STM32 Secure Patching Bootloader Quick Start Guide

By Firmware Modules Inc.

March 23, 2023

V1.4.0

Introduction

The *STM32 Secure Patching Bootloader* is a firmware and software product designed by Firmware Modules Inc for use with the popular **STM32** family of microcontrollers (MCUs) by STMicroelectronics. As its name suggests, the *STM32 Secure Patching Bootloader* is a bootloader system offering a secure way to update the firmware on an STM32 device using patch files (or regular full-image files) over a wide range of delivery modes. This bootloader system is designed to be extremely easy to integrate into an existing project: The on-target bootloader binary is pre-built for specific targets and there is nothing to configure. It is a **plug'n'play system** that just works! Unlike other bootloader and firmware update systems, you do not need to be a bootloader expert to get "production level" firmware update functionality.

Integrating the stm32-secure-patching-bootloader into an existing project is a simple **five step process**:

1. Adding bootloader files to your **project repository**.
2. Configuring **STM32CubeIDE**.
3. Adjusting your project's linker script **.ld** file.
4. Adjusting your project's **system_stm32xxxx.c** file.
5. Generating your project's encryption and signing keys.

Each step is described below. We have also posted projects you can use as a reference in our forks of the STM32 Cube repositories found at https://github.com/orgs/firmwaremodules/repositories .

Integration

Integration Step 1 of 5

Add bootloader files to your project repository.

- Create a directory called `Bootloader` in your project root.
- Copy the directories `Include`, `Linker`, `Scripts`, `Tools` into it.
- Copy the `Libs` directory, but only the board subdirectories that you are needing. E.g. copy DISCO-F769I directory into your Libs directory if you are using that board.
- As an alternative, you can add the bootloader files as a git submodule with *git submodule add https://github.com/firmwaremodules/stm32-secure-patching-bootloader Bootloader*.

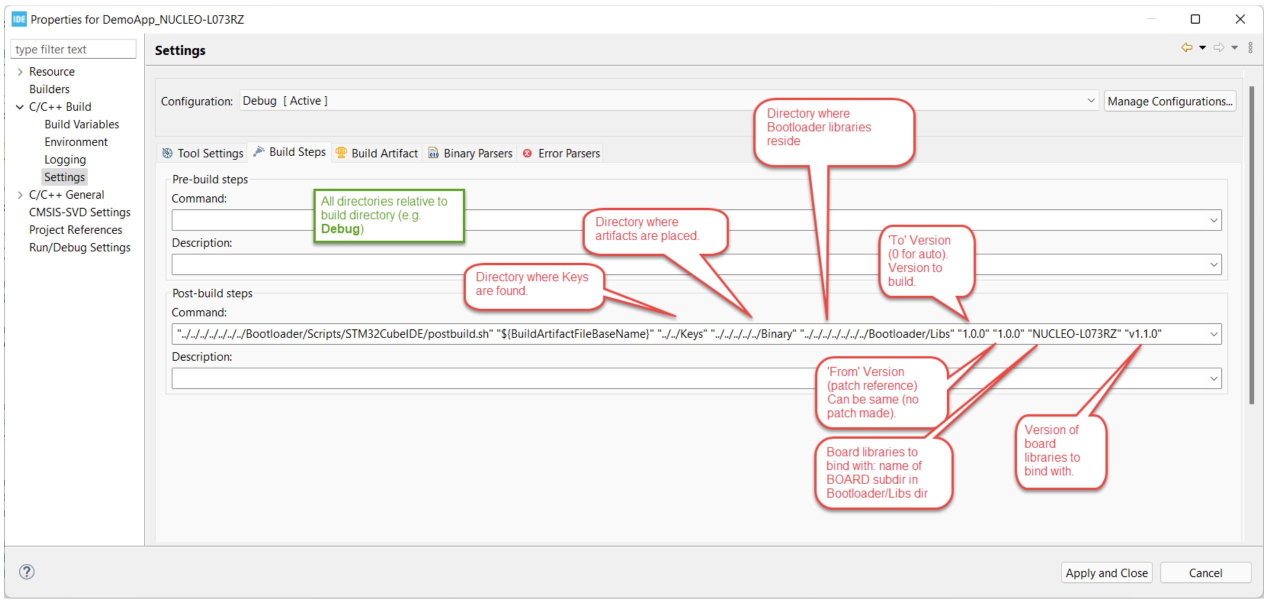
Integration Step 2 of 5

Configure STM32CubeIDE.

