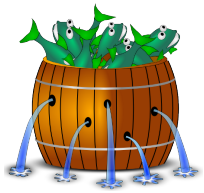

**ADVANCED OPERATING SYSTEMS**
**Milestone 0: Familiarization**
**Fall Term 2016**

Assigned on: **22.09.2016** Due by: **29.09.2016**

This is the first milestone in the course. The goal here is to get you familiar with the build environment for Barrelfish on the PandaBoard, and also do some very simple device programming. At this stage it's pretty low-level; we'll get on to more deep OS concepts later in the course.

# 1 Check Your Hardware

You will need several pieces of hardware from us for this course.

The first is the PandaBoard-ES itself. Please take great care of it — it's quite fragile. Always transport it in its box, protected by bubble wrap. Also, be gentle when plugging and unplugging connectors, and pressing buttons. You should not need to unplug cables very often.

You are also going to get two cables: One is a mini-USB cable which plugs into connector J18 on the PandaBoard. This powers the PandaBoard from the USB port on your computer, and is also used to boot the PandaBoard from your computer by downloading the OS image into memory. The other cable is a USB-to-RS232 adaptor, which plugs into the large 9-way D connector on the PandaBoard. We use this for console output.

# 2 The Software Toolchain

In this course you will be compiling and linking OS code for the PandaBoard, booting the machine over the USB cable, and monitoring the output via the console. The software we use runs on Linux, and we have provided all the software you need to do this on the lab machines located in CAB H 56 and H 57.

The toolchain for building the OS consists of:

- The GCC compiler version 4.7.3, configured to compile code for ARM processors using the Linux ABI. Other versions may work, but we will only provide support for this one. The `gcc-arm-linux-gnueabi` package on Ubuntu 14.04 provides this. GCC version 4.9.2 as provided by Debian 8.5 (jessie) is known to work as well.
- The Glorious Glasgow Haskell Compilation System (GHC), version 7.6.3. Some of the tools you will use to build the OS are written in Haskell, and we use GHC to compile them. Again, Ubuntu 14.04 and Debian 8.5 provide a compatible package.
- Standard GNU/Linux tools like `bash`, `awk`, `make`, etc.
- `picocom`, a terminal emulation program for interacting over the RS-232 line.

All these are provided in a NFS share (`/pub/aos/tools/`) accessible from the lab machines in CAB H56/H57.

You are, of course, welcome to install these tools on whatever machine you like for use in the course, and many people do so successfully. On Ubuntu 14.04 running the following commands should be sufficient to get all the tools setup:

```
$ sudo apt-get install build-essential bison flex cmake \
qemu-system-arm ghc libghc-src-exts-dev libghc-ghc-paths-dev \
libghc-parsec3-dev libghc-random-dev libghc-ghc-mtl-dev libghc-src-exts-dev \
libghc-async-dev gcc-arm-linux-gnueabi g++-arm-linux-gnueabi libgmp3-dev \
cabal-install curl freebsd-glue libelf-freebsd-dev libusb-1.0-0-dev
$ cabal install bytestring-trie
```

The only configurations we fully support, however, are installing the tools on Ubuntu 14.04 as shown above and the toolchain provided as an NFS share on the lab machines. If you're on Ubuntu 16.04, we provide a patch which you can apply (`tools/ubuntu1604_toolchain.patch`) to your checkout. For any other configurations we can help to some extent with getting things working "natively" on your laptop or desktop machine, but there are simply too many variations for us to support them all. Ultimately, if you choose to use an environment other than the lab machines or Ubuntu 14.04, you're on your own.

# 3 Connect the PandaBoard and Configure the Hardware

**a)** Log in to one of the lab machines using your NetzID and make sure that you can access the AOS NFS share in `/pub/aos/tools/`.

**b)** Plug the RS-232 adaptor cable into the PandaBoard and the other end into your computer.

**c)** Plug the mini-USB cable into the PandaBoard and the other end into your computer. The PandaBoard should be powered up, and the LED close to the SD card slot on the PandaBoard should light up.

You should avoid using unpowered USB hubs to connect the PandaBoard, as they are probably not capable of providing enough power to the PandaBoard.

