Lab 3

ET014G Programming of Embedded Systems

Erlend Røed Myklebust

Spring 2016

Mittuniversitetet
MID SWEDEN UNIVERSITY

NTNU – Trondheim
Norwegian University of
Science and Technology

Prep 1) What is the difference between EIC and INTC?

The Interrupt Controller (INTC) is the main handler for interrupts on the AVR32UC3A0512. It collects interrupt requests from the peripherals using the interrupt request lines (IREQ – 64 groups with 32 lines each) and provides the CPU with and autovector for each active interrupt, starting with the highest prioritized request. The External Interrupt Controller (EIC) on the other hand, is more like an extension that provides extra features. It is a separate module to the INTC, but is connected to the INTC via eight IREQ-lines and is reliant on the INTC for most of its operation. The EIC allows *any* pin to become an external interrupt pin, regardless of its association to other peripheral modules, and provides some unique features like Non-Maskable Interrupts (NMI) and wake-from-sleep.

Prep 2) How many PWM channels are there?

The PWM module on the AVR32UC3A0512 provides the user with up to seven PWM channels.

Prep 3) Which clock source is required to be running when GPIO interrupts are handled?

The GPIO module is connected to the INTC via the PBA bus on the AVR32UC3A0512. Thus, the PBA clock must be running for GPIO interrupts to be handled.

Prep 4) In the *tc_example3.c*, identify the structure that accepts parameters for waveform generation.

Given that *tc_example3.c* in ASF v.3.29.0 is the same as the one referred to in this question, the parameter configuration for the TC waveform is defined in the variable *waveform_opt.* It is a variable of type *tc_waveform_opt_t* and resides within the local function *tc_init*.

Prep 5) Why is the variation in duty cycling important and where can you use it?

In a PWM signal, the duty cycle denotes the relative relationship between how long the signal is *high* versus how long it is *low* within a single period. For example, a 100 Hz PWM signal running at 75% duty cycle will alternate between spending 7.5 ms in the high state and 2.5 ms in the low state. A PWM signal like this could be used to turn an LED on and off very fast, effectively dimming the light intensity, or it could be used to control the speed of a DC motor.

Task) Develop a PWM function generator with analog control and functionality to display frequency and duty cycle. The components required are potentiometer, ADC, TC, PWM, LCD display and interrupts.

Solving this task was largely an exercise in how to make different ASF drivers work together well enough to allow the required 50 Hz sampling rate. To my surprise, this went really well and I was able to complete the solution in just a few days. I did however discover a few problems related to the desired “real-time” nature of the program.

First of all, the PWM frequency range is too big in my opinion. The potentiometer on the EVK1100 board is quite small and only turns about 225° from zero to maximum, which combined with the 10-bit ADC results in 4.55 digitally represented values per degree. Add in the fact that, in the pdf-version of the task, the required step size in PWM frequency is 100 Hz per step. This equates to about 444 Hz per degree on the POT, making fine-tuning virtually impossible. Needless to say, the adjustment of PWM frequency when updating every 20ms is very unstable. It is constantly jumping between nearby frequencies. The same problem is not present for updating the PWM duty cycle however, because the duty cycle values only span from 0 to 100. A quick and dirty solution to all of this, is simply to adjust the frequency first, then push PB0 and stay in the PWM duty cycle mode. This has the effect of freezing the PWM frequency at whatever value was read by ADC when the push button interrupt occurred, resulting in a much more stable PWM signal.

The second discovery is related to the built in period and cycle update functionality of the PWM module in the AVR32UC3A0512. By writing a new value into the CUPD register and then setting the correct value in the CPD-bit of the CMR register, the new value in CUPD will automatically be transferred to either the CPRD or the CDTY register at the end of the current PWM period. This has the effect of updating either the PWM frequency or the PWM duty cycle of a running PWM signal respectively. And the problem is just that! It can only update one of the values at the time.

