U-Boot overview

Stable: 10.10.2019 - 17:16 / Revision: 10.10.2019 - 13:34

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

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

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

2 U-Boot overview

The same U-Boot source can generate two pieces of firmware used in SPL and U-Boot STM32 MPU boot chain:

- Trusted boot chain: TF-A as FSBL and U-Boot as SSBL
- Basic boot chain: SPL as FSBL and U-Boot as SSBL

Zoom out to STM32MPU Embedded Software
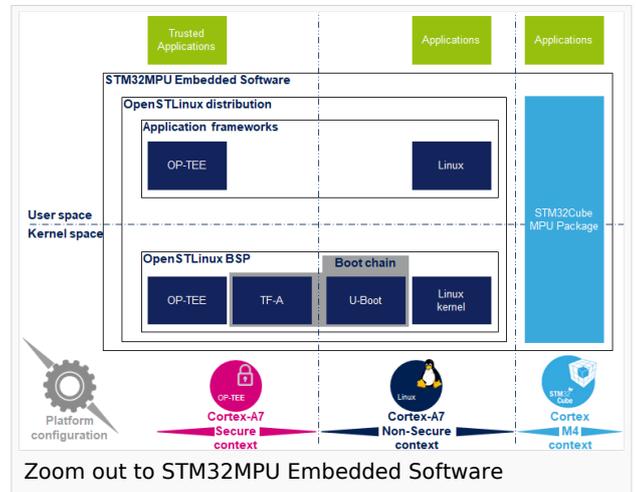
> ⚠️ **The basic boot chain cannot be used for product development (see Boot chains overview for details).**

It is provided only as an example of the simplest SSBL and 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. They apply to:

- power
- secure access to registers
- limited features (STM32CubeProgrammer / boot from NAND Flash memory).

No fix is planned for these limitations.

2.1 SPL: FSBL for basic boot

The **U-Boot SPL** or **SPL** is the first stage bootloader (FSBL) for the basic boot chain.
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.

2.2 U-Boot: SSBL

U-Boot is the default second-stage bootloader (SSBL) for STM32 MPU platforms. It is used both for trusted and basic boot chains. 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)

2.3 SPL phases

SPL executes the following main phases 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[1]: README.falcon).

2.4 U-Boot phases

U-Boot executes the following main phases 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 and SPL 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 in the build directory, with the same makefile target. .
Proceed as follows:

- Select a pre-defined 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[2]
  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 SPL and U-Boot binaries:

- By default, it is appended at the end of the code (CONFIG_OF_SEPARATE).
- It is embedded in the U-Boot binary (CONFIG_OF_EMBED). This is 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: arch/arm/dts/stm32mp*.dts .

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

or provide a precompiled device tree blob (using 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 define 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 each node** used in SPL or in U-Boot before relocation and for each used handle (clock, reset, pincontrol).

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) 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 - boots default, i.e., 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:

- for *e*•MMC/SD card boot (CONFIG_ENV_IS_IN_EXT4), in the bootable ext4 partition "bootfs" in in file CONFIG_ENV_EXT4_FILE="uboot.env".
- for NAND boot (CONFIG_ENV_IS_IN_UBI), in the two UBI volumes "config" (CONFIG_ENV_UBI_VOLUME) and "config_r" (CONFIG_ENV_UBI_VOLUME_REDUND).
- for NOR boot (CONFIG_ENV_IS_IN_SPI_FLASH), in 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:

- for serial/usb: execute the `stm32prog` command.
- for mmc boot (e•MMC, SD card), boot only on the same device (bootcmd_mmc...).
- for nand boot, boot with on ubifs partition on nand (bootcmd_ubi0).
- for nor boot, use the default order e•MMC (SDMMC 1)/ NAND / SD card (SDMMC 0) / SDMMC2 (the default bootcmd: distro_bootcmd).

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

You can then change this configuration:

- either permanently in your board file (default environment by CONFIG_EXTRA_ENV_SETTINGS or change CONFIG_BOOTCOMMAND value) 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 of 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 a **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)
        - from GITHUB[3], with `git` command

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

        - from the Mainline U-Boot in official GIT repository [4]

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

5.1.1 ARM cross compiler

A cross compiler [5] 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:
    - gcc v8 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-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-
```

5.2 Compilation

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

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

Optionally **KBUILD_OUTPUT** can be used to change the output directory to compile several targets or not to compile in the source directory. For example:

```
PC $> export KBUILD_OUTPUT=../build/basic
```

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

```
PC $> export DEVICE_TREE=stm32mp157c-ev1
```

Examples from STM32MP15 U-Boot:

Three configurations are supported for STM32MP15x lines 🛈:

- **stm32mp15_trusted_defconfig**: trusted boot chain, U-Boot (without SPL) is unsecure and uses Secure monitor from TF-A
- **stm32mp15_optee_defconfig**: trusted boot chain, U-Boot (without SPL) is unsecure and uses Secure monitor from SecureOS = OP-TEE
- stm32mp15_basic_defconfig: basic boot chain, with an SPL as FSBL, U-BOOT is secure and installs monitor with PSCI

The board diversity is only managed with the device tree.

```
PC $> export KBUILD_OUTPUT=../build/trusted
PC $> make stm32mp15_trusted_defconfig
PC $> make DEVICE_TREE=stm32mp157c-<board> all
```

```
PC $> export KBUILD_OUTPUT=../build/optee
PC $> export DEVICE_TREE=stm32mp157c-<board>
PC $> make stm32mp15_optee_defconfig
PC $> make all
```

```
PC $> make stm32mp15_basic_defconfig
PC $> make DEVICE_TREE=stm32mp157c-<board>  all
```

Use help to list other targets:

```
PC $> make help
```

5.3 Output files

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

Two binary formats are used for stm32mp devices:

- STM32 image format (*.stm32), 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).
- uImage (*.img) format, file including a U-Boot header, managed by SPL and U-Boot (for kernel load)

The U-Boot generated files are the following

- For **Trusted boot chain** (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

- For **Basic boot chain** (SPL is used as FSBL)
  - **u-boot-spl.stm32** : SPL binary with STM32 image header, loaded by ROM code
  - **u-boot.img** : U-Boot binary with uImage header, loaded by SPL

The files used to debug with gdb are

- u-boot : elf file for U-Boot
- spl/u-boot-spl : elf file for SPL

6 References

1. ↑ https://www.denx.de/wiki/pub/U-Boot/MiniSummitELCE2013/2013-ELCE-U-Boot-Falcon-Boot.pdf
2. ↑ https://en.wikipedia.org/wiki/Xconfig
3. ↑ https://github.com/STMicroelectronics/u-boot
4. ↑ https://gitlab.denx.de/u-boot/u-boot.git or https://github.com/u-boot/u-boot
5. ↑ https://en.wikipedia.org/wiki/Cross_compiler

Random Access Memory

Read Only Memory