Embedded Systems
Laboratory Exercise 9

Using the ADC to Implement a Simple Oscilloscope

The purpose of this exercise is to use the *AD7928* Analog-to-Digital Conversion (ADC) chip on a DE-series board to implement a simple oscilloscope. An oscilloscope is a device that is often used to display a periodic analog signal on a screen. The oscilloscope allows a designer to observe various characteristics of the displayed signal, such as its amplitude, shape, frequency, rise time, and distortion. In this lab, you will create a simple oscilloscope by writing a C-language program that reads data from the ADC chip.

In this writeup we will assume that you are using the DE1-SoC board, but the same instructions can be applied for other boards such as the DE10-Nano or DE10-Standard. On the DE1-SoC board, your oscilloscope will draw waveforms by using the VGA display in the DE1-SoC Computer. If you are using the DE10-Standard board, then you may wish to display waveforms on its built-in LCD display. To test your oscilloscope, you will write a kernel module that produces a square wave signal on a pin of an expansion connector on the DE1-SoC board. You will use a wire to connect the signal on this pin to an input of the ADC.

Using the ADC

The *DE1-SoC Computer* contains a controller that provides access to the *AD7928* chip. As illustrated in Figure 1, the ADC port comprises eight 12-bit registers starting at the base address `0xFF204000`. The first two registers have dual purposes, acting as both data and control registers. By default, the ADC port updates the A-to-D conversion results for all ports only when instructed to do so. Writing to the control register at address `0xFF204000` causes this update to occur. Reading from the register at address `0xFF204000` provides the conversion data for channel 0. Reading from the other seven registers provides the conversion data for the corresponding channels. The values in the channel data registers range from 0 to 4095 (which is $2^{12} - 1$), which correspond to input voltages of 0 V to 5 V. It is also possible to have the ADC port continually request A-to-D conversion data for all channels. This is done by writing the value 1 to the control register at address 0xFF204004. The *R* bit of each channel register in Figure 1 is used in Auto-update mode. The *R* bit is set to 1 when its corresponding channel is refreshed and set to 0 when the channel is read.

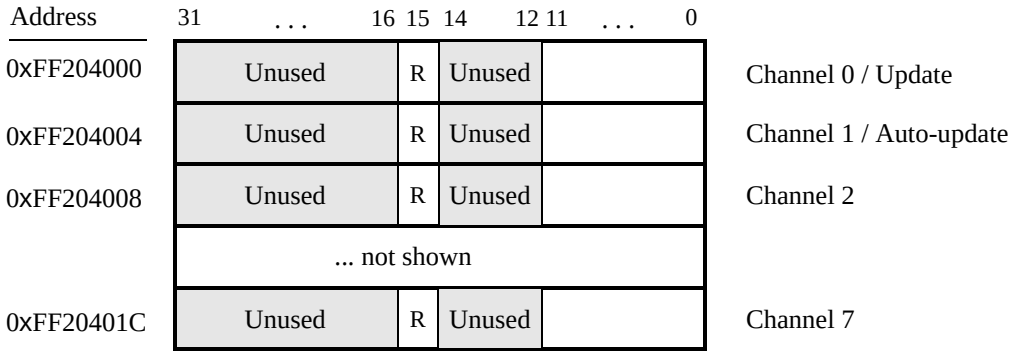

Figure 1: ADC port registers.

Figure 2 shows the connector on the DE1-SoC board that is used with the ADC port. Analog signals in the range of 0 V to the V_{CC5} power-supply voltage can be connected to the pins for channels 0 to 7. This connector is located on the left side of the DE1-SoC board, near the power switch.

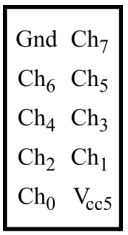

Figure 2: ADC connector.

Part I

Write a C-language program to read the converted digital value on channel 0 of the ADC, and repeatedly print the value to the terminal. To verify that you are correctly reading the values, try the following:

- Use a wire to connect the Ch_0 pin to the *Gnd* pin and see that the value read is equal, or very close, to zero.
- Use the wire to connect Ch_0 to the power-supply value V_{CC5} and verify that the value read is equal, or very close, to 4095.
- Note: to avoid possible damage to your board, be careful not to accidentally connect the *Gnd* pin to the V_{CC5} power pin.