We need to add the **postbuild.sh script** command line, update the include and linker paths, and link with the stm32-secure-patching-bootloader application interface. **C/C++ Build -> Settings -> Build Steps**

Postbuild script command examples for some targets:

- "../../../../../../../Bootloader/Scripts/STM32CubeIDE/postbuild.sh" "${BuildArtifactFileBaseName}" "../../Keys" "../../../../../Binary" "../../../../../../../Bootloader/Libs" "0.0.0" "0.0.0" "NUCLEO-L073RZ" "v1.4.0"
  "../../../../../../../Bootloader/Scripts/STM32CubeIDE/postbuild.sh" "${BuildArtifactFileBaseName}" "../../Keys" "../../../../../Binary" "../../../../../../../Bootloader/Libs" "0.0.0" "0.0.0" "DISCO-F769I" "v1.4.0"


Properties for DemoApp_NUCLEO-L073RZ
type filter text
Resource
Builders
C/C++ Build
Build Variables
Environment
Logging
Settings
C/C++ General
CMSIS-SVD Settings
Project References
Run/Debug Settings
Settings
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
Pre-build steps
Post-build steps
Apply and Close
Cancel
All directories relative to build directory (e.g. Debug)
Directory where Bootloader libraries reside
Directory where artifacts are placed.
Directory where Keys are found.
'To' Version (0 for auto). Version to build.
'From' Version (patch reference) Can be same (no patch made).
Board libraries to bind with: name of BOARD subdir in Bootloader/Libs dir
Version of board libraries to bind with.

Configuration: Debug [Active]

Pre-build steps
Command:
Description:

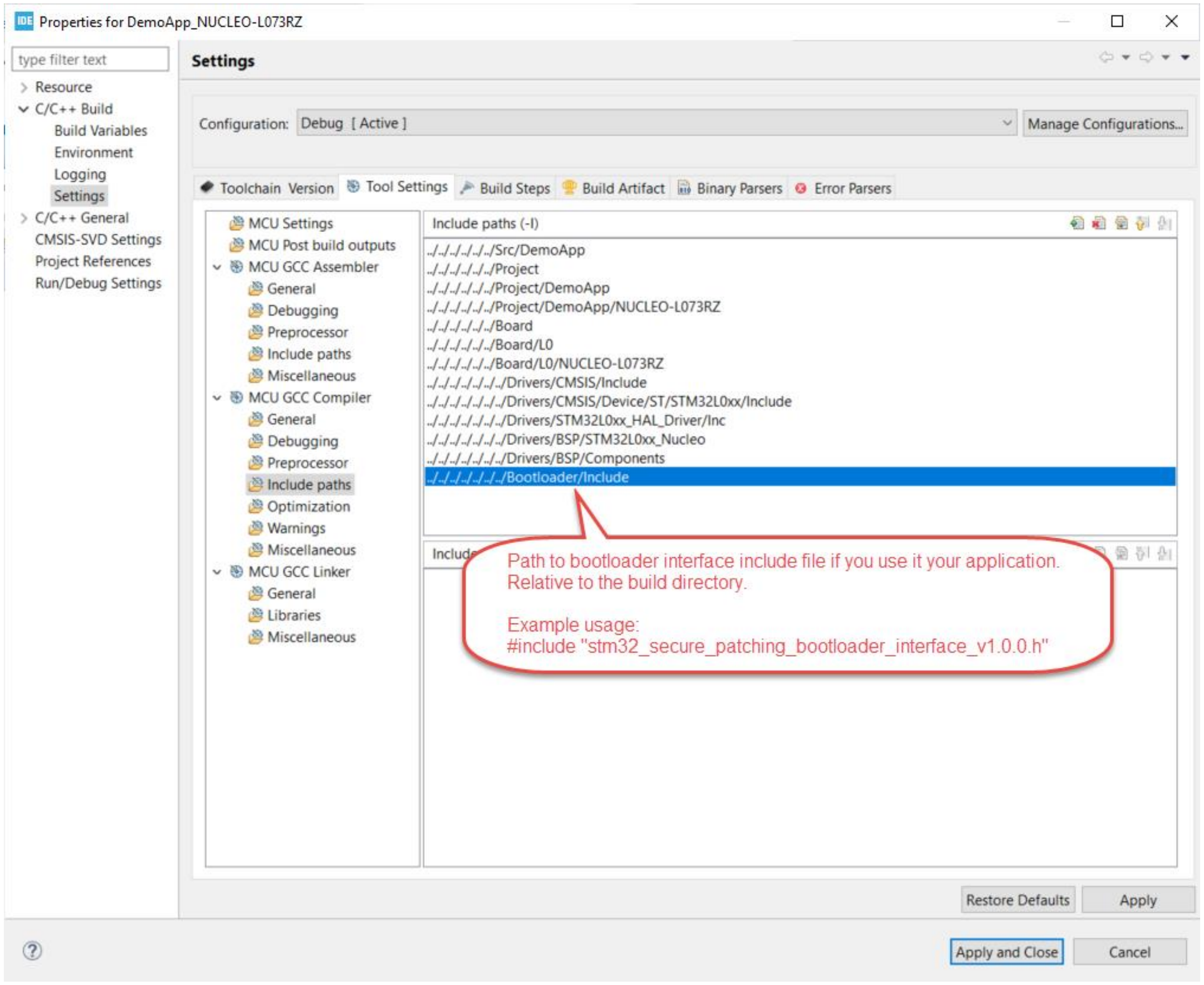
Post-build steps
Command:

```
"../../../../../../../Bootloader/Scripts/STM32CubeIDE/postbuild.sh" "${BuildArtifactFileBaseName}" "../../Keys" "../../../../../Binary" "../../../../../../../Bootloader/Libs" "1.0.0" "1.0.0" "NUCLEO-L073RZ" "v1.1.0"
```

Description:

Note: a 'to' version of "0.0.0" means to auto-generate the 3-digit semantic build version number from the git repository info generated by running 'git describe –tags –always –dirty'. Simply tag with 'git tag v1.0.0' to generate a build with version v1.0.0. Intermediate builds such as v1.0.0-3-g9c124b3 are also automatically captured as v1.0.3 (the -3- is added to the patch digit).

Add path to bootloader include: **C/C++ Build -> Settings -> MCU GCC Compiler -> Include paths : ../../../../../../../Bootloader/Include**



Configuration: Debug [Active]

Include paths (-I)

