U-Boot - How to debug

Contents

Stable: 26.05.2020 - 15:42 / Revision: 26.05.2020 - 15:41

Stable: 26.05.2020 - 13:42 / Revision: 26.05.2020 - 13:41

A quality version of this page, approved on *26 May 2020*, was based off this revision.

Contents

1 Debug with console

Trace and error are available on the console defined in the **chosen** node of the device tree by the **stdout-path** field, for example on serial0=uart4:

```
chosen {
        stdout-path = " serial0:115200n8";
};
aliases {
        serial0 = &uart4;
};
```

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.

2.2 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
```

2.3 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
```

2.4 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.

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

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: 15.02.2021 - 15:02 / Revision: 12.01.2021 - 09:16

A quality version of this page, approved on *15 February 2021*, was based off this revision.

Contents

1 Article purpose

This article provides the basic information needed to start using the **GDB**[1] application tool.

It explains how to use this GNU debugger tool connected to your ST board target via Ethernet or via ST-LINK, and how to perform cross-debugging (IDE, gdb viewer tool or command line) on Arm®Cortex®-A7 side for Linux® application, Linux® kernel (including external modules), or Arm® Cortext®-M4 firmware.

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**
GDB	Debugging tools	The GNU Project debugger, **GDB**[1], allows monitoring program execution, or what the program was doing at the moment it crashed.	✗*	✓	✗**	✗	✗	✓
			** Cross compile gdb and openocd binaries are required and only available from Developer Package.* *** It is recommended to use the Developer Package to run the gdb debug session, which provided all dependencies*			** Cross compile gdb and openocd binaries are required and only available from Distribution Package.*		

The GDB can perform four main types of actions (plus other corollary actions) to help you detect bugs when running your software or application:

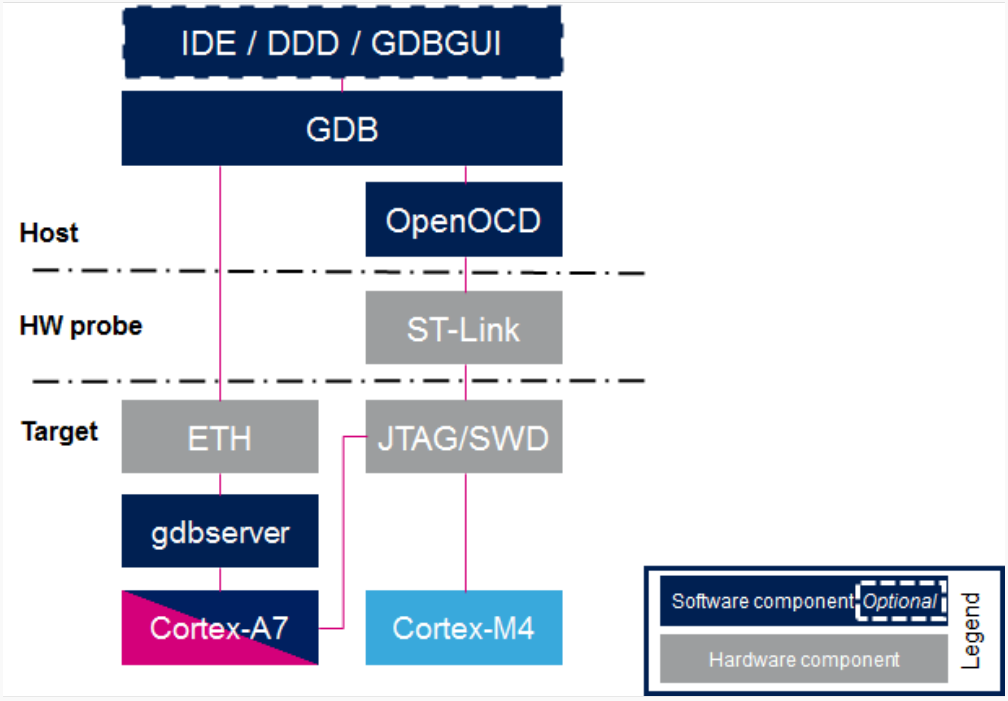
- Start the program, specifying anything that might affect its behaviour.
- Make the program stop on specific conditions.
- Examine what happened when the program stopped.
- Update the program in order to test a bug correction, and jump to the next one.

