Exercise 0 - Getting Ready

In this course we will be working with the STM32F302R8 32-bit ARM microcontroller embedded on an STM Nucleo64 development board. Pin-out diagrams for the development board as well as the datasheet and reference manual for the microcontroller are available on DTU Inside. The STM32 is connected to an mbed expansion board which contains a number of peripherals we will be playing around with later on.

Before starting the exercises, the following must be completed:

1. Install PuTTY 0.7.
2. Install EmBitz 1.11.
3. Install the USB Driver (run `stlink_winusb_install.bat`)
4. Connect the Nucleo STM32F302R8 board to your PC through a mini-USB cable.

PuTTY will be used for serial communication with the board, while programming and debugging will be done using EmBitz. Both pieces of software are available on DTU Inside in the course folder. The versions we provide might not be completely up to date, but it simplifies bug-fixing if all students use the same version which is known to work.

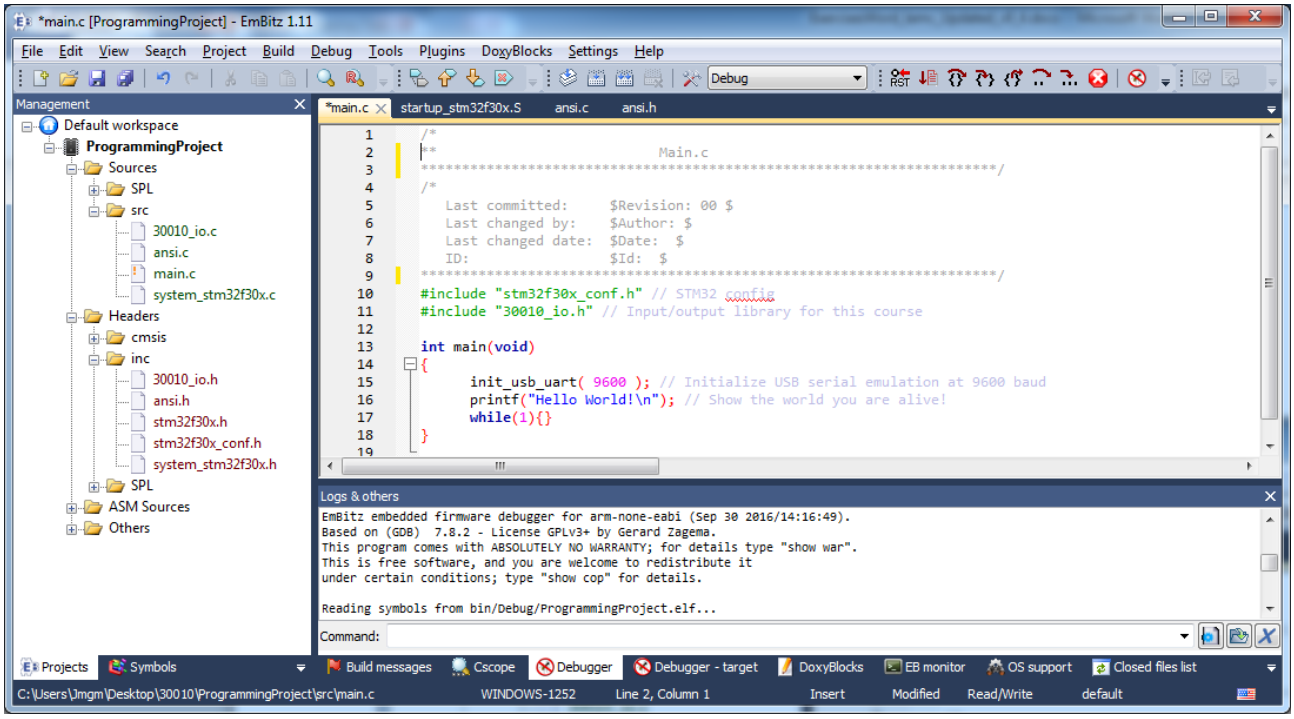
Exercise 0.1 - Getting Started with the STM32

With everything ready we will begin by setting up a project for the STM32 chip. Note that this guide has been written for EmBitz 1.11, your mileage may vary if using other versions.

- Begin by starting up EmBitz and click Select File - New - Project...
- Select the `STmicro-ARM` template and click Go. Click Next.
- Choose a project title and file path to your liking. Click Next.
- Make sure the compiler is set to ARM GCC Compiler (EmBitz - bare-metal) and click Next with everything else as default.
- Select `Cortex M4 (F3xx - F4xx)` and click Next
- Select `STM32f30x (F30x - F31x)` and click Next
- Choose `STM32F302R8` for the processor, leave everything else as default and click Finish
- Click OK on the two pup-up windows (debug interface options and ST-link settings) to close them.

Note: In some cases, it will be necessary to use a different debugger, otherwise the debugging functionality will not work. Please follow the steps below to setup Embitz with OpenOCD x64:

- Download and unpack OpenOCD 0.10.0 to a desired location (eg. C:\OpenOCD)
- Go to emBitz -> Debug -> Interfaces and select the GDB Server tab.
- Set Selected Interface to openOCD and click Settings >>
- Configure OpenOCD using the following parameters:
    - **Board:** leave empty, **Interface:** stlink-v2-1**, Target:** stm32f3x_stlink, **JTAG Speed (KHz):** 4
    - **#HW breakpoints**: 1
- Click OK to go back.
- Click the Browse button and navigate to the OpenOCD 0.10.0 folder (eg. C:\OpenOCD)
- Go into the bin-x64 folder and select the openocd.exe.



The project has now been created and we are ready to create our first program. In the sidebar to the left you should now see a list of folders pertaining to your project as shown in the figure above. If this is not the case, make sure that the `Projects` tab is selected instead of the `Symbols` tab in the bottom left. At the moment we are interested in the folder called Sources which contains two sub-folders called `SPL` and `src`. `SPL` holds the source files for a hardware library that will make things a bit easier down the line, but we can ignore it for now. The `src` folder contains the source files for our project and this is where we will be spending most of our time. At the moment `src` holds two auto-generated files: `system_stm32f32x.c` which is used for chip configuration, and `main.c` which will hold our initial program.

At the moment, all our program does is to load a header file containing some configuration parameters and start an endless loop. Let's change that!

