1 What if we have no software framework?

Sometimes, if we use microcontrollers manufactured by smaller brands or we use new models we don't have a software framework like we do for the UC3A0512.
In this chapter we explain the necessary steps needed assuming we only have the datasheet. We do this at the example of a GPIO pin. More precise, we are going to control the GPIO pin connected to LED 0.

1.1 Using LED 0

Every microcontroller has its own datasheet. You should already have the datasheet for the UC3A0515 (*AT32UC3A0_A1_Series_Complete.pdf*). If we look at the index of the document we can find the chapter *"22. General-Purpose Input/Output Controller (GPIO)"*. This chapter contains all we need to know if we want to use one of the GPIO pins.

Read this chapter carefully before you use the module!

Once you read the chapter you can start programming. Since we don't have a software framework we need to build the references to the respective registers by ourself. Chapter *22.5 General Purpose Input/Output (GPIO) User Interface* provides us with the needed information.

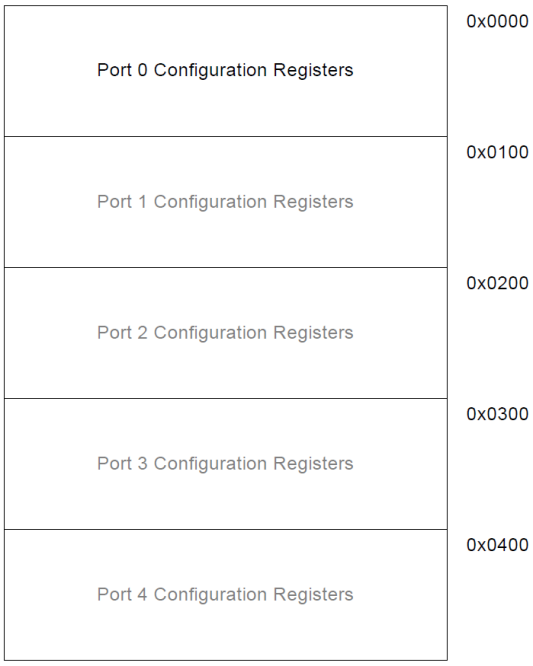


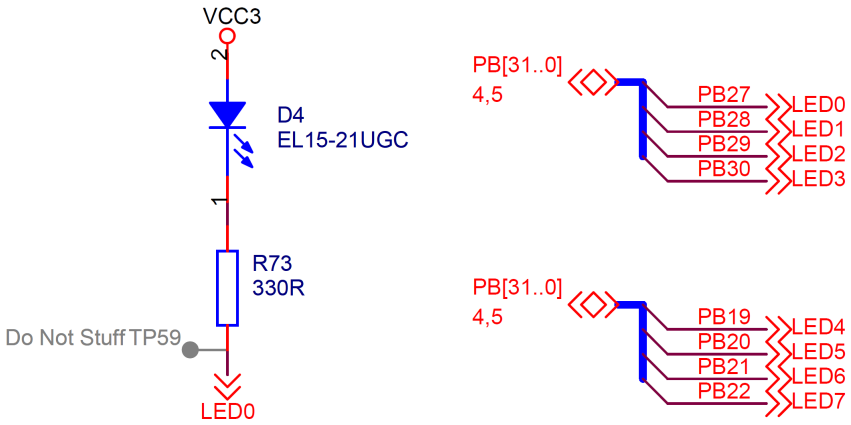
Figure 1: Register map of the GPIO module

As mentioned in the datasheet, the ports PA, PB, PC and PX do not directly correspond to the GPIO ports! However you can use the following equations to compute the needed values from the GPIO number.

$$\text{GPIOPort} = \lfloor \frac{\text{GPIO number}}{32} \rfloor$$

$$\text{GPIO Pin} = \text{GPIO number} \mod 32$$

Since we don't know the GPIO number yet, we need to take a look at the schematics of the EVK1100 (*EVK1100_SCHEMATICS_REVD.pdf*). We can find the LED0 on sheet 7. Figure 2 shows the schematics. We see that LED0 is connected to PB27.



