# Dmesg and Linux kernel log

# Contents



Stable: 30.03.2021 - 08:00 / Revision: 29.03.2021 - 20:36

Stable: 30.03.2021 - 08:00 / Revision: 29.03.2021 - 20:00

A quality version of this page, approved on *30 March 2021*, was based off this revision.

Contents

# 1 Article purpose

The purpose of this article is to provide information about the Linux® kernel log including configuration, and to detail usage of dmesg command.

# 2 Introduction

Linux kernel is able to print log and trace messages, which are by default stored in a ring buffer.

The same messages can also be displayed, applying filter, on uart/console using serial port. This is defined in the kernel command-line, with the "console" parameter. See [1] for detail.

dmesg is a shell command on the kernel console, which also displays the content of the ring buffer, with filter or not (default).

# 3 printk function

The simplest way to get some debug information from the kernel code is by printing out various information with the kernel's equivalent of printf - the printk function and its derivatives.

```
printk("My Debugger is Printk\n");
```

See elinux.org[2] for reference. This information will be sent to the console, and also stored in a ring buffer.

You can also check to the printk-format.txt[3] document provided in the Linux kernel package to get detail about syntax and formatting.

# 4 Linux kernel ring buffer

The Linux kernel also manages a ring buffer to store log and trace messages.

The size of the buffer cannot be modified in runtime, and its default size value is 2^CONFIG_LOG_BUF_SHIFT bytes.

To change it, there are 3 possibles ways:

- Modify CONFIG_LOG_BUF_SHIFT value in defconfig file or use the config fragment file:

```
In example for 64K : CONFIG_LOG_BUF_SHIFT=16
```

- or use the Linux kernel menuconfig update

```
 Location:
   -> General setup
     -> Kernel log buffer size (16 => 64KB, 17 => 128KB)
```

- Or modify kernel arguments[4] in kernel command-line (via bootargs value in device tree, or directly in extlinux uboot config file)

```
bootargs = "root=/dev/mmcblk0p5 rootwait rw console=ttySTM0,115200 log_buf_len=65536";
```

This ring buffer can be displayed using *dmesg* command (see dmesg).

# 5 Loglevels

As reference, please see elinux.org[5].

The log level is used by the kernel to determine the importance of a message and to decide whether it should be presented to the user immediately, by printing it to the current console.

For this, the kernel compares the log level of the message to the console_loglevel (a kernel variable) and if the priority is higher (i.e. a lower value) than the console_loglevel, the message will be printed to the current console. As example, if console_loglevel=5, all messages with log level 0 to 4 will be displayed.

Please note that all messages with loglevel lower or equal to KERN_INFO level are stored in the ring buffer.

## 5.1 loglevels values

| Name | String | Meaning | alias functions | dev alias function |
|---|---|---|---|---|
| KERN_EMERG | "0" | Emergency messages, system is about to crash or is unstable | pr_emerg | dev_emerg |
| KERN_ALERT | "1" | Something bad happened and action must be taken immediately | pr_alert | dev_alert |
| KERN_CRIT | "2" | A critical condition occurred like a serious hardware/software failure | pr_crit | dev_crit |
| KERN_ERR | "3" | An error condition, often used by drivers to indicate difficulties with the hardware | pr_err | dev_err |
| KERN_WARNING | "4" | A warning, meaning nothing serious by itself but might indicate problems | pr_warning | dev_warn |
| KERN_NOTICE | "5" | Nothing serious, but notably nevertheless. Often used to report security events | pr_notice | dev_notice |
| KERN_INFO | "6" | Informational message e.g. startup information at driver initialization | pr_info | dev_info |
| KERN_DEBUG | "7" | Debug messages | pr_debug, pr_devel if DEBUG is defined | dev_dbg |

**"Loglevels table"**

Important: please note that Higher priority message is loglevel 0

## 5.2 Set loglevel filter value for console

### 5.2.1 Default values

To determine your current console_loglevel on the target you can verifiy with the following command:

```
Board $> cat /proc/sys/kernel/printk
7       4           1       7
current default_msg minimum default_console
```

The first integer shows you the current console_loglevel; the second the default log level, see Use loglevel in the kernel source for log and trace.

This is defined at compilation:

- Current console loglevel via CONFIG_CONSOLE_LOGLEVEL_DEFAULT=7 (defined in file *lib/Kconfig.debug*)
- Default message loglevel via CONFIG_MESSAGE_LOGLEVEL_DEFAULT=4 (defined in file *lib/Kconfig.debug*)
- Minimum console loglevel via #define CONSOLE_LOGLEVEL_MIN 1 (defined in file *include/linux/printk.h*)
- **Default console** loglevel is **equal to CONFIG_CONSOLE_LOGLEVEL_DEFAULT**

### 5.2.2 Using kernel command-line

The console loglevel can be also set via a kernel command-line parameter if you want to use a different value than one specify by CONFIG_CONSOLE_LOGLEVEL_DEFAULT.
For example:

```
root=/dev/mmcblk0p5 rootwait rw console=ttySTM0,115200 loglevel=4
```

In that case only messages with a higher priority than KERN_WARNING (means < 4, KERN_EMERG to KERN_ERR ) will be displayed on the console.

2 ways to add this command-line parameter which is set in the extlinux.conf file of boot partition:

- If using SD card, this is possible to edit the file on host PC:

```
Insert SD card on host PC
Check for mounting boot partition (i.e. /media/$USER/bootfs)
Check for your HW config (i.e. booting on mmc0 (SD Card) with ev1 board)
PC $> cd /media/$USER/bootfs/mmc0_stm32mp157c-ev1_extlinux/
PC $> gedit extlinux.conf
Add loglevel=8 at the end of APPEND line
Save and insert SD card on the board
```

- If using SD Card or eMMC, this is possible to edit the file directly on the board side:

```
When software is boot
Mount boot partition
Board $> mount /dev/mmcblk0p4 /boot (if not already done)
Update the kernel command line
Board $> cd /boot
Board $> cd mmc0_stm32mp157c-ev1_extlinux (case SD card on ev1 board)
Modify extlinux.conf to add loglevel=8 at the end of APPEND line by using ‘vi’ editor
Save and reboot the board
```

### 5.2.3 Using sysfs in runtime

To change your current console_loglevel simply write to this file:

```
Board $> echo <loglevel> > /proc/sys/kernel/printk
```

or using dmesg command.

As example:

```
Board $> echo 8 > /proc/sys/kernel/printk          # Temporary increase loglevel to
display messages up to loglevel 8
```

In that case, every kernel messages will appear on your console, as **all priority higher than 8 (lower loglevel values)** will be displayed.

**Please note that after reboot, this configuration is reset.**

### 5.2.4 Using menuconfig before compilation

As values are defined first at compilation step, this is also possible to set them (**CONFIG_CONSOLE_LOGLEVEL_DEFAULT** and **CONFIG_MESSAGE_LOGLEVEL_DEFAULT**) using the Linux kernel Menuconfig tool (Menuconfig or how to configure kernel):

```
Symbol: CONSOLE_LOGLEVEL_DEFAULT [=7]
Location:
  -> Kernel hacking
    -> printk and dmesg options
      (7) Default console loglevel (1-15)

Symbol: MESSAGE_LOGLEVEL_DEFAULT [=4]
Location:
  -> Kernel hacking
    -> printk and dmesg options
      (4) Default message log level (1-7)
```

## 5.3 Use loglevel in kernel source for log and trace

### 5.3.1 Using printk

A loglevel information can be added in the printk function call, with the following syntax.

```
printk(KERN_ERR "something went wrong, return code: %d\n",ret);
```

When not present, default loglevel value is given by CONFIG_MESSAGE_LOGLEVEL_DEFAULT (usually "4" =KERN_WARNING)

### 5.3.2 Using dedicated functions

In the loglevels table above, there are some alias functions *pr_* and *dev_*.

These functions are defined to replace *printk + loglevel info inside*, in order to simplify syntax.

```
pr_err("something went wrong, return code: %d\n",ret);
```

*dev_* functions are taken one more parameter to provide more information about current device or driver where message is coming from.

- Example for *pr_info*

```
pr_info("%s%s%s at %s (irq = %d, base_baud = %d) is a %s\n",
        port->dev ? dev_name(port->dev) : "",
        port->dev ? ": " : "",
        port->name,
        address, port->irq, port->uartclk / 16, uart_type(port));
```

will display information below:

```
[ 0.919488] 40010000.serial: ttySTM0 at MMIO 0x40010000 (irq = 41, base_baud = 6046875)
is a stm32-usart
```

- Example for *dev_info*

```
dev_info(&pdev->dev, "interrupt mode used for rx (no dma)\n");
```

will display information below, including device reference automatically:

```
[    1.046700] stm32-usart 40010000.serial: interrupt mode used for rx (no dma)
```

# 6 earlyprintk

**earlyprintk** is a Linux kernel debug feature useful to get traces for kernel issues which happen before the normal console is initialized.