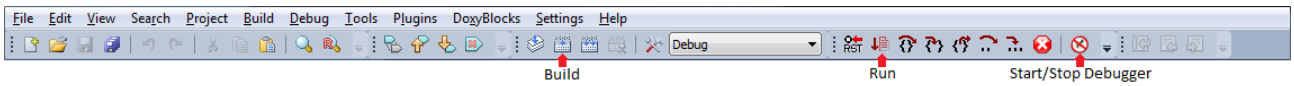
- Open `main.c` and add the following code snippet:

```
#include "stm32f30x_conf.h" // STM32 config
#include "30021_io.h" // Input/output library for this course

int main(void)
{
   init_usb_uart( 9600 ); // Initialize USB serial emulation at 9600 baud
   printf("Hello World!\n"); // Show the world you are alive!
   while(1){}
}
```

This first line includes `stm32f30x_conf.h` - the same configuration library as before - which will simplify future hardware interfacing. The second line includes `30021_io.h` which is a special IO library made for this course that enables us to get started quickly. Without it you would have to do a lot of work to configure the serial port before being able to communicate with the PC.

`init_usb_uart(9600)` initializes the USART2 port to 9600 baud, 8 data bits, no parity bit, and 1 stop bit. The built-in ST-LINK debugger has a USB serial emulator that is connected to USART2 thus enabling easy communication with the PC. The `uart_putc()` and `uart_getc()` functions are provided to read/write single characters via USART2. The `printf()` function normally writes to stdout but has been overwritten to write to the USART2 instead. Contrary to most programs that run on a PC, we don't want the main function to ever return. Therefore, an endless “while” loop is included.

- Build the program by pressing F7 or clicking on the Build button shown in the figure below.



This will immediately fail as `30021_io.c` and `30021_io.h` must be added to the project.

- Download `30021_io.c` and `30021_io.h` from the DTU Inside folder and copy `30021_io.c` to `<PROJECT LOCATION>/src` and `30021_io.h` to `<PROJECT LOCATION>/inc`.
- Include these files one by one by right clicking on the project in the sidebar and selecting Add Files...
- After selecting each file, a window will pop up. Make sure both Debug and Release are check marked on this window.
- Build the program once again.

Exercise 0.2 - Uploading Code to the STM32

To upload the program to the microcontroller simply press F8 or click the button marked "Start/Stop Debugger" in the previous figure. When uploading the program, the built-in debugger is launched and the program is paused at the beginning. Pressing F5 or clicking on the button marked "Run" will un-pause the program and enable debugging. Alternatively, pressing F8 stops the debugger and allows the program to run freely. Both methods work for now and we will get back to debugging shortly.

- Upload the program to the STM32.
- Start the program running on the microcontroller.

The program will now print "Hello world!" to the PC, but we can't actually see it yet. For that we need PuTTY.

- Start the Device Manager (or “Enhedshåndtering”) on your PC.
- Under Ports (COM & LPT) should be an entry called STMicroelectronics STLink Virtual COM Port (COMx) where the "x" represents a number. Take note of this number.
- Startup PuTTY and select “Serial” for the connection type.
- Write COMx in the Serial line field where "x" is the number from before and click “Open”

A blank terminal window should now pop up. If you get a message saying that the selected serial port doesn't exist, make sure you wrote down the correct number from the device manager and try again. If it still doesn't work, try closing PuTTY and unplugging the USB cable and plugging it in again.

- Press the Reset button on the STM32 development board or press F5 if the debugger is paused. Hopefully you should see "Hello World!" being written in the terminal.

Exercise 0.3 - Debugging with the STM32

The on-board debugger allows for line-by-line stepwise execution of the program while it is running on the microcontroller. The table below shows an overview of the available debugging commands which can be found in the Debug drop-down menu. Breakpoints can also be set while the debugger is not running.

Command	Description	Shortcut
Reset	Resets the program to the starting point.	
Run	Lets the program run freely.	F5
Stop	Pauses the program at whichever instruction is currently being executed.	
Next Line	Executes on line at a time.	F10
Step Into	Steps into subroutines, otherwise works as Next Line.	F11
Step Out	Executes the rest of the subroutine, steps out and stops.	Shift+F11
Next Instruction	Executes the next machine code instruction (similar to Next Line).	Ctrl+F10
Step Instruction	Steps into machine code (similar to Step Into).	Ctrl+F11
Run to Cursor Line	Executes the program until it reaches the line the cursor is currently on.	F4
Insert Breakpoint	Inserts a breakpoint at the current line. When the program is set to run it will pause every time it reaches a breakpoint.	F9
Remove All Breakpoints	Self-explanatory.	Ctrl+Shift+F9

Let's test the debugger.

- Replace the code of your main file with the following.

```
#include "stm32f30x_conf.h" // STM32 config
#include "30021_io.h" // Input/output library for this course

int8_t power(int8_t a, int8_t exp) {
// calculates a^exp
    int8_t i, r = a;
    for (i = 1; i <= exp; i++)
        r *= a;
        return(r);
}

int main(void)
    {
        int8_t a;
        init_usb_uart( 9600 ); // Initialize USB serial at 9600 baud
        printf("\n\n x x^2 x^3 x^4\n")
        for (a = 0; a < 10; a++)
            printf("%8d%8d%8d%8d\n",a, power(a, 2), power(a, 3), power(a, 4));
        while(1){}
}
```

- The above code cannot compile. Fix the syntax errors and try again.
- You should now see a table with values of $x; x^2; x^3; x^4$ for `x=1::10` printed in the terminal. But something is definitely not right! Some of the printed values are negative. Why is that?

Beyond the negative values, the numbers are also not calculated correctly as is evident from looking at 2^2, for example. Let's try using the debugger to fix the problem.

- Start the debugger by pressing F8 or stop/restart it from the debugging menu if it is already running.

- Move the cursor to the first line in the `power()` function (the line starting with `int8_t i`) and select Debug - Run to cursor line (F4).

It would be nice to know what is going on with `i` and `r`, so let's add them to a watch list.

- Select Debug - Edit watches...
- Click Add and enter the name of the first variable, i.e., `i`.
- Do the same thing for `r`.
- Close the watch editor and open the watch list by selecting Debug - Debugging Windows - Watches.
- Step through the program using F10 until you reach the line after `return(r);`.

In the Watches window you can see the values of your watches written in red when they are changed and in black otherwise.

