

Lab 2

ET014G Programming of Embedded Systems

Erlend Røed Myklebust

Spring 2016

Task 1) Write a program with the following function; When nothing is pressed, LED1 is on and a corresponding status message of LED1 is displayed on the terminal. When a push button1 is pressed, LED1 is off, LED2 is on and status message of LED2 is displayed on the terminal. Use interrupts and USART.

The RS232 serial port is vastly outdated by now, and virtually no modern PC still include one. This complicates matters when trying to use USART to communicate with the EVK1100, because its hardware is designed to only allow USART over the two RS232 ports. Thus, an active interface cable is necessary to convert from RS232 to USB.

Normally, this should not present much of a problem, but I quickly realized that ‘normally’ is a relative term. Over the course of several days, I tried different configurations of cable driver versions, USB ports, baud rate configurations and firmware functions, but there were still persistent glitches in the USART communication. Specifically, when trying to transmit characters from the PC terminal to the EVK1100, nothing were received on the RS232 connector of the EVK1100, and because most of the firmware read functions include while-loops, the program execution got stuck. Writing characters from the EVK1100 to the PC terminal, however, seemed to be working most of the time, but every once in a while a character was lost or replaced by gibberish.

In the end, the problems were found to be caused by a bad interface cable. Once the cable was replaced, the USART communication worked as intended, both RX and TX, and I was able to advance with the task.

However, because of this hiccup, I ended up with an additional solution to the task. Whilst debugging the broken interface cable, I developed an alternative code, using USB CDC instead of USART. I will not go into details about this version, since CDC is not part of the task, but I have included it in the GitHub repository (Appendix A);

Now on to the USART solution:

The project name for this solution is Lab_2_Task_1_USART. 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: *Delay routines*, *Generic board support*, *GPIO*, *INTC*, *System Clock Control* and *USART serial interface*. See Appendix A for the full code.

The program begins by configuring the CPU clock to 48 MHz (done in `conf_clock.h`), initializing the board and enabling GPIO for the LEDs. Then the USART GPIO and USART options are defined and the `usart_init_rs232()`-function is called (Figure 1). This enables USART communication on the `UART_1` port on the EVK1100. Figure 2 details the configuration and enabling of the push button interrupt. Finally, we get to the meat of the program – the main loop (Figure 4) and the interrupt routine (Figure 3). In essence, their combined functionality is an interrupt driven state machine that switches between `STATE_1` (LED1 = ON, LED2 = OFF) and `STATE_2` (LED1 = OFF, LED2 = ON), where `STATE_1` is the initial state. Following each interrupt, the state is changed and a debug string is written to USART. The `run_once` variable is used to ensure that the same state is not reapplied for every loop cycle. Figure 5 depicts an example USART output on the PC terminal.

```

// Define USART GPIO pin map
static const gpio_map_t USART_GPIO_MAP =
{
    {USART_RXD_PIN, USART_RXD_FUNCTION},
    {USART_TXD_PIN, USART_TXD_FUNCTION}
};

// Define USART options
static usart_options_t usart_options =
{
    .baudrate          = 9600,
    .charlength        = 8,
    .paritytype        = USART_NO_PARITY,
    .stopbits          = USART_1_STOPBIT,
    .channelmode       = USART_NORMAL_CHMODE
};

// Assign GPIO
gpio_enable_module(    USART_GPIO_MAP,
sizeof(USART_GPIO_MAP) / sizeof(USART_GPIO_MAP[0]) );

// Initialize USART
usart_init_rs232(USART, &usart_options, sysclk_get_pba_hz());

```

Figure 1. USART GPIO, options and initialization

```

// Initialize interrupt module
INTC_init_interrupts();

// Define handler and configure interrupt with INT1 priority
INTC_register_interrupt(&push_button_interrupt_handler,
    AVR32_GPIO_IRQ_0 + (GPIO_PUSH_BUTTON_1/8),
    AVR32_INTC_INT1);

// Enable falling edge interrupt on Push Button 1
gpio_enable_pin_interrupt(GPIO_PUSH_BUTTON_1, GPIO_FALLING_EDGE);

// Enable global interrupts
Enable_global_interrupt();