## 6.1 Linux kernel configuration

In order to enable earlyprintk feature, the Linux kernel configuration must activate **CONFIG_DEBUG_LL**, **CONFIG_STM32MP1_DEBUG_UART** and **CONFIG_EARLY_PRINTK** using the Linux kernel Menuconfig tool (Menuconfig or how to configure kernel):

```
Symbol: DEBUG_LL
Depends on: DEBUG_KERNEL
Location:
  -> Kernel hacking
    -> arm Debugging
      [*] Kernel low-level debugging functions

Symbol: STM32MP1_DEBUG_UART
Location:
  -> Kernel hacking
    -> arm Debugging
      [*] Kernel low-level debugging functions
        [*] Kernel low-level debugging port
        (X) Use STM32MP1 UART for low-level debug

Symbol: EARLY_PRINTK
Depends on: DEBUG_LL
Location:
  -> Kernel hacking
    -> arm Debugging
      [*] Kernel low-level debugging functions
      [*] Early printk
```

## 6.2 Serial port configuration

When enabling the Linux kernel configuration **CONFIG_STM32MP1_DEBUG_UART**, it configures the addresses of the UART registers to be used.

By default, on STM32MP1 boards, UART4 is used for console for Linux kernel and by extension at all boot stages.

In case the UART port is different on a new board, you must apply the following changes:

- Update value for **CONFIG_DEBUG_UART_PHYS**, to select the UART port for the debug console

```
Symbol: DEBUG_UART_PHYS [=0x40010000]
Location:
  -> Kernel hacking
    -> arm Debugging
      [*] Kernel low-level debugging functions
            Kernel low-level debugging port (Use STM32MP1 UART for low-level debug)
      (0x40010000)  Physical base address of debug UART
```

- Update value for **CONFIG_DEBUG_UART_VIRT**, to define the associated virtual address to be used

```
Symbol: DEBUG_UART_VIRT [=0xFE010000]
Location:
  -> Kernel hacking
    -> arm Debugging
      [*] Kernel low-level debugging functions
            Kernel low-level debugging port (Use STM32MP1 UART for low-level debug)
      (0xFE010000)  Virtual base address of debug UART
```

Following rules to be respected for defining the virtual address:

- The 20 low weight bits (21 in case LPAE is enabled) must be kept in order to align region size of 1MB (2MB in LPAE is enabled).
- It must be mapped at the upper address of the vmalloc area, in order to not be overwritten by kernel which is stating from lower addresses: i.e here we select 0xFE0xxxxx

```
CONFIG_DEBUG_UART_PHYS: 0x40010000 /* UART4 */
CONFIG_DEBUG_UART_VIRT: 0xFE010000
```

- Please find below table for USART/UART of STMP32MP1:

| Name | Physical base address | Virtual base address |
|---|---|---|
| USART 1 | 5c000000 | **FE0**00000 |
| USART 2 | 4000e000 | **FE0**0e000 |
| USART 3 | 4000f000 | **FE0**0f000 |
| UART4 | 40010000 | **FE0**10000 |
| UART5 | 40011000 | **FE0**11000 |
| USART 6 | 44003000 | **FE0**03000 |
| UART7 | 40018000 | **FE0**18000 |
| UART8 | 40019000 | **FE0**19000 |

**Note that the UART port used for console must be aligned for all components of the boot chain: FSBL(TF-A), SSBL-U-Boot) and Linux kernel**

Especially because the Linux kernel do not configure all the setting registers for the UART port, as this is done by SSBL (U-Boot - How to debug)
See also How to debug TF-A BL2 for FSBL changes)

## 6.3 Linux kernel boot command configuration

The Linux kernel boot command must contain the command-line parameter **earlyprintk**.

For instance, the kernel *bootargs* can be modified in the following ways:

- Mount a boot partition from the Linux kernel console, and then update the extlinux.conf file using the vi editor (see man page [6], or introduction page[7] ). For example:

```
Board $> mount /dev/mmcblk0p4 /boot
Board $> vi /boot/mmc0_stm32mp157c-ev1_extlinux/extlinux.conf
```

or

- Edit the extlinux.conf file by using UMS (USB Mass Storage): see How to use USB mass storage in U-Boot for details.

To mount partitions (mmc 0:microSD card / mmc 1: eMMC):

- Press any key to stop at U-Boot execution when booting the board.

```
Board $> ...
Board $> Hit any key to stop autoboot:  0
Board $> STM32MP>
```

- Then

```
STM32MP> ums 0 mmc 0
```

- Check for the boot partition mounted on your host PC (/media/*$USER*/bootfs)
- Edit the extlinux file corresponding to your setup (/media/*$USER*/bootfs/mmc0_stm32mp157c-ev2_extlinux/extlinux.conf

- Update the kernel command line by adding the **earlyprintk** parameter:

```
root=/dev/mmcblk0p6 rootwait rw earlyprintk console=ttySTM0,115200
```

Save and quit file update, and then reboot the board.

## 6.4 Get trace

Earlyprintk traces are pushed automatically to the serial console defined as seen previously, and also added to the kernel ring log buffer.

# 7 dmesg command

As reference, please see man page[8].

The Kernel ring buffer can be displayed using *dmesg* command. It will display on the console all the content of the ring buffer.

- It is possible to filter messages following the loglevels:

```
Board $> dmesg -n <loglevel>
```

In that case, only messages with a value lower (**not lower equal**) than the console_loglevel will be printed.

Here, *<loglevel>* can be a numeric value, but also a string:

```
Supported log levels (priorities):
  emerg (0)
  alert (1)
   crit (2)
    err (3)
   warn (4)
 notice (5)
   info (6)
  debug (7)
```

As example:

```
Board $> dmesg -n 8          # Temporary change loglevel to display messages up to debug
level
or
Board $> dmesg -n debug
```

In that case, every kernel messages will appear on your console, as **all priority higher than 8 (lower loglevel values)** will be displayed.

- It is possible to clear the dmesg buffer

```
Board $> dmesg -c         # Display the full content of dmesg ring buffer, and then clear
it
Board $> dmesg -C         # Clear the dmesg ring buffer
```

# 8 /var/log/messages file system entry

An other way to display the content of the Linux kernel log is to look at the content of the file /var/log/messages.

It contains general system activity messages from the start-up. It also provides useful information about origin of the message, and log level.

# 9 Dynamic debug message

These messages are using the loglevel 7 (KERN_DEBUG).

Please see How to use the kernel dynamic debug article.

## 10 References


- Linux Serial Console
- https://elinux.org/Debugging_by_printing
- https://www.kernel.org/doc/Documentation/printk-formats.txt
- The kernel's command-line parameters
- https://elinux.org/Debugging_by_printing#Log_Levels
- http://ex-vi.sourceforge.net/vi.html
- http://ex-vi.sourceforge.net/viin/paper.html
- http://man7.org/linux/man-pages/man1/dmesg.1.html




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

former spelling for e•MMC ('e' in italic)

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Low layer of STM32Cube

Universal Asynchronous Receiver/Transmitter

Universal Synchronous/Asynchronous Receiver/Transmitter

First Stage Boot Loader

Trusted Firmware for Arm® Cortex®-A

Second Stage Boot Loader

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

User-space Mode Setting

Stable: 13.10.2021 - 12:36 / Revision: 14.09.2021 - 09:05

A quality version of this page, approved on *13 October 2021*, was based off this revision.

Contents

# 1 Article Purpose

This article explains how to debug TF-A BL2 firmware.
Debug is specifically linked to the TrustZone environment.

There are two main ways to debug TF-A, using traces inside the code, or by using JTAG to access the secure world. The focus here is on the solution integrated in OpenSTLinux: debug over gdb (ST-Link or JTAG based)

# 2 Debugging

## 2.1 TF-A Version number

The starting point for debugging is to identify the TF-A BL2 version used in the target. Debug and release versions are displayed on the console with the following format:

```
NOTICE: BL2: v2.0(debug):<tag>
```

- v2.0 is the TF-A BL2 version used.
- (debug) : Build mode enable
- <tag> : Git reference resulting from the **git describe** command at build time

## 2.2 Debug with traces

TF-A allows RELEASE and DEBUG modes to be enabled:

- RELEASE mode builds with default LOG_LEVEL to 20, which only prints ERROR and NOTICE traces
- DEBUG mode enables the -g build flag (for debug build object), and defaults LOG_LEVEL to 40

With the debug LOG_LEVEL, you can add console traces such as ERROR, NOTICE, WARNING or INFO. Please refer to include/common/debug.h .

**You cannot build code with LOG_LEVEL set to 50 (the highest level); there are too many traces and TF-A does not fit in the SYSRAM. If required, change some VERBOSE settings to INFO.**
**Another possibility is to add the following lines at the beginning of the .c file, before the includes:**

```
#undef LOG_LEVEL
#define LOG_LEVEL 50
```

For both modes, you must ensure that the UART is properly configured:

- BL2: UART traces are enabled by default

Traces and errors are available on the console defined in the chosen node of the device tree by the stdout-path property:

```
chosen {
        stdout-path = "serial0:115200n8";
};
```

Traces will only be available after the console has been fully registered. Prior to that, the traces are not displayed except panic messages, so it is recommended to use a debugger.

The panic messages are displayed through the crash console. It uses hard-coded UART settings. Those settings should be adapted to your platform, in the file plat/st/stm32mp1/stm32mp1_def.h , after the comment:

```
/* For UART crash console */
```

You will have to setup the base address of the UART to use, its clock source frequency and the GPIO configuration of the UART TX pin.

