ENSC351: Real Time and Embedded Systems – Fall 2016

Project:

UART Performance Monitor

Prepared by Graham Holland with content from Eric Matthews

Edited by May Siksik Fall 2016

1 Project Overview

The goal of this project will be to create your own VGA and UART drivers that will be used to create a graphical performance monitor for the UART driver that will run on an embedded Linux system (PetaLinux) [1]. Through working on this project you will be exposed to elements of real-time programming and learn to interface with the hardware system through writing low-level kernel device drivers.

1.1 Quad Organization

Students will be organized into teams of four known as a "quad". Each quad will be divided again into pairs, Group 1 (Pair A) and Group 2 (Pair B). The project work has been split up into two "streams". Each stream consists of four "tasks". Groups will complete all tasks in their "stream". For evaluation, each group/pair will demonstrate completion of their task to a TA. After the demonstration, the two groups in a quad will integrate their work into a single program. This integrated program will serve as the starting point for the next task. See the following table for the distribution of the work.

Table 1 - Project Timeline

(Pair A) Work	(Pair B) Work
Stream 1 Task 1	Stream 2 Task 1
Integrate	
Stream 1 and 2 Task 2	
Stream 1 Task 3	Stream 2 Task 3
Integrate	
Task 4 Integrate (FINAL DEMO)	

1.2 Task Demos

At the end of tasks 1, 2, and 3 students will be required to demonstrate their work. Each pair will do a demo (2 pairs per quad) demonstrating that their code functions properly. Students will also be expected to answer questions about their work.

The task will be marked based on functionality of the code and if it meets required specifications. The code must be well written with good coding practices. Furthermore, each quad member must demonstrate that they understand their task thoroughly and how they implemented it. This will be done through a series of questions about their task.

After the demo for a task is done, the two streams will merge their code and will work from this integrated point for their next task.

1.3 Final Demo

The final demo will consist of the merged code from task 4 in streams 1 and 2. A demo for the individual task 4's in streams 1 and 2 will not be required. The final project will be demoed as a quad with both streams 1 and 2 integrated. The demo will be marked based on functionality and if it reached required specifications. In addition, all four members will be questioned about their respective tasks to demonstrate their understanding.

1.4 Background

Due to the constrained resources of the embedded system used for this project good design and coding practices are a must. As such, in this document, wherever possible, suggestions are made on the approaches to specific problems that are suitable for this system.

As in the labs your project will be run in a Linux environment (PetaLinux) [1] running on an embedded ARM based system on the Zedboard [2]. A high-level diagram showing the main component blocks of the system is given below in Figure 1 . The boxes with blue text are the peripherals that you will be writing device drivers for in this project.

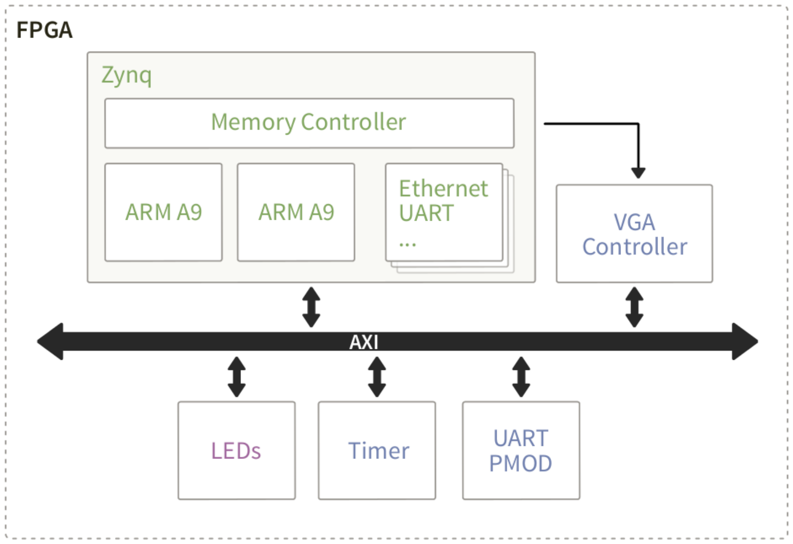

Figure 1 - High-Level Simplified System Block Diagram

2 Stream 1 Task 1: Timer Driver

2.1 Timer Device Driver

In this task you will be developing a simple timer driver to use in your project. You will be required to write a driver for this timer and also an Interrupt Service Routine (ISR).

