Lab 4.0
Mini-Project

Lab 3.0 + 3.1 – Embedded Linux Systems (recap)

In lab 3.0 you created a full hardware system centered around a dual-core ARM A9 hard processor system (HPS). The processor had many *hard* peripherals (DDR3 memory, SD/MMC controller, Ethernet controller, …) and a few custom *soft* peripherals which you implemented in the previous labs (PWM generators, MCP3204 ADC controller, Lepton thermal camera controller).

In lab 3.1, you used the hardware system developed in lab 3.0 and installed a preloader, bootloader, and finally a Linux-based operating system on the platform (Ubuntu 14.04.5) in order to have software support for many of the ARM processor's hard peripherals. At the end of lab 3.1, you could either use an "on-site" serial console or a remote SSH connection to log into the system and use all its infrastructure ☺.

Lab 4.0 – Mini-Project

Goal

The goal of this mini-project is to write *software* for an embedded Linux system that uses the various peripherals developed in the previous labs. For the sake of proposing a more exotic list of projects, we provide you with 2 new hardware modules in this lab (framebuffer manager & VGA sequencer) as well as a custom Linux framebuffer driver. Collectively, these 3 components allow your Linux applications to access an LCD.

Essentially, the goal of the lab can be summarized in the following steps:

1. Enabling Linux's framebuffer support.
2. Compiling a custom framebuffer driver for our framebuffer manager and VGA sequencer.
3. Loading the custom framebuffer driver into the kernel at runtime.
4. Writing an application which uses the peripherals on the development board and the framebuffer driver to do something cool ☺.

You will not be modifying the new hardware units or framebuffer driver we provide you, but we will still go through some theory in the beginning to motivate their design. Once their purpose is understood, you can then move on towards the actual software implementation of your mini-project.

Note: Although the design of the framebuffer manager and the VGA sequencer is outside the scope of CS-309, we encourage you to take *CS-473 – Embedded Systems* next year to pop their hood and see how these systems are built ☺.

René Beuchat, Philémon Favrod, Sahand Kashani

Please go through the README contained in the lab template before continuing. Start what the README is asking, then let it run in the background while you read the assignment.

Hardware

As can be seen in Figure 1, the PrSoC extension board for the DE0-Nano-SoC has an LCD interface, so it would be great if we could use it.

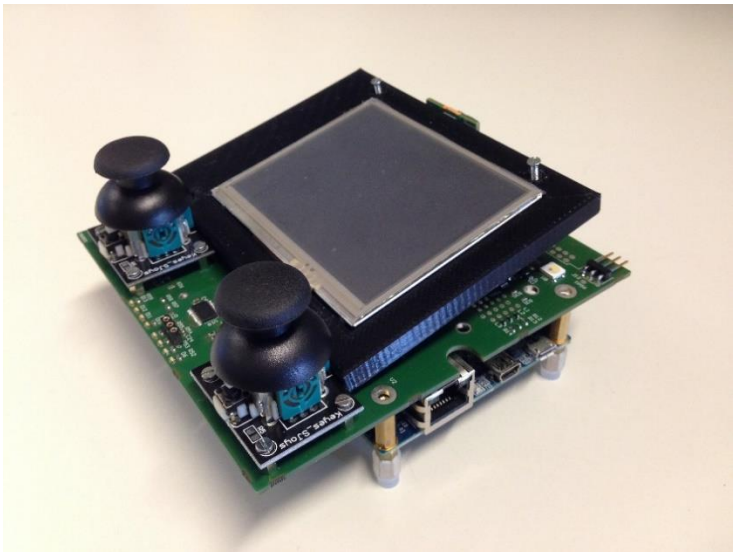


FIGURE 1. PRSOC EXTENSION BOARD LCD INTERFACE
(NOTE THAT 3 LCD INTERFACES ARE AVAILABLE, BUT ONLY 1 CAN BE MOUNTED AT A GIVEN TIME)

As for the peripherals developed in the previous labs (PWM, MCP3204, Lepton), the CPU cannot directly connect to the LCD connected to the FPGA's pins, so we need to create a *programmable interface* in order to interact with it. Displays come in numerous sizes and offer a wide range of functionalities, but their operating principle is fundamentally similar: In general, a display needs to be sent data in a certain format and under specific timings constraints in order for the frame to be successfully displayed.

