Report from Lab Assignment #1
TDT4258 Energy Efficient Computer Systems

Emil Taylor Bye
Sigve Sebastian Farstad
Odd M. Trondrud

February 15, 2013

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 toolchain. Interrupts were used rather than busy-waiting to achieve better energy efficiency. The goal of this assignment was to introduce students to programming microcontrollers in an energy-efficient fashion, as well as introducing students to the GNU toolchain, and specifically GNU Makefiles and using GDB as a debugging tool.

Contents

I Introduction 1

II Description and Methodology 1

1 Experimental Procedure 1

2 Configuration of the STK1000 1
2.1 Jumpers . . . 1
2.2 GPIO connections . . . 2

3 Programming environment 2
3.1 JTAGICE . . . 2
3.2 GNU Debugger . . . 3
3.3 Make . . . 3
3.4 Other tools . . . 3

4 Development of the program 3
4.1 Setting up the LEDs . . . 4
4.2 Setting up the buttons . . . 5
4.3 Setting up the interrupts . . . 5
4.3.1 Enabling Interrupts . . . 5
4.3.2 Loading the interrupt routine . . . 6
4.4 Interrupt Routine . . . 6
4.5 Program flow . . . 8
4.6 Register Overview . . . 11

III Results and Tests 11

5 Energy Efficiency 11

6 Testing 13
6.1 Button Functionality Test . . . 13
6.2 Measurement of Power Consumption . . . 13

7 Discussion 14

IV Evaluation of Assignment 14

V Conclusion 14

Part I
Introduction

This report presents a solution to assignment #1 of TDT4258 at NTNU, spring 2013. The objective of this lab assignment is to write a program for the STK1000 development board which allows the user to move an LED ”paddle” across a rudimentary LED array playing field. The paddle is represented by a single lit LED. It should be possible to move the paddle to adjacent places in the LED array by pressing button 0 to move right, or by pressing 2 to move left.

This report details the development of the paddle move program as a solution to the assignment, and presents a discussion around the energy efficiency of the solution.

The STK1000 is a development board from Atmel which offers a complete development environment for Atmel's AT32AP7000 processor. It offers a multitude of different peripheral I/O devices, of which this assignment will be using an array of LEDs and some push buttons. The processor is an ARM32 processor, and will for this assignment be only running the assembled output of hand-coded AVR32 assembly code, without an operating system.

In the spirit of the course's name, *Energy Efficient Computer Systems*, considerable effort is put into making the program as energy efficient as possible.

Part II
Description and Methodology

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

1 Experimental Procedure

The described procedures were carried out in the PC lab on the fourth floor of the IT-Vest building at NTNU, room 458. The lab is stocked with development PCs and boxed AVR32STK1000 development boards with all necessary extra equipment, such as cables and a JTAGICE. An AVR32STK1000 board was removed from its box and the jumpers were set to the appropriate positions ([9], pg 37). A computer was booted up and the unboxed AVR32STK1000 was connected to it using a USB JTAGICE. The assembly program was then developed on the development PC. 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. The development of the program is elaborated upon in section 4: Development of the program.

Although the GNU debugger would have been a useful tool during development, it was not used, as the GDB setup instructions in the compendium [9] were erroneous, and efforts to find a work-around did not prove to be successful.

During this development, the code was manually tested on the STK1000 by uploading the program using the JTAGICE.

Finally the current throughput over the board's various pins was measured 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 [9].