Deliverables:

- Complete initialization code
- Add ioctl support
- Add interrupt support

Start by reading the data sheet for the AXI Timer/Counter core [3]. Pay special attention to how each register operates, the offset address for each register, and how to enable, disable, and clear interrupts. Using the example code for the LED device driver, write a device driver for the timer. The LED driver only supported reading from a single register at the base address. You will need to add additional support for reading and writing to all of the timer registers. Keeping interrupts disabled write a simple test program that will use your timer driver, so that you can become comfortable with its operation.

2.2 Timer Interrupt Service Routine

Start by reading chapters 6, and 10 of Linux Device Drivers 3rd Edition [4]. Chapter 6 will cover Asynchronous Notification, and Chapter 10 covers Interrupt Handling.

Before we can use an Interrupt we must install a software handler. Interrupt Lines are a limited resource in the system, and if we don't install a software handler the Operating System will simply acknowledge and then ignore any interrupts from our timer. To install a software handler we will use the function `request_irq()`. This function tells the operating system to run our ISR whenever an interrupt is generated by the timer. However, there is a problem with this solution, the ISR that will be run is part of the driver, and cannot transfer or receive data from user-space. It also cannot call any functions that may cause it to sleep. This is a problem since our application runs in user-space.

A solution to this problem is to use Asynchronous Notification (see http://lwn.net/images/pdf/LDD3/ch06.pdf, pag 35). Using Asynchronous Notification our driver ISR will send a signal to our user-level program. This signal will cause a second User-Level ISR to execute. This User-Level ISR can access user-level data, can sleep, and can be interrupted by interrupts.

Hint: AXI frequency is 100MHz

Note: For your reference, You are given the following files to utilize in the development of the driver. Please download them from CANVAS

- timer_ioctl.h: header file defining register offsets, ioctl commands
- timer_test.c: test program

3 Stream 2 Task 1: Preparing a Framework for Graphics Support

<u>Deliverables:</u>

- Support for compositing rectangles of any color (ARGB) to any partially resident location on screen
- Support for compositing any sized image (ARGB) to any partially resident location on screen

For this task you will be creating a software rendering framework for displaying images and text to your screen. You will be provided with an existing VGA driver that you will interface with before the next task in which you will be writing your own driver.

3.1 Background

Before you can start on this task you need a little bit of background on computer graphics, how the image data is stored and how to work in the ARGB colour space. This section will provide you with a very brief background on this area.

3.1.1 Pixel Format

Representing a pixel can be done in many different ways, the most common of which when working with computer screens is RGB format. One common representation is 8 bits for each colour component along with 8 bits for an alpha channel (ARGB). This is the configuration supported by the hardware display block for this project. One of the easiest ways to obtain the raw pixel data of an image in this format is to use GIMP [5], which has an export *Raw image data* option, that writes a binary file with only the pixel data (i.e. no image header information) which will be highly convenient for this part of the project. GIMP has been installed on the lab PCs and is available from the Applications -> Graphics menu.

3.1.2 Alpha Blending

As our chosen pixel format supports an alpha channel we will require that you support a limited form of alpha compositing, that is support for drawing a partial transparent image on top of another image. To do this you don't overwrite the destination pixels, but blend the two with the following equation:

$$P_r = P_{src}r * P_{src}a + P_{dst}r * (1 - P_{scr}a)$$

where the subscript *scr* is for the source pixel and *dst* is for the destination pixel. For this task we only require that you support alpha blending where the destination has an alpha value of one. You are welcome to support the more complex case where the destination surface has an alpha of less than one, but are not required to.

3.2 Image to Memory Mapping

The first step for this task is to have an understanding of how images are typically stored in memory. As an image is two-dimensional and memory is one-dimensional there needs to be a mapping between the two in order to store and access our image in memory. The standard approach to do this is to store the image as a sequential array of rows as illustrated in Figure 2.