3 Overview of GDB setup for STM32MPU

3.1 GDB setup paths

Two paths can be used in the STM32MPU environment for GDB setup:

- **gdb gdbserver** path through Ethernet, **used for Cortex-A7 Linux applications**.
  In that case, two software components are required, one on the target and the other on the host PC.
- **gdb JTAG/SWD** path through OpenOCD and ST-LINK, **used both for Cortex-M4 firmware and Cortex-A7 Linux kernel**.
  In that case, only one software component is required on the host PC.



GDB paths scheme

Two components are included in OpenSTLinux Developer Package for GDB setup:

- **gdbserver**: **embedded on target rootfs** and used as remote access for a host connection
- **arm-ostl-linux-gnueabi-gdb**: **embedded on host PC side**, cross-compiled gdb binary that manages the connexion between the host computer and the target board

3.2 JTAG and SWD debug port

The STM32MPU features two debug ports through the embedded CoreSight™ component that implements an external access port for connecting debugging equipment:

- A 5-pin standard JTAG interface (JTAG-DP)
- A 2-pin (clock + data) “serial-wire debug” port (SW-DP)

These two modes are mutually exclusive since they share the same I/O pins.

Refer to STM32MP15 reference manuals for information related to JTAG and SWD.

4 Installing the GDB tool

This tool is made of two parts, one on the host PC, and a second on the target (only for debugging Linux applications).

4.1 Installing the GDB tool on your host PC

Below is the list of required components:

- The cross-compiled GDB binary
- The OpenOCD binary and configuration files
- The symbols files of all BSP components (TF-A, U-Boot and Linux kernel), corresponding to the images of the OpenSTLinux Starter Package.

4.1.1 Using the STM32MPU Embedded Software distribution

4.1.1.1 Developer Package

Only the Developer Package can be used, since it is the only one which provides all the required components.

4.1.2 Using the STM32MPU Embedded Software distribution for Android™

4.1.2.1 Distribution Package

Only Distribution Package can be used, since it is the only one which provides all the required components.

4.2 Installing the GDB on your target board

On the target board, only the gdbserver binary is required for debugging Linux applications.

It is available by default within the Starter Package, which provides images linked to the Developer Package.

Below information is related to the Android™ distribution
It is also available by default within the Starter Package for Android™

5 Getting started

This chapter describes the two ways for debugging OpenSTLinux BSP components (TF-A, U-Boot and Linux kernel including external modules), Linux applications and Cortex-M4 firmware: by using GDB commands or by using a GDB user interface such as gdbgui, DDD or IDE.

5.1 Prerequisites

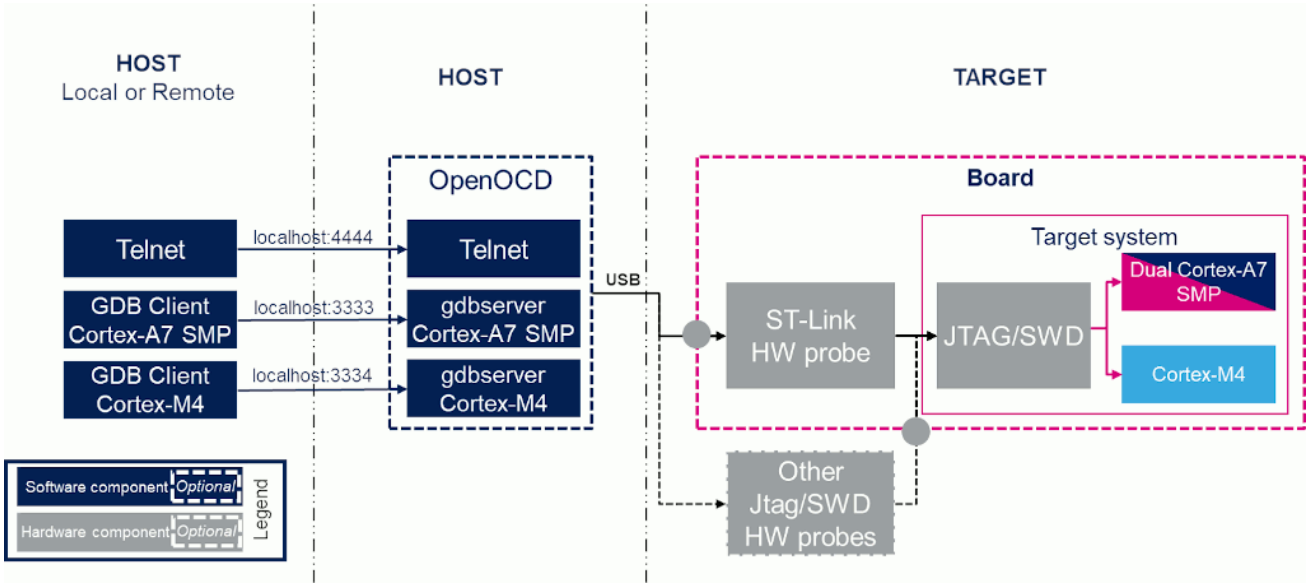
The target board is up and running.