- Press F10 again to enter the `main()` function.
- Press F10 again. This will execute the `power()` function, including all the remaining calls to `power()`.
- Reset the simulation using Debug - Reset.
- Place your cursor on the line containing `init_usb_uart(9600);` and press F4.
- Press F11 (step into). This will cause you to enter the `init_usb_uart()` function.
- Press Shift+F11 to get out of this strange place and return to the comfort of the `main()` function!
- Press F10 until the first two lines containing only zeros and ones have been written to the console.

You can use breakpoints to stop at a specific point in the code whenever it is executed. Let's do that for the `power()` function.

- Place the cursor on the line containing `for (i = 1; i <= exp; i++)` and press F9 to place a breakpoint. Alternatively, you can click on the margin just to the right of the line number.
- Press F5 to run the program until the breakpoint is reached.
- Use these debugging tools to see what happens with `i` and `r` and fix the problem!

Now you should be all set to start playing with the STM32F302R8 processor. Time to do some proper programming!

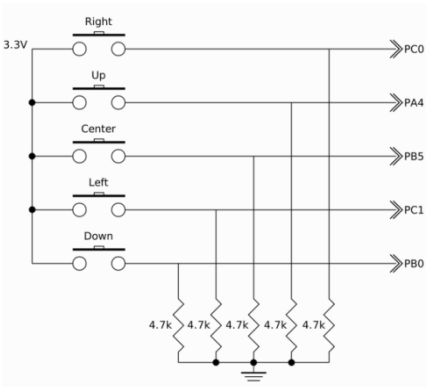
Exercise 1 – Using the General-purpose I/Os, timers, interrupts

This exercise covers the General-purpose I/Os that can be used to interface the microcontroller to the real world. You will learn how to interface the joystick (input) and the RGB LED (output).

You will need the documents ”STM32F302x8_Reference_Manual”,” STM32F302x8_Datasheet” and “STM32F30xx31xxSPL_en.DM00068049” (SPL - Standard Peripheral Library) for the next couple of exercises. These documents can be on DTU Inside and will be referred to as [RM], [DS] og [SPL], respectively. When you first open the documents, you may be a bit intimidated by the fact that [RM] is 1080(!) pages long, but do not fret - we will only need a few sections from it.

Exercise 1.1 - Detecting a Joystick Interaction

You will need the ”General-purpose I/Os (GPIO)” chapter (pp. 155) from [RM] when doing this exercise.



The STM32F302R8 microcontroller has 51 *General Purpose Input/Output* (GPIO) pins all of which are available to us on the STM32 Nucleo development board we are using. The pins are controlled through 5 I/O ports (labelled A, B, C, D, and F) with each port controlling up to 16 pins. Each GPIO pin can be used either as input or output and most of them are connected to on-chip peripheral functions such as timers or serial communication devices (see alternate functions in table 15 of [DS], pp. 45).

The joystick on the mbed expansion board is connected to pins PC0 (right), PA4 (up), PB5 (center), PC1 (left), and PB0 (down). The schematic for the connections is shown to the right. Each I/O port is controlled through a few functions given in chapter 12 “12 General-purpose I/Os (GPIO)” in [SPL p. 234].

The complete guide on how to use the GPIOs can be found on page 235 [SPL] and will be summarized here.

Before a port can be used the clock on that ports needs to be enabled. For this a function exist “*RCC_AHBPeriphClockCmd(uint32_t RCC_AHBPeriph, FunctionalState NewState)”* more info at [SPL p. 330].

The GPIO ports can do a host of different things so a setup process is needed. The manual describes this on page 235 [SPL]. Shortly said for our need this setup for input and output are given below for one pin on one port.

The following code snippet shows how to set pin PC0 to input and PA9 to output:

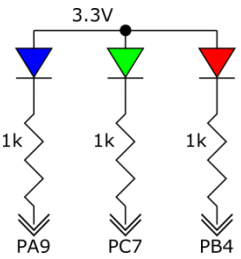
```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC,ENABLE); // Enable clock for GPIO Port C
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA,ENABLE); // Enable clock for GPIO Port A

GPIO_InitTypeDef GPIO_InitStructAll; // Define typedef struct for setting pins

GPIO_StructInit(&GPIO_InitStructAll); // Initialize GPIO struct

GPIO_InitStructAll.GPIO_Mode = GPIO_Mode_IN; // Set as input
GPIO_InitStructAll.GPIO_PuPd = GPIO_PuPd_DOWN; // Set as pull down
GPIO_InitStructAll.GPIO_Pin = GPIO_Pin_4; // Set so the configuration is on pin 4

GPIO_Init(GPIOA, &GPIO_InitStructAll); // Setup of GPIO with the settings chosen

// Sets PA9 to output
GPIO_StructInit(&GPIO_InitStructAll); // Initialize GPIO struct
GPIO_InitStructAll.GPIO_Mode = GPIO_Mode_OUT; // Set as output
GPIO_InitStructAll.GPIO_OType = GPIO_OType_PP; // Set as Push-Pull
GPIO_InitStructAll.GPIO_Pin = GPIO_Pin_9; // Set so the configuration is on pin 9
GPIO_InitStructAll.GPIO_Speed = GPIO_Speed_2MHz; // Set speed to 2 MHz
                                // For all options see SPL/inc/stm32f30x_gpio.h

GPIO_Init(GPIOA, &GPIO_InitStructAll); // Setup of GPIO with the settings chosen
```

Using the GPIO chapter from [SPL p. 234] and the above information, please do the following:

- Find out what the default state of the GPIO pins are and initialize the GPIO pins as input. Create a function for this purpose, you can call it `initJoystick()`.
- Find out if the signal lines are high or low when the joystick is pushed in a certain direction.
- Create a function, you can call it `readJoystick()`, that returns the state of the joystick. It should return 8 bits using the following format:
  bit 0: 1 if the joystick is pressed up, 0 otherwise
  bit 1: 1 if the joystick is pressed down, 0 otherwise
  bit 2: 1 if the joystick is pressed left, 0 otherwise
  bit 3: 1 if the joystick is pressed right, 0 otherwise
  bit 4: 1 if the joystick is pressed center, 0 otherwise
  bit 5-7: 0
- Write the current joystick direction to PuTTY. Note: You should only write the direction when it is changed.

