E8 02 56-08 3C 85 EA 01 05 86 EA ...#...V<.....  
 00000240 01 06 85 FA 47 F5 A3 FA-87 F5 86 FA 47 F6 A5 FA ...G.....G..  
 00000250 87 F6 8E B9 EE D1 F0 BC-01 F0 FF 01 02 F0 07 02 .....  
 00000260 32 B1 10 F8 01 3B 01 3A-83 EA 01 03 13 B1 F8 D1 2.....  
 00000270 00 20 70 47 01 38 70 47-00 2D 06 BF 35 46 03 38 ...pG.8pG-...5F.8  
 00000280 07 38 15 F0 01 0F 07 D1-01 30 15 F4 80 7F 02 BF ...8.....0..  
 00000290 01 30 15 F4 C0 3F 01 3D-F0 BC 01 38 70 47 00 BF ...?..0...8pG..  
 000002A0 30 B5 BB B0 14 24 44 22-00 21 06 A8 04 F0 74 FF 0...\$D"!...t..  
 000002B0 22 46 00 21 01 A8 04 F0-6F FF 00 21 8C 22 17 A8 "F I ...l .."

Ready. Press F1 for Help. 0/5F74 0 HEX

Figure 3 - Binary output of a typical build

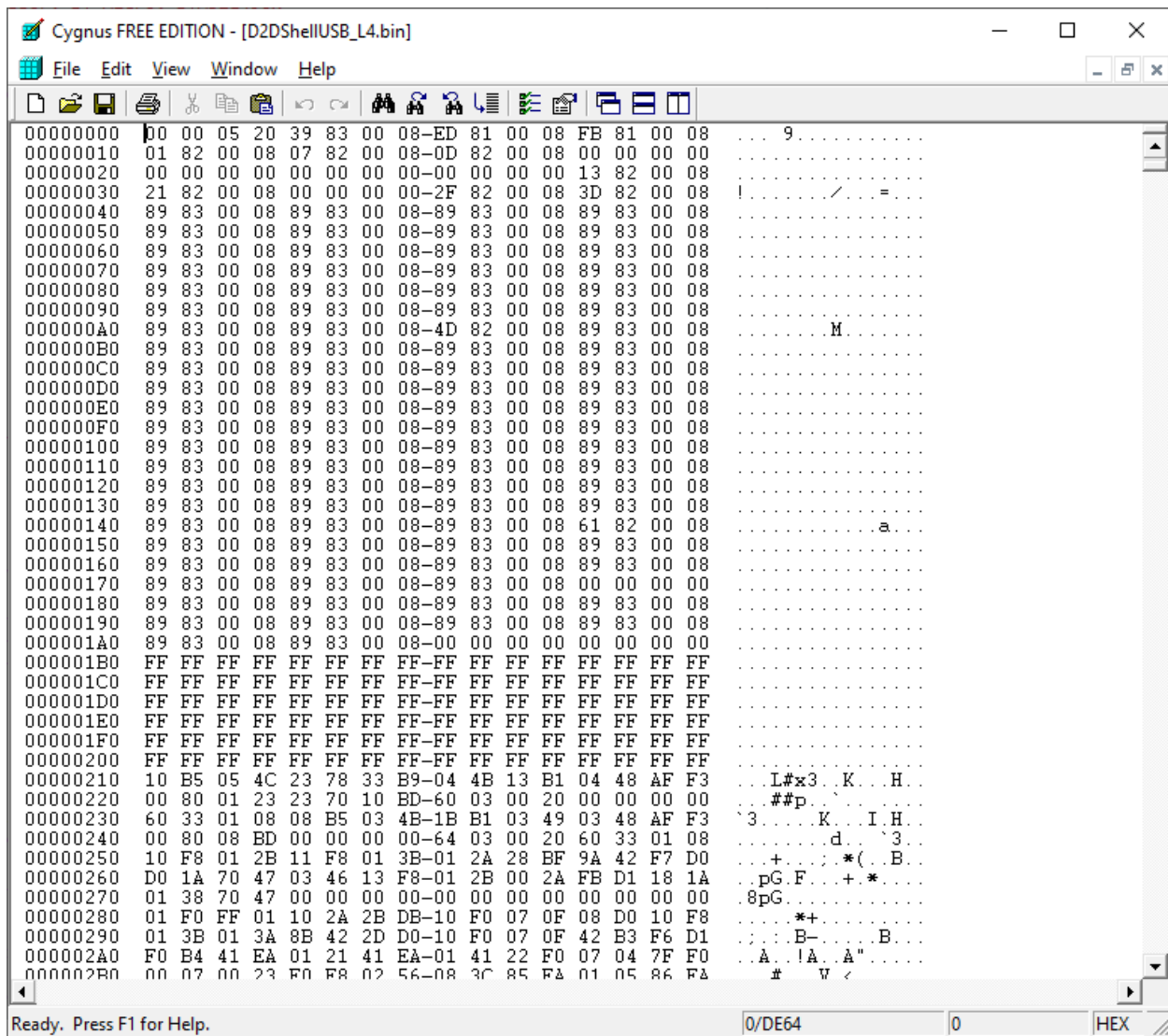


Figure 4 - Binary output of an application build intended for use with Secure Bootloader Plus

Too summarize:

1. Copy the folder “SBLp” into your project folder then add the Driven 2 Design supplied files **ID\_DataBlock\_XXXX.s** and **SblUtility.c** to your project build.
2. Configure your firmware tool to output a raw binary file as well as the standard .elf file.
3. Edit your linker script file (\*.ld) to include the added Memory regions making sure that their ORIGIN and LENGTH values are correct. See the Driven 2 Design supplied device specific manual for these values.
4. Add the code sections to their correct memory regions in the .ld file.
5. Your build binary file should appear as Figure 4 above with the ID Data Block (.iddata section) at the correct address for your specific device.



## 4.0 Securing your binary image for upload to your board through Secure Bootloader using FLASH File Guardian

Plus

Here is an image of the FLASH File Guardian program. It will be referred to throughout the following text and referenced as FFG.