```

Figure 2. Configuration and enabling of interrupt

```

void push_button_interrupt_handler(void)
{
    if (gpio_get_pin_interrupt_flag(GPIO_PUSH_BUTTON_1))
    {
        // Switch to next state based on current state
        switch (state_indicator)
        {
            case STATE_1:
                state_indicator = STATE_2;
                break;
            case STATE_2:
                state_indicator = STATE_1;
                break;
        }

        // Set run_once variable to true
        run_once = TRUE;

        // Clear interrupt flag to allow new interrupts
        gpio_clear_pin_interrupt_flag(GPIO_PUSH_BUTTON_1);
    }
}

```

Figure 3. Interrupt routine

```

while (1)
{
    // If an interrupt has happened and run_once is true...
    if (run_once)
    {
        switch (state_indicator)
        {
            // ... and if current state is STATE_1...
            case STATE_1:
                // Activate LED0 and deactivate LED1
                gpio_set_pin_low(LED0_GPIO);
                gpio_set_pin_high(LED1_GPIO);
                // Send debug message over USART
                usart_write_line(USART,"-----\r\n");
                usart_write_line(USART,"Interrupt detected on PB1\r\n");
                usart_write_line(USART,"STATE_1 engaged!\r\n");
                usart_write_line(USART,"LED1 = ON\r\n");
                usart_write_line(USART,"LED2 = OFF\r\n");
                usart_write_line(USART,"-----\r\n");
                break;

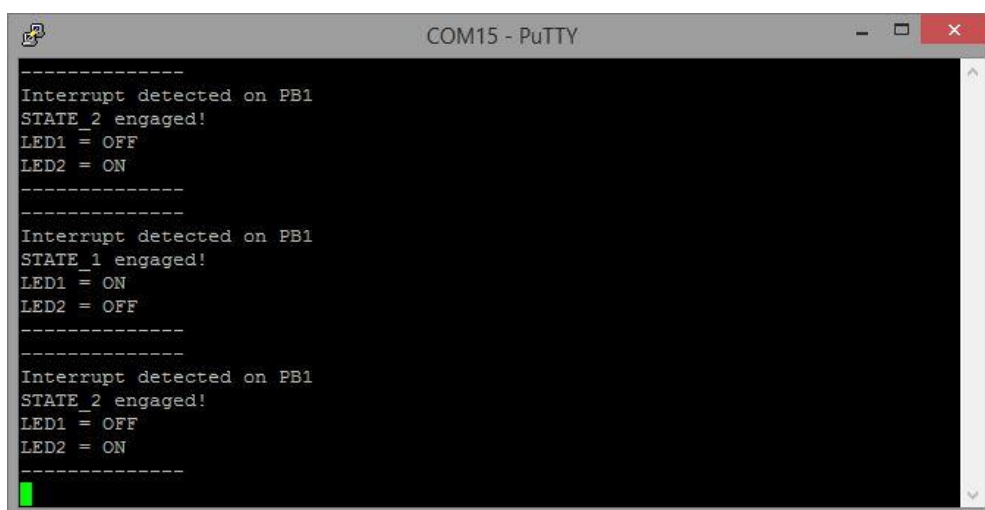
            // ... and if current state is STATE_2...
            case STATE_2:
                // Activate LED1 and deactivate LED0
                gpio_set_pin_low(LED1_GPIO);
                gpio_set_pin_high(LED0_GPIO);
                // Send debug message over USART
                usart_write_line(USART,"-----\r\n");
                usart_write_line(USART,"Interrupt detected on PB1\r\n");
                usart_write_line(USART,"STATE_2 engaged!\r\n");
                usart_write_line(USART,"LED1 = OFF\r\n");
                usart_write_line(USART,"LED2 = ON\r\n");
                usart_write_line(USART,"-----\r\n");
                break;
        }

        // Reset run_once to false
        run_once = FALSE;
    }

    // Otherwise, do nothing!
}

