CSEE 4840
Embedded System Design
Lab 2: Using C, Linux, Sockets, and USB

Stephen A. Edwards
Columbia University

Spring 2020

Code and compile C under Linux on the DE1-SoC board. Implement a primitive Internet chat client that communicates with a server. Receive keystrokes and draw on a framebuffer.

1 Introduction

Unlike the first lab, this lab only involves developing software. We supply a platform on an SD card that consists of Linux running on the ARM processors on the FPGA on the DE1-SoC board. A FPGA configuration adds a video framebuffer.

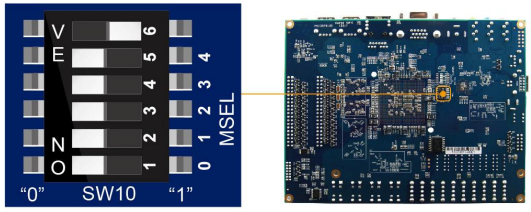
You will implement an Internet-based chat client on this platform. When a user types a line of text on the attached USB keyboard, it will appear on the video display. When s/he presses *Enter*, the contents of the line should be sent through the Ethernet port to a chat server, which will then broadcast it to all its connected clients. You can set up a chat server yourself and test it with *telnet*, or debug it with your friends.

These instructions are written referring to the workstations in 1235 Mudd, which are named `micro01.ee.columbia.edu` through `micro34.ee.columbia.edu`. This lab can be done using your own laptop, but you may have to install terminal emulation software. You will also need a VGA monitor, a USB keyboard, and an wired Internet connection.

2 Booting the Board

Set the FPGA configuration mode switches (SW10, on the underside of the board) to 100000.

This setting for the MSEL switches instructs the FPGA to accept its configuration from the ARM processors.



If this is set differently, the system may not boot, or may boot but not produce video.

We provided you with pre-flashed micro SD cards with the lab 2 environment, but you may also flash your own. Download *lab2-img.tar.gz* from the class website, unpack it to create the (sparse) 16 GB *lab2-16G.img* file, and then flash it using *dd* (slow) or *bmaptool* (much faster).

Insert the micro SD card for lab 2 into the socket on the board (upper right).

Connect your workstation to the board using the mini USB cable that came with the kit. The connector is at the upper right corner of the board.

On your workstation, start the *screen* terminal emulator in a new window as follows:

```
screen /dev/ttyUSB0 115200
```

This establishes a 115200-baud serial connection to the HPS system on the board through an FTDI USB serial chip. The USB serial port should appear when the cable is connected, even if the board is not yet powered on.

Control-a k will terminate *screen*, or just unplug the mini USB cable.

Power on the board. You should quickly see boot messages that include

```
U-Boot SPL 2013.01.01 (Jan 12 2019 - 19:40:48)
BOARD : Altera SOCFPGA Cyclone V Board
CLOCK: EOSC1 clock 25000 KHz

reading u-boot.img


U-Boot 2013.01.01 (Jan 12 2019 - 19:41:00)
CPU   : Altera SOCFPGA Platform

  Hit any key to stop autoboot:  5
```

Let the boot continue. It should configure the FPGA and start the kernel (*zImage*):

```
reading u-boot.scr
226 bytes read in 4 ms (54.7 KiB/s)
## Executing script at 02000000
reading soc_system.rbf
7007184 bytes read in 344 ms (19.4 MiB/s)
## Starting application at 0x3FF79598 ...
## Application terminated, rc = 0x0
reading zImage
4877224 bytes read in 240 ms (19.4 MiB/s)
reading soc_system.dtb
31245 bytes read in 7 ms (4.3 MiB/s)
## Flattened Device Tree blob at 00000100
   Loading Device Tree to 03ff5000, end 03fffa0c ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.19.0 (sedwards@zaphod) (gcc version 6.2.0
(Sourcery CodeBench Lite 2016.11-88)) #5 SMP Sat Jan 19 01:44:12 EST 2019
[    0.000000] CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7),
```

The kernel should eventually mount the root directory on the SD card and start */sbin/init*:

```
[    1.835185] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data
mode. Opts: (null)
[    1.843283] VFS: Mounted root (ext4 filesystem) on device 179:2.

[    1.885157] Run /sbin/init as init process
```

Soon, Ubuntu will start running and the messages will start looking like

```
Welcome to Ubuntu 16.04.5 LTS!

[  OK  ] Listening on Journal Socket (/dev/log).
```

and will eventually present a login prompt:

```
Ubuntu 16.04.5 LTS de1-soc ttyS0

de1-soc login:
```

Login as `root` with password `CSee4840!` Change the password by running *passwd*.

Connect your board to the network using an Ethernet cable, then start the network:

```
root@de1-soc:~# ifup eth0
```

The system will report some *DHCPDISCOVER* messages followed by *DHCPOFFER* and *DHCPACK*. You can force the system to always start *etho* on boot by adding "`auto eth0`" to */etc/network/interfaces*, but only do this if you will always be connected to the network.

