
Department of Computer Science

TDT4258 - Low-level Programming

# Assignment 1

*Group Number:* 14
*Board Number:* EFM17/3

*Group Members:*
Stian Strømholm
Torbjørn Bratvold
Håkon Kindem
Stian G. Hubred

September, 2020

# Table of Contents



# List of Figures



# List of Tables

# 1 Introduction

This report contains the description of two assembly programs for the EFM32GG Microcontroller which allows a user to use four buttons on a gamepad to interact with eight LEDs on the same gamepad. The first program is implemented with polling, and the second program replaces the polling with the use of interrupts. To analyze the difference between the two solutions, the power consumption has been measured.

The complete project with source code and compiled files can be found at Bratvold et al. [2020].

## 1.1 Polling solution

In the solution using polling, the CPU will continuously check the status of the buttons. This program will in short allow the user to turn on and off the LEDs and rotate them to the left and right. This way the user can produce any of the $2^8$ combinations of on or off LEDs.

## 1.2 Interrupt solution

Using interrupts, the program can minimize unnecessary CPU usage by only interacting with the buttons when they are pressed. Since the interaction between buttons and LEDs is the only function of our program, the CPU can be put to sleep when no interrupts are present and thus conserving energy. From the users perspective, the program has the same behaviour as in the polling solution, turning on/off LEDs and shifting them to the left/right.

# 2 Detailed Tasks

## 2.1 Polling

In this assembly program, the reset handler begins by calling the setup_clock function, which uses the CMU to enable the GPIO clock. It then calls the setup_leds function, where the drive strength of the pins corresponding to the LEDs are set high, before they are configured as outputs. To complete the setup of the LEDs, all the pins except one are set high, which means only one will emit light as the LEDs are active low. Then the setup_buttons function is called, where the buttons are set up as inputs, and internal pull-ups are enabled to keep the input pins stable. The reset handler is concluded by starting the polling function. This is shown in Listing 1 through Listing 4. Constants used in the program, such as 'CMU_BASE' in Listing 2, are listed in Appendix F.

```
        .thumb_func
_reset:
          bl setup_clock
          bl setup_leds
          bl setup_buttons
          b polling
```

Listing 1: This figure shows the assembler function _reset, which runs when the microcontroller starts or resets.

```
        .thumb_func
setup_clock:
          // setup GPIO clock
          ldr r1, =CMU_BASE
          ldr r2, [r1, #CMU_HFPERCLKEN0]
```

```
        mov r3, #1
        lsl r3, r3, #CMU_HFPERCLKEN0_GPIO
        orr r2, r2, r3
        str r2, [r1, #CMU_HFPERCLKEN0]

        bx lr
```

Listing 2: This figure shows the assembler function setup_clock.

```
        .thumb_func
setup_leds:
        // Set high drive strenght
        ldr r1, =GPIO_PA_BASE
        mov r2, #0x2
        str r2, [r1, #GPIO_CTRL]

        // Set LED Pins to output
        mov r2, #0x55555555
        str r2, [r1, #GPIO_MODEH]

        // Set pins low, except the far right one
        mov r2, #0b01111111
        lsl r2, r2, #8
        str r2, [r1, #GPIO_DOUTSET]

        bx lr
```

Listing 3: This figure shows the assembler function setup_leds.

```
        .thumb_func
setup_buttons:
        // Set button pins to input
        ldr r1, =GPIO_PC_BASE
        mov r2, #0x33333333
        str r2, [r1, #GPIO_MODEL]

        // Enable internal pull-up for buttons
        mov r2, #0xff
        str r2, [r1, #GPIO_DOUT]

        bx lr
```

Listing 4: This figure shows the assembler function setup_buttons.

```
                .thumb_func
polling:
        // Loop initialization
        ldr r4, =GPIO_PC_BASE
        ldr r5, [r4, #GPIO_DIN]
        and r5, r5, #0xff

loop:
        ldr r6, [r4, #GPIO_DIN]
        and r6, r6, #0xff
        eor r0, r5, r6 // find what bits have changed
        and r0, r0, r5 // find what bits have changed AND were high at last poll

        // Check for button SW5 (left)
        and r1, r0, #0x10
```

```
cmp r1, #0
it ne
blne led_rl // Rotate LEDs left

// Check for button SW7 (right)
and r1, r0, #0x40
cmp r1, #0
it ne
blne led_rr // Rotate LEDs right

// Check for button SW6 (left)
and r1, r0, #0x20
cmp r1, #0
it ne
blne led_on // Turn on the rightmost LED

// Check for button SW8 (down)
and r1, r0, #0x80
cmp r1, #0
it ne
blne led_off // Turn off the rightmost LED

mov r5, r6 // update 'previous value'
b loop
```

Listing 5: This figure shows the assembler function polling.