Consider the following scenario: The current PWM frequency is 100 kHz and the PWM duty cycle is 75%. On register level, these quantities are represented by the values *period* and *cycle* respectively.

$$period = \frac{pba\_clock}{prescaler * pwm\_frequency} \rightarrow \frac{12\ MHz}{1 * 100\ kHz} = 120$$

$$cycle = \frac{period}{\left(\frac{100}{100 - pwm\_duty\_cycle}\right)} \rightarrow \frac{120}{\left(\frac{100}{100-75}\right)} = 30$$

Then the user wants to change the PWM frequency to 100 Hz by using the update functionality. Rewriting the equations above, gives us a new period value of 120,000 and a new duty cycle value of 30,000. The problem is that only the new period value is transferred to the CPRD register. The CDTY register still contains the value 30. This results in a 100 Hz PWM running at 99.975%, virtually equal to a constant DC *high* signal. Another update, this time to the CDTY, is required to correct the problem, but that will not take effect until another PWM period has passed.

This is why I am using the re-initialization method for updating the PWM parameters, because it instantaneously reconfigures the PWM signal with a complete configuration.

Unfortunately, because it is not synchronized with the current PWM period, the resulting PWM signal may be slightly choppy, but at least the frequency and duty cycle is set correctly.

The project name for this solution is Lab_3_Task. It was developed using Atmel Studio 6.2 with ASF v.3.29.0, AVR32/GNU compiler and linker v.4.4.7, and relies on the following ASF drivers: *ADC, EIC, Generic board support, GPIO, INTC, LCD Display DIP204B, PM Power Manager, PWM, SDRAM* and *TC Timer/Counter.* See Appendix A for the full code.

Note: functions for *ADC, INTC, LCD Display, PWM* and *TC* are included in separate .h-files to improve the readability of *main.c.*

The program starts by configuring the CPU clock to 48 MHz and the PBA clock to 12 MHz, initializing the board, LEDs, LCD Display SPI and the SDRAM. Next, it fills the SDRAM with two 1024 Byte lookup tables. These are used to correlate ADC measurements of the potentiometer with desired frequency values in Hz and duty cycle values in percentage. Then the program continues by configuring the interrupts, one for TC events and one for push button 0 events, followed by an initialization of the ADC, PWM and TC. The TC is configured to waveform 2 (WAVSEL = 0b10) with trigger on compare match with value RC and is set to use clock source 4 (PBA clock / 32 = 375,000 Hz). Sampling time is defined as 20ms, yielding and RC value of 7500. Finally, the ADC and the TC is started and the program enters the main while loop. When a TC interrupt occurs, the *erm_tc_handler* is called and a single identifier (*interrupt_identifier*) is set to the value INT_TC before returning to normal code execution. Within the main while loop, a polling structure keeps checking said identifier to see if an interrupt has occurred. If yes and equal to INT_TC, then it performs an ADC reading of the POT and restarts the ADC. Next, it checks to see the status of a second identifier (*change_identifier*) and depending on its value, either looks up the corresponding frequency or duty cycle value in the SDRAM, before updating the PWM configuration (*erm_pwm_update_config* –function) and re-initializing the PWM signal (*erm_pwm_update_channel* –function). The PWM is always configured to output on channel 3 (pin PB22), which enables the user to see a representation of the PWM signal on LED8.
In addition, LED1 and LED2 toggles each time the TC interrupt and the push button interrupt occurs respectively.

Figure 1 illustrates the contents of the LCD display and the oscilloscope measurement for a 100 Hz PWM signal running at 50 % duty cycle.

Figure 2 illustrates the contents of the LCD display and the oscilloscope measurement for a 5000 Hz PWM signal running at 25 % duty cycle.

Figure 3 illustrates the contents of the LCD display and the oscilloscope measurement for a 5000 Hz PWM signal running at 75 % duty cycle.

Figure 4 illustrates the contents of the LCD display and the oscilloscope measurement for a 100 kHz PWM signal running at 63 % duty cycle.

