Lab 7
Using *mmap* to Control Hardware and *systemd* for Auto Startup

EELE 467
SOC FGPAS I
HARDWARE-SOFTWARE CODESIGN

Assignment Date: 11/5/2019
Due Date: 11/12/2019

In this lab, we will write a Linux user space application that uses **mmap()** to control our LED control custom component. This application will directly read and write to the component's registers. We will then use **systemd** to run our application automatically upon boot.

When developing custom hardware, we will usually want a driver with a convenient API that makes interacting with our custom hardware simple from the software side. Often times, the engineer writing the application code won't be an expert in the hardware and register design, so we want to abstract away those details. In our case, we are both the hardware and software engineer, but we can still make our lives easier by making the driver easy to interact with.

Although we will eventually write a slick custom driver, doing so while we are actively developing the hardware is often unfeasible—the hardware design may still change, and we don't necessarily know if everything works correctly yet. As a first step to verify that the hardware is working properly, we will often just read and write to the registers directly. This is the approach we will take in this lab.

Developers Setup

In Lab 3 you configured your computer so that the DE10-Nano board would boot over Ethernet from files located in your Ubuntu VM. We will be using this setup for Lab 7. The FPGA bit stream configuration file soc_system.rbf was put into the TFTP directory `/srv/tftp/de10nano/AudioMini_Passthrough`. Now that you have you have created your custom LED component, instead of loading in the .sof file via JTAG, you need to convert the .sof file to the soc_system.rbf so it will get loaded when your system boots from your Ubuntu VM.

Converting .sof to .rbf

The .sof file (SRAM Object File) is the file you used to program the FPGA from the Quartus Programmer tool via JTAG. However, the .sof file needs to be converted to a .rbf (Raw Binary File) file format. To make the conversion, follow these steps when you are in Quartus and after the Quartus project has compiled successfully.

1. In Quartus select the menu: File → Convert Programming Files.

2. Select the Programming File Type to be Raw Binary File (.rbf)
3. Select the Mode to be Fast Passive Parallel x16
4. Click on the SOF Data then click Add File and browse to the soc_system.sof file
5. Edit the name of the output file to be soc_system.rbf
6. Click the Generate button
7. Copy soc_system.rbf to the Ubuntu VM directory `/srv/tftp/de10nano/AudioMini_Passthrough` and make sure that it has the appropriate read permissions.

Creating the User Space Application

Our application will use **mmap()** to map physical memory addresses into the virtual address space of the calling process. We will first open `/dev/mem`, which is a file that is an image of the main physical memory. Then we will memory map the base address of our custom component, which will give us a pointer to that memory address; we can then read and write to that memory just like we do with any other pointer. After we're done, we will unmap our component's memory and close `/dev/mem`.

The function prototype of **mmap()** and **munmap()** are shown below.

```
void *mmap(void *addr, size_t len, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

- `void *addr` is the virtual address **mmap** will map the physical memory to; setting this to `NULL` lets **mmap** choose the virtual address.
- `size_t len` is the amount of memory that needs to be mapped (the component address space).
- `int prot` determines access permissions—these flags are bit-wise OR'd.
- `int flags` defines information about the handling of the mapped data—these flag are also bit-wise OR'd.
- `int fd` is the file descripter to map from.
- `off_t offset` is the offset from `fd`.
- Refer to the **mmap** man page by typing `man mmap` into a Linux terminal window.

An outline of typical **mmap()** usage is shown below. COMPONENT_BASE is the base address of your custom component in Platform Designer. COMPONENT_SPAN is how much memory your custom component takes up in the memory address space.

```c
#include <sys/mman.h> // include mmap

// Open /dev/mem as read/write and make the operations synchronous
// Note: synchronous means that the write will flush data and all associated
// metadata to the underlying hardware.  By the time a write returns, data
// will have been transferred to your custom register.
devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);

// check if the file couldn't be opened
if (devmem_fd < 0) {
    // handle error
}

// map our component's base address
// addr=NULL means let the kernel choose the virtual address mapping
// len=COMPONENT_SPAN means the size of the mapping is the span of your component
// prot=PROT_READ | PROT_WRITE means memory may be read and written to
// flags=MAP_SHARED means other processes mapping the same area can see updates
// fd=devmem_fd means use the memory device we just opened
// offset=COMPONENT_BASE means use the base address of the custom component
component_base = mmap(NULL, COMPONENT_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED,
 ↪  devmem_fd, COMPONENT_BASE);

// check if mmap failed
if(component_base == MAP_FAILED) {
    // handle error
}

// do register operations
...