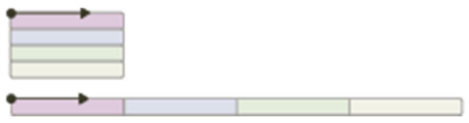

Figure 2 - Memory Layout of an Image

For example, if our image is 100x50 pixels in size then to access the 5th pixel in the 12th row then 5 + 12*100 would give us the pixel location.

3.3 Compositing to the Screen

After familiarizing yourself with how images are laid out in memory, the next thing to consider is how to draw to the screen. The screen's data is stored just like an image with the height and width being the screen size. In order to draw to the screen you need to copy your image data and composite to the correct pixels with the screen. There is an additional complexity here as well however, in that if your image is not entirely resident on the screen, then you must copy only the portion of your image that overlaps the screen. An example of some of the cases you need to consider is given in Figure 3.

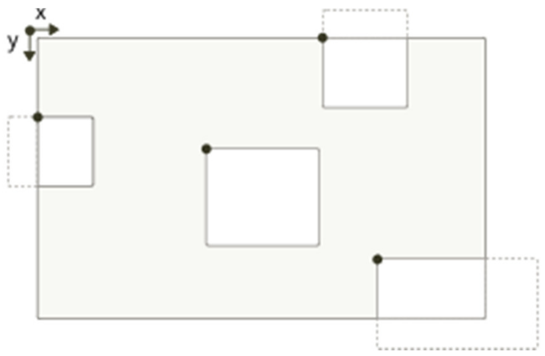

Figure 3 - Drawing an image to the screen

For this task you are required to support drawing an image to the screen with alpha blending, clipping the image contents as needed.

After you use GIMP to export images to RAW format (without any header), in your application you need to open the image file and use mmap to access the image.

```
image_fd = open("/home/root/font_10.raw",O_RDONLY);
fstat (image_fd, &sb);
img.mem_loc = (int*)mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED,
image_fd, 0);
```

The VGA hardware supports a resolution of 640x480 and each pixel has 4 bytes (R, G, B, A). In order to use the provided VGA driver, you need to mmap the opened device file with the appropriate buffer size. The mapping between the buffer and the screen is the same as if it were an image.

```
fd = open("/dev/vga_driver",O_RDWR);
```

```
buffer = (int*)mmap(NULL, BUFFER_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
```

4 Stream 1 and 2 Task 2: Implementing a VGA Driver

Deliverables:

- VGA driver that works as a drop-in replacement of the previously provided VGA driver.

In this task you will be creating a custom VGA driver to replace the one provided to you in the GUI Support task. The driver will allocate the frame buffer in kernel-space and allow access to the frame buffer from userspace through the `mmap` function.

4.1 Background

The VGA driver implementation, in terms of code written, is quite minimal, however the details of the implementation are very important in order to get a fully working VGA driver. The VGA peripheral works by reading a frame of pixels from DDR memory 60 times a second. It has its own hardware interface that is responsible for reading the data that is independent of the processor. This adds a couple complications. First of all, the VGA controller needs its frame to be contiguous in physical memory, so it's required that the memory be allocated in kernel space. Second of all, because both the processor and the VGA hardware will be interacting with memory their views of memory must be kept consistent with each other. As this type of interaction is actually quite common in real systems the Linux kernel provides APIs for this through the `linux/dma-mapping.h` header. For detailed information that you should read before you start on this task, please take a look at chapter 15 of Linux Device Drivers 3rd Edition [4] as it covers the special procedures for allocating memory that is to be used with a hardware DMA block in addition to the use of mmap.

4.2 Device Driver

As the device driver is to work as drop-in replacement for the previous VGA driver the only interface you are required to implement is a mmap function that maps the kernel-space frame buffer to the user-space application. The memory for the frame buffer should be allocated on module loading. The VGA hardware block has a single register which stores the location in physical memory in which the frame buffer exists and is located at offset zero in the VGA peripheral's address space. Once the frame buffer has been allocated the physical address should be written to this register.

