NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4258 - Low-level Programming

Assignment 1

Group Number: 1
Board Number: EFM17/18

Group Members:
Kasper Søreide
Ole Sivert Aarhaug
Leik Lima-Eriksen
Bjørn Brodtkorb
Audun Asdal

September, 2020

Table of Contents

1 Introduction

This report explains the setup of two possible solutions to control GPIO-pins on the EFM32GG, and compares the performance in terms of latency, complexity and power consumption. The first solution utilizes polling. In essence, this consists of continuously checking whether the button is pressed or not in an infinite loop, and toggling the LED accordingly. In the second solution, we take advantage of the interrupt service routines and the deep sleep functionality. Here, the MCU is in deep sleep mode by default. When a button is toggled, an interrupt is triggered and wakes up the MCU from deep sleep. The LED is then toggled to its correct state, and the MCU goes to sleep again. One would intuitively expect that the first proposal has a much higher power consumption and a lower response time because it is busy waiting. But is this really the case?

2 Detailed Tasks

We will refere to functions and part of our code but not have it inline in these sections. Please see appendix A for polling version and appendix B for the interrupt version

2.1 Common setup

Altough being quite different solutions to the same problem, they both have some setup in common. First of all, the EFM32GG has a lot of power saving features, and so it requires the GPIO clock (CMU_HFPERCLKEN0 and CMU_HFPERCLKEN0_GPIO) to be enabled for those GPIO pins which are in use. In our case, this happens to be the GPIO0 clock. The setup is done uppermost in the _reset section. Next, we need to specify the output driver strength - in other words, how much power the output pins should be able to push through the connected LEDs. This is done by writing directly to register $GPIO\_PA\_CTRL$ and configuring the drivemode bits.

The buttons do not have pull-ups connected to the external circuitry. This would normally cause a problem, since we would have had the pins floating when the buttons are not pressed down. However, the MCU does have support for internal pull-up resistors which can be enabled programatically. This has been done in _btn_mode_loop, where we access the $GPIO\_MODEL/H$ registers and set the corresponding bits to 1 on the pins which needs to be pulled up. Lastly, we need to specify the direction of the GPIOs which the buttons and the LEDs are connected to. This is done uppermost in _btn_mode_loop and _led_mode_loop and is again done by setting the corresponding bits in $GPIO\_MODEL/H$

2.2 Polling version

The polling solution of the problem is organized around an infinite loop. Every time it executes one loop iteration, the button pin data status register is read from $GPIO\_DIN$ and fed into the LED output register $GPIO\_DOUT$. There are eight buttons on the game board which are mapped to their respective bits in a status register $GPIO\_DIN$. In the same way, each bit in a different GPIO output register $GPIO\_DOUT$ corresponds to whether or not a LED is turned on. In order to trigger an update of the LEDs based on the button status, the button DIN register is simply shiftet eight bits to the left and transferred to LED DOUT. This is continously done within a loop. The status on each button is reflecten on the corresponding connected LED.

Currently, no power saving has been implemented in this solution, but this does not mean that no such features are feasible. One way of decreasing the power consumption is by choosing a lower clock cycle. The frequency of the clock is intimately related to the power consumption, and so this could save us a lot of power. THE EFM32GG does come with an Ulta Low Frequency RC oscillator running at 1kHz after prescaling, which is considerably lower than its normal clock cycle of 48MHz. This is however not as easy as it seems, since it requires the manipulation of multiple configuration register, which is quite cumbersome and error prone to do in assembly (more commonly done in

C/C++). Thus it has been left out of this solution. Another drawback of the clock cycle reduction is that the MCU executes instructions more slowly, which in turn results in a high latency. However, the main loop is only a few instructions long, and so the delay would not be any longer than a couple of milliseconds.