// unmap the component
result = munmap(COMPONENT_BASE, COMPONENT_SPAN);

// check if munmap failed
if(result < 0) {
    // handle error
}

// close /dev/mem
close(devmem_fd);
```

Now that we're familiar with **mmap()**, we'll start creating our application. First, we'll have to generate some header files that tell us where our custom component is located, then we'll configure our build environment, and then we'll write the application.

Creating System Header Files

In order to interact with our component, we need to know where it is in memory. Fortunately, Quartus provides us with a tool, `sopc-create-header-files`, which creates header files with this information. The tool is located in the Quartus install directory at ..\quartus\sopc_builder\bin. This command needs to be executed inside the **embedded command shell** in **Windows**.

1. In the Quartus project root directory, create an `include` directory

   ```
   mkdir include
   ```

2. Create system header files

   ```
   sopc-create-header-files soc_system.sopcinfo --output-dir include
   ```

3. Navigate to the include directory

   ```
   cd include
   ```

4. Look at the `hps_0_arm_a9_0.h` file. This file contains the memory addresses of everything in the system from the perspective of the ARM processor, In particular, look at the defines for your LED control component; we will use those later.

   ```
   cat hps_0_arm_a9_0.h
   ```

   Note: Depending on how the HPS component was named, the name could be hps rather than hps_0.

Configuring the Build Environment

We will be cross-compiling our code for the ARM processor, so we will need to setup our build environment appropriately. In the Box folder located at:
https://montana.box.com/s/utydbgdw9gikmkbm6j2dmak17llp4izu
there is an environment setup script (arm_env.sh) and a Makefile we will use. You will have to download both of these and put them in the Ubuntu VM.

Environment Setup Script

When cross-compiling or using specialized toolchains, it is common to have a script that will setup the appropriate environment variables for you. For example, you may need to use a different `gcc` version and other specific software that you don't have in your `PATH` (`PATH` is an environment variable that tells the OS where to look for executables). Rather than keeping these programs in your `PATH` permanently, it is often preferable to set them up temporarily for a single shell. This is where environment scripts come in: you source them once and then your current shell has all the necessary environment variables set. In fact, you've already used an environment setup script when you launched the Quartus Embedded Command Shell.

Our environment script will export our CROSS_COMPILE environment variable and change the prompt so we can visually tell that we are in the ARM cross-compile environment.

1. Download arm_env.sh from Brightspace and put it in ~/software.
2. Navigate to the software directory

   ```
   cd ~/software
   ```

3. Source the script to setup the environment

   ```
   source arm_env.sh
   ```

   You should now see that the prompt looks different, which indicates the script worked correctly.

Makefile

The Makefile template on Brightspace is a fairly generic Makefile that can be used to compile code for x86 and ARM at the same time. It can also be used to compile for just x86 or ARM with make x86 or make arm, respectively. There is also a help target you can run with make help to list all available targets.

Our code will include the hps_0_arm_a9_0.h header file, which lives in a different directory than our code. In our Makefile, we can add "include directories" which tell gcc where to look for include files; we will add the directory where hps_0_arm_a9_0.h lives as an include directory.

Let's configure our Makefile.

1. Create a directory for your application

   ```
   mkdir ~/software/led_control_mmap
   ```

2. Navigate to that directory

   ```
   cd ~/software/led_control_mmap
   ```

3. Download the Makefile from Brightpsace and put it in your led_control_mmap directory.
4. Add the include directory for hps_0_arm_a9_0.h to the Makefile; this file is assumed to be in your Quartus project directory, which is shared in Virtual Box. To do this, change the line

   ```
   INCLUDE_DIRS=.
   ```

   to

   ```
   INCLUDE_DIRS=. /media/sf_quartus_project/include
   ```

5. Change the name of the executable to led_control_mmap

   ```
   EXEC=led_control_mmap
   ```

6. Change the name of the source file to led_control_mmap.c