## 2.3 Debug with GDB

The BL2 boot stage is only executed during a boot sequence. To break into BL2, connect to the target and break.

Load symbols to the correct offset:

```
(gdb) add-symbol-file <path_to_build_folder>/bl2/bl2.elf <load_address>
```

BL2 load address can be found in the generated **tf-a-<board>.map** file:

```
...
 *fill*         0x000000002ffc3000    0x20000
...
                0x000000002ffea000                __BL2_IMAGE_START__ = .  -> BL2 Load
address
 *(.bl2_image*)
 .bl2_image     0x000000002ffea000    0xf4a5           build/stm32mp1/stm32mp1.o
                0x000000002fff94a5                __BL2_IMAGE_END__ = .
                0x000000002fff94a5                __DATA_END__ = .
                0x000000002fff94a5                __TF_END__ = .
...
```

In this example:

- BL2 load address is 0x2ffea000.

You can load all your symbols directly:

```
(gdb) add-symbol-file <path_to_build_folder>/bl2/bl2.elf 0x2ffea000
```

### 2.3.1 Boot from a flashed storage

TF-A BL2 runs in SYSRAM and is executed in CPU secure state. One can debug TF-A BL2 through JTAG using an ST-Link or the JTAG output, depending on the board. It is recommended to use the Wrapper_for_FSBL_images to be able to interrupt the initial boot sequence at the early beginning (before BL2 code execution).

Set your hardware breakpoint and reset:

```
(gdb) hb bl2_entrypoint
(gdb) monitor reset halt
(gdb) continue
```

### 2.3.2 Boot from programmer mode

If TF-A BL2 is not yet flashed on an external storage, it is possible to debug the programmer (USB or UART) boot sequence. The ROM code is opening non secure debug during the programmer wait loop. It is possible to connect GDB during this stage. You can load all your symbols directly:

```
(gdb) add-symbol-file <path_to_build_folder>/bl2/bl2.elf 0x2ffea000
```

Set your hardware breakpoint and continue:

```
(gdb) hb bl2_entrypoint
(gdb) continue
```

Load the TF-A BL2 built with the USB/UART support using the STM32CubeProgrammer.
Ex in USB mode:

```
   PC $> STM32_Programmer_CLI -c port=usb1 -w tf-a-<board>.stm32 0x01 -s
```

GDB will break at the first BL2 instruction ready for step by step debugging session.

```
Breakpoint 1, bl2_entrypoint () at bl2/aarch32/bl2_el3_entrypoint.S:18
18                mov        r9, r0
```

Trusted Firmware for Arm® Cortex®-A

Boot Loader stage 2

TrustZone®

*Arm® and TrustZone® are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

debug and test protocol, named from the Joint Test Action Group that developed it

spelling for older versions of STLink, ST in-circuit debugger and programmer for the STM8 and STM32 microcontroller families

Universal Asynchronous Receiver/Transmitter

General-Purpose Input/Output (A realization of open ended transmission between devices on an embedded level. These pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and applications requirements.)

Transmit

GNU dedugger, a portable debugger that runs on many Unix-like systems

Central processing unit

Stable: 22.03.2021 - 14:32 / Revision: 16.02.2021 - 14:56

A quality version of this page, approved on *22 March 2021*, was based off this revision.

This page explains how to use the U-Boot command "ums" to update an SD card or eMMC on the device.

# 1 ums command

In U-Boot, you can directly export the available block devices (sd/mmc/usb) as USB mass storage devices with ums command:

```
Board $> help ums
ums - Use the UMS [USB Mass Storage]

Usage:
ums <USB_controller> [<devtype>] <dev[:part]>  e.g. ums 0 mmc 0
    devtype defaults to mmc
```

This U-Boot command "ums" is infinite (a loop in USB treatments), and the U-Boot console is blocked until user enters a Ctrl-C.

## 2 Exporting a block device

On ST boards, the OTG USB controller device index is 0, SD card = "mmc 0" and, when available, eMMC = "mmc 1".
You can check the device connected on an SDMMC with the U-Boot command "mmc info".

You can also export a USB device connected to the USB host controller (USBH) = "usb 0".

Then execute one of the following commands:

```
                                Comments
Board $> ums 0 mmc 0 --> start ums on SD card
Ctrl-C
```

```
                                Comments
Board $> ums 0 mmc 1 --> start ums on eMMC
Ctrl-C
```

```
                                Comments
Board $> usb start   --> start USB host controller
Board $> ums 0 usb 0 --> start ums on USB device 0 (USB key for example)
Ctrl-C
Board $> usb stop    --> stop USB host controller
```

After a delay (of up to 15 seconds), the host sees the exported block device and you can use any command on the PC to access the partitions of the exported memory (dd, mount, cp, rsync). A Ctrl-C is needed to stop the command.

See also How to manually update bootloaders.

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

former spelling for e•MMC ('e' in italic)

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

User-space Mode Setting

Stable: 02.11.2020 - 10:48 / Revision: 19.10.2020 - 12:09

A quality version of this page, approved on *2 November 2020*, was based off this revision.

Contents

# 1 Introduction

As prerequisite to reading this article, please refer to the Dmesg and Linux kernel log page.

"Dynamic debug is designed to allow you to dynamically enable/disable kernel code to obtain additional kernel information. Currently, if **CONFIG_DYNAMIC_DEBUG** is set, all **pr_debug()/dev_dbg()** calls can be dynamically enabled per-callsite." extracted from the Linux kernel documentation[1].

The related debugfs entry is usually:

```
/sys/kernel/debug/dynamic_debug/control
```

Note that the verbose **dev_vdbg()** calls cannot be dynamically activated.

When the dynamic debug traces are activated, the trace results are printed in **dmesg (or /proc/kmsg)**, and in the console if console loglevel is set to 8.

# 2 More technical information

The dynamic debug trace configuration is done through a **control** file in the **debugfs** filesystem: *<debugfs>/dynamic_debug /control*

The command includes keywords and flag elements (for details see the Linux kernel documentation[1]).

- Keywords

Possible keywords are:

```
func : function name
file : source filename
module : module name
format : format string
line : line number (including ranges of line numbers)
```

The colored keywords above are illustrated by examples in the next chapter.

- Flags

The flag specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

```
- : remove the given flags
+ : add the given flags
= : set the flags to the given flags
```

Possible flags are:

```
f : Include the function name in the printed message
l : Include line number in the printed message
m : Include module name in the printed message
p : Causes a printk() message to be emitted to dmesg
t : Include thread ID in messages not generated from interrupt context
```

# 3 Examples

- Track all dev_*dbg/pr_debug() in a **file** (you can add several files if necessary):

```
Board $> mount -t debugfs none /sys/kernel/debug
Board $> echo "file stm32-adc.c +p" > /sys/kernel/debug/dynamic_debug/control
```

Note that just the file name or full file path can be given, here *stm32-adc.c* or *drivers/iio/adc/stm32-adc.c*

- Track only one **line** with dev_dbg() in a **file** (you can add several files and several lines if necessary, please use the last line number of the function call):

```
Board $> echo "file stm32-adc.c line 1438 +p" > /sys/kernel/debug/dynamic_debug/control
```

- For an entire "**module** (module means ~.ko, so not applicable for a statically linked driver)":

```
Board $> echo "module cfg80211 +p" > /sys/kernel/debug/dynamic_debug/control
```

- If you want to list all available traces *(warning: it is a long file so you may need to use "tee" or another solution to save it)*:

```
Board $> cat /sys/kernel/debug/dynamic_debug/control | tee /tmp/dynamic_log.log
```

- For instance, if you are looking for a particular **file** to find a particular **line**:

```
Board $> cat /sys/kernel/debug/dynamic_debug/control | grep adc
drivers/iio/adc/stm32-adc.c:1515 [stm32_adc]stm32_adc_conf_scan_seq =p "%s chan %d to %s%
d\012"
drivers/iio/adc/stm32-adc.c:1438 [stm32_adc]stm32_adc_awd_set =p "%s chan%d htr:%d ltr:%
d\012"
drivers/iio/adc/stm32-adc.c:2182 [stm32_adc]stm32_adc_dma_start =p "%s size=%d watermark=%
d\012"
drivers/iio/adc/stm32-adc.c:2304 [stm32_adc]stm32_adc_trigger_handler =p "%s bufi=%d\012"
drivers/iio/adc/stm32-adc.c:2443 [stm32_adc]stm32_adc_chan_of_init =p "Configured to use
injected\012"
drivers/iio/adc/stm32-adc.c:2364 [stm32_adc]stm32_adc_of_get_resolution =p "Using %u bits
resolution\012"
```

- Multiple commands can be written together, separated by ';' or '\n'.

```
Board $> echo "file stm32-adc.c +p; file stm32-adc-core.c +p" > /sys/kernel/debug
/dynamic_debug/control
```

- A another method is to use a wildcard. The match rule supports * (matches zero or more characters) and ? (matches exactly one character). For example, you can match all USB drivers:

```
Board $> echo "file drivers/usb/* +p" > /sys/kernel/debug/dynamic_debug/control
```

# 4 Synchronous tracing on the console

In the case of a crash, or impossibility to call dmesg, it is sometimes useful to have traces synchronously emitted on the console.
Only error, warning and informational traces are emitted synchronously on the console (that is, loglevel=5), so if you need to see the lower level traces too, you need to change the console loglevel to "8".

```
<enable the conditional traces>
Board $> echo 8 > /proc/sys/kernel/printk
or
Board $> dmesg -n 8
or
Board $> dmesg -n debug
```

Please follow this article to get a serial console for the target: How to get Terminal

**As all traces are now synchronously emitted, real-time is affected**

If you want to return to the default console log level, you have to get this default value from the procfs entry */proc/sys/kernel/printk*:

```
Board $> cat /proc/sys/kernel/printk
8       4       1       7
Board $> dmesg -n 7
Board $> cat /proc/sys/kernel/printk
7       4       1       7
```

# 5 Debug messages during boot process

In order to activate debug messages during the boot process, even before userspace and debugfs exist, use the kernel's command-line parameter: **dyndbg**

For instance, the kernel *bootargs* can be modified in the following ways:

- Mount a boot partition from the Linux kernel console, and then update the extlinux.conf file using the vi editor (see man page [2], or introduction page[3] ). For example:

```
Board $> mount /dev/mmcblk0p4 /boot
Board $> vi /boot/mmc0_stm32mp157c-ev1_extlinux/extlinux.conf
```

or

- Edit the extlinux.conf file by using UMS (USB Mass Storage): see How to use USB mass storage in U-Boot for details.

