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

Stephen A. Edwards
Columbia University

2015

Learn how to code and compile C under Linux on the SoCKit board. Implement a primitive Internet chat client that communicates with a server. Draw text on a framebuffer and receive keystrokes

1 Introduction

Unlike the first lab, this lab only involves developing software. We supply a platform consisting of Linux running on the ARM processors on the FPGA on the SoCKit board. A FPGA configuration adds a simple video framebuffer.

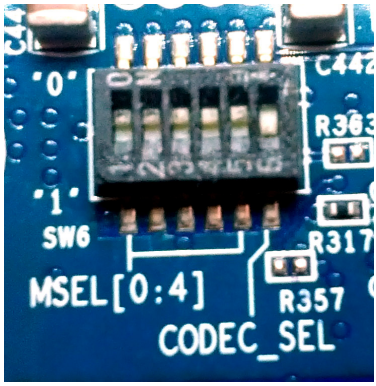
You will implement an Internet-based chat client on this platform. When a user types a line of text on the attached USB keyboard, it will appear on the video display. When s/he presses *Enter*, the contents of the line should be sent through the Ethernet port to a chat server we have running on the SoCKit board network we have running in the lab, which will then broadcast it to all the other connected chat clients.

2 Booting the Board

Set the FPGA configuration mode switches (SW6, on the underside of the board) to 000001.

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

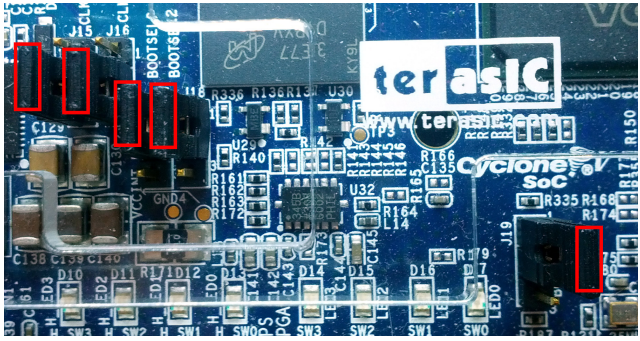
If this is set differently, the system will likely not produce video but may work otherwise.



Set jumpers J15–J19 such that BOOTSEL[2:0] is 111 and CLKSEL[1:0] is 00.

BOOTSEL controls how the ARM processor boots; 111 sets it to boot from the onboard serial flash "QSPI."

BOOTSEL is set differently, nothing will appear on the serial debugging port and Linux will not start.



Power on the board. After a few seconds, start the *screen* terminal emulator on the desktop workstation by typing in a shell window

```
screen /dev/ttyUSB0 57600
```

This establishes a 57600-baud serial connection to the HPS system on the board through the right USB connector on the board (which appears as */dev/ttyUSB0*). If you do this before you power on the board, *screen* will report that it "cannot find the PTY."

You should see a countdown followed by boot messages that begin something like

```
Waiting for PHY auto negotiation to complete. done
ENET Speed is 100 Mbps - FULL duplex connection
BOOTP Broadcast
DHCP client bound to address 192.168.1.104
*** Warning: no boot file name; using 'C0A80168.img'
```

and finally end with

```
Config file found
lab2:   lab2
                kernel: 4840-2015/lab2/uImage
                append: console=ttyS0,57600 root=/dev/nfs rw nfsroot=192.16
                initrd: 4840-2015/lab2/soc_system.rbf
        fdt: 4840-2015/lab2/socfpga.dtb
Enter choice:
```

Type `lab2` and Enter to start booting Linux on the SoCKit board. The board uses TFTP to download three files: *soc_system.rbf*, a configuration for the FPGA that includes the framebuffer; *uImage*, the Linux kernel; and *socfpga.dtb*, a small file with information about peripherals.[1]

Loading these files looks like

```
TFTP from server 192.168.1.2; our IP address is 192.168.1.104
Filename '4840-2015/lab2/soc_system.rbf'.
Load address: 0x2000000
Loading: ##################################################################
```