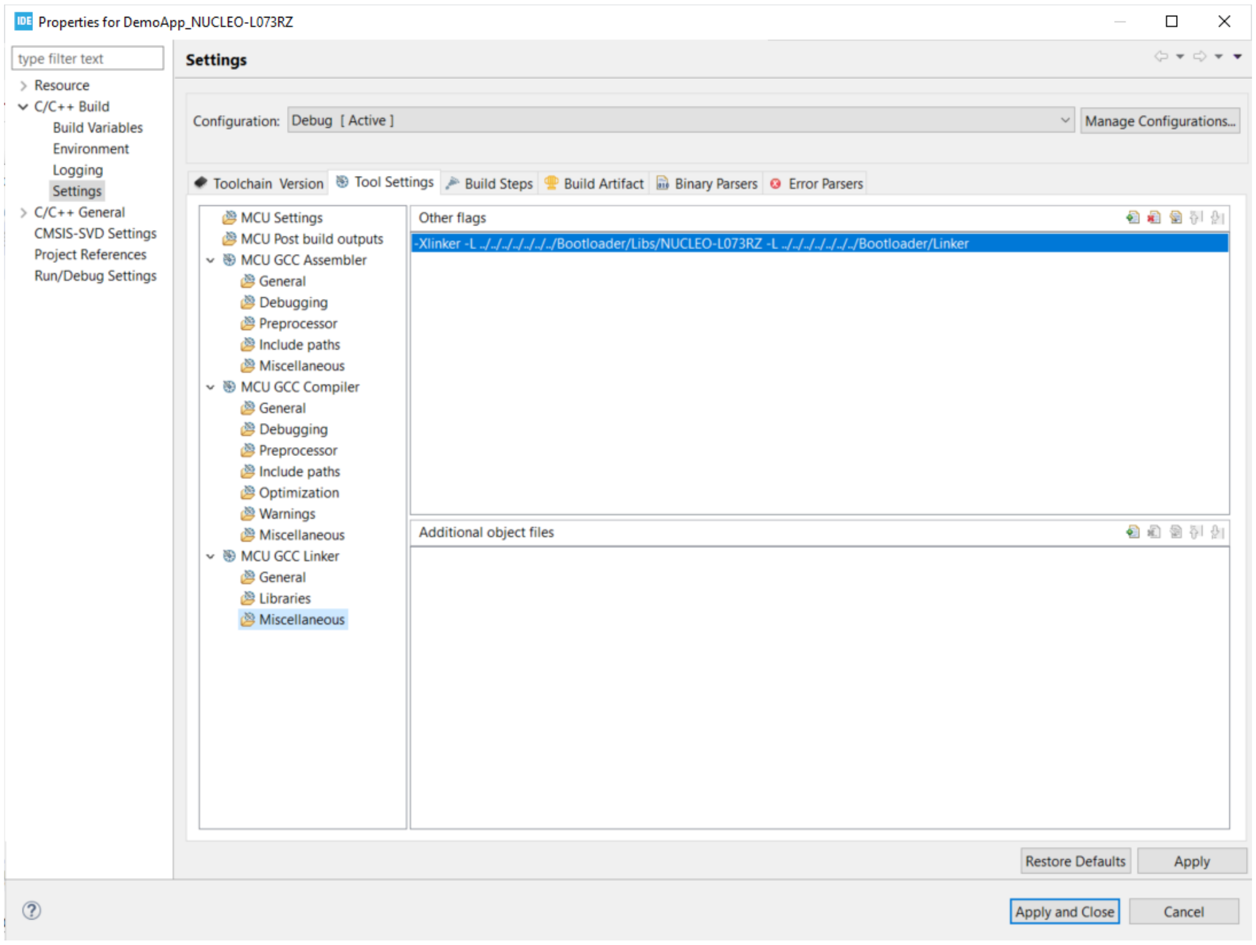
```
../../../../../../Src/DemoApp
../../../../../../Project
../../../../../../Project/DemoApp
../../../../../../Project/DemoApp/NUCLEO-L073RZ
../../../../../../Board
../../../../../../Board/L0
../../../../../../Board/L0/NUCLEO-L073RZ
../../../../../../../Drivers/CMSIS/Include
../../../../../../../Drivers/CMSIS/Device/ST/STM32L0xx/Include
../../../../../../../Drivers/STM32L0xx_HAL_Driver/Inc
../../../../../../../Drivers/BSP/STM32L0xx_Nucleo
../../../../../../../Drivers/BSP/Components
../../../../../../../Bootloader/Include
```

Path to bootloader interface include file if you use it your application.
Relative to the build directory.

Example usage:
#include "stm32_secure_patching_bootloader_interface_v1.0.0.h"

Your application uses the bootloader include file to access bootloader services like in-application updates (SE_PATCH_XXXX) or getting bootloader or application version information.