Part II

To test your oscilloscope, you will need to generate a signal to input into your oscilloscope. In this part, you will write a Linux* kernel module that will output a square wave onto a parallel port. Your kernel module should create the waveform on pin D_0 of the parallel port called JP1 in the DE1-SoC Computer. Connector JP1 is one of the large, 40-pin, connectors on the board, and pin D_0 is at the top right corner of the connector.

The frequency of the square wave generated by your kernel module should be configurable between 10 Hz to 160 Hz, in 10 Hz increments. The frequency is selected by the four leftmost switches on the board, SW_{9-6} (SW_{3-0} for the DE10-Nano). An SW_{9-6} value of `0000` would correspond to 10 Hz, `0001` to 20 Hz, and so on. To generate the square wave, your kernel module should use a *hardware timer* to generate interrupts at regular intervals. You can use the interval timer implemented in the FPGA called *FPGA Timer0*. The register interface for this timer has the base address `0xFF202000`. As shown in Figure 3 this timer has six 16-bit registers. To use the timer you need to write a suitable value into the *Counter start value* registers (there are two, one for the upper 16 bits, and one for the lower 16 bits of the 32-bit counter value). To start the counter, you need to set the *START* bit in the *Control* register to 1. Once started the timer will count down to 0 from the initial value in the *Counter start value* register. The counter will automatically reload this value and continue counting if the *CONT* bit in the *Control* register is 1. When the counter reaches 0, it will set the *TO* bit in the *Status* register to 1. This bit can be cleared under program control by writing a 0 into it. If the *ITO* bit in the control register is set to 1, then the timer will generate an ARM* interrupt each time it sets the *TO* bit. The timer clock frequency is 100 MHz. The interrupt ID of the timer is 72. Follow the instructions in the tutorial *Using Linux on DE-Series Boards* to register this interrupt ID with the Linux kernel and ensure that it invokes your kernel module whenever the interrupt occurs.

The timer interval should equal half of the period of the square wave to be generated. Your interrupt handler should invert the value on pin D_0 of JP1 upon every interrupt, creating the desired square wave. Your interrupt handler should check to see if SW_{9-6} have been altered, and adjust the square wave frequency as needed. An illustration of the programmer registers in a parallel port appears in Figure 4. It has a *data* register at an assigned *base* address. Each bit in this register could be either an input or output, depending on the contents of the *direction*

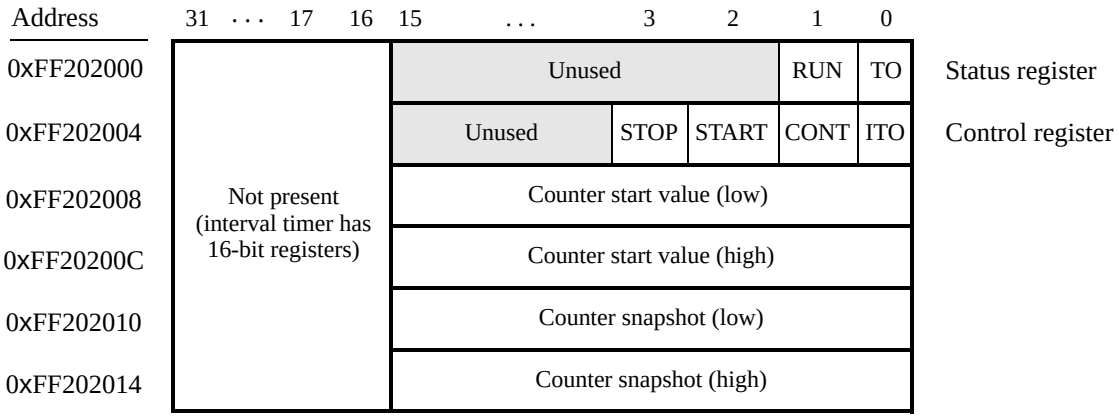

Figure 3: The FPGA Timer0 register interface.

register. Setting a bit in the direction register to 1 makes the corresponding data bit an output, while 0 makes it an input. The *interruptmask* and *edgecapture* registers shown in the figure allow the port to generate ARM processor interrupts. For this exercise, you need to use only the data and direction registers. The base address of port JP1 is `0xFF200060`. Connector JP1 (and JP2) is a 40-pin connector located on the right-hand side of the DE1-SoC board. As mentioned before pin D_0 is at the top right corner of the connector.