Then, the Linux kernel should boot and print messages that begin

```
Starting kernel ...

Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset
Linux version 3.8.0-00111-g85cc90f (sedwards@zaphod) (gcc version 4.6.3
(Sourcery CodeBench Lite 2012.03-56) ) #2 SMP Tue Feb 4 21:04:47 EST 2014
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=10c5387d
```

Finally, it will mount the root filesystem and give a shell prompt. Run *ntpdate* to set the time.

```
VFS: Mounted root (nfs filesystem) on device 0:11.
devtmpfs: mounted
Freeing init memory: 188K

Last login: Thu Jan  1 00:00:08 UTC 1970 on ttyS0
root@linaro-nano:~# ntpdate pool.ntp.org
 7 Feb 03:26:02 ntpdate[933]: step time server 66.175.209.17 offset 1423277
root@linaro-nano:~#
```

[1] See the "How Linux Boots on the SoCKit Board" document on the class website for more details.

3 Compiling the Skeleton Lab 2 Files

Unpack the *lab2.tar.gz* file[2] in the root directory on the SoCKit board and rename it according to your UNI ("uni1234" below):

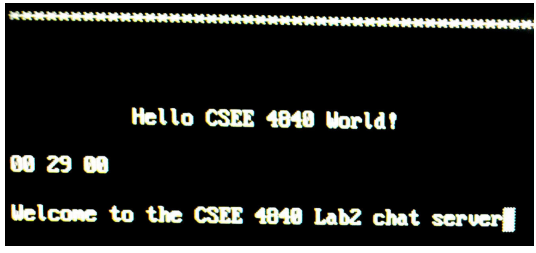
```
root@linaro-nano:~# cd
root@linaro-nano:~# tar zxf lab2.tar.gz
root@linaro-nano:~# mv lab2 uni1234
root@linaro-nano:~# cd uni1234
root@linaro-nano:~/uni1234# ls
Makefile  fbputchar.c  fbputchar.h  lab2.c  usbkeyboard.c  usbkeyboard.h
root@linaro-nano:~/uni1234# make
cc -Wall   -c -o lab2.o lab2.c
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
```

Now, run the initial lab2 code:

```
root@linaro-nano:~/uni1234# ./lab2
Welcome to the CSEE 4840 Lab2 chat server
```

On the display driven by the SoCKit board, you should see a "hello world" message.

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



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@linaro-nano:~/uni1234# ./lab2
Error: connect() failed.  Is the server running?
```

it probably means that the chat server is not running on the server in the lab. Ask the instructor or a TA to restart it.

[2] *lab2.tar.gz* can also be downloaded from the class website

4 Editing and Saving Files

Your source files should appear on your workstation in a directory in */sockit/lab2/root/*. Edit them using your favorite editor (e.g., emacs, vi, nano) on the workstation. Because the */sockit/lab2/root/* directory is shared between the SoCKit board and your workstation, any modifications you make to files on your workstation will be instantly visible on the SoCKit board.

On the SoCKit board, run *make* to recompile your files and *./lab2* to run them.

When you are done working for this session, run *make lab2.tar.gz* in your directory under */sockit/lab2/root/* on your workstation and copy the *lab2.tar.gz* file to your home directory on the workstation.

Note: To be able to read your newly made *lab2.tar.gz* file on the workstation, you may need to change its ownership or permission on the SoCKit board with *chown* or *chmod*.

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 that supplies 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, representing red, green, and blue intensities and an unused byte sometimes used to represent transparency.

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; our chat server is "192.168.1.1". 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 all 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 nearly 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 have to find it. Because there are so many kinds of USB devices, we need to look at each one 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[3] 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."

[3] http://www.usb.org/developers/hidpage/Hut1_12v2.pdf

8 Threads

Reading from a socket and reading from the USB keyboard are blocking, meaning the various reader functions 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* (on the SoCKit boards and on the class homepage). 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.
  - Separate the screen into two parts with a horizontal line between. Use the bottom two rows as the user's text input area, and the rest of the screen 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

Find an overworked TA or instructor, show him/er your working chat application, demonstrate that it is sending well-formed packets, run *make lab2.tar.gz* on the SoCKit board in your *lab2* directory to collect all the source code, and submit your *lab2.tar.gz* via Courseworks.