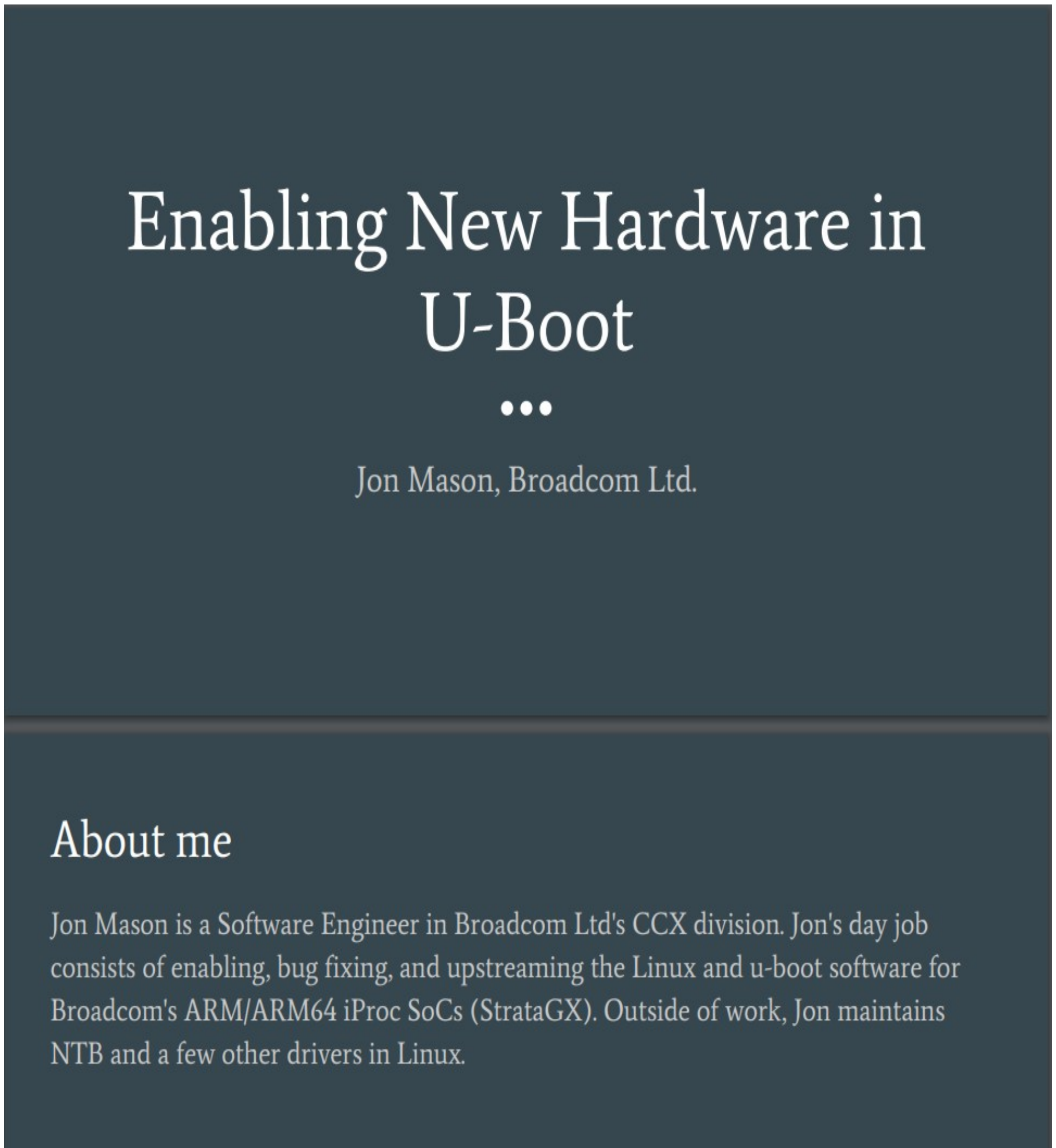
Bootloaders

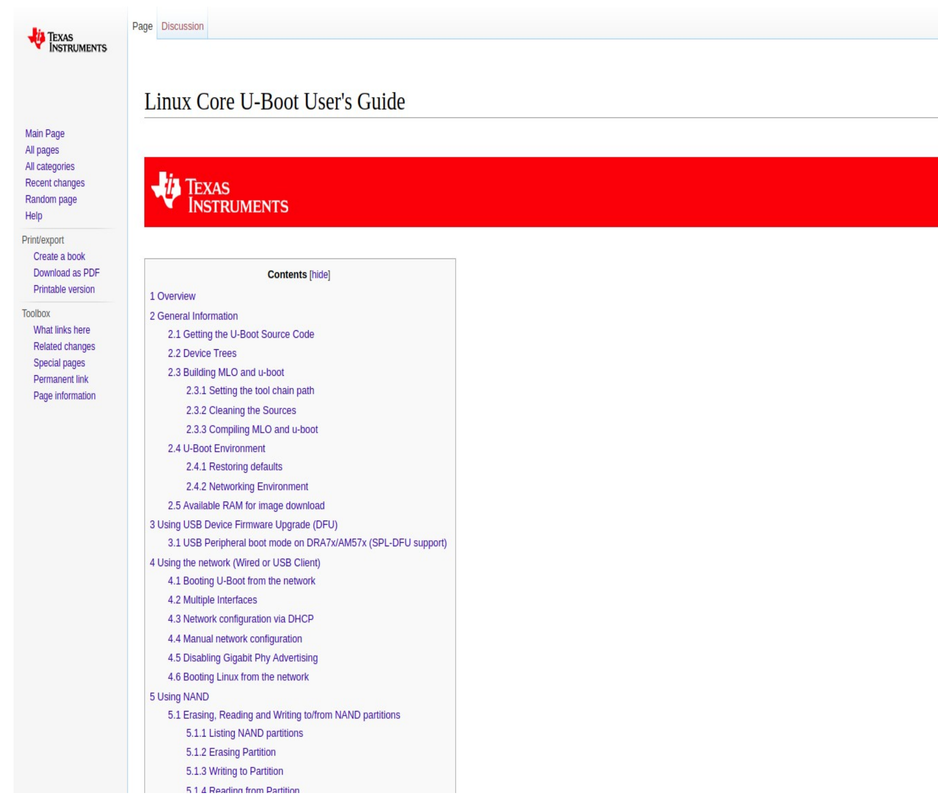
With U-Boot as an example

From https://training.ti.com/bootloading-101

A bootloader can be as simple or as complex as the author wants it to be.

Who cares about this kind of software?





Hardware vendors supply board support packages (BSP) that include bootloaders

Uses of boot-loaders

- Boot a larger OS (e.g. linux) from disk to RAM
  - Initialize RAM
  - Initialize communication with host machine (UART)
    - Needed if embedded platform doesn't have SD card/ flash to hold the kernel image
    - To change configuration parameters (if needed)
  - Initialize communication with a network server
    - Needed if remote updates are needed

    ....

    ....
- Write **bare metal code** for embedded platforms

Uses of boot-loaders

- Boot a larger OS (e.g. linux) from disk to RAM
- Write bare metal code for embedded platforms

Copy from Network/Flash (different kinds of flash memory) to RAM

Image	File Name	RAM Address	Flash
u-boot	u-boot	u-boot_addr_r	u-boot_addr
Linux kernel	bootfile	kernel_addr_r	kernel_addr
device tree	fdtfile	fdt_addr_r	fdt_addr
ramdisk	ramdiskfile	ramdisk_addr_r	ramdisk_addr

Types of source code in U-Boot

Pure initialization code: This code always runs during U-Boot's own bring-up

Drivers: Code that implements a set of functions, which gives access to a certain piece of hardware. Much of this is found in drivers/, fs/ and others

Commands: Adding commands to the U-Boot shell, and implementing their functionality, typically based upon calls to driver API. These appear as common/cmd_*.c

U-Boot source code directory structure

/arch	Architecture specific files
/arc	Files generic to ARC architecture
/arm	**Files generic to ARM architecture**
/m68k	Files generic to m68k architecture
/microblaze	Files generic to microblaze architecture
/mips	Files generic to MIPS architecture
/nds32	Files generic to NDS32 architecture
/nios2	Files generic to Altera NIOS2 architecture
/openrisc	Files generic to OpenRISC architecture
/powerpc	Files generic to PowerPC architecture
/riscv	Files generic to RISC-V architecture
/sandbox	Files generic to HW-independent "sandbox"
/sh	Files generic to SH architecture
/x86	Files generic to x86 architecture