(a) LED0 connection (b) Connection to the controller

Figure 2: Schematics of the connection of LED0

Now we need to find out to what GPIO port and pin PB27 is connected to. If we take a look at the chapter *"12. Peripherals"* we get information about all peripherals connected to the microcontroller. In subsection *"12.7 Peripheral Multiplexing on I/O lines"* we can find a table stating all functions connected to the individual pins. GPIO is such a function. As shown in the figure below, the pin PB27 is connected to GPIO59.

Table 12-9. GPIO Controller Function Multiplexing

7	11	PB24	GPIO 56	TC - B0	USART1 - DSR	
8	13	PB25	GPIO 57	TC - A1	USART1 - DTR	
9	14	PB26	GPIO 58	TC - B1	USART1 - RI	
10	15	PB27	GPIO 59	TC - A2	PWM - PWM[4]	
14	19	PB28	GPIO 60	TC - B2	PWM - PWM[5]	

Figure 3: Part of the peripheral multiplexing table

By plugging "59" in the above equations we know that the LED0 is connected to:

- Port: 1
- Pin: 27

Now we can go back to the GPIO chapter of the datasheet. Since we now know the port and pin we need we can start to write our defines for the register addresses.

```
/* Absolute GPIO_BASE_ADDRESS */
#define GPIO_BASE_ADDRESS      0xFFFF1000

/*Define the GPIO_PORT1_OFFSET*/
#define GPIO_PORT1_OFFSET      0x100
```

For our convenience we can connect them right away.

```
/* Absolute GPIO_PORT1_ADDRESS */
#define GPIO_PORT1_ADDRESS     GPIO_BASE_ADDRESS+GPIO_PORT1_OFFSET
```

Now we have a shortcut to the start of the register set of port 1. *"Table 22-2 GPIO Register Map"* gives us information about the relative offsets of the registers we already know (remember Lab 1, Chapter *"3.3.3 Hardware registers"*). To increase readability of our code we define the offsets we need as well (since this is an example we don't define all of them):

```
/* Relative register offsets */
#define GPIO_GPER_OFFSET       0x00
#define GPIO_ODER_OFFSET       0x40
#define GPIO_OVR_OFFSET        0x50
```

Up until know we have the start address of the GPIO port 1 register set and the offsets of the registers we need. Now we can define the pointer to the actual register, so we can read and write the register values.

```
/* Some defines so we can access the GPIO port 1 registers directly */
#define GPIO_PORT1_GPER   (*((volatile uint32_t*)(GPIO_PORT1_ADDRESS + GPIO_GPER_OFFSET)))
#define GPIO_PORT1_ODER   (*((volatile uint32_t*)(GPIO_PORT1_ADDRESS + GPIO_ODER_OFFSET)))
#define GPIO_PORT1_OVR    (*((volatile uint32_t*)(GPIO_PORT1_ADDRESS + GPIO_OVRC_OFFSET)))
```

Try to explain the C-code above. What is done there?
With this information we can now write the code to initialize the GPIO-pin.

```
void LED0_init(void){
  /* Select the pin we are going to use as GPIO */
  GPIO_PORT1_GPER |= (0x01 << 27);
  /* Enable the corresponding diver for that pin */
  GPIO_PORT1_ODER |= (0x01 << 27);
  /* Set the value of that pin to HIGH */
  GPIO_PORT1_OVR  |= (0x01 << 27);
}
```

Note the OR expression ($|=$) we use to set the pin. This is done since we don't want to affect the other bits some other function might has set already.
Now we can write a function to set the output value:

```
void LED0_output(uint8_t value){
  if(value != 0){
    /* Set the value on the LED0 pin to HIGH */
    GPIO_PORT1_OVR |= (0x01 << 27);
  }else
    /* Set the value on the LED0 pin to LOW */
    GPIO_PORT1_OVR &= ~(0x01 << 27))
  }
}
```

Note the expression we use to clear the bit to turn LED0 off. Write down the Boolean expression to get a better understanding of what happens.

1.1.1 There are different ways...

There are several ways to get the same result. The above example is meant to be simple.

A more elegant way would be to define a structure type containing all registers of one GPIO port (*"Table 22-2 GPIO Register Map"*). This structure can then be used for all GPIO ports and we don't need to define everything multiple times. Note you need to be careful with the variables in the structure, they should have the same size as the registers. If there is a "hole" in the register map you need to put a dummy variable of the same size in the struct in order to generate the same "hole".