2.3 Interrupt version

To implement an improved interrupt based solution, the update of the LEDs will occur in a interrupt routine. Global interrupts has to be unmasked in the PSR by running the instruction "cpsie i". An interrupt signal to the processor will then cause a vector table lookup, stack frame saving (lazy stacking depending on the configuration) and a jump to the vector address. Normally in such systems multiple interrupt signals can trigger at the same time. This is solved with an interrupt controller which handles the interrupt serializing and priority ordering. Often in a Cortex-M MCU this interrupt controller is the NVIC. This must be configured for each interrupt with a priority. The _enable_interrupt routine enables the NVIC for a spesific interupt number. The priority controller is not configured. The routine set the bit corresponding to the interrupt number in the $NVIC\_ISER$. At this point a peripheral can assert the NVIC request line and trigger an interrupt. Lastly the peripheral has to be configured to assert the interrupt line when an event occurs. The peripheral offers flexibility by including multiple trigger sources. This is configured in the following registers; $GPIO\_EXTIPSELL$, $GPIO\_EXTIFALL$, $GPIO\_EXTIRISE$, $GPIO\_IFC$ and $GPIO\_IEN$.

To be able to know what to do when an interrupt occures, an interrupt vector table is used. This can be see in the section .section .vectors. This will hold the address of the routine to jump to after the stack frame saving. Since we have enabled interrupts on all the buttons, the function gpio_handler is called every time a button is pressed. In the same was as in the polling version the button status register is just written directly to the LED data register. This way one button, read as one bit in the button status register is directly translated to one bit in the LED register, and therefore controls one LED. The interrupt flags are also cleared by reading what current interrupt is called $GPIO\_IF$ and clearing it by writing the same bit to $GPIO\_IFC$. If this is not done the interrupt will continue to trigger even witout a button press.

When the interrupt routine returns, the microcontroller goes back to deep sleep. This turns off large parts of the microcontroller, and uses a slower clock. As speed equals power, it is natural to assume that the microcontroller will use considerably less current by slowing the speed. Turning off parts off the microcontroller will of course also help the current consumption.

To put the microcontroller into deep sleep some bits in the SCR register has to be set. The three bits that we set in the SCR register which are called SEVONPEND, SLEEPDEEP and SLEEPONEXIT. SEVONPEND makes the microcontroller wake on external event interrupts. SLEEPDEEP sets the microcontroller to go into the deep sleep state, where the clock is stopped. The microcontroller still has the FCLK enabled to be able to sense interrupts. The SLEEPONEXIT bit makes the microcontroller immidiatly go to sleep when an interrupt handler is done. This makes the microcontroller wake only when an event occurs, and it stays in sleep all the remaining time. When this register is set with these "wfi" is called. This stands for "Wait For Interrupt". This means an interrupt is needed to be able to wake the microcontroller again and continue processing.

3 Results

The solutions have been compiled, linked and tested. The results are listed below in figure 1 and figure 2 as well as table 1. As we can see, the polling based solution uses considerably more power than the interrupt based solution.

Figure 1: Current consumption for the polling solution when idle, when one button is pressed continously and when all buttons are pressed respectively.

The current consumption for the interrupt based solution is shown in figure 2. The values might be a little hard to read, but are also shown in table 1.

The current consumption for the two versions of the code are shown i table 1. The power consumption is shown both when the LEDs power consumption is counted and when it's not.

Table 1: Current consumption for several cases.

	Idle	All buttons pushed
Polling	3.5mA	3.8mA
Interrupts	1.3uA	80.8uA
Polling with LEDs counting	3.5mA	79.8mA
Interrupts with LEDs counting	1.3uA	79.8mA

