#EEDS
Report from Lab Assignment #1
TDT4258 Energy Efficient Computer Systems

Emil Taylor Bye
Sigve Sebastian Farstad
Odd M. Trondrud

February 14, 2013

Part I
Abstract

This report presents a solution to assignment #1 of TDT4258 at NTNU, spring 2013. In the assigment, an Atmel STK1000 development board was programmed to display a moveable LED "paddle" using AVR32 assembly and the GNU tool chain. Interrupts were used rather than busy-waiting to achieve better energy efficiency. The goal of this assignment was to introduce students to programming microcontrollers, as well as introducing students to GNU Makefiles and using GDB as a debugging tool.

The board was successfully programmed so that a single LED is lit at a time, representing a paddle, which responds to the appropriate buttons being pushed.

In the process we learned that the STK1000 development board should not be put in `sleep 5` and that the computers in the lab should not be rebooted or turned off.

In conclusion, our understanding of and experience with the GNU toolchain, assembly programming, Makefiles, technical manuals and interrupt handling in assembly for the AVR32 has been greatly lifted.

Part II
Introduction

The objective of this lab assignment is to make a program for the STK1000 which allows the user to control which of the LED diodes on a row of diodes should light up. This should be accomplished by changing the selected diode to an adjacent diode, either to the right by pressing button 0 or to the left by pressing button 2, effectively letting the move a "paddle" along the row of LEDs.

In order to solve this task, it is essential to have a basic understanding of the STK1000 and microcontrollers in order to correctly set up the hardware for input/output. It is also important to possess basic knowledge of assembly for the AVR32 in order to be able to program, and more specifically how to handle interrupts in assembly in order to achieve high energy efficiency.

Part III
Description and Methodology

This section describes how the paddle program was developed. It covers setup and configuration, tools and program details.

1 Experimental Procedure

The first thing we did was locate the lab, on the fourth floor of the IT-Vest building, room 458. We removed one of the AVR32STK1000 boards from their box and set the jumpers to the appropriate positions ([1], pg 37). We booted up one of the computers and connected the AVR32STK1000 to it. Our first piece of code simply enabled all the LEDs and turned them on. Once we got the LEDs working we enabled the buttons. We followed this up by writing code to turn on just one of the LEDs, designated the "paddle" per the assignment's description ([1], pg 37). Shortly thereafter we gave up on trying to figure out GDB. Code was written to move right when `SW0` is pushed, and left when`SW2` is pushed, employing arithmetic shift to move the paddle bit in the appropriate direction. However, a single push of either button caused the paddle to disappear, due to a combination of us simply letting the paddle's bit overflow when it reached either edge and the *bouncing effect* ([1], pg 22). We altered our code so that the paddle would "loop around" to the opposite side of the row of LEDs when it is "pushed off" either side. When pushing either buttons, all the LEDs would light up briefly in turn at such high a speed that it looked like they were all lit simultaneously, again due to the bouncing effect. Implementing debouncing ([1], Fig. 2.9a) seemed like the next logical step, so the next thing we did was change our program to an interrupt-based solution. Then we implemented debouncing.

We had an issue where the paddle would move over one additional LED when either button was pushed (i.e. from `LEDn` to `LEDn+2` rather than from `LEDn` to `LEDn+1`), which we fixed by, in effect, ignoring every second registered push of either buttons. Finally we measured the current on the board's various pins while an interrupt-based program was running, and again with a busy-waiting program in order to compare the energy efficiency of the two solutions.

2 Configuration of the STK1000

2.1 Jumpers

The STK1000 has 10 jumpers that can be set to configure the board. For this assignment the jumpers were set as specified on page 37 in [1].