Add search paths to bootloader linker files: **C/C++ Build -> Settings -> MCU GCC Linker -> Miscellaneous** : -Xlinker -L ../../../../../../../Bootloader/Libs/NUCLEO-L073RZ -L ../../../../../../../Bootloader/Linker



Configuration: Debug [Active]

Other flags

```
-Xlinker -L ../../../../../../../Bootloader/Libs/NUCLEO-L073RZ -L ../../../../../../../Bootloader/Linker
```

The bootloader linker files provide symbols necessary to correctly locate your application in the device's flash and SRAM memory as well as to call bootloader APIs if your application does that.

Integration Step 3 of 5

Adjust your application's linker script.

At minimum we need to tell the linker where your application's new start offset is in flash and RAM. This is defined by the bootloader's linker configuration file **stm32-secure-patching-bootloader-linker-gcc_<BOARD>_<version>.ld**.

It defines some symbols that we may need, minimally it is **STM32_SECURE_PATCHING_BOOTLOADER_SLOT0_START**. Then your application's interrupt vector table has to be offset from this by the minimum size of your vector table (to account for the image header), also defined as VECTOR_SIZE in the linker configuration file.

```
_estack = 0x20050000;    /* end of RAM specific to the device. */

/* For this example end of RAM on STM32L496ZG is 0x20050000 = 320 KB*/
INCLUDE stm32-secure-patching-bootloader-linker-gcc_NUCLEO-L496ZG_v1.4.0.ld

APPLI_region_intvec_start__  = STM32_SECURE_PATCHING_BOOTLOADER_SLOT0_START + VECTOR_SIZE;
APPLI_region_ROM_start     = STM32_SECURE_PATCHING_BOOTLOADER_SLOT0_START  + VECTOR_SIZE +
VECTOR_SIZE;
APPLI_region_ROM_length    = STM32_SECURE_PATCHING_BOOTLOADER_SLOT0_END -
APPLI_region_ROM_start + 1;
APPLI_region_RAM_start     = STM32_SECURE_PATCHING_BOOTLOADER_RAM_START;
APPLI_region_RAM_length     = _estack - APPLI_region_RAM_start;

MEMORY
{
  ISR_VECTOR (rx)   : ORIGIN = APPLI_region_intvec_start__, LENGTH = VECTOR_SIZE
  APPLI_region_ROM  : ORIGIN = APPLI_region_ROM_start, LENGTH = APPLI_region_ROM_length
  APPLI_region_RAM  : ORIGIN = APPLI_region_RAM_start, LENGTH = APPLI_region_RAM_length
}
```

To your SECTIONS, add the following:

```
  /* Extra ROM section (last one) to make sure the binary size is a multiple of the AES
block size (16 bytes) */
  .align16 :
  {
    . = . + 1;         /* _edata=. is aligned on 8 bytes so could be aligned on 16
bytes: add 1 byte gap */
    . = ALIGN(16) - 1; /* increment the location counter until next 16 bytes aligned
address (-1 byte)   */
    BYTE(0);           /* allocate 1 byte (value is 0) to be a multiple of 16 bytes
*/
  } > APPLI_region_ROM
```

A multi-target generic linker script that works with most STM32 targets is provided in the bootloader **Linker** directory. See this reference project linker script.

For a TouchGFx multi-segment application (assets on external QSPI or OSPI flash), use this template:

```
/* For this example end of RAM on STM32L4R9I is 0x200A0000 = 640 KB*/
INCLUDE stm32-secure-patching-bootloader-linker-gcc_DISCO-L4R9I_v1.4.0.ld

APPLI_region_intvec_start__  = STM32_SECURE_PATCHING_BOOTLOADER_SLOT0_START + VECTOR_SIZE;
APPLI_region_ROM_start    = STM32_SECURE_PATCHING_BOOTLOADER_SLOT0_START  + VECTOR_SIZE +
VECTOR_SIZE;
APPLI_region_ROM_length   = STM32_SECURE_PATCHING_BOOTLOADER_SLOT0_END -
APPLI_region_ROM_start + 1;
APPLI_region_RAM_start    = STM32_SECURE_PATCHING_BOOTLOADER_RAM_START;
APPLI_region_RAM_length    = _estack - APPLI_region_RAM_start;

MEMORY
{
  ISR_VECTOR (rx)   : ORIGIN = APPLI_region_intvec_start__, LENGTH = VECTOR_SIZE
  APPLI_region_ROM  : ORIGIN = APPLI_region_ROM_start, LENGTH = APPLI_region_ROM_length
  APPLI_region_RAM  : ORIGIN = APPLI_region_RAM_start, LENGTH = APPLI_region_RAM_length
  QSPI (rx)         : ORIGIN = STM32_SECURE_PATCHING_BOOTLOADER_MULTISEG_START, LENGTH = 64M
}
```

