Report
Exercise 1
TDT4258

Group 1
Jan Bremnes
Magnus Kirø

Table of contents

Abstract:

In this exercise we learned to code assembly on the STK1000 with an avr32 processor. We programmed the LEDs, and the buttons to switch the LEDs on and off. We expanded and made the buttons cause interrupts. This is described in detail under “Description and Methodology”.

Introduction:

This report covers Exercise 1 in the course TDT4258 Mikrokontroller Systemdesign. The exercise presented us with the problem of I/O programming in assembly, directly on a microcontroller, without the help of an operating system. The requirements were as follows:

> “*Write a program in assembly that lets a user, by pressing the buttons, move a “paddle” to the left or right on the row of LEDs. The paddle may be represented as a single lighted LED, and by pressing button 0 the light moves to the right, and by pressing button 1, the light moves to the left.*
> *It’s required that you write an interrupt routine to read the button state, and the LEDs shall be updated in the main loop of the program. It’s also required that you use a makefile, and that you debug the program with GDB through JTAGICE ”*
>
> *- translated from the exercise description in Øvingshefte[1]*

If we wanted to, and had the time, we could add some extra functions of our own

The goals of this exercise was to learn about the general architecture of the AVR32 microcontroller and the STK1000 development board. To gain an understanding of object files and the what the linker does. To learn about the use of GNU-tools, such as the GNU Assembler (GNU AS), the GNU Linker (GNU LD), GNU Make and the GNU Debugger (GDB). Learn assembly programming and interrupt handling on the AVR32, and parallell I/O to control buttons and LEDs.

The microcontroller we used, was an AT32AP7000 from Atmel, on the STK1000 development board.

Øvingsheftet, gave us a detailed description of the task at hand, and supplied quite a lot of useful information. The AVR32 Architecture Document[2], and the documentation for the AT32AP7000[3], contained all the information needed about the assembly language and inner workings of the processor.

Description and Methodology

We first started with familiarising ourselves with the development kit and the task at hand. This was mainly done by reading the supplied Øvingsheftet and browsing through the supplied documentation.

STK1000 and AVR32

The AVR32 is a 32bits RISC-processor, designed by Atmel. It has a seven step pipeline and is capable of executing three different instructions in parallel. The specific microcontroller model that we used was an AT32AP7000, mounted on the STK1000 development board. More information can be found in the øvingsheftet[1] and in the AVR32[2] and AT32AP7000[3] data sheets. The I/O-buses on the STK1000 communicates with the microcontroller through an I/O-controller called Parallel I/O (PIO). On the card there is a general I/O-bus that's not connected to anything special, and can be used by any I/O purpose defined by the user, in this case, us. This bus is called General Purpose I/O (GPIO) and it can be connected to different units with cables.

