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 Step 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 Step 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 4.6.3, configured to compile code for ARM processors using the Linux ABI. This compiler can be obtained from Mentor Graphics online as part of the Sourcery CodeBench Lite tool suite, version 2012.03-57.
- The Glorious Glasgow Haskell Compilation System (GHC), version 7.4.1. Some of the tools you will use to build the OS are written in Haskell, and we use GHC to compile them.
- Standard GNU/Linux tools like bash, awk, make, etc.
- picocom, a terminal emulation program for interacting over the RS-232 line.
- The usbboot tool for booting the PandaBoard.
All these are provided in a NFS share (/pub/aos/tools13/) accessible from the lab machines.

You are, of course, welcome to install these tools on whatever machine you like for use in the course, and some people do. The configuration we support, however, is the use of our toolchain provided as NFS share on the lab machines - 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, you’re on your own.

You can get an exact copy of that toolchain in the file /pub/aos/tools13/tools.tar.gz

3 Step 3: Connecting the PandaBoard and configuring 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/tools13/.

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 reader 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 Step 4: Checking that the PandaBoard boots

a) Copy the code for the first assignment from the NFS share to somewhere convenient in you home directory:

   $ mkdir $HOME/aos-part1
   $ cd $HOME/aos-part1
   $ tar xvf /pub/aos/handout/m0.tar.gz

b) Setup your environment by executing the following line (assuming you’re using bash as your shell):

   $ source /pub/aos/tools13/aos_env.sh

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

c) Make sure the toolchain is setup properly:

   $ ghc --version
   $ arm-none-linux-gnueabi-gcc --version
   $ picocom -h
   $ which usbboot

d) Run picocom to monitor output on the serial cable, using:

   $ picocom /dev/ttyUSB0

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

e) Use usbboot to send a compiled image to pandaboard. We have supplied an initial test image for you to try: this will print something to the console, and also flash one of the LEDs on the PandaBoard.

   Type:

   $ usbboot milestone0_test_image
Now hold down the reset button (the white button closest to the SD card reader) for a couple of seconds and then release it. If you’ve done everything right, you should see the following output in picocom:

```
[ aboot second-stage loader ]
MSV=00000000
jumping to 0x82001000...
Barrelfish OMAP44xx CPU driver starting at addr 0x83fa0000
...
Welcome to Advanced Operating Systems 2013!
```

You should also see the LED flashing on and off.

5  Step 5: Understanding 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 somewhat 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.

However, the basic principles 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 the one in the PandaBoard) is reset, it executes a jump to address 0x00000000. On the PandaBoard, this already contains code which handles the next stage of the boot process, which in our case downloads an image over the USB port and executes it.

The usbboot program itself takes the boot image that we have given it and wraps it inside another program called the "2nd stage boot loader", 2ndstage.bin. When 2ndstage.bin runs, it copies the image to an area of RAM and jumps to it.

We’re not yet done: Barrelfish itself likes to boot with several available programs. 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 like to package a number of different files into a single boot image, using a format devised for booting PC called “multiboot” (link).

Since the PandaBoard is expecting to run a single program (strictly speaking, 2ndstage.bin is expecting to run a single program), we use a tool called “molly” to package a set of programs into a single executable which we then boot. When it runs on the PandaBoard, this executable, which is generated by molly, unpacks the programs into memory, records where they are, and then jumps to the first one, which is conventionally the kernel.
6 Step 6: Building a simple bootable image

We’ll now build a simple image that we can boot on the PandaBoard. For this, we’ll need some code, which you already got as part of the tarball you extracted to your home directory earlier.

$ cd $HOME/aos-part1

This tree is a (very small) 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/omap44xx/boot.S. This is the first code which is executed after the bootloader, and contains about 10 assembly instructions which:

a) Put the processor into “privileged” mode
b) Set up the stack pointer to refer to a statically-allocated area of memory (in the kernel’s data segment)
c) Load a register to point to another area of memory containing information about the multiboot image
d) Jump to another function, called arch_init.

Loading the stack pointer and argument register are sufficient for us 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/omap44xx/init.c. Right now it simply enters an infinite loop.

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” generates a Makefile. This is then used in the second step to build the operating system.

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). 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

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> -a armv7

This tells hake to build a Makefile which can build Barrelfish for ARMv7-based cores.

Now, build the kernel:

$ make aos_image
$ ls -l aos_image

If you want to see what this does at a high level, find the file symbolic_targets.mk and look for the rule defining milestone0.

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 usbboot ./milestone0_1_image, but you won’t see any output beyond the bootloader, since the code does nothing. We’ll fix this.
7 Step 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 simply UART driver that can output a character to the device.


Opening this document for the first time is a little scary - it’s 5,868 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 when we will, 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 UART identical devices on the PandaBoard processor, by 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.

b) How to program the UART is explained in Section 23.3.5 in 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 code to initialize the UART is already provided in omap_uart.c.

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, and is 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 4767 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 4752 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() which does this. It should loop waiting the 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(), and which in turn calls serial_putchar(), so you should be able to now use this. This makes debugging subsequent code much easier.

8 Step 8: Flashing LEDs

Now that you can write strings to the console port, the final exercise for this milestone is to flash the “D2” LED (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 on page 53, and you’ll see that it’s GPIO line WK8. This is on the first GPIO block on the OMP44xx (GPIO1), 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 write 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.

**Submission**

You should submit your code as a tar-ball 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 and image.
- Explain your code for serial output and LED control to the tutor.
9 Resources

- ARM architecture reference manual
- Pandaboard ES System Reference Manual
- AOS Styleguide
- Multiboot
- Hake Barreliish TechNote
- More links on Resources section on AOS course-page