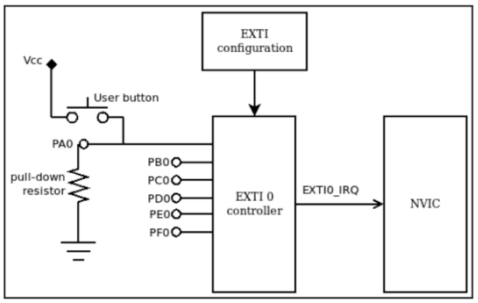
Exercise 1.2 - Controlling the RGB LED

The RGB LED is connected to the mbed expansion board as shown in the figure to the right. By selectively turning on different combinations of LEDs a total of 7 colors (red, green, blue, cyan, magenta, yellow, and white) can be created. Red is connected to PB4, green to PC7, and blue to PA9.



- Write a function that initializes the GPIO used for the LED. You can call it `initLed()`.
- Write a function that turns the LED to a specific color depending on the argument. You can call it `setLed(...)`.
- Write a program that lights up the LED in different colors depending on which direction the joystick is pressed.

Exercise 1.3 – Interrupt based input



Having to constantly poll the IDR (Independent Data Register) is not very convenient in cases, where the application must react to state changes. Instead, it is possible to map the digital inputs to interrupt lines, making it possible for the microcontroller to detect and react to input signal events without wasting CPU cycles. The interrupt (IRQ) signal will cause the microcontroller to pause its current activities and execute the interrupt routine(s) instead. 36 interrupt/event lines are available and all GPIO are mapped to the EXTI (extended interrupts and events) controller using 16 external interrupt lines. PA0-PF0 are mapped to EXTI0, PA1-PF1 are mapped to EXTI1 and so on. The full list of extended interrupts can be found on page 215-216 in [RM]. The EXTI controller have a chapter in [SPL p.207] where a guide on how to set it up and use it are described.

The full guide can be found on page 209 [SPL] and will be summarized here.

Start by enabling the clock on *RCC_APB2Periph_SYSCFG*. It provides the following registers, some of which must be set:

Register	Address Offset	Size [Bits]	Description
EXTI			Base Address
EXTI_IMR	0x00	32	Enable/Disable the interrupt for each EXTI line
EXTI_EMR	0x04	32	Enable/Disable the event trigger for each EXTI line
EXTI_RTSR	0x08	32	Set to enable rising edge triggers (0->1 transition)
EXTI_FTSR	0x0C	32	Set to enable rising edge triggers (1->0 transition)
EXTI_SWIER1	0x10	32	Set to enable interrupt service routine execution
EXTI_PR1	0x14	32	Get/Set pending interrupts

In this part of the exercise, you will enable interrupts for at least one joystick switch – eg. PB0, PC0, ...

To configure an external interrupt, it is necessary to configure both the EXTI peripheral and the Nested vectored interrupt controller (NVIC). It is assumed that the GPIO pin(s) has already been setup as input.

1. Select the input source pin for the EXTI line using *SYSCFG_EXTILineConfig().*
    a. This sets the connections between the GPIO pin and the EXTI line.
2. Select the mode (**interrupt**, event) and configure the trigger selection (**Rising**, falling or both) using *EXTI_Init().*
    a. As with the GPIO a EXTI_InitTypeDef have to be made and configured (read more on page 208 [SPL]).
3. Configure NVIC IRQ channel mapped to the EXTI line using *NVIC_Init()*. [SPL p. 284]
    a. As with the GPIO a NVIC_InitTypeDef have to be made and configured (read more on page 284 [SPL]).

Ex:

```
 // interrups
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG,ENABLE);
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB,EXTI_PinSource5); // sets port B pin 5 to the IRQ
// define and set setting for EXTI
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line = EXTI_Line5; // line 5 see [RM p. 215]
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
EXTI_Init(&EXTI_InitStructure);
// setup NVIC
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);

NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_Init(&NVIC_InitStructure);


void EXTI9_5_IRQHandler(void){
    if(EXTI_GetITStatus(EXTI_Line5) != RESET){
          printf("Right : %d  | Up : %d  |  Center : %d  |  Left : %d  |  Down : %d
          \n",GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_0),GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_4),GP
          IO_ReadInputDataBit(GPIOB,GPIO_Pin_5),GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_1),GPIO_Rea
          dInputDataBit(GPIOB,GPIO_Pin_0));
          GPIO_WriteBit(GPIOC,GPIO_Pin_7,Bit_RESET);
          EXTI_ClearITPendingBit(EXTI_Line5);
          }
}
```

Configure the EXTIXX bits in one of the four SYSCFG_EXTICRX registers (11.1.2 in [RM]) to map the GPIO pin(s) to the appropriate interrupt lines (EXTI0-EXTI15). The registers are accessed by `SYSCFG->EXTICR[0 to 3] &= …` When setting up an interrupt with the NVIC we need to fill out the interrupt function handler so the μC knows what do to when the interrupt happens.

Write your interrupt service routine (ISR) for the EXTI line used. A list of names for all the possible interrupt functions can be found in ASM Sources/src/startup_stm32f30x.S, line 233 onwards. If more than one pin is mapped to the same interrupt (eg. PB0 and PC0), check the source of the interrupt inside the ISR and act accordingly. Some EXTI lines share the same interrupt handler (eg. EXTI15_10_IRQn and EXTI9_5_IRQn). Remember to clear the pending bit for the interrupt line, by setting the EXT_PR register. For EXTI0, the ISR is as follows:

```
void EXTI0_IRQHandler(void){
//Do some important stuff here!...
EXTI_ClearITPendingBit(EXTI_Line0); //Clear the interrupt pending bit
}
```

NOTE: Remember to enable the clock for the SYSCFG module!

