# SDK for OpenSTLinux distribution

# Contents



Stable: 07.01.2021 - 12:41 / Revision: 07.01.2021 - 12:35

Stable: 07.01.2021 - 12:41 / Revision: 07.01.2021 - 12:38

A quality version of this page, approved on *7 January 2021*, was based off this revision.

Contents

# 1 Article purpose

This article aims to give **general information** about the software development kit (SDK) for the OpenSTLinux distribution.

**Information**

To install and use efficiently the last release of the OpenSTLinux SDK, please read the Developer Package article relative to the Series of your STM32 microprocessor: Category:Developer Package

# 2 Introduction

The **software development kit (SDK)** for the OpenSTLinux distribution is a customization of the Yocto SDK[1], which provides a **stand-alone cross-development toolchain** and libraries tailored to the contents of a specific image. The OpenSTLinux SDK is part of the STM32MPU Embedded Software Developer Package.

The **SDK might be generated**, through the STM32MPU Embedded Software Distribution Package, during the compilation of a software release, which guarantees the alignment of this SDK with the software images (binaries) built for the Starter Package of the STM32MPU Embedded Software: see SDK development cycle model.

It provides a more "traditional" toolchain experience than the full Yocto project (OpenEmbedded) development environment available through the Distribution Package of the STM32MPU Embedded Software.

**It simplifies the workflow for application developers**: it has no dependency on the Yocto project used for its generation (Distribution Package), and can be installed on any host machine. Note that many SDKs can coexist on the same host machine.

# 3 Why use the SDK, and how?

The OpenSTLinux SDK gives developers an efficient development cycle (compilation, deployment on target, and debug).

Using this SDK, developers take advantage of the Yocto project development environment (to quickly develop, deploy and test applications, or any other piece of software, as part of images running hardware), without having to understand all the Yocto project mechanisms that might seem somewhat complex.

## 3.1 SDK development cycle model

A developer can install the SDK on a machine (host PC), and use it to develop within any piece of software (for example, an application, kernel drivers or kernel modules).

Basically, the developer has to:

- get the software images (binaries) of the software release associated with the SDK (see Starter Package)
- install the SDK for the targeted hardware (see SDK installation)
- run the SDK environment setup script (see SDK startup)
- develop and test the piece of software

When the development is finished (the source code is ready to be shared with other developers), it should be integrated into the whole software. For this, the Distribution Package must be used.

Through the Distribution Package, new images (binaries) and a new SDK are generated (see How to create an SDK for OpenSTLinux distribution).

# 4 SDK content

The OpenSTLinux SDK is based on the **standard** Yocto project SDK.

A standard SDK consists of the following:

- a cross-development toolchain: this toolchain contains a compiler, linker, debugger, and various miscellaneous tools
- libraries, headers, and symbols (target and native sysroots): the libraries, headers, and symbols are specific to the image (that is, they match the image)
- an environment setup script: this *.sh file, once run, sets up the cross-development environment by defining variables and preparing it for SDK use

## 4.1 Cross-development toolchain

The cross-development toolchain consists of a cross-compiler, a cross-linker and a cross-debugger that are used to:

- develop user-space applications for targeted hardware
- modify a software component that already exists in the images, and that is delivered as source code in the Developer Package (for example the Linux kernel or U-Boot)

This cross-development toolchain is created by running a toolchain installer script (see SDK installation).

It works with a matching target sysroot (see below).

## 4.2 Native and target sysroots

The native and target sysroots contain the required headers and libraries for generating binaries that run on the target architecture.

The target sysroot is based on the target root file system image that is built through the Distribution Package of the STM32MPU Embedded Software and uses the same metadata configuration as that used to build the cross-toolchain.

For any software baseline, this process guarantees the alignment between:

- the content (source code) of the Distribution Package
- the target root file system image (binary) of the Starter Package
- the target sysroot (headers and libraries) of the Developer Package
- the configuration of the cross-toolchain of the Developer Package

# 5 SDK installation

The OpenSTLinux SDK is installed on the host development machine by running the *.sh installation script.