4.3 Testing

For testing purposes you can directly use the result of your previous milestone where you developed the image drawing support. During the initial debug phase, you may want to test the device driver internally by drawing to the frame buffer during initialization.

Rubric:

1pt) Students show understanding of what they are supposed to do and are capable to describe their code and the problems they encountered
1pt) The driver is successfully compiled and mounted. The probe function features all required operations
1pt) The mmap operaton is added to the driver and works correctly
1pt) Both Driver and Application code are readable clearly structured, with appropriate functions (i.e. not everything is packaged in the main)
1pt) All operation that may lead to runtime errors (IOCTL commands, file opening operations, critical driver operations) are covered by appropriate safety checks

5 TASK 3: UART Interface from Kermit

For the implementation of both streams of task 3 we will need to use a UART interface connected to our KERMIT console. While you will still load and run all your drivers and applications from the usual ssh interface, you will have to establish a Kermit connection to implement an IO console to your user application.

The connection from your user space application and the Kermit console is realized as follows:

```
#include <termios.h>
[....]

Int serial_fd;
struct termios tio;

memset(&tio, 0, sizeof(tio));
serial_fd = open("/dev/ttyPS0",O_RDWR);
if(serial_fd  == -1)printf("Failed to open serial port... :( \n");

tcgetattr(serial_fd, &tio);
cfsetospeed(&tio, B115200);
cfsetispeed(&tio, B115200);
tcsetattr(serial_fd, TCSANOW, &tio);

// Every time you need to read from the Kermit console you can use
read(serial_fd, &c, 1);
```

Note that every time you will operate a read operation, you will be competing with the serial console for chars. Try to design a way so that Kermit is not stealing any useful char information to your application.

6 Stream 1 Task 3: Creating a Performance Monitor

This deliverable should be deployed by the students that designed the timer driver

Deliverables:

- Collect basic statistics from the operation of the UART including: characters received, characters sent, time from system reset, plus user created metrics
- Utilize from user space the standard OS services and the timer driver code to determine the cycle count of various operations including: average read time, average write time, time from system start, time from console start, plus user created metrics plus user created metrics

This task offers greater flexibility in its implementation than previous tasks, you are required to use ioctl calls to retrieve information from the timer driver, but the exact implementation is up to you. You can also use OS services when you can. Try to be as readable as possible in developing your code, implementing separate functions, careful safety checks (when opening files, processing ioctl commands, etc).

Both the clarity of the code and the originality of your proposal will be considered in the evaluation.

Rubric:

1pt) The program is capable of receiving/sending chars to/from the Kermit console

1pt) The program is capable of receiving/sending chars to/from the Kermit console and count sent and read chars

1pt) All required metrics are collected and displayed in the user application

1pt) User defined metrics are relevant, original, and implemented in a satisfactory fashion

0.5 pt) All safety check are correctly implemented in the application and in the code of the utilized driver modules

0.5 pt) The code is clearly written, structured and readable, appropriately subdivided in functions

7 Stream 2 Task 3: Creating a VGA Console

This deliverable should be deployed by the students that designed the VGA test

Deliverables:

- Support for compositing any sized sub-image from an image (ARGB) to any partially resident location on screen
- Support for displaying ASCII strings on the screen
- Demonstrate use of the VGA screen as a serial console

This task is largely a simple continuation of the graphics library support task. In this task you will be extending that work by adding support for drawing sub-images to the screen and then using this function to develop support for printing ASCII characters to the screen. With this support you will use the VGA display as a console for printing letters received from kermit.

7.1 Working with a Sub-Image

Before we can get to support for drawing strings to the screen you need some additional functionality beyond displaying images to the screen and that is displaying only a sub-image within an image as our text is going to be stored in one large image. Figure 4, illustrates how a sub-image can be accessed from a larger image.