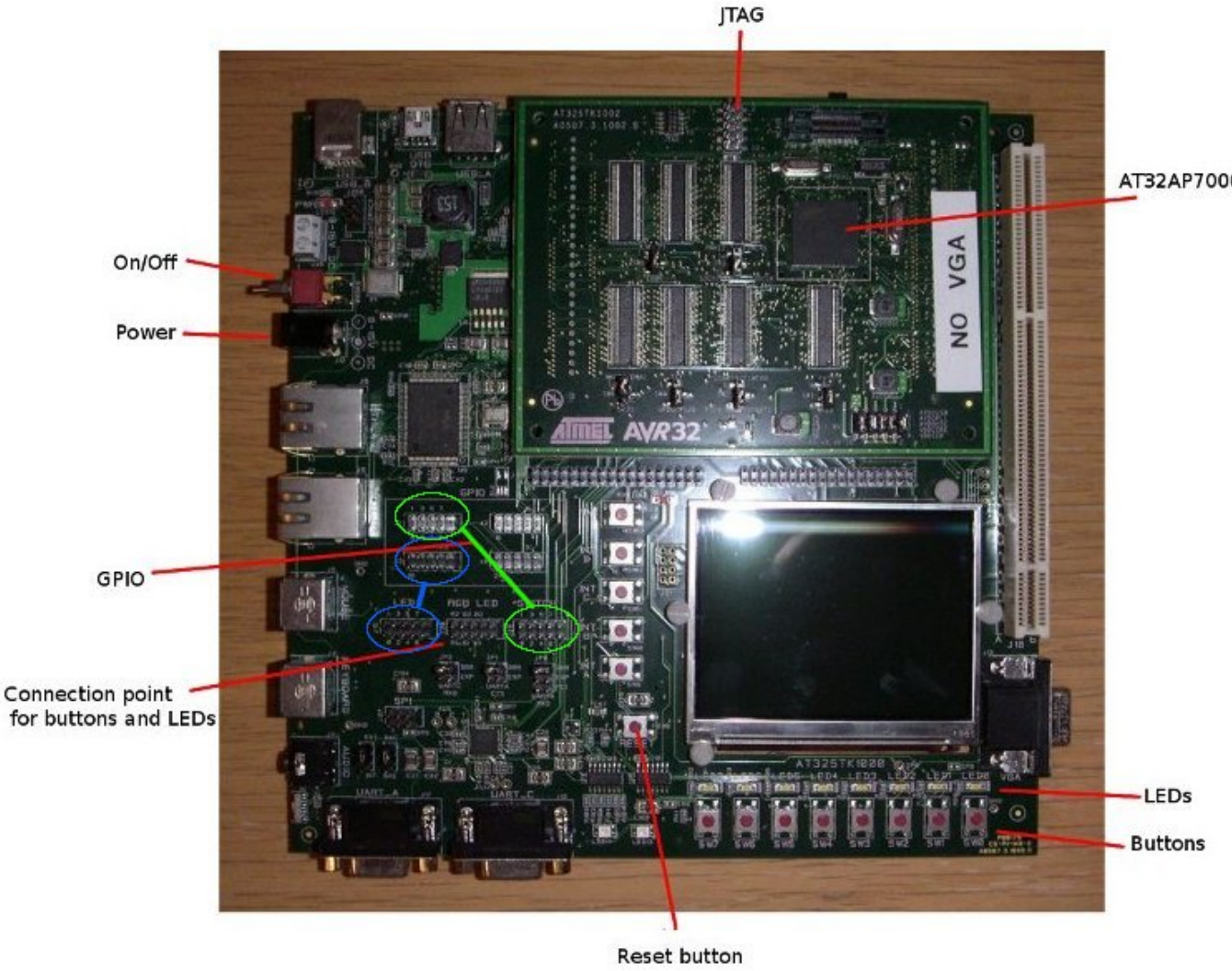

Fig 1. STK1000

Fig 1 above, is a photo of the STK1000, taken from Øvingsheftet. The green and blue circles represent GPIO-port we connected the LEDs and buttons to; green is the buttons, blue is the LEDs. When set up in this way, PIOC controls the LEDs and PIOB controls the buttons.
The buttons will work as input, and the LEDs as output.

Fig 2. JTAGICE

Since we're programming at the most basic level, and without the aid of an operating system, we have to use a special unit called JTAGICE(Fig 2) to be able to program the microcontroller. This unit is connected to the STK1000 in one end, and to the computers USB-port on the other. The JTAGICE allows us to write programs to the flash memory mounted on the STK1000

GNU Development Tools

We did our programming on a computer with Kubuntu GNU/Linux. The exercise text, suggested to use Emacs as our text editor, since GDB can easily be used together with Emacs, but we decided to use Kate, and ran GDB in the terminal (Konsole). None of us had used Emacs before, so we found it easier to use the tools we were familiar with.

To help us get started, we were provided with a file containing a few lines of example code in assembly, a makefile, and a file containing a list with the names and addresses of useful registers for I/O (io.s).
We did not write our own makefile, which was recommended, but not mandatory, to do. The makefile we used were given to us by our student assistant after we encountered some connectivity problems which both he and we were unable to solve. The makefile he gave us,

and which we have included, automates the task of connecting to the JTAGICE, so all we had to do to upload our program to the micro controller, was to enter “make upload” in the terminal, and the connection and upload would be taken care of automatically.

Make is a program that builds projects and programs based on a set of rules you define. The rules are stored in makefiles. In our project this file contains information about how to compile the program, how to pack it and that it should be uploaded to the micro controller and run after compilation. All this is done, and when the make process is done we are running the program in debug mode from our computer.
The make-process creates two files when compiling. a .o file and a .elf file. The .elf file is the file that works with the micro controller and that is the only of the two files we use. In later exercises it will be useful for us to have the .o file. When compiling C code, the high level code from C is converted down to low level code which is stored in the .o file.

