Lab 1

ET014G Programming of Embedded Systems

Erlend Røed Myklebust

Spring 2016

Mittuniversitetet
MID SWEDEN UNIVERSITY

NTNU – Trondheim
Norwegian University of
Science and Technology

Task 1) Ports and Pins

The AVR32UC3A0512 microcontroller comes in a LQFP144 package with 144 physical connectors, of which 109 are GPIO pins.

These GPIO pins are assigned to four different groups called Ports:

- Port A with 31 pins (GPIO 0 – GPIO 30)
- Port B with 31 pins (GPIO 32 – GPIO 63)
- Port C with 6 pins (GPIO 64 – GPIO 69)
- Port X with 40 pins (GPIO 70 – GPIO 109)

On register level, these ports are rearranged to conform to the 32-bit register length of the microcontroller. They are also renamed to Port 0 – 4 in order to avoid confusion. The following formulas are used to determine the register level GPIO Port and GPIO Pin numbers for a given pin:

$$GPIO\ Port = floor\left(\frac{GPIO\ number}{32}\right)$$

$$GPIO\ Pin = GPIO\ number\ \%\ 32 \quad (modulo)$$

By this logic, GPIO 61, or PB29, corresponds to GPIO Port 1 and GPIO Pin 29.
The pin port and the gpio port appears to line up perfectly in the case of GPIO 61, but this is not always the case for other pins. For example, GPIO 95, or PX05, does not corresponds to GPIO Port 3 and GPIO Pin 5 as one might assume, but rather GPIO Port 2 and GPIO Pin 31.

Most pins have alternative functions, or peripheral multiplexing as it's referred to in the MCUs data sheet, that can be configured by manipulating the Peripheral Mux Registers.
In the case of GPIO 61 these are:

- Function A: USART2 - RXD
- Function B: PM – GCLK[1]
- Function C: EBI – NCS[2]

Task 2) Write a program that sets the CPU frequency to 12 MHZ from OSC0. The program gets value from push button0 and turn on LED6 by using polling method.

Figure 1 code shows the program in its entirety, with exception of the included ASF drivers. These are: Generic Board Support, GPIO – General-Purpose Input/Output and PM Power Manager. It was developed in Atmel Studio 6.2 with ASF v.3.29.0 and AVR32/GNU compiler and linker v.4.4.7.

```
#include <asf.h>

#define F_CPU FOSC0

int main (void)
{
        // Set Oscillator 0 (FOSC0 @ 12 MHz) as main CPU clock
        pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);

        // Initialize the EVK1100 and its pin configuration
        board_init();

        // Enable GPIO on pin PX16 (Push Button 0 on the EVK1100)
        gpio_enable_gpio_pin(GPIO_PUSH_BUTTON_0);

        // Enable GPIO on pin PB21 (LED 6 on the EVK1100)
        gpio_enable_gpio_pin(LED6_GPIO);

        // Set PB21 (LED 6 on the EVK1100) as output
        gpio_configure_pin(LED6_GPIO, GPIO_DIR_OUTPUT);

        // Main while loop
        while (1)
        {
                // If Push Button 0 is pressed (active low) ...
                if (gpio_pin_is_low(GPIO_PUSH_BUTTON_0))
                {
                        // ... activate LED 6
                        gpio_set_pin_low(LED6_GPIO);
                }

                // Otherwise ...
                else
                {
                        // ... deactivate LED 6
                        gpio_set_pin_high(LED6_GPIO);
                }
        }
}       // END
```

Figure 1. C code for Task 2 of Lab 1

Task 3) Write a C program that multiplies large numbers, measures the calculation time for each multiplication and prints the results on the LCD display. Show results for both optimization level 0 and 1.
What exactly is meant by optimization level 0 and 1 and how is it used?

This C program is developed with the following ASF drivers: CPU Cycle Counter, FLASHC – Flash Controller, Generic Board Support, GPIO – General-Purpose Input/Output, INTC – Interrupt Controller, LCD Display – DIP204B-4ORT01 and PM Power Manager.
It was developed in Atmel Studio 6.2 with ASF v.3.29.0 and AVR32/GNU compiler and linker v.4.4.7.

Figure 2 shows the main function of the program. Its intention is to initialize the necessary components and write a start string to the LCD.

Figure 3 shows the LCD initialization code that is included in the main function by the lcd_spi.h. This code enables the GPIO ports for the SPI bus and the LCD backlight, in addition to configuring the SPI.

Figure 4 shows the Interrupt handler for Push Button 0. This is where the actual multiplications and cycle count measurement takes place. It is also responsible for writing the results to the LCD.

Optimization 0 and results for –O0:

Compiling code with Optimization level 0 is essentially the same as not using optimization at all; the compiler performs no real optimization. This results in a fast compilation time and ensures an exact execution of the code, but at the cost of not reducing the program memory size and not improving execution speed.