U-Boot source code directory structure

/api	Machine/arch independent API for external apps
/board	**Board dependent files**
/cmd	**U-Boot commands functions**
/common	**Misc architecture independent functions**
/configs	**Board default configuration files**
/disk	**Code for disk drive partition handling**
/doc	**Documentation (don't expect too much)**
/drivers	**Commonly used device drivers**
/dts	Contains Makefile for building internal U-Boot fdt.
/examples	Example code for standalone applications, etc.
/fs	Filesystem code (cramfs, ext2, jffs2, etc.)
/include	Header Files
/lib	Library routines generic to all architectures
/Licenses	Various license files
/net	Networking code
/post	Power On Self Test
/scripts	Various build scripts and Makefiles
/test	Various unit test files
/tools	Tools to build S-Record or U-Boot images, etc.

While the source code is not too small

You can control what gets compiled based on configuration files

```
$make rpi_3_defconfig
```

Huge Number of hardware specific configurations

```
rijurekha@rijurekha-Inspiron-5567:~/u-boot/u-boot-2017/u-boot-2017.11$ ls configs/
Display all 1191 possibilities? (y or n)
10m50_defconfig                      ge_b450v3_defconfig                  mx28evk_auart_console_defconfig
3c120_defconfig                      ge_b650v3_defconfig                  mx28evk_defconfig
A10-OLinuXino-Lime_defconfig         ge_b850v3_defconfig                  mx28evk_nand_defconfig
A10s-OLinuXino-M_defconfig           geekbox_defconfig                    mx28evk_spi_defconfig
A13-OLinuXino_defconfig              goflexhome_defconfig                 mx31ads_defconfig
A13-OLinuXinoM_defconfig             gose_defconfig                       mx31pdk_defconfig
A20-Olimex-SOM-EVB_defconfig         gplugd_defconfig                     mx35pdk_defconfig
A20-OLinuXino-Lime2_defconfig        gt90h_v4_defconfig                   mx51evk_defconfig
A20-OLinuXino-Lime2-eMMC_defconfig   gurnard_defconfig                    mx53ard_defconfig
A20-OLinuXino-Lime_defconfig         guruplug_defconfig                   mx53cx9020_defconfig
A20-OLinuXino_MICRO_defconfig        gwventana_emmc_defconfig             mx53evk_defconfig
A20-OLinuXino_MICRO-eMMC_defconfig   gwventana_gw5904_defconfig           mx53loco_defconfig
A33-OLinuXino_defconfig              gwventana_nand_defconfig             mx53smd_defconfig
a64-olinuxino_defconfig              h2200_defconfig                      mx6cuboxi_defconfig
adp-ae3xx_defconfig                  h8_homlet_v2_defconfig               mx6dlarm2_defconfig
adp-ag101p_defconfig                 harmony_defconfig                    mx6dlarm2_lpddr2_defconfig
Ainol_AW1_defconfig                  highbank_defconfig                   mx6qarm2_defconfig
alt_defconfig                        hikey_defconfig                      mx6qarm2_lpddr2_defconfig
am335x_baltos_defconfig              hrcon_defconfig                      mx6qsabrelite_defconfig
am335x_boneblack_defconfig           hrcon_dh_defconfig                   mx6sabreauto_defconfig
am335x_boneblack_vboot_defconfig     hsdk_defconfig                       mx6sabresd_defconfig
am335x_evm_defconfig                 huawei_hg556a_ram_defconfig          mx6slevk_defconfig
am335x_evm_norboot_defconfig         Hummingbird_A31_defconfig            mx6slevk_spinor_defconfig
am335x_evm_nor_defconfig             Hyundai_A7HD_defconfig               mx6slevk_spl_defconfig
am335x_evm_spiboot_defconfig         i12-tvbox_defconfig                  mx6sllevk_defconfig
am335x_evm_usbspl_defconfig          ib62x0_defconfig                     mx6sllevk_plugin_defconfig
am335x_hs_evm_defconfig              icnova-a20-swac_defconfig            mx6sxsabreauto_defconfig
am335x_hs_evm_uart_defconfig         iconnect_defconfig                   mx6sxsabresd_defconfig
am335x_igep003x_defconfig            ids8313_defconfig                    mx6sxsabresd_spl_defconfig
am335x_shc_defconfig                 igep0032_defconfig                   mx6ul_14x14_evk_defconfig
```

