ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Wrapper for FSBL images

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Contents

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Stable: 18.06.2021 - 07:40 / Revision: 18.06.2021 - 07:40

A quality version of this page, approved on *18 June 2021*, was based off this revision.

Contents

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Article purpose

This article provides the basic information needed to start using the application tool **stm32wrapper4dbg**.

It explains how to use the tool to wrap an existing FSBL image and create a **debug FSBL image**, suitable for attaching a debugger at boot.

This article also covers the case of **closed devices** that require a signed image.

2 Introduction

The following table provides a brief description of the tool, as well as its availability depending on the software packages:

✓: this tool is either present (ready to use or to be activated), or can be integrated and activated on the software package.

✗: this tool is not present and cannot be integrated, or it is present but cannot be activated on the software package.

Tool			STM32MPU Embedded Software distribution			STM32MPU Embedded Software distribution for Android™		
Name	Category	Purpose	Starter Package	Developer Package	Distribution Package	Starter Package	Developer Package	Distribution Package
stm32wrapper4dbg	Debugging tools	The stm32wrapper4dbg tool wraps an STM32 FSBL image to enable a debugger to halt the boot at the first executable instruction of FSBL.	✗*	✓	✓**	✗	✗	✗
			** Cross compiled gdb and openocd binaries are required and only available from the Developer Package.* *** It is recommended to use the Developer Package to run the gdb debug session, which provides all dependencies.*					

Please refer to boot chain overview for a complete description of the boot process.

To debug the FSBL code, the debugger must halt the execution:

- either at the very first instruction (entry point) of the FSBL,
- or earlier in the ROM code and then proceed till the FSBL entry point.

The ROM code execution cannot be halted deterministically. In case of closed devices, it cannot be halted at all. The stm32wrapper4dbg tool enables the debugger to halt the execution at FSBL entry point.

2.1 Workaround (not recommended)

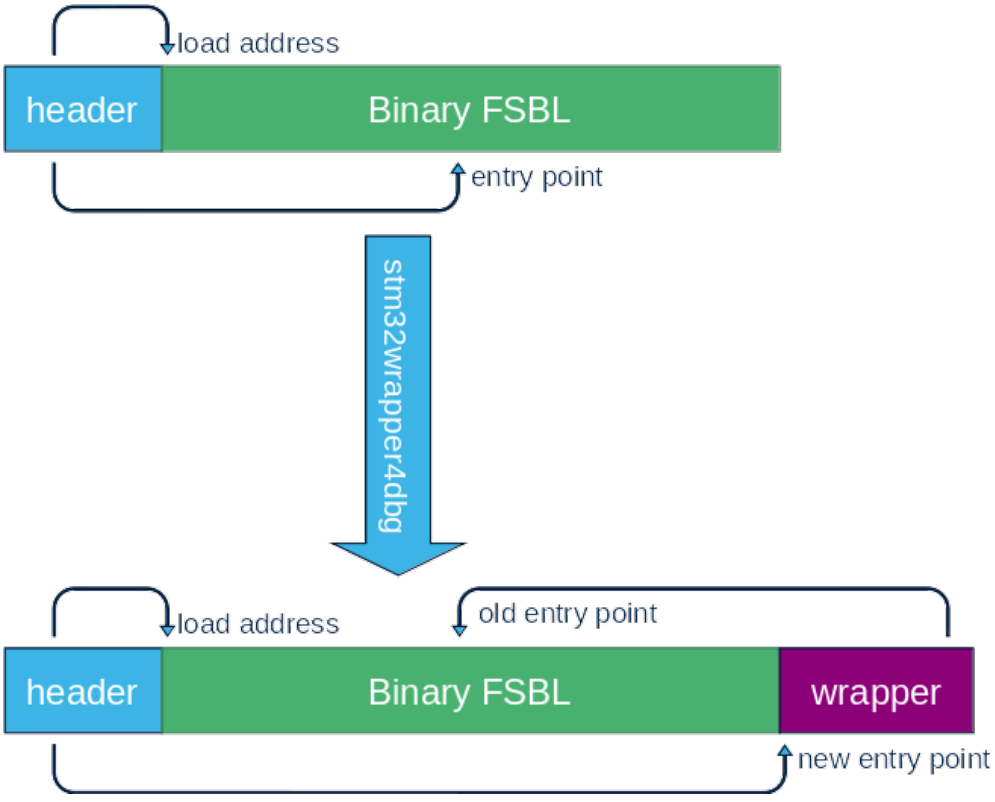
To halt the execution at FSBL entry point, a possible workaround consists in modifying the FSBL code, adding an infinite loop at the entry point. The boot process stops looping at the entry point. The debugger can then attach the device and halt the execution. The debugger then skips the infinite loop, advancing the execution to the first real FSBL instruction.

The workaround above has few drawbacks:

- By modifying and recompiling the FSBL code, the memory layout of the FSBL binary is also modified. This can prevent reproducing the error conditions that have to be debugged.
- People in charge of debugging must have knowledge of FSBL recompiling and (only in case of closed devices) signing it.
- Preexisting FSBL binary cannot be debugged.

2.2 Wrapper for FSBL

To halt the execution at FSBL entry point, STMicroelectronics proposes a less invasive method through the tool stm32wrapper4dbg. The image below describes how stm32wrapper4dbg works.



The stm32wrapper4dbg tool takes an existing FSBL image and generates a new FSBL image by appending (or prepending) a binary wrapper to the image and updating the image header accordingly. Using the new image causes the FSBL binary to be loaded in memory at the same address as before. The memory layout does not change.