# 4 Check That the PandaBoard Boots

**a)** Clone the AOS Barrelfish repository to somewhere convenient in your home directory, and check the week 1 (milestone 0) branch out, giving it a meaningful new name, such as `week1_work` (see the course website for the correct repository to pull from):

```
$ git clone repository bf_aos
$ cd bf_aos
$ git checkout -b week1_work week1
```

If you're running Ubuntu 16.04: Apply toolchain patch mentioned in section 2:

```
$ patch -p1 < tools/ubuntu1604_toolchain.patch
```

**b)** Assuming you're working on a lab machine in CAB: Setup your environment by executing the following line (assuming you're using bash as your shell):

```
$ source /pub/aos/tools/aos_env.sh
```

You will have to do this in every shell you are using throughout the course.

Note: If you're working on a lab machine the GCC binary is called `arm-linux-gnu-gcc` rather than `arm-linux-gnueabi-gcc` as mentioned below!

**c)** Make sure the toolchain is setup properly:

```
$ ghc --version
$ arm-linux-gnueabi-gcc --version
$ picocom -h
```

**d)** Create a build directory:

First, create a directory to build your files in. This should not be in the source tree. Also, if you're working on the lab machines we strongly recommend that you put the build directory on the local hard disk (which is mounted under `/local`, and will be a *lot* faster than NFS). If you're using `/local` you should create a directory with your user name and put your stuff in there. For example:

```
$ mkdir -p /local/<your nethz-name>/build_milestone_0
```

**e)** Build the Makefile by typing:

```
$ cd /local/<your nethz-name>/build_milestone_0
$ <path to source tree>/hake/hake.sh -s <path to source tree>
```

We will explain what happens here in more detail in section 6 of this document.

**f)** Run `picocom` to monitor output on the serial cable, using:

```
$ picocom -b 115200 -f n /dev/ttyUSB0
```

Note: You can quit `picocom` by using key-sequence Ctrl-a Ctrl-x.

**g)** Use `usbboot` to send a compiled image to the PandaBoard. We have supplied an initial test image for you to try: this will print something to the console, and also flash the LEDs.

In the build directory, type:

```
$ make tools/bin/usbboot
$ tools/bin/usbboot <path to source tree>/milestone0_test_image
```

Now press the reset button (the white button closest to the SD card reader). If you've done everything right, you should see something like the following output in `picocom`:

```
[ aboot second-stage loader ]

Entry point is 8000E980
Reading 278756 bytes to 80000000

...

kernel ARMv7-A: Welcome to AOS.
```

You should also see the LEDs flashing alternately.

# 5 Understand Booting

The rest of this milestone consists of reproducing the output of the boot image you have just tested, and understanding how it all works.

What just happened?

The process for booting any machine is complex, and usually very specific to a particular piece of hardware. For example, the way we have just booted our test kernel on the PandaBoard is very different from the way that a PC or an iPhone boots.

The basic principles, however, are always the same:

**a)** Put the machine hardware into a known state.

**b)** Copy a program to a well-known location in memory.

**c)** Jump to a well-known address in that program (the 'entry point').

This three-step process often happens multiple times in sequence when a machine boots.

When an ARM processor (such as that in the PandaBoard) is reset, it executes a jump to known, but processor-specific address (`0x40030000` for the PandaBoard). On the PandaBoard, this address points to ROM code, that performs basic hardware initialisation, sufficient to load the second-stage bootloader over an external interface, such as USB, into the on-chip memory at address `0x40300000` (See OMAP4460 TRM table 2-1, p. 294, for the memory map 2).

This second-stage bootloader is supplied by `usbboot`, and is called `aboot`. If you are interested, you can see its source code in `tools/usbboot/aboot.c`. `aboot` configures the DRAM controller (the on-chip memory doesn't need configuration), and then loads the image supplied to `usbboot` into RAM (usually at `0x80000000`), jumping to its start address.

We're not yet done: Barrelfish needs a number of different executables to boot, and itself boots in multiple stages. One a single core, Barrelfish is a bit like a microkernel (actually, it combines ideas from microkernels and exokernels), and so runs much of its functionality in separate processes. For this reason, we package a number of different files into a single boot image, using a format devised for booting PCs called *multiboot* (link).