To your SECTIONS, add the following:

```
  /* Extra ROM section (last one) to make sure the binary size is a multiple of the AES
block size (16 bytes) */
  .align16 :
  {
    . = . + 1;         /* _edata=. is aligned on 8 bytes so could be aligned on 16
bytes: add 1 byte gap */
    . = ALIGN(16) - 1; /* increment the location counter until next 16 bytes aligned
address (-1 byte)   */
    BYTE(0);           /* allocate 1 byte (value is 0) to be a multiple of 16 bytes
*/
  } > APPLI_region_ROM


  /* Section for placement of resources into QSPI segment
   * This is not contiguous with the "ROM" segment, but must also be aligned
   * to 16 bytes for AES requirement.
   */
  ExtFlashSection :
  {
    *(ExtFlashSection)

    . = . + 1;         /* _edata=. is aligned on 8 bytes so could be aligned on 16
bytes: add 1 byte gap */
    . = ALIGN(16) - 1; /* increment the location counter until next 16 bytes aligned
address (-1 byte)   */
    BYTE(0);           /* allocate 1 byte (value is 0) to be a multiple of 16 bytes
*/
  } >QSPI
```

Integration Step 4 of 5

Adjust your application's system_stm32xxxx.c file.

This file programs the VTOR register (Vector Table Offset Register) by your application in the early startup phase so that interrupts invoke your application's handlers at the linked offset instead of the bootloader's handlers at the start of flash (0x08000000).

system_stm32xxxxxx.c :

```
extern uint32_t APPLI_region_intvec_start__;
#define INTVECT_START ((uint32_t)& APPLI_region_intvec_start__)

void SystemInit(void)
{
   ...

   /* Configure the Vector Table location add offset address ------------------*/
   SCB->VTOR = INTVECT_START;
}
```

Integration Step 5 of 5

Generating your project's encryption and signing keys.

Use the **make_keys_v7m.bat** (or **make_keys_v6m.bat** only for **L0, F0** and **G0** targets) script under **Scripts** to call a Python tool to generate the AES encryption key (firmware confidentiality) and the ECDSA public verification and private signing keys (firmware authenticity).   Example on Windows systems to place keys in a **Keys** directory:

```
c:\stm32-secure-patching-bootloader-demoapp\Bootloader\Scripts>make_keys_v7m.bat
..\..\App\Project\DemoApp\DISCO-F769I\STM32CubeIDE\Keys

make_keys.bat : Generate new secure keys for stm32-secure-patching-bootloader

Making ..\..\App\Project\DemoApp\DISCO-F769I\STM32CubeIDE\Keys/Cipher_Key_AES_CBC.bin

Making ..\..\App\Project\DemoApp\DISCO-F769I\STM32CubeIDE\Keys/Signing_PrivKey_ECC.txt

Making ..\..\App\Project\DemoApp\DISCO-F769I\STM32CubeIDE\Keys/machine.txt
```

Run **make_keys_vXm <path to directory to contain key files>**

If you're not using Windows, then all you need to do is look inside make_keys_vXm.bat and run the Python scripts directly. The **Keys** directory is referenced by the postbuild.sh post-build command line in the IDE. This directory can be anywhere and called anything by adjusting the post-build command line.