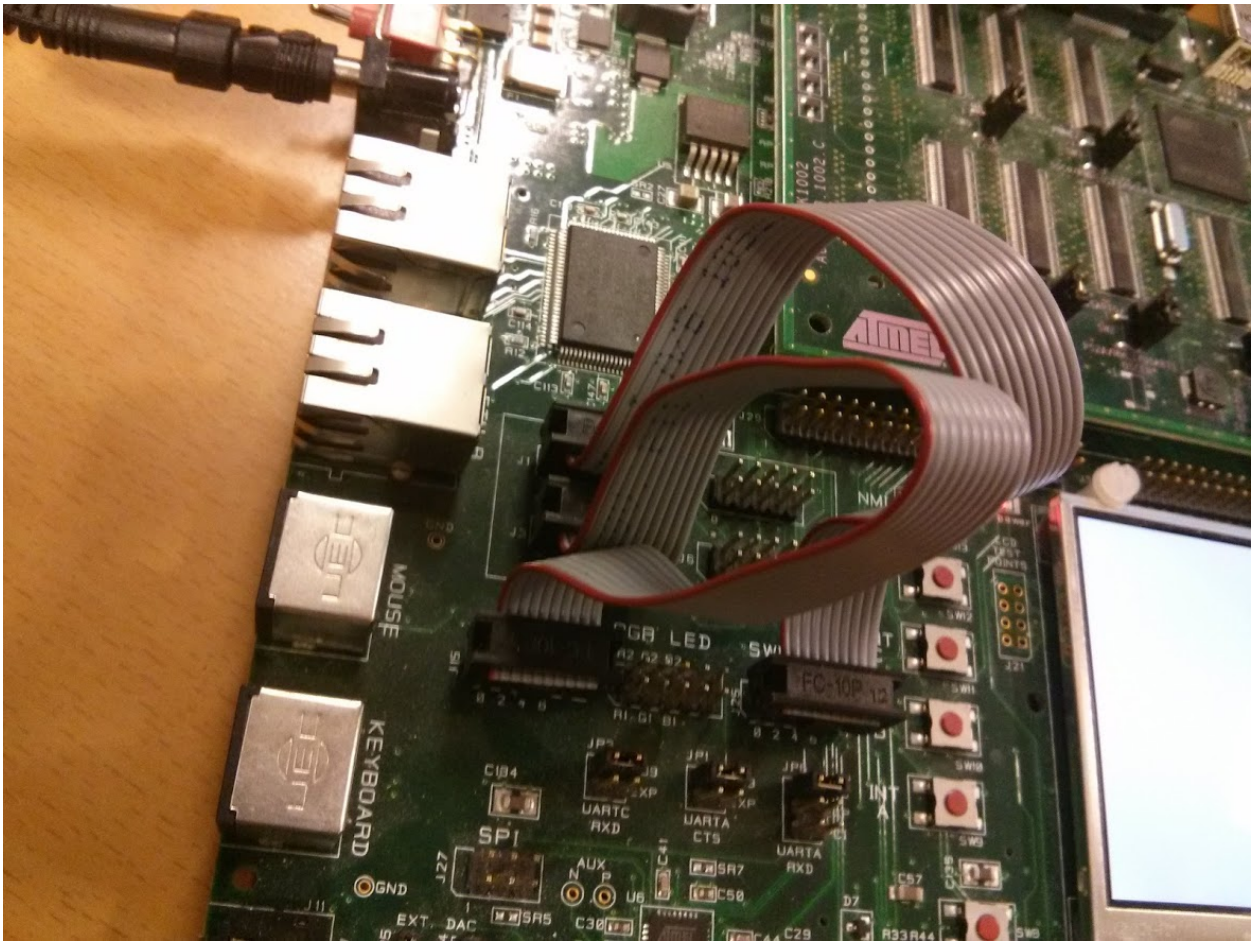

Figure 1: Flat cables connecting GPIO with switches and LEDs. Note the orientation of the flat cables.

2.2 GPIO connections

The STK1000 provides a general purpose input/output interface (`GPIO`) with 32 signal lines. 16 of the 32 available lines were connected to on-board I/O devices on the STK1000 in this assignment. The I/O devices in use were 8 on-board LEDs, used as a rudimentary paddle display, and 8 on-board switches, used as player controls.

The buttons were connected to `GPIO0-GPIO7` (`J1` on the STK1000) using a flat cable as in figure 1. This maps the buttons to ports `0-7` of `PIOB`. The choice of low port numbers `0-7` is convenient for coding, and the choice of `PIOB` as opposed to `PIOC` is purely mnemonic ('B' for buttons).

The LEDs were connected to `GPIO16-GPIO23` (`J3` on the STK1000) using a flat cable as in 1. This maps the LEDs to ports `0-7` of `PIOC`. Having the same port numbers for the buttons and the LEDs is a nice convenience for cleaner and more efficient code, as translation from button ports to LED ports is not necessary.

3 Programming environment

3.1 JTAGICE

The AP7000 sisterboard on the STK1000 provides a JTAG interface which is used for programming and debugging of the board. The development PC was connected to the JTAG interface of the STK1000 using an Atmel JTAGICE mk II (firmware 7.29). The JTAGICE does not require external power as long as it is

connected to the PC over USB.

3.2 GNU Debugger

The instructions presented in the compendium were followed in an attempt to employ the GNU debugger, but the proxy connection could not be established. Because of this, the GNU debugger did not play a important role in the development of the solution. After the main development of the program was complete, the proper setup procedure for the debugger was discovered by another group and subsequently shared. For completeness, the already developed program was debugged using the GNU debugger, to confirm that debugging did indeed work. As the setup procedure differs somewhat from that of the compendium, it is reproduced in its entirety here.

```
$ avr32gdbproxy &
$ avr32-gdb oeving1.elf
(gdb) target remote:4711
```

3.3 Make

GNU Make, a scriptable build tool, was employed in the development of the paddle program. The handout files included a samle Makefile, which was used mostly without modification. After the development of the program was complete, and how to use the debugger became clear (see the previous section), a make command to quickly enter debug mode was included. This command could be invoked by writing `make debug`. Aditionally, and somewhat self-referentially, a Makefile for easy compiling of this LaTeX report was used.

