CPRE 488 Homework 3 Spring 2024

Linux Device Drivers
Jonathan Hess
GitHub Page

Problem 1

> Decode the following I2C waveform, assuming the top signal is SDA and the bottom signal is SCL. What transaction is being requested? Defend your answer.

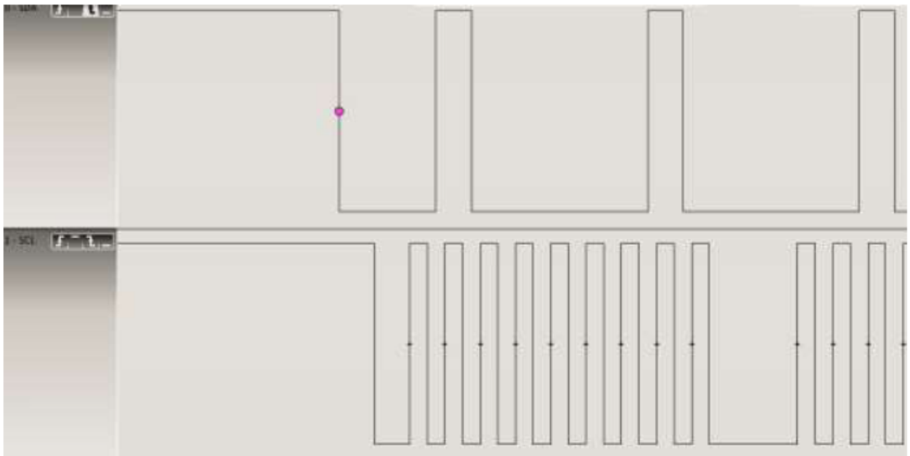

Figure 1: The provided I2C signal

The first thing to examine is that the edge of the clock does not matter.

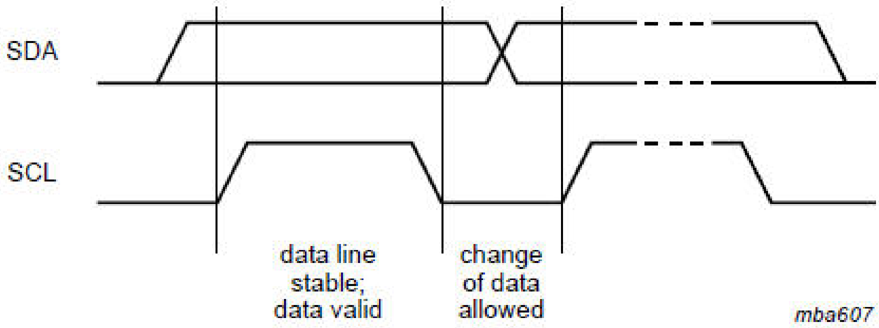

Figure 2: When data is set and read on an I2C signal[1]

This is because the change on the edges is illegal. Only when the clock is set low can the value be set.

Using this information we can get a binary value of signal.

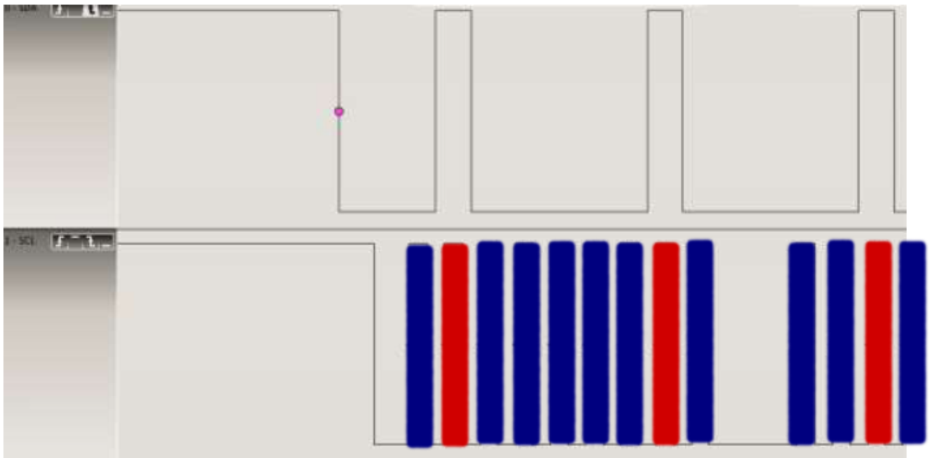

Figure 3: The colored I2C signal where red = 1 and blue = 0

The total binary value is 0b0100000100010. This can be broken up into the different frames.
I am making an assumption that this is at the beginning of the communication. This will need to be confirmed. This can be confirmed by the start of the signal where the SDA is pulled low before the SCL (normally not allowed). This is the signal for the start of the messaging.
The first frame is the 7 bit address. We know it is 7 bit and not 10 bit because the first bits of the signal are not 11110.[2] ADDR is 0b0100000 The second frame is the read write bit which is 1 (the controller is requesting).
The next frame is the ACK with one bit. The bit tells if there was an error. It is 0.

What is concerning is that a lot of the images describing the protocol show the SDA returning high after acknowledgment. What I think is happening is that the responder is setting the SDA low immediately after acknowledgment (because it has to be ready for the clock signal).