We just use the command “make upload”, and the make-file does the rest, compiles the code and uploads it to the micro controller, and starts the connection to the board and starts the debugging. “make upload” points to the upload: tag in the makefile. So when GNU Make runs the makefile, it starts running from “upload:”. Our deviation from the standard makefile is to add the following three lines:
- avr32gdbproxy -f 0,8Mb -a remote:1024 &
- sleep 2
- avr32-gbd oeving1.elf -x connect
These lines connects to the development board, sleeps for 2 cycles and then starts the debugging and testing of the newly uploaded program.
In the GNU debug tool we use the standard command set shown by typing ”help” in addition to break, set and continue as the two most commonly used.

We broke the development of the program into four stages; lighting the LEDs, read buttons, get buttons to move the light and finally adding interrupt handling.

Step 1: Lighting the LEDs

This was our first objective, getting a LED of our choice to light up. Neither of us had much experience with assembly, so these first steps took us quite some time to accomplish. We had been provided with a few lines of assembly code, which we did not find all that helpful at first. There weren’t much progress the first few days, as we had not yet gotten access to the computers in the computer lab, so we didn’t get anything done the first time we met our student assistant. We were unsure how to begin, and couldn’t even manage to write to a PIO-register. So we basically lost the first week, but that was our own fault as we had forgotten to get IDI-accounts.
The next week we got some help from our student assistant and managed to get a LED to light up. This first step consisted merely of deciding on an initial value to give the LEDs, and then

having a main program loop which continnously wrote these values to the proper registers. We ran the “make upload” command to upload the program to the micro controller, and could confirm that the correct LED turned on. Using the GDB, we halted the program, changed the values for the LEDs, and let the program continue. In this way we were able to turn LEDs on and off. We had completed our first objective, and could move on to the next.

To activate the LEDs, we did the following:
1. Enable LEDs
    -set PIOC_PIO_PER register to the correct values which(the LEDs to activate, 0x0ef)
2. Make the LEDs Output.
    -set PIOC_PIO_OER to the same values as point 1.
3. And now we can turn on a LED by setting the SODR register for this LED and turn it of by setting the CODR. Set/Clear Output Data Register

Step 2 & 3: Read button state, and move LED light with the buttons

The program at this point, was of little use. As mentioned, to change which LEDs should be on or off, we had to use the debugger to halt the program, change the value of some registers, and then let the program continue. Getting the buttons to work as input, would let us change the LEDs by the push of a button.

As we had already managed to activate the LEDs and use them as output, it was easy to activate the buttons and use them as input, as these two are done in basically the same way. Setting up the buttons was done quite quickly, but when we tried to use them to control the LEDs we realized this wasn’t going to be as easy as we thought. We were able to move the light to the right, but the light went out if we tried to move it to the left. This was caused by a poor decision when writing the program, a decision made because we were not familiar working with numbers as actual binary numbers. We had to rethink our logic, and instead of using two different values for the enabling and disabling of LEDs, and arithmetic operations such as divide and multiply, we used one value for the active LEDs, do a left or right shift and then use the one’s complement of this value to disable the correct LEDs.