CPU I/O bottleneck

If we look back at the previous labs, we see that we exclusively employed the CPU to perform all I/O between the programmable interfaces and an application program:

- The PWM generator and the MCP3204 had tiny data footprints as only a few registers were read/written by the CPU to use the devices, so CPU I/O was not a bottleneck.
- The Lepton required more involved I/O as a complete image had to be read back from the Lepton interface's internal on-chip memory. However, given the reduced resolution (80x60) and low frame rate (9 FPS) of the Lepton's sensor, it was still possible for the CPU to read frames as fast as the device would output them.

The situation is quite different for the LCDs as their resolution is larger (320x240 or 480x272) and are able to display frames at much higher data rates. CPUs are fast on compute workloads, but they are very slow on I/O intensive workloads. In a previous study [1, p. 14], the author used the same hardware platform used in the labs and measured the amount of time it took the ARM processor to programmatically write a 320x240 frame to the LCD. The results were clear: The CPU spent 0.05% of its time processing an image in software and 31% of its time writing the image to the LCD. This results in high CPU utilization and low frame rates on the display (≈ 16 FPS), a phenomenon commonly called "lag".

In conclusion, it is crucial that the CPU only limit itself to processing an image and that *another* entity which is optimized for I/O perform the actual data transfer to the LCD.

DMA units

In systems, a special category of devices exists whose sole purpose is to move data efficiently from a source address to a destination address in a given address space. Such devices are called *Direct Memory Access (DMA)* units.

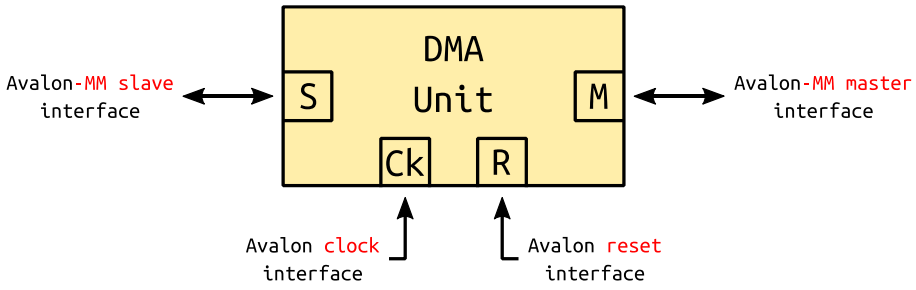


FIGURE 2. GENERIC DMA UNIT

DMA units generally have 2 memory interfaces:

- A *slave* interface which allows the CPU to program the *source address*, *destination address*, and *length* of the desired transfer.
- A *master* interface which the DMA unit uses to efficiently transfer data from the source to the destination.

Solving I/O bottleneck

Essentially, the solution to the I/O bottleneck is for the CPU to process a frame, but to delegate its transportation from memory to the LCD to the DMA unit for efficiency. The table below summarizes the alternative methods of writing a frame to the LCD.

Bad (high CPU utilization + low I/O)	Good (low CPU utilization + high I/O)
1. CPU generates frame in memory. 2. CPU writes frame to LCD.	1. CPU generates frame in memory. 2. CPU programs DMA unit with source address of frame (memory). 3. CPU programs DMA unit with destination address of frame (LCD). 4. CPU programs DMA unit with length of frame. 5. CPU instructs DMA to start. 6. CPU does something else… 7. DMA unit sends interrupt to CPU when transfer to LCD finished.

TABLE 1. COMPARISON OF DATA TRANSFER METHODOLOGIES

Design

As previously stated, designing a DMA unit and an LCD interface is outside the scope of CS-309, so we provide you with the full design of these 2 units for this lab. The high-level block diagram of the DMA unit ("frame manager") and of the LCD interface is shown in Figure 3.