The tarball file (*SDK-[...].tar.xz*) that contains this script is named as follows: **SDK-<host machine>-<version>.tar.xz** where:

| | |
|---|---|
| <host machine> | Host machine on which the SDK is installed:<br>• x86_64 (only 64-bit host machines are supported) |
| <version> | Software release version; example:<br>• openstlinux-5.4-dunfell-mp1-20-11-12 |

Example:

- en.SDK-x86_64-stm32mp1-openstlinux-5-4-dunfell-mp1-20-11-12.tar.xz

The steps for the OpenSTLinux SDK installation, are:

- Download, on the host machine, the SDK tarball file (*SDK-[...].tar.xz*)
- Decompress the tarball file

```
$ tar xvf en.SDK-[...].tar.xz
```

- The **installation script** is named:

  **<image>-<distro>-<machine>-<host machine>-toolchain-<Yocto release>-<version>.sh**

  where:

| | |
|---|---|
| <image> | Image name; example:<br>• st-image-weston |
| <distro> | Distribution name; example:<br>• openstlinux-weston |
| <machine> | Machine name; example:<br>• stm32mp1 |
| <host machine> | Host machine on which the SDK is installed:<br>• x86_64 (only 64-bit host machines are supported) |
| <Yocto release> | Release number of the Yocto Project; example:<br>• 3.1 (aka dunfell) |
| <version> | Software release version; example:<br>• openstlinux-5.4-dunfell-mp1-20-11-12 |

  Example:

  - st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-openstlinux-5.4-dunfell-mp1-20-11-12.sh

- If necessary, change the permissions on the installation script so that it is executable:

```
$ chmod +x <image>-<distro>-<machine>-<host machine>-toolchain-<Yocto release>-
<version>.sh
```

- The SDK is self-contained and by default is installed into */opt/st/<machine>/<Yocto release>-<version>*

  Example:

  - /opt/st/stm32mp1/3.1-openstlinux-5.4-dunfell-mp1-20-11-12

  However, running the installation script with the *-d* option allows an installation directory to be chosen

  Check that the write permissions in the installation directory (either the default one, or the customized one) are granted

**Information**

Recommendation: for an STM32MPU Embedded Software release, install the software image from the Starter Package, the SDK and the source codes from the Developer Package in the same top directory; indeed, these packages are linked

- Run the installation script

```
$ ./<image>-<distro>-<machine>-<host machine>-toolchain-<Yocto release>-<version>.sh
```

Example (with an installation directory */local/SDK/<Yocto release>-<version>* different from the default one)

```
$ ./st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-2.6-openstlinux-20-02-19.
sh -d /local/SDK/2.6-openstlinux-20-02-19
ST OpenSTLinux - Weston - (A Yocto Project Based Distro) SDK installer version 2.6-
openstlinux-20-02-19
==========================================================================================
============================
You are about to install the SDK to "/local/SDK/2.6-openstlinux-20-02-19". Proceed[Y/n]? Y
Extracting SDK.....................................................................
.........................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the
environment setup script e.g.
 $ . /local/SDK/2.6-openstlinux-20-02-19/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-
linux-gnueabi
```

The OpenSTLinux SDK install is now complete.

Refer to Standard SDK directory structure for details of the resulting directory structure of the installed SDK.

# 6 SDK startup

To use an installed SDK, its environment setup script must be run.

This setup script is located in the SDK installation directory (per default, */opt/st/<machine>/<Yocto release>-<version>*).

It must be run once in each new working terminal.

This **environment setup script** is named:

**environment-setup-<target>-<distro>-linux-gnueabi**

Where:

| | |
|---|---|
| <target> | Target architecture for cross-toolchain; example:<br>• cortexa7t2hf-neon-vfpv4 |
| <distro> | Distribution name; example:<br>• openstlinux_weston |

Example

- environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi

- Run the environment setup script

```
 $ source <SDK installation directory path>/environment-setup-<target>-<distro>-linux-gnueabi
```

Example: here, the SDK installation directory (*/local/SDK/<Yocto release>-<version>*) is different from the default one

```
 $ source /local/SDK/3.1-openstlinux-20-11-12/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

- Many environment variables are then defined:

```
SDKTARGETSYSROOT - the path to the sysroot used for cross-compilation
PKG_CONFIG_PATH - the path to the target pkg-config files
CONFIG_SITE - a GNU autoconf site file preconfigured for the target
CC - the minimal command and arguments to run the C compiler
CXX - the minimal command and arguments to run the C++ compiler
CPP - the minimal command and arguments to run the C preprocessor
AS - the minimal command and arguments to run the assembler
LD - the minimal command and arguments to run the linker
GDB - the minimal command and arguments to run the GNU Debugger
STRIP - the minimal command and arguments to run 'strip', which strips symbols
RANLIB - the minimal command and arguments to run 'ranlib'
OBJCOPY - the minimal command and arguments to run 'objcopy'
OBJDUMP - the minimal command and arguments to run 'objdump'
AR - the minimal command and arguments to run 'ar'
NM - the minimal command and arguments to run 'nm'
TARGET_PREFIX - the toolchain binary prefix for the target tools
```

```
      CROSS_COMPILE - the toolchain binary prefix for the target tools
      CONFIGURE_FLAGS - the minimal arguments for GNU configure
      CFLAGS - suggested C flags
      CXXFLAGS - suggested C++ flags
      LDFLAGS - suggested linker flags when you use CC to link
      CPPFLAGS - suggested preprocessor flags
```

The OpenSTLinux SDK is started.

# 7 References

- Yocto Project Application Development and Extensible Software Development Kit (eSDK)

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

Linux® is a registered trademark of Linus Torvalds.

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

also known as

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

Stable: 17.06.2020 - 15:26 / Revision: 16.01.2020 - 13:43

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to a Developer Package (whatever the microprocessor device and the board).

The Developer Package is specified in the Which Package better suits your needs article.

# Pages in category "Developer Package"

The following 3 pages are in this category, out of 3 total.

- How to cross-compile with the Developer Package
- STM32MP1 Developer Package
- STM32MP1 Developer Package for Android

Stable: 17.06.2020 - 15:26 / Revision: 16.01.2020 - 13:44

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to a Distribution Package (whatever the microprocessor device and the board).

The Distribution Package is specified in the Which Package better suits your needs article.

# Pages in category "Distribution Package"

The following 6 pages are in this category, out of 6 total.

- How to add a customer application
- How to create your own machine
- How to cross-compile with the Distribution Package
- How to customize the Linux kernel
- STM32MP1 Distribution Package
- STM32MP1 Distribution Package for Android

