U-Boot programming: A tutorial -- Part II

Example: Adding GPIO support

U-Boot supports GPIO on several platforms, but is often not enabled. We'll briefly walk through the process of enabling this functionality, possibly for a new platform.

Looking in the common/ directory, it's quite easy to spot the cmd_gpio.c file. It contains code handling the "gpio" command in U-Boot's shell.

Its last few lines say:

```
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");
```

This declares do_gpio() as the function to handle for the "gpio" command. It also says that up to **two** arguments are allowed (the number 3 includes the command itself), and that autorepeat is not allowed. There are also the short and long help strings. The U_BOOT_CMD macro is explained in doc/README.commands.

The do_gpio() function is in essence an argc/argv function, with some extra parameters that are not used in most implementations. So it's down to parsing the arguments and call lower-level functions, usually from some driver API.

It's quite pointless to go through cmd_gpio.c in detail — the principle of how the commands are interpreted and executed is easily understood by reading it and other cmd_*.c files. In most cases, it's simpler than one would expect.

As for the interaction with the build system, it's quite simple as well: Somewhere in common/Makefile, there's a line saying

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

Without getting into the tangled parts of the build system, this line means that if CONFIG_CMD_GPIO is defined in the Config chain, cmd_gpio.c will be compiled into cmd_gpio.o, and linked into the global project. The U_BOOT_CMD() macro makes sure that the command is enlisted in the U-Boot shell interpreter.

But we're not done yet. For example, in cmd_gpio.c, there a place saying

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

The functions gpio_direction_input() and gpio_get_value() are not defined in cmd_gpio.c itself. Rather, it's expected that they'll be defined by some other file (a "driver") which is compiled and linked into the global executable.

As one would expect, there are many candidate C files in drivers/gpio/, each named after the platform it's intended for. They all implement more or less the same set of functions, and in a pretty similar way. The low-level implementation varies, of course.

Only one of these can be compiled, of course. Which one depends which CONFIG is defined (if at all).

The Makefile in drivers/gpio/ is similar, with lines saying

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

Related pages

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

```
COBJS-$(CONFIG_XILINX_GPIO)      += xilinx_gpio.o
COBJS-$(CONFIG_ADI_GPIO2)        += adi_gpio2.o

[ ... ]
```

So all in all, to use Xilinx' GPIO, the two following lines should appear in the board's configuration file:

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

Note that sometimes additional configuration variables are necessary. Which ones and what they mean is often easiest to deduce by reading the driver's source code.

To add a driver for a yet unsupported piece of hardware, it's recommended to look at the existing drivers, and see if there's one that works almost as required. Sometimes it's possible to tweak it with #ifdef's to make it support the desired hardware.

If this is not the case, the recommended practice is to make copy of the driver that seems to be closest to the desired functionality, and make changes as necessary. In order to include the new driver in the build system, add a COBJ-$() += ... line in the Makefile of the same directory, and a corresponding (new) CONFIG flag to enable it.

Available API

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.

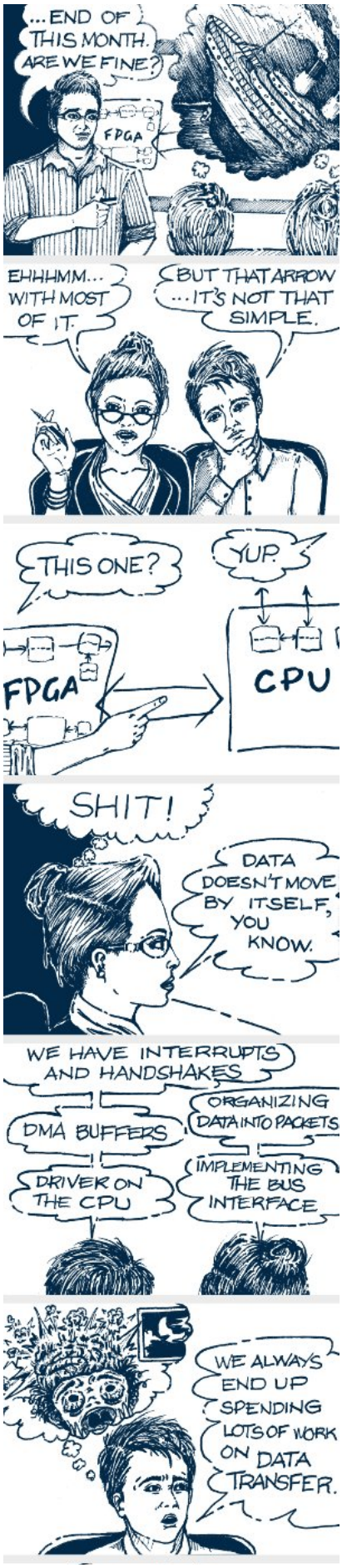
There are a few functions in the API that are worth to mention:

- 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.

Continue to part III, which briefly deals with U-Boot's own bring-up process.

Further reading

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



Xillybus' IP core offers a simple and intuitive solution for host / FPGA interface over PCIe and AXI buses. Xilinx or Altera, Windows or Linux, they are all supported.

Click here for more information.

© Copyright 2010-2020 Xillybus Ltd. | Email for inquiries: **general@xillybus.com**