1.2 Pulse-Width Modulation

Pulse-Width Modulation (PWM) is a modulation technique to allow the control of the power supply of electrical devices. This can be for example the brightness of a LED or the speed of an engine. The voltage level is controlled by switching the supply voltage on and off. If the switching happens fast enough the sink experiences a voltage equivalent to the mean voltage value. However switching needs to happen at a frequency much faster than what would be noticed by the sink.
The signal can be described using its duty cycle. The duty cycle describes the percentage of one period in which the signal is active and is given in percent.
Figure 4 illustrates a PWM signal with period T. The duty cycle of the signal is 75%. As described above this is experienced by the sink as a voltage of $75\% \cdot U$ (red curve).

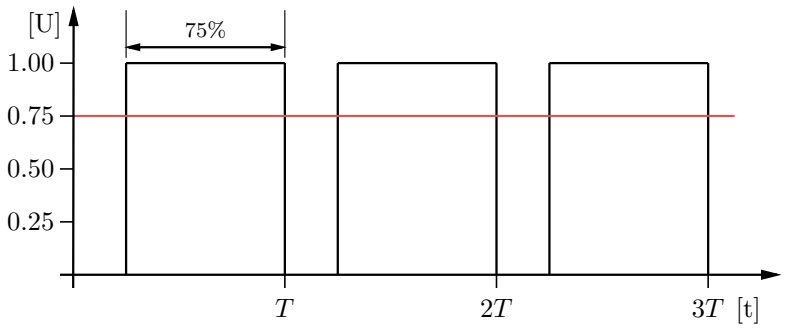


Figure 4: Pulse-Width Modulation

1.3 Assignment

Not all microcontroller have as many modules on them as the UC3A0512 on the EVK1100. However many applications use PWM signals. In this assignment we generate a PWM signal using the timer module on the board (don't use the PWM module!). Timer modules are present on almost all available microcontroller.

We can use a timer to generate the PWM signal shown in Figure 4:
Write a program that generates a static PWM signal using the timer. The signal should affect the GPIO pin connected to LED0. Don't use the framework functions and program the registers directly. You find information about the registers in the processors datasheet. In fact don't use any of the predefined Atmel header files, the only allowed header external header file is *stdio.h* for the standard integer types.
Use the schematics to get information about the location of the GPIO pin LED0 is connected to.

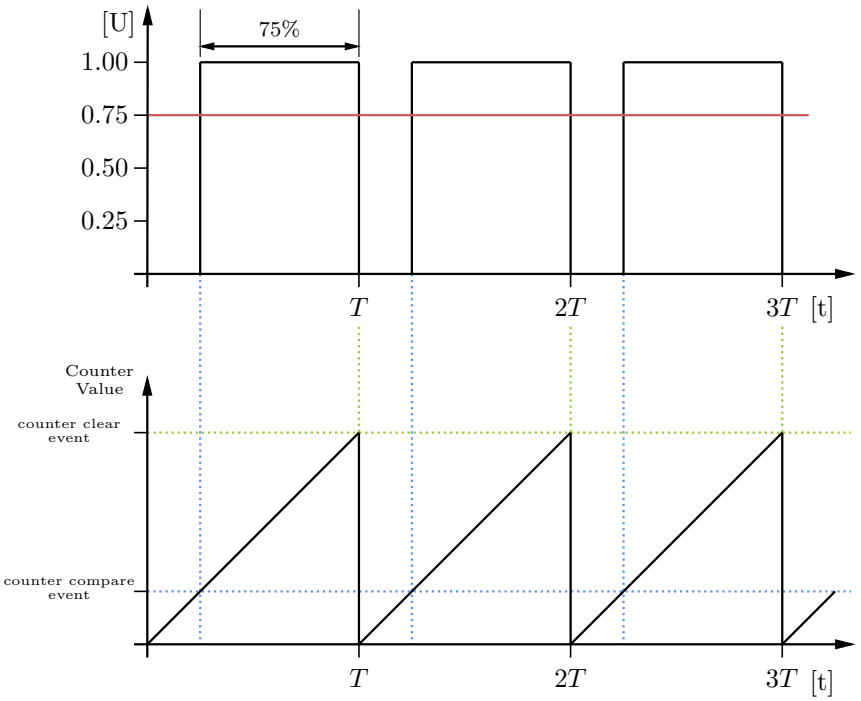


Figure 5: Pulse-Width Modulation using a Timer

Show and verify the resulting PWM signal by using the DAQ with its oscilloscope functionality. Verify that the timings of the measured signal are as expected.