To mount partitions (mmc 0:microSD card / mmc 1: eMMC):

- Press any key to stop at U-Boot execution when booting the board.

```
Board $> ...
Board $> Hit any key to stop autoboot:  0
Board $> STM32MP>
```

- Then

```
STM32MP> ums 0 mmc 0
```

- Check for the boot partition mounted on your host PC (/media/*$USER*/bootfs)
- Edit the extlinux file corresponding to your setup (/media/*$USER*/bootfs/mmc0_extlinux/stm32mp157f-dk2_extlinux.conf

- Update the kernel command line, adding the dyndbg parameter:

```
root=PARTUUID=e91c4e10-16e6-4c0e-bd0e-77becf4a3582 rootwait rw console=ttySTM0,115200 dynd
bg="file drivers/usb/core/hub.c +p"
```

Save and quit file update, and then reboot the board.

**Note**: to display these debug messages in the console, in addition to the dmesg, add **loglevel=8** in the kernel command line.

- Reboot the board and check for a kernel command-line, and that debug messages are present in the dmesg output

# 6 References

- 1.01.1 Documentation/admin-guide/dynamic-debug-howto.rst
- http://ex-vi.sourceforge.net/vi.html
- http://ex-vi.sourceforge.net/viin/paper.html

- Useful external links

| Document link | Document Type | Description |
|---|---|---|
| The dynamic debugging interface (lwn.net) | User guide | http://lwn.net |
| Documentation/dynamic-debug-howto.txt (lwn.txt) | User guide | http://lwn.net |
| Dynamic debug howto (kernel.org) | Standard | http://www.kernel.org |



Debug File System (See https://en.wikipedia.org/wiki/Debugfs for more details)

Process File System (See https://en.wikipedia.org/wiki/Procfs for more details)

User-space Mode Setting

former spelling for e•MMC ('e' in italic)

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Stable: 31.03.2021 - 08:47 / Revision: 26.03.2021 - 08:44

A quality version of this page, approved on *31 March 2021*, was based off this revision.

Contents

# 1 Linux configuration genericity

The process of building a kernel has two parts: configuring the kernel options and building the source with those options.

The Linux® kernel configuration is found in the generated file: .config.

.config is the result of configuring task which is processing platform defconfig and fragment files if any.

For OpenSTLinux distribution the defconfig is located into the kernel source code and fragments into stm32mp BSP layer :

- arch/arm/configs/**multi_v7_defconfig**

Every new kernel version brings a bunch of new options, we do not want to back port them into a specific defconfig file each time the kernel releases, so we use the same defconfig file based on ARM SoC v7 architecture.
STM32MP1 specificities are managed with fragments config files.

- meta-st/meta-st-stm32mp/recipes-kernel/linux/linux-stm32mp/<kernel version>/**fragment-*.config**

.config result is located in the build folder:

- build-openstlinuxweston-stm32mp1/tmp-glibc/work/stm32mp1-ostl-linux-gnueabi/linux-stm32mp/5.10.10-rc0/build/**.config**

**To modify the kernel options, it is not recommended to edit this file directly.**

- A user runs either a text-mode :

```
   PC $> make config
starts a character based question and answer session (Figure 1)
```

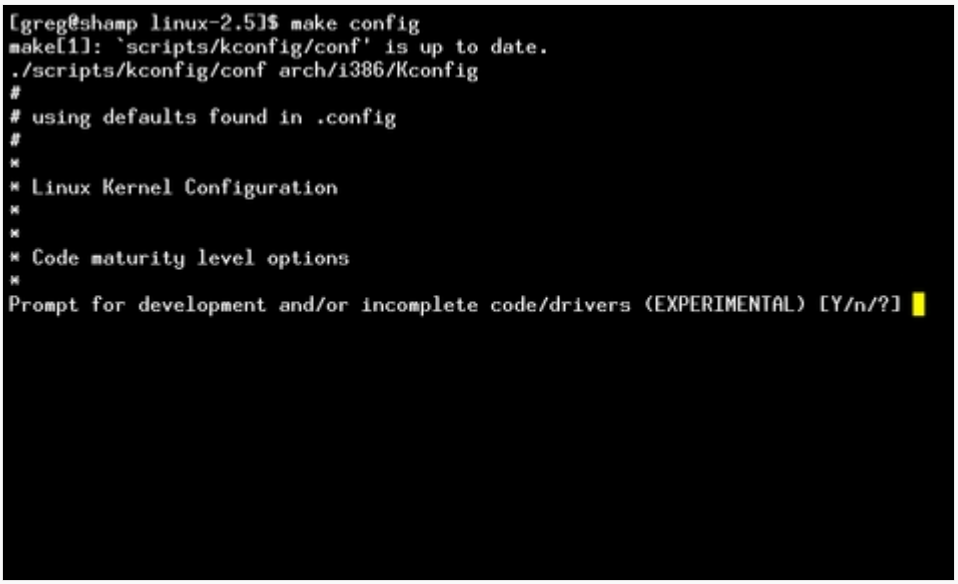



Figure 1. Configuring the kernel with make config

```
   PC $> make
menuconfig
starts a terminal-
oriented
configuration tool
(using ncurses)
(Figure 2)
The ncurses text
version is more
popular and is run
with the make
menuconfig option.
Wikipedia Menuconfig[1
] also explains how
to "navigate" within
the configuration
menu, and highlights
main key strokes.
```

- or a graphical kernel

configurator :

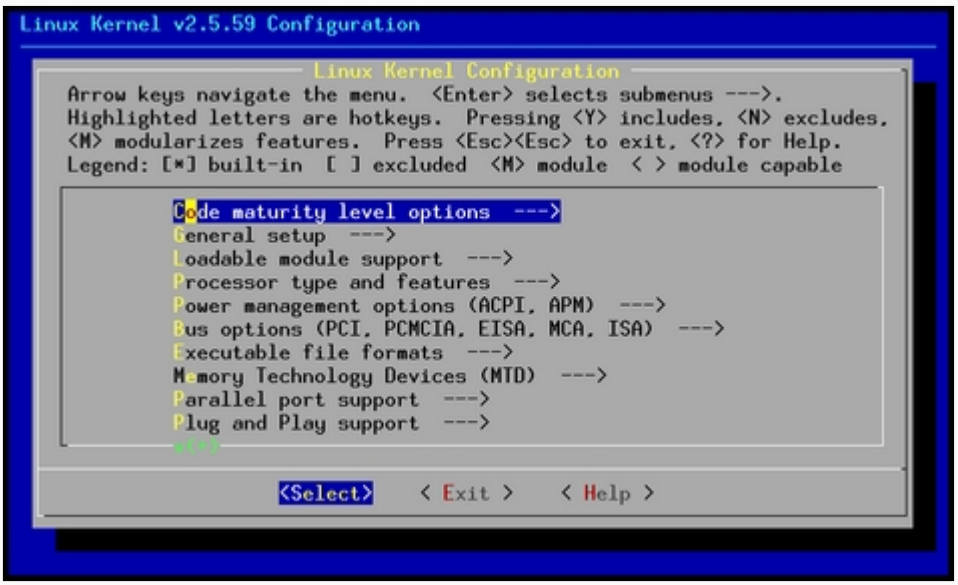



Figure 2. Make menuconfig makes it easier to back up and correct mistakes

**PC $>** make xconfig starts a X based configuration tool (Figure 3)

Ultimately these configuration tools edit the .config file.

An option indicates either some driver is built into the kernel ("=y") or will be built as a module ("=m") or is not selected.

The unselected state can either be indicated by a line starting with "#" (e.g. "# CONFIG_SCSI is not set") or by the absence of the relevant line from the .config file.