With –O0, I achieved the following results:

- For z = x * y (z = 12345678 * 87654321): 8 clock cycles and 1 µs
- For c = a * b (c = 1234.5678 * 8765.4321): 53 clock cycles and 5 µs

Optimization 1 and results for –O1:

With Optimization level 1 enabled, the compiler attempts to reduce the program memory size and increase the execution speed. This will increase compilation time significantly, and may cause inaccuracies because the code is no longer executed exactly as it was written.

With –O1, I achieved the following results:

- For z = x * y (z = 12345678 * 87654321): 1 clock cycles and 1 µs
- For c = a * b (c = 1234.5678 * 8765.4321): 1 clock cycles and 1 µs

These results seems a bit weird. Maybe the optimization level 1 causes the compiler to pre-calculate the z and c based on that x, y, a and b are fixed values?

```
#include "asf.h"
#include "board.h"
#include "compiler.h"
#include "dip204.h"
#include "intc.h"
#include "gpio.h"
#include "pm.h"
#include "delay.h"
#include "spi.h"
#include "conf_clock.h"
#include "lcd_spi.h"
#include "stdlib.h"
#include "stdio.h"

static void push_button_0_interrupt_handler(void);

int main (void)
{
        // Set Oscillator 0 (FOSC0 @ 12 MHz) as main CPU clock
        pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);

        // Initialize the EVK1100 and its pin configuration
        board_init();

        // Initialize correct pins for the SPI
        lcd_spi_pin_init();

        // Initialize SPI MASTER, enable SPI and initialize LCD
        lcd_spi_init();

        // Disable all interrupts
        Disable_global_interrupt();

        // Initialize interrupt module
        INTC_init_interrupts();

        // Enable rising edge interrupt on Push Button 0
        gpio_enable_pin_interrupt(GPIO_PUSH_BUTTON_0, GPIO_FALLING_EDGE);

        // Define handler and configure interrupt with INT1 priority
        INTC_register_interrupt(&push_button_0_interrupt_handler,
                                AVR32_GPIO_IRQ_0 + (GPIO_PUSH_BUTTON_0/8),
                                AVR32_INTC_INT1);

        // Enable global interrupts
        Enable_global_interrupt();

        // Write start string to LCD display
        dip204_set_cursor_position(2,2);
        dip204_write_string("Press PB0 to start");
        dip204_set_cursor_position(2,3);
        dip204_write_string("Use PB0 to scroll");

        // Main while loop
        while (1)
        {
                // Do nothing
        }
} // END
```

Figure 2. Main function of Task 3 C code

```c
#ifndef LCD_SPI_H_
#define LCD_SPI_H_

void lcd_spi_pin_init(void);
void lcd_spi_init(void);


// The DIP204B is connected to the SPI1 channel on the EVK1100 as opposed to the SPI0 channel
// (which is mostly for the optional external SPI connector), and must be configured in
// accordance with this fact.

spi_options_t spiOptions =
{
         .reg          = DIP204_SPI_NPCS,     // Set SPI channel, in this case it's 1 (INT1, not INT0)
         .baudrate     = 1000000,             // Set desired baud rate
         .bits         = 8,                   // Set data character length
         .spck_delay   = 0,                   // Set delay for first clock after slave select
         .trans_delay  = 0,                   // Set delay between each transfer/character
         .stay_act     = 1,                   // Set chip to stay active after last transfer
         .spi_mode     = SPI_MODE_0,          // Select SPI mode
         .modfdis      = 1                    // Disable mode fault detection
};

void lcd_spi_pin_init(void)
{
         // The DIP204B LCD display on the EVK1100 board is connected on the following PINs:
         // PA15 (MUX function B: SPI1_CLK) = LCD Pin 6 (CLK)
         // PA16 (MUX function B: SPI1_MOSI) = LCD Pin 5 (MOSI)
         // PA17 (MUX function B: SPI1_MISO) = LCD Pin 7 (MISO)
         // PA19 (MUX function B: SPI1_CS2) = LCD Pin 4 (CS)
         // PB18 (MUX function B: PWM_6) = LCD pin 18 (Backlight)

         // Disabling the GPIO seems a bit counter-intuitive, but it's not to be used as
         // General Purpoise IO (GPIO) anymore, it's now supposed to be driven by the clock
         // or the SPI.

         AVR32_GPIO.port[0].gperc    = 1<<15; // Disable GPIO on PA15 (LCD CLK)
         AVR32_GPIO.port[0].pmr1c    = 1<<15; // Clear PMR1
         AVR32_GPIO.port[0].pmr0s    = 1<<15; // Set PMR0 = MUX mode B

         AVR32_GPIO.port[0].gperc    = 1<<16; // Disable GPIO on PA16 (LCD MOSI)
         AVR32_GPIO.port[0].pmr1c    = 1<<16; // Clear PMR1
         AVR32_GPIO.port[0].pmr0s    = 1<<16; // Set PMR0 = MUX mode B

         AVR32_GPIO.port[0].gperc    = 1<<17; // Disable GPIO on PA17 (LCD MISO)
         AVR32_GPIO.port[0].pmr1c    = 1<<17; // Clear PMR1
         AVR32_GPIO.port[0].pmr0s    = 1<<17; // Set PMR0 = MUX mode B

         // The DIP204B is connected to the SPI1 channel on the EVK1100 along with the SD/MMC
         // card slot, and is selected as slave by using the second chip select line PA19 (SPI1_CS2).
         // PA18 (SPI1_CS1) is used to select the SD/MMC card slot as slave.

         AVR32_GPIO.port[0].gperc    = 1<<19; // Disable GPIO on PA19 (LCD CS2)
         AVR32_GPIO.port[0].pmr1c    = 1<<19; // Clear PMR1
         AVR32_GPIO.port[0].pmr0s    = 1<<19; // Set PMR0 = MUX mode B
}

void lcd_spi_init(void)
{
         spi_initMaster(DIP204_SPI, &spiOptions);          // Initialize the AVR32 as SPI MASTER
         spi_selectionMode(DIP204_SPI, 0, 0, 0);           // Set selection mode
         spi_enable(DIP204_SPI);                           // Enable SPI
         spi_setupChipReg(DIP204_SPI, &spiOptions, FOSC0); // Configure registers on MASTER
         dip204_init(backlight_IO, true);                  // Initialize LCD
         dip204_hide_cursor();                             // Hide cursor
}

#endif /* LCD_SPI_H_ */
```