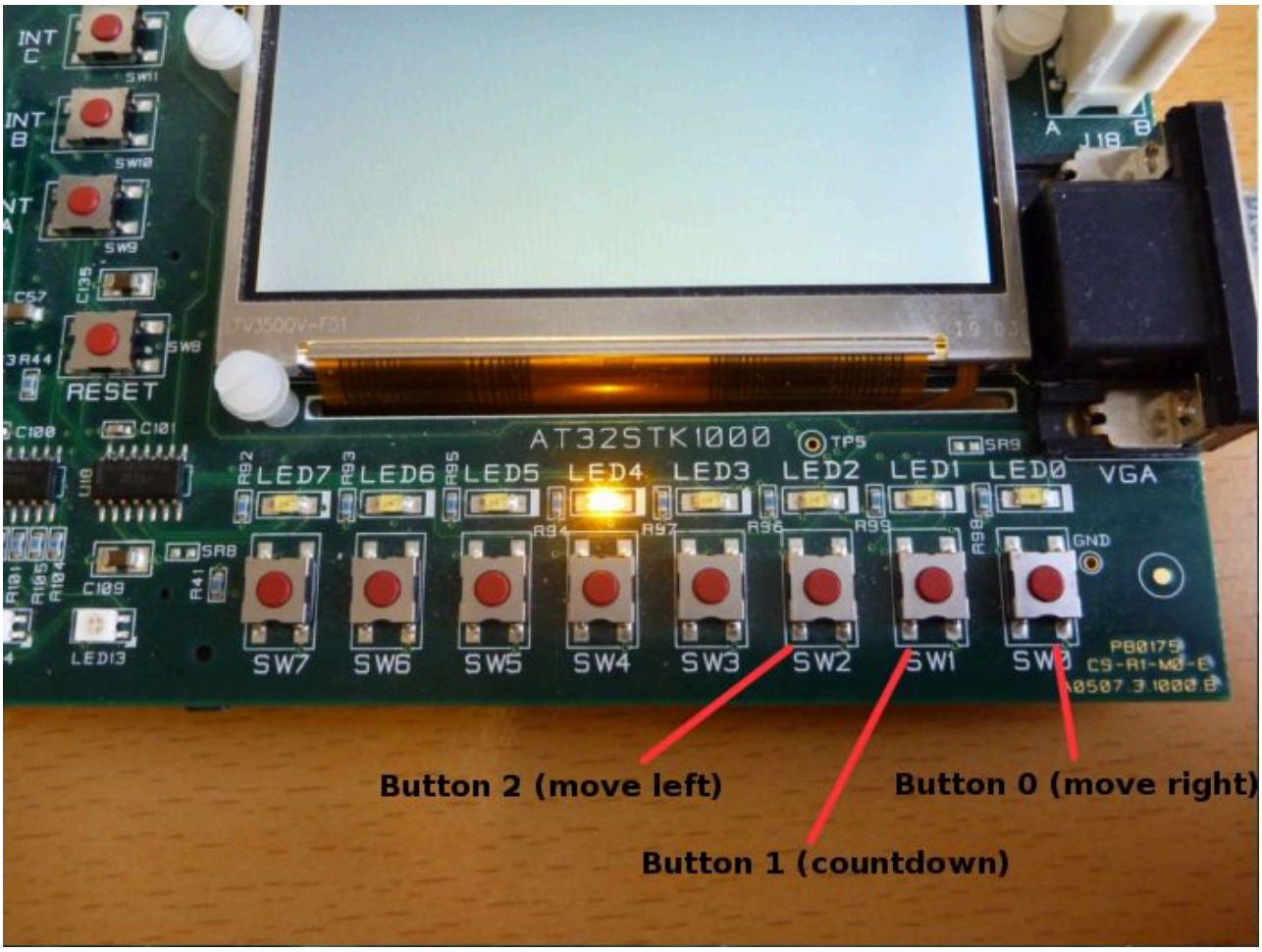

Fig 3. Active buttons and initial state of LEDs

Now we were able to move the light left and right by pressing the buttons, (button 0 = right, button 2 = left) but we did not have full control over them, as a single press on a button could cause the light to jump several steps, or not move at all, instead of moving in a controlled fashion. Something was causing the program to interpret the press of a button not as one input, but several. This is because when a button is pressed, it gains momentum, and will bounce against the *contact* surface several times before settling. Even though it can take only milliseconds for the button to settle, the processor is operating at such great speeds, that it registers several, if not all, of the bounces as a separate button press. To counter this, we had to add a debounce routine, to keep the processor busy while the button settles. The debounce routine starts when a button is pressed, and it consists of decrementing a counter, in this case the value stored in a specific register, looping until that value reaches zero. When that happens, the program reads the button state, resets the counter and branches to the appropriate subroutine. We used the debounce-example in øvingsheftet, so that specific part of the code is not ours (*line 67 - 71 in the source code*). Since the AVR32 is quite a fast micro controller processor, we started with a relative high value for the counter, (0x01FFFFF) and gradually decreased it before settling for 0X0FFFF, which gave the button enough time to stabilize and allowed for rapid consecutive activation of the buttons.

The buttons is set up in much the same way as the LEDs. We activate them, turns on output and input, and then use them.

1. Enable:
    -set PIOB_PIO_PER to the right value to turn on the buttons you want.
    -0x07 is the value for turning on button 1-3(corresponding to SW0-2 on the board)
2. Activating actions.
    -set PIOB_PIO_PUER to the same values as PIO_PER to get the same buttons.
3. Read button status.
    -read the PIOB_PIO_PDSR to see what button is pushed.
4. do something when the button is pushed.

Step 4: Adding interrupt handling

Continuously polling the buttons in the main loop of the program, ties up resources and is a waste of clock cycles. Instead of the processor asking the buttons if they've been pressed, it's better to let the buttons tell the processor themselves. This way, the processor can do other tasks, and if a button is pressed, the processor receives an interrupt, stops what it's doing, takes care of the interrupt, and then resumes whatever it was working on.
Interrupt handling was done by allowing the chosen buttons to generate interrupts, by setting some register values, and giving the interrupt an autovector value. After doing this, when a button is pressed, an interrupt is generated and the processor receives the contents of the EVBA-register together with the autovector supplied by the interrupt. These two values are AND'ed together and the processor then jumps to the address specified by the result of this operation, which is the address given to the interrupt routine.

