TDT4258 Microcontrollers System Design
Exercise 1

Interrupt-Based I/O in Assembly Language on a Microcontroller

Vegard Edvardsen
Jean Niklas L'orange
Caroline Sæhle

February 20, 2012

Abstract

Many products today need an embedded system in order to work as intended. They take data from some input, process the data on a microprocessor, and possibly sends data on some output-port if that is the desired functionality. This report investigates how the AVR32 microprocessor from Atmel handles I/O and interrupts at assembly level, as well as how programming such a microprocessor works. This was done by creating a small program in assembly code where the user is able to control the position of an LED-paddle by pressing buttons. We also learnt how and why bouncing occurs and how we can prevent the errors it can cause through debouncing. The report also explains how to set up and enable interrupts, and explains instruction behaviour which people not familiar with assembly language may not know about or may forget during programming.

Contents

1 Introduction

We were given the task of making "a program which enables a player to control a 'paddle' on the row of LED-diodes", as stated in the compendium given in the course Microcontroller System Design (TDT4258) [3]. The main goal of this task was to get a good understanding in how to program I/O and interrupt handling in assembly code for the AVR32 microprocessor, as well as understand and learn how to use the GNU-toolchain that is used to assemble, link and debug this microprocessor.

To solve the task given, we built the solution incrementally. In this way, we could easily test whether our program worked well or not: If there were some errors, we would be able to find the error in the small amount of lines added since last time. It made us also repeat the upload and debug stage multiple times, which made us more proficient in using the tools and made it easier to use the more complex parts of the tools the closer to the task solution we got.

2 Description and methodology

As described by the task description in the compendium [3], a recommended step-by-step approach to this exercise was as follows:

1. Setting up and connecting the STK1000 board.
2. Downloading the skeleton code and then compiling, uploading and debugging this code.
3. Expanding the skeleton code with code to turn on the LEDs.
4. Writing code to read the values of the buttons.
5. Further expanding the program with an interrupt routine.
6. Modifying the interrupt routine to do the work of reading the buttons and changing the LED values accordingly.

This is more or less the approach we went with, with the exception of writing the interrupt routine before the button handling code.

2.1 Hardware setup and toolchain usage

To set up the board, the jumpers were set to their given values and I/O cables were connected to the LED and button pins, all as according to the compendium. The board was connected to the JTAGICE debugging device with the appropriate connector, while the JTAGICE in turn was connected to the computer with a USB cable.

Compiling the skeleton code, and also our own code as the development progressed, was done using the supplied *makefile* with the utility `make`. `make` then took care of calling the required programs – the AS assembler, the LD linker and the microcontroller programmer utility `avr32program`.

2.2 I/O setup and LED control

Our first exercise in the code was to get the LEDs to light up. This required us to learn the workings of the GPIO model of the STK1000 board.

As instructed by the compendium [3], the I/O ports are controlled and accessed by addressing into memory space indicated by the `AVR32_PIOB` and `AVR32_PIOC` constants. However, we quickly realized that these 32-bit addresses would not fit as immediate values in any instruction for a 32-bit RISC platform.

To accomodate these large addresses, they would need to be stored together with the program code in the object files and thus in memory when run, such that they would be accessible at a pre-calculated offset from the program counter. By labelling the data values in the assembly code, the compiler would take care of calculating these offsets.

Thus, after having defined the constants with e.g. `piob:  .int AVR32_PIOB`, they could then be accessed by using the `lddpc` instruction, for instance like this: `lddpc r0, piob`. `lddpc` means to load a value into a register from a memory location given as a displacement to the program counter [1].

Once the base addresses for the I/O control fields were loaded into a register, the actual setup of the LED I/O pins was a matter of storing the correct values into these fields.

2.3 Interrupt handling

Next, we attacked the problem of setting up an interrupt routine. The compendium contained a thorough description of how this is done [3]. The main steps are reiterated here:

- Point the `EVBA` system register to the address of the interrupt routine.
- Set the *autovector* (offset) for interrupts from PIOB to 0, as `EVBA` already points to our desired interrupt routine.
- Read PIOB's Interrupt Status Register (`ISR`) to clear any pending interrupts.
- Enable interrupts in the CPU using the `csrf 14` instruction (clear CPU status flag 14, which disables interrupts).