```

Figure 4. Main while loop



```

COM15 - PuTTY
-----
Interrupt detected on PB1
STATE_2 engaged!
LED1 = OFF
LED2 = ON
-----
Interrupt detected on PB1
STATE_1 engaged!
LED1 = ON
LED2 = OFF
-----
Interrupt detected on PB1
STATE_2 engaged!
LED1 = OFF
LED2 = ON
-----

```

Figure 5. USART output on PC terminal

Task 2) Write a program that fills the SDRAM with a repeating test pattern: 0x00, 0x01 ... 0xFF. Then use PDCA to write the entire content of SDRAM to a 2GB SD/MMC:

Fully solving this task was a very time consuming and somewhat frustrating ordeal. Even though I knew very well what I had to do, and how to structure the code, the API and its abysmal description regarding the SD/MMC and PDCA prevented me from actually writing it successfully. Furthermore, some of the key functions needed to open and close a PDCA write session to the SD/MMC were missing in ASF v.3.29.0 and had to be added manually from an older version. I was initially not aware of this, and spent a lot of time trying to use the included ASF functions to no avail. However, after learning about the missing functions and after a lot of searching and interacting on various online forums, I was eventually able to piece together the correct function sequence required to write a single sector of the SD/MMC using PDCA. The rest followed quickly.

The project name for this solution is Lab_2_Task_2_Complete. 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: *Generic board support, GPIO, PDCA, PM Power Manager, SD/MMC SPI card access, SDRAM, USART Debug*. See Appendix A for the full code.

Note: additional functions for the *SD/MMC SPI card access* was added to make it fully compatible with the *PDCA* driver. Said functions are included in *extra_pdca_functions.h*.

The program starts by configuring the CPU clock to 60 MHz and the PBA clock to 30 MHz, initializing the USART debug on the UART_1 port, initializing the board and calling the *sd_mmc_resources_init()* – function. This locally defined function configures the SPI GPIO, defines the SPI options and initialize the SPI module with PBA clock speed. See Figure 6. Next, the SDRAM size is determined, the HSB default master is set and the SDRAM is initialized with CPU clock speed. Figure 8 depicts the loop responsible for filling the SDRAM with the test pattern. After the SDRAM loop is done, a function called *local_pdca_init()* is called. See Figure 7. This locally defined function configures and initializes the RX and TX channels for the PDCA. Note that only the TX channel is used in this program, but I have still left in the RX channel configuration and initialization because I intend to use this code for future reference. Moving on with the program, Figure 9 presents the loop responsible for writing the contents of SDRAM to the SD/MMC. Specifically, it starts out by determining how many 512 Byte sectors is needed to fit the entire 32 MB SDRAM (16384 sectors), and then begins looping through, starting with sector 0 and ending with sector 16383. For each sector, an SPI PDCA writing session is opened, the PDCA channel is loaded with 512 Bytes of the SDRAM contents and then the transfer is enabled. After waiting for the transmission to end, the PDCA is disabled and the write session is closed. A progress indicator is added to keep the user from unwittingly aborting the loop. This also makes it easy to spot any potential breakdowns in the code progression.