Perform the following:

1. Create a file called *signal_generator.c* and write your kernel module in this file.
2. Your kernel module should show the values of the SW switches on the red lights LEDR. Also, it should show the value of the waveform frequency, which is a number between 10 and 160, on the seven-segment displays HEX3-HEX0 (if available on your board). Figures 5-7 show the programming registers in the switch, light, and seven-segment display ports.
3. Create a Makefile, compile your kernel module, and insert it into the kernel.

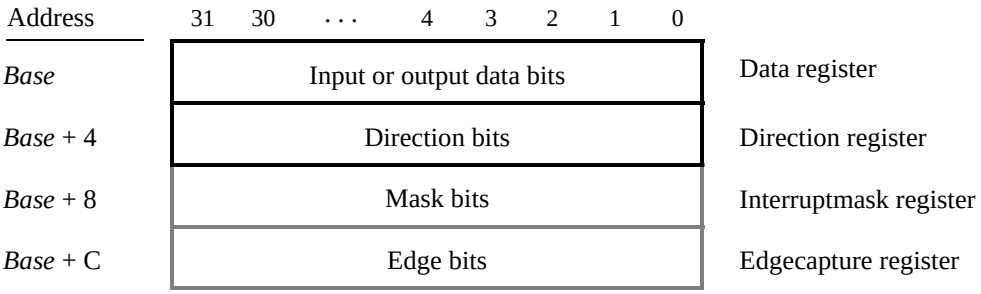

Figure 4: The programming registers in a parallel port.

Part III

In this part you will write a C-language program to implement your oscilloscope. Your program will sample the input waveform on channel 0 of the ADC, and display the signal graphically on the VGA output. The x-axis on the display will represent a timespan of 100 ms, and the y-axis will represent a voltage range of 0 V to 5 V.

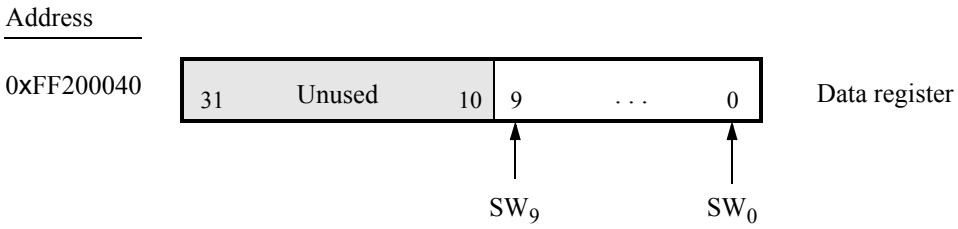

Figure 5: The SW switch port.

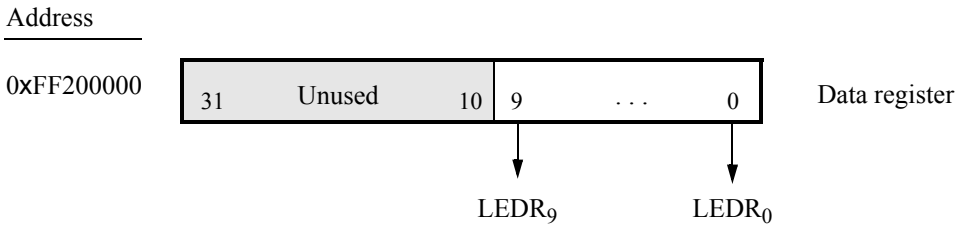

Figure 6: The LEDR light port.

An oscilloscope samples the input signal for some defined period of time, then displays that stretch of recorded data on the screen. This stretch of data capture is referred to as the *sweep*. Your oscilloscope will have a sweep of 100 ms. A sweep should start when the oscilloscope detects one of two triggers: a rising edge of the square wave, or a falling edge of the square wave. Switch SW_0 should control which of the two triggers is to be detected, with position 1 selecting the rising edge, and 0 the falling edge. Upon completing each sweep, your oscilloscope will draw the newly captured waveform on the display, then re-arm the trigger to start another sweep as soon as the next trigger is detected.