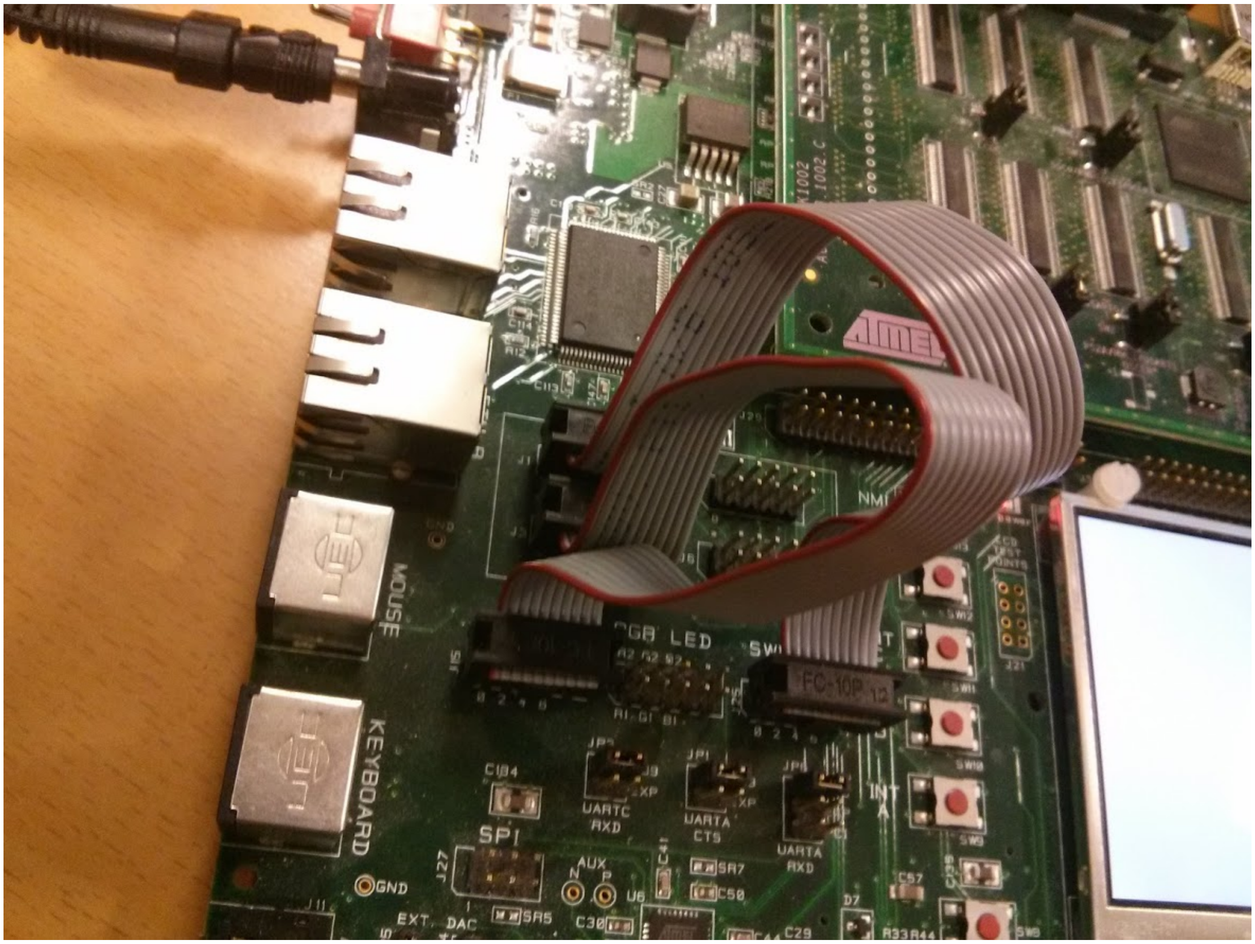

Figure 1: Flat cables connecting `GPIO` with switches and LEDs. Note the orientation of the flat cables.

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 figure 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 STK1002 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 [9] were followed in an attempt to employ the GNU debugger, but a proxy connection could not be established during the debugger setup, rendering the debugger unusable. Because of this, the GNU debugger did not play a important role in the development of the program. 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 in listing 1, in the form of a Make-command from the program's Makefile.

Listing 1 : Makefile code for automatic debugging of the STK1000

```
debug :
	avr32gdbproxy &
	sleep 3
	(echo target remote:4711;cat) | avr32-gdb oeving1.elf
	killall avr32gdbproxy
```

3.3 Make

GNU Make, a scriptable build tool, was employed in the development of the paddle program. The handout files included a sample 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`. Additionally, and somewhat self-referentially, a Makefile for easy compiling of this LaTeXreport 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.
- AVR32-specific flavors of GNU's `as` and `ld` to assemble and link executables.
- `avr32program` was used to program the STK1000 with the JTAGICE.

4 Development of the program

This section details the steps taken during the development of the program. Features were added iteratively, starting with simple LED and button integration, and moving on to more sophisticated interrupt-oriented logic/program flow. The first piece of code written simply enabled all the LEDs and turned them on. Once the LEDs were working, buttons were enabled. The next step was to write code to turn on just one of the LEDs, designated the "paddle" per the assignment's description ([9], pg 37).

Code was written to move the paddle right when `SW0` was pushed, and left when `SW2` was 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. This was due to a combination of the paddle's bit overflowing when it reached either edge, making the paddle disappear, and the fact that a single button press was registered many, many times. The latter happened because a button press was registered for each iteration of the program's main loop while a button was held down. The code was altered so that the paddle would loop around to the opposite side of the row of LEDs when it was pushed off either side.