3.4 Other tools

vim was employed as the authors' text editor of choice.

git was used for version control.

The project is hosted in a private github repository.

The report was written with LaTeX.

4 Development of the program

This section details the steps taken during the development of the program. Initially, a bare-bones program was developed with minimal functionality, to get familiar with the environment. Features were added iteratively, starting with simple LED and button integration, and moving on to more sophisticated interrupt-oriented logic/program flow.

4.1 Setting up the LEDs

On the STK1000, the connection to the output LEDs must be set up before the LEDs can be used in a program. First, the I/O pins that the LEDs are connected to must be enabled. In this assignment, the LEDs were connected to the pins `GPIO16-23`, corresponding to `PIO C` lines `0-7`. To enable the correct I/O pins, we must therefore set to 'high' bits `0-7` of the `PIO C PIO Enable Register` (`PIOC PER`), as in listing 1. Here, `r3` is the base offset of `PIOC`, `AVR32_PIO_PER` is the `PIO Enable Register` offset, and `r6` contains the bitfield indicating which pins to enable.

Listing 1 : Enable the I/O pins

```
99      /* enable IO pins for the LEDs */
100     st.w r3[AVR32_PIO_PER], r6
```

Second, the I/O pins must be set to act as output pins, as opposed to input pins. This is done by setting to 'high' the corresponding bits (`0-7`) of the `PIO C Output Enable Register` (`PIOC OER`), as

in listing 22. Here, `r3` is the base offset of `PIOC`, `AVR32_PIO_OER` is the `Output Enable Register` offset, and `r6` contains the bitfield indicating which pins to set as outputs.

Listing 2 : Set the pins to act as output pins

```
102     /* set the IO pins to be outputs */
103     st.w r3[AVR32_PIO_OER], r6
```

Once this is done, LEDs can be turned on by writing the appropriate bits to `PIO C Set Output Data Register` (`PIOC SODR`), as in listing 3. Here, `r3` is the offset of `PIOC`, `AVR32_PIO_SODR` is the `Set Output Data Register` offset, and `r4` contains the bitfield indicating which LEDs to switch on.

Listing 3 : Switch on LEDs

```
176     /* turn on the appropriate LED */
177     st.w r3[AVR32_PIO_SODR], r4
```

Analogously, LEDs can be turned of by writing the approtiate bits to `PIO C Clear Output Data Register` (`PIOC CODR`), as in listing 4. Here, `r3` is the offset of `PIOC`, `AVR32_PIO_SODR` is the `Set Output Data Register` offset, and `r4` contains the bitfield indicating which LEDs to switch off.

Listing 4 : Switch on LEDs

```
173     /* turns all LEDs off*/
174     st.w r3[AVR32_PIO_CODR], r6
```

4.2 Setting up the buttons

The connection to the input buttons must be set up before the buttons can be used in a program. First, the I/O pins that the buttons are connected to must be enabled. In this assignment, the buttons were connected to the pins `GPIO0-7`, corresponding to `PIO B` lines `0-7`. To enable the correct I/O pins, we must therefore set to 'high' bits `0-7` of the `PIO B PIO Enable Register` (`PIOB PER`), as in listing X. Here, `r2` is the base offset of `PIOB`, `AVR32_PIO_PER` is the `PIO Enable Register` offset, and `r6` contains the bitfield indicating which pins to enable.

Listing 5 : Enabling I/0 pins

```
105     /* enable IO pins for the buttons */
106     st.w r2[AVR32_PIO_PER], r6
```

Second, the pull-up resistors for the buttons must be enabled. This is because ¡reason¿. This is done by setting to 'high' the corresponding bits (`0-7`) of the `PIO B Pull-Up Enable Register` (`PIOC PUER`), as in listing x. Here, `r2` is the base offset of `PIOB`, `AVR32_PIO_PUER` is the Pull-Up Enable Register offset, and `r6` contains the bitfield indicating which pull-up resistors should be enabled.

Listing 6 : Enabling pull-up resistors

```
108     /* enable pull-up resistors */
109     st.w r2[AVR32_PIO_PUER], r6
```

Once this is done, the button state can be read by reading the appropriate bits from `PIO B` Pin-Data Status Register (`PIOB PDSR`), as in listing x. Here, `r2` is the offset of `PIOB`, `AVR32_PIO_PDSR` is the Pin-Data Status Register offset.

Listing 7 : Reading the button state

```
225     /* read button status from piob */
226     ld.w r12, r2[AVR32_PIO_PDSR]
```

4.3 Setting up the interrupts

Before interrupts can be utilized we have to configure the board in an appropriate manner. This process is outlined in section 2.5 of [1].