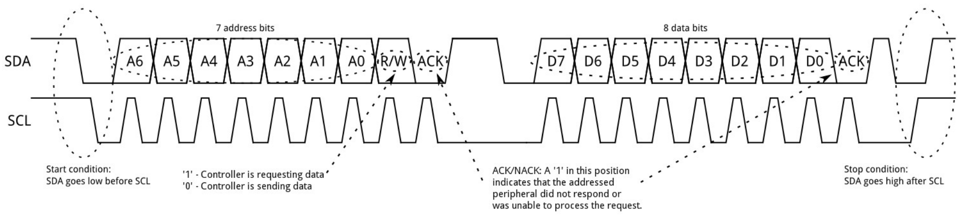

Figure 4: An example message with the I2C protocol with idle in between the address and data[2]

So we know that the controller is requesting data from a 7 bit addressable responder. It was a valid request because the aknowledge bit was set low and we even seen some of the data at the very end "0010...".

Problem 2

Fun with Documentation. Unlike in previous labs, where it was possible to get by without reading any of the background material or documentation, the driver development in MP-3 will make you a very sad panda if you don't have a good feel for embedded Linux.

Problem 2a

For starters, Xilinx maintains a very nice wiki on porting Linux to Zynq-based devices. In MP-3, we will not be following their directions precisely, but it provides a useful overview of several of the major steps involved. Read through the wiki (starting here: http://www.wiki.xilinx.com/Getting+Started), and in your writeup, describe the 4 files that are needed to boot Linux on Zynq.

The main boot order is shown in the following picture

Figure 5: Image of the embedded software stack[3]

The four files need are as follows[5]:

The zImage file which contains the compressed Linux image is the OS that will be running on the board.
The BOOT.BIN file which contains the First Stage Bootloader (FSBL) which loads the hardware bitstream.
The dtb which is the device tree blob that contains the information about the hardware for the linux OS.
The ramdisk8M.image.gz which allows the some of the RAM to be used as a disk drive for file storage.

For the sources, so far I have found the following.

The linux OS image file that we need is the image to put onto the SD card have easily accessible versions. They are avialable at the Linux Prebuilt Images page.

The PetaLinux BSP can be found on the website as well. We are using the Zynq 7000 board which can be seen here. This means that we are using the ZC702 PetaLinux BSP.

Problem 2b

> Next, open up the Zynq-7000 technical reference manual, which is available on the CprE 488 course webpage: http://class.ece.iastate.edu/cpre488/resources/ug585-Zynq-7000-TRM.pdf. Read all 1800+ pages (or just Chapter 6, your choice) and provide a summary, in your own words, of the steps involved in the Zynq boot process. Where does code get executed from, how does the programmable logic get programmed, etc.?

When the S_POR_B reset pin changes the boot begins.

The boot process first begins by loading the BootROM which can configure the system and peripherals for the first stage boot loader. The first stage boot loader is responsible for further setup of the processor with configuring the system memory (DDR RAM), loading the bitstream, and then loads the second stage boot. The second stage boot or the universal boot (U-boot) loads and starts the kernel for our OS (linux). Then the U-Boot will hand off control of the memory and other components to the OS.

As for the locations of these processes.
The BootROM is on the chip and is non writable as it is Read Only Memory. The FSBL code which will be loaded by the BootROM can be on the flash memory or on the SD card. After this point any boot is considered a secondary boot. The U-boot and Linux image are handled independently of the specific hardware.

The programmable logic is defined in the bitstream which will be loaded by the FSBL if it was included.

> Finally, check out the Linux Device Drivers reference, also available on the course webpage: http://class.ece.iastate.edu/cpre488/resources/ldd3.pdf. Specifically, focus on Chapters 1-3 and 13. In your writeup, answer the following three questions:

Problem 2a

> What are the three different classes of devices in Linux? Which class is most appropriate for a USB missile launcher? Briefly defend your answer.

Linux defines devices as either a char module, a block module, or a network module.

The missile launcher would be best served by the character device. This is because we will want to be sending data to the missile launcher at all times in a stream of data rather than as a chunk of data.

Problem 2b

> From a programmer's perspective, what makes driver development different from user-space application development? For example, why can we not call libc functions such as printf()? Why is floating-point code not allowed?

Programming drivers is very different from applications development. This is because you are working directly with memory and run in privileged mode. Also the libc functions are not available because those are created in the user space with the kernel.

Problem 2c

> What is the difference between major and minor device numbers? Does it matter which device number we use for our missile launcher? What is an “urb” in the context of a USB device drive

The major vs minor device numbers differentiate between different devices or different drivers. An example is multiple partitions on one drive. The major device number would tell what device driver such as one for a HDD whereas a minor device number would specify which partition on that HDD.

"urb" stands for USB Request Block. This is used for asynchronous communication between the USB host and the usb device.[4]

References

[1] I2C Programming & Scope Detection

[2] I2C SparkFun

[3] Zynq-7000 Start Up Guide

[4] USB Request Block (URB)

[5] Stages of the Zynq Linux boot process