Embedded Systems Exam Bonus Project

Project name: Green Gardening

Contributers: Leander Lauenburg, Christoph Clement, Anselm Coogan

18.01.18

1 Hardware

2 Software

As stated in the previous chapter we used the free IDE *PsoC Creator* to program the micro controller. This software offers a three layer approach.

In the first layer, the top design, you can graphically configure the technical layout and wiring of the components you are using from the micro controller. In the second layer you assign the actual pins of the board to the different components configured in the top design. Lastly, the third layer is writing the code that is executed on the micro controller.

The entire source code discussed here (and successive versions) can be inspected in this git repository: https://github.com/AnselmC/GreenGardening

2.1 Top design

The top design enables you to easily make a circuit diagram for your project. You have access to all the actual components of the micro controller, as well as so-called off-board components that are there for documentation and clarification purposes only (colored blue). The components we used for this project are: a real-time clock, a universal asynchronous receiver transmitter, and digital input and output pins.

2.1.1 Real-time clock

The built-in real-time clock component can be configured extensively and has a resolution of one second. We set the clock to use the 24-hour time format and synched it to Central European Time.

2.1.2 UART

The universal asynchronous receiver transmitter component allows for setting up a bidirectional data connection on a given port. We are using this for debugging purposes.

2.1.3 Digital input pins

The input pins in our project are the light sensor and the moisture sensor. We configured these input pins to have dedicated interrupts. The interrupt for the light sensor is activated on both falling and rising edges, i.e. from light to dark and vice versa. This is because the light sensor is situated so that it is independent of the LED strips and only senses the environment light. Thus, if the LED's have been switched on because it was previously too dark and now the surrounding light suffices the LED's should be switched off. In contrast the interrupt for the moisture sensor is only activated on a falling edge, i.e. when it gets too dry.

2.1.4 Digital output pins

As output pins we have set up the water pump and the LED strips.

2.2 Pin connections

The second layer is actually called *Design Wide Resources*. However, we are only using the pin assignment part in our project. Here you get an overview of the micro controller and can then assign the components configured in the top design with the components on the board and the components that have been connected via the digital or analog GPIO pins.

2.3 C Code

PsoC Creator automatically generates an API from the top design previously configured. This allows you to control all components with very little self-written C code. In our main.c we have defined the following global variables used throughout our code:

- WATERING_TIME_IN_SEC: the time the watering pump should be on for one watering session
- STARTING_HOUR and ENDING_HOUR: the timeframe in which the plant should be exposed to light - in 24 h format (e.g. from STARTING_HOUR: 10 to ENDING_HOUR 20)
- NUMBER_OF_MEASUREMENTS: The number of measurements a sensor should make in one setting
- TIME_BETWEEN_MEASUREMENTS_MILSEC: The time (in milliseconds) to wait between the single measurements
- BUFFER_SIZE: The array size of the buffer used for writing debug information to the console
- ON and OFF: Macros used for more clarity in code

In order to use less CPU time and power, we are working with interrupts. There are two interrupts: one for the moisture sensor and one for the light sensor. The interrupts are triggered on both falling and rising edges of the sensors. This means that whenever the light changes from too dark to light or vice versa, or the moisture sensor changes from not moist enough to moist and vice versa the interrupts are called. In PsoC Creator these interrupts are configured in main.c as follows (for an arbitrary pin named Pin_A):

```
CY_ISR(<<name_of_interrupt_handler>>)
{
    /* Code to be executed on interrupt */

    Pin_A_ClearInterrupt();
}
```

The name of the interrupt handler can be whatever you choose. However, best practice is to name it *Pin_A_Handler*. It is essential to clear the interrupt after the code has been executed, otherwise the interrupt can?t be called again. This is done with the generated *Pin_A_ClearInterrupt()* function. In order to enable interrupt handling, in your *main()* method you will need to call *CyGlobalIntEnable*. Then, you need to initialize every interrupt like this: *Pin_A_int_StartEx(Pin_A_Handler)* The StartEx function of your interrupt-enabled pin is also automatically generated by PsoC Creator. This means that in our project the *main()* function solely consists of initializing the interrupts and an endless for-loop with no-code. Thus, the CPU can sleep until one of the sensors interrupts it.

```
int main(void)
{
    /* Enable global interrupts. */
    CyGlobalIntEnable;

    /* Place initialization/startup code here */
    UART_Start();
    RTC_Start();
    Pin_Light_Sensor_int_StartEx(Pin_Light_Sensor_Handler);
    Pin_Moisture_Sensor_int_StartEx(Pin_Moisture_Sensor_Handler);

    for(;;)
    {

    }
}
```

Our two interrupt functions each call one function, namely *checkLight()* and *checkMoisture()*. *checkLight()* first calls the real-time clock component to return the current time and then extracts the current hour. It then checks whether the returned hour is in

the designated timeframe. If it isn't, the light is turned off and the function terminates. If it is, then we take a pre-defined amount of measurements (as configured in the global variables) with a pre-defined delay between each measurement. If the light sensor only measures enough light for less than half of the measurements, the LED's are turned on and the function terminates. Else, the lights are turned off and the function terminates.

```
void checkLight()
{
    /* Get current hour */
    int hour = RTC_GetHours(RTC_GetTime());

    /* Check whether hour is outside of designated time frame
    if so, turn it off */
    if(hour < STARTING_HOUR || hour > ENDING_HOUR){
        Pin_LED_blue_Write(OFF);
    }
    else {
        int positive_measurements = NUMBER_OF_MEASUREMENTS;
        /* Make several measurements of the light sensor */
        for(int i = 0; i < NUMBER_OF_MEASUREMENTS; i++){
            positive_measurements -= Pin_Light_Sensor_Read();
            CyDelay(TIME_BETWEEN_MEASUREMENTS_IN_MILSEC);
        }
        /* If the light sensor catches not enough light,
        turn on the LED's */
        if (positive_measurements < NUMBER_OF_MEASUREMENTS/2)
        {
            Pin_LED_blue_Write(ON);
        }
        /* Else turn them off */
        else
        {
            Pin_LED_blue_Write(OFF);
        }
    }
}
```

The *checkMoisture()* function operates similarly. Again, we take a pre-defined amount of measurement to rule out anomalies. Then, if less than half of these measurements indicate that the soil is too dry, the water pump is turned on for a specified amount of time and then turned off again after which the function terminates. If the soil is moist enough, nothing happens and the function terminates.

```
void checkMoisture()
{
```

```
    int positive_measurements = NUMBER_OF_MEASUREMENTS;
    /* Make several measurements of the moisture sensor */
    for(int i = 0; i < NUMBER_OF_MEASUREMENTS; i++){
        positive_measurements -= Pin_Moisture_Sensor_Read();
        CyDelay(TIME_BETWEEN_MEASUREMENTS_IN_MILSEC);
    }

    /* If the ground isn't wet enough,
    water the plant for a specified amount of time */
    if (positive_measurements < NUMBER_OF_MEASUREMENTS/2)
    {
        /* Turn pump on */
        Pin_pump_Write(ON);
        /* Wait for specified amount of watering time */
        CyDelay(WATERING_TIME_IN_SEC * 1000);
        /* Turn pump off */
        Pin_pump_Write(OFF);
    }
}
```

Throughout the code we are outputting debug information that was critical for development. For this we are using the UART component. To use it it must be be initialized in the *main()* method as can be seen above. Then, when you want to write to the connected pod (e.g. a terminal on your development machine) you can do so like this:

```
sprintf(buffer, "Let's go\r\n");
UART_UartPutString(buffer);
```

3 Learnings from the lecture