Listing 5 shows the polling of the buttons, and how the program implements the behaviour of the LEDs based on the input pins. Register R5 will be used to store the previous value of the button input pins (BIP), and so before the loop begins, this register is initialized by loading the value of the BIP into the register. Note that we only care about the 8 least significant bits in the GPIO_DIN register, as those are the bits that represent the buttons.

Inside the loop, we first load the current value of the BIP into register R6. We then find what bits have changed since the last poll by executing a bitwise xor operation between R5 and R6. The result is placed in register R0, and a '1' bit in this register now represents that the corresponding button have changed its value since the last poll. By performing a bitwise and operation between the R0 and R5 register, and storing the result back in the R0 register, R0 will then show what buttons used to be high, but have now changed. These are the buttons we wish to act on, remember that the buttons are active-low.

For every relevant button, we then check if the corresponding bit in the R0 register is 1, and if it is, we branch to the relevant function/action. In this program,

- pressing button SW5 (left) will rotate the lights to the left
- pressing button SW7 (right) will rotate the lights to the right
- pressing button SW6 (up) will light the rightmost LED
- pressing button SW8 (down) will turn off the rightmost LED

This gives the user the opportunity to create any of the $2^8$ possible combinations of on or off LEDs. Once ull four buttons have been checked, and all the corresponding handles have been branched to, or not, the value of the R6 register is stored in the R5 register, and the loop begins anew. Note that this polling function will be running continuously for the remainder of the programs lifetime.

The assembler functions led_rl, led_rr, led_on and led_off, which are all utilized in the code shown in Listing 5, can be found in Appendix A through D.

## 2.2 Interrupts

Interrupts enable use of power saving measures, but also enables the controller to run other program code while an interrupt handles the LED control, removing the need for polling through loops.

There are similarities in the reset handler in the program using interrupts instead of polling, as shown in Listing 6. The clock for the GPIO controller is configured, and the LEDs and buttons are configured as outputs and inputs in the same way as before. Additionally in this program, interrupt generation and handling is enabled, and GPIO interrupts are set for 1 to 0 transitions on the input pins, as seen in Listing 7. This means that when the buttons are pressed down, the program will jump to the handle at address 0x44 in the exception vector table for even pins, and the handle at address 0x6c in the exception vector table for odd pins. In our program, this is the function gpio_handler in both cases. The exception vector table can be viewed in full in Appendix E. Next the energy mode is set to the more energy efficient mode EM2, as shown in Listing 8. The functions setup_clock, setup_leds, and setup_buttons are the same as in the polling solution, and can be found in Listing 2, Listing 3, and Listing 4 respectively. At the end of the reset handler, the wfi (Wait For Interrupt) operation is executed, and the microcontroller will be put in sleep mode until an interrupt is generated.

```
        .thumb_func
_reset:
          bl setup_clock
          bl setup_leds
          bl setup_buttons
          bl setup_interrupts
          bl enable_energy_mode
          wfi
```

Listing 6: This figure shows the _reset handler for the interrupt solution, which runs when the microcontroller starts or resets.

```
        .thumb_func
setup_interrupts:
          // Set up interrupts
          ldr r1, =GPIO_BASE
          mov r2, #0x22222222
          str r2, [r1, #GPIO_EXTIPSELL]

          // Set interrupts on falling edge
          mov r2, #0xff
          str r2, [r1, #GPIO_EXTIFALL]

          // Enable interrupt generation
          mov r2, #0xff
          str r2, [r1, #GPIO_IEN]

          // Enable interrupt handling
          ldr r1, =ISER0
          ldr r2, =0x802
          str r2, [r1]

          bx lr
```

Listing 7: This figure shows the assembler function setup_interrupts, which enables GPIO falling edge triggered interrupts for the BIP.

```
        .thumb_func
enable_energy_mode:
```

```
        // Set the energy mode
        ldr r1, =SCR
        mov r2, #6
        str r2, [r1]

        bx lr
```