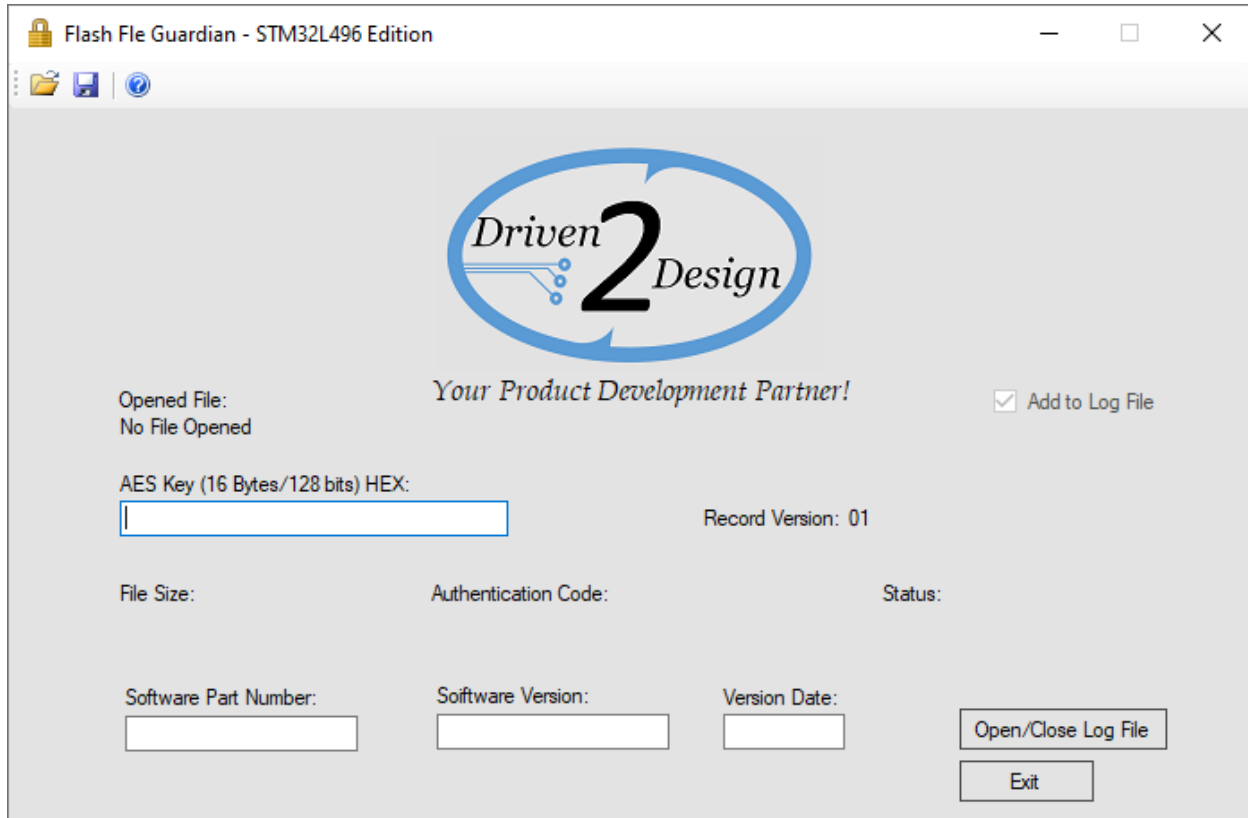


Figure 5 - The Flash File Guardian GUI

FLASH File Guardian will be referred to as FFG for the remainder of the manual. FFG will open only 2 types of files. A binary image prepared and built as explained above (.bin file) and a binary image file previously secured by FFG (.ffg file). After compiling and linking your application as described in section 2 you must now secure your binary image file using FFG. The result of FFG securing your binary image is an .ffg file.

Open FFG and then click on the “File Open” tool button (the folder icon on the far left of the tool strip). The **File Open** dialog will be displayed. Use the File Open Dialog to navigate to and open your release binary image (.bin file). Once opened, FFG will check for the .idata section between the vector table and the .text (code) section. If this area is not set to 0xFF FFG will reject and close the file. If this happens an error message will be displayed. Once you have opened a valid binary image file FFG will fill in these 3 data items on the GUI.

1. The file’s full path and name under **Opened File:**
2. The size of the file under **File Size:**
3. The file’s status under **Status:**

The Status field should now read **Raw Binary Image**.

#### 4.1 Software Part Number:

This is as the field title suggests and is provided for your company's software part number. It may contain most any character without restriction up to a maximum of 16 characters. Use what works for you and your company.

#### 4.2 Software Version:

This field may contain most any characters without restriction up to a maximum of 16 characters. It will be up to you the end user to select and adhere to a version numbering format that will work for you.

#### 4.3 Version Date:

This field is auto filled by FFG with the current system date upon opening a binary file for encryption. If the version date is not the current date it may be edited prior to saving the .ffg file and the format is strictly MM/DD/YYYY without exception. Therefore 2/6/2020 is illegal and should rather be 02/06/2020