4.3.1 Enabling Interrupts

Since we connected the buttons to PIO port B, we will be detailing how we enabled interrupts from PIO port B. First, we set the appropriate bits in the Interrupt Enable Register to 1.

Listing 8 : The base address of PIO port B is loaded into r2.

```
84      lddpc r2, piob_offset
```

Listing 9 : The masks of Button 0 and Button 2 are loaded into r5

```
97      mov r5, SW_0 | SW_2
```

Listing 10 : Interrupts are enabled for Button 0 and Button 2

```
114     /* turn on button interrupts for SW0 and SW2 */
115     st.w r2[AVR32_PIO_IER], r5
```

Before we enable interrupts for Button 0 and Button 2, we defensively disable interrupts for everything by loading `0xff` into the Interrupt Disable Register. Finally, then enable interrupts globally by setting the bit GM (Global Interrupt Mask) in the processor's status register to 0.

Listing 11 : Enable interrupts globally.

```
126     /* finally, enable interrupts! */
127     csrf SR_GM
```

4.3.2 Loading the interrupt routine

After enabling interrupts, we have to inform the processor about what to do when it receives an interrupt. First, we specify the address of our interrupt routine to be used as the autovector when the interrupt controller receives an interrupt from group 14.

Listing 12 : The base address of the interrupt controller is loaded into r7.

```
86      lddpc r7, intc_base
```

Listing 13 : The address of our interrupt routine is loaded into r8.

```
87      mov r8, button_interrupt_routine
```

Listing 14 : The address of our interrupt routine is stored in IPR14's register in the interrupt controller.

```
120     /* set button_interrupt_routine to handle button interrupts */
121     st.w r7[AVR32_INTC_IPR14], r8
```

Then we have to set the processor's EVBA[1] register to the desired value, i.e. zero.

[1] Exception Vector Base Address

Listing 15 : Setting the EVBA to 0. 0 is stored in r1.

```
117     /* set the EVBA to 0, as specified in the compendium */
118     mtsr 4, r1
```

This is because the interrupt routine address is calculated by performing bitwise logical AND on the EVBA and autovector, where the autovector is represented by the 14 least significant bits in the interrupt routine address. By setting the EVBA to 0, the interrupt routine address simply becomes the autovector, which we have specified as the address of our interrupt routine.

4.4 Interrupt Routine

Our interrupt routine first reads the state of the buttons by calling another routine which stores the buttons' state in `r12`.

Listing 16 : Read button state.

```
190     /* read button state */
191     rcall read_buttons
```

Once the buttons' state has been read, the debouncing routine ([1], Figure 2.9a) is called to prevent bouncing.

Listing 17 : Debounce!

```
193     /* software debounce to stop button glitching */
194     rcall debounce
```

The debouncing routine keeps the processor busy by repeatedly subtracting one from some value[2] until it reaches zero. As we have not yet notified the processor that the interrupt has been handled, this prevents further interrupts from being registered until the debouncing is finished. The last thing our interrupt routine does before returning is notify the processor that the interrupt has been handled, by reading the Interrupt Status Register. The value is stored in r0, which is our designated free-for-all register.

Listing 18 : Reading the Interrupt Status Register

```
196     /* notify that the interrupt has been handled */
197     ld.w r0, r2[AVR32_PIO_ISR]
```

4.5 Refactoring and Modularization

YO!

4.6 Program flow

This section presents a detailed overview of the program flow of the final interrupt-oriented paddle program.

4.7 Register Overview

The STK1000 has 13 general purpose registers named r0-r12. The programmer is free to use these registers for whatever they want. However, several conventions are commonplace to introduce a certain degree of structure.

Conventions:

r0 is a scratch register, used for intermediate calculations and such. r1 holds the constant 0.

...

r8, r9, r10, r11, r12:

[2]This value is specified as the `DEBOUNCE` constant in our code.