- For the selected interrupt line, choose a signal change that will trigger the interrupt. The signal change can be rising edge, falling edge – or both. Triggers are set using the EXTI_RTSR (rising) and EXTI_FTSR (falling) registers.
- Unmask the interrupt line(s) by setting the corresponding lines in the EXTI_IMR. The event triggers (EXTI_EMR) are not needed, as they are only used for waking up the MCU in case of an event.
- Set the priority for the interrupt vector in the NVIC and enable the interrupt in the NVIC:

```
NVIC_SetPriority(EXTI0_IRQn, priority); // Set interrupt priority
NVIC_EnableIRQ(EXTI0_IRQn); // Enable interrupt
```

- Write your interrupt service routine (ISR) for EXTI0. A list of names for all the possible interrupt functions can be found in ASM Sources/src/ startup_stm32f30x.S, line 233 onwards. If more than

one pin is mapped to the same interrupt (eg. PB0 and PC0), check the source of the interrupt inside the ISR and act accordingly. Some EXTI lines share the same interrupt handler (eg. `EXTI15_10_IRQn` and `EXTI9_5_IRQn`). Remember to clear the pending bit for the interrupt line, by setting the EXT_PR register. For EXTI0, the ISR is as follows:

```
void EXTI0_IRQHandler (void){
//Do some important stuff here!...
EXTI->PR |= EXTI_PR_PR0; //Clear the interrupt pending bit
}
```

- Confirm that the interrupt is running – eg. by turning on an LED when it is triggered.

Exercise 1.4 – Timers

In this exercise, we will be taking a look at timers. You will create a stopwatch with two split times, all of which will be shown in PuTTY:

```
Stop watch

Time since start:   0:01:04.--
Split time 1:       0:26:12.21
Split time 2:       -:--:--.--
```

For this exercise you will need to consult the "General-purpose timers (TIM2/TIM3/TIM4)" chapter (pp. 550) of [RM] and the “General-purpose timers (TIM)” chapter in [SPL p. 447]. The STM32 chip has several timers that can be used for different purposes such as PWM and general time keeping. We will be focusing on timer 2 and operating it in *up-counting mode.* In this mode, the timer counts up to a reload. When the reload value is reached an interrupt is generated and the timer starts over. The timer has a prescaler which divides the input clock (64 MHz) with (n + 1) where n = [0...65535]. If you look at the table below (or in section 21.4 of [RM]) you will see that each timer is associated with a lot of different registers, but luckily, we only need to deal with 5 of them, namely CR1, ARR, PSC, DIER, and SR.

Register	Address Offset	Size [Bits]	Description
TIM2			Port base address
TIM2->CR1	0x00	16	Primary Configuration
TIM2->CR2	0x04	32	Secondary Configuration
TIM2->SMCR	0x08	32	Slave Mode Control
TIM2->DIER	0x0C	32	DMA/Interrupt Enable
TIM2->SR	0x10	32	Status
TIM2->EGR	0x14	32	Event Generation
TIM2->CCMR[1,2]	0x18 - 0x1C	32 + 32	Compare Mode
TIM2->CCER	0x20	32	Compare Enable
TIM2->CNT	0x24	32	Counter
TIM2->PSC	0x28	16	Prescaler
TIM2->ARR	0x2C	32	Auto-reload
TIM2->CCR[1,..,4]	0x34 - 0x40	32 + 32 + 32 + 32	Capture/Compare
TIM2->DCR	0x48	16	DMA Control
TIM2->DMAR	0x4C	16	DMA Address

To configure the timer, you will have to do the following:

To use the Timer in Timing (Time base) mode, the following steps are mandatory: [p. 455 in SPL]

1. Enable TIM clock using *RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx, ENABLE)* function
2. Fill the *TIM_TimeBaseInitStruct* with the desired parameters.
3. Call *TIM_TimeBaseInit(TIMx, &TIM_TimeBaseInitStruct)* to configure the Time Base unit with the corresponding configuration

4. Enable the NVIC if you need to generate the update interrupt.
5. Enable the corresponding interrupt using the function *TIM_ITConfig(TIMx, TIM_IT_Update)*
6. Call the *TIM_Cmd(ENABLE)* function to enable the TIM counter.

What parameters the *TIM_TimeBaseInitStruct* needs will be disused below.

Eg:

```
// Timer
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);

  TIM_TimeBaseInitTypeDef TIM_InitStructure;
  TIM_TimeBaseStructInit(&TIM_InitStructure);
  TIM_InitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_InitStructure.TIM_Period = 6138;
  TIM_InitStructure.TIM_Prescaler = 100;

  TIM_TimeBaseInit(TIM2,&TIM_InitStructure);
  // NVIC for timer
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
  NVIC_Init(&NVIC_InitStructure);

  TIM_ITConfig(TIM2,TIM_IT_Update,ENABLE);

  TIM_Cmd(TIM2,ENABLE);
```

Write to TIM2->CR1 to disable the timer and configure the mode.

$$f_{CK\_PSC} = 64MHz$$

1. Write to TIM2->ARR to set the reload value (ARR).

2. Write to TIM2->PSC to set the prescaler value (PSC); i.e. configure the counter clock frequency

3. Configure the timer interrupt and the NVIC

4. Write to TIM2->CR1 to enable the timer and begin counting.

The counter clock frequency CK_CNT is given by:

$$\text{CK\_CNT} = f_{CK_{PSC}}/(TIM_{Prescaler} + 1)$$

The timer period for the upcount mode is given by the following equation:

$$T_{in\_upcount\_mode} = \frac{(1 + TIM_{Period}) \cdot (1 + TIM_{Prescaler})}{f_{CK\_PSC}}$$

If you want an interrupt every microsecond you could use:

$$TIM_{Period} = \frac{T_{in\_upcount\_mode} \cdot f_{CK_{PSC}}}{1 + TIM_{Prescaler}} - 1 = \frac{1 \times 10^{-6}\text{s} \cdot 64 \times 10^{6}\text{Hz}}{1 + 0} - 1 = 63$$

For this exercise we will need the interrupt to be triggered every 1/100th second to have an accurate stop watch. The serial communication is much too slow to output this in real time, so you should only output the 1/100 seconds information when the clock is stopped.

In the setup for the NVIC a priority can be placed on the different interrupts so if multiple interrupts are triggered simultaneously. The priority is a 4-bit number (0-15) with lower values meaning higher priority.

As before after setting up the NVIC and using *TIM_ITConfig()* to setup the interupt for the timer the special ISR function must be written and remember to clear the pending bit.

Configure timer 2 (CR1 register) with the following settings:

No UIFREMAP remapping, No clock division (CKD), Non-buffered auto-reload preload (ARPE),

Edge-aligned up-counting mode (CMS + DIR), One-pulse mode disabled (OPM), Any update request source (URS), Update events enabled (UDIS). Hint: see pp. 596-597 in [RM].

- Set the required pre-scale and reload value.
- Enable the timer (EN).
- Activate the timer 2 interrupt as described below.