5.2 Debug OpenSTLinux BSP components

5.2.1 Setting up GDB / OpenOCD debug path environment

- **Architecture**

The figure below shows the architecture corresponding to the GDB/OpenOCD connected to Cortex-A7 and Cortex-M4 cores.
Note: The ST-LINK probes available on the STM32MP1 board can be used through a USB interface, as well as any other external probes through the Trace or JTag connector.



- **Prerequisites**

  The Developer Package must be installed. It provides the SDK, the debug symbol files and the source files for TF-A, U-Boot and Linux kernel (refer to STM32MP1 Developer Package).
  The debug symbol files contain the symbols for the TF-A, U-Boot and Linux kernel binaries (from the Starter Package image) that have been flashed on the board.

Below information is related to the Android™ distribution

The Distribution Package must be installed.

It's required to load sources:

- Load bootloader sources (TF-A and U-Boot): `load_bootloader` (see How to build bootloaders for Android for more details)
- Load OP-TEE sources: `load_tee` (see How to build TEE for Android for more details)
- Load Linux kernel sources: `load_kernel` (see How to build kernel for Android for more details)

It's also required to rebuild the kernel (by default `vmlinux` is not available and the modules are stripped):

- Build Linux kernel and modules: `build_kernel -gi` (see How to build kernel for Android for more details)

- **Environment configuration files**

  To speed up environment setup for debugging with GDB, download two configuration files, and install them on your PC (*under the home directory: $HOME/gdbscripts/*). You can then customize them:

  - Setup.gdb: main configuration file in which you can define the debug context you want to use (Refer to Boot chain overview for details). Possible combinations are:

  - Trusted boot chain:

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 TF-A(BL32)	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
(0) - Boot	✓	✓	✓	✓
(1) - Runnin				

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 TF-A(BL32)	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
target	✗	✗	✓	✓

- Trusted boot chain with OP-TEE:

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 OP-TEE	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
(0) - Boot	✓	✓	✓	✓
(1) - Running				

Debug mode	(1) Cortex-A7 TF-A(BL2)	(2) Cortex-A7 OP-TEE	(3) Cortex-A7 SSBL(U-Boot)	(4) Cortext-A7 Linux kernel
g ta r g et	⊗	⊗	✓	✓

- Path_env.gdb: customization file to define all the paths to source and symbol files, which can either be directly retrieved from the **Developer Package** (refer to the Example of directory structure for Packages), or result from a new compilation. **In both cases, update the paths corresponding to your environment.**
Please read carefully the comments provided in this file to help you updating source and symbol paths.

Store these files locally on your host PC, **check for name cast (Setup.gdb and Path_env.gdb)** and update them accordingly.

Below information is related to the Android™ distribution

The STM32MPU distribution for Android™ integrates by default the secure OS OP-TEE. By consequence:

- remove `symload_bl32` and `symadd_bl32` from `Path_env.gdb`.
- set `set $debug_trusted_bootchain = 1` in `Setup.gdb`.