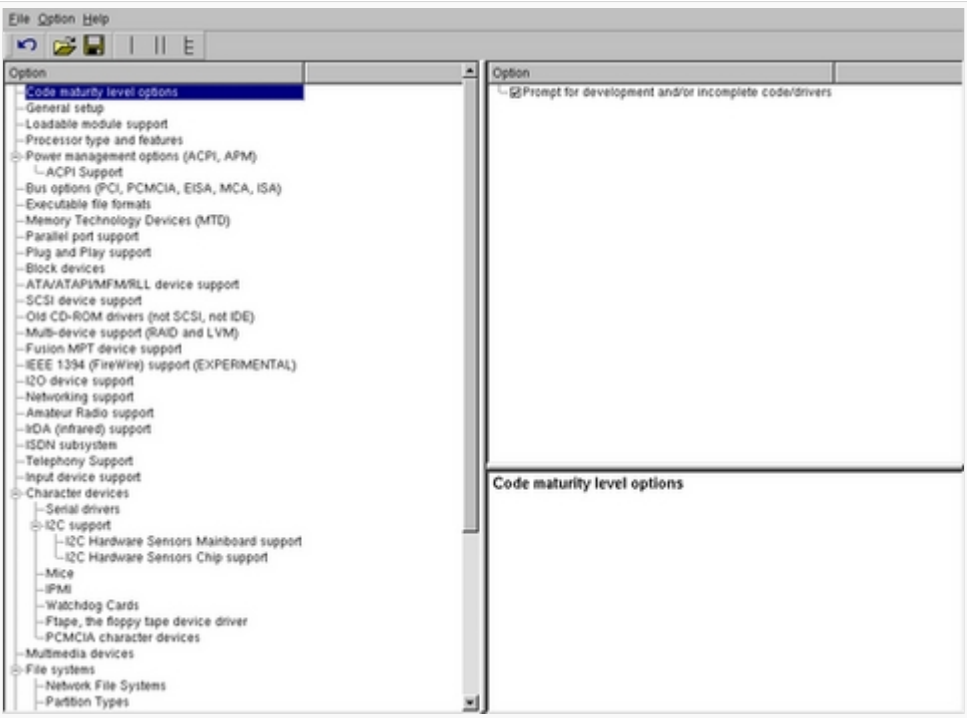



Figure 3. The Qt-Based make xconfig

The 3 states of the main selection option for the SCSI subsystem (which actually selects the SCSI mid level driver) follow. Only one of these should appear in an actual .config file:

```
CONFIG_SCSI=y
CONFIG_SCSI=m
# CONFIG_SCSI is not
set
```

# 2 Menuconfig and Developer Package

For this use case, the prerequesite is that OpenSTLinux SDK has been installed and configured.

To verify if your cross-compilation environment has been put in place correctly, run the following command:

```
PC $> set | grep CROSS
CROSS_COMPILE=arm-ostl-linux-gnueabi-
```

For more details, refer to *<Linux kernel installation directory>/README.HOW_TO.txt* helper file (the latest version of this helper file is also available in GitHub: README.HOW_TO.txt ).

- Go to the <Linux kernel build directory>

```
PC $> cd <Linux kernel build directory>
```

- Save initial configuration (to identify later configuration updates)

```
PC $> make arch=ARM savedefconfig
Result is stored in defconfig file
PC $> cp defconfig defconfig.old
```

- Start the Linux kernel configuration menu

```
PC $> make arch=ARM menuconfig
```

- Navigate forwards or backwards directly between feature
  - un/select, modify feature(s) you want
  - When the configuration is OK : exit and save the new configuration

```
useful keys to know:
enter: enter in config subdirectory
space: hit several times to either select [*], select in module [m] or unselect [ ]
/: to search for a keyword, this is usefull to navigate in tree
?: to have more information on selected line
```

- Compare the old and new config files after operating modifications with menuconfig

```
PC $> make arch=ARM savedefconfig
```

Retrieve configuration updates by comparing the new defconfig and the old one

```
PC $> meld defconfig defconfig.old
```

- Cross-compile the Linux kernel (please check the load address in the *README.HOW_TO.txt* helper file)

```
PC $> make arch=ARM uImage LOADADDR=<loadaddr of kernel>
PC $> cp arch/arm/boot/uImage install_artifact/boot/
```

- Update the Linux kernel image on board

```
PC $> scp install_artifact/boot/uImage root@<board ip address>:/boot/
```

If the */boot* mounting point doesn't exist yet, please see how to create a mounting point

- Reboot the board

```
Board $> cd /boot; sync; systemctl reboot
```

Note that this use case modifies the configuration file in the Linux kernel build directory, not in the Linux kernel source directory: this is a temporary modification useful for a prototyping.

- To make this temporary modification permanent, the delta between defconfig and defconfig.old must be saved in a configuration fragment file (fragment-*.config) based on fragment.cfg file, and the Linux kernel configuration/compilation steps must be re-executed (as explained in the README.HOW_TO.txt helper file).

# 3 Menuconfig and Distribution Package

- Start the Linux kernel configuration menu

```
PC $> bitbake virtual/kernel -c menuconfig
```

- Navigate forwards or backwards directly between feature
  - un/select, modify feature(s) you want
  - When the configuration is OK : exit and save the new configuration

```
useful keys to know:
enter: enter in config subdirectory
space: hit several times to either select [*], select in module [m] or unselect [ ]
/: to search for a keyword, this is usefull to navigate in tree
?: to have more information on selected line
```

- Cross-compile the Linux kernel

```
PC $> bitbake virtual/kernel
```

- Update the Linux kernel image on board

```
PC $> scp <build dir>/tmp-glibc/deploy/images/<machine name>/uImage root@<board ip
address>:/boot
```

If the */boot* mounting point does not exist yet, please see how to create a mounting point

- Reboot the board

```
Board $> cd /boot; sync; systemctl reboot
```

Note that this use case modifies the configuration file in the Linux kernel build directory, not in the Linux kernel source directory: this is a temporary modification useful for a prototyping.

- To make this temporary modification permanent, it must be saved in a configuration fragment file (fragment-*.config) based on **fragment.cfg** file, and the Linux kernel configuration/compilation steps must be re-executed: **bitbake <name of kernel recipe>**.

# 4 References

- Wikipedia Menuconfig

Linux® is a registered trademark of Linus Torvalds.

Board support package

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

Stable: 09.03.2021 - 10:11 / Revision: 09.03.2021 - 10:09

A quality version of this page, approved on *9 March 2021*, was based off this revision.

Contents

# 1 Debug with console

Trace and error are available on the U-Boot console which use `stdout-path` defined in the `chosen` node of the Linux kernel device tree as described in the Linux kernel binding[1].

See page How to configure U-Boot for your board for configuration details.

By default, the macros used by U-Boot (debug(), pr_debug()...) do not print any trace; to activate the debug traces on a specific file, you need to enable the **DEBUG** compilation flag and change the LOGLEVEL for the file:

- define DEBUG before any include in the <file>.c file

```
#define DEBUG
#undef CONFIG_LOGLEVEL
#define CONFIG_LOGLEVEL 8
```

- with a Makefile

```
CFLAGS_<file>.o+= -DDEBUG -DCONFIG_LOGLEVEL=8
```

For details, see doc/README.log .

If U-Boot fails before the console configuration (in the first stage of U-Boot execution), trace is not available.

In this case, you need to:

- debug with GDB (see the next chapter)

or,

- activate the debug UART feature:
  - add in defconfig of U-Boot configuration
    - CONFIG_DEBUG_UART
    - CONFIG_DEBUG_UART_STM32
  - adpat the function board_debug_uart_init(): that configures the required resources (pad, clock) before initialization by the U-Boot driver.
    This function needs to be adapted for your board.

# 2 Debug with GDB

With OpenSTLinux, you can directly use GDB script Setup.gdb:

- GDB#U-Boot_execution_phase
- GDB#U-Boot_boot_case

Or for manual GDB connection, you need to:

1. get the elf files for U-Boot and/or SPL
(u-boot and u-boot-spl available in the build directory)
2. connect GDB to the target
3. **reset with attach** the target with the gdb **"monitor reset halt"** command:
execution is stopped in ROM code or at the beginning of FSBL execution.
4. load the symbols of the binary to be debugged with commands available in next chapter:
#Load U-Boot symbol, #Load SPL symbol, #Load SPL code and debug
5. start execution with the **"continue"** command

## 2.1 Load U-Boot symbol

With U-Boot relocation, symbols are more difficult to load.

See https://www.denx.de/wiki/DULG/DebuggingUBoot

If you connect GDB on running target, you can load the debug symbols:

- Before relocation with **"symbol-file"** command:

```
(gdb) symbol-file u-boot
```

- After relocation with **"add-symbol-file"** command to relocate the symbol with the code offset = gd->relocaddr:

```
(gdb) symbol-file u-boot                              --> only for "gd_t" definition
(gdb) set $offset = ((gd_t *)$r9)->relocaddr          --> get relocation offset
(gdb) symbol-file                                     --> clear previous symbol
(gdb) add-symbol-file u-boot $offset
```

The following GDB example script automatically loads the U-Boot symbol before and after relocation for a programmed board, after **"monitor reset halt"** command:

```
(gdb) thbreak *0xC0100000
(gdb) commands
> symbol-file u-boot
> thbreak relocate_code
> commands
  > print "RELOCATE U-Boot..."
  > set $offset = ((gd_t *)$r9)->relocaddr
  > print $offset
  > symbol-file
```

```
    > add-symbol-file u-boot $offset
    > thbreak boot_jump_linux
    > continue
    > end
  > continue
  > end
```

This script uses a temporary hardware breakpoint **"thbreak"** to load the symbol when U-Boot code is loaded in DDR by FSBL = TF-A or SPL at the U-Boot entry point (CONFIG_SYS_TEXT_BASE = 0xC0100000).