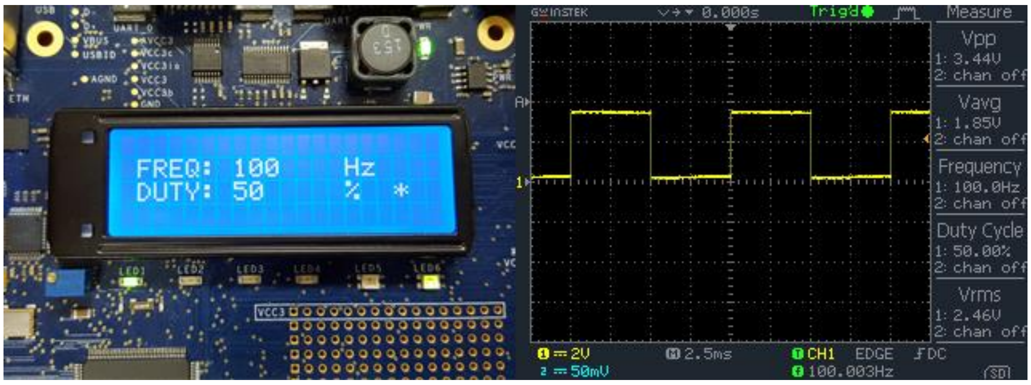


Figure 1. Example showing 100 Hz PWM @ 50% duty cycle

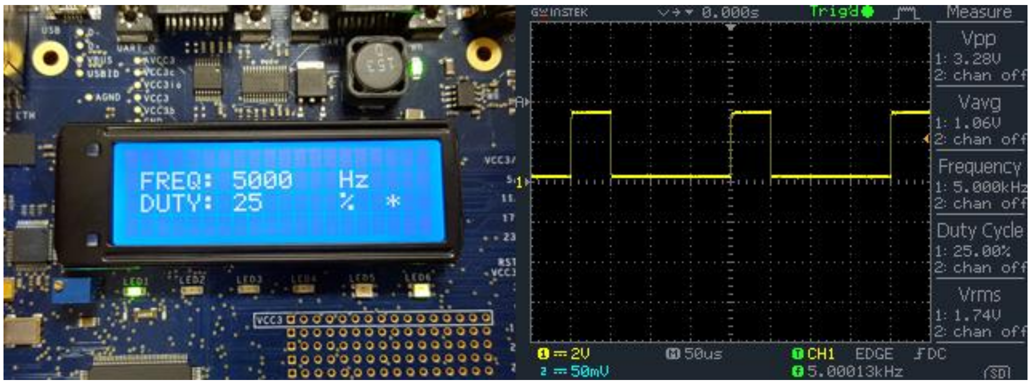


Figure 2. Example showing 5 kHz PWM @ 25% duty cycle

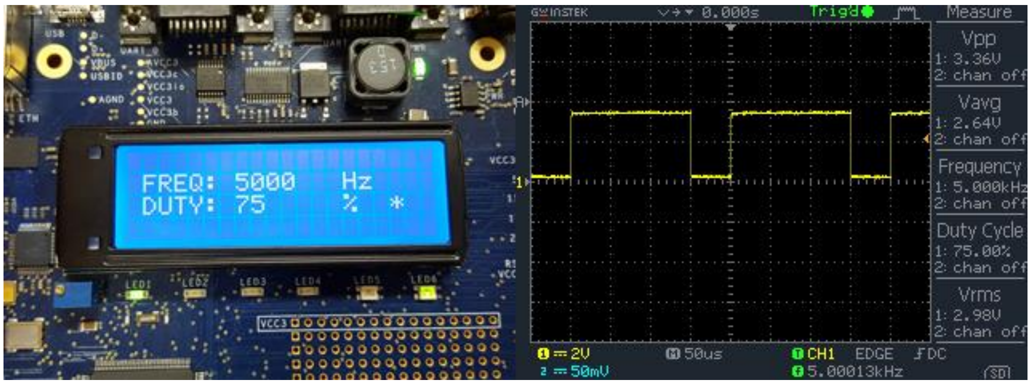


Figure 3. Example showing 5 kHz PWM @ 75% duty cycle

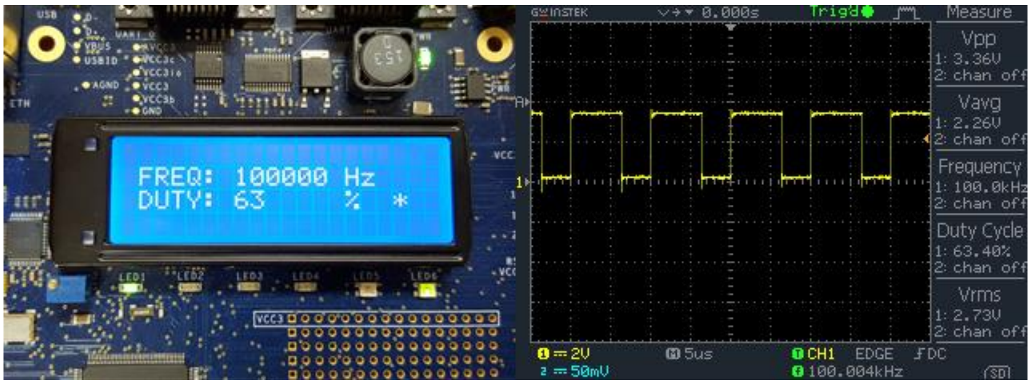
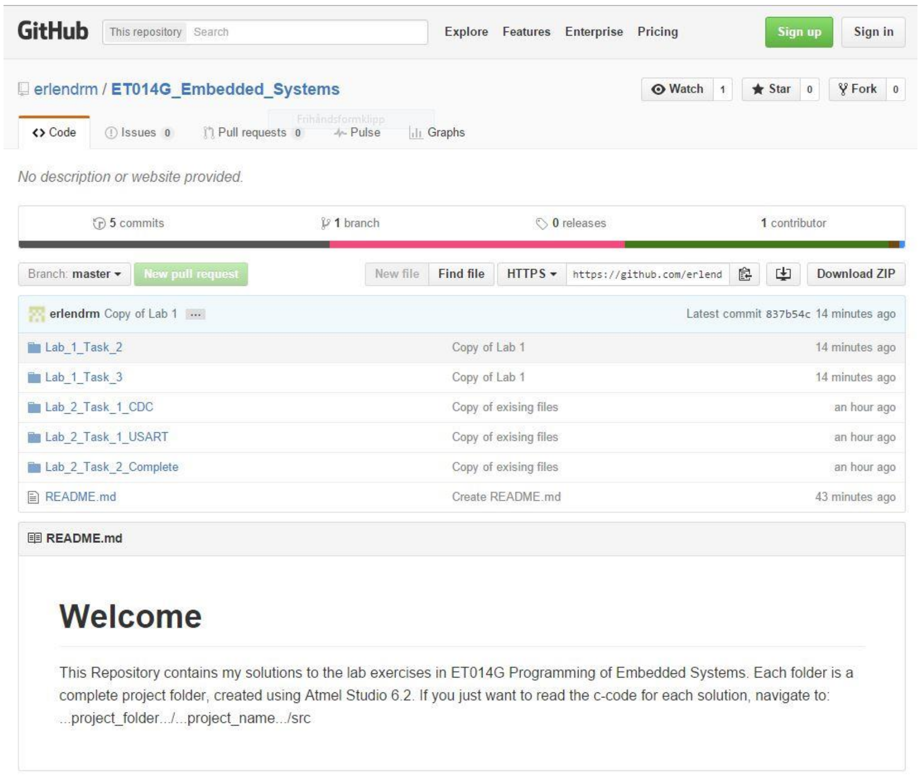


Figure 4. Example showing 100 kHz PWM @ 63% duty cycle

END

Appendix A: Code Repository (GitHub)


GitHub
This repository Search
Explore Features Enterprise Pricing
Sign up Sign in
Watch 1 Star 0 Fork 0
Code Issues 0 Pull requests 0 Pulse Graphs
5 commits 1 branch 0 releases 1 contributor
Branch: master New pull request
New file Find file HTTPS https://github.com/erlend Download ZIP

erlendrm / ET014G_Embedded_Systems

No description or website provided.

erlendrm Copy of Lab 1 … Latest commit 837b54c 14 minutes ago

Lab_1_Task_2	Copy of Lab 1	14 minutes ago
Lab_1_Task_3	Copy of Lab 1	14 minutes ago
Lab_2_Task_1_CDC	Copy of exising files	an hour ago
Lab_2_Task_1_USART	Copy of exising files	an hour ago
Lab_2_Task_2_Complete	Copy of exising files	an hour ago
README.md	Create README.md	43 minutes ago

README.md

Welcome

This Repository contains my solutions to the lab exercises in ET014G Programming of Embedded Systems. Each folder is a complete project folder, created using Atmel Studio 6.2. If you just want to read the c-code for each solution, navigate to: ...project_folder.../...project_name.../src

The GitHub repository “ET014G_Embedded_Systems” contains my code solutions for this laboratory exercise in form of complete project folders.

If you only wish to read the code files for each solution, simply navigate to *Lab_3_Task/Lab_3_Task/src* from the main repository. There you will find the *main.c* and all the additional include files. Note that *main_old_backup.c* is an older version of the code and can safely be ignored.

Please use the following static link to reach the repository:

github.com/erlendrm/ET014G_Embedded_Systems