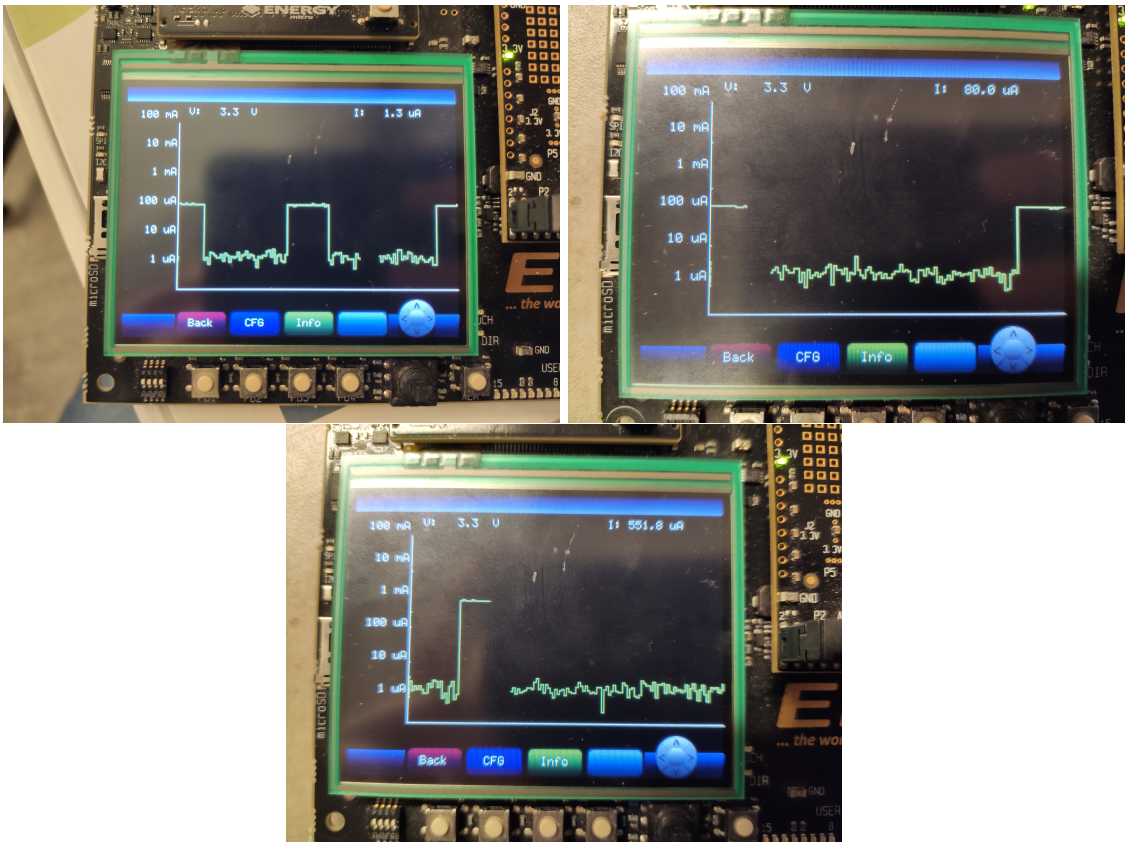

Figure 2: Current consumption for the interrupt based solution when idle, when one button is held continously and when all buttons are pressed respectively.

4 Discussion

We see that the polling based solution consumes more than 2000 times as much power as the interrupt based solution consumes when idle. We also see that the LEDs dominates the power consumption totally, so that if the LEDs are on, it doesn't really matter whether the polling version or the interrupt solution is used. If the LEDs are supposed to be normally off and the current consumption is important, this makes the choice between the two versions easy.

The difference in the responsiveness between the interrupt version and the polling version is hard to state exactly. The interrupt will cause a stack frame saving which will take additional cycles comapred to the polling version. However, this difference might be negligble compared to bus matrix delay. In terms of complexity the interrupt based solution is a bit more complex than the polling solution. The polling version is a simple loop that reads and writes a register, and that's it. The interrupt based solution recuires the vector table to be setup and the NVIC to be programmed. However, these are almost allways present in a system so the added complexity is subtile. In order to minimize power consuption sleep mode has to be set up. This is possible since the exit from sleep mode is interrupt driven. Sleep isn't really nessessary to use interrupts, but it really makes sense to implement sleep together with the interrupt part, as this utilizes the power efficiency. Because of the improved performance and power efficiency, and almost no added complexity, the interrupt mode is almost allways chosen over polling.