It allows the symbol to be loaded only when code is executed to avoid DDR access before DDR initialization.

# 3 Debug SPL with GDB

SPL is also supported with *"stm32mp15_basic_defconfig"* but only for U-Boot_SPL:_DDR_interactive_mode.

**This alternate boot chain with SPL is not supported/promoted by STMicroelectronics to make product.**

## 3.1 Load SPL symbol

To debug SPL with GDB on a Flashed device, ROM code loads the binary and the GDB script just loads the SPL symbols:

```
(gdb) symbol-file u-boot-spl
```

## 3.2 Load SPL code and debug

Sometimes you need to debug SPL execution on an unprogrammed target (for example for board bring-up), so you can use GDB to load the SPL code in embedded RAM and execute it.

When execution is stopped in ROM code, you need to execute the **"load"** commands, depending on the compilation flags defined in U-Boot device tree to load the SPL code and the associated device tree:

- CONFIG_OF_SEPARATE = dtb appended at the end of the code, not present in the elf file (default configuration)

```
(gdb) file u-boot-spl
(gdb) load
(gdb) set $dtb =  __bss_end
(gdb) restore spl/dt.dtb binary $dtb
```

- CONFIG_OF_EMBED = dtb embedded in the elf file (debug configuration)

```
(gdb) file u-boot-spl
(gdb) load
```

## 3.3 Debug the first SPL instructions

Sometime the SPL code execution is stopped by the gdb command "monitor reset halt" after the first instructions.

To debug this part, you can modify the code: add a infinite loop in SPL code to wait the gdb connection.

For example in arch/arm/mach-stm32mp/spl.c :

```
void board_init_f(ulong dummy)
{
        struct udevice *dev;
        int ret;

        /* volatile is needed to avoid gcc optimization */
```

```
        volatile int stop = 0;
        /* infinite debug loop */
        while ( !stop ) ;

        arch_cpu_init();
```

when gdb is attached and the SPL symbols are loaded, the infinite loop is interrupted by :

```
(gdb) set var stop=1
```

And you can debug the SPL first instruction by gdb commands.

# 4 References

- Documentation/devicetree/bindings/chosen.txt the Linux kernel binding for chosen node

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Linux® is a registered trademark of Linus Torvalds.

GNU dedugger, a portable debugger that runs on many Unix-like systems

Universal Asynchronous Receiver/Transmitter

Secondary Program Loader, *Also known as* ***U-Boot SPL***

Read Only Memory

First Stage Boot Loader

Doubledata rate (memory domain)

Trusted Firmware for Arm® Cortex®-A

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Stable: 19.10.2021 - 13:54 / Revision: 19.10.2021 - 13:54

A quality version of this page, approved on *19 October 2021*, was based off this revision.

Contents

# 1 Das U-Boot

Das U-Boot ("the Universal Boot Loader" or U-Boot) is an open-source bootloader that can be used on ST boards to initialize the platform and load the Linux® kernel.

- Official website: https://www.denx.de/wiki/U-Boot
- Official manual: U-Boot project documentation and https://www.denx.de/wiki/DULG/Manual
- Official **source code** is available under git repository at [1]

Read the README file before starting using U-Boot. It covers the following topics:

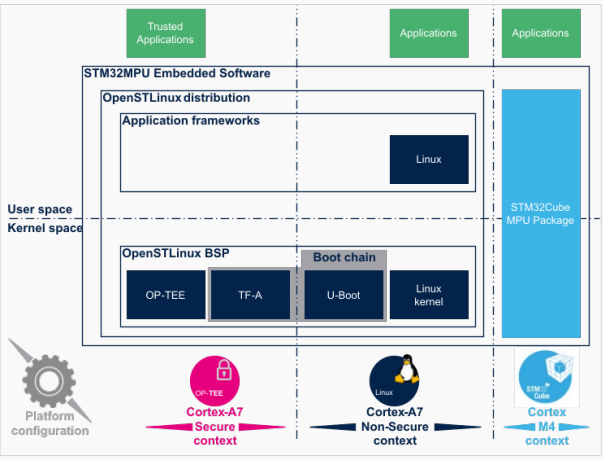
- source file tree structure
- description of CONFIG defines
- instructions for building U-Boot
- brief description of the Hush shell
- list of common environment variables

Do go further, read the documentations available in doc/ and the documentation generated by `make htmldocs` [1].

# 2 U-Boot overview

The STM32 MPU boot chain uses Trusted Firmware-A (TF-A) as FSBL and U-Boot as SSBL.

The same U-Boot source can also generate an alternate FSBL named SPL. The boot chain becomes: SPL as FSBL and U-Boot as SSBL.




Zoom out to STM32MPU Embedded Software

**This alternate boot chain with SPL cannot be used for product development.**

## 2.1 SPL: alternate FSBL

### 2.1.1 SPL description

The **U-Boot SPL** or **SPL** is an alternate first stage bootloader (FSBL).
It is a small binary (bootstrap utility) generated from the U-Boot source and stored in the internal limited-size embedded RAM.
SPL main features are the following:

- It is loaded by the ROM code.
- It performs the initial CPU and board configuration (clocks and DDR memory).
- It loads the SSBL (U-Boot) into the DDR memory.

### 2.1.2 SPL restrictions

**SPL cannot be used for product development.**

SPL is provided only as an example of the simplest FSBL with the objective to support upstream U-Boot development. However, several known limitations have been identified when SPL is used in conjunction with the minimal secure monitor provided within U-Boot for basic boot chain. These limitations apply to:

- power management
- secure access to registers
- limited features (STM32CubeProgrammer / boot from NAND Flash memory)
- SCMI support for clock and reset (not compatible with latest Linux kernel device tree)

There is no workaround for these limitations.

### 2.1.3 SPL execution sequence

**SPL** executes the following main steps in SYSRAM:

- **board_init_f()**: driver initialization including DDR initialization (mininimal stack and heap: CONFIG_SPL_STACK_R_MALLOC_SIMPLE_LEN)
- configuration of heap in DDR memory (CONFIG_SPL_SYS_MALLOC_F_LEN)
- **board_init_r()**: initialization of the other drivers activated in the SPL device tree
- loading and execution of U-Boot (or Kernel in Falcon mode[2]: README.falcon ).

## 2.2 U-Boot: SSBL

### 2.2.1 U-Boot description

**U-Boot** is the second-stage bootloader (SSBL) of boot chain for STM32 MPU platforms.

SSBL main features are the following:

- It is configurable and expendable.
- It features a simple command line interface (CLI), allowing users to interact over a serial port console.
- It provides scripting capabilities.
- It loads the kernel into RAM and gives control to the kernel.
- It manages several internal and external devices such as NAND and NOR Flash memories, Ethernet and USB.
- It supports the following features and commands:
  - File systems: FAT, UBI/UBIFS, JFFS
  - IP stack: FTP
  - Display: LCD, HDMI, BMP for splashcreen
  - USB: host (mass storage) or device (DFU stack)

### 2.2.2 U-Boot execution sequence

**U-Boot** executes the following main steps in DDR memory:

- **Pre-relocation** initialization (common/board_f.c): minimal initialization (such as CPU, clock, reset, DDR and console) running at the CONFIG_SYS_TEXT_BASE load address.
- **Relocation**: copy of the code to the end of DDR memory.
- **Post-relocation initialization**:(common/board_r.c): initialization of all the drivers.
- **Command execution** through autoboot (CONFIG_AUTOBOOT) or console shell.
  - Execution of the boot command (by default bootcmd=CONFIG_BOOTCOMMAND):

  for example, execution of the command `bootm` to:
    - load and check images (such as kernel, device tree and ramdisk)
    - fixup the kernel device tree
    - install the secure monitor (optional) or
    - pass the control to the Linux kernel (or to another target application)

# 3 U-Boot configuration

The U-Boot binary configuration is based on

- **Kbuild infrastructure** (as in Linux Kernel, you can use `make menuconfig` in U-Boot)

The configurations are based on:

  - options defined in Kconfig files (CONFIG_ compilation flags)
  - the selected configuration file: configs/stm32mp*_defconfig

- **other compilation flags** defined in include/configs/stm32mp*.h (these flags are progressively migrated to Kconfig)

The file name is configured through CONFIG_SYS_CONFIG_NAME.

For STM32MP15x lines, the include/configs/stm32mp1.h file is used.

- DeviceTree: U-Boot binaries include a device tree blob that is parsed at runtime

All the configuration flags (prefixed by CONFIG_) are described in the source code, either in the README file or in the documentation directory .

For example, CONFIG_SPL activates the SPL compilation.

Hence to compile U-Boot, select the <target> and the device tree for the board in order to choose a predefined configuration.

Refer to #U-Boot_build for examples.

## 3.1 Kbuild

Like the kernel, the U-Boot build system is based on configuration symbols (defined in Kconfig files). The selected values are stored in a **.config** file located in the build directory, with the same makefile target. .