Huge Number of hardware specific configurations

```
/* Automatically generated - do not edit */
#define CONFIG_SYS_ARCH  "arm"
#define CONFIG_SYS_CPU   "armv7"
#define CONFIG_SYS_BOARD "zynq"
#define CONFIG_SYS_VENDOR "xilinx"
#define CONFIG_SYS_SOC    "zynq"
#define CONFIG_BOARDDIR board/xilinx/zynq
#include <config_cmd_defaults.h>

#include <config_defaults.h>
#include <configs/zynq_zed.h>
#include <asm/config.h>
#include <config_fallbacks.h>
#include <config_uncmd_spl.h>
```

```
#ifndef __CONFIG_ZYNQ_ZED_H
#define __CONFIG_ZYNQ_ZED_H

#define PHYS_SDRAM_1_SIZE (512 * 1024 * 1024)

#define CONFIG_ZYNQ_SERIAL_UART1
#define CONFIG_ZYNQ_GEM0
#define CONFIG_ZYNQ_GEM_PHY_ADDR0       0

#define CONFIG_SYS_NO_FLASH

#define CONFIG_ZYNQ_SDHCI0
#define CONFIG_ZYNQ_QSPI
#define CONFIG_ZYNQ_BOOT_FREEBSD

#include <configs/zynq_common.h>

#endif /* __CONFIG_ZYNQ_ZED_H */
```

Hardware vendors create these config files and add them to the source repo

Board	SD Boot	eMMC Boot	NAND Boot	UART Boot	Ethernet Boot	USB Ethernet Boot	USB Host Boot	SPI Boot
AM335x GP EVM	am335x_evm_defconfig		am335x_evm_defconfig	am335x_evm_defconfig	am335x_evm_defconfig	am335x_evm_defconfig		am335x_evm_spiboot_defconfig
AM335x EVM-SK	am335x_evm_defconfig			am335x_evm_defconfig		am335x_evm_defconfig		
AM335x ICE	am335x_evm_defconfig			am335x_evm_defconfig				
BeagleBone Black	am335x_evm_defconfig	am335x_evm_defconfig		am335x_evm_defconfig				
BeagleBone White	am335x_evm_defconfig			am335x_evm_defconfig				
AM437x GP EVM	am43xx_evm_defconfig		am43xx_evm_defconfig	am43xx_evm_defconfig	am43xx_evm_defconfig	am43xx_evm_defconfig	am43xx_evm_usbhost_boot_defconfig	
AM437x EVM-Sk	am43xx_evm_defconfig						am43xx_evm_usbhost_boot_defconfig	
AM437x IDK	am43xx_evm_defconfig							am43xx_evm_qspiboot_defconfig (XIP)
AM437x ePOS EVM	am43xx_evm_defconfig		am43xx_evm_defconfig				am43xx_evm_usbhost_boot_defconfig	
AM572x GP EVM	am57xx_evm_defconfig			am57xx_evm_defconfig				
AM572x IDK	am57xx_evm_defconfig							
AM571x IDK	am57xx_evm_defconfig							
DRA74x/DRA72x/DRA71x EVM	dra7xx_evm_defconfig	dra7xx_evm_defconfig	dra7xx_evm_defconfig (DRA71x EVM only)					dra7xx_evm_defconfig(QSPI)
K2HK EVM			k2hk_evm_defconfig	k2hk_evm_defconfig	k2hk_evm_defconfig			k2hk_evm_defconfig
K2L EVM			k2l_evm_defconfig	k2l_evm_defconfig				k2l_evm_defconfig
K2E EVM			k2e_evm_defconfig	k2e_evm_defconfig				k2e_evm_defconfig
K2G GP EVM	k2g_evm_defconfig			k2g_evm_defconfig	k2g_evm_defconfig			k2g_evm_defconfig
K2G ICE	k2g_evm_defconfig							
OMAP-L138 LCDK	omapl138_lcdk_defconfig		omapl138_lcdk_defconfig					

Initialization code

U-Boot is one of the first things to run on the processor, and may be responsible for the most basic hardware initialization. On some platforms the processor’s RAM isn’t configured when U-Boot starts running, so the underlying assumption is that U-Boot may run directly from ROM (typically flash memory).

The bring-up process’ key event is hence when U-Boot copies itself from where it runs in the beginning into RAM, from which it runs the more sophisticated tasks (handling boot commands in particular). This self-copy is referred to as “relocation”.

Almost needless to say, the processor runs in “real mode”: The MMU, if there is one, is off. There is no memory translation nor protection. U-Boot plays a few dirty tricks based on this.