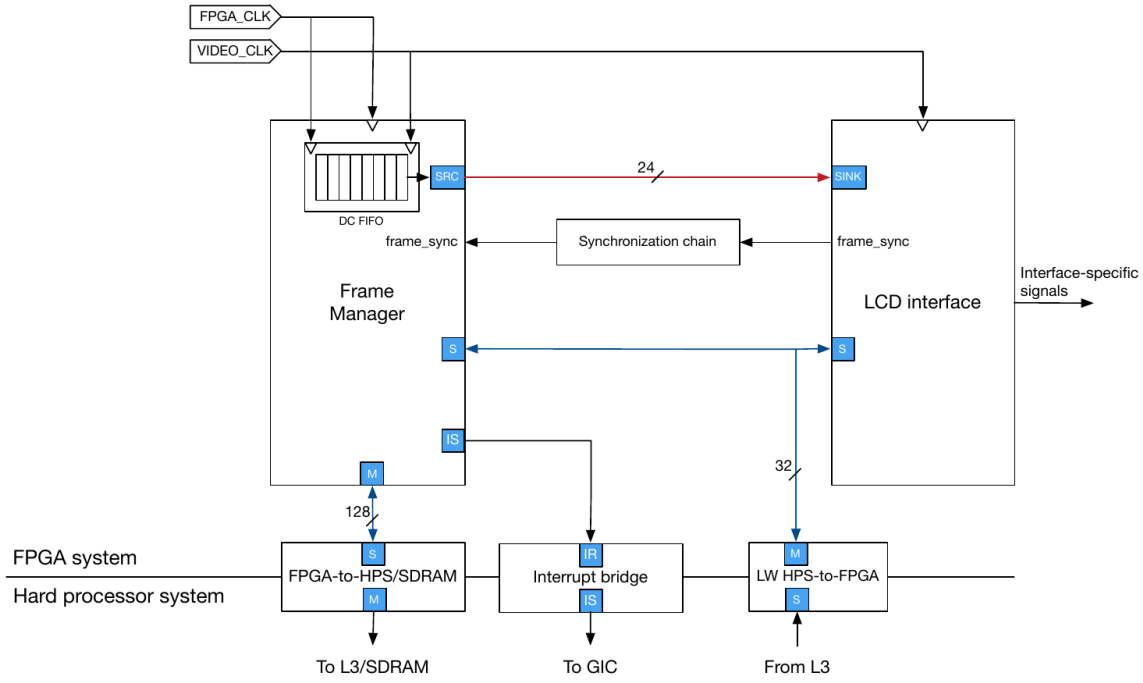


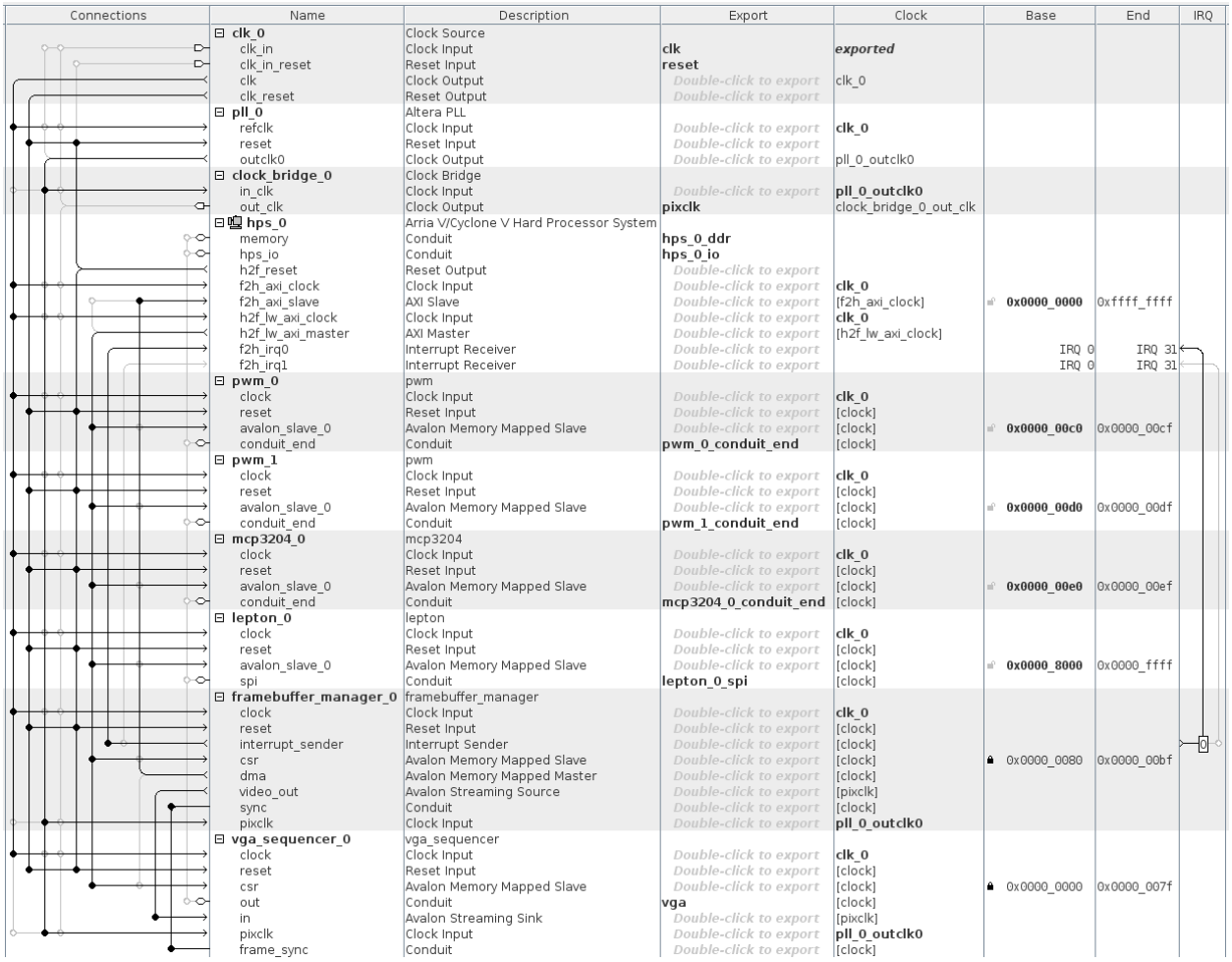
FIGURE 3. VIDEO DMA + LCD INTERFACE (FAVROD, 2016, P. 15)