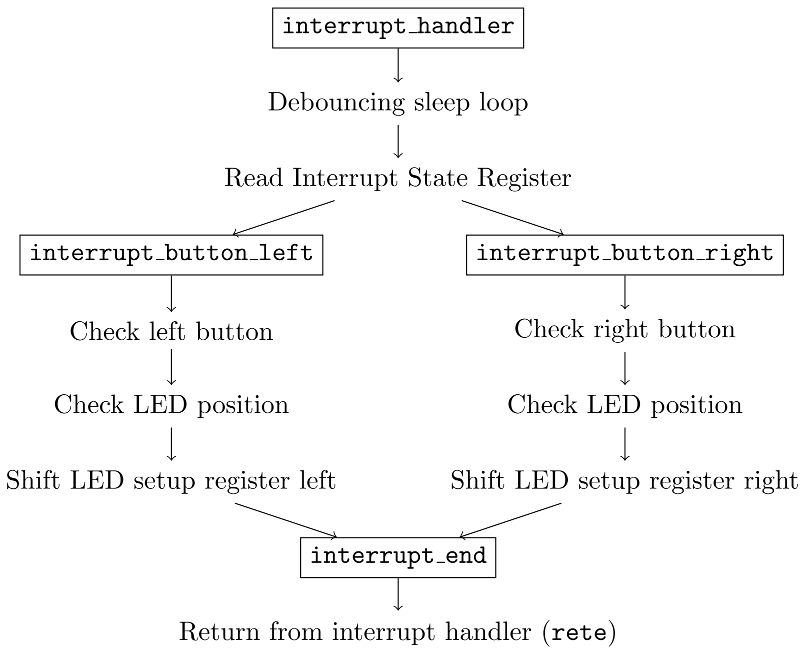

Figure 1: Program flow in interrupt handler

This was in addition to actually writing the interrupt routine. This routine in itself was quite straight-forward to implement. We didn't use any "unsafe" registers, so there was no need to use the stack. The interrupt routine was thus just a regular routine with a `rete` instruction at the end.

2.4 Responding to buttons

After making sure the interrupt routine functioned as intended, we filled it in with code for reading the button pins and responding accordingly. The resulting interrupt routine is sketched out in figure 1.

As detailed in the next chapter, our interrupt routine starts with a wait loop to avoid bouncing effects on the buttons. Afterwards, the routine reads PIOB's (the I/O port connected to the buttons) Interrupt State Register, to determine which button generated the interrupt. The code then proceeds to separate sections for the left and right button accordingly. These sections of the code perform the same kinds of steps, but suited to their corresponding buttons and directions.

First, the data pin for the button is checked to see whether the button is pushed down. This is needed because an interrupt is generated both when a button is pressed and released. If the button is not pushed down, the code skips to `interrupt_end`.

REGISTER	USAGE
r0	Base address for I/O port B (PIOB).
r1	Base address for I/O port C (PIOC).
r2	Base address for interrupt controller (INTC).
r8	Scratch register 1.
r9	Scratch register 2.
r11	Constantly 0xff (to save a few instructions).
r12	The current LED setup (in bits 0-7).

Table 1: Overview of register usage

The code then checks that shifting the LED is possible. That is, if the LED is already at the edge of the row of lights, the code aborts to `interrupt_end`.

After these checks, the interrupt routine is ready to move the light. This is done by shifting the value of a register dedicated for this purpose, `r12`. (A full listing of the registers used is provided in table 1.) Note that the interrupt handler only changes the value of this register, not the actual values on the I/O pins. This is left for the main loop, as we shall see soon.

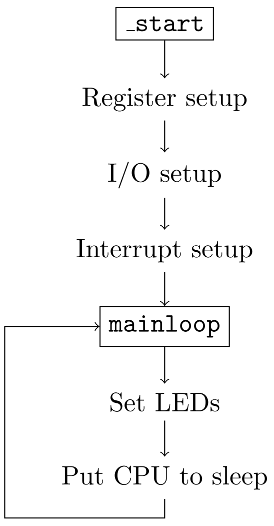

Figure 2: Main program flow

2.5 Main program loop

All that remained after a finished interrupt routine, was a main part of the program that interacted with the interrupt routine properly. As already implemented, the program started with initialization of registers, setup of I/O (LEDs and button) and setup of interrupt handling. In addition to this, the program was supplied with a main loop. Its only task was to update the LED I/O pins accoring to the LED setup register `r12`.

However, we didn't want the CPU to continually do this update in an endless loop. Therefore, we *put the CPU to sleep* at the end of the main loop. This would halt the CPU's execution of the code until otherwise woken up by e.g. an interrupt. This was exactly the behaviour we wanted, as we wanted the mainloop to update the LEDs after an interrupt.