Proceed as follows:

- Select a predefined configuration (defconfig file in configs directory ) and generate the first **.config**:

```
PC $> make <config>_defconfig.
```

- Change the U-Boot compile configuration (modify .config) by using one of the following five `make` commands:

```
PC $> make menuconfig --> menu based program
PC $> make config  --> line-oriented configuration
PC $> make xconfig --> QT program[3]
PC $> make gconfig --> GTK program
PC $> make nconfig --> ncurse menu based program
```

You can then compile U-Boot with the updated .config.

Warning: the modification is performed locally in the build directory. It will be lost after a `make distclean`.

Save your configuration to be able to use it as a defconfig file:

```
PC $> make savedefconfig
```

This target saves the current config as a defconfig file in the build directory. It can then be compared with the predefined configuration (configs/stm32mp*defconfig).

The other makefile targets are the following:

```
 PC $> make help
....
Configuration targets:
  config          - Update current config utilising a line-oriented program
  nconfig         - Update current config utilising a ncurses menu based
                    program
  menuconfig          - Update current config utilising a menu based program
  xconfig          - Update current config utilising a Qt based front-end
  gconfig          - Update current config utilising a GTK+ based front-end
  oldconfig          - Update current config utilising a provided .config as base
  localmodconfig  - Update current config disabling modules not loaded
  localyesconfig  - Update current config converting local mods to core
  defconfig          - New config with default from ARCH supplied defconfig
  savedefconfig   - Save current config as ./defconfig (minimal config)
  allnoconfig           - New config where all options are answered with no
  allyesconfig           - New config where all options are accepted with yes
  allmodconfig           - New config selecting modules when possible
  alldefconfig    - New config with all symbols set to default
  randconfig          - New config with random answer to all options
  listnewconfig   - List new options
  olddefconfig           - Same as oldconfig but sets new symbols to their
                   default value without prompting
```

## 3.2 Device tree

Refer to doc/README.fdt-control for details.

The board device tree has the same binding as the kernel. It is integrated within the U-Boot binaries: u-boot.bin

- By default, it is appended at the end of the code (CONFIG_OF_SEPARATE).
- It can be embedded in the U-Boot binary (CONFIG_OF_EMBED). This is particularly useful for debugging since it enables easy .elf file loading.

The U-Boot device tree (u-boot.dtb) can be also provided as external file loaded by FSBL when U-Boot code is started (u-boot-nodtb.bin: code without device tree): device tree address is provided as boot parameter (in r2 register).

A default device tree is available in the defconfig file (by setting CONFIG_DEFAULT_DEVICE_TREE).

You can either select another supported device tree using the DEVICE_TREE make flag. For stm32mp boards, the corresponding file is <dts-file-name>.dts in arch/arm/dts/stm32mp*.dts , with <dts-file-name> set to the full name of the board:

```
PC $> make DEVICE_TREE=<dts-file-name>
```

or provide a device tree blob (dtb file) resulting from the dts file compilation, by using the EXT_DTB option:

```
PC $> make EXT_DTB=boot/<dts-file-name>.dtb
```

The SPL device tree is also generated from this device tree. However to reduce its size, the U-Boot makefile uses the fdtgrep tool to parse the full U-Boot DTB and identify all the drivers required by SPL.

To do this, U-Boot uses specific device-tree flags to determine if the associated driver is initialized prior to U-Boot relocation and /or if the associated node is present in SPL :

- **u-boot,dm-pre-reloc** => present in SPL, initialized before relocation in U-Boot
- **u-boot,dm-pre-proper** => initialized before relocation in U-Boot
- **u-boot,dm-spl** => present in SPL

In the device tree used by U-Boot, these flags **need to be added in all the nodes** used in SPL or in U-Boot before relocation, and for all used handles (clock, reset, pincontrol).

To obtain a device tree file **<dts-file-name>.dts** that is identical to the Linux kernel one, these U-Boot properties are only added for ST boards in the add-on file **<dts-file-name>-u-boot.dtsi**. This file is automatically included in **<dts-file-name>.dts** during device tree compilation (this is a generic U-Boot Makefile behavior).

# 4 U-Boot command line interface (CLI)

Refer to U-Boot Command Line Interface.

If CONFIG_AUTOBOOT is activated, you have CONFIG_BOOTDELAY seconds (2s by default, 1s for ST configuration) to enter the console by pressing any key, after the line below is displayed and bootcmd is executed (CONFIG_BOOTCOMMAND):

```
Hit any key to stop autoboot:  2
```

## 4.1 Commands

The commands are defined in cmd/*.c . They are activated through the corresponding **CONFIG_CMD_*** configuration flag.

Use the `help` command in the U-Boot shell to list the commands available on your device:

```
Board $> help
```

Below the list of all commands extracted from U-Boot Manual (**not-exhaustive**):

- Information Commands
  - bdinfo - prints Board Info structure
  - coninfo - prints console devices and information
  - flinfo - prints Flash memory information
  - iminfo - prints header information for application image
  - help - prints online help
- Memory Commands
  - base - prints or sets the address offset
  - crc32 - checksum calculation
  - cmp - memory compare
  - cp - memory copy
  - md - memory display
  - mm - memory modify (auto-incrementing)
  - mtest - simple RAM test
  - mw - memory write (fill)
  - nm - memory modify (constant address)
  - loop - infinite loop on address range
- Flash Memory Commands
  - cp - memory copy
  - flinfo - prints Flash memory information
  - erase - erases Flash memory
  - protect - enables or disables Flash memory write protection
  - mtdparts - defines a Linux compatible MTD partition scheme
- Execution Control Commands
  - source - runs a script from memory
  - bootm - boots application image from memory

- go - starts application at address 'addr'

- Download Commands
  - bootp - boots image via network using BOOTP/TFTP protocol
  - dhcp - invokes DHCP client to obtain IP/boot params
  - loadb - loads binary file over serial line (kermit mode)
  - loads - loads S-Record file over serial line
  - rarpboot- boots image via network using RARP/TFTP protocol
  - tftpboot- boots image via network using TFTP protocol
- Environment Variables Commands
  - printenv- prints environment variables
  - saveenv - saves environment variables to persistent storage
  - setenv - sets environment variables
  - run - runs commands in an environment variable
  - bootd - default boot, that is run 'bootcmd'
- Flattened Device Tree support
  - fdt addr - selects the FDT to work on
  - fdt list - prints one level
  - fdt print - recursive printing
  - fdt mknode - creates new nodes
  - fdt set - sets node properties
  - fdt rm - removes nodes or properties
  - fdt move - moves FDT blob to new address
  - fdt chosen - fixup dynamic information
- Special Commands
  - i2c - I2C sub-system
- Storage devices
- Miscellaneous Commands
  - echo - echoes args to console
  - reset - performs a CPU reset
  - sleep - delays the execution for a predefined time
  - version - prints the monitor version

To add a new command, refer to doc/README.commands .

## 4.2 U-Boot environment variables

The U-Boot behavior is configured through environment variables.

Refer to Manual and README / Environment Variables.

On the first boot, U-Boot uses a default environment embedded in the U-Boot binary. You can modify it by changing the content of CONFIG_EXTRA_ENV_SETTINGS in your configuration file (for example ./include/configs/stm32mp1.h) (see README / - Default Environment).

This environment can be modified and saved in the boot device. When it is present, it is loaded during U-Boot initialization:

- To boot from *e*•MMC/SD card (CONFIG_ENV_IS_IN_MMC): at the end of the partition indicated by config field "u-boot,mmc-env-partition" in device-tree (for ST boards: partition named "fip" in ecosystem release v3.0.0 with FIP support or partition named "ssbl" without FIP support).

- To boot from NAND Flash memory (CONFIG_ENV_IS_IN_UBI): in the two UBI volumes "config" (CONFIG_ENV_UBI_VOLUME) and "config_r" (CONFIG_ENV_UBI_VOLUME_REDUND).
- To boot from NOR Flash memory (CONFIG_ENV_IS_IN_SPI_FLASH): the u-boot_env mtd parttion (at offset CONFIG_ENV_OFFSET).

### 4.2.1 env command

The `env` command allows displaying, modifying and saving the environment in U-Boot console.

```
Board $> help env
env - environment handling commands

Usage:
env default [-f] -a - [forcibly] reset default environment
env default [-f] var [...] - [forcibly] reset variable(s) to their default values
env delete [-f] var [...] - [forcibly] delete variable(s)
env edit name - edit environment variable
env exists name - tests for existence of variable
env print [-a | name ...] - print environment
env print -e [name ...] - print UEFI environment
env run var [...] - run commands in an environment variable
env save - save environment
env set -e name [arg ...] - set UEFI variable; unset if 'arg' not specified
env set [-f] name [arg ...]
```

Example: proceed as follows to restore the default environment and save it. This is useful after a U-Boot upgrade:

```
Board $> env default -a
Board $> env save
```

You can also use the command activated by CONFIG_CMD_ERASEENV:

```
Board $> env erase
```

### 4.2.2 bootcmd

"bootcmd" variable is the autoboot command. It defines the command executed when U-Boot starts (CONFIG_BOOTCOMMAND).