The frame manager is a generic DMA unit capable of fetching a frame from the ARM processor's memory through the heavyweight FPGA2HPS bridge. The frames are then streamed to an LCD interface which handles the peculiar timings of each LCD device. This modular design allows one to recycle the frame manager in case one decides to change the LCD for another model.

The VHDL sources of the frame manager and the LCD interface can be found under the "`hw/hdl/displays/`" directory in the lab template.

Qsys design

Figure 4 shows the Qsys system developed in the previous labs with the addition of the frame manager and LCD interface.



Connections	Name	Description	Export	Clock	Base	End	IRQ
	clk_0	Clock Source					
	clk_in	Clock Input	**clk**	***exported***			
	clk_in_reset	Reset Input	**reset**				
	clk	Clock Output	*Double-click to export*	clk_0			
	clk_reset	Reset Output	*Double-click to export*				
	pll_0	Altera PLL					
	refclk	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*				
	outclk0	Clock Output	*Double-click to export*	pll_0_outclk0			
	clock_bridge_0	Clock Bridge					
	in_clk	Clock Input	*Double-click to export*	**pll_0_outclk0**			
	out_clk	Clock Output	**pixclk**	clock_bridge_0_out_clk			
	hps_0	Arria V/Cyclone V Hard Processor System					
	memory	Conduit	**hps_0_ddr**				
	hps_io	Conduit	**hps_0_io**				
	h2f_reset	Reset Output	*Double-click to export*				
	f2h_axi_clock	Clock Input	*Double-click to export*	**clk_0**			
	f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_clock]	**0x0000_0000**	0xffff_ffff	
	h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**			
	h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi_clock]			
	f2h_irq0	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
	f2h_irq1	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
	pwm_0	pwm					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_00c0**	0x0000_00cf	
	conduit_end	Conduit	**pwm_0_conduit_end**	[clock]			
	pwm_1	pwm					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_00d0**	0x0000_00df	
	conduit_end	Conduit	**pwm_1_conduit_end**	[clock]			
	mcp3204_0	mcp3204					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_00e0**	0x0000_00ef	
	conduit_end	Conduit	**mcp3204_0_conduit_end**	[clock]			
	lepton_0	lepton					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_8000**	0x0000_ffff	
	spi	Conduit	**lepton_0_spi**	[clock]			
	framebuffer_manager_0	framebuffer_manager					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	interrupt_sender	Interrupt Sender	*Double-click to export*	[clock]			0
	csr	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0000_0080	0x0000_00bf	
	dma	Avalon Memory Mapped Master	*Double-click to export*	[clock]			
	video_out	Avalon Streaming Source	*Double-click to export*	[pixclk]			
	sync	Conduit	*Double-click to export*	[clock]			
	pixclk	Clock Input	*Double-click to export*	**pll_0_outclk0**			
	vga_sequencer_0	vga_sequencer					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	csr	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0000_0000	0x0000_007f	
	out	Conduit	**vga**	[clock]			
	in	Avalon Streaming Sink	*Double-click to export*	[pixclk]			
	pixclk	Clock Input	*Double-click to export*	**pll_0_outclk0**			
	frame_sync	Conduit	*Double-click to export*	[clock]			

FIGURE 4. QSYS SYSTEM WITH LCD INTERFACE

A few comments about the design:

- The 3 LCDs available on the PrSoC extension board all support a VGA interface, hence why the Qsys component of the LCD interface is called “vga sequencer”.
- Each LCD is different and needs to be clocked at a specific frequency. Upon inspection of the appropriate datasheet, we determine that the LCD used in this lab needs to be clocked at 9 MHz. Since there does not exist any 9 MHz clock source on the FPGA (only a 50 MHz clock is available), we need to add a Phase-Locked Loop (PLL) which takes as input the standard 50 MHz clock and creates a 9 MHz output clock for the LCD interface.
- The DMA unit uses interrupts to signal back to the CPU upon completion of a data transfer. Therefore, we need to connect the interrupt output of the DMA unit to the interrupt ports of the ARM processor.

Two such ports are available for interrupts coming from the FPGA fabric (`f2h_irq0` & `f2h_irq1`), our design uses the first one.

Great, we now have all the hardware needed to efficiently display an image on an LCD. The hardware design is now over and we can switch to the relevant tasks for your mini-project.

Software

Linux applications and DMA units

Our hardware system supports efficient mechanisms for transferring frames from memory to an LCD. It is therefore possible for us to write a C program which generates a frame, and then programs the DMA unit following the steps described in Table 1 in order to transfer the frame to the LCD. From a high-level point of view, it seems trivial to program a DMA unit, but there are a few subtle facts one must be aware of since we are programming on an operating system:

- On any application running in Linux, memory space reserved for large buffers is generally allocated on the heap in 4KB chunks called "pages" (the page size is configurable). Furthermore, for protection between processes, the addresses returned by the operating system's memory allocator are *virtual* addresses.
- Pages are *contiguous* in the *virtual* address space, but are *non-contiguous* in the *physical* address space.
- A DMA unit uses *physical* addresses and requires that the data to be transferred be *contiguous* in physical memory.

With all this in mind, programming the DMA unit becomes a non-trivial task as a user space application has no way of determining the physical address of a page or of instructing Linux to allocate the memory contiguously in physical memory.

All the problems above are related to the fact that Linux assigns virtual addresses for all elements in a user space application. There is a way to get a DMA unit working in user space, but it involves manipulating Linux's view of memory before the system boots, and we will not get to that in CS-309 (come to CS-473 if you want to do this).

Instead of the "hack" hinted above, let's try to do things correctly here. All operating systems supporting a GUI have some abstraction for a display. This abstraction comes in the form of APIs describing standard operations supported by all displays. The advantage of using such standard abstractions is that applications which know nothing about the specifics of the display hardware can still use the abstraction's APIs to control the display. In Linux (and most other Oss), the generic term for the memory region used to hold an image to be displayed on a screen is the *framebuffer*. Linux actually supports many different framebuffer abstractions, but we'll aim to use the simplest one on our platform.

The summary of what we need to do is the following:

1. Find a way to enable Linux's framebuffer abstraction

2. Write a Linux framebuffer driver which wraps the implementation details of our framebuffer hardware (framebuffer manager + VGA sequencer) under Linux's standard framebuffer abstraction.
3. Use any application that uses the same framebuffer abstraction as us.
4. Profit ☺

Enabling Linux's framebuffer support

It is undeniable that, starting from the hardware system design, we did many steps in Lab 3.1 before we could actually boot into the Linux system running on the development board. However, if you look back at the steps we performed in order to *configure* Linux, you realize that we didn't much more than what one would have done for a standard C program. All the commands we executed in order to compile Linux were essentially the following:

```
git clone https://github.com/altera-opensource/linux-socfpga.git
cd linux-socfpga
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
make socfpga_defconfig
make zImage
```

The only step we did to truly configure Linux was "`make socfpga_defconfig`". This command set a series of compilation flags which tailored Linux to the *hard* peripherals available on the "socfpga" architecture (the common architecture of all Altera Cyclone V chips). However, this default configuration is very limited and doesn't expose many of the advanced features offered by Linux and which we are accustomed to on our personal Linux machines. A *very* brief list of unsupported features is:

- CPU frequency scaling
- Virtualization
- USB
- …
- *Framebuffer support*

In some sense, it is normal not to enable all these options by default since we are working on an embedded system with a small code footprint, but we need to enable Linux's framebuffer driver and rebuild our kernel before we can have access to it.

