Rapport Øving 1
TDT4258 Mikrokontroller og systemdesign

Thomas Rootwelt
Eivind Larsen

February 20, 2012

Abstract

The STK1000 developer board and AVR-32 based microprocessor (both made bu Atmel) will be used for the remaining exercises, and getting to know how to develop for these systems is important. Developing for micro controllers differs from other development in that code needs to be compiled for an other system and that debugging needs to happen remotely. There exists tools to make this easy. Assembly language is normally the language closest to machine code that programmers have to work with. It consist mostly of instructions that the processors can execute directly (once translated). Understanding it is important in understanding hoe processors implement higher functions. In this paper we will describe a program written in assembly for the STK1000 with an AVR-32 based microcontroller. This program initialises a row of 8 led lights as output and a row of 8 buttons as input. It then turns on one of the lights and enables interrupts on two of the buttons. It then goes to sleep. Using an interrupt routine, it moves the led that is currently on to the left or right when the left or right button is pressed. Additionally, upon reaching the end, it will circle around. The interrupt routine implements an anti-bouncing loop.

Contents

1 Introduction

The goal of this exercise was to learn about the STK1000 and the AVR-32 and how develop assembly code for these using GNU tools. To do this we made assembly code to paddle a led light by pressing one of two buttons. The main loop of the program just sleeps, as this is intended to be "everything else the computer does", and therefore the interrupt routine handles the entire job of reading the input and setting the lights. As there are no other programs running however, we have not written any code to save registers and restore these to their original state upon the return of the interrupt. The program works when more than one button is pressed and loops around when the light reaches led 1 or led 8 and is moved to the right and left respectively. It is also possible to debug the program while it is running on the STK1000 using the GNU debugger.

2 Description and methodology

The whole exercise was done in small increments. We started by learning how to control the LEDs. Then we made registering button presses work, followed by altering the LEDs on a button press. Finally we combined everything together with an interrupt routine. We will now look more closely on each step.

2.1 Overall hardware setup

All jumper settings were checked and done as described in the exercise description. We connected the LEDs with an 8-pin flat cable to the GPIO pins 0-7, which are mapped to PIOB pins 0-7. A second 8-pin flat cable connected buttons 0-7 to GPIO pins 16-23, which are mapped to PIOC pins 0-7.

2.2 Controlling the LEDs

With our setup the 8 LEDs can be controlled through PIO port B pins 0 - 7. We started by enabling these pins on the IO-controller. This is done by writing 0b11111111 to the address for PIOB + offset for the enable register (called PIO_PER). Next we enabled output on the pins by writing 0b11111111 to the address for PIOB + offset for the enable output register (called PIO_OER). Which LEDs are actually turned on can be controlled by writing to the register with offset PIO_SODR, and turned off by writing to the register with offset PIO_CODR, together with an 8-bit bitmask corresponding to LED 0-7.

CPU Register	Contains
`r0`	Address for PIO port B `PIOB`.
`r1`	Address for PIO port C `PIOC`.
`r2`	Address for interrupt controller `INTC`.
`r7`	Bitmask for enabled LEDs (lower most 8 bits)

Table 1: CPU registers used

2.3 Buttons

The buttons SW0-7 were in our case connected to GPIO pins 16-23, which in turn are connected to PIO port C pins 0-7. These are enabled by writing to PIO port Cs address plus the offset PIO_PER. Since these are reading changes in value, we also need to enable the pull-up resistors by writing to the corresponding enable register. The status of these buttons can be read by copying from address (reading register) PIO_C + PIO_PDSR, where the lower-most byte corresponds to the status of the 8 buttons.

2.4 Interrupts

The AVR32 supports interrupt driven IO. To enable interrupts on our buttons several things must be taken care of. First we need to tell the IO-controller to enable interrupts on the buttons. This is done by writing to register with address PIOC + offset PIO_IER. Next the interrupt controller needs to get an address (14 bits) in the register corresponding to interrupts from PIO_C on line 0. The code at this address is the interrupt routine code that will be called. This address is called the autovector. The full address for the code called however, is the result of the variable EVBA + the autovector. In this exercise we can set the EVBA to 0 and just pass the address for where our interrupt routine resides, which helps simplify. Lastly we must enable interrupts for the whole system by setting the GM bit, done by the instruction `csrf 16`.

2.5 Main program

The main program is a loop structured as the flowchart of Figure 1. We start off however by loading the addresses of PIOB, PIOC and the IO controller into registers r0, r1 and r2, respectively (not shown in figure). We then continue to set up the different io controllers and cpu as described in the sections above. The autovector is set to the address of the `handle_interrupt` routine. The routine first determines if one of our predetermined buttons caused the interrupt, then figures out which one of these buttons is pressed right now, if any. If the correct button is pressed during the interrupt, the program then does a logical right or left shift of the LED lights, according to the button that was pressed. If the result after the logical shift is too big, or too small, the LEDs are reset to light up on the other end. E.g. if the value of the enabled LEDs is 0 after a logical