Important Note

These keys are also used by the bootloader to differentiate between compatible firmware loads! That is, the stm32-secure-patching-bootloader will attempt to update to any firmware that is presented that can be successfully decrypted and signature verified. There is no "product ID" field in the header. *The keys are the product ID.* Of course, the point of having this type of security is to prevent loading of firmware except for those loads you explicitly authorize by way of signing during the build (automatically done by postbuild.bat). But if you are running multiple products or even incompatible hardware revisions within the same product line, you must ensure to have unique keys setup for each STM32CubeIDE build project (the path to the keys is specified in the postbuild.sh post-build command line and in these examples is called **Keys**).

Build Products

The *stm32-secure-patching-bootloader* **postbuild** process produces three or four artifacts and places them in the specified output directory (typically **Binary**).

Naming conventions

Combined binary:

- `BOOT_<Your Artifact Name>_<Board Name>_<version>.[hex | bin]`

Update binary:

- `<Your Artifact Name>_<Board Name>_<version>.[sfb | sfbp]`

Combined binaries are for loading onto the target by a programming tool such as STM32CubeProgrammer during production.

Update binaries are for distribution to customers or to your apps and cloud services for in-field updates.

1. Combined binaries: `BOOT_DemoApp_NUCLEO-L073RZ_v1.0.0.hex`, `BOOT_DemoApp_NUCLEO-L073RZ_v1.0.0.bin` (.bin produced only when MultiSegment is not enabled. In other words, TouchGFX firmware builds don't generate combined .bin files.)
2. Secured in-field firmware full update file: `DemoApp_NUCLEO-L073RZ_v1.0.0.sfb` (full image)
3. Secured in-field firmware patch update file: `DemoApp_NUCLEO-L073RZ_v1.0.0_v1.1.0.sfbp` (patch - produced only if reference version, e.g., v1.0.0 exists in Binary\PatchRef)

Note: the <version> appended to the file names is a direct copy of either the post-build command-line's 'to' version, or else of the output of 'git decribe –tags –always –dirty' if the 'to' version is in auto mode "0.0.0" or "0". When using the auto mode (recommended), use git tagging to set the version. A semantic version tag may be prepended with a 'v' or not. Firmware Modules projects like to prepend versions with a 'v', so a tag might be 'v1.3.1'. Either way works.

Also note: the firmware image header contains a 4-byte field for version and therefore cannot directly accommodate a non-semantic version such as v1.3.1-7-gc32667a that may arise during development load testing. The version generator embedded into the postbuild.sh script can safely handle this scenario by adding the "number of commits since" value to the build number, so v1.3.1-7 becomes 1.3.8 in the header.

Notes

General Notes

- Patch files .sfbp only work if the source version already exists on the target device. E.g. to update from v1.1.7 to v1.2.0, version v1.1.7 must exist on the device as the active image in SLOT0.
- To update from v1.0.0 to v1.4.0 when only patches of v1.0.0_v1.1.0, v1.1.0_v1.2.0, v1.2.0_v1.3.0, v1.3.0_v1.4.0 are available, each patch must be applied in series. For each one, firmware is installed by the bootloader through a reboot cycle.
- The build system (STM32CubeIDE) postbuild command line must explicitly set what the reference firmware version is to build a patch from. With this flexibility it is possible to build patches from any version. Commonly this 'from version' is updated from one release to the next to build a series of patchable updates (v1.0.0->v1.1.0, v1.1.0->v1.2.0 etc). Of course, you could create a patch from v1.0.0->v1.2.0 and post this to your update server; if your 'patch picker' algorithm was smart enough it could select the best patch from a list of available patches. It is important to set this 'from version' before you make your final commit and tag and build. The procedure is to set 'from version' to v1.0.0, commit, then tag v1.1.0, then do a build to create the patch file v1.0.0_v1.1.0.sfbp (assuming that v1.1.0 was already built and available in the PatchRef directory).
- Full image files .sfb can be used to update to any version and do not depend on what is currently on the device. These could serve as a fall-back in case the hypothetical 'patch picker' implemented in the user application cannot find a suitable patch for download.
- The 'patch picker' is implemented in the USB flash drive updater to select the first patch in the root directory that matches the current version on the device and contains the highest 'to' version. It is able to perform a 'patch chain' update as described above by repeatedly rebooting and updating to the next higher patch version.