Since the PandaBoard is expecting to run a single program (strictly speaking, `aboot` is expecting to run a single program), we use a tool called `arm_boot` to package a set of programs into a single (ELF) executable which we then boot. This ELF file contains the CPU driver (kernel), loaded, relocated and ready to run. It also contains the ELF images for any other executables in the boot image, which are all described in the *multiboot header*. You can get a feel for the layout of this image, by looking at the ELF section table. Typing `readelf -S milestone0_test_image` will list the section headers, which will look something like this:

```
There are 18 section headers, starting at offset 0x498b0:

Section Headers:
  [Nr] Name              Type      Addr     Off    Size
  [ 0]                   NULL      00000000 000000 000000
  [ 1] .boot             PROGBITS  80000000 004000 00e980
  [ 2] .text             PROGBITS  8000e980 012980 0038c4
  [ 3] .rodata           PROGBITS  80012244 016244 000ac4
  ...
  [12] .cpudriver        PROGBITS  80018000 01c000 00b000
  [13] .elf.cpu_aos_m0 PROGBITS  80023000 027000 021000
  [14] .multiboot        PROGBITS  80044000 048000 0000e4
  ...
```

The first few sections (up to number 11 here), are the *boot shim*, which is in effect a *third-stage* bootloader, and is responsible for enabling the MMU, and setting up the CPU driver's address space, before jumping to the CPU driver's entry point. The CPU driver itself is present twice: once as an unprocessed ELF file in section `.elf.cpu_aos_m0` (13) (for loading it onto cores other than the first), and again as a fully loaded and relocated executable image, in section `.cpudriver` (12). The multiboot header, which describes the layout of this initial image, so that the kernel can find the user-level tasks to load, is in section `.multiboot` (14).

In later milestones, you will add user-level tasks to your boot image, which will appear in the output of `readelf` as sections with names like `.elf.binary_name`.

# 6 Build a Bootable Image

You'll now build your own image that we can boot on the PandaBoard. For this, we'll need some code, which you already got when you checked out the week 1 branch from the repository earlier.

This tree is a subset of the main Barrelfish build tree, but contains most of the files needed to build a single, bootable program for the PandaBoard. There's also a lot of extra code that isn't relevant to this milestone, but it does give you an idea of the complexity of even a research OS like Barrelfish.

The kernel in Barrelfish is called the CPU driver, and sits in the source directory `kernel`. In this directory are some files that are portable across all kinds of machines, and others that are specific to particular instruction set architectures, processor models, or platforms. The latter are in the various directories under `kernel/arch`.

To start, take a look at `kernel/arch/armv7/cpu_entry.S`. This is the first code which is executed after the boot shim (the third-stage bootloader), and contains a sequence of 4 assembly instructions which:

**a)** Load the kernel stack pointer.

**b)** Find the *global offset table* (GOT), which points to the kernel's global variables.

**c)** Save the GOT pointer in a special machine register, so that it doesn't need to be manually recalculated on every kernel entry.

**d)** Jump to another function, called `arch_init`.

Loading the stack pointer and initialising the GOT is enough to enter code compiled with a C compiler, so `arch_init` can handily now be written in C. You'll find it in the file `kernel/arch/armv7/aos/init.c`. Right now it tries to print some status information, and then call a function named `blink_leds()`.

The first step is to see if we can build this trivial kernel.

The build process for Barrelfish uses two steps. In the first step, a program called *hake* (7) generates a Makefile. This is what we did earlier when we built the Makefile before building `usbboot`. This is then used in the second step to build the operating system.

Remember, the Makefile can be built by typing:

```
$ cd /local/<your nethz-name>/build_milestone_0
$ <path to source tree>/hake/hake.sh -s <path to source tree>
```

This tells hake to build a Makefile which can build Barrelfish

Also, the `usbboot` tool can be built by typing:

```
$ make tools/bin/usbboot
```

Next, build the kernel:

```
$ make armv7_aos_m0_image
$ ls -l armv7_aos_m0_image
```

If you want to see what this does at a high level, find the file `platforms/Hakefile` and look for the line beginning with `armv7Image "armv7_aos_m0"`.

This should work with no compilation errors. If so, congratulations! You've built a Barrelfish CPU driver that does nothing. You can, of course, boot this on your PandaBoard using `tools/bin/usbboot armv7_aos_m0_image`, but you won't see any output beyond the boot shim, since the code to write to the UART and to blink the LEDs is missing. You'll now fix this.

# 7 Console output

Now we want some output. The serial device on the PandaBoard (the bit that drives the 9-way D connector you've plugged a cable into) is called a UART, and it's part of the OMAP4460 chip which is at the heart of the PandaBoard. We'll write a very simple UART driver that can output a character to the device.