During a sweep your oscilloscope should capture one sample for each x coordinate of the VGA display. The samples should be captured in regular intervals. For example, if $x = 320$, then a sample should be captured every $100\ ms/320 = 0.3125\ ms$. To capture data at regular intervals, you will need to use an interval timer like you did in Part II. However, instead of directly controlling a hardware timer using its register interface, you will use a Linux application programmer interface (API) for creating and using timers. This API allows you to create and set an interval timer which will send a *signal* to your program upon each timeout. As part of your program, you can implement a *handler* function for the signal to capture a sample on each timeout. The timer API is provided by the library `<time.h>`, and its use is illustrated in Figure 8. When compiling, you must add the flag "-lrt" to the end of your `gcc` command to link in the *Realtime* library which includes `<time.h>`.

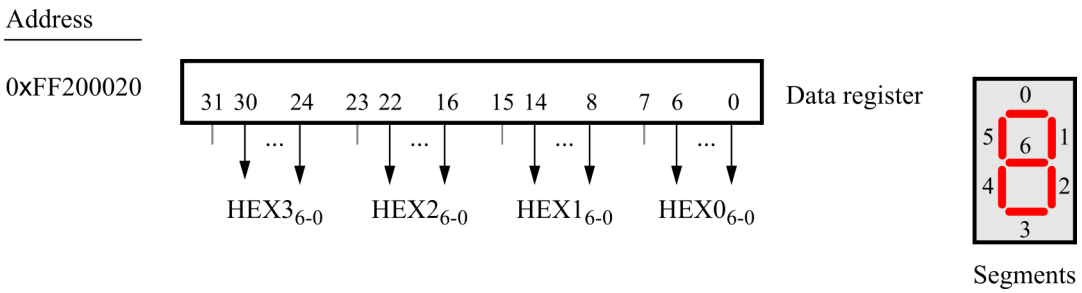

Figure 7: The HEX seven-segment display ports.

Key lines of code are described below:

- Lines 1-2: The header files that provide the timer functions.
- Lines 4-6: The timer specification structure which is used to set the timer to count 0.3125 ms intervals. The `it_interval` variable specifies when the initial timeout occurs after setting the timer (set to 312500 ns, which equals 0.3125 ms). The `it_value` variable specifies the repeating timeout interval after the initial timeout.
- Lines 7-9: The timer specification structure for stopping a timer. Notice that the variables are set to 0, which turns off the timer.
- Lines 12-15: The function that handles a timer timeout. This function is called every 0.3125 ms.
- Lines 19-27: Code that uses the Linux signalling API (provided by `<signal.h>`) to register the function `timeout_handler(...)` as the handler for the `SIGALM` signal. The timer that we will use generates the `SIGALM` signal upon each timeout, thus invoking the `timeout_handler(...)` function.
- Line 30: Creates a monotonically increasing timer. At this point, the timer is inactive.
- Line 33: Arms the timer, making it timeout every 0.3125 ms. Upon each timeout, the timer will send the `SIGALM` signal to this program, which will invoke the `timeout_handler(...)` function.
- Line 36: Stops the timer by configuring it with a zero for both `it_interval` and `it_value`.

In your oscilloscope program, whenever a trigger condition is satisfied you should call the function `timer_settime` as shown in line 33 to start a sweep. After the samples for the sweep have been collected by your timeout handler, you should call `timer_settime` again, as illustrated in line 36, to stop the timer.