Typical stages in initialization code

- Pre-relocation initialization (possibly directly from flash or other kind of ROM)
- Relocation: Copy the code to RAM.
- Post-relocation initialization (from proper RAM).
- Execution of commands: Through autoboot or console shell
- Passing control to the Linux kernel (or other target application)

Typical stages in initialization code

The sequence for the ARM architecture can be deduced from arch/arm/lib/crt0.S, which is the absolutely first thing that runs. This piece of assembly code calls functions as follows (along with some very low-level initializations):

- board_init_f() (defined in e.g. arch/arm/lib/board.c): Calls the functions listed in the init_sequence_f function pointer array (using initcall_run_list()), which is enlisted in this file with a lot of ifdefs. This function then runs various ifdef-dependent init snippets.
- relocate_code()
- coloured_LED_init() and red_led_on() are directly called by crt0.S. Defining these functions allow hooking visible indications of early boot progress.
- board_init_r() (defined in arch/arm/lib/board.c): Runs the initialization as a "normal" program running from RAM. This function never returns. Rather,
- board_init_r() loops on main_loop() (defined in common/main.c) forever. This is essentially the autoboot or execution of commands from input by the command parser (hush command line interpreter).
- At some stage, a command in main_loop() gives the control to the Linux kernel (or whatever was loaded instead).

Secondary Program Loader

The SPL (Secondary Program Loader) boot feature is irrelevant in most scenarios, but offers a solution if U-Boot itself is too large for the platform’s boot sequence. For example, the ARM processor’s hardware boot loader in Altera’s SoC FPGAs can only handle a 60 kB image. A typical U-Boot ELF easily reaches 300 kB (after stripping).

The point with an SPL is to create a very small preloader, which loads the “full” U-Boot image. It’s built from U-Boot’s sources, but with a minimal set of code.

So when U-Boot is built for a platform that requires SPL, it’s typically done twice: Once for generating the SPL, and a second time for the full U-Boot.

The SPL build is done with the CONFIG_SPL_BUILD is defined. Only the pre-location phase runs on SPL builds. All it does is the minimal set of initializations, then loads the full U-Boot image, and passes control to it.

Example boot process in Altera’s Cyclone V SoC FPGA

- The ARM processor loads a hardcoded boot routine from an on-chip ROM, and runs it. There is of course no way to change this code.
- The SD card’s partition table is scanned for a partition with the partition type field having the value 0xa2. Most partition tools will consider this an unknown type.
- The 0xa2 partition is expected to contain raw boot images of the preloader. Since there’s a 60 kB limit on this stage, the full U-boot loader can’t fit. Rather, the SPL (”Secondary Program Loader”) component of U-boot is loaded into the processor.
- The U-boot SPL, which functions as the preloader, contains board-specific initialization code, that the correct UART is used, the DDR memory becomes usable and the pins designated as GPIO start to behave like such, etc. One side-effect is that the four leftmost LEDs are turned off. This is a simple visible indication that the SPL has loaded.
- The SPL loads the “full U-boot” image into memory, and runs it. The image resides in the 0xa2 partition, immediately after the SPL’s boot images.
- U-boot launches, counts down for autoboot, and executes its default boot command (unless a key is pressed on the console, allowing an alternative boot or change in environment variables through the shell).
- In the example setting, U-boot loads three files from the first partition of the SD device, which is expected to be FAT: The kernel image as uImage (in U-boot image format), the device tree as socfpga.dtb, and the FPGA bitstream as soc_system.rbf.
- The kernel is launched.

Example boot process in Altera's Cyclone V SoC FPGA

- The AF […] o way to char […]
- The SD […] st partitio […]
- The 0x […] s stage, […] s loaded […]
- The U […] ect UART is use[…] […]. One side-e […] loaded.
- The S […] immed[…]
- U-boo […] sed on the co […]
- In the […] to be FAT: T […] A bitstre […]
- The ke […]

```
U-Boot SPL 2012.10 (Nov 04 2013 - 19:29:22)
SDRAM: Initializing MMR registers
SDRAM: Calibrating PHY
SEQ.C: Preparing to start memory calibration
SEQ.C: CALIBRATION PASSED
DESIGNWARE SD/MMC: 0
```

```
U-Boot 2012.10 (Nov 04 2013 - 19:29:32)

CPU   : Altera SOCFPGA Platform
BOARD : Altera SOCFPGA Cyclone 5 Board
DRAM:  1 GiB
MMC:   DESIGNWARE SD/MMC: 0
In:    serial
Out:   serial
Err:   serial
Net:   mii0
Hit any key to stop autoboot:  5
```

Add new functionality

The typical way to add a completely new functionality to U-Boot is

- writing driver code
- writing the command front-end for it
- enable them both with CONFIG flags

In some cases, a segment is added in the initialization sequence, in order to prepare the hardware before any command is issued.