Follow the steps below to enable Linux's framebuffer support:

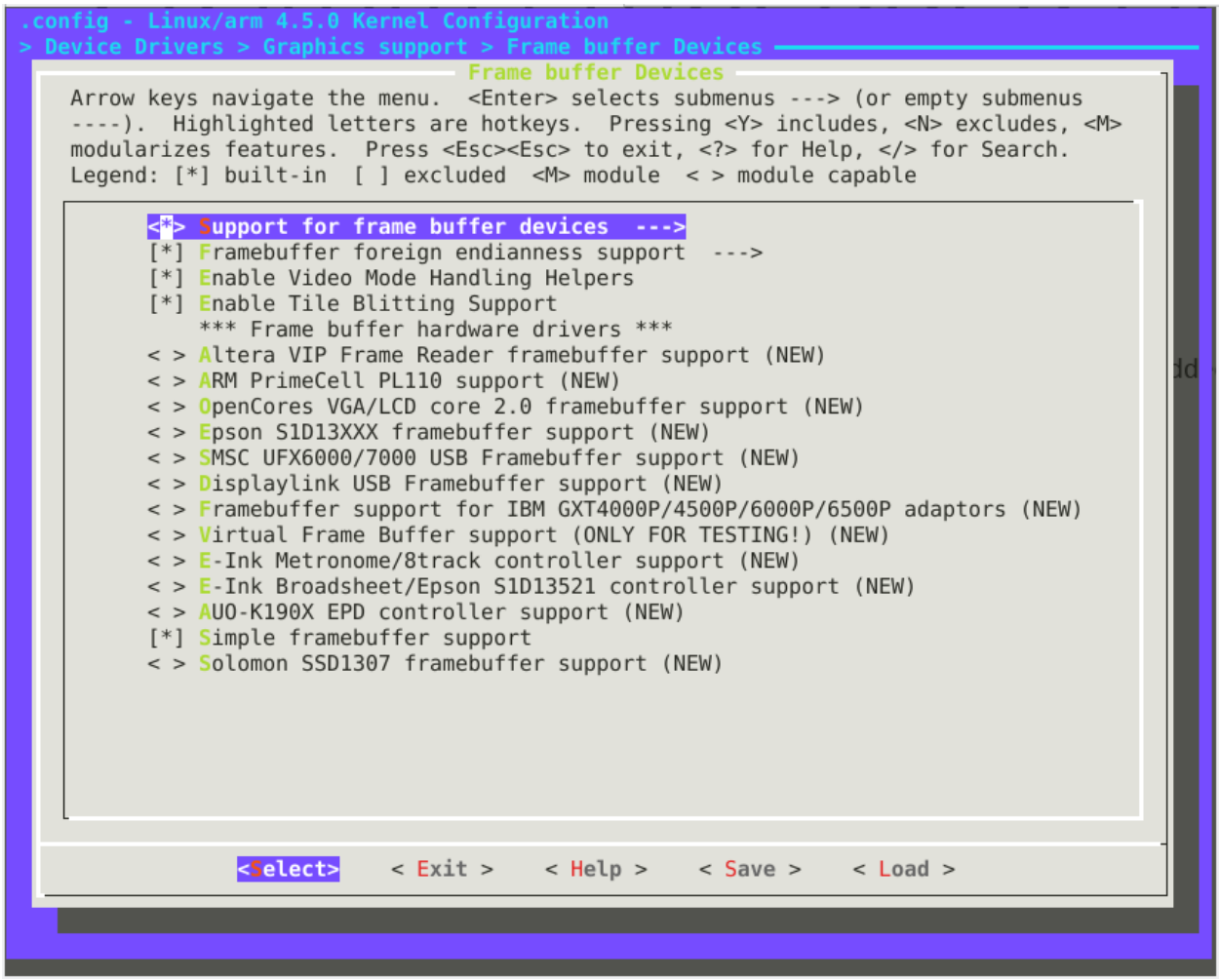
1. Change working directory to Linux source tree directory.
   `$ cd "sw/hps/linux/source"`

2. Compile for the ARM architecture.
   `$ export ARCH=arm`

3. Use cross compiler instead of standard x86 version of gcc.
   `$ export CROSS_COMPILE=arm-linux-gnueabihf-`

4. Manually configure the kernel to enable framebuffer support.
   ```
   $ make menuconfig
   ```

5. Go do Device Drivers → Graphics Support → Frame buffer Devices, and enable the following settings:
   - Support for frame buffer devices
   - Framebuffer foreign endianness support
   - Enable Video Mode Handling Helpers
   - Enable Tile Blitting Support
   - Simple framebuffer support



6. Push ESC a few times until you are asked whether you want to save your configuration before quitting.
7. Save the kernel configuration.
8. Compile kernel.
   ```
   $ make zImage
   ```

9. Copy compiled kernel to associated sdcard directory.
   ```
   $ cp “sw/hps/linux/source/arch/arm/boot/zImage” “sdcard/fat32/”
   ```

At this stage we have an updated kernel with support for framebuffer devices, and have placed the custom kernel in the same location the automated script did.