Security Notes

- The stm32-secure-patching-bootloader strikes a balance between ease of integration, broad platform support and selection of security features that are enabled.
- The bootloader system is designed so that your products and firmware update files can be distributed into uncontrolled environments. This means firmware confidentiality is ensured through AES encryption, and firmware and device integrity are ensured through ECDSA (digital signatures). The AES encryption and the ECDSA *public* verification keys are stored in your device's internal flash. The AES encryption and ECDSA *private* signing key are typically stored in your firmware development repository (e.g. on GitHub, your developer's and build PCs).
- Those that would attempt to undermine your products and solutions know that these keys exist. Thus, protection of these keys is paramount. You must create a root of trust on each of the devices you deploy by way of enabling **RDP Level 2** through the stm32-secure-patching-bootloader **production** build (you get this build when you register the bootloader at firmwaremodules.com). This build will automatically check and enable RDP Level 2 on each boot to help mitigate potential **RDP regression** attack vectors (yes these do exist). Note that when RDP Level 2 is enabled, you permanently forfeit the ability to connect a debugger to your devices (a good thing when security is concerned).

- The bootloader system is *not* designed to protect your device and secrets from your own firmware application. If you don't trust your application or the way it works (e.g. you think it might offer a way for an attacker to read out contents of memory) then this bootloader system is not for you, or at least you must be aware that *internal* protections such as proprietary readout protection (PCROP) or firewalls are not implemented. Don't offer a command-line interface that has memory peek and poke commands.
- Finally, you should assume that if an attacker has physical access to your device and is determined enough (time and resources) then your secrets will eventually be extracted. You need to weigh the expense and effort someone (or some group) would go to obtain your firmware update AES encryption key and public signature verification key, and the "payoff" that may be associated with that effort.

Release Notes

V1.4.0 - Mar 2023

- Add support for new platforms and boards: G0 (NUCLEO-G0B1RE) H7 (DISCO-H745I) WL (NUCLEO-WL55JC) F4 (DISCO-F469I) L4 (NUCLEO-L476RG)
- Prints size of binaries detected in each slot in diagnostic output.
- Bootloader disables cache before launching application on all boards that use cache (prevents faulting when application tries to re-enable already enabled cache in some cases).
- Fixes YMODEM load button trigger wrong state for NUCLEO-L452RE.
- Optimization: greatly speeds up patching updates (3x or more) on large binaries utilizing external flash.
- Optimization: removes one redundant header verification in virgin device or two redundant header verifications in devices that have undergone at least one update cycle. Saves between 50 - 2000 ms per header verification of bootup time depending on MCU family.
- Ensures hardware CRC is powered up when needed during SE_PATCH_Data() API calls.
- Adds make_keys_vXm.bat scripts to automatically generate machine.txt file (often overlooked otherwise).

V1.3.0 - Nov 2022

- Now works with applications built using **STM32CubeIDE 1.9 and later** including version 1.10.1.
- Simplifies application integration process by removing the need to link with a library for access to SE_PATCH (in-application firmware update) APIs. Now, the SE_PATCH engine APIs are available to all applications by default.
- Adds new platform support for **STM32L5** and the DISCO-L562E board.
- Adds STM32L4 platform board B-L4S5I-IOT01A.

V1.2.0 - No documentation updates.

V1.1.0 - May 2022

- Adds new platform support for STM32WLE5 and LORA-E5-DEV board.
- Adds specific support for the B-L072Z-LRWAN1 board.
- Adds README to each library package to describe flash layout and bootloader configuration.
- Changes postbuild command to allow user to specify location of bootloader Libs directory.
- Removes vector offset and multiseg address parameters in the postbuild command script (these are now defined in the bootloader library package artifacts).

V1.0.0 - Dec 2021

- Initial Release