Example: Enable GPIO

cmd/gpio.c

```
U_BOOT_CMD(gpio, 3, 0, do_gpio,
  "input/set/clear/toggle gpio pins",
  "<input|set|clear|toggle> <pin>\n"
  "    - input/set/clear/toggle the specified pin");
```

drivers/gpio/*

```
if (sub_cmd == GPIO_INPUT) {
    gpio_direction_input(gpio);
    value = gpio_get_value(gpio);
}
```

cmd/Makefile

```
COBJS-$(CONFIG_CMD_GPIO) += cmd_gpio.o
```

drivers/gpio/Makefile

```
[ ... ]
COBJS-$(CONFIG_BCM2835_GPIO)    += bcm2835_gpio.o
COBJS-$(CONFIG_S3C2440_GPIO)    += s3c2440_gpio.o
COBJS-$(CONFIG_XILINX_GPIO)     += xilinx_gpio.o
COBJS-$(CONFIG_ADI_GPIO2)       += adi_gpio2.o

[ ... ]
```

```
#define CONFIG_CMD_GPIO
#define CONFIG_XILINX_GPIO
```

Existing U-Boot commands

go - start application at address 'addr'
run - run commands in an environment variable
bootm - boot application image from memory
bootp- boot image via network using BootP/TFTP protocol
bootz - boot zImage from memory
....
diskboot- boot from IDE devicebootd - boot default, i.e., run 'bootcmd'
loads - load S-Record file over serial line
loadb - load binary file over serial line (kermit mode)
md - memory display
mm - memory modify (auto-incrementing)
....
cmp - memory compare
crc32 - checksum calculation
i2c - I2C sub-system
sspi - SPI utility commands
base - print or set address offset
printenv- print environment variables
setenv - set environment variables
saveenv - save environment variables to persistent storage
protect - enable or disable FLASH write protection
erase- erase FLASH memory
....
bdinfo - print Board Info structure
iminfo - print header information for application image
coninfo - print console devices and informations
....
mtest - simple RAM test
icache - enable or disable instruction cache
dcache - enable or disable data cache
reset - Perform RESET of the CPU
echo - echo args to console
version - print monitor version
help - print online help
? - alias for 'help'

Available C APIs useful in adding new functionality

Every function within U-Boot can be accessed by any code, but some functions are more used than others. Looking at other drivers and cmd_*.c files usually gives an idea on how to write new code. Much of the classic C API is supported, even things one wouldn’t expect in a small boot loader.

There are a few functions in the API that are worth to mention:

- Registers are accessed with writel() and readl() etc. like in Linux, as defined in arch/arm/include/asm/io.h
- The environment can be accessed with functions such as setenv(), setenv_ulong(), setenv_hex(), getenv(), getenv_ulong() and getenv_hex(). These, and other functions are defined in common/cmd_nvedit.c
- printf() and vprintf() are available, as well as getc(), putc() and puts().
- There’s gunzip() and zunzip() for uncompressing data.
- The lib/ directory contains several library functions for handling strings, CRC, hash tables, sorting, encryption and others.
- It’s worth looking in include/common.h for some basic API functions.

Get source code and compile U-Boot

- Compile?
  - cross compile on x86 for ARM
  - sudo apt-get install gcc-arm-linux-gnueabi
  - export CROSS_COMPILE=aarch64-linux-gnu-
- Version issues
  - U-boot git clone gets 2018 version, that needs gcc > 6.0
  - The above for Ubuntu 14.04 has gcc 4.3.7
  - http://releases.linaro.org/components/toolchain/binaries/6.2-2016.11/arm-linux-gnueabihf/
    has a cross compiler with gcc 6. Download, untar and set CROSS_COMPILE accordingly
    - export CROSS_COMPILE=~/linaro-toolchain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
  - gives compilation error (reported in u-boot bugs)
  - finally got a 2017 version of u-boot from http://ftp.denx.de/pub/u-boot/