Compiling a custom device tree

Drivers are board-agnostic, otherwise they would need to be rewritten from scratch for each board. Therefore, a compiled structure called the *Device Tree* is loaded in the kernel's memory by the bootloader. The kernel makes this structure available to its drivers so that they can read their configuration parameters from it. In such a framework, a new board "only" requires a device tree to be written.

One of the first tasks performed by our framebuffer driver is to read the device tree to determine the configuration of the board and of your Qsys design. Some parameters are compulsory, some aren't. For the driver to work, the following parameters must be defined:

- The compatible string[1] set to "prsoc,display";
- The address of the frame manager Avalon-MM slave port as viewed from the HPS;
- The width and height of the screen and the buffer;
- The address of the LCD interface's Avalon-MM slave port as viewed from the HPS; and
- The IRQ number of the frame manager.

The non-compulsory parameter is "prsoc,reg-init". It defines the initialization sequence of the registers in a key-value fashion where the key is the register's offset and the value is what should be written to it. The offsets are computed from the address of the LCD interface's Avalon-MM slave port specified in the device tree above. If this parameter isn't provided, the LCD interface is left as is.

You can find the custom device tree for the ER TFT043 LCD under the "sw/hps/linux/device_tree/" directory in the lab template.

10. Copy the custom device tree to the standard kernel location for all device trees:
```
$ cp "sw/hps/linux/device_tree/socfpga_cyclone5_de0_sockit_prsoc.dts" \
     "sw/hps/linux/source/arch/arm/boot/dts/socfpga_cyclone5_de0_sockit_prsoc.dts"
```

11. Compile the custom device tree:
```
$ make socfpga_cyclone5_de0_sockit_prsoc.dtb
```

12. Copy compiled device tree to associated sdcard directory. Note that U-Boot expects to find a device tree with the name "socfpga.dtb", so we rename our compiled device tree accordingly when copying it to the target destination.
```
$ cp "sw/hps/linux/source/arch/arm/boot/dts/socfpga_cyclone5_de0_sockit_prsoc.dtb" \
     "sdcard/fat32/socfpga.dtb"
```

[1] The compatible string is the driver identifier of a device tree node, i.e. a driver specifies the compatible strings for which it must be loaded. Therefore, if you do not specify it, our framebuffer driver won't be loaded.

We now have an updated device tree containing information relative to the hardware design of our framebuffer manager and VGA sequencer. The information contained in this device tree will be read by our framebuffer driver in order to control the underlying devices.

For more information about the device tree configuration, feel free to look up [2].

Compiling the framebuffer driver

Finally, the last piece of software needed is the framebuffer driver itself. Writing a device driver is also outside the scope of CS-309, so we provide the full source code of the framebuffer driver in "`sw/hps/linux/driver/`".

13. Change directory to where the device driver source code is located.
    `$ cd "sw/hps/linux/driver/fbdev/"`

14. Build the device driver.
    `$ make`

You can find the compiled framebuffer driver under the single file "`prsoc_fbdev.ko`".

Overwrite sdcard fat32 partition contents

15. Since we have updated the kernel and device tree, we need to overwrite these files on our original sdcard. You can just plug the sdcard into your computer and overwrite the zImage and socfpga.dtb files located on the sdcard card with the files contained in the "`sdcard/fat32`" directory.

Testing the LCD!

16. Power up the device, then log in remotely through SSH (or locally with a serial connection).

17. We provide you with some demonstration files you can use to test the system. You can find all sample files in "`sw/hps/application/lab_4_0/displays/`":

    a. batman_320x240.jpg
    b. batman_480x272.jpg
    c. fb_multiple_buffering_example.c
    d. fbv

    The 1st and 2nd files are simply sample images of the most badass superhero which we will project on the LCD. The 3rd file is a demo of how you can use the framebuffer driver to control the LCD from user space. You can base yourselves off this demo in order to design your mini-project. The 4th file is a low-level image viewer which writes input images directly to a framebuffer. It is very useful for testing if the driver works as expected.

18. Compile "fb_multiple_buffering_example.c", then send all demonstration files to the development board. The easiest way would be to just `scp` the files from your host directly to the board.