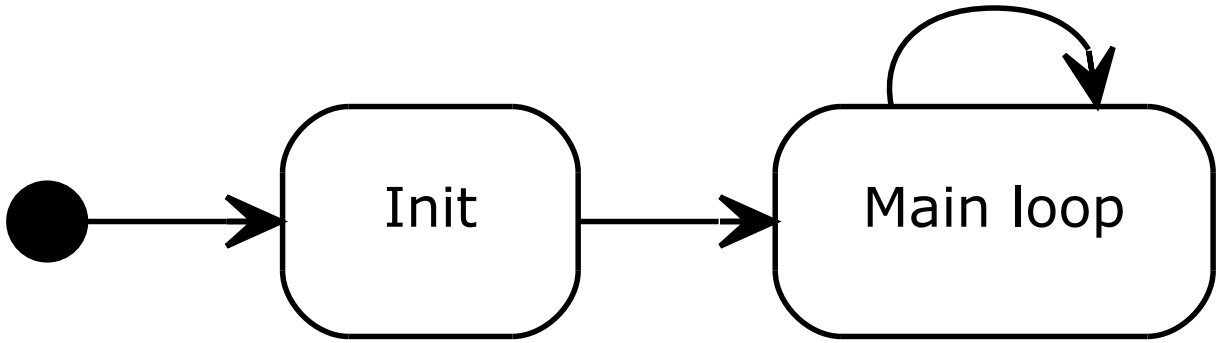

Figure 2: Main superficial program flow

Figure 3: Initialization

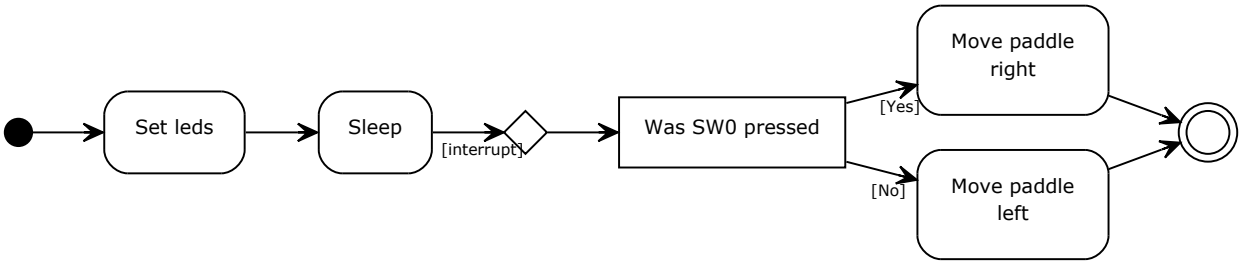

Figure 4: Main loop

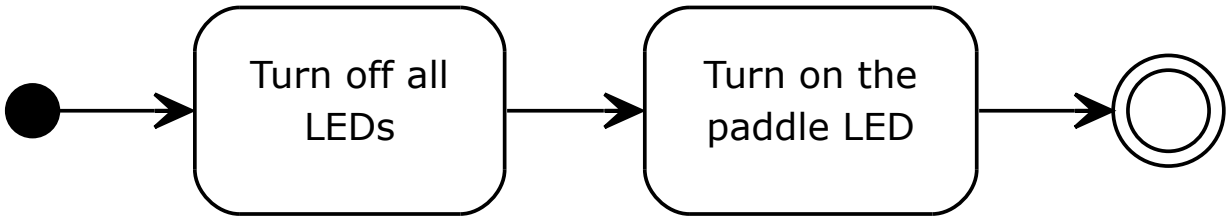

Figure 5: Set leds subroutine

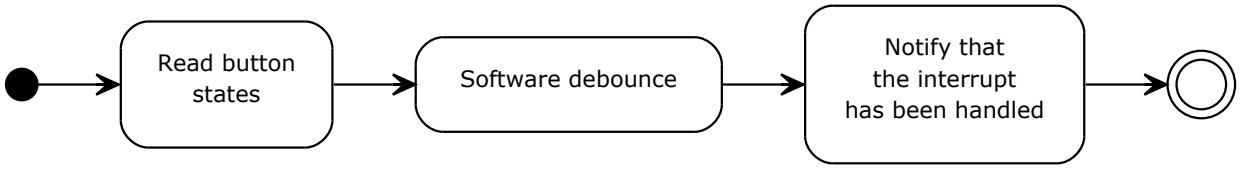

Figure 6: Button interrupt routine

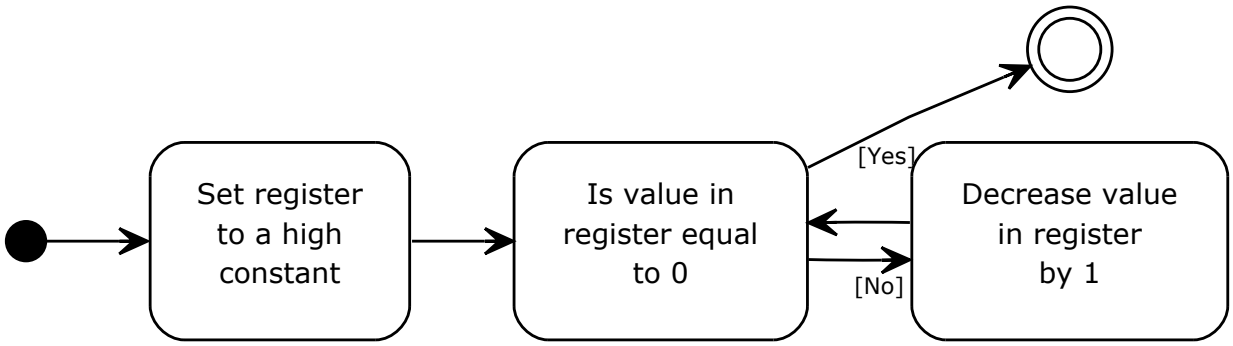