Listing 8: This figure shows the assembler function enable_energy_mode, which sets the energy mode to EM2 in the EMU.

```
        .thumb_func
gpio_handler:
        // Load the IF register to see what triggered the interrupt
        // and write it to the IFC register to clear the interrupt
        ldr r4, =GPIO_BASE
        ldr r5, [r4, #GPIO_IF]
        str r5, [r4, #GPIO_IFC]

        // Write LR to the stack to preserve its value through function calls
        push {lr}

        // Check if it was button SW5 (left)
        and r6, r5, #0x10
        cmp r6, #0
        it ne
        blne led_rl

        // Check if it was button SW7 (right)
        and r6, r5, #0x40
        cmp r6, #0
        it ne
        blne led_rr

        // Check if it was button SW6 (up)
        and r6, r5, #0x20
        cmp r6, #0
        it ne
        blne led_on

        // Check if it was button SW8 (down)
        and r6, r5, #0x80
        cmp r6, #0
        it ne
        blne led_off

        // Pop the original value of LR back, and return
        pop {lr}
        bx lr
```

Listing 9: GPIO interrupt handler

Listing 9 shows the code executed in the GPIO interrupt handler. The interrupt is immediately cleared to make sure its not re-run without another GPIO signal. This is followed by a code segment similar to the polling solution which checks button status and branches to the corresponding function/action. Rather than comparing the current values of the BIP with the previous values of the BIP, as in the polling solution, the GPIO_IF register is used to determine the source of the interrupt. Note that the value of the link register is pushed to the stack before any function calls are executed, so that the same value can be retrieved with a pop operation before returning from

the interrupt. The interrupt clears and is returned to executing wfi instruction at the end of the handler. The assembler functions led_rl, led_rr, led_on, and led_off are the same as used in the polling solution, and can be found in Appendix A through D.

## 2.3 Energy consumption

In Figure 1 to Figure 5, the power consumption of the microcontroller is displayed for different situations. The different situations and their respective power consumption are summarized in Table 1.

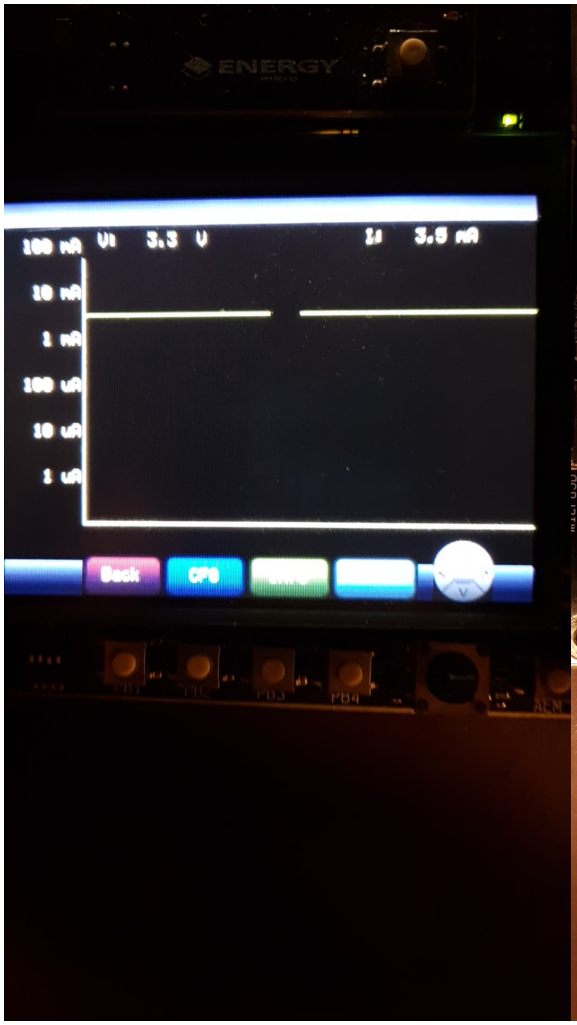


Figure 1: Power consumption with polling and no lit leds ($3.5mA$)