Hint: since the internal clock is not accurate you might need to use the external real-time clock on the board.

1.4 Optional assignment

Often it is needed to generate a sinus signal, for example to drive a motor or a speaker. However not all microcontroller implement modules which offer such functionality. For the optional assignment you will modulate a sinus signal using the functionality you implemented in the main assignment. To do this we use the Digital Direct Synthesis (DDS) method. Divide the time domain in even slots (the period of the PWM). For each slot the duty cycle of the PWM is set to match the average value of the sine wave in the same slot. Figure 6 depicts both, the sine wave and the modulated curve. Note: A lookup table can be used to hold the duty cycle values for each slot. Make sure to select a suitable frequency of your output signal and a suitable slot size.
The modulated curve can be transformed into a clean sinus signal by connecting a lowpass filter

(usually a Chebyshef Lowpass filter with 12.5 KHz can be used for sampling frequency of 32 KHz). Since this is hardware based we do not do this for this optional assignment.

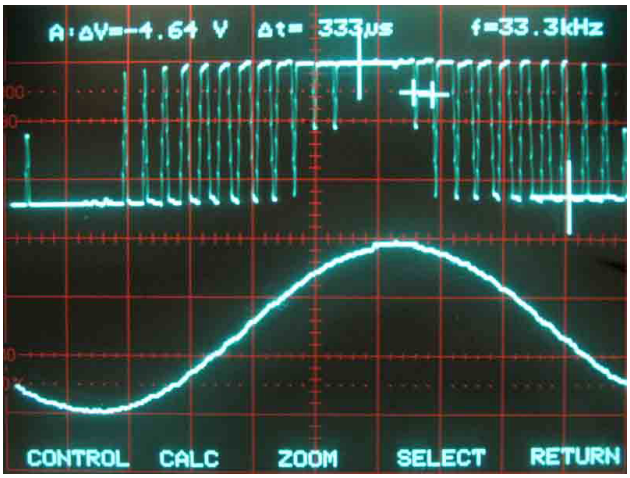


Figure 6: On top the modulated signal and on the bottom the resulting sine wave.

1.5 Measurement

We use the myDAQ to verify the quality of the generated signal. The banana connections can't be used as input if the oscilloscope functionality is used. Thus we have to use either input AI0 or AI1. In the following it is assumed that AI0 is used for the measurement. We also assume that LED0 is used for the PWM signal. LED0 is connected to GPIO59 which is accessible on connection J26, pin 31. The myDAQ needs to be connected as shown in Figure 7.

1.6 Comments

1.6.1 Debugging

Not all registers can be written if the execution is stopped during debugging. The timer register RA is not written if the execution stops at a breakpoint and single steps are performed at the respective instructions.

1.6.2 PCB Label

The labels at the PCB are not equal to the labels in the schematic. LED0 in the schematic is LED1 on the board, the same goes for the following LEDs.

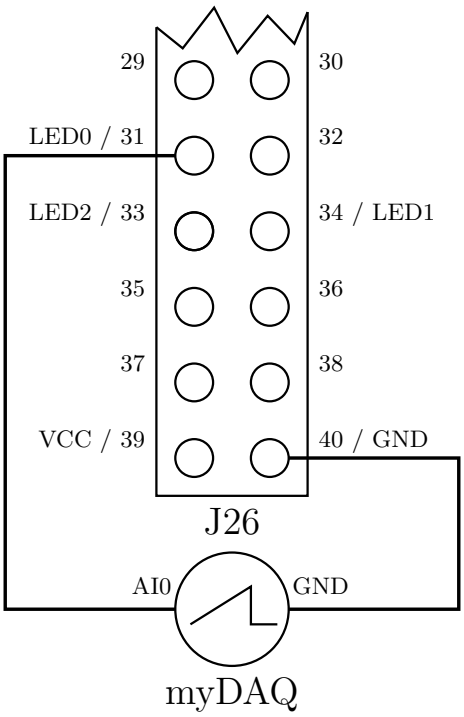


Figure 7: myDAQ connection