U-Boot programming: A tutorial -- Part I

Introduction

It's often desirable to make slight changes to U-Boot in order to adapt it to custom hardware. For example, supporting board-specific features or adding a few routines that give the end-user signs that the device has indeed powered on, and that something is happening while the boot process takes place.

The short tutorial focuses on U-Boot for ARM, but the techniques used on other architectures are similar and often exactly the same. It's assumed that the reader is familiar with U-Boot usage at the command level as well as compilation and deployment.

It's most recommended to read the README file in the project's root directory first. It covers the following topics:

- The source file tree structure
- The meaning of the CONFIG defines
- Instructions for building U-Boot.
- How to port U-Boot to new platforms
- A brief description of the Hush shell
- How to build the Linux image (mkimage)
- A list of common environment variables
- The "Hello world" example and how to use it

boards.cfg contains a list of supported boards. It's worth to take a look at it as well.

This tutorial was written with respect to U-Boot version v2013.07, but the principles apply for a wide range of versions.

Sensible Hacking

The immediate instinct when encountering a large chunk of software sources is to look for the first place to inject a small hack, and hardcode the necessary functionality. Not only will this probably lead to daunting re-hacking and recompilations in the future, but it's unnecessary: U-Boot is actually laid out to make it easy to add custom functionality.

One can divide possible modifications into three sorts:

- Modifications in U-Boot's initialization process, so that a custom board's specific hardware is set up early enough
- Adding support to specific hardware, by virtue of adding or modifying low-level drivers
- Expanding the command interface to support a needed functionality, possibly as a front-end for new hardware

It may be tempting to add a few lines of hack code in the board's initialization routine to perform a specific operation. This will most likely work, but as just mentioned, hardcoding has its disadvantages. Writing a small custom driver and command support is by far more elegant and reusable, if the hardware's setup can be deferred to the command execution stage.

This tutorial is divided into three parts: A general view on U-Boot (this part), a hands-on explanation on how to add functionality (part II) and some background on U-Boot's bring-up process, for those who need to initialize something very early (part III).

Software organization

Linux kernel hackers will feel relatively comfortable with U-Boot, as much of the coding style and organization is inspired by the Linux kernel. The structure is however simpler at the cost of less flexibility. There's no intermediate layer between the drivers and the user front-end.

For example, to get the value of a GPIO pin, just call gpio_get_value(gpio) with the GPIO's pin number from anywhere in the code. There is no place for more than one GPIO driver to be compiled into the system: Only one source file, which defines this function, may be enabled for compilation, or the linking will fail. And of course, if gpio_get_value() is used somewhere, this one source file must be compiled.

Related pages

- U-Boot programming: A tutorial -- Part II
- U-Boot programming: A tutorial -- Part III
- Preparing a Uboot image for Altera's Cyclone V SoC FPGA

So a “hardware driver” in U-Boot is just a piece of code that implements a set of functions that are linked into the global name space. It kinda makes sense for a utility that needs to be compact: There’s no point compiling in anything that isn’t used, and most of the time there’s a fixed set of hardware involved, with one instance of each kind, at most.

One driver may, of course, depend on the other. For example, the SOFT_I2C driver depends on two GPIO pins that are connected to an I2C device. These pins are accessed using the GPIO’s API functions. Any other piece of software can access the GPIO API as well (hopefully not the same pins).

Behind the scenes of make XXX_config

Anyone who has built U-Boot has typed something like

```
$ make zynq_zed_config
```

before compiling the project. Many have also spotted that there’s a /include/configs/ directory, in which corresponding files are found, for example zynq_zed.h, which reads something like this (GPLv2 header at the top not shown here):

```
#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 */
```

This is in fact just the tip of the iceberg. Among others, this “make zynq_zed_config” command generates include/config.h. as follows:

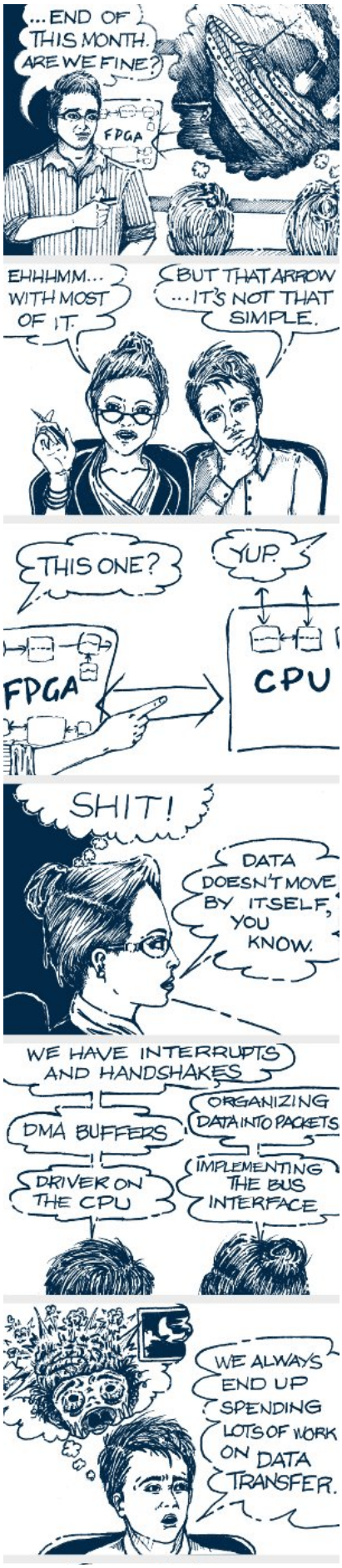
```
/* Automatically generated - do not edit */
#define CONFIG_SYS_ARCH  "arm"
#define CONFIG_SYS_CPU   "armv7"
#define CONFIG_SYS_BOARD "zynq"
#define CONFIG_SYS_VENDOR "xilinx"
#define CONFIG_SYS_SOC    "zynq"
#define CONFIG_BOARDDIR board/xilinx/zynq
#include <config_cmd_defaults.h>

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

The included config files contains other #define (and possibly #undef) statements, most of which for variables with a CONFIG_* prefix.

At this point, it’s quite clear that there’s a somewhat tangled set of header files that define CONFIG_* compilation variables (and others). These affect the way the code is compiled in two ways:

1. All CONFIG_* variables that are defined turn into Makefile variables in include/autoconf.mk, where each variable that is just defined (as opposed to assigned a value) gets the value “y”. This is used in the Makefiles in each source directory to pick which files are compiled and linked into the main executable.
2. These compilation variables are used in the compiled C sources directly, and may contain



target-specific attributes.

Unlike the Linux kernel, there is no KConfig utility, so these definitions are made in board-specific h-files. To add a new “make config” target, create a new config file in include/configs/ (or better, copy a similar configuration file), and add a line in boards.cfg.

The three functions of code in U-Boot

One can divide the code into three types (parallel to the sorts of modifications, mentioned above):

- Pure initialization code: This code always runs during U-Boot’s own bring-up. More about the stages of this in part III.
- “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

All three code types are strongly influenced by the CONFIG defines. For example, a CONFIG that enables the compilation of a certain driver may also cause a snippet of initialization code to opt in with #ifdef.

The typical way to add a completely new functionality to U-Boot is writing driver code, writing the command front-end for it, and 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.

Continue to part II, which explains how to add functionality.

Further reading

- U-Boot programming: A tutorial -- Part II
- U-Boot programming: A tutorial -- Part III
- Preparing a Uboot image for Altera’s Cyclone V SoC FPGA