
	<p>Professorship Computer Engineering  <b>Automotive Software Engineering</b>          Prof. Dr. Dr. h. c. Wolfram Hardt          Dr. René Bergelt</p>	
<p>Practical Unit 1</p>	<p>Embedded Programming</p>	<p>Winter Semester 2024/25</p>

## Contents

1. Introduction .....	1
1.1. ECUs in modern vehicles .....	3
1.2. Sensors - ECUs- Actuators .....	3
1.3. Programming Microcontrollers .....	3
2. Development Environment .....	5
2.1. Evaluation Board .....	5
2.2. Compiling and Building Your Project .....	6
2.3. Flashing Your Project to the ECU .....	7
3. Tasks .....	8
3.1. Pin Configuration .....	8
3.2. Task 1 - Reading and Displaying the Potentiometer Value .....	8
3.3. Task 2 - Reading and Displaying the Light Sensor Value .....	9
3.4. Task 3 - Implementing a Binary Counter .....	10
Bibliography .....	12

## 1. Introduction

For more than a hundred years, motor vehicles are produced and improved. The use of micro-electronics provided new possibilities for engineering and implementation of complex functionalities for automobiles. Today most car drivers use their automobile for more than one hour per day, this results in increasing demands of the driver to the vehicle. To meet all the demands the usage of hardware and software in automobiles increases rapidly. The fast and reliable reaction of automotive functions is assured by software, electronic control units and their communication amongst each other. Furthermore former mechanical functions were replaced by software, electronic control units, sensors and actuators. Today luxury class automobiles contain up to 80 connected electronic control units i.e. engine management, airbag control, power locks and electric windows. In the case of the electric windows the software automatically reverses/stops the window motor if it senses an obstruction while closing. Nowadays 90% of the innovations in car development concern the electronics and the software. That includes the conversion of classic mechanic functions as well as new features for example the automatic parking system. For the practical automotive specific terms are necessary which are explained in Table 1.

Table 1: Automotive specific terms and definitions

Term	Definition
Embedded system	Computer with a dedicated purpose
Distributed system	Computer network which communicates by message exchange and realizes a common task
Interactive system	Hardware and software components which process user input and provide results in a way for humans to perceive
Reactive system	Hardware and software components which interact permanently (often cyclically) with the environment. Reactive systems normally do not terminate.
Real-time system	Computer system, which guarantees the calculating of a job within a predefined deadline (time limit).
Communication	The communication in the automotive domain describes the message exchange of distributed and embedded systems with the help of protocols.
Reliability	Ability of a system to consistently perform its required function within a specified time limit without degradation or failure.
Availability	Proportion of time a system is in a functioning condition.
Security	Extent to which a system is protected against data corruption, destruction, interception, loss or unauthorized access.
Monitoring	Observing of the current system status to detect failures and to initiate counteractions.
Electronic Control Unit (ECU)	An embedded system which controls one or more of the electrical subsystems in a vehicle. Often realizes a closed control loop by reading sensors and control actuators based on the readings
Sensors	Electronic device used to measure a physical phenomenon such as temperature, pressure or loudness and convert it into an electronic signal
Actuators	Transforms electronic signals from an ECU into mechanical / physical actions.

## 1.1. ECUs in modern vehicles

Increasing customer demands, such as lower fuel consumption, improving of driving safety and comfort are closely related to the use of more and more electronics and software in modern automobiles. The major requirements for ECUs in today's automobiles are the following:

- Reliable under rough and changing ambient conditions like temperature, humidity and vibration
- High demands regarding dependability and availability
- High demands regarding security

ECUs work as a reactive system. The driver has no direct influence on their functionality besides activating or deactivating functions or specifying some simple parameters such as target speed for the Cruise Control system.

## 1.2. Sensors - ECUs- Actuators

Sensors deliver the data which is then analyzed by the ECU. They measure physical quantities and convert them into electric quantities, so they can be processed. There are various types of sensors that react to different physical quantities, such as air pressure, humidity, light intensity, distance, temperature or density. The ECU periodically reads the sensor data and processes the implemented software. At this point it is decided whether an actuator is activated or not. If the sensor data differs between the periods in most cases the actuators are accessed by the ECU to react on the environmental changes.

Figure 1 shows the relation between sensor, ECU and actuator. The concrete components shown are the ones required to realize an electric window. A simple switch serves as the sensor. Once the switch is pressed the ECU activates the actuator, in this case the electric window motor, to either open or close the window.

Another typical example is climate control, where the sensor is a thermometer and the actor is the heating / cooling unit. Based on the target temperature the ECU heats or cools the inside of the car or stays idle.

Furthermore the actuators can be accessed if sensor data has exceeded a certain threshold. A simple example is the headlight which is automatically switched on at nightfall. A photodiode measures the light intensity and forwards it to the ECU. If the intensity of the light falls below a threshold the headlights are turned on.