Appendix

A Polling code

```
.syntax unified

.include "efm32gg.s"

/////////////////////////////////////////////////////////////////////////////
//
// Exception vector table
// This table contains addresses for all exception handlers
//
/////////////////////////////////////////////////////////////////////////////

.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   dummy_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   dummy_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

.section .text

        /////////////////////////////////////////////////////////////////////////////
        //
        // Reset handler
        // The CPU will start executing here after a reset
        //
        /////////////////////////////////////////////////////////////////////////////


/* Buttons C[0..7] */
/* Leds A[8..15]   */

.globl  _reset
.type   _reset, %function
.thumb_func
_reset:
        /* Enable the GPIO clock */
        ldr r1, =CMU_BASE
        ldr r2, [r1, #CMU_HFPERCLKEN0]
        mov r3, #1
        lsl r3, r3, #CMU_HFPERCLKEN0_GPIO
        orr r3, r3, r2
        str r3, [r1, #CMU_HFPERCLKEN0]
```

```
        /* Set driver stregth */
        ldr r1, =GPIO_PA_BASE
        mov r2, #2
        str r2, [r1]

        /* LED initialization */
        mov r3, #8
        mov r2, #0

_led_mode_loop:
        lsl r2, r2, #4
        orr r2, #4
        subs r3, r3, #1
        bne _led_mode_loop
        ldr r1, =GPIO_PA_BASE
        str r2, [r1, #GPIO_MODEH]

        /* Turn all pins off */
        ldr r2, =0xffff
        str r2, [r1, #GPIO_DOUTSET]

        /* Button initialization */
        mov r3, #8
        mov r2, #0

_btn_mode_loop:
        lsl r2, r2, #4
        orr r2, #2
        subs r3, r3, #1
        bne _btn_mode_loop
        ldr r1, =GPIO_PC_BASE
        str r2, [r1, #GPIO_MODEL]

        /* Set the pull-mode to pull-up */
        ldr r2, =0xffff
        str r2, [r1, #GPIO_DOUTSET]


/* For testing button without interrupt */
_button_loop:
        ldr r1, =GPIO_PC_BASE
        ldr r2, [r1, #GPIO_DIN]
        lsl r2, r2, #8
        ldr r1, =GPIO_PA_BASE
        str r2, [r1, #GPIO_DOUT]
        b _button_loop



/*
 * Standard interrupt handling. CPU saves the stack frame consisting
 * of R0-R3, R12, LR, PC, PSR.
 */
.thumb_func
gpio_handler:
        b .

.thumb_func
```

```
dummy_handler:
        b .
```