For stm32mp, CONFIG_BOOTCOMMAND="run bootcmd_stm32mp":

```
Board $> env print bootcmd
 bootcmd=run bootcmd_stm32mp
```

"bootcmd_stm32mp" is a script that selects the command to be executed for each boot device (see ./include/configs/stm32mp1.h), based on generic distro scripts:

- To boot from a serial/usb device: execute the `stm32prog` command.
- To boot from an *e*•MMC, SD card: boot only on the same device (bootcmd_mmc...).
- To boot from a NAND Flash memory: boot on ubifs partition on the NAND memory (bootcmd_ubi0).
- To boot from a NOR Flash memory: use the SD card (on SDMMC 0 on ST boards with bootcmd_mmc0)

```
Board $> env print bootcmd_stm32mp
```

You can then change this configuration:

- either permanently in your board file
  - default environment by CONFIG_EXTRA_ENV_SETTINGS (see ./include/configs/stm32mp1.h)
  - change CONFIG_BOOTCOMMAND value in your defconfig

```
CONFIG_BOOTCOMMAND="run bootcmd_mmc0"
```

```
CONFIG_BOOTCOMMAND="run distro_bootcmd"
```

- or temporarily in the saved environment:

```
Board $> env set bootcmd run bootcmd_mmc0
Board $> env save
```

Note: To reset the environment to its default value:

```
Board $> env default bootcmd
Board $> env save
```

## 4.3 Generic Distro configuration

Refer to doc/README.distro for details.

This feature is activated by default on ST boards (CONFIG_DISTRO_DEFAULTS):

- one boot command (bootmcd_xxx) exists for each bootable device.
- U-Boot is independent from the Linux distribution used.
- bootcmd is defined in ./include/config_distro_bootcmd.h

When DISTRO is enabled, the command that is executed by default is include/config_distro_bootcmd.h :

```
bootcmd=run distro_bootcmd
```

This script tries any device found in the 'boot_targets' variable and executes the associated bootcmd.

Example for mmc0, mmc1, mmc2, pxe and ubifs devices:

```
bootcmd_mmc0=setenv devnum 0; run mmc_boot
bootcmd_mmc1=setenv devnum 1; run mmc_boot
bootcmd_mmc2=setenv devnum 2; run mmc_boot
bootcmd_pxe=run boot_net_usb_start; dhcp; if pxe get; then pxe boot; fi
bootcmd_ubifs0=setenv devnum 0; run ubifs_boot
```

U-Boot searches for an **extlinux.conf** configuration file for each bootable device. This file defines the kernel configuration to be used with `bootm` command:

- bootargs
- files to start the OS:
  - kernel (uImage) + device tree + ramdisk files (optional)

- FIT image, including all these needed files (for details see doc/uImage.FIT/howto.tx )

## 4.4 U-Boot scripting capabilities

"Script files" are command sequences that are executed by the U-Boot command interpreter. This feature is particularly useful to configure U-Boot to use a real shell (hush) as command interpreter.

See U-Boot script manual for an example.

# 5 U-Boot build

See U-Boot Documentation.

## 5.1 Prerequisites

- a PC with Linux and tools:
  - see PC_prerequisites
  - #ARM cross compiler
- U-Boot source code
  - the latest STMicroelectronics U-Boot version
    - tar.xz file from Developer Package (for example STM32MP1) or from latest release on ST github [4]
    - from GITHUB[5], with `git` command

```
PC $> git clone https://github.com/STMicroelectronics/u-boot
```

  - from the Mainline U-Boot in official GIT repository [6]

```
PC $> git clone https://source.denx.de/u-boot/u-boot.git
```

## 5.2 ARM cross compiler

A cross compiler [7] must be installed on your Host (X86_64, i686, ...) for the ARM targeted Device architecture. In addition, the $PATH and $CROSS_COMPILE environment variables must be configured in your shell.

You can use gcc for ARM, available in:

- the SDK toolchain (see Cross-compile with OpenSTLinux SDK)

PATH and CROSS_COMPILE are automatically updated.

- an existing package

For example, install gcc-arm-linux-gnueabihf on Ubuntu/Debian: (**PC $>** sudo apt-get.

- an existing toolchain:
  - latest gcc toolchain provided by arm (https://developer.arm.com/open-source/gnu-toolchain/gnu-a/downloads/)
  - gcc v7 toolchain provided by linaro: (https://www.linaro.org/downloads/)

For example, to use *gcc-arm-9.2-2019.12-x86_64-arm-none-linux-gnueabihf.tar.xz* from arm, extract the toolchain in $HOME and update your environment with:

```
PC $> export PATH=$HOME/gcc-arm-9.2-2019.12-x86_64-arm-none-linux-gnueabihf/bin:$PATH
PC $> export CROSS_COMPILE=arm-none-linux-gnueabihf-
```

For example, to use **gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi.tar.xz**
from https://releases.linaro.org/components/toolchain/binaries/7.2-2017.11/arm-linux-gnueabi/
Unzip the toolchain in $HOME and update your environment with:

```
PC $> export PATH=$HOME/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi/bin:$PATH
PC $> export CROSS_COMPILE=arm-linux-gnueabi-
```

## 5.3 Compilation

In the U-Boot source directory, select the defconfig for the <target> and the <device tree> for your board and then execute the `make all` command:

```
PC $> make <target>_defconfig
PC $> make DEVICE_TREE=<device tree> all
```

Use help to list other targets than all:

```
PC $> make help
```

Optionally

- **KBUILD_OUTPUT** can be used to change the output build directory in order to compile several targets in the source directory. For example:

```
PC $> export KBUILD_OUTPUT=<path>
```

- **DEVICE_TREE** can also be exported to your environment when only one board is supported. For example:

```
PC $> export DEVICE_TREE=<device-tree>
```

The result is the following:

```
PC $> export KBUILD_OUTPUT=<path>
PC $> export DEVICE_TREE=<device tree>
PC $> make <target>_defconfig
PC $> make all
```

Examples from STM32MP15 U-Boot:

The boot chain for STM32MP15x lines use stm32mp15_trusted_defconfig:

```
PC $> make stm32mp15_trusted_defconfig
PC $> make DEVICE_TREE=stm32mp157f-dk2 all
```

```
PC $> export KBUILD_OUTPUT=../build/stm32mp15_trusted
PC $> export DEVICE_TREE=stm32mp157c-ev1
PC $> make stm32mp15_trusted_defconfig
PC $> make all
```

## 5.4 Output files

The resulting U-Boot files are located in your build directory (U-Boot or KBUILD_OUTPUT).

Since ecosystem release v3.0.0 , two U-Boot files are used by ST boards to generate FIP used by FSBL TF-A, with or without OP-TEE support:

- BL33_CFG=u-boot.dtb : the U-Boot device tree, selected by DEVICE_TREE, loaded by TF-A BL2 and amended by secure monitor (SPMIN or OP-TEE)
- BL33=u-boot-nodtb.bin: the U-Boot executable, loaded by TF-A BL2 started by secure monitor with BL33_CFG as parameter

Nota: All the compiled device tree are available in $KBUILD_OUTPUT/arch/arm/dts/*.dtb.
You can select them as BL33_CFG without U-Boot recompilation.

See TF-A_overview for FIP details.

The file used to debug with gdb is

- u-boot : elf file for U-Boot

For ecosystem release v2.1.0 : u-boot.stm32 : U-Boot binary with STM32 image header, including device tree selected by DEVICE_TREE, loaded by TF-A

This behavior can be restored if you activate CONFIG_STM32MP15x_STM32IMAGE in your defconfig of ecosystem release v3.0.0

This temporary option is only introduced to facilitate the FIP migration but it will be removed in the next EcosystemRelease.

The STM32 image format (*.stm32) is managed by mkimage U-Boot tools and Signing_tool. It is requested by ROM code and TF-A without FIP support (see STM32 header for binary files for details).

# 6 References

- https://u-boot.readthedocs.io/en/stable/index.html
- https://www.denx.de/wiki/pub/U-Boot/MiniSummitELCE2013/2013-ELCE-U-Boot-Falcon-Boot.pdf
- https://en.wikipedia.org/wiki/Xconfig
- https://github.com/STMicroelectronics/u-boot/releases
- https://github.com/STMicroelectronics/u-boot
- https://source.denx.de/u-boot/u-boot.git or https://github.com/u-boot/u-boot
- https://en.wikipedia.org/wiki/Cross_compiler

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Linux® is a registered trademark of Linus Torvalds.

First Stage Boot Loader

Secondary Program Loader, *Also known as* ***U-Boot SPL***

Second Stage Boot Loader

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Read Only Memory

Central processing unit

Doubledata rate (memory domain)

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.

System control and management interface

Microprocessor Unit

High-Definition Multimedia Interface (HDMI standard)

Device Firmware Upgrade

Device Tree Binary (or Blob)

Memory Technology Device

Trivial File Transfer Protocol (https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol)

Dynamic Host Configuration Protocol (See https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol for more details)

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

MultimediaCard

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

Firmware Image Package is a packaging format used by TF-A

Serial Peripheral Interface

Operating System

Flattened uImage Tree is a packaging format used by U-Boot

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

Trusted Firmware for Arm® Cortex®-A

Open Portable Trusted Execution Environment

Boot Loader stage 3-3

Boot Loader stage 2