The UART is documented in the Technical Reference Manual for the OMAP4460, S23.3 (2).

Opening this document for the first time is a little scary - it's over 5,000 pages long (the table of contents alone is 258 pages all told). However, don't be afraid — we won't be using most of the capabilities of the chip during this course, and even for those that we will use, most of the documentation in this manual isn't relevant to us (it includes power management, internal architectural details, etc.).

Here's what you need to know:

**a)** There are actually 4 identical UART devices on the PandaBoard processor, but we are interested in UART3, which is the one connected to our serial port. The registers for this device start at address `0x48020000` in physical memory (see table 2-4, p. 300, in the TRM).

**b)** How to program the UART is explained in Section 23.3.5 of the manual, but we *won't* be doing anything this complicated. In particular, we won't be using interrupts, DMA, or FIFOs. Instead, we just want to get a single character out of the serial line for debugging purposes. Furthermore, the UART will already have been initialised by the bootloader, by the time you get to `arch_init()`. You just need to figure out how to send characters.

**c)** To send a character to the serial line, your code first needs to wait until the UART is ready to send a character. This happens when the internal transmit FIFO, which the UART uses to buffer characters, is empty, indicated by bit 5 (named `TX_FIFO_E`) of the Line Status Register (`UART_LSR`) becoming set (i.e. 1). This register is described on page 4737 of the manual, and for our UART you can see that it sits at address `0x48020014`.

To actually send a character once the UART is ready, we write the character into the Transmit Holding Register (`UART_THR`), which is at address `0x48020000` and is described on page 4722 of the manual (though there is not a lot to say about it!).

You now know all you need to know to write a character to the serial port, so you should go ahead and fill in the body of the simple function called `serial_putchar()`, in `kernel/arch/armv7/aos/plat_omap44xx.c` which does this. It should loop waiting for `TX_FIFO_E` to be 1, and then write the character into `UART_THR`.

Test this out by adding a line to `arch_init()` to call your new function with an argument of 42. Rebuild the kernel, boot it, and you should see an asterisk ('*') as the last thing in the output on picocom.

Once you've got this, of course, you can try it with other characters. However, the kernel has a minimal C library which includes `printf()`, which in turn calls `serial_putchar()`, so you should be able to now use this. This makes debugging subsequent code much easier.

# 8 Flash the LEDs

Now that you can write strings to the console port, the final exercise for this milestone is to flash LED D2 (the one nearest the SD card reader) on the PandaBoard.

This LED is controlled by one of the General-Purpose I/O lines. Take a look at the PandaBoard ES System reference manual (4) on page 47, and you'll see that it's GPIO line WK8. This is on the first GPIO block on the OMP44xx (GPIO1), according to the OMAP4460 datasheet (3), and page 298 of the TRM tells us that this is at address `0x4a310000`.

Programming the GPIO pins is described in Chapter 25 of the TRM, and they can do a lot of different things (they're *general purpose*). However, all we need to turn the LED on or off is two registers: Output Enable or `OE`, and Data Output, or `DATAOUT`. These registers have a bit for each of the 32 I/O pins controlled by the GPIO1 block, and you need to set the correct `OE` bit for the LED to enable the pin, and then set or clear the correct `DATAOUT` bit to turn the LED on or off.

Your task is to flash the LED on and off about once per second, by implementing the function `blink_leds()`, again in `kernel/arch/armv7/aos/plat_omap44xx.c`.

## Submission

You should submit your code as a tarball through the submission system accessible from course website before the specified deadline.

## THE MILESTONE

- Build and boot an image on the PandaBoard.
- Demonstrate console output from a booted image.
- Demonstrate flashing the LED on and off.

## CHALLENGES

- Handle console input as well as output, and build a toy command line interpreter in the kernel.
- Flash LED D1.

## ASSESSMENT

- You will need to demonstrate that you can build and boot an image.
- Explain your code for serial output and LED control to the tutor.

# 9 Resources

1) ARM architecture reference manual
2) OMAP4460 Technical Reference Manual
3) OMAP4460 Datasheet
4) Pandaboard ES System Reference Manual
5) AOS Styleguide
6) Multiboot
7) Hake Barrelfish TechNote
8) More links in the resources section on the AOS course page.