Embedded System Drivers Report

Carlos R. Davila (010779067)[1]

[1]*Computer Engineering Department, San Jose State University*

[1]carlos.davila@sjsu.edu

***Abstract*— This paper is an overview of the process required to design and implement a GPIO driver in an embedded system. It covers the essentials of how to analyze the board specifications and write/compile a kernel module that will enable an LED connected to the pin headers. This exercise was completed with the use of a Raspberry Pi Version 3.**

***Keywords*— Embedded, GPIO, Driver, Memory Map, Raspberry Pi**

I. Introduction & Project Goals

This project focuses on the implementation of a driver to turn on and off an LED through the GPIO pins of the embedded board. We cover how to analyze the documentation provided, how to translate the board specifications into the driver, and provide an overview of how the driver works.

II. Memory Maps

Typically, one interacts with an I/O device through the use of special purpose registers. For example, one can read registers to know about a device status, or write to registers -- asking the device to perform a particular action. Most microprocessors today offer a way to communicate with these device registers through memory mapped I/O. This means that registers on the target device (the ones you need to read from and write to) are accessible to you as a region of contiguous memory addresses. Thus, to write or read from a particular register on your target I/O device, all you have to do is write the value into a particular memory address. Any data sent to that memory address, is actually sent to the corresponding I/O device; how this is done is not in the scope of this paper. The only hard part is looking through the specification, and figuring out the register assignments, and what the memory mapped address is.

III. Example

Let's consider a non-real and simple example for illustration. Let's say, John has a speaker device that makes a barking sound on command. The device has one register that is 32 bits wide. The specification for this barking machine says that one must write a single 1, and all other 0's in order for the device to bark. It does not matter where the 1 is placed, as long as there is only 1 bit enabled, and all other bits are 0's. So writing:

0x00000001 == 0x01000000
(both make the device bark!)

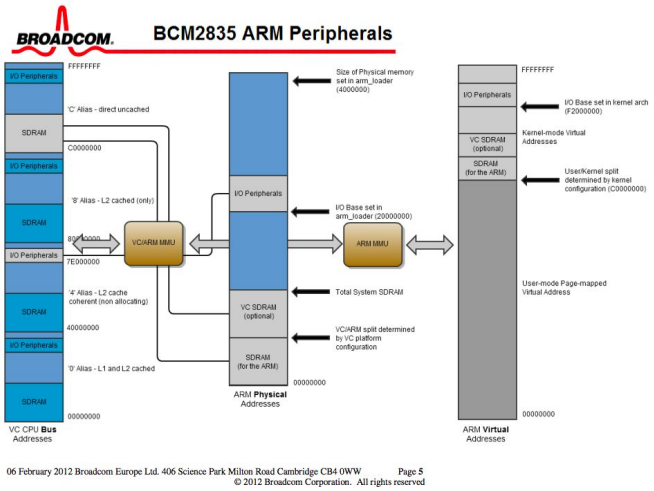
Now we have an understanding of what the device does, and how we make it bark. Next, let's assume that this device is memory mapped onto your CPU memory space. This means that the CPU has reserved a special memory address, just for that barking register. Let's say that register is located at address *0x40000003* (completely made up!). All we need to do is figure out how to write to this memory location. Let's write a driver in C to write to it. Assuming there is no virtual memory or anything like that, our code might look something like:

```
*((uint32_t *) 0x40000003) = 0x00000001;
```

*(uint32_t *) 0x40000003* means: get the number 0x40000003 (address to our memory mapped I/O), make it into a pointer to memory.
**((uint32_t *) 0x40000003)* means: Now assign the value at that pointer to be 0x00000001. (Dereferenced the pointer)
When that is executed, the value will be directed to your barking device, and your device will bark!

IV. Raspberry Pi Memory Mapping

In order to learn more about how the memory mapping is done on the RPi, we need to look at Broadcom's spec sheet. It shows the following picture explaining the memory layout of the device:



One can quickly notice that there are three types of memory addresses. They are the VC CPU Bus Addresses, ARM Physical Addresses, and the ARM Virtual Addresses. This is important — understanding which ones you are supposed to use in your program, and how they translate is essential to making your driver work correctly. Broadcom provides key information in their spec sheet that explains what the address range for our peripherals is.

The specification mentions that device registers are located in Physical Address starting at 0x3F000000. Our GPIO pin will be found somewhere in that range. But we also found out that Broadcom provides all of the documentation in terms of Bus Addresses. So, like they mention, any register that is denoted as address 0x7Ennnnnn (Bus Address) should be interpreted as 0x3Fnnnnnn (Physical Address). So we now know how to translate a peripheral a *Bus Address* into a *Physical Address*.

Because the CPU addresses memory through the use of Virtual Addresses that are translated by the MMU, we need a way to map the known Physical Address space given by the Broadcom spec (0x3F000000 - 0x3FFFFFFF) into Virtual Address space. The Linux kernel has a system call for this specific purpose. By using *ioremap()*, the kernel provides a way to access a specific physical address space. Now that we understand how to map a physical address to a virtual address for access from our driver, we need to figure out what actual registers we need to change, and what their physical addresses are.

V. Raspberry Pi GPIO Special Registers

Broadcom spec mentions that there are a total of 54 general-purpose I/O lines. The following are the high level steps that need to be taken by the driver:

1. Change the GPIO GPFSELn to assign the functionality of individual I/O pins.
    a. For the RPi, there are 5 GPFSELn registers, and for pin 21 (selected for this paper), we need to update GPFSEL2 bits 5-3 to read 0b001.
2. When you want to set the pin to high, you write to the GPSETn register at the bit of your pin (bit 21, for pin 21).
3. If you want to set the pin to low, then you write to the GPCLRn register at the bit of your pin (bit 21, for pin 21).