B Interrupt code

```
.syntax unified

.include "efm32gg.s"

/*
 * The vector table at address 0 or whatevers defines in SCB->VTOR is what
 * the processor look up into, to find the right interrupt address to run. The
 * negative numbers is defines for fault exceptions and OS suff like SVC and
 * SysTick. The positive numbers from 0 is user vendor implemented exceptions.
 * Normally you should add an alias in the assembly routine, so you can just
 * override the functions, but this is also OK...
 */
.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

.section .text

/* Buttons C[0..7] */
/* Leds A[8..15]   */

.globl  _reset
.type   _reset, %function
.thumb_func
_reset:
        /* Enable the GPIO clock */
        ldr r1, =CMU_BASE

        /* Read the content of the clock enable register in the clock management
         * unit and stores it on r2 */
        ldr r2, [r1, #CMU_HFPERCLKEN0]

        /* Moves the number 1 into r3 */
        mov r3, #1

        /* Bitshifts the number 1 the number of position defined in CMU_HFPERCLKEN0_G
        lsl r3, r3, #CMU_HFPERCLKEN0_GPIO

        /* Perform bitwise or between the r3 and r2. This gives the final bitmask */
```

```
        orr r3, r3, r2

        /* Store the bitmask back again.  */
        str r3, [r1, #CMU_HFPERCLKEN0]

        /*
         * The above part is much simple if the registers implements a set-register
         * and a clear register.
         */

        /* Set driver stregth */
        ldr r1, =GPIO_PA_BASE

        /* Move the number 2 into r2 */
        mov r2, #2

        /* Stores the number 2 in the address of GPIO_PA_BASE aka the first register *
        str r2, [r1]

        /* LED initialization */

        /* R3 wil hold the number of times to loop => 8 */
        mov r3, #8
        mov r2, #0

        /* Label - when writing b label this functions is branched to */
_led_mode_loop:
        /* Sets the mode value to 4 and bitshift r0 times. This starts at zero
         * thus bitshifting zero times and increment with 4 each time */
        lsl r2, r2, #4

        /* Move in 4 to the lowest 4 bits in r2 */
        orr r2, #4

        /* Subtract the loop counter, the s suffix means that the CPSR register will
         * be updated as well. This means the status flags will be updated. A normal
         * sub + a cmd will do the same thing */
        subs r3, r3, #1

        /* Branches to the lable if the result of the status operation subs was not
         * zero */
        bne _led_mode_loop

        /* The loop has iterated 8 times and the r2 hold the entire bitmask for
         * all the leds */

         /* Load the GPIO_PA_BASE address into r1 */
        ldr r1, =GPIO_PA_BASE

        /* Store the final bitmask into the GPIO_PA_BASE + GPIO_MODEH */
        str r2, [r1, #GPIO_MODEH]

        /* Turn all pins off */
        ldr r2, =0xffff

        /* Store a 1 on the lowest 16 bits intro the DOUT register. Thus turing
         * off all leds */
        str r2, [r1, #GPIO_DOUTSET]
```

```
        /* Button initialization */
        mov r3, #8
        mov r2, #0

_btn_mode_loop:
    /* Bitshift the gpio mask with 4 each time. This first time does not matter
         * since the value is zero. */
        lsl r2, r2, #4

        /* Stor the button operation 2 into the right 4 bits describing the button
         * in the MODEL */
        orr r2, #2

        /* Subtract the loop counter, look above for the subs */
        subs r3, r3, #1

        /* If not done 8 times branch to the label */
        bne _btn_mode_loop

        /* Moves the GPIO_PC_BASE into the r1 register */
        ldr r1, =GPIO_PC_BASE

        /* Stores the entire bitmask into the MODEL register in PC */
        str r2, [r1, #GPIO_MODEL]

        /* Set the pull-mode to pull-up */

        /* In input mode the pullup and pulldown are controlled with the out register
         * usally this is controlled with a separate register */
        ldr r2, =0xffff

        /* Enable pullup on all lines */
        str r2, [r1, #GPIO_DOUTSET]

        /* Enable button interrupt */
        /* Just load the value directly */
        ldr r4, =0x22222222

        /* Load the GPIO_BASE address into r5 */
        ldr r5, =GPIO_BASE

        /* Store the number containing the interrupt configuration into EXTIPSELL */
        str r4, [r5, #GPIO_EXTIPSELL]