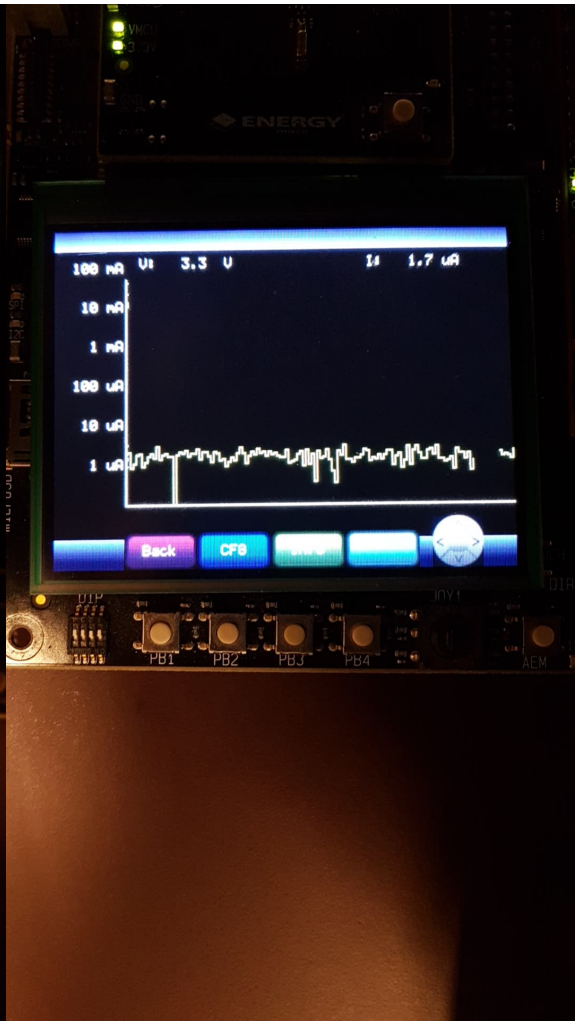


Figure 2: Power consumption with interrupt and no lit leds ($1.7\mu A$)