Stable: 17.06.2020 - 15:27 / Revision: 16.01.2020 - 13:43

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to a Starter Package (whatever the microprocessor device and the board).

The Developer Package is specified in the Which Package better suits your needs article.

# Pages in category "Starter Package"

The following 3 pages are in this category, out of 3 total.

- STM32MP15 Discovery kits - Starter Package
- STM32MP15 Evaluation boards - Starter Package
- STM32MP15 Evaluation boards - Starter Package for Android

Stable: 17.11.2020 - 16:27 / Revision: 30.07.2020 - 08:18

A quality version of this page, approved on *17 November 2020*, was based off this revision.

When an OpenSTLinux distribution has been modified, it is pertinent to build a new software development package that integrates the modifications, and to redistribute this SDK to developers (see SDK development cycle model).

# 1 Prerequisites

The Distribution Package relative to your STM32 microprocessor Series is installed: Category:Distribution Package.

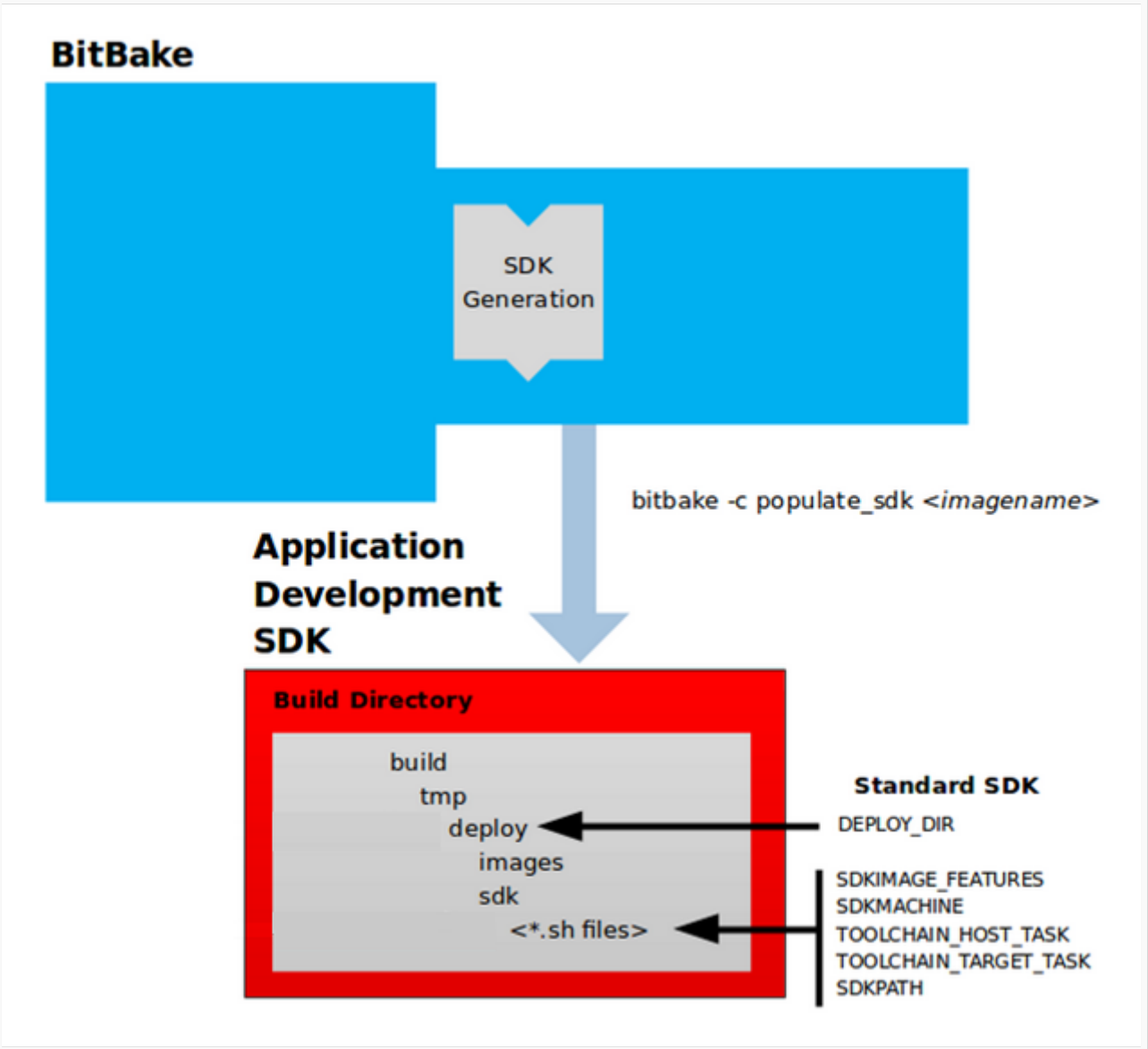
On the installation:

- some pieces of software might have been modified or integrated
- the build environment script has been executed
- the selected image has been rebuilt

# 2 SDK generation

The OpenEmbedded build system uses BitBake to generate the software development package (SDK) installation script.

For more information about the SDK, see the SDK for OpenSTLinux distribution article.




Standard SDK generation
Source: Yocto Project[1]

The *do_populate_sdk* task helps to create the standard SDK and handles two parts: a target part and a host part. The target part is built for the target hardware and includes libraries and headers. The host part is the part of the SDK that runs on the host machine.

- Check that the build environment script has been executed, and that the current directory is the build directory of the OpenSTLinux distribution (for example, *openstlinux-20-06-24/build-openstlinuxweston-stm32mp1')*
- Generate the SDK installation files (including the installation script) for a standard SDK with the following command :

```
PC $> bitbake -c populate_sdk <image>
```

Where:

| | |
|---|---|
| <image> | Image name;<br>example:<br>• st-image-weston |

Example:

**PC $>** bitbake -c populate_sdk st-image-weston

- The SDK installation files (*<image>-<distro>-<machine>-<host machine>-toolchain-<Yocto release>-snapshot.**) are written to the *deploy/sdk* directory inside the build directory *build-<distro>-<machine>* as shown in the figure above

Where:

| | |
|---|---|
| <host machine> | Host machine on which the SDK is generated<br>• x86_64 (64-bit host machine; this is the only supported value) |
| <Yocto release> | Release number of the Yocto Project; example:<br>• 3.1 (aka dunfell) |

Example