After the interrupt handling was added, the main loop of the program merely consisted of two operations; deactivating and activating the LEDs. Whenever one of the active buttons is pressed, the program jumps to the interrupt routine, debounces and reads the PIO-PDSR-register to see which button has been pressed, and then jumps to the correct subroutine. Here, the registers storing the values for which LEDs should be active and inactive, are given new values and then the interrupt routine is terminated, returning control to the main program loop and restoring the state of the system. Back in the main loop, the LEDs are updated with the new values.

The following is a step by step description of how to enable interrupts for the buttons

1. Activate the buttons
2. Clear interrupts.
    -set PIOB_PIO_IDR register to disable all interrupts.
2. Enable interrupts on buttons.
    -set the PIOB_PIO_IER register to enable interrupts on wanted buttons.
3. set EVBA

-set the EVBA register to 0x00.

4. Enable interrupts on PIOB.
    -set the IPR14 register to enable interrupts on PIOB.
    -here the value you put in IPR14 is the address where the interrupt will go.
5. Turn on interrupts globally.

Extra function: Binary countdown

We decided to add an extra function besides the mandatory requirements. By pressing button 1 the program will do a countdown, starting at the value of the lighted LED and down to zero. The countdown will be displayed as binary numbers, represented by the LEDs. For instance, if LED4 is on, the countdown will start at 16_{dec}.
Unfortunately, after implementing this function, we discovered that we were unable to interrupt an ongoing countdown. In this case, this doesn't mean anything else than a slight annoyance, but not being able to stop or pause an interrupt, could have more serious consequences in other applications. We did not have enough time to find a solution to this problem, but we decided to keep the added function anyway.

When a user presses button 1, the program jumps to the specified subroutine. Here, the contents of register 1 and register 2 are pushed onto the stack, because these registers will be used during the binary countdown and we need to store them on the stack so that the program can return to it's previous state, with the original values, after the countdown is complete.
After pushing the registers onto the stack, register 1 is given a new value; 0xff. When this value is stored in the PIOC_CODR-register, all the LEDs will be turned off. Register 10 is given a new value, which will cause the countdown to go slow enough for us to see.
The program now moves onto the "Timer"-routine. Just like in the debounce-routine, register 10 is decremented until it reaches zero, at which point, the program will branch to the "Count"-routine. In this routine, the counter (register 10) is reset, and the value in register 1 is stored in PIOC_CODR, which causes all the LEDs to be turned off.
Now register 2 is decremented, and the new value is stored in PIOC_SODR, turning on the appropriate LEDs. If register 2 is zero, the program branches to "countEnd", otherwise it returns to the timer and keeps going until register 2 reaches zero.
When the countdown has finished, the program reaches "countEnd", where the values that were previously stored on the stack, is now popped off the stack and stored back in registers 1 and register 2. The values are popped in the opposite order that they were pushed. After restoring the values in the registers, the rete-instruction is reached, and the program returns to the state it was in before the interrupt. Fig 4 shows the countdown in action.