By default, Linux thinks your terminal is only 80×24; this may be changed by *stty*, e.g.,

```
stty rows 43
stty cols 132
```

3 Installing Development Software

By design, the lab 2 SD card image includes very little; you need to add additional software to complete lab 2. You should only need to do this once.

Connect your board to the network, configure the network interface, update package information, and bring everything up-to-date.

```
ifup eth0
apt update
apt upgrade -y
```

For lab 2, install the C compiler, *make*, and *libusb*:

```
apt install -y gcc make libusb-1.0-0-dev
```

You will probably want to install a terminal-based text editor. Here are two options:

```
apt install -y nano
apt install -y vim-tiny
```

You may also install the larger *vim* or *emacs-nox* packages.

Scp is convenient for copying files to and from the DE1-SoC board (via Ethernet). You can install it with

```
apt install -y openssh-client
```

Wget is convenient for getting files from the class webpage:

```
apt install -y wget
```

Finally, you can recover some space after these packages are installed with

```
apt clean
```

4 Compiling and Running the Skeleton Lab 2 Files

First, connect a USB keyboard and VGA monitor to your board. Both kinds of connectors are along the top of your board (VGA uses a rounded trapezoid holding 19 pins).

Copy the *lab2.tar.gz* file to your board. You can download it directly from the class website with *wget*:

```
wget http://www.cs.columbia.edu/~sedwards/classes/2020/4840-spring/lab2.tar.gz
```

You can also use *scp* to copy files from your workstation (provided it is running an SSH server) or simply copy the file onto your SD card after you mount it on your workstation.

Next, unpack and edit the provided lab2 skeleton:

```
tar zxf lab2.tar.gz
cd lab2
vi lab2.c
```

Put your name(s) in the comments at the beginning of *lab2.c*.

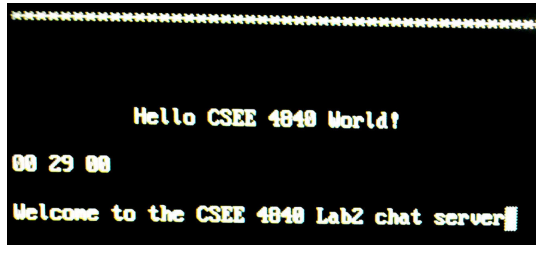
Set the *SERVER_HOST* value to the IP address of the chat server you are going to use. We will try to keep a server running on *micro32.ee.columbia.edu* whose address is 128.59.64.152; you can also run your own server (the *chat_server.py* Python script), but update the script to indicate the IP address of your server machine and make sure the port of the chat server (42000) is not blocked by a firewall.

Next, compile and run the skeleton code:

```
root@de1-soc:~/lab2# make
cc -Wall   -c -o lab2.o lab2.c
cc -Wall   -c -o fbputchar.o fbputchar.c
cc -Wall   -c -o usbkeyboard.o usbkeyboard.c
cc -Wall -o lab2 lab2.o fbputchar.o usbkeyboard.o -lusb-1.0 -pthread
root@de1-soc:~/lab2# ./lab2
Welcome to the CSEE 4840 Lab2 chat server
```

On the VGA monitor driven by the board, you should see a "hello world" message.

When a key is pressed on the USB keyboard, this skeleton client will display three hexadecimal numbers indicating the message received.



This skeleton client also displays messages received from the chat server.

The skeleton client will quit (return to a command prompt) if you press Esc on the keyboard.

If you get an error like

```
root@de1-soc:~/lab2# ./lab2
Error: connect() failed.  Is the server running?
```

there may not be a chat server running, you may have the wrong *SERVER_HOST* value, your chat server may be behind a firewall, or something is wrong with the board's connection to the network.

5 The Framebuffer

A framebuffer is a region of memory that is displayed as pixels on a monitor. For this lab, we are supplying you with an FPGA configuration and Linux kernel with a framebuffer device named `/dev/fb0`.

To use this device in a user-level program, open the device file and call `mmap(2)` to make it appear in the process's address space. In *fbputchar.c*, the *fbopen()* function does this for you. Also in this file is the *fbputchar()* function, which displays a single character on the screen, and *fbputs()*, which displays a string. See *lab2.c* for a simple demonstration of their use.

Once mapped, the framebuffer memory appears as a sequence of pixels in the usual raster order: the upper left pixel appears first, followed by the one just to its right. The next row of pixels starts immediately after the first row ends.

Each pixel is a group of four bytes; the first three represent red, green, and blue intensities; the fourth is unused.

For this lab, you may want to add functions that clear the framebuffer, scroll a region of the framebuffer (consider using *memcpy()*), draw lines, etc. You may also want to modify *fbputchar()* to use different colors, a different font, etc.

6 Networking

We will use Internet protocols to communicate to and from a chat server. Each computer connected to the Internet has a numeric IP address; the `micro01` workstation is "128.59.64.121". Within each computer, servers communicate on ports, which are numbered starting from 1. For example, webservers listen on port 80 and *ssh* uses port 22. Our chat server uses port 42000.

"Sockets" is the standard API for network communcation in Linux. You send and receive data to and from programs on remote computers using *read()* and *write()* system calls.