        /* Enable rising and falling edge detection */
        ldr r4, =0xff

        /* Store 8 ones in both the falling andr rising edge register, the interrupt
         * the interrupt enable register */
        str r4, [r5, #GPIO_EXTIFALL]
        str r4, [r5, #GPIO_EXTIRISE]
        str r4, [r5, #GPIO_IFC]
        str r4, [r5, #GPIO_IEN]

        /* Enable NVIC */
        /* Place the interrupt number to enable on R0 and branch with link (AAPCS) */
        mov r0, #1
```

```
        bl _enable_interrupt

        /* Place the interrupt number to enable on R0 and branch with link (AAPCS) */
        mov r0, #11
        bl _enable_interrupt

        cpsie f

        /* Deepsleep after returning from interrupt handler */
        /* If oscillators are enabled they should minimize current consumption. Typic
         * the internal RC oscillator is enabled. But the internal SCLK 32 KHz uses
         * less power. There is more clocks in a MCU than the processor clock*/

        /* Writes a 1 to the following bits SEVONPEND in order or make up from WFE
         * SLEEPDEEP to signal that the PCLK can be stopped and SLEEPONEXIT to
         * sleep in ISR exit */
        mov r7, #0b10110
        ldr r8, =SCR
        str r7, [r8]
        wfi

/* For testing button without interrupt */
_button_loop:
        /*
         * The button loop reads the content of the data in register for the GPIO
         * port C. This register is manipulated to contain all the button bits in
         * the LSByte (lease significant byte) and stored to the data out register
         * on the GPIO port A. This is where the LED sits
         */
        /* Places the GPIO_PC_BASE address inside r1 */
        ldr r1, =GPIO_PC_BASE

        /* Load r2 with the register at address GPIO_PC_BASE + GPIO_DIN */
        ldr r2, [r1, #GPIO_DIN]

        /*
         * Performs a logical shift left on the content on R2 aka the value at
         * address GPIO_PC_BASE + GPIO_DIN. That means that the content is shfted
         * 8 places to the left
         */
        lsl r2, r2, #8

        /* Places the GPIO_PA_BASE address inside r1 */
        ldr r1, =GPIO_PA_BASE

        /* Stores the content of R2 to the address of GPIO_PA_BASE + GPIO_DOUT */
        str r2, [r1, #GPIO_DOUT]

        /*
         * Relative unditional branch. Note that the range varies depending on
         * running in Thumb or ARM mode. If the processor runs in Thumb but supports
         * Thumb2 code the b.w can extend the branch range from 2K to 32M
         *
         * This statement branches to the _button_loop
         */
        b _button_loop

/* Takes in NVIC interrupt number in R0 */
```

```
_enable_interrupt:
        /*
         * This is a function for enabling interrupt thought the Cortex NVIC
         * peripheral on the processor. According to the AAPCS or ARM architecture
         * procedure calling standard the registers used for functions calls are
         * normally the R0-R3. This means that in order to call a function the
         * arguments is placed in these registers (and on stack). This is much more
         * interesting when looking at AArch64.
         */

        /*
         * Push is a alias for stmdb store-multiple-decrement before or sdmfd store
         * multiple fully descending. These uses the stack pointer as an argument
         * stmdb sp! and and optional ! in order to update the sp after the decrement
         * This comes from the nature of the Cortex fully descending stack. It does
         * not matter here, but on more complex processors you have multiple stack
         * pointers and then this is useful.
         */
        push {r1-r3}

        /* Load the address of NVIC_ISER into r2 */
        ldr r2, =NVIC_ISER

        /* Move the number 32 into r1 */
        mov r1, #32

        udiv r3, r0, r1  /* Unsigned divide, r3 = r0/r1 */

        /* Takes the content of R3 and multiply by 4 */
        lsl r3, r3, #2

        /* Performs a and operation between #31 which is 32 - 1 and r0 */
        and r0, r0, #31 /* r0 modulo 32 */

        /* Moves the number 1 into r1 */
        mov r1, #1

        /* Takes the content of r1 aka 1 and bitshift it r0 places to the left */
        lsl r1, r1, r0

        /* Store the manipulated bitmask back into its register NVIC_ISR + register o
        str r1, [r2, r3]