This section of code can be used as inspiration:

```
RCC->APB1ENR |= RCC_APB1Periph_TIM2; // Enable clock line to timer 2
TIM2->CR1 = 0x...; // Configure timer 2
TIM2->ARR = 0x...; // Set reload value
TIM2->PSC = 0x...; // Set prescale value
```

Just like with the GPIO pins, we have to enable the clock line to the timer peripheral before it will function. Next, we have to enable the interrupt to actually use the timer for anything. This is done by writing:

```
...

TIM2->DIER |= 0x0001; // Enable timer 2 interrupts

...
```

But we also need to enable the interrupt in the Nested Vectored Interrupt Controller (NVIC) which actually handles the interrupts. This is achieved by writing:

```
NVIC_SetPriority(TIM2_IRQn, priority); // Set interrupt priority interrupts
NVIC_EnableIRQ(TIM2_IRQn); // Enable interrupt
```

TIM2_IRQn (28) represents the timer 2 peripheral in the NVIC subsystem. Other valid interrupts can be found in `Headers/SPL/inc/stm32f30x.h`, line 167 onwards.

The first line sets the interrupt priority which determines what will be done first if multiple interrupts are triggered simultaneously. The priority is a 4-bit number (0-15) with lower values meaning higher priority. Next, we enable the interrupt by writing to the Interrupt Set Enable register. An interrupt can similarly be disabled by writing to the Interrupt Clear Enable register (NVIC_DisableIRQ(TIM2_IRQn)). It is also possible to disable all interrupts using __disable_irq() and re-enable them using __enable_irq().

As before, the ISR must be defined. This routine will be executed each time an interrupt occurs. For TIM2, the ISR is as seen below. Remember to clear the pending bit for the interrupt line, by clearing TIM2->SR.

```
void TIM2_IRQHandler(void) {
...  //Do whatever you want here, but make sure it doesn't take too much time
    TIM_ClearITPendingBit(TIM2,TIM_IT_Update); // Clear interrupt bit
}
```

At this point you should be good to have a go at making your stop watch (THIS PART IS OPTIONAL):

- Configure timer 2 to generate an interrupt at 100 Hz with the highest priority.
- Configure the interrupt update function such that it updates a structure containing hours, minutes, seconds, and hundredths of seconds. Note: The variables should be made global and volatile - why?
- Output the time to PuTTY every time the second variable is changed.
  Note: Do NOT call `printf()` from the interrupt handler. It is WAY to slow!
- Use the joystick to create a stop watch with two split times:

Center	=	Start/stop
Left	=	Split time 1
Right	=	Split time 2
Down	=	Stop clock and set time to 00:00

Note: When you copy a structure containing h:m:s:hs, an interrupt may occur during copy. You should therefore disable all interrupts during a copy to avoid this.

Exercise 1.5 – Printing text

In the previous exercises, you used `printf()` to print fixed point numbers to the PC over the UART connection. The function `printf()` also supports more advanced formatting as seen below:

```
printf("Value = %02ld\n", val);
```

The '`%`'-sign tells the function that it should be expecting a number, the '`0`' that follows is a flag that signifies that we want to print it with leading zeroes, the '`2`' indicates that the number should be printed with at least two digits, the '`l`' specifies the type of the number to be a `long`, and the '`d`' denotes the type of number to print to be a signed integer. In general the syntax adheres the following format:

```
%[parameter][flags][width][.precision][length]type
```

Arguments in `[]`-braces are optional. Wikipedia has a nice overview of which values all the different arguments can take: https://en.wikipedia.org/wiki/Printf_format_string and https://en.wikipedia.org/wiki/C_data_types

There is a similar function called `sprintf()` which can be used to write numbers to strings:

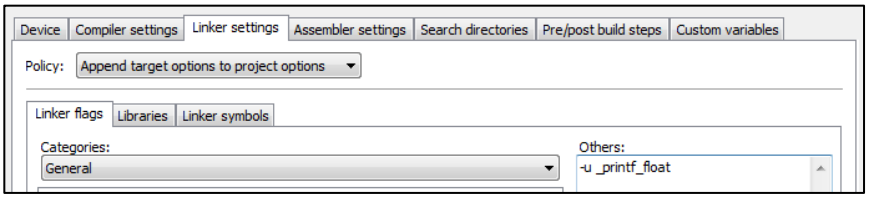
```
uint8_t a = 10;
char str[7];
sprintf(str, "a = %2d", a);
```

This will result in character array `str` containing `"a = 10\0"`. The string contains 7 characters, including the '\0' character that terminates the character array.

- ~~Use `sprintf()` and `led_write_string()` to print the stopwatch status and split times, as it was done in exercise 2.4~~

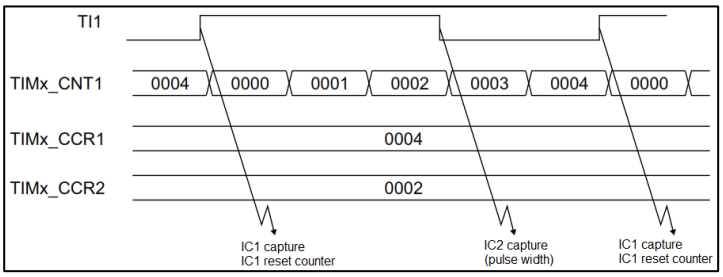
Note: Floating point variables are not supported by `printf()` per default. Therefore, it is necessary to add a flag to the linker configuration:

- Open "Project|Build options|Linker settings|Linker" flags and add "-u _printf_float" to "Others"



You should now be able to print floating point values using `printf()`/`sprintf()`. Please note that this requires slightly more memory!

Exercise 1.6 – Pulse width measurement



For this exercise, you will verify the accuracy of the STM32 timer clock by comparing it to a reference clock, generated by a Tektronix AFG2021 function generator available in the laboratory. You will use the knowledge from the previous exercises to design this system. Additionally, you will learn how to setup GPIO for "Alternate Functions". Timer 2 will be used for this exercise. For a detailed timer block diagram of Timer 2, refer to Figure 189 in the [RM]. The timers can be configured in Input capture mode, where the Capture/Compare Registers can be used to latch the counter value when an external trigger signal has been detected.

Capture/Compare Setup [SPL p. 455]: To measure the period (in the TIMx_CCR1 register) and the duty cycle (in the TIMx_CCR2 register) of a signal applied to the TI1 signal, the following steps are necessary:

More about each step is described below.