## 1.3. Programming Microcontrollers

Microcontrollers are integrated circuits which contain a processor, memory and specific hardware periphery modules. The hardware modules can perform digital and analog functions like general purpose in- and output (GPIO), pulse width modulation (PWM), timers (PIT = periodic

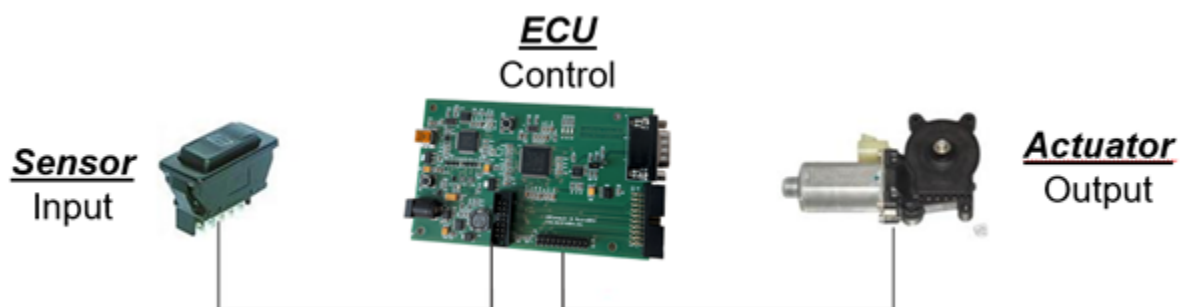


Figure 1: Components needed for a closed control loop

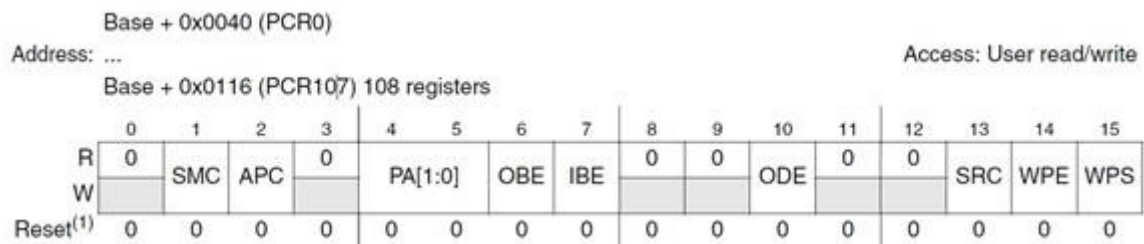


Figure 2: PCR register structure

interrupt timers), and analog to digital converters (ADC). Sometimes peripheral modules for communication are also integrated, e.g. CAN, FlexRay, Ethernet, SENT, I2C, and SPI. These modules can be accessed and configured through integrated registers. Microcontrollers are generally programmed in Assembler or in C language. In this practical course, the high-level language C is used to program the ECUs. This kind of programming is similar to programming software for PCs. But in embedded systems, the programs access hardware registers directly to configure the peripheral modules by setting certain bits to activate a desired peripheral function on a particular microcontroller pin. Once the pins are configured for a particular peripheral function, other registers are used either to control or to receive a value from the environment. For this purpose, the memory addresses of the registers are used, which are defined in the reference manual of the specific microcontroller. In general, the pins of the microcontroller are organized by ports. Every port of the microcontroller has a set of pins. Each port can only perform certain tasks due to limited support for remapping peripheral functions. More details can be found in the reference manual. Figure 2 shows an extract of the reference manual of the microcontroller, which can be found at [1]. It shows the structure of the pad configuration register (PCR) of the System Integration Unit Lite (SIUL) module, which is used to configure digital in- and outputs.

To access the PCR registers in C code, it is possible to directly access the base address of the registers or to write **SIU.PCR[X].R**, which is a macro defined in the "xpc56el.h" header file. The meaning of bit fields is also described in the reference manual.

### Example:

Objective: Configure port **B8** of an SPC56EL60 microcontroller as a digital output and then set it low.

Look into the reference manual and find the port pin. As shown in Figure 4, port pin **B8** supports general purpose in- and output (GPIO) and the corresponding PCR number is 24 (**PCR[24]**).

To configure an I/O pin the "output buffer enable" bit (**OBE**) must be set. In Figure 3 the OBE bit is at position 6. To set this bit, "0x0200" (hexadecimal), "0b0000001000000000" (binary), or "512" (decimal) could be written to the register. In C it can look like this for port **B8** of the STM SPC56EL60 microcontroller:

```
SIU.PCR[24].R = 0x0200;    /* configures port B8 as output */
```

Every module on the microcontroller can be configured in a similar way. In the reference manual all functionalities and registers are described. These are the basics of microcontroller