At this point, 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 dimly lit simultaneously. This is again because of the massive amount of times the main loop would iterate whilst a button was pressed, causing the paddle to cycle the LED row at break-neck speeds. Implementing some sort of button cool-down to reduce the frequency of registered button presses would be the next step if we wanted to continue working with the polling approach to button handling, but we opted rather to migrate to an interrupt-based approach. This choice had the benefit of making the program more energy efficient, as well as it being a hard requirement of the assignment. Implementing interrupts for the buttons introduced hardware bouncing problems, which were solved by implementing software debouncing ([9], Fig. 2.9a).

An issue arised where the paddle would move over one additional LED when a button was pushed (i.e. from `LEDn` to `LEDn+2` rather than from `LEDn` to `LEDn+1`). This was caused by the fact that lifting a button from a pressed state to an unpressed state generated an unwanted interrupt. This was fixed by ignoring every second registered push of each button.

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 bits `0-7` of the `PIO C` PIO Enable Register (`PIOC PER`) to 1, as in listing 2. Here, `r3` is the base address of `PIOC`, `AVR32_PIO_PER` is the PIO Enable Register offset, and `r6` contains the bit field indicating which pins to enable.

Listing 2 : Enable the I/O pins

```
/* enable IO pins for the LEDs */
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 the corresponding bits (`0-7`) of the `PIO C` Output Enable Register (`PIOC OER`) to 1, as in listing 3. Here, `r3` is the base address of `PIOC`, `AVR32_PIO_OER` is the Output Enable Register offset, and `r6` contains the bit field indicating which pins to set as outputs.

Listing 3 : Set the pins to act as output pins

```
/* set the IO pins to be outputs */
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 4. Here, `r3` is the address of `PIOC`, `AVR32_PIO_SODR` is the Set Output Data Register offset, and `r4` contains the bit field indicating which LEDs to switch on.

Listing 4 : Switch on LEDs

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

Analogously, LEDs can be turned off by writing the appropriate bits to `PIO C` Clear Output Data Register (`PIOC CODR`), as in listing 5. Here, `r3` is the address of `PIOC`, `AVR32_PIO_SODR` is the Set Output Data Register offset, and `r4` contains the bit field indicating which LEDs to switch off.

Listing 5 : Switch off LEDs

```
/* turns all LEDs off*/
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, bits `0-7` of the `PIO B` PIO Enable Register (`PIOB PER`) must therefore be set to 1, as in listing 6. Here, `r2` is the base address of `PIOB`, `AVR32_PIO_PER` is the PIO Enable Register offset, and `r6` contains the bit field indicating which pins to enable.

Listing 6 : Enabling I/0 pins

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

Second, the pull-up resistors for the buttons must be enabled. This is done by setting high' the corresponding bits (`0-7`) of the `PIO B` Pull-Up Enable Register (`PIOC PUER`) to 1, as in listing 7. Here, `r2` is the base address of `PIOB`, `AVR32_PIO_PUER` is the Pull-Up Enable Register offset, and `r6` contains the bit field indicating which pull-up resistors should be enabled.

Listing 7 : Enabling pull-up resistors

```
/* enable pull-up resistors */
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 8. Here, `r2` is the address of `PIOB` and `AVR32_PIO_PDSR` is the Pin-Data Status Register offset. The button state is stored in `r12`.

Listing 8 : Reading the button state

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

4.3 Setting up the interrupts

Before interrupts can be utilized the board has to be configured in an appropriate manner. This process is outlined in section 2.5 of [9], and further detailed here.

4.3.1 Enabling Interrupts

Since the buttons were connected to PIO port B in this solution, how to enable interrupts from PIO port B will be detailed in this report. First, the appropriate bits must be set to 1 in the Interrupt Enable Register, see listing 9, 10 and 11.

Listing 9 : The base address of PIO port B is loaded into r2.

```
lddpc r2, piob_offset
```

Listing 10 : The masks of Button 0 and Button 2 are loaded into r5

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

Listing 11 : Interrupts are enabled for Button 0 and Button 2

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

Before interrupts are enabled for Button 0 and Button 2, interrupts are defensively disabled for everything by loading `0xff` into the Interrupt Disable Register. Finally, interrupts are enabled globally by setting the GM (Global Interrupt Mask) bit in the processor's status register to 0 (see listing 12).

Listing 12 : Enable interrupts globally.

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

4.3.2 Loading the interrupt routine

After enabling interrupts, the processor must be informed about what to do when it receives an interrupt. First, the address of the interrupt routine to be used as the autovector when the interrupt controller receives an interrupt from group 14 is specified. The code for this is displayed in listing 13, 14 and 15.