```
PC $> ls tmp-glibc/deploy/sdk/
st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-snapshot.host.manifest
st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-snapshot.license
st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-snapshot-license_content.
html
st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-snapshot.sh
st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-snapshot.target.manifest
st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-snapshot.testdata.json
```

The main final output is the cross-development toolchain installation script (*.sh* file), which includes the environment setup script.

Note that several OpenEmbedded variables exist that help configure these files. The following list shows the variables associated with a standard SDK:

```
    DEPLOY_DIR: points to the deploy directory.
    SDKMACHINE: specifies the architecture of the machine on which the cross-development
tools are run to create packages for the target hardware.
    SDKIMAGE_FEATURES: lists the features to include in the "target" part of the SDK.
    TOOLCHAIN_HOST_TASK: lists packages that make up the host part of the SDK (that is,
the part that runs on the SDKMACHINE). This variable allows packages other than the
default ones to be added.
    TOOLCHAIN_TARGET_TASK: lists packages that make up the target part of the SDK (that
is, the part built for the target hardware).
    SDKPATH: Defines the default SDK installation path offered by the installation script.
```

# 3 Reference list

- http://www.yoctoproject.org/documentation

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

also known as

Stable: 15.04.2020 - 14:15 / Revision: 15.04.2020 - 14:12

A quality version of this page, approved on *15 April 2020*, was based off this revision.

This article describes the structure of the OpenSTLinux standard SDK installation directory:

```
SDK
├── environment-setup-<target>-<distro>-linux-gnueabi  Environment setup script
├── site-config-<target>-<distro>-linux-gnueabi        Configuration file for the target
├── sysroots                                           Root file systems
│   ├── <target>-<distro>-linux-gnueabi                Target sysroot (libraries,
headers, and symbols)
│   │   ├── bin
│   │   ├── boot
│   │   ├── config
│   │   ├── dev
│   │   ├── etc
│   │   ├── home
│   │   ├── lib
│   │   ├── media
│   │   ├── mnt
│   │   ├── proc
│   │   ├── run
│   │   ├── sbin
│   │   ├── sys
│   │   ├── tmp
│   │   ├── usr
│   │   └── var
│   └── <host machine>-<distro>_sdk-linux              Native sysroot (libraries,
headers, and symbols)
│       ├── bin
│       ├── environment-setup.d
│       ├── etc
│       ├── lib
│       ├── sbin
│       ├── usr
│       └── var
└── version-<target>-<distro>-linux-gnueabi            Version file for the target
```

Where:

| | |
|---|---|
| <target> | Target architecture for cross-toolchain; examples (non exhaustive list):<br>• cortexa7hf-neon-vfpv4 |
| <distro> | Distribution; examples (non-exhaustive list):<br>• openstlinux_weston |

| | |
|---|---|
| <host machine> | Host machine on which the SDK is installed; examples (non-exhaustive list):<br>• x86_64 (64-bit host machine) |

The installed SDK consists of:

- an environment setup script for the SDK
- a configuration file for the target
- a version file for the target
- the root file systems (sysroots) needed to develop objects for the target system

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



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:

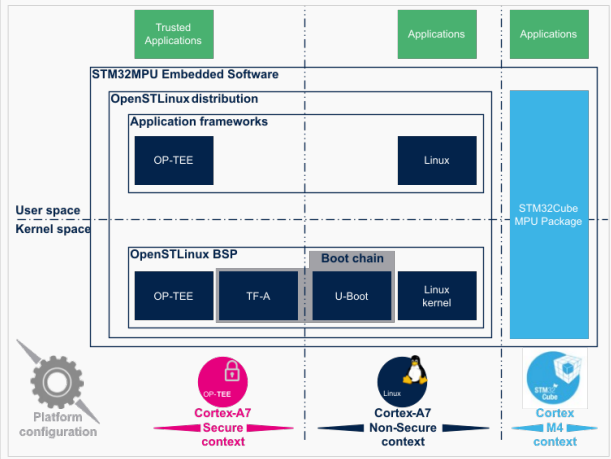
- source file tree structure
- description of CONFIG defines
- instructions for building U-Boot
- brief description of the Hush shell
- list of common environment variables

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