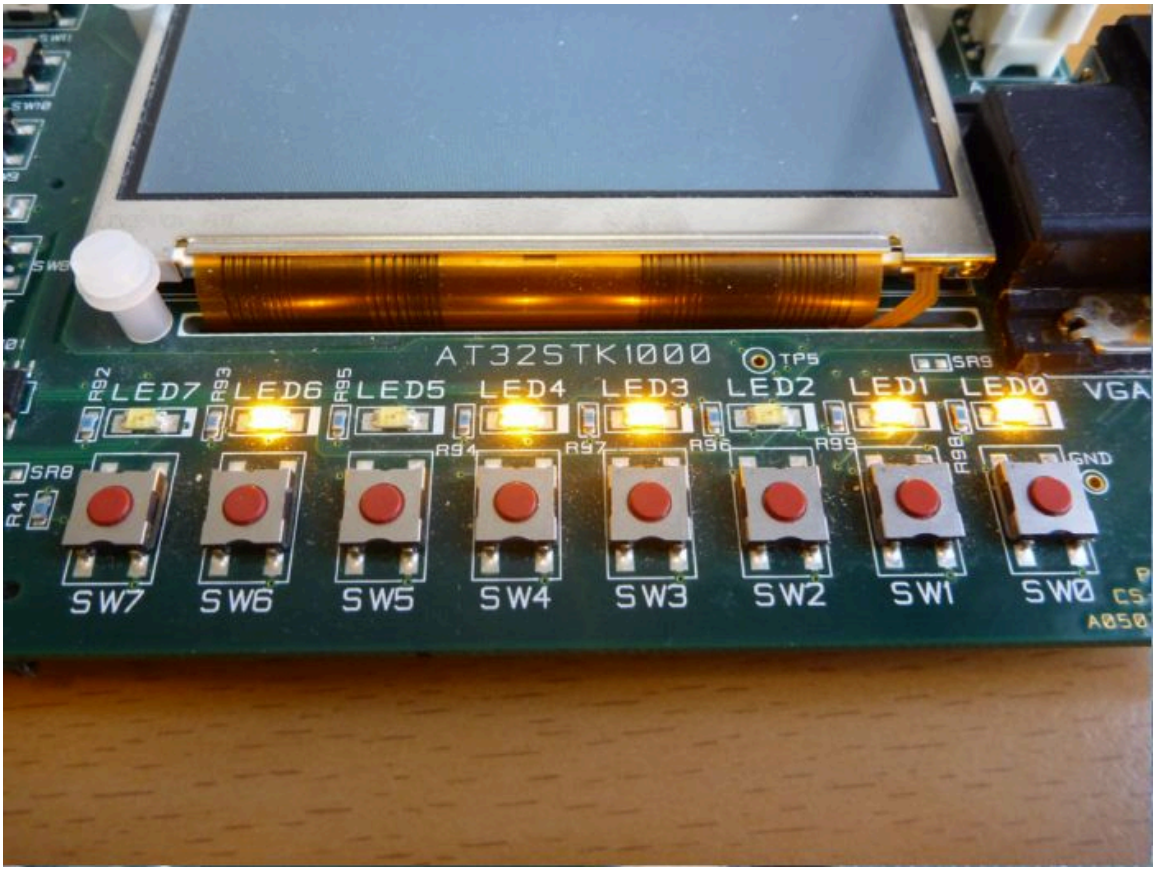

Fig. 4 During countdown

Testing

Testing was done by trial and error. The first function we tested was the lighting of a single LED. In the beginning we gave the LED on/off registers a value in the source code, and added a loop which let us halt the running program and then change the registers value before continuing. Further testing was done in pretty much the same manor. Code - compile - upload - run. and hope it works properly. Then push some buttons to see if it does. And then repeat the sequence.
When the program did not behave as expected, we used the GDB to set breakpoints and check register values at different stages of the program. The values for the debounce and countdown timers were chosen by starting with a high initial value, and gradually decreasing it until the delay was satisfactory

Evaluation of the exercise

The exercise was over all quite good. In the beginning we had some difficulties, when we were

sitting in the lab looking like question marks and wondering why the code did not work. Then we read some more and asked some well placed “what?!” questions to get us going again.
So in retrospect the exercise was nicely balanced. Not too difficult and not too easy. We were given enough information to know what to do, but not enough to know how to do it. Compared to many other courses, where the answer to many problems, is no more than a google search away, this exercise forces us to come up with the answers ourselves. And even though that could cause a lot of frustration sometimes, we really appreciate it now that we’re finished. We feel that we have learned a lot from it, and have had great fun doing it. So much in fact, that we’re really looking forward to the next exercise.

Conclusion

We managed to meet all the requirements given in this exercise, and were able to throw in an extra function of our own, the binary countdown. After adding this, we discovered that we had no way to abort an interrupt and that presented a problem. We would have preferred to implement the needed functionality for interrupting an interrupt, but we didn’t have enough time. We decided to keep the added functionality, as it demonstrates pushing and popping register values to and from the stack, and because we thought it was a fun thing to include.
Throughout the exercise we had some ups and downs in terms of progress. Some drawbacks, and a lot of moments where we found ourselves completely stuck. The problems we had, we can now look back on and easily understand, and some of them could have been avoided had we read the øvingsheftet more thoroughly in the beginning.
To conclude, we feel that we’re starting to get the hang of microcontroller programming and assembly programming

Acknowledgements

Stud.ass, Stian Fredrikstad - for helping us get started, and for providing us with a great makefile which saved us a lot of time while debugging

References

1.*Øvingshefte i TDT4258 Mikrokontroller systemdesign
2. *AVR32 32-bit Microcontroller, AT32AP7000 datasheet
    - Atmel
3. *AVR32 Architecture Document
    - Atmel