Listing 13 : The base address of the interrupt controller is loaded into r7.

```
lddpc r7, intc_base
```

Listing 14 : The address of our interrupt routine is loaded into r8.

```
mov r8, button_interrupt_routine
```

Listing 15 : The address of our interrupt routine is stored in IPR14's register in the interrupt controller.

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

Then processor's EVBA[1] register must be set to the desired value, i.e. zero (see listing 16).

Listing 16 : Setting the EVBA to 0. 0 is stored in r1.

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

This is because the interrupt routine address is calculated by adding together 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

The button interrupt routine first reads the state of the buttons by calling another routine which stores the buttons' states in `r12`, as detailed in listing 16.

Listing 17 : Read button state.

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

Once the buttons' state has been read, the debouncing routine ([9], Figure 2.9a) is called to prevent bouncing (see listing 18).

Listing 18 : Debounce!

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

[1]Exception Vector Base Address

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 the interrupt routine does before returning is notify the processor that the interrupt has been handled, by reading the Interrupt Status Register (see listing 19). The value is stored in `r0`, which is the designated free-for-all register.

Listing 19 : Reading the Interrupt Status Register

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

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

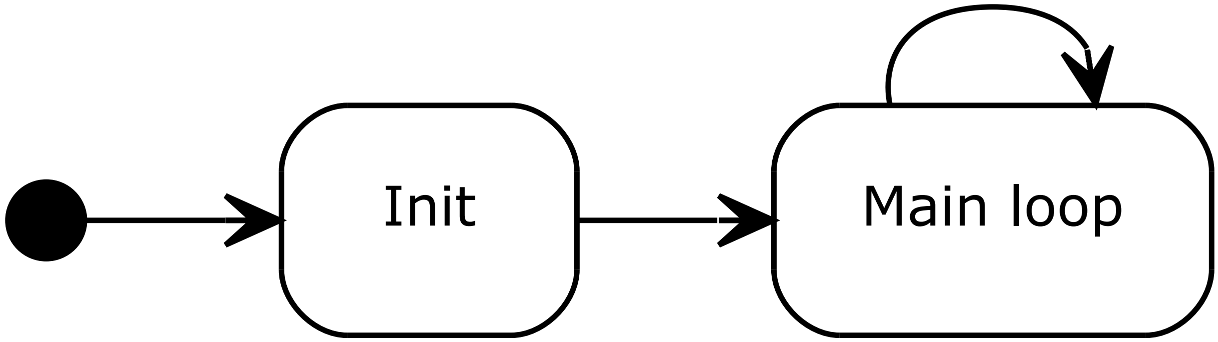

Figure 2: Main superficial program flow

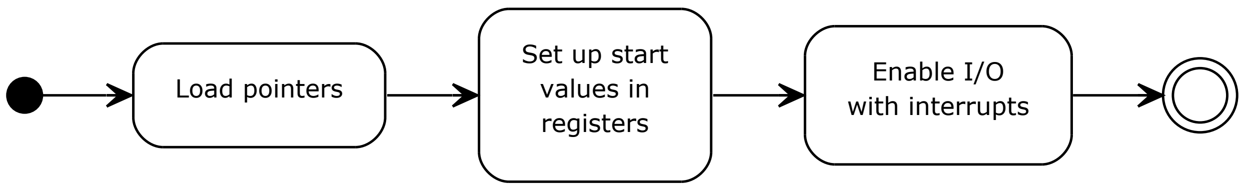

Figure 3: Initialization

4.5 Program flow

This section presents a detailed overview of the program flow of the final interrupt-oriented paddle program, relying heavily on UML flow diagrams to convey the program structure.

The entire program follows the classic init-loop structure, as pictured in figure 2. Note that there is no deinit code, as the program is intended to run forever in an environment without an operating system.

The rest of the program is mainly subroutine-based, making use of labels as subroutine names, and returning values with the `ret` instruction. This allows for relatively modular and readable code, which is a challenge in general, but especially much so when writing in assembly.

Figure 3 details the init section of the program.

Figure 4 details the main loop of the program. From this main loop, many sub-routines are called. These sub-routines are detailed in figures 5, 9, and 10.

At any time, a button interrupt may occur, at which time the button interrupt routine is called. The button interrupt routine is detailed in figure 6. The button interrupt routine calls the subroutines `read_buttons` (figure 8), and `debounce` (figure 7).