Figure 10 depicts an example of the USART debug after completing a run-through of the program, while Figure 11 and Figure 12 presents the 16383rd sector of the SD/MMC before and after respectively.

```

static void sd_mmc_resources_init(void)
{
    // GPIO pins used for SD/MMC interface
    static const gpio_map_t SD_MMC_SPI_GPIO_MAP ={
        {SD_MMC_SPI_SCK_PIN, SD_MMC_SPI_SCK_FUNCTION },
        {SD_MMC_SPI_MISO_PIN, SD_MMC_SPI_MISO_FUNCTION},
        {SD_MMC_SPI_MOSI_PIN, SD_MMC_SPI_MOSI_FUNCTION},
        {SD_MMC_SPI_NPCS_PIN, SD_MMC_SPI_NPCS_FUNCTION}};

    // SPI options.
    spi_options_t spiOptions ={
        .reg          = SD_MMC_SPI_NPCS,
        .baudrate     = PBA_HZ,
        .bits         = SD_MMC_SPI_BITS,
        .spck_delay   = 0,
        .trans_delay  = 0,
        .stay_act     = 1,
        .spi_mode     = 0,
        .modfdis      = 1};

    // Assign I/Os to SPI.
    gpio_enable_module(SD_MMC_SPI_GPIO_MAP,
        sizeof(SD_MMC_SPI_GPIO_MAP) / sizeof(SD_MMC_SPI_GPIO_MAP[0]));

    // Initialize as master.
    spi_initMaster(SD_MMC_SPI, &spiOptions);

    // Set SPI selection mode: variable_ps, pcs_decode, delay.
    spi_selectionMode(SD_MMC_SPI, 0, 0, 0);

    // Enable SPI module.
    spi_enable(SD_MMC_SPI);

    // Initialize SD/MMC driver with SPI clock (PBA).
    sd_mmc_spi_init(spiOptions, PBA_HZ);
}

```

Figure 6. Function for configuring and initializing the SPI

```

static void local_pdca_init(void)
{
    // Config options for the RX channel
    pdca_channel_options_t pdca_options_SPI_RX ={
        .addr = ram_buffer,
        .size = 512,
        .r_addr = NULL,
        .r_size = 0,
        .pid = AVR32_PDCA_CHANNEL_USED_RX,
        .transfer_size = PDCA_TRANSFER_SIZE_BYTE};

    // Config options for the TX channel
    pdca_channel_options_t pdca_options_SPI_TX ={
        .addr = sdram,
        .size = 512,
        .r_addr = NULL,
        .r_size = 0,
        .pid = AVR32_PDCA_CHANNEL_USED_TX,
        .transfer_size = PDCA_TRANSFER_SIZE_WORD};

    // Init PDCA transmission channel
    pdca_init_channel(AVR32_PDCA_CHANNEL_SPI_TX, &pdca_options_SPI_TX);

    // Init PDCA Reception channel
    pdca_init_channel(AVR32_PDCA_CHANNEL_SPI_RX, &pdca_options_SPI_RX);
}

```

Figure 7. Function for configuring and initializing the PDCA

```

// Determine the increment of steps for progress indicator
progress_inc = (sdram_size + 50) / 100;

// Fill SDRAM with test pattern: 0x00, 0x01, ... 0xFF
for (i = 0, j = 0, k = 0; i < sdram_size; i++)
{
    // Write progress indicator to USART
    if (i == k * progress_inc)
    {
        print_dbg("\rFilling SDRAM with test pattern: ");
        print_dbg_ulong(k++);
        print_dbg_char('%');
    }

    // Fill SDRAM byte i with value j, and then increment j
    sdram[i] = j;
    j++;

    // Debugger for ensuring the pattern
    if (j > 0xFF)
    {
        j = 0;
    }
}

```

Figure 8. Loop for filling the SDRAM with the test pattern

```

// Calculate amount of SD/MMC sectors needed to fit entire SDRAM (512 B per sector)
number_of_sd_sectors = sdram_size / 512;

// Determine the increment of steps for progress indicator
progress_inc = number_of_sd_sectors / 100;

// Loop for writing entire SDRAM to SD/MMC one sector at a time
for (i = 0, j = 0; i < number_of_sd_sectors; i++)
{
    // Open PCDA write session to SD/MMC sector i
    if (sd_mmc_spi_write_open_PDCA(i))
    {
        // Load contents of SDRAM on the SPI_TX channel
        pdca_load_channel( AVR32_PDCA_CHANNEL_SPI_TX, sdram, 512);

        // Enable PDCA
        pdca_enable(AVR32_PDCA_CHANNEL_SPI_TX);

        // Wait for transmission to end
        while(!(pdca_get_transfer_status(AVR32_PDCA_CHANNEL_SPI_TX)&2));

        // Disable PDCA
        pdca_disable(AVR32_PDCA_CHANNEL_SPI_TX);

        // Close PCDA write session
        sd_mmc_spi_write_close_PDCA();
    }

    // Write progress indicator to USART
    if (i == j * progress_inc)
    {
        print_dbg("\rTransferring SDRAM to SD/MMC: ");
        print_dbg_ulong(j++);
        print_dbg_char('%');
    }
}