        /* Pop the used registers from stack, ldmia sp! or ldmfd sp! can also be used
        pop {r1-r3}

        /*
         * Since this funcitons is called by a bl label the processor executes a
         * branch with link. This means that the PC is places into the LR and then
         * the branch is performed. This is right but not intuitive. The cortex
         * processor PC acctually points 4 bytes ahead in ARM mode and 2 bytes ahead
         * in Thumb mode. Therefore by storing the lr the subroutine will jump too
         * the NEXT intstruciton. This is the case for branches but NOT for interrupt
         * On bigger processors (and the NVIC) must therefore adjust the LR
         * (acctually the stack pointer) by -4 in ARM mode and -2 in Thumb mode in
         * order to be able to return the right place. In case of a fault exception
         * this adjustment is 8 bytes in ARM mode because of the pipeline
         */
```

```
        bx lr

/* Turn on LED based on index in R0 */
_led_on:
        /* Look here for the push operation _enable_interrupt */
        push {r1-r2}

        /* Moved number 1 to register r1 */
        mov r1, #1

        /* Adds 8 to the value in R0, aka whatever you called the function with */
        add r0, r0, #8

        /* Moved the bit in r1 the number of places indicated by the pin number
         * pluss 8, this has to do with the port name */
        lsl r1, r1, r0

        /* Loads register r2 with the value of GPIO_PA_BASE aka the base address
         * for port A */
        ldr r2, =GPIO_PA_BASE

        /* Writes the previous generated bitmask to the OUTCLR register. This
         * operation only cares for the bits which is one */
        str r1, [r2, #GPIO_DOUTCLR]

        /* Pop the used registers from stack, ldmia sp! or ldmfd sp! can also be used
        pop {r1-r2}

        /* Look here for the bx lr operation _enable_interrupt */
        bx lr

/* Turn off LED based on index in R0 */
_led_off:
        /* Look here for the push operation _enable_interrupt */
        push {r1-r2}
        mov r1, #1
        add r0, r0, #8
        lsl r1, r1, r0
        ldr r2, =GPIO_PA_BASE
        str r1, [r2, #GPIO_DOUTSET]
        pop {r1-r2}
        bx lr

/*
 * Standard interrupt handling. CPU saves the stack frame consisting
 * of R0-R3, R12, LR, PC, PSR. The Cortex-M has this feature included unlike
 * other processors where this must be done manually. Then, in order to use
 * nesting like this, the LR_irq and CPSR_irq must also be push to the correct
 * stack to avoid register corruption
 */
.thumb_func
gpio_handler:
        /*
         * LDR {type}{cond} Rt, [Rn, #offset] load with immediate offset. This can
         * be used with either pre-indexed or post-indexed addressing. For eksempel
         * to load the address BARE = 56 into register R0, one can use
         * ldr r0, =GPIO_BASE. Note however that different assemblers use different
         * syntax here, and the big difference between .s and .S files.
```

```
         */
        ldr r0, =GPIO_BASE
        /* Load r1 with the register at address GPIO_BASE + GPIO_IF */
        ldr r1, [r0, #GPIO_IF]

        /* Stores the value of r1 aka the interrupt status register into the
         * interrupt clear register in order to clear the flag */
        str r1, [r0, #GPIO_IFC]

        /* Reads the gpio button and turns the respective button on */
        ldr r1, =GPIO_PC_BASE
        ldr r2, [r1, #GPIO_DIN] /* Read the DIN values [0:7] */
        lsl r2, r2, #8 /* Button data in port -> moved to [8:15] */

        /* Loads the base address of the GPIO_PA_BASE into r2 */
        ldr r1, =GPIO_PA_BASE

        /* Inverts the second operand and stored the result in r1 */
        //mvn r1, r1

        /* Stores the inverted value in the DOUT register */
        str r2, [r1, #GPIO_DOUT]

        /* Branch to the LR which was stacked in the stack frame by the processor
         * upon entering the interrupt routine */
        bx lr

/* The cortex-M does not have a GIC or APIC interrupt controller to enable
 * spesific interrupt and add handlers. Instead all the functions addresses
 * are defined in the vector table. This dummy handler is used when an interrupt
 * is enabled but the interrupt vector is not overridden. Check out WEAK attribute
 * in C or alias in assembly */
.thumb_func
dummy_handler:
        b .
```