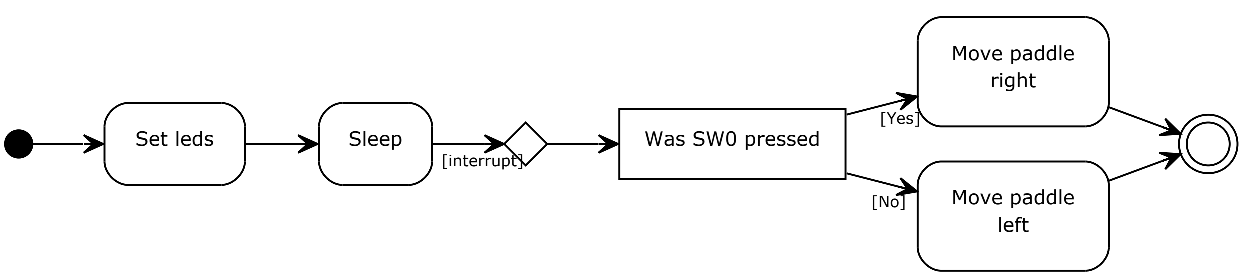

Figure 4: Main loop

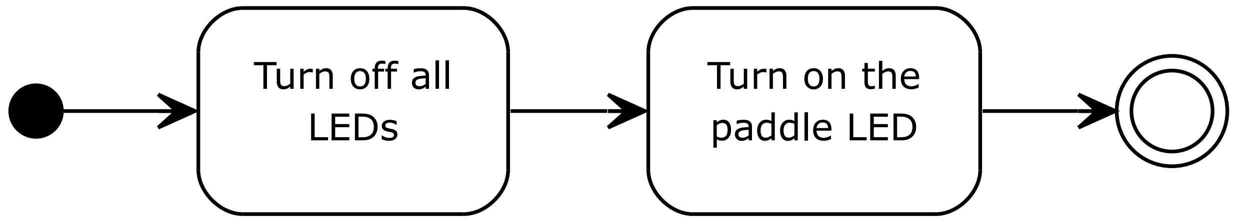

Figure 5: Set leds subroutine

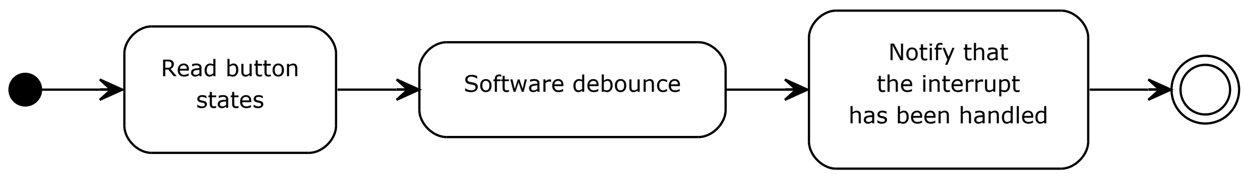

Figure 6: Button interrupt routine

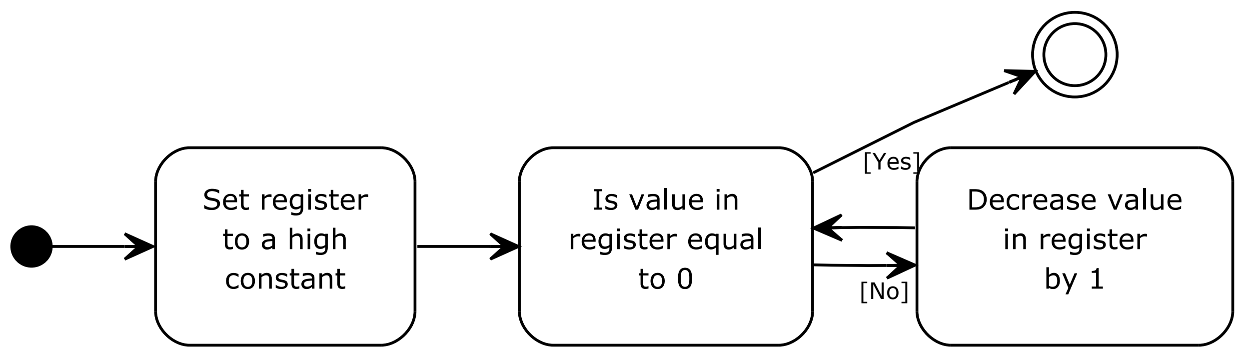

Figure 7: Debounce subroutine

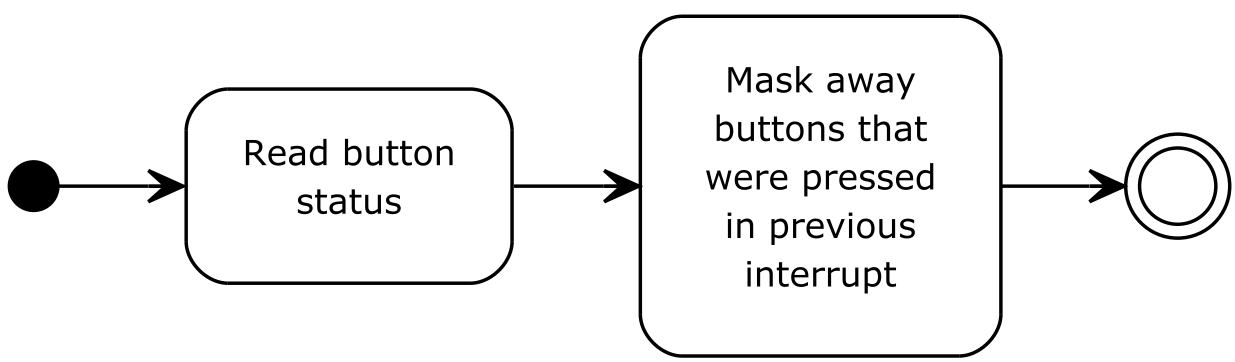

Figure 8: Read buttons subroutine

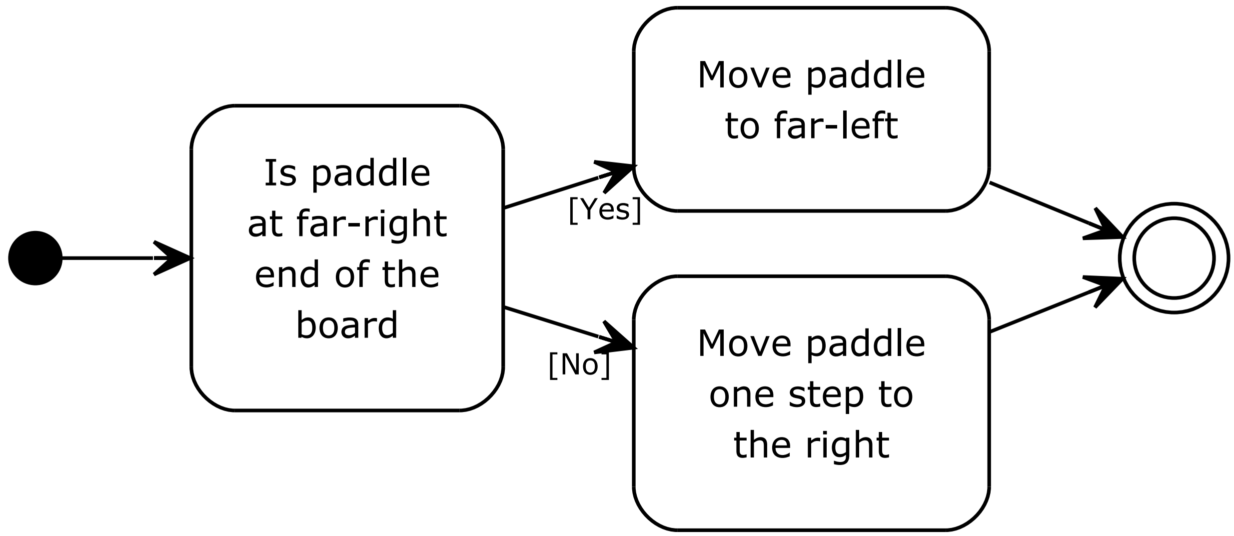

Figure 9: Move paddle right subroutine

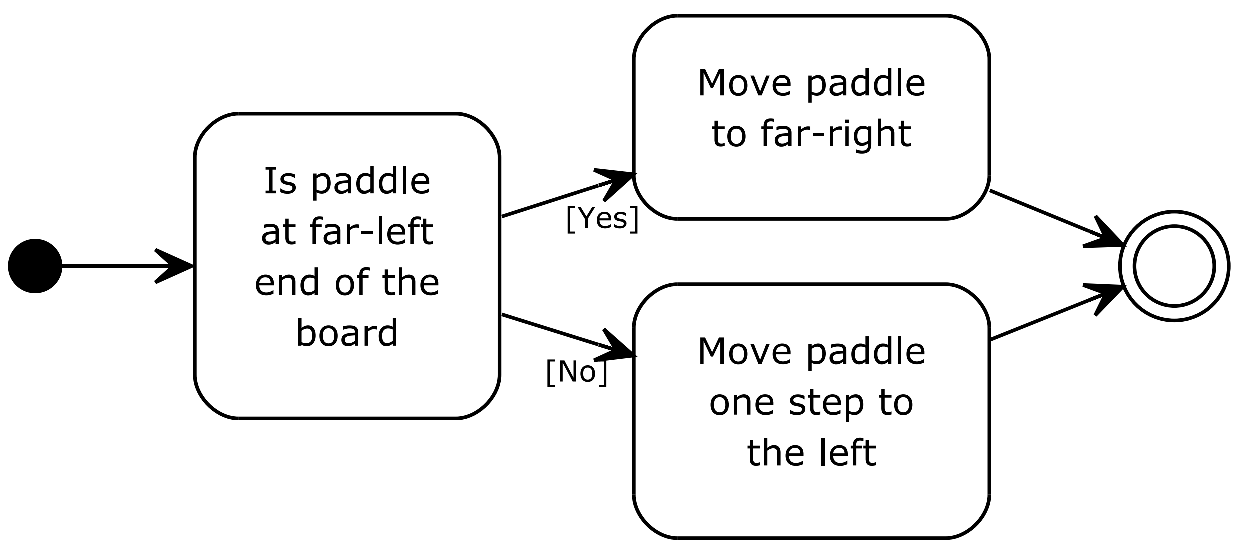

Figure 10: Move paddle left subroutine (analogous to move paddle right subroutine, but included for completeness)

Register	Usage
`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

Table 1: Overview over register use

4.6 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. `r8-12` are used to hold parameters when calling a routine. `r12` is used to hold the return value when returning from a routine.

On the basis of these conventions, a set of rules has been laid down governing what each register of the processor is used for. This list of registers and their usage is presented in table 1.

Part III
Results and Tests

5 Energy Efficiency

The STK1000 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, see figure 11. Current flow through the pins was measured by connecting an ammeter in series. Current flow was measured for 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 button cool-down timers. 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) [5].

The measurements can be found in table 2.

`VDDCORE` (core power supply) and `VDDIO` (I/O power supply) are the interesting measurements, as `AVDDUSB` (usb power supply), `AVDDPLL` (PLL power supply) and `AVDDOSC` (Oscillator power supply) are not actively affected in the assignment.

Using the sum of the power consumption over `VDDCORE` and `VDDIO` 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.

Pins	Interrupt-based idle	active	Busy-wait-based idle	active
`AVDDUSB` (1,2)	1.77mA	1.77mA	1.77mA	1.77mA
`AVDDPLL` (3,4)	1.97mA	1.97mA	1.97mA	1.97mA
`AVDDOSC` (5,6)	0.00mA	0.00mA	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