Perform the following:

1. Create a C program that implements your oscilloscope.
2. Compile your program, appending the required flag "-lrt" to your `gcc` command so that it links in the *Realtime* library. For instance, if your source-code is stored in a file *part3.c*, then an appropriate command would be `gcc -o part3 part3.c -lrt`.
3. Ensure that pin D_0 of connector JP1 is connected to Ch_0 of the ADC, and that your kernel module from Part II is inserted into the Linux kernel. Run your oscilloscope program and observe the VGA display. Try toggling SW_{9-6} and observe the change in the displayed waveform. The oscilloscope's display should look similar to Figure 9. In this figure, the waveform has a frequency of 100 Hz, as 10 periods of the signal are visible within the 100 ms span. Since the JP1 pins operate at 3.3 V, the waveform spans roughly $\frac{2}{3}$ of the 5 V vertical range.

```c
#include <signal.h>
#include <time.h>
...
struct itimerspec interval_timer_start = {
   .it_interval = {.tv_sec=0,.tv_nsec=312500},
   .it_value    = {.tv_sec=0,.tv_nsec=312500}};
struct itimerspec interval_timer_stop  = {
   .it_interval = {.tv_sec=0,.tv_nsec=0},
   .it_value    = {.tv_sec=0,.tv_nsec=0}};
timer_t interval_timer_id;

// Handler function that is called when a timeout occurs
void timeout_handler(int signo) {
   ... code not shown
}
int main(void)
{
   ...
   // Set up the signal handling
   struct sigaction act;
   sigset_t set;
   sigemptyset (&set);
   sigaddset (&set, SIGALRM);
   act.sa_flags = 0;
   act.sa_mask = set;
   act.sa_handler = &timeout_handler;
   sigaction (SIGALRM, &act, NULL);

   // Create a monotonically increasing timer
   timer_create (CLOCK_MONOTONIC, NULL, &interval_timer_id);
   ...
   // Starting the timer
   timer_settime (interval_timer_id, 0, &interval_timer_start, NULL);
   ...
   // Stopping the timer
   timer_settime (interval_timer_id, 0, &interval_timer_stop, NULL);
}
```

Figure 8: Using the timer functions.

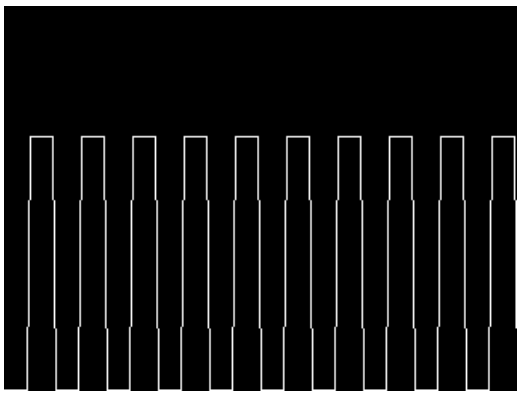

Figure 9: Screenshot of the oscilloscope program's video output.

Part IV

Oscilloscopes usually have a sweep control function that allows a user to control the sweep duration. Add such a function to your oscilloscope, by using the KEY inputs of the DE1-SoC Computer. Pressing KEY_0 should cause the sweep time to increase by 100 ms, and pressing KEY_1 should decrease the sweep time. Figure 10 shows the programming registers in the KEY port. When a KEY is pressed, the corresponding bit-position in both the *Data* and *Edgecapture* registers is set to 1. The *Data* register bit is cleared as soon as the KEY is released, while the *Edgecapture* register bit remains set until manually cleared under program control. An *Edgecapture* register bit can be cleared by writing a 1 into it.

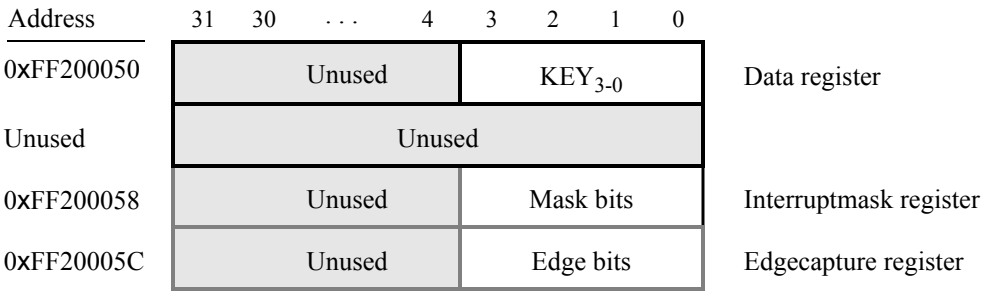

Figure 10: The KEY port.