Figure 3. LCD initialization code

```c
uint8_t global_counter = 0;

#if __GNUC__
__attribute__((__interrupt__))
#elif __ICCAVR32__
__interrupt
#endif
static void push_button_0_interrupt_handler(void){
         if (gpio_get_pin_interrupt_flag(GPIO_PUSH_BUTTON_0)){
                  float             c                           = 0;
                  float             a                           = 1234.5678;
                  float             b                           = 8765.4321;
                  uint64_t          z                           = 0;
                  uint32_t          x                           = 12345678;
                  uint32_t          y                           = 87654321;
                  uint32_t          cycle_count                 = 0;
                  uint32_t          cycle_in_ms                 = 0;
                  char              cycle_count_string[9];
                  char              cycle_in_ms_string[9];

                  dip204_clear_display();

                  switch (global_counter){
                  case 0 :
                           // Get the current cycle count
                           cycle_count = Get_sys_count();
                           // Calculate x * y
                           z = x * y;
                           // Put cycle count difference in cycle_count
                           cycle_count = ((Get_sys_count()) - cycle_count);
                           // Convert cycle count to milliseconds
                           cycle_in_ms = cpu_cy_2_us(cycle_count, F_CPU);
                           // Convert results to strings
                           sprintf(cycle_count_string, "%lu", cycle_count);
                           sprintf(cycle_in_ms_string, "%lu", cycle_in_ms);
                           // Write results to LCD Display
                           dip204_set_cursor_position(4,1);
                           dip204_write_string("Case: z = x * y");
                           dip204_set_cursor_position(4,3);
                           dip204_write_string("Cycles: ");
                           dip204_write_string(cycle_count_string);
                           dip204_set_cursor_position(4,4);
                           dip204_write_string("In us:  ");
                           dip204_write_string(cycle_in_ms_string);
                           // Increment global counter to 1
                           global_counter = 1;
                  break;

                  case 1 :
                           // Get the current cycle count
                           cycle_count = Get_sys_count();
                           // Calculate a * b
                           c = a * b;
                           // Put cycle count difference in cycle_count
                           cycle_count = ((Get_sys_count()) - cycle_count);
                           // Convert cycle count to milliseconds
                           cycle_in_ms = cpu_cy_2_us(cycle_count, F_CPU);
                           // Convert results to strings
                           sprintf(cycle_count_string, "%lu", cycle_count);
                           sprintf(cycle_in_ms_string, "%lu", cycle_in_ms);
                           // Write results to LCD Display
                           dip204_set_cursor_position(4,1);
                           dip204_write_string("Case: c = a * b");
                           dip204_set_cursor_position(4,3);
                           dip204_write_string("Cycles: ");
                           dip204_write_string(cycle_count_string);
                           dip204_set_cursor_position(4,4);
                           dip204_write_string("In us:  ");
                           dip204_write_string(cycle_in_ms_string);
                           // Reset global counter
                           global_counter = 0;
                  break;
                  }

                  // Clear interrupt flag to allow new interrupts
                  gpio_clear_pin_interrupt_flag(GPIO_PUSH_BUTTON_0);
         }
}
```

Figure 4. Interrupt handler code

END