1. Enable TIM clock using RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx,ENABLE) function
2. Configure the TIM pins by configuring the corresponding GPIO pins
    a. As done before but now in AF mode (see example below)
3. Configure the Time base unit as described above
    a. Setting are described below
4. Fill the TIM_ICInitStruct with the desired parameters including:
    a. TIM Channel: TIM_Channel
    b. TIM Input Capture polarity: TIM_ICPolarity
    c. TIM Input CApture filter value: TIM_ICFilter
5. Call TIM_PWMIConfig(TIMx, &TIM_ICInitStruct) to configure the desired channels with the corresponding configuration and to measure the frequency and the duty cycle of the input signal
6. Enable the NVIC or the DMA to read the measured frequency.

7. Enable the corresponding interrupt (or DMA request) to read the Captured value, using the function TIM_ITConfig(TIMx, TIM_IT_CCx) (or TIM_DMA_Cmd(TIMx,TIM_DMA_CCx))
8. Call the TIM_Cmd(ENABLE) function to enable the TIM counter.

- Use TIM_GetCapturex(TIMx); to read the captured value.

```
GPIO_StructInit(&GPIO_InitStructAll);
// Then set things that are not default.
GPIO_InitStructAll.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructAll.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructAll.GPIO_PuPd = GPIO_PuPd_DOWN;
GPIO_InitStructAll.GPIO_Speed = GPIO_Speed_50MHz;

GPIO_Init(GPIOx, &GPIO_InitStructAll);
GPIO_PinAFConfig(GPIOx, GPIO_PinSourcey,GPIO_AF_z); //Sets pin y at port x to alternative function z
```

After enabling the clock signal find a suitable GPIO A pin that can be mapped to TIM2_CH1, CH2 or CH3. Possible candidates include PA0, PA1 or PA5. For a full list, refer to the "Alternate function" Table 14-18 in the datasheet [DS]. Please note that PA2 and PA3 can **NOT** be used, as they are used for the UART2! Also keep in mind that some GPIO pins are connected to the mbed hardware such as the potentiometers joystick.

Hint: Look in the Nucleo_302R8_Pinout PDF for the header pinouts!

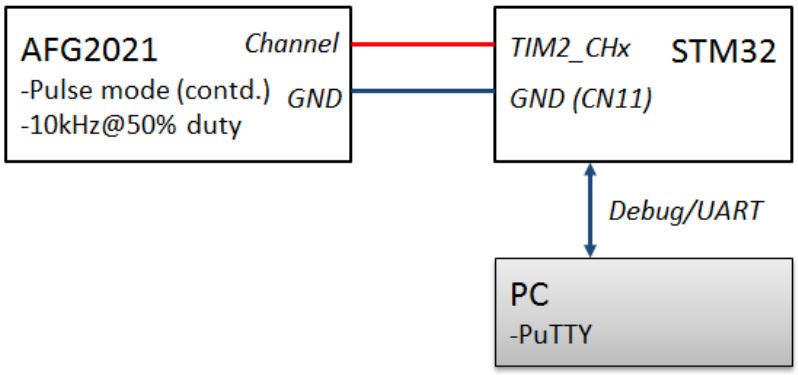
- Write a function – eg. GPIO_set_AF_PAx() and use this function to initialize the selected pin in "Alternate Function" mode.
- Settings for the TIM_ICStructInit
    - Set the input filter (TIM_ICFilter). This filter can be used to allow the input signal to settle before triggering a transition. We will leave the filtering disabled for now.
    - Set the input prescaler (TIM_ICPrescaler). The capture should occur on each valid transition so the prescaler can be disabled for now
- Settings for the TIM_TimeBaseInitStruct

Before we can start, it is necessary to setup the Timer 2 parameters, as it was done in the previous exercise. This time, however, we do not use the timer to trigger an interrupt on overflow but instead use it as a continuous counter, counting until the maximum value is reached. Therefore, the Period is just set to its max value and the TIM_ITConfig setting the TIM_IT_CCx to trigger the interrupt.

- Calculate the prescaler so it results in a counter clock frequency of 1MHz; i.e. a period of 1uS.
- Determine the maximum interval (in seconds) that you can count until the counter overflows.

- Determine the expected values in CCR1 and CCR2, assuming that a 10kHz, 50% duty PWM signal is given as input. Is the counting frequency high enough to obtain adequate resolution?
- Configure the timers prescaler value that you calculated earlier.
- Set the auto-reload register to the maximum possible value. (i.e. 0xFFFFFFFF)
- Enable the counter by setting TIMx_CR1 accordingly.
    - *TIM_Cmd(TIM2,ENABLE);*
- Enable CC1 interrupts and enable TIM2 interrupts in the NVIC, as it was done in exercise 2.4:
    - *TIM_ITConfig(TIM2,TIM_IT_CC1,ENABLE);*
- As before, the ISR must be defined for `TIM2_IRQHandler()`. For TIM2, the ISR below can be used as a template. The CC1 interrupt will trigger every time the TIMx_CCR1 register is updated. Remember to clear the pending bit for the interrupt line, by clearing the flag by using the function `TIM_ClearITPendingBit()`:

```
void TIM2_IRQHandler(void) {
    TIM_ClearITPendingBit(TIM2, TIM_IT_CC1);

    ICValue1 = TIM_GetCapture1(TIM2); // Period
    ICValue2 = TIM_GetCapture2(TIM2); // Duty/Width
    ICValid = 1;
}
```



The measurement setup is seen in the image above. Connect the function generator to the STM32 board and generate a pulse with a frequency of 10 kHz at 0 to 3.3V. The duty cycle must be 50%!

NOTE: Make sure that you do NOT input negative voltages to the microcontroller! Positive voltages only!

Set limits as the first thing after turning it on under the output menu.