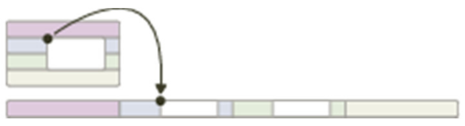

Figure 4 - Accessing a sub-image within an image

7.2 String Rendering

With the ability to render out a sub-image from an image you now have all the tools you need for drawing strings to the screen. The way this is done is to use a specially constructed image where the location of a given character matches its ASCII code, an example of which is shown below: if you like (but it is not mandatory) you can use as reference the file example_192x368.raw available on CANVAS/Files/Lab_Material/Project.

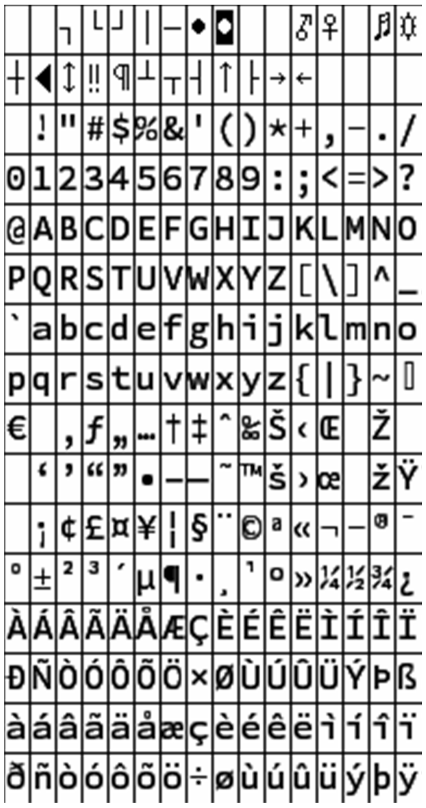

Figure 5 - ASCII Character Sheet

7.3 VGA Console

Once your string drawing functionality is complete, the final step is to use the VGA screen as a console. Create an user-space application that interacts with both your UART and VGA drivers to display received characters to the VGA and to send, and display, characters that are written to the serial port.

The display should support

1) line-wrapping when encountering a newline '\n' or at the end of a line (go to the beginning of following line)
2) screen clear when the screen fills (no need to scroll up, just clear back to empty screen)

8 Task 4: Creating a Graphical UART Performance Monitor

(This is a full single deliverable for the whole group. It includes complete final reporting)

Deliverables:

- Full integration of all Tasks
- Performance metrics displayed onscreen along with VGA console

In this task you will be integrating all previous tasks to create a VGA console that displays runtime performance characteristics of the UART interface.

8.1 Graphical Display

Starting with your UART VGA console, integrate your performance monitoring code. Again, in this task you will be offered some flexibility in how you demonstrate your work. You may choose to integrate the use of escape sequences (e.g. CTRL \#) to allow commands to be passed from the serial console to set the display of various metrics. Another possibility is to reserve a portion of the screen for performance metrics and have the different metrics cycle through at a fixed rate, possibly on a separate thread. The main requirement is that the display still operates as a text based console, with the ability to display all the collected metrics. In addition, you will be asked to discuss and justify the approach taken to display the metrics.

Rubric:

1pt) The program is capable of receiving/sending chars to/from the Kermit console

1pt) Students are able to write letters received via Kermit to screen , but there is no scrolling or newline

1pt) The console correctly implements writing, newline, scrolls back and down at the end of the line

1pt) The screen clears when it is full

0.5 pt) All safety checks are correctly implemented in the application and in the utilized driver modules

0.5 pt) The code is clearly written, structured and readable, appropriately subdivided in functions

9 References

[1] "PetaLinux Tools," [Online]. Available: www.xilinx.com/tools/petalinux-sdk.htm.

[2] "Zedboard," [Online]. Available: zedboard.org.

[3] "LogiCORE IP AXI Timer (v1.02a) Data Sheet," [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v1_03_a/axi_timer_ds764.pdf.

[4] J. Corbet, A. Rubini and G. Kroah-Hartman, Linux Device Drivers, Third Edition, O'Reilly Media, Inc., 2005.

[5] "the GNU Image Manipulation Program," [Online]. Available: http://www.gimp.org.

[6] "LogiCORE IP AXI UART Lite (v1.01a) Data Sheet," [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite_ds741.pdf.