#### 4.4 AES Key

The AES key field is a 32 character ASCII HEX representation of the 16 byte (128 bit) encryption key. For security reasons the AES Key does not become a part of the ID Data and is not embedded in the output file. This is the encryption key used by the encryption and decryption algorithms implemented by Flash File Guardian and Secure Bootloader Plus and is selected by the end user, you. The key you use must also have been built into Secure Bootloader Plus residing in your board. **IMPORTANT! Safeguard your encryption keys for the security of your IP.** It is a bit cumbersome to enter these 32 characters but once a key is entered and used for encryption you may open the log file and perform a copy and paste to enter your key for later versions of your code. The keys you use are kept in the log file. Clicking on Open/Close Log File will open the log file in notepad for you.

#### 4.5 Saving The File

After entry of the AES key, The Firmware Part Number and the Firmware Version Number the file may be saved. Clicking the save button (disk icon in the tool strip) will save the file. The Save File Dialog will open. FFG auto fills the file name with your entered Firmware part number and Firmware Version Number and appends ".ffg" to the end. You may accept this file name or edit it to your own specification but .ffg must remain the final extension if it is to be used with MCU Flasher or Secure Bootloader for USB Host and file operation.

As soon as the .ffg file is saved and closed Flash File Guardian reopens the .ffg then reads decrypts and verifies the integrity of the file and displays the status of this check under **Status:** on the GUI. This ensures the integrity of your encrypted firmware .ffg file.

These are the steps performed by FFG when the file is saved to a secure (.ffg) file.

1. The ID Data is inserted into the ID Data
  - a. Software Part Number
  - b. Software Version Number
  - c. Version Date
  - d. The file's size (same as your raw binary output)
  - e. The file name that you specified
2. A proprietary 32 bit authentication code (modified CRC) is then calculated on the entire contents of the file and is inserted into the ID Data Block.

3. The file is then encrypted and output to the .ffg file and closed.
4. The file is then reopened and decrypted. All data including the authentication calculation are verified for correctness and the status of this check is displayed.

The original binary file is not altered by this process but is only read by FFG.

The resulting .ffg file may now freely be distributed for product updates without fear of theft or reverse engineering.