As example, the full paths for STM32MP157F-EV1 board required in `Path_env.gdb`:

- symload_optee/symadd_optee: *<Your_Android_distribution_path>*/device/stm/stm32mp1-tee/prebuilt/stm32mp157f-ev1/optee_os/tee-stm32mp157f-ev1.elf
- symload_bl2: *<Your_Android_distribution_path>*/device/stm/stm32mp1-bootloader/prebuilt/fsbl/stm32mp157f-ev1/tf-a-bl2-stm32mp157f-ev1-optee.elf
- symload_uboot/symadd_uboot:
  - case microSD card: *<Your_Android_distribution_path>*/device/stm/stm32mp1-bootloader/prebuilt/ssbl/stm32mp157f-ev1/u-boot-stm32mp157f-ev1-trusted-fbsd.elf
  - case eMMC: *<Your_Android_distribution_path>*/device/stm/stm32mp1-bootloader/prebuilt/ssbl/stm32mp157f-ev1/u-boot-stm32mp157f-ev1-trusted-fbemmc.elf
- symload_vmlinux: *<Your_Android_distribution_path>*/device/stm/stm32mp1-kernel/prebuilt/vmlinux-stm32mp1

5.2.2 Configuring GDB and OpenOCD for attachment on a running target

When the target board is running, you can attach the GDB only during one of following phases:

5.2.2.1 U-Boot execution phase

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 3
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 1
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When the configuration is complete, jump to Running OpenOCD and GDB.

5.2.2.2 *Linux kernel execution phase*

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 4
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 1
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When the configuration is complete, jump to Running OpenOCD and GDB.

5.2.3 Configuring GDB and OpenOCD for attachment on boot

You can attach the GDB during target boot only in the following cases:

- TF-A(BL2) boot case;
- TF-A(BL32) boot case;
- OP-TEE boot case;
- U-Boot boot case;
- Linux kernel boot case.

To handle the cases above, the FSBL image has to be wrapped through the tool stm32wrapper4dbg. This operation allows the debugger to halt the target at the very first instruction of FSBL.

Below information is related to the Android™ distribution
FSBL image is wrapped automatically when using the option `-g` with `flash-device` or `provision-device` (see How to populate boards for Android for more details)

Warning

In these cases, the target board will automatically reboots when the GDB starts.

5.2.3.1 *TF-A(BL2) boot case*

In that case, the GDB breaks in bl2_entrypoint function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 1
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.2 *TF-A(BL32) boot case*

In that case, the GDB breaks in sp_min_entrypoint function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 2
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.3 OP-TEE boot case

In that case, the GDB breaks in _start function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 2
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 1
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.4 U-Boot boot case

In that case, the GDB breaks in in _start function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 3
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.3.5 Linux kernel boot case

In that case, the GDB breaks in stext function.

Select the right configuration in Setup.gdb:

```
# Set debug phase:
#        1: Attach at Cortex-A7 / TF-A(BL2)
#        2: Attach at Cortex-A7 / TF-A(BL32) or OP-TEE
#        3: Attach at Cortex-A7 / U-Boot
#        4: Attach at Cortex-A7 / Linux kernel
set $debug_phase = 4
```

```
#        0: Attach at boot
#        1: Attach running target
set $debug_mode = 0
```

```
# Set debug trusted bootchain:
#        0: TF-A BL2 / TF-A BL32 / U-Boot / Linux kernel
#        1: TF-A BL2 / OP-TEE / U-Boot / Linux kernel
set $debug_trusted_bootchain = 0 or 1 #depending on your software boot chain configuration
```

When this operation is complete, jump to Running OpenOCD and GDB.

5.2.4 Running OpenOCD and GDB

- Prerequisites

Before running OpenOCD and GDB, check that the target board is in the right state.

For all configurations except GDB attachment to a running SSBL (U-Boot), the board has to operate in OpenSTLinux running mode.

In case of attachment to a running SSBL (U-Boot) configuration, the board target must be in U-Boot console mode:

```
#Reboot the target board

#Press any key to stop at U-Boot execution when booting the board
STM32MP> ...
STM32MP> Hit any key to stop autoboot:  0
STM32MP>
```

When you are in the expected configuration, two different consoles must be started: **one for OpenOCD** and **one for GDB**.
The SDK environment must be installed on both console terminals.