Table 2: Results of the current consumption measurements of the STK1002

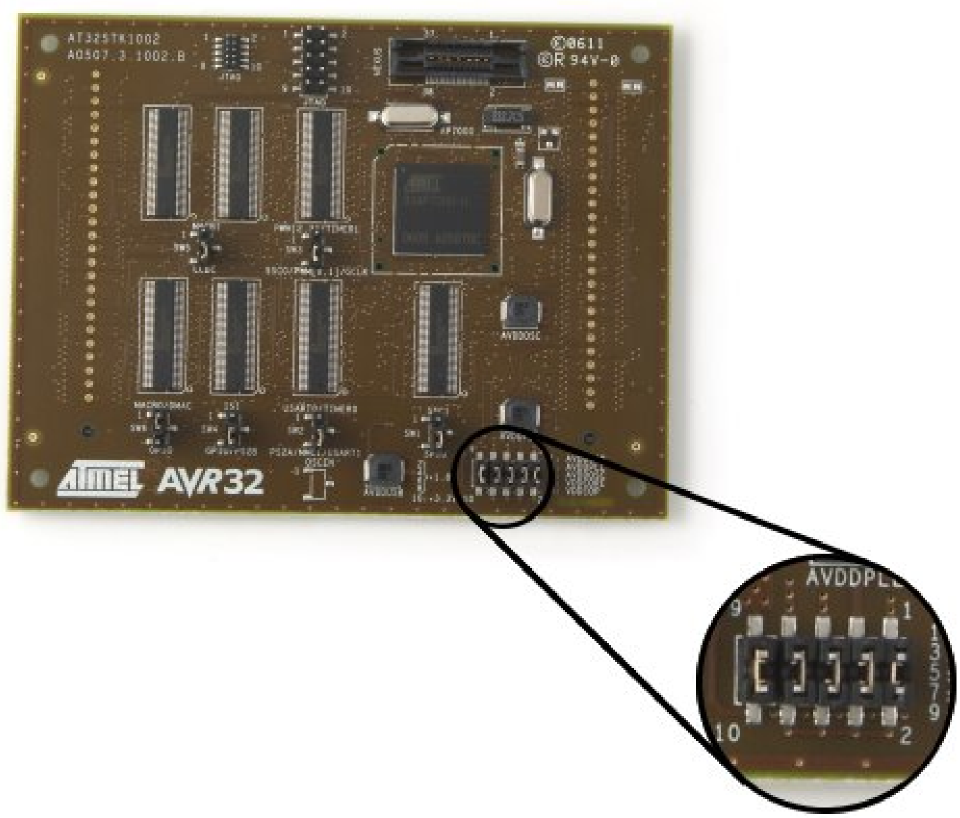

Figure 11: Location of the power pins on the STK1002. Image courtesy of ATMEL.

The power consumption of `VDDCORE` and `VDDIO` of the interrupt-based version is $30.6mA \cdot 1.8V + 4.90mA \cdot 3.3V = 71.3mW$.

The power consumption of the busy-wait version is $25.8mA \cdot 1.8V + 8.05mA \cdot 3.3V = 73mW$.

The difference in power consumption is around 2%, which is near-negligible.

6 Testing

All tests assume that you are in possession of at least one (1) functional STK1000 development board (with cables), a JTAGICE mkII firmware version 7.29 (with USB cable) and a computer with software and hardware capable of interfacing with the JTAGICE.

6.1 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. Optionally, LED0 could be considered to be to the left of LED7, while LED7 could be considered to be to the right of LED0.

6.2 Measurement of Power Consumption

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

- Ammeter
- 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 ammeters's probes, such that the ammeter is connected in series, acting as a replacement for the removed jumper.
3. Press the `RESET` button on the STK1000.

4. Read the measurement from the display of the ammeter.

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, replace the jumper.

7 Discussion

For the current consumption tests, it is likely that the measuring equipment, as well as methods of measurement, could be improved. It seems somewhat unlikely that the transition from a busy wait-based approach to an interrupt-based approach should result in only near-negligible differences in power efficiency, based on previous experiences. One way to measure the current consumption of the entire STK1000 would be to connect a beefier ammeter to the main power supply of the board. Unfortunately, further investigation of the currency consumption measurements falls outside of the scope of these authors' resources.

On the topic of energy efficiency, the power consumption could probably be reduced further by reducing the CPU's clock speed ([3], pg 933-). The program's code can be optimized to use less clock cycles, and fewer clock cycles equals lower power consumption, as the processor can spend more time in a low-power state. The code in this assignment is already reasonably well energy optimized - it spends most of its time in a sleeping state. It also uses as few instructions and does as little IO as possible to the degree that it can without entirely defenestrating readability. The code could be further optimized by rewriting it so that it employs less branches. Branching is expensive in terms of clock cycles, as missed branch predictions will make the CPU pipline less efficient.

Part IV

Evaluation of Assignment

The assignment itself was both challenging and interesting. It provided an environment in which wits and the ability 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 [7] 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 the authors tested was sleep 5. Of course, this rendered the card unusable. Explicitly stating somewhere (e.g. in the compendium) that sleep 5 should never be used will probably reduce the amount of soft-bricked boards by quite a bit.
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, but the authors realize that fixing this probably requires significant resources. In summary: 9/10, would recommend this course to other students.

Part V

Conclusion

A working paddle move program was successfully implemented in an energy-efficient fashion as per the assignment's specifications, despite the difficulties encountered along the way. Although the power consumption tests showed near-negligible changes in the various energy efficiency optimizations of the program, there is reason to doubt their relevance and accuracy. The authors' 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.

References

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

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

[3] Atmel. Avr32 at32ap7000 preliminary. http://www.atmel.com/Images/doc32003.pdf.

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

[5] Caltek/BST. Bst bs1901w manual. http://www.docstoc.com/docs/67828383/BS1901W.

[6] University of Toronto Faculty of Applied Engineering. Lab reports. http://www.engineering.utoronto.ca/Directory/students/ecp/handbook/documents/lab.htm.

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

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

[9] Department of Computer and NTNU Information Science. Lab assignments in tdt4258 energy efficient computer systems. http://www.idi.ntnu.no/emner/tdt4258/_media/kompendium.pdf.

All internet resources were checked on 2013-02-14.