Figure 7: Debounce subroutine

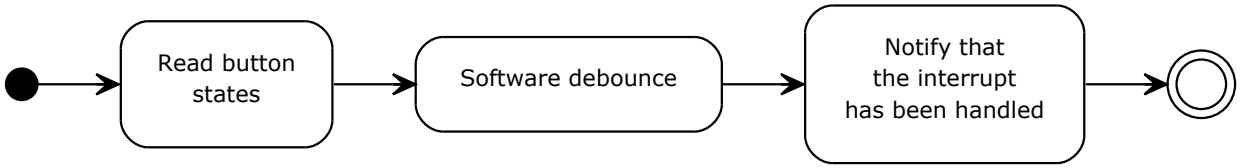

Figure 8: Read buttons subroutine

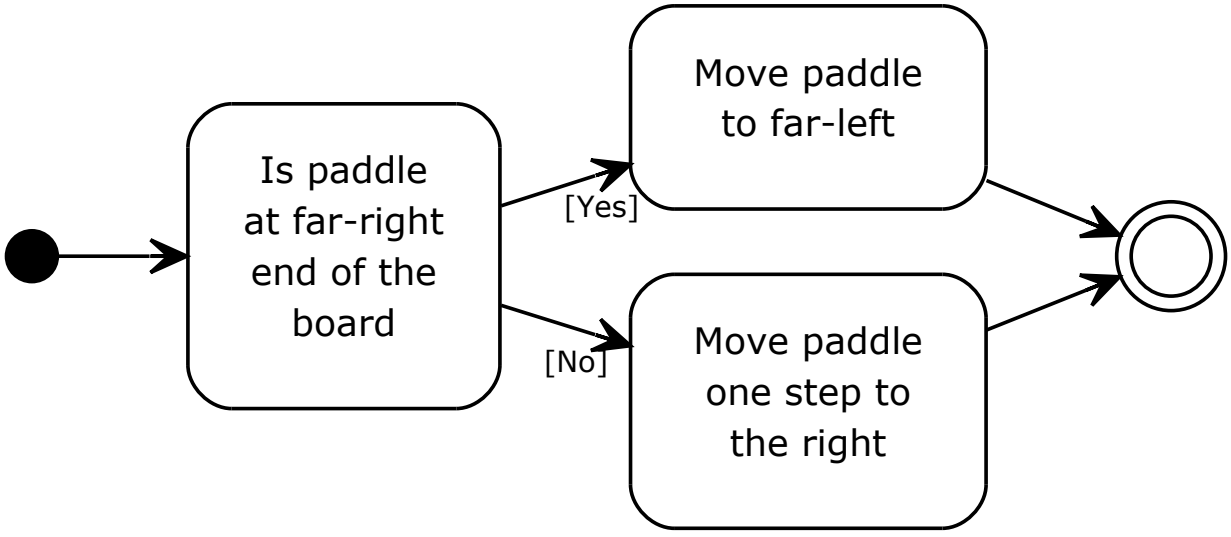

Figure 9: Move paddle right subroutine

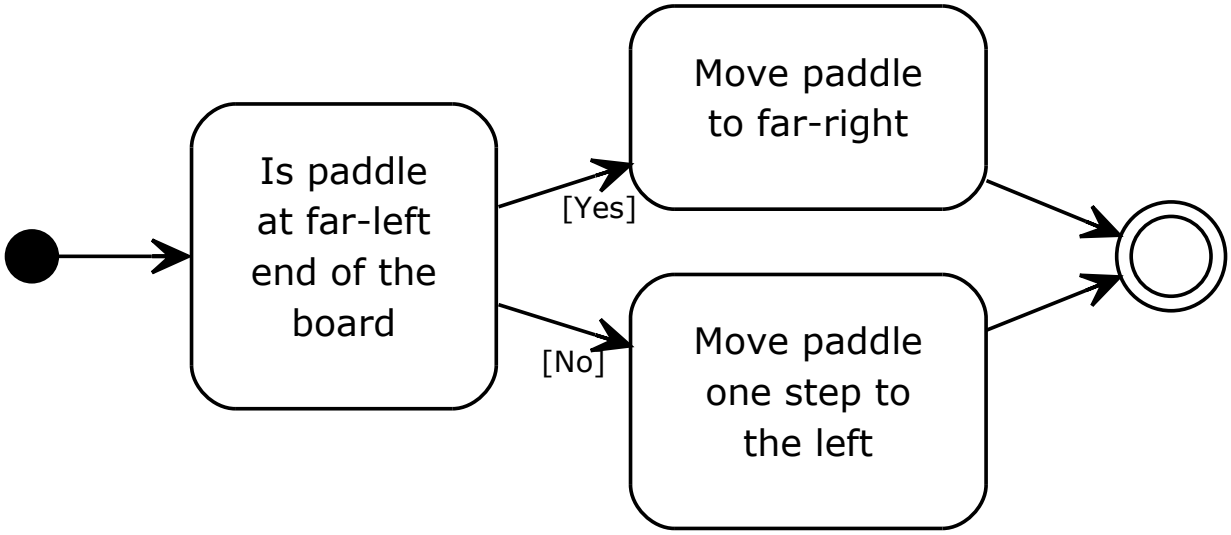

Figure 10: Move paddle left subroutine (analogous to move paddle right subroutine, but included for completeness