```
SRCS=led_control_mmap.c
```

Now when you run make, `gcc` will look in all of the directories in `INCLUDE_DIRS` for header files.

This Makefile also puts the ARM executable in the DE10-Nano's rootfs on the VM. Before we can use this Makefile, we have give our user permissions to write to the rootfs.

Rootfs Permissions

To give ourselves proper permissions to write to `/root/software` in the rootfs, we will create a `dev` group, add our user to it, then give that group permissions to `/root/software`.

1. Create `dev` group

   ```
   sudo groupadd dev
   ```

2. Add yourself to the group

   ```
   sudo gpasswd -a username dev
   ```

   where `username` is your username

3. Navigate to the rootfs directory

   ```
   cd /srv/nfs/de10-nano/ubuntu-rootfs
   ```

4. Make a directory for our software binaries

   ```
   mkdir -p root/software
   ```

5. Make `dev` the owning group for everything in `/root`

   ```
   sudo chown -R root:dev root
   ```

6. Give the `dev` group read/write/execute permissions on all folders in `/root`

   ```
   sudo find root -type d -exec chmod g+rwxs {} +
   ```

7. Give the `dev` group read/write permissions to all files in in `/root`

   ```
   sudo find root -type f -exec chmod g+rw {} +
   ```

8. Log out and and log back in for the your user to be added to the group.

Now when we run the Makefile, we will be able to automatically put the ARM executable in the ARM's rootfs.

The Application

In ~/software/led_control_mmap, create a file led_control_mmap.c with the content shown in the code listing below, which is also on Brightspace. You will need to fill in the `// pattern logic goes here...` portion. ***Extra credit* will be given to the most concise (in number of lines) pattern logic implementations.**

To compile the code, type `make arm`; this will build the code and put the executable in /root/software in the DE10-Nano's rootfs. The newly built executable will show up on the DE10-Nano immediately. Since we are accessing memory addresses specific to our custom component, we won't want to run the executable on the VM; doing so would likely result in segfaults or destroying some important values in memory. In a larger project, we might set up a dummy mmap() function that would allow us to emulate writing to our custom component on our x86 host; this would help facilitate unit testing if we were doing more complicated operations.

In the serial console on the DE10-Nano, run the code by navigating to /root/software and typing `./led_control_mmap`. You can stop the code by pressing ctrl-c.

NOTE: The define names QSYS_LED_CONTROL_0_SPAN and QSYS_LED_CONTROL_0_BASE are found in hps_0_arm_a9_0.h; these names may differ depending on how you named the component in Platform Designer.

```c
#include <stdio.h>
#include <sys/mman.h>   // mmap functions
#include <unistd.h>     // POSIX API
#include <errno.h>      // error numbers
#include <stdlib.h>     // exit function
#include <stdint.h>     // type definitions
#include <fcntl.h>      // file control
#include <signal.h>     // catch ctrl-c interrupt signal from parent process
#include <stdbool.h>    // boolean types

#include <hps_0_arm_a9_0.h> // Platform Designer components addresses

/**********************
* Register offsets
**********************/
/* Remember that each register is offset from each other by 4 bytes;
   this is different from the vhdl view where each register is offset by 1 word
   when we do the addressing on the avalong bus!
   Also note that your offsets might be different depending on how you assigned
   addresses in your qsys wrapper.

   Here we specify the offsets in words rather than bytes because we type cast
   the base address returned by mmap to a uint32_t*. Thus when we increment that
   pointer by 1, the memory address increments by 4 bytes since that's the size
   of the type the pointer points to.
```

```c
*/
#define HS_LED_CONTROL_OFFSET 0x0
// Define the other register offsets here

// flag to indicate whetehr or not we've recieved an interrupt signal from the OS
static volatile bool interrupted = false;

// graciously handle interrupt signals from the OS
void interrupt_handler(int sig)
{
    printf("Received interrupt signal. Shutting down...\n");
    interrupted = true;
}


int main()
{
    // open /dev/mem
    int devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);

    // check for errors
    if (devmem_fd < 0)
    {
        // capture the error number
        int err = errno;

        printf("ERROR: couldn't open /dev/mem\n");
        printf("ERRNO: %d\n", err);

        exit(EXIT_FAILURE);
    }

    // map our custom component into virtual memory
    // NOTE: QSYS_LED_CONTROL_0_BASE and QSYS_LED_CONTROL_0_SPAN come from
    // hps_0_arm_a9_0.h; the names might be different based upon how you
    // named your component in Platform Designer.
    uint32_t *led_control_base = (uint32_t *) mmap(NULL, QSYS_LED_CONTROL_0_SPAN,
        PROT_READ | PROT_WRITE,MAP_SHARED, devmem_fd, QSYS_LED_CONTROL_0_BASE);

    // check for errors
    if (led_control_base == MAP_FAILED)
    {
        // capture the error number
        int err = errno;

        printf("ERROR: mmap() failed\n");
```

```c
        printf("ERRNO: %d\n", err);