- OpenOCD console

```
# First console for starting openocd with configuration file
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> openocd -f <board.cfg>
```

> **Below information is related to the Android™ distribution**
>
> If required, it's possible to rebuild `openocd` for your machine (default available `x86`):

```
PC $> load_openocd
PC $> build_openocd -i
```

Start `openocd` with configuration file in a first console:

```
PC $> cd <Your_Android_distribution_path>
PC $> ./device/stm/stm32mp1-openocd/prebuilt/openocd -s ./device
/stm/stm32mp1-openocd/prebuilt/scripts -f <board.cfg>
```

Possible target configuration files for *<board.cfg>*:

Target board	Adapter	SWD mode board.cfg	JTAG mode board.cfg
STM32MP157C-EV1	ST-LINK *	board /stm32mp15x_ev1_stlink_swd.cfg	board /stm32mp15x_ev1_stlink_jtag.cfg
STM32MP157C-EV1	U-LINK2	board /stm32mp15x_ev1_ulink2_swd.cfg	board /stm32mp15x_ev1_ulink2_jtag.cfg
STM32MP157C-EV1	J-LINK	board /stm32mp15x_ev1_jlink_swd.cfg	board/stm32mp15x_ev1_jlink_jtag.cfg
STM32MP157X-DK2	ST-LINK *	board/stm32mp15x_dk2.cfg	⊗**

** Both v2 and v3 are supported.*

*** JTAG wires are not connected in DK2.*

Note: It is recommended to use SWD, which is faster than JTAG.

- GDB console

Warning

The GDB must be executed from the directory where the scripts have been installed (i.e. $HOME /gdbscripts/)

```
# Second console for starting the GDB
PC $> cd $HOME/gdbscripts/
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> $GDB -x Setup.gdb
```

Below information is related to the Android™ distribution

Take Care: Ensure the GCC toolchain is installed using `bspsetup` command in your Distribution Package

Start `GDB` in a second console:

```
PC $> cd $HOME/gdbscripts/
PC $> <Your_Android_distribution_path>/prebuilts/gcc/linux-x86
/arm/<toolchain_version>-x86_64-arm-eabi/bin/arm-eabi-gdb -
x=$HOME/gdbscripts/Setup.gdb
```

5.2.5 To know more about Linux kernel debug with GDB

Please refer to Debugging the Linux kernel using the GDB.

5.2.6 Access to STM32MP registers

5.2.6.1 Using gdb command line

- The following monitoring commands can be used to read a register value:

```
(gdb) monitor mdb <phys_address> [count] #Display target memory as 8-bit bytes
(gdb) monitor mdh <phys_address> [count] #Display target memory as 16-bit bytes
(gdb) monitor mdw <phys_address> [count] #Display target memory as 32-bit bytes
```

For example: Read RCC_MP_APB1ENSETR register on STM32MP1 to check RCC APB1 peripheral enable status.

```
(gdb) monitor mdw phys 0x50000a00 #full 32bits value result requested
0x50000a00: 00010000 --> UART4 is enable as explained in STM32MP15 reference manuals
```

- The following monitoring commands can be used to set a register value:

```
(gdb) monitor mwb <phys_address> <value> [count] #Write byte(s) to target memory
(gdb) monitor mwh <phys_address> <value> [count] #Write 16-bit half-word(s) to target
memory
(gdb) monitor mww <phys_address> <value> [count] #Write 32-bit word(s) to target memory
```

For example: Write RCC_MP_APB1ENCLRR register on STM32MP1 to clear the UART4 RCC of APB1 peripheral, then reenable it:

```
(gdb) monitor mww phys 0x50000a04 0x00010000 #full 32bits value given

# You can then check that UART4 is disable by reading the RCC_MP_APB1ENSETR register:
(gdb) monitor mdw phys 0x50000a00
0x50000a00: 00000000

# You can also check that the console is disabled by going on running the GDB
(gdb) c

# When the GBD has stopped running, you can re-enable UART4 RCC:
(gdb) monitor mww phys 0x50000a00 0x00010000
```

5.2.6.2 Using CMSIS-SVD environment

The CMSIS-SVD environment is useful to get detailed information on registers, such as name and bit descriptions.

It is based on python scripts and svd files which contain the description of all registers.

Refer to CMSIS-SVD environment and scripts for more details.

5.3 Debug Cortex-M4 firmware with GDB

The Arm Cortex-M4 core firmware can also be debugged using the GDB in command line (without IDE).

Either engineering boot mode and production boot mode are supported.

Please refer to the **Hardware Description** (Category:STM32 MPU boards) of your board for information on the Boot mode selection switch.