r0	Scratch register used to hold intermediate values.
r1	Constant: 0
r2	Constant: base offset to PIOB
r3	Constant: base offset to PIOC
r4	Variable: holds the position of the paddle
r5	Constant: 5
r6	Constant: 0xff
r7	Variable: holds the previous button state
r8	Constant: pointer to button interrupt routine
r9	(not used)
r10	(not used)
r11	(not used)
r12	Holds return value from routines

used to hold parameters when calling a routine.
r12:
used to hold the return value when returning from a routine.
For a list of registers and what they are used for in the paddle move program see **??**.

Part IV
Results and Tests

This is the results and tests section.

4.8 Energy Efficiency

The STK1000AP7000 offers 5 power pin pairs that may be used to measure current consumption of the STK1002 sister board, and therefore also energy efficiency. These 5 pins are situated on the daughterboard,

	Interrupt-based		Busy-wait-based	
Pins	idle	active	idle	active
AVDDUSB (1,2)	1.77mA		1.77mA	
AVDDPLL (3,4)	1.97mA		1.97mA	
AVDDOSC (5,6)	0.00mA		0.00mA	
VDDCORE (7,8)	30.6mA	30.5mA	26.0mA	25.8mA
VDDIO (9,10)	4.90mA	4.99mA	8.05mA	8.13mA

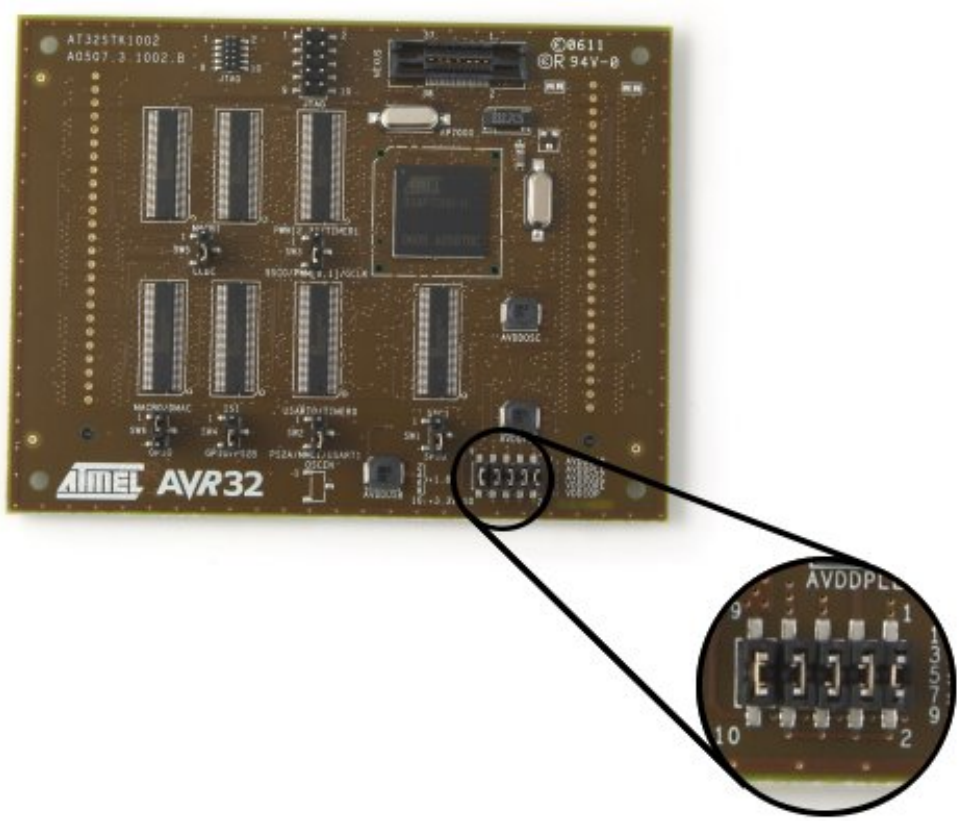

Figure 11: Location of the power pins on the STK1002. Image courtesy of ATMEL.

see figure 11. Current flow through the pins was measured by connecting an ampere meter in series. Current flow was measured on all five pin pairs on two different versions of the program: the final optimized interrupt-based program, and an earlier busy-wait based program without debouncing. For both programs current flow was measured twice: once with the program in an idle state, and once where the paddle was moved about by a user frantically pressing `SW0` and `SW2` as fast as possible. The ampere meter used was a BST BS1901W, with a reported accuracy of +-(1.2% +3d) CITE http://www.docstoc.com/docs/67828383/BS1901W .
The measurements can be found in table **??**.