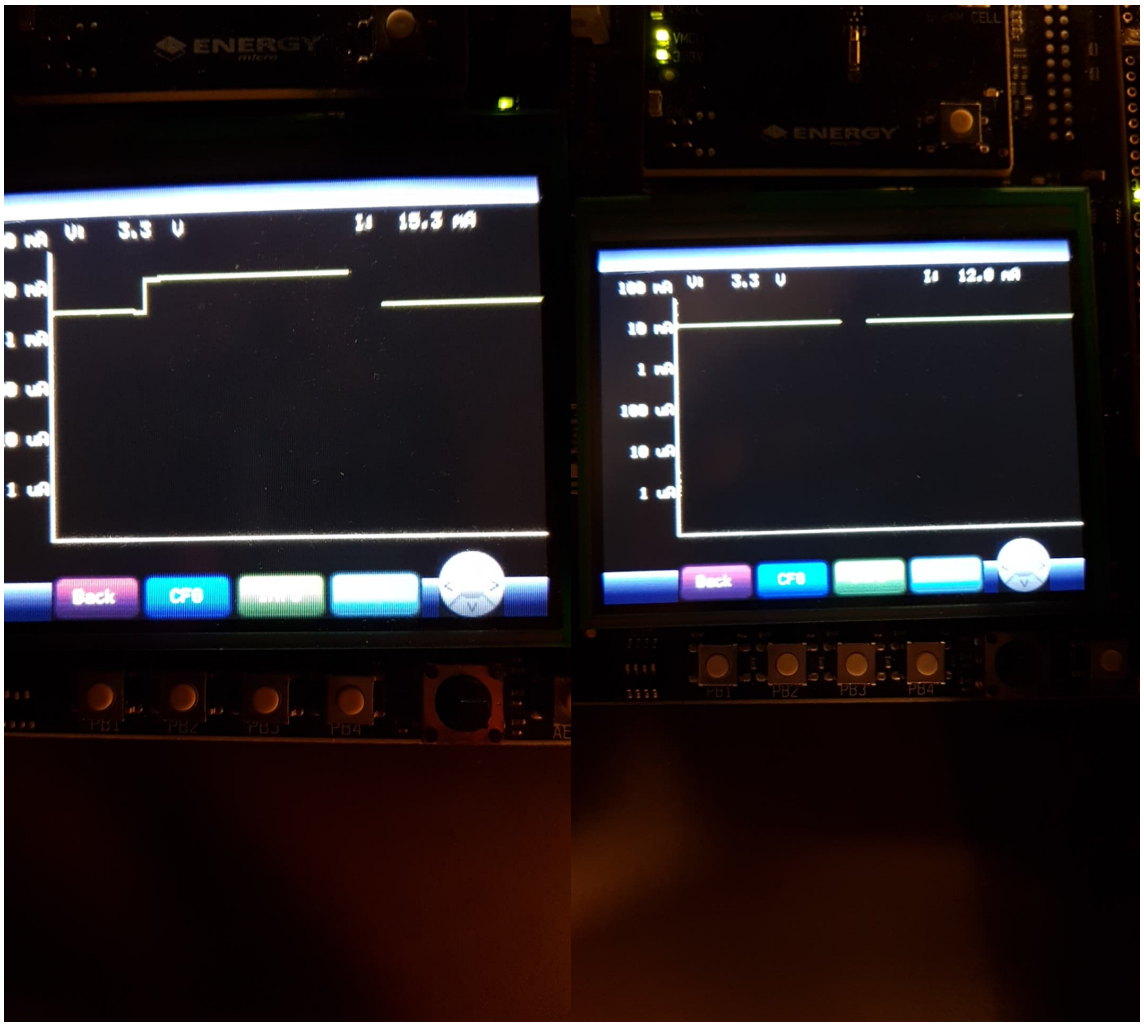


Figure 3: Power consumption with polling and one lit led

Figure 4: Power consumption with interrupt and one lit leds

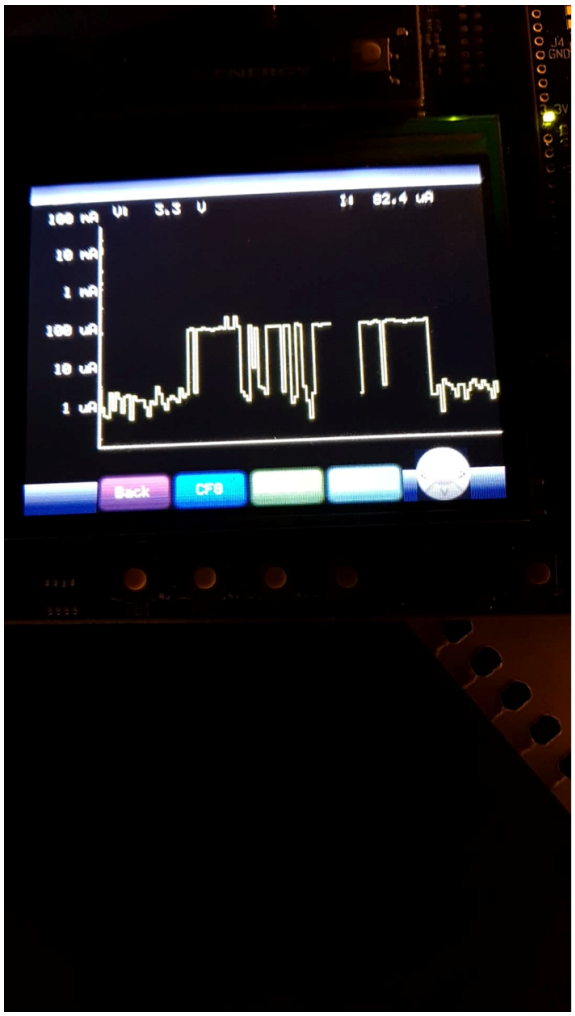


Figure 5: Interrupts. Buttons pressed in high frequency and no lit LEDs

| **Situation** | **Power consumption** |
|---|---|
| Polling, no active LEDs | 3,5mA |
| Interrupts, no active LEDs | 1,7$\mu$A |
| Polling, one lit LED | 15,3mA |
| Interrupts, one lit LED | 12mA |
| Interrupts, high frequency button pressing, no active LEDs | 100$\mu$A |

Table 1: Power consumption of the microcontroller for different situations

We see from Figure 1 and Figure 2 that we have a power consumption of 3.5$mA$ with polling and 1.7$\mu$ with interrupt. The power consumption with interrupts varied around +/-0.3$\mu$ from 1.7$\mu A$. This variation can be a result of low signal-to-noise ratio.

In Figure 3 and Figure 4 we also compare polling to interrupts, but this time with one lit LED on the game pad. The power consumption with polling was 15.3$mA$ and 12.0$mA$ with interrupts.