```

Figure 9. Loop for writing the contents of SDRAM to the SD/MMC

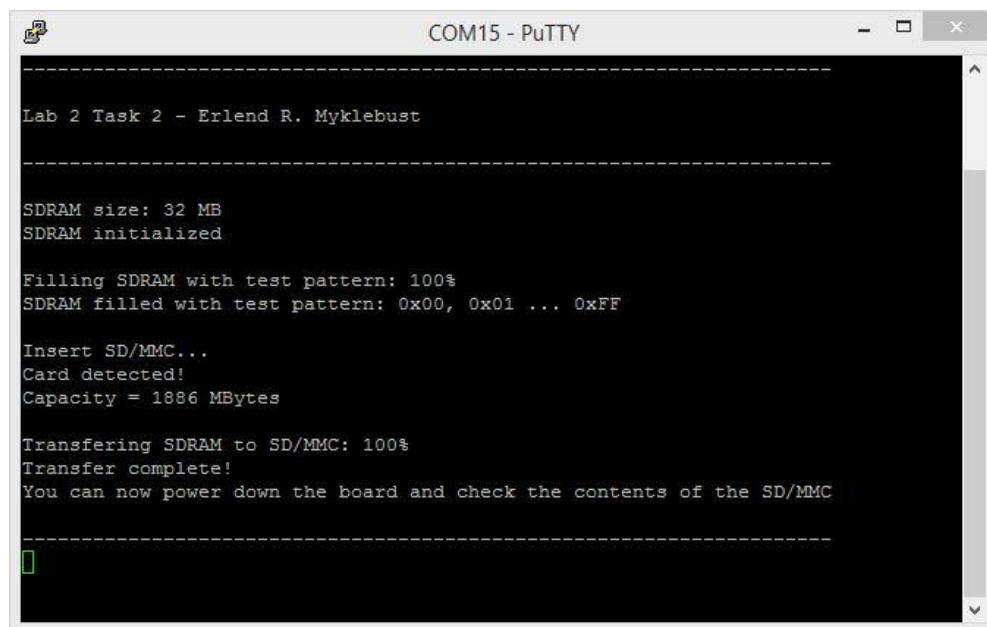


Figure 10. USART debug example

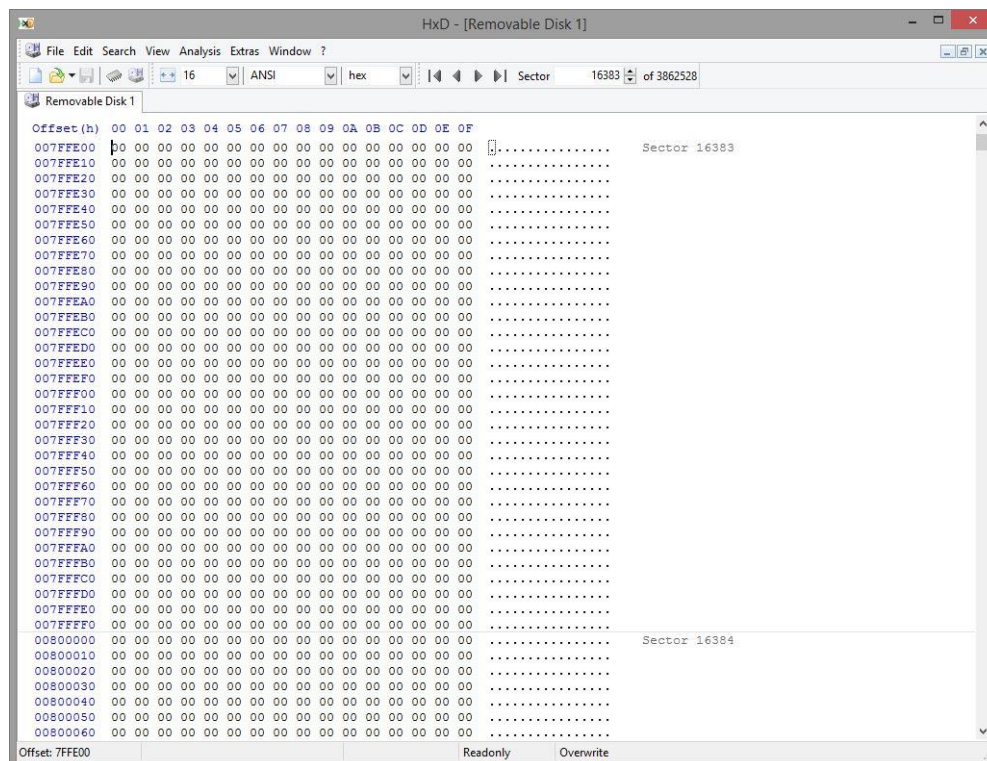
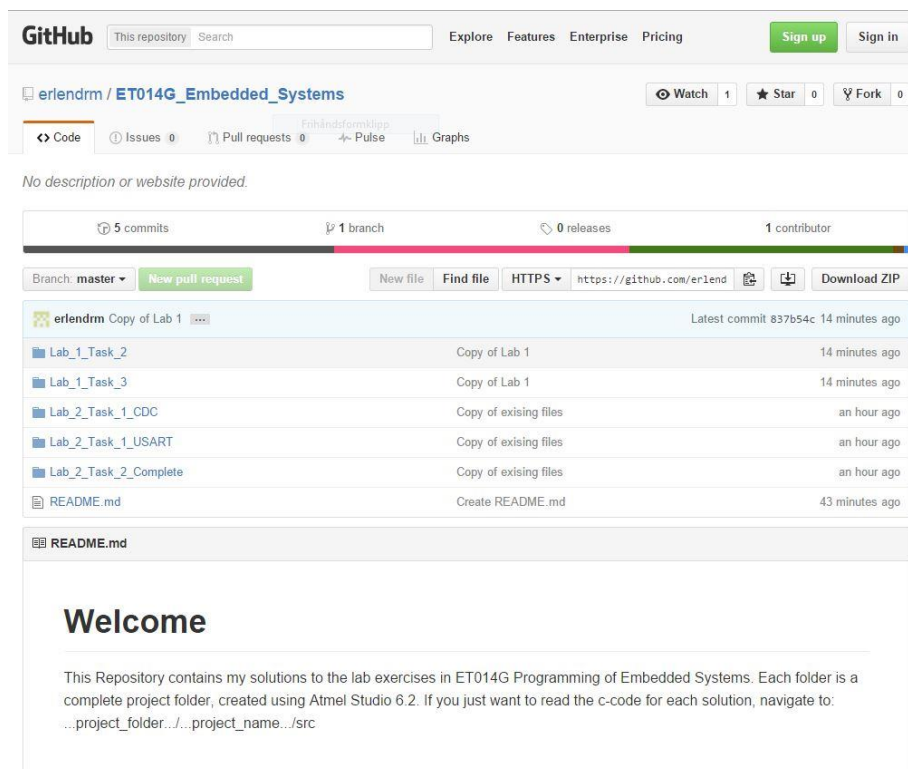


Figure 11. Contents of the SD/MMC before running the program



Page 8

Appendix A: Code Repository (GitHub)



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 main c-files for each solution, simply navigate to *Lab_2_Task_1_USART/Lab_2_Task_1_USART/src/* for task 1 and *Lab_2_Task_2_Complete/Lab_2_Task_2_Complete/src/* for task 2 from the main repository. There you will find the *main.c* for each task. Note that in task 2, there is an additional .h-file called *extra_pdca_functions.h*.

Please use the following static link to reach the repository:

github.com/erlendrm/ET014G_Embedded_Systems