19. Finally, follow the sequence of commands illustrated in Figure 5 to lookup the properties of the framebuffer driver, insert it into the kernel, then use it through 2 applications. The "`fb_multiple_buffering_example`" demo should cycle through a full-screen red, green and blue display, whereas the "`fbv`" demo should display the image provided in argument directly onto the LCD.

```
[09:38:38] sahand@thinkpad:/hom/sah $ ssh root@10.42.0.2
root@10.42.0.2's password:
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.5.0-00160-gffea805 armv7l)

 * Documentation:  https://help.ubuntu.com/
Last login: Wed May 17 09:38:01 2017 from 10.42.0.1
root@DE0-Nano-SoC:~# ls -lh
total 3.5M
-rw-r--r-- 1 root root 8.3K May 17 09:23 batman_320x240.jpg
-rw-r--r-- 1 root root  57K May 17 09:23 batman_480x272.jpg
-rwxr-xr-x 1 root root 8.5K May 17 09:23 fb_multiple_buffering_example
-rwxr-xr-x 1 root root 3.2M May 17 09:23 fbv
-rw-r--r-- 1 root root 138K May 17 09:24 prsoc_fbdev.ko
root@DE0-Nano-SoC:~# modinfo prsoc_fbdev.ko
filename:       /root/prsoc_fbdev.ko
license:        GPL
alias:          of:N*T*Cprsoc,display
depends:
vermagic:       4.5.0-00160-gffea805 SMP mod_unload ARMv7 p2v8
root@DE0-Nano-SoC:~# lsmod
Module                  Size  Used by
root@DE0-Nano-SoC:~# insmod prsoc_fbdev.ko
root@DE0-Nano-SoC:~# ./fb_multiple_buffering_example
^C
root@DE0-Nano-SoC:~# ./fbv batman_480x272.jpg
```

FIGURE 5. USING THE FRAMEBUFFER DRIVER

If all goes well, you should get a result similar to Figure 6.

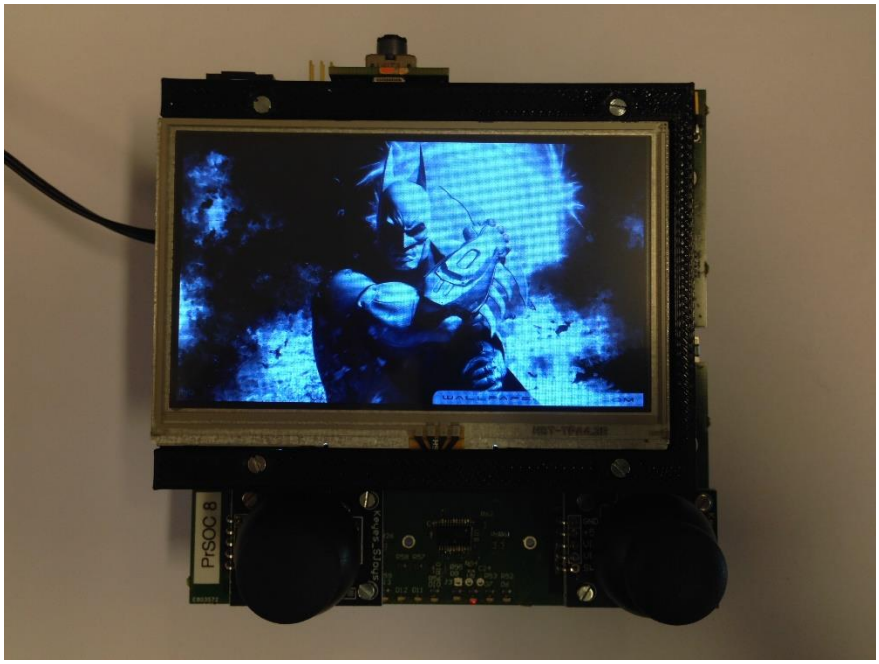


FIGURE 6. ./FBV BATMAN_480X271.JPG

Congratulations, you now have a functional framebuffer driver loaded on your system, and ready to be used. You can now use the framebuffer driver and the various peripherals developed in the previous labs to create a cool embedded Linux demonstration project.

We have provided a short list of project suggestions on Moodle. Feel free to propose modifications or new ideas ☺.

References

[1] P. Favrod, "From FPGA to Linux - An embedded system exploration," 2016.

[2] P. Favrod, "PrSOC - Using the LCDs," 2016. [Online]. Available: https://wiki.epfl.ch/prsoc/lcds.