Figure 5 shows the consumption of the interrupt program when buttons are pressed in a high frequency manner. No LEDs were lit in this situation. The average energy, highly dependent on how the buttons are pressed of course, seems to be around 100$\mu$A

### 2.4 Discussion of results and observations

By comparing the energy consumption of the polling and interrupt solution, we can see that the interrupts clearly leads to lower power consumption. When there are no lit leds, the polling solution consumes 3,5$mA$ while the interrupt solution only consumes 1,7$\mu$. This highlights the effect of the CPU going into deep sleep instead of constantly checking the status of the input pins. When one LED is lit, the polling solution uses 15,3$mA$ compared to 12.0$mA$ with interrupts. Since a LED is lit, the power consumption is rather high in both cases, but the interrupt solution still uses less power.

In the scenario where the buttons were pushed in a high frequency manner, the interrupt solution rises to an average of around 100$\mu$A. This is still less than the 3,5$mA$ that the polling solution produces in a "equivalent" situation. The results seem to indicate that the polling solution consistently uses ∼3,4mA more than the interrupt solution. The frequency of the interrupt generation was in this case limited by a human pressing buttons. It can be worth noting that a computer can generate interrupts at a much higher frequency, and this might result in the polling implementation being more energy efficient.

## 3 Conclusion

In this report, two different ways of implementing a program which allows a user to control eight LEDs on a gamepad using four buttons on the same gamepad are discussed. One solution uses polling, and the other uses interrupts. Through power measurements, it is clear that using interrupts is superior compared to polling in terms of energy efficiency, although a scenario where this might not be the case have been discussed.

Interrupts seem to simplify program execution whenever a controller is to execute specific actions based on inputs or discrete events, such as GPIO transitions, timing events, or fault handling within the controller. In this case the code can be cleanly structured by removing the handling of these events from the main program loop, separating them into separate event handlers which are coupled to specific interrupts. Furthermore this enables vastly more complex functionality as the CPU is freed from monitoring these events, leaving it to complete other tasks.

# Bibliography

Torbjørn Bratvold, Håkon Kindem, Stian Strømholm, and Stian G. Hubred. Github for tdt4258 lab 1, 2020. URL `https://github.com/t-ber/TDT4258-Maskinn-r-programmering.git`.

# Appendix

## A Assembler function led_rl

```
                    .thumb_func
led_rl: // ROTATE LEDS LEFT (shift leds left by shifting bits right)
                    // Get the current value of the LEDs
                    ldr r0, =GPIO_PA_BASE
                    ldr r1, [r0, #GPIO_DOUT]
                    mov r3, #0xff
                    lsl r3, #8
                    and r1, r1, r3

                    // Invert the bits so that 1 = on, 0 = off
                    // and shift them to the right so we can work with immediate values
                    eor r1, r1, r3
                    lsr r1, r1, #8

                    // Check if leftmost bit is on
                    // If it is, turn on the rightmost bit after shifting
                    and r2, r1, #0b00000001
                    lsr r1, r1, #1
                    cmp r2, #0b00000001
                    it eq
                    orreq r1, r1, #0b10000000

                    // Shift bits back, turn those on and others off
                    lsl r1, r1, #8
                    str r3, [r0, #GPIO_DOUTSET]
                    str r1, [r0, #GPIO_DOUTCLR]

                    bx lr
```

## B Assembler function led_rr

```
                .thumb_func
led_rr: // ROTATE LEDS RIGHT (we shift leds right by shifting bits left)
                // Get the current value of the LEDs
                ldr r0, =GPIO_PA_BASE
                ldr r1, [r0, #GPIO_DOUT]
                mov r3, #0xff
                lsl r3, #8
                and r1, r1, r3

                // Invert the bits so that 1 = on, 0 = off
                // and shift them to the right so we can work with immediate values
                eor r1, r1, r3
                lsr r1, r1, #8

                // Check if rightmost bit is on
                // If it is, turn it off before shifting,
                // and turn on the leftmost bit after shifting
                and r2, r1, #0b10000000
                cmp r2, #0b10000000
                ittte eq
                andeq r1, r1, #0b01111111
                lsleq r1, #1
                orreq r1, r1, #0b00000001
                lslne r1, r1, #1

                // Shift bits back, turn those on and others off
                lsl r1, r1, #8
                str r3, [r0, #GPIO_DOUTSET]
                str r1, [r0, #GPIO_DOUTCLR]

                bx lr
```