5.3.1 Debug Cortex-M4 firmware in engineering boot mode

As in previous chapter Running OpenOCD and GDB, both OpenOCD and GDB have to be started on separate consoles.

OpenOCD has to be executed in the same way as in the mentionned chapter.

GDB, instead, has to be executed with a different command line, without the script but with the path of ELF file containing the firmware to be debugged:

```
# Second console for starting the GDB
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> $GDB $HOME/path/to/file.elf
```

Then, the following commands has to be typed in the GDB console to start the execution of a firmware and halt it at the beginning of main():

```
(gdb) target extended-remote localhost:3334
(gdb) load
(gdb) thbreak main
(gdb) continue
```

Once the execution halts at main(), GDB will return the prompt allowing the debug of the firmware.

5.3.2 Debug Cortex-M4 firmware in production boot mode

In production mode the firmware is started by Linux, independently from GDB. Nevertheless, GDB can set a breakpoint before the firmware is started by Linux; thus the firmware will halt at the breakpoint, allowing GDB to debug the firmware.

Please check in Linux remoteproc framework overview how to start and stop a firmware using Linux remoteproc framework.

At first, verify that no firmware is running on Cortex-M4 or, eventually, stop it.

Then, start OpenOCD and GDB as in previous chapter Debug Cortex-M4 firmware in engineering boot mode.

Type the following commands in the GDB console:

```
(gdb) target extended-remote localhost:3334
(gdb) thbreak main
(gdb) continue
```

Finally, let Linux start the firmware. The firmware execution will halt at main() and GDB will return the prompt allowing the debug of the firmware.

Warning

Use on GDB command line the exact same ELF file that is used by Linux to read and run the

firmware. If the files are not the same, the symbols on GDB will not match the firmware in execution.

5.4 Debug Linux application with gdbserver

5.4.1 Enable debug information

Once your program is built using the sdk toolchain, make sure that the **-g** option is enabled to debug your program and add the necessary debug information.

*Note: If an issue occurs during debugging, you can also force gcc to "not optimize code" using the **-O0** option.*

- Example of a simple test program build:

```
PC $> $CC -g -o myappli myappli.c
```

- Example based on Hello World: refer to "hello world" user space example

Edit and update the makefile for the user space example:

```
...
# Add / change option in CFLAGS if needed
-# CFLAGS += <new option>
+ CFLAGS += -g
...
```

5.4.2 Remote debugging using gdbserver

In this setup, an **ethernet link must be set between the host PC and the target board**.

Once your program is installed on the target (using ssh or copied from an SDcard), you can start debugging it.

- On target side: based on "Hello world" user space example

```
Board $> cd /usr/local/bin
Board $> ls
   hello_world_example
Board $> gdbserver host:1234 hello_world_example
  Process main created; pid = 11832 (this value depends on your target)
  Listening on port 1234
```

- Your target waits for remote PC connection, and then starts debugging.
- Launch the GDB command from your source file folder (easier source loading)

The SDK environment must be installed.

```
PC $> cd <source file folder path>
PC $> ls
    hello_world_example  hello_world_example.c  hello_world_example.o
kernel_install_dir  Makefile
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
PC $> $GDB
   GNU gdb (GDB) X.Y.Z
   ...
   This GDB was configured as "--host=x86_64-ostl_sdk-linux --target=arm-ostl-linux-
gnueabi".
   ...
(gdb)
```

- Connect to the target and load the source file:

```
(gdb) target remote <IP_Addr_of_Board>:1234
  Remote debugging using <IP_Addr_of_Board>:1234
  Reading /home/root/test from remote target...
  ...
(gdb) break 16 (line number in the source file)
(gdb) continue
```

- The target program breaks on the breakpoint. Proceed until the end of the program:

```
(gdb) continue
  Continuing.
  [Inferior 1 (process 16204) exited normally]
(gdb) quit
```

5.5 User interface application

5.5.1 Text user interface (TUI) mode

This user interface mode is the first step before using the graphical UI as GDBGUI or DDD.

The TUI can be very useful to map source code with instruction.

Please go through the online documentation [2][3].

5.5.2 Debugging with GDBGUI

Please refer to the dedicated gdbgui article.