The *main* function in *lab2.c* creates, opens, and listens to a socket. Abstractly, this looks like

```
// Create an Internet socket
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

// Connect to the server
#define IPADDR(a,b,c,d) (htonl(((a)<<24)|((b)<<16)|((c)<<8)|(d)))
#define SERVER_HOST IPADDR(192,168,1,1)
#define SERVER_PORT htons(42000)
struct sockaddr_in serv_addr = { AF_INET, SERVER_PORT, { SERVER_HOST } };
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

// Write to the socket
write(sockfd, "Hello World!\n", 13);

// Read from the socket
#define BUFFER_SIZE 128
char recvBuf[BUFFER_SIZE];
read(sockfd, &recvBuf, BUFFER_SIZE - 1));
```

Note that each of these functions can fail in various ways and their return values must be checked for errors.

7 USB

We use the *libusb* C library for communicating with the USB keyboard. The USB protocol is rich and complicated, allowing it to work with peripherals as diverse as keyboards, hard drives, and speakers; *libusb* hides many of the details, especially those related to intializing and communicating with the USB controller chip.

USB is a networking protocol like IP, but assumes a simple, tree-shaped network consisting of a single host connected to peripherals and hubs that fan out. While it is possible to directly address the tree structure of the network, *libusb* allows us to ignore it.

To communicate with a USB keyboard, we first have to find its address. Because there are so many kinds of USB devices, we will look at each connected device and determine if it is a keyboard before attempting to receive keystrokes from it.

The code in the *openkeyboard()* function in *usbkeyboard.c* does this: it initializes *libusb*, enumerates all the currently connected devices, then checks each one to see if it is part of the "Human Interface Device" (HID) class and speaks the keyboard protocol (HID devices also include mice). If *openkeyboard()* finds a keyboard, it attempts to connect to it.

In *lab2.c*, keypress events are received from the USB keyboard using the libusb function *libusb_interrupt_transfer()*. This returns an eight-byte packet consisting of a byte indicating which modifier keys (such as Shift) are pressed, an unused byte, and six bytes holding keycodes of pressed keys or 0.

USB keyboards use their own, non-ASCII keycodes. Consult section 10 (page 53) of the USB Implementer's Forum documentation[1] for details.

The skeleton code in *lab2.c* receives and displays the modifier and the first two keycode bytes. For example, when the "A" key is pressed, it displays "00 04 00," and when it is released, "00 00 00." Shift-A produces "02 04 00," and Ctrl, A, and C together give "01 04 06."

[1] https://www.usb.org/sites/default/files/documents/hut1_12v2.pdf

Reading from a socket and reading from the USB keyboard are functions that block, meaning they do not return until new data is available. This is a problem because we must be able to receive messages from other users while we are typing.

A solution is to spawn threads. These are effectively separate program counters within the same program; we can have one waiting for networking communication while the other waits for events from the keyboard.

In *lab2.c*, we spawn one thread to receive data from the network, leaving the main program to handle the USB keyboard. The basic template is this:

```
#include <pthread.h>

pthread_t network_thread;
void *network_thread_f(void *)
{
  // Code to be run "in parallel" with the main program
}

int main()
{
  // Start the network thread
  pthread_create(&network_thread, NULL, network_thread_f, NULL);

  // Do stuff "in parallel" with the network thread

  // Wait for the network thread to terminate
  pthread_join(network_thread, NULL);
}
```

Threads can communicate with each other and the main program through global variables. To avoid race conditions (i.e., where one thread is reading while the other writing), the *pthread* library provides *mutexes* (mutual exclusion constructs) that can be used to enforce exclusive access to global variables.

9 What to Do

Start from the partially working skeleton in *lab2.tar.gz* and extend it as follows:

- Make the display work properly and look good. *fbputchar.c* has the framebuffer initialization code and some simple character generation code.
  - Clear the screen when the program starts.
  - Split the screen into two parts with a horizontal line. Have the user enter text on the bottom two rows; use the rest to record what s/he and other users send.
  - When a packet arrives, print its contents in the "receive" region. Don't forget to wrap long messages across multiple lines.
  - When printing reaches the bottom of the area, you may either start again at the top, or scroll the entry region of the screen.
  - Implement a reasonable text-editing system for the bottom of the screen. Have input from the keyboard display characters there and allow users to erase unwanted characters and send the message with return. Clear the bottom area when a message is sent.
  - Display a cursor where the user is typing. This could be a vertical line, an underline, or a white box.
- Make the keyboard input work. Specifically,
  - Convert the USB keycodes into ASCII to display and send them over the network.
  - Make both shift keys work (i.e., do upper and lowercase characters)
  - Make the left and right arrow keys work
  - Make the backspace key work
- Complete the network communication
  - When your client receives a packet from the server, display it on the next line at the top of the screen.
  - When the user presses return, have your client send to the server the text s/he has been typing and display it in the text area at the top of the sceen.

10 What to turn in

Put you name and UNI in the comments in the *lab2.c* file.

Run *make lab2.tar.gz* on the board in your *lab2* directory to collect all the source code, and submit your *lab2.tar.gz* via Courseworks.

Demonstrate your working lab2 to a TA during his/er office hours.