## C Assembler function led_on

```
                    .thumb_func
led_on: // TURN ON THE RIGHTMOST LED
                    ldr r1, =GPIO_PA_BASE
                    mov r2, #0x80
                    lsl r2, r2, #8
                    str r2, [r1, #GPIO_DOUTCLR]

                    bx lr
```

## D Assembler function led_off

```
                    .thumb_func
led_off: // TURN OFF THE RIGHTMOST LED
                      ldr r1, =GPIO_PA_BASE
                      mov r2, #0x80
                      lsl r2, r2, #8
                      str r2, [r1, #GPIO_DOUTSET]

                      bx lr
```

## E Exception Vector Table

```
.section .vectors

  .long stack_top /* Top of Stack */
  .long _reset /* Reset Handler */
  .long dummy_handler /* NMI Handler */
  .long dummy_handler /* Hard Fault Handler */
  .long dummy_handler /* MPU Fault Handler */
  .long dummy_handler /* Bus Fault Handler */
  .long dummy_handler /* Usage Fault Handler */
  .long dummy_handler /* Reserved */
  .long dummy_handler /* Reserved */
  .long dummy_handler /* Reserved */
  .long dummy_handler /* Reserved */
  .long dummy_handler /* SVCall Handler */
  .long dummy_handler /* Debug Monitor Handler */
  .long dummy_handler /* Reserved */
  .long dummy_handler /* PendSV Handler */
  .long dummy_handler /* SysTick Handler */

  /* External Interrupts */
  .long dummy_handler
  .long gpio_handler /* GPIO even handler */
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long gpio_handler /* GPIO odd handler */
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
  .long dummy_handler
```

```
.long dummy_handler
.long dummy_handler
.long dummy_handler
```

## F Constants specific to the EFM32GG Microcontroller

```
////////////////////////////////////////////////////////////////////////////
//
// Various useful I/O addresses and definitions for EFM32GG
//
////////////////////////////////////////////////////////////////////////////


  ////////////////////////////////////////////////////////////////////
  // GPIO

  GPIO_PA_BASE = 0x40006000
  GPIO_PB_BASE = 0x40006024
  GPIO_PC_BASE = 0x40006048
  GPIO_PD_BASE = 0x4000606c
  GPIO_PE_BASE = 0x40006090
  GPIO_PF_BASE = 0x400060b4

  // register offsets from base address
  GPIO_CTRL = 0x00
  GPIO_MODEL = 0x04
  GPIO_MODEH = 0x08
  GPIO_DOUT = 0x0c
  GPIO_DOUTSET = 0x10
  GPIO_DOUTCLR = 0x14
  GPIO_DOUTTGL = 0x18
  GPIO_DIN = 0x1c
  GPIO_PINLOCKN = 0x20

  GPIO_BASE = 0x40006100

  // register offsets from base address
  GPIO_EXTIPSELL = 0x00
  GPIO_EXTIPSELH = 0x04
  GPIO_EXTIRISE = 0x08
  GPIO_EXTIFALL = 0x0c
  GPIO_IEN = 0x10
  GPIO_IF = 0x14
  GPIO_IFC = 0x1c

  ////////////////////////////////////////////////////////////////////
  // CMU

  CMU_BASE = 0x400c8000

  CMU_HFPERCLKDIV = 0x008
  CMU_HFPERCLKEN0 = 0x044

  CMU_HFPERCLKEN0_GPIO = 13

  ////////////////////////////////////////////////////////////////////
  // NVIC

  ISER0 = 0xe000e100
  ISER1 = 0xe000e104
  ICER0 = 0xe000e180
  ICER1 = 0xe000e184
```

```
ISPR0 = 0xe000e200
ISPR1 = 0xe000e204
ICPR0 = 0xe000e280
ICPR1 = 0xe000e284
IABR0 = 0xe000e300
IABR1 = 0xe000e304
IPR_BASE = 0xe000e400
IPR0 = 0x00
IPR1 = 0x04
IPR2 = 0x08
IPR3 = 0x0c

////////////////////////////////////////////////////////////////////
// EMU

EMU_BASE = 0x400c6000

EMU_CTRL = 0x000

////////////////////////////////////////////////////////////////////
// System Control Block

CR = 0xe000ed10
```