5.5.3 Debugging with DDD

GNU DDD is a graphical front-end for command-line debuggers. Please refer to dedicated web page for details[4].

5.5.4 Debugging with IDE

Please refer to STM32CubeIDE.

6 To go further

6.1 Useful GDB commands

When using the GDB in command line mode, it is important to know some basic GDB commands, such as run software, set breakpoints, execute step by step, print variables and display information.

Please refer to GDB commands article.

6.2 Core dump analysis using GDB

The core dump generated for an application crash can be analysed by using the GDB.

Developer Package components, such as SDK and symbol files, must installed. Please refer to STM32MP1 Developer Package .

The symbol file containing the debug symbol of the application in which the crash occurred must also be available.

- First enable the SDK environment:

```
PC $> source <Your_SDK_path>/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

- Then play the core dump with GDB:

```
PC $> $GDB <path_to_the_binary> <path_to_the_core_dump_file>
```

6.3 Tips

- Managing multiple debug setups

To manage multiple debug setups for different software versions, create different Path_env.gdb files with different names, and call the expected file in the Setup.gdb file:

```
...
#######################################################################

# Set environment configuration
#Path_env.gdb
source Path_env_dk2_18_12_14.gdb

#######################################################################
...
```

7 References

- 1.01.1 https://www.gnu.org/software/gdb
- https://sourceware.org/gdb/onlinedocs/gdb/TUI.html#TUI
- https://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_19.html
- https://www.gnu.org/software/ddd
- Useful external links

Document link	Document Type	Description
Using kgdb, kdb and the kernel debugger internals	User Guide	KGDB documentation guide
Welcome to the GDB Wiki	User guide	GDB Wiki
Building GDB and GDBserver for cross debugging	User Guide	Explain how to build gdb for target and host
A GDB Tutorial with Examples	Training	Debugging a simple application

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

(Software)Integrated development/design/debugging environment

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Cortex®

Linux® is a registered trademark of Linus Torvalds.

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

Serial Wire Debug

Board support package

Trusted Firmware for Arm Cortex-A

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

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

Open Portable Trusted Execution Environment

Boot Loader stage 2

Boot Loader stage 3-2

Second Stage Boot Loader

Operating System

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

First Stage Boot Loader

Reset and Clock Control

Cortex Microcontroller Software Interface Standard

Executable and linkable file

User Interface

Stable: 01.03.2021 - 10:54 / Revision: 01.03.2021 - 10:53

A quality version of this page, approved on *1 March 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:

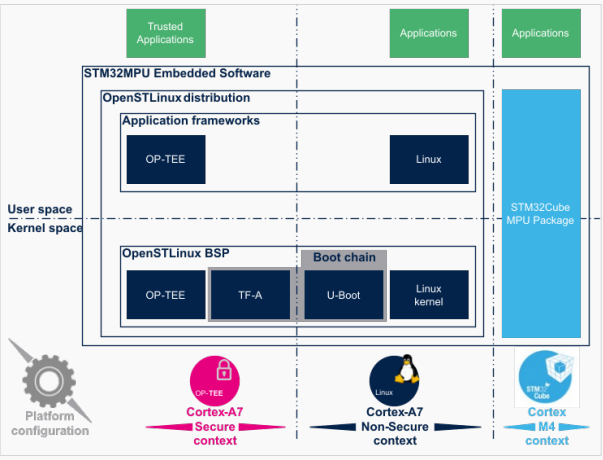
- source file tree structure
- description of CONFIG defines
- instructions for building U-Boot
- brief description of the Hush shell
- list of common environment variables

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

Warning

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

Warning

SPL cannot be used for product development.

SPL is provided only as an example of the simplest SSBL 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:

- 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.

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 (partition named "ssbl" for ST boards).
- 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
```

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:

- bootargs
- kernel + device tree + ramdisk files (optional)
- FIT image

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

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).

The U-Boot generated files when TF-A is used as FSBL, with or without OP-TEE:

- **u-boot.stm32** : U-Boot binary with STM32 image header, loaded by TF-A

The STM32 image format (*.stm32) is managed by mkimage U-Boot tools and Signing_tool. It is requested by ROM code and TF-A (see STM32 header for binary files for details).

The files used to debug with gdb are

- u-boot : elf file for U-Boot

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)

Serial Peripheral Interface

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