See figure 2 for the program flow in the main part of the program.

3 Results and tests

We did not have a formal testing protocol during this assignment. We completed the code in two stages - one to familiarise ourselves with the function of the LEDs, and one to handle interrupts and button input.

The first stage consisted of a lot of trial-and-error coding in an attempt to make the row of LEDs cycle, and at what speed. Doing this was a good way to familiarise ourselves with the microcontroller and its quirks, and was a great help later in the assignment. Our practice involved turning the correct LEDs on and off, making sure it looped (using `rol` to shift the LED along for each loop, and `ror` to shift it back to the starting point), and making them cycle through all the LEDs at the right speed (using a waiting loop). Testing, here, mainly consisted of changing something, and observing whether the LEDs on the microcontroller behaved correctly. Any errors were rooted out by stepping through the code while observing the LEDs.

The second stage consisted of implementing our interrupts and reading from the buttons. At this point, we took the opportunity to tidy up our code (removing the LED cycle sections) and comment it. We then implemented the interrupts for buttons 0 and 2, testing for which button was pushed and handling the next action accordingly. Again, we had no formal testing structure to test our program, and simply took turns pushing the “left” and “right” buttons to make sure the LED moved correctly.

3.1 Contact bouncing and debouncing

At this point we encountered the problem with *contact bouncing*. The actual movement when pushing the button adds noise to the input, and it would make our program behave incorrectly because it was reading the bounce input as actual data. Pushing the button only once could cause the LED to move more than one step in the relevant direction. To prevent this, we delayed reading the input until it had settled, and implemented debouncing in our interrupt handler (see figure 3). The debouncing mechanism is very simple - it loops 1000 times to force a wait, and by the time the CPU has cycled through, the input has settled to the correct value. At first we tried waiting 100 loops, but still experienced a small degree of bouncing, depending on how we pressed the button. 1000 loops took care of this problem without causing a noticeably delayed response time.

```
        /* The interrupt handler. */
interrupt_handler:
        /* Loop 1000 times to avoid bouncing. */
        mov r8, 1000
interrupt_handler_sleep_start:
        sub r8, 1
        cp.w r8, 0
        breq interrupt_handler_sleep_end
        rjmp interrupt_handler_sleep_start
interrupt_handler_sleep_end:
```

Figure 3: Delay loop for debouncing in the interrupt routine

3.2 Further testing

After this, we tried testing various button-pressing combinations to ensure things worked like expected. Holding a button down for a longer period caused the LED to move only one step in the right direction regardless of holding time, and pressing a button while another was already pressed down produced a result as if the two buttons were pushed in sequence. Pressing two buttons at approximately the same time would cause the LED to move one step in one direction, and one back in quick succession. We also tried pushing a button repeatedly and quickly to see if we could move the LED one step "too far" and thereby turn off all the LEDs. This attempt was unsuccessful. For a larger project we would probably need to write a better test protocol, but our program was simple enough for our unstructured tests to indicate that our program worked as desired.

4 Evaluation of the assignment

This was a suitable challenge for computer science students who have not worked with processors at an assembly level before. We needed to use "trial and error" methods to solve the assignment, and we consider this a good way to learn about the architecture of a microprocessor and the programs used in development of microprocessor software.

It was easy to know where to start thanks to the good assignment description, and we could try out a small program and then develop that small program into a solution to the assignment without having to redesign the whole thing. It was also easy to test whether the assembly code you had programmed worked as intended through the usage of *STK1000 development board* and the *GNU Debugger.*

The assignment provided practical and useful experience that feels relevant considering the study line and future courses. It seems likely that we will need what we learnt during the execution of the assigment sometime in the future.

5 Conclusion

We successfully completed the assignment as per specification, and learnt something very useful about implementing input from analog buttons or switches. Before this, we had not heard about contact bouncing and the problems it could cause.

After this assignment, we have become familiar with some simple usage of common development tools such as `make`, `gdb`, `as` and `ld`. We have successfully used assembly language to program a microcontroller, and used interrupt-based I/O to detect button pushes and control LEDs. Through a carefully laid out program flow, we've managed to eliminate all obvious errors that could arise.

References

[1] Atmel, *AVR32 Architecture Document*, Atmel Corporation, 2011.

[2] Atmel, *AT32AP7000 Reference*, Atmel Corporation, 2009.

[3] NTNU, *Lab Assignments in TDT4258 Microcontroller System Design*, Computer Architecture and Design Group, Department of Computer and Information Science, Norwegian University of Science and Technology, 2011.