VDDCORE and VDDIO are the interesting measurements, as AVDDUSB (usb power supply), AVDDPLL (PLL power supply) and AVDDOSC (Oscillator power supply) are not used in the assignment.

Using the sum of the power consumption over each of the 5 measuring points during the idle state as an indicator of the total power consumption of the AP7000, we can evaluate the relative energy efficiency performance of the two programs' use of the AP7000.

The power consumption of the interrupt-based version is $1.77mA * 1.8V + 1.97mA * 1.8V + 0.00mA * 1.8V + 30.6mA * 1.8V + 4.90mA * 3.3V = 78mW$.

The power consumption of the busy-wait version is $1.77mA * 1.8V + 1.97mA * 1.8V + 0.00mA * 1.8V + 25.8mA * 1.8V + 8.05mA * 3.3V = 79.7mW$.

The difference in power consumption is around 1%, which is negligable.

4.9 Testing

All tests assume that you are in possession of at least one (1) functional STK1000 development board (with cables), a JTAGICE mkII (with USB cable) and a computer with software and hardware capable of interfacing with the JTAGICE.

4.9.1 Button Functionality Test

This test aims to uncover if pushing Button 0 has the desired effect (the desired effect from pushing Button 0 is that the paddle moves one LED to the right.)
Prerequisites:

- One (1) finger
- Functional eyesight

Procedure:

1. Upload the code to the board (e.g using `make upload`).
2. Push the board's reset button.
3. Note the paddle's position.
4. Push Button 0.
5. Note the paddle's position.

If the paddle's position in the last step is not one to the right of its position in step #3, Button 0 does not have the appropriate functionality. The test can be refactored to test Button 2: simply push Button 2 instead in step #4 and note that the test will have failed if the paddle's position in step #5 is not one to the left of its position in step #3.

4.9.2 Measurement of Power Consumption

This test measures the board's power consumption by removing a jumper and connecting an amperemeter in series. Power consumption is traditionally preferably measured while the board is turned on and
Prerequisites:

- Multimeter or amperemeter
- Hands, or other tool with similar prehensile ability
- STK1000 development board with the desired program code running.

Procedure:

1. Remove the jumper from the pins that are to be measured.
2. Touch the pins that are now free from the jumper with the measuring device's probes.
3. Press the `RESET` button on the STK1000.
4. Read the measurement from the display of the measurement device.

The board can optionally be interacted with while performing the last step of the procedure in order to measure power consumption during certain interactions. When the test has been performed it is recommended to replace the jumper.

4.10 Discussion

maybe we could turn down the clock speeds, see pagenumer 933 of http://www.atmel.com/Images/doc32003.pdf

Part V
Evaluation of Assignment

The assignment itself was both challenging and interesting. It provides an environment in which wits and being able to tough it out come in handy. However, information regarding certain aspects of the assignment could have been stressed more. For example, during the intro lecture to the assignment [6] it was mentioned that we should be careful with the board's sleep modes. Having no prior experience with the STK1000, the first sleep mode we tested was sleep 5.
Getting a chance to "play around" with the lab equipment was very enjoyable, however it wouldn't hurt if the lab computers were a bit more robust (i.e. they should probably survive a reboot). Also the lab could have better air conditioning.
9/10 would recommend to younger students.

Part VI
Conclusion

This is the place for the conclusion.

References

[1] Lab assignments in tdt4258 energy efficient computer systems. http://www.idi.ntnu.no/emner/tdt4258/_media/kompendium.pdf.

[2] Lab reports. http://www.engineering.utoronto.ca/Directory/students/ecp/handbook/documents/lab.htm.

[3] Power jumpers on ap7000. http://support.atmel.no/knowledgebase/avr32studiohelp/com.atmel.avr32.tool.stk1000/html/jumper_settings.html#Power_jumpers.

[4] Atmel. Avr assembler user guide. http://www.atmel.com/Images/doc1022.pdf.

[5] Atmel. Avr32 architecture document. http://www.atmel.com/images/doc32000.pdf.

[6] Stefano Nichele. Introduction, tdt4258 energy efficient computer systems. http://www.idi.ntnu.no/emner/tdt4258/_media/intro.pdf.

[7] Stefano Nichele. Tutorial lecture for exercise 1. http://www.idi.ntnu.no/emner/tdt4258/_media/ex1.pdf.