        // cleanup and exit
        close(devmem_fd);
        exit(EXIT_FAILURE);
    }

    // create pointers for each register
    uint32_t *hs_led_control = led_control_base + HS_LED_CONTROL_OFFSET;
    // Define the other register pointers here


    // display each register address and value
    printf("*************************\n");
    printf("register addresses\n");
    printf("*************************\n");
    printf("hs_led_control address: 0x%p\n", hs_led_control);
     // Print the other register addresses here

    printf("*************************\n");
    printf("register values\n");
    printf("*************************\n");
    printf("hs_led_control: 0x%08x\n", *hs_led_control);
    // Print the other register values here


    // set the component into software control mode
    *hs_led_control = 1;

    // clear all of the LEDs
    *led_reg = 0;

    /* run a pattern on the LEDS until we are interrupted with a SIGINT signal.
       The pattern "fills" the LEDS from right to left, i.e.
        00011000
        00111100
        01111110
        11111111
        01111110
        00111100
        00011000
        ... repeat
       where 0 indicates off and 1 indicates on. This pattern repeats.
       Sleep for 0.1 seconds between each pattern-step with usleep(0.1*1e6)

       Extra credit will be given to the most concise (in number of lines)
       pattern logic implementation.
       */
```

```c
    signal(SIGINT, interrupt_handler); // catch the interrupt signal
    while(!interrupted)
    {
        // pattern logic goes here...
        usleep(0.1*1e6);
    }

    // set the component back into hardware control mode
    *hs_led_control = 0;

    // unmap our custom component
    int result = munmap(led_control_base, QSYS_LED_CONTROL_0_SPAN);

    // check for errors
    if (result < 0)
    {
        // capture the error number
        int err = errno;

        printf("ERROR: munmap() failed\n");
        printf("ERRNO: %d\n", err);

        //cleanup and exit
        close(devmem_fd);
        exit(EXIT_FAILURE);
    }

    // close /dev/mem
    close(devmem_fd);

    return 0;
}
```

Writing systemd Units

To run our code automatically at startup, we will use the systemd init system. Essentially, we will create a service that executes our program, and this service will get started by systemd.

We will create the systemd unit file in the **Ubuntu VM**.

1. Navigate to the DE10-Nano rootfs

   ```
   cd /srv/nfs/de10-nano/ubuntu-rootfs
   ```

2. Set this as our new root directory

```
sudo chroot .
```

3. Navigate to to software directory

```
cd /root/software
```

4. Create a bash script `led_control_mmap.sh`

```
#!/bin/sh

# run our application
/root/software/led_control_mmap
```

5. Make the script executable

```
chmod +x led_control_mmap.sh
```

6. Navigate to `/etc/systemd/system`

```
cd /etc/systemd/system
```

7. Create a systemd service file `led_control_mmap.service` with the following

```
[Unit]
Description=Run led_control_mmap application

# this says what the service will do (i.e. start the shell script)
[Service]
Type=simple
ExecStart=/root/software/led_control_mmap.sh

# this indicates that the service will run when reach the login
[Install]
WantedBy=multi-user.target
```

8. Enable the service

```
systemctl enable led_control_mmap.service
```

9. Exit the chroot

```
exit
```

Testing the Service

To make sure the service file works, you can start it manually. On the DE10-Nano, type `systemctl start led_control_mmap` to start the service. You can type `systemctl stop led_control_mmap` to stop the service, and `systemctl status led_control_mmap` to check the status.

If the service properly runs your code, you're ready to test if it's enabled. Reboot the DE10-Nano. When you reach the login, the LEDs should be displaying the pattern.

Instructor Verification Sheet

Make sure you get this page signed and turned in to get credit for your lab

Lab 7

Using *mmap* to Control Hardware and *systemd* for Auto Startup

EELE 467
SOC FGPAS I
HARDWARE-SOFTWARE CODESIGN

Due Date: 11/12/2019

Name : ______________________________

Demo: Show that your software pattern shows up on the LEDs before you login; also show that your hardware control state machine takes over when you stop the program. To do this, perform the following steps:

1. Power up and log into the DE10-Nano and let your software run for a while
2. Find the process id of your application

   `ps aux | grep led_control_mmap`

   `ps aux` shows all of the processes, and `grep led_control_mmap` finds ones with the name `led_control_mmap`. The process id (pid) is found in the second column of the command's output.
3. Send the `SIGINT` signal to the process

   `kill -s SIGINT pid`

   where `pid` is the pid of your process.
4. Observe your hardware control state machine take over control of the LEDs

Verified: ______________________ Date: ______________________