Since the new image entry point is in the wrapper, it will be executed before the FSBL. The wrapper opens the debug port, waits 2 seconds for the debugger to attach, then:

- if no debugger attaches, the wrapper jumps to execute the FSBL, thus proceeding in the boot process while keeping the debug port open,

- if a debugger attaches, the wrapper jumps and halts at the entry point of the FSBL. The debugger will identify the halt condition.

The new wrapped image has to replace the original image in Flash memory for the whole debug session.

For a complete description of the FSBL image header, go through header for STM32 binary files.

3 Prerequisites

Use either the Distribution Package or the Developer Package.

4 Installing the stm32wrapper4dbg tool

The tool is installed on the host PC as part of the Developer Package. It is also included as internal tool in the Distribution Package, which automatically creates a wrapped FSBL each time a new FSBL is built.

No installation is thus required.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

5 Getting started with the Distribution Package

The Distribution Package creates the wrapped FSBL each time a new FSBL is built. No additional commands are required from the user.

All the wrapped FSBLs are available in the dedicated "debug" subfolders of "arm-trusted-firmware".

5.1 Flashing the wrapped FSBL

If a signed image is required, sign the image in "arm-trusted-firmware/debug/" following the description given in image signing.

If STM32CubeProgrammer is used for writing to the Flash memory, create a modified Flash layout "tsv" and replace the name and location of the original FSBL image file by the ones of the wrapped FSBL file, as described in flash layout description.

6 Getting started with Developer Package

To generate a Trusted Firmware-A BL2 as FSBL, go through TF-A for how to compile.

6.1 Wrapping an FSBL image

Use the following command to wrap an *original.stm32* FSBL image and create a *wrapped.stm32* wrapped FSBL image:

```
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> stm32wrapper4dbg -s original.stm32 -d wrapped.stm32
```

All the FSBL images created by STMicroelectronics toolchains are loaded at the lower available memory address, thus stm32wrapper4dbg appends the wrapper at the end of the image memory map. Instead, on customized FSBLs, the memory map can be different and the only available space for the wrapper can be located at the beginning of image memory map. The *-b* flag can be used to generate a wrapped image that has the wrapper before the FSBL. This also means that the load address is decremented by the size of the wrapper.

```
PC $> stm32wrapper4dbg -s original.stm32 -d wrapped.stm32 -b
```

6.2 Wrapping a signed FSBL image

When stm32wrapper4dbg is used on a signed FSBL image, it detects the signature and displays a warning message. Since stm32wrapper4dbg cannot sign the wrapped image (it has no access to the user's private keys), the user has to sign the wrapped image using the usual signing process, as explained in image signing.

6.3 Loading the wrapped FSBL to Flash memory

The wrapped FSBL image has to replace the FSBL present in the board Flash memory.

If STM32CubeProgrammer is used for writing to the Flash memory, modify the Flash layout "tsv" to replace the name and location of the original FSBL image file with the wrapped FSBL file, as described in flash layout description.

If the Flash memory is an SD card, it is also possible to plug the SD card in the Linux® PC that runs the Development Package and let stm32wrapper4dbg write the wrapped image directly into the first partition of the SD card. Assuming the SD card is recognized as **/dev/sdb** by the Linux PC, run:

```
PC $> stm32wrapper4dbg -s original.stm32 -d /dev/sdb1
```

In addition, since the second partition of the SD card contains a copy of the original image, it is possible to write into the first partition a wrapped version of the FSBL already stored in the second partition with the command

```
PC $> stm32wrapper4dbg -s /dev/sdb2 -d /dev/sdb1
```

7 Modifying a Flash layout tsv file

Create a copy of an existing Flash layout "tsv" file, compatible with your board, for example:

```
PC $> cp flashlayout_st-image-weston/trusted/FlashLayout_sdcard_stm32mp157c-dk2-trusted.
tsv flashlayout_st-image-weston/trusted/debug_FlashLayout_sdcard_stm32mp157c-dk2-trusted.
tsv
```

then edit it and replace in the lines of FSBL

```
P 0x04 fsbl1 Binary mmc0 0x00004400 arm-trusted-firmware/tf-a-stm32mp157c-dk2-trusted.
stm32
P 0x05 fsbl2 Binary mmc0 0x00044400 arm-trusted-firmware/tf-a-stm32mp157c-dk2-trusted.
stm32
```

the path of the FSBL image with the debug version

```
P 0x04 fsbl1 Binary mmc0 0x00004400 arm-trusted-firmware/debug/debug-tf-a-stm32mp157c-dk2-
trusted.stm32
P 0x05 fsbl2 Binary mmc0 0x00044400 arm-trusted-firmware/debug/debug-tf-a-stm32mp157c-dk2-
trusted.stm32
```

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

8 References

None

First Stage Boot Loader

Read Only Memory

Flash memories combine high density and cost effectiveness of EPROMs with the electrical erasability of EEPROMs. For this reason, the Flash memory market is one of the most exciting areas of the semiconductor industry today and new applications requiring in system reprogramming, such as cellular telephones, automotive engine management systems, hard disk drives, PC BIOS software for Plug & Play, digital TV, set top boxes, fax and other modems, PC cards and multimedia CD-ROMs, offer the prospect of very high volume demand.

Boot Loader stage 2

Software development kit (A programming package that enables a programmer to develop applications for a specific platform.)

SD memory card (https://www.sdcard.org)

Linux® is a registered trademark of Linus Torvalds.