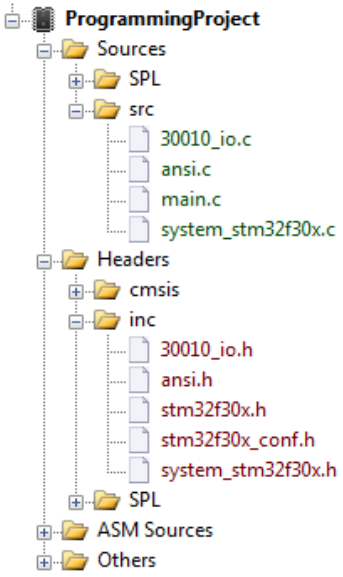
- Write a formula to calculate the period and duty cycle from the two counter values. Proceed to calculate the period and duty cycle from the contents of CCR1 and CCR2. You can use the `TIM_GetCapture1() and TIM_GetCapture1()` functions to obtain the latest values stored in the two registers.
- How accurate is the measurement? Can you think of a way to improve the accuracy? Why is/isn't the measurement accurate?
- Determine the highest frequency that can be measured by the instrument. How can the frequency be increased further?
- How can the Input prescaler (IC1PSC) and the input filter (IC1F) parameters in the timer configuration be used to improve the accuracy? Do they affect the max input frequency or measurement interval?

You should now improve the usability of the instrument:

- Compute and display the period, frequency and duty cycle in real time using PuTTY.
- Calculate the average of the last ten measurements and display using PuTTY.
- Use the joystick “center” button to pause/unpause the measurement and “down” to reset the instrument.
- Use the LEDs to indicate whether the measurement is running or paused (eg. green/red)
- Use the LED to indicate each time an interrupt is triggered, i.e. when a measurement is complete.

Exercise 1.7 – Application structure

The previous functions that you have created could potentially be used in many different programs. As such, it makes sense to put them all in a separate C-file and then importing that into the project. This not only makes maintaining the code base easier, it also makes the main-file of our program a lot easier on the eyes.



We'll start by creating a new C-file in EmBitz.

- Create a new empty file by selecting File - New - Empty File (Ctrl+Shift+N).
- Click Yes when asked whether to include the new file into the current project.
- Save the file in `<PROJECT LOCATION>/src` and call it `ansi.c`.
- Create a similar file in `<PROJECT LOCATION>/inc` and call it `gpio.h`.

When you're done your program structure should look like the figure on the right.

The ansi.c should contain the functions that you've created to control the terminal.

- Move all your GPIO related functions to gpio.c / gpio.h (remember any necessary `#includes`).

The header file, `gpio.h`, should contain types, constants, and function declarations related to ANSI based terminal control. As an example, for the supplied color functions it would look like:

```
void setLed(uint8_t val);
```

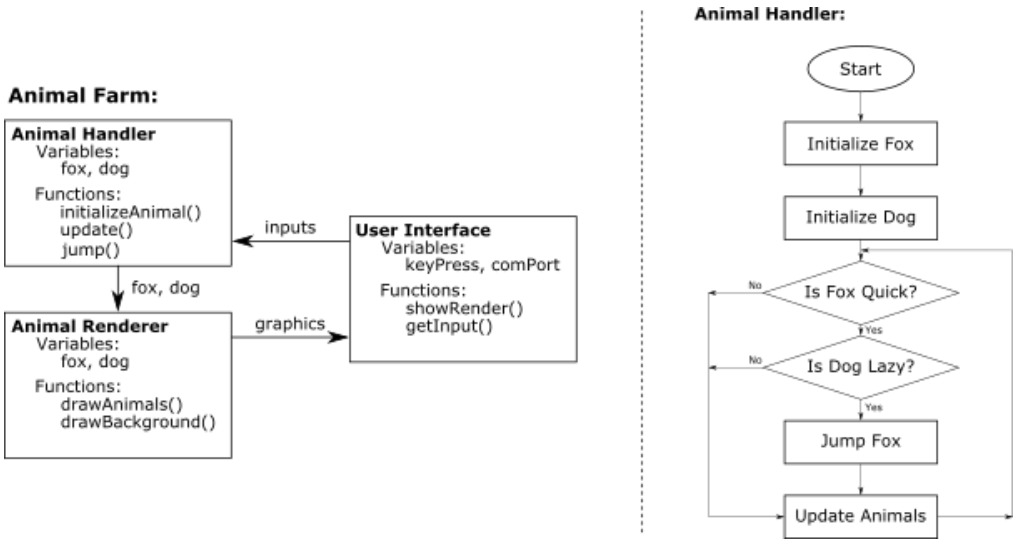
Sometimes it is necessary to include the same header file multiple times which can slow down compilation times as well as cause errors in some instances. These issues can be avoided by using the following template:

```
#ifndef _GPIO_H_
#define _GPIO_H_
#include <...>; // Whatever needs to be included

#define ... // Whatever needs to be defined

void functionName( type parameter );

#endif /*_GPIO_H_ */
```

This piece of code tells the compiler to only include the function definitions once in the project. It is suggested to follow this example when creating new header files and it will make a significant difference when working with large projects. You will notice that the library included in Exercise 1 (`30021_io.h` and `30021_io.c`) follows the same structure of having a source and a header file.

Exercise 1.8 – Documentation

For each of your exercises, you will have to create a flowchart and a block diagram describing the program as well as descriptions of the functions you've made. The image below shows an example of how a block diagram for a program called Animal Farm and a flow chart for a subcomponent called Animal Handler may look:



Similarly, a function description for a function Jump may look like:

Jump	Jumps fox over dog
Syntax:	`void Jump(fox* f, dog* d);`
Parameters:	`f`: A fox. Should be quick, preferably brown. `d`: A dog. Should be lazy.
	The function takes a fox and a dog and jumps the fox over the dog. If the fox is not quick an error will be thrown.

- Create a block diagram for you program. It will be very simple.
- Create a flowchart of your main loop.
- Create function descriptions for the functions you've created.

Report

Write a report of the exercise. The report should be about 5 pages long including a description, a chart, a table of used ports, documentation of results (eg. screendumps) and a well commended featured source code including header file.

Participants on a team draw up a single report with a cover, which show participants names, student number and group number. It should not be specified in the text, who wrote what. The report is sent as DOC or PDF file to Campusnet/Assignment the following Monday at 23.59. The team will have an annotated report back via e -mail.

The report should typically include most of these standard points:

Title

Abstract Subject , methods, results.

Introduction general description, no details.

Problem definition What is it about.

problem limitation What you cannot with this method or this hardware.

Theory How do the various components and methods.

Hardware Diagram and table of used ports.

Software Function Description (no source code here).

Test method/software What worked, what did not and why.

Conclusion Must correspond to the problem formulation.

Appendix Source code, diagrams, possibly. Print drawings,

Data Sheets (only important pages), references and links.