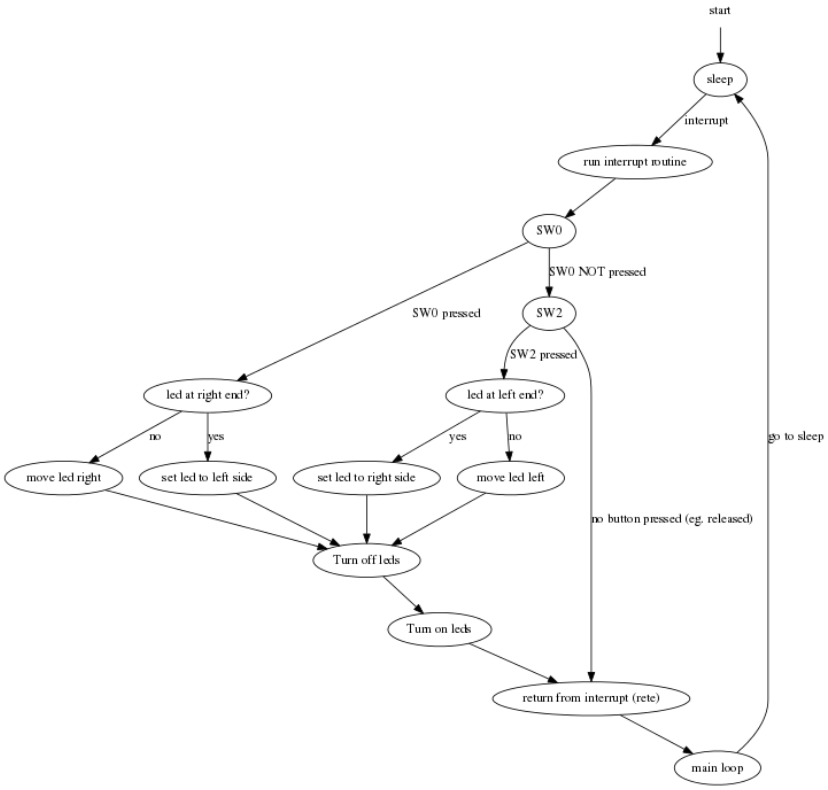

Figure 1: Flowchart showing the flow of the main program loop and the interrupt routine. Notice the button precedence.

right shift, it means that we should reset the light to show up on the other side. This works similarly for the left to right button, where we check if the value is bigger than the 8 lower bits. We then clear all the glowing LEDs and enable the row with the new bitmask, `rete` (return from interrupt) and go to sleep again. An overview of CPU registers in particular use is given in Table 1.

3 Results and Tests

The result was a working program that used interrupts to run a routine that would read if a certain button was pressed and move the led appropriately. It waits in an infinite (sleeping) loop until an interrupt occurs. It then reads the value of the buttons pressed from the I/O device where it is mapped in memory and updates the led lights if either of the two designated buttons are pressed. One button has precedence over the other, and pressing other buttons at the same time will not affect the outcome. A sleep loop is used to counter bouncing on button presses. The program is written in assembly.

The finished program was tested by pressing one or more buttons to see if we

could create unexpected behavior. It was cycled to the left and right so that it looped around to make sure that this worked. Then several buttons was pressed at once to see if this influenced behavior. Everything worked except for when one of the two control buttons is held down, and the other pressed. When this generates an interrupt, the button that has precedence over the other will dominate and the light might move in a way other than intended. This bug can be fixed by saving which button generated the interrupt and checking this when deciding which way to move the light, but we did not have time to implement this.

Earlier in development testing was done by running the program and debugging using the GNU debugger. We would compare register values with what we had on paper to verify that the code was working as intended after each instruction. Most importantly, we would start the program at the beginning and go through it instruction by instruction to see check if we had loaded the intended values into registers. After we had verified that the values we loaded into registers were indeed the ones we had intended, we moved on to testing by pressing buttons.

4 Evaluation of assignment

GNU tools are really helpful in programming and debugging. Assembly is tedious work and unsuited for more complex tasks. We have learned how to program the basics of a STK1000 and AVR32 based microcontroller, and also to always run all tests again when we alter the code to fix a bug.

Our experience with the assignment was very good, and we learned a lot of the nescessary skills to continue on to the next assignment. We learned how to look up instructions and settings in the datasheets, how to use the debugger and basic usage of the hardware. The compendium was also immensely helpful, with a good balance of plain hints, relevant information and tutorials (eg. the GNU debugger).

5 Conclusion

The program works as expected, except for a bug when pressing and holding both buttons, and can be debugged using GNU. It does still waste a few cycles doing useless work, but there is no other work to do, so this isn't a problem. Debugging revealed some flaws in the code, albeit we figured it out a bit late to fix it. This is because some tests were performed before the final alterations to the code were made. These alterations were made to correct bugs, but introduced new bugs that were not tested for. For future projects, we will rerun all tests when altering the code.

References

[1] Atmel, *AVR32 Architecture Document*, 2011. http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf .

[2] Atmel, *AT32AP7000 Datasheet*, 2009. http://www.atmel.com/dyn/resources/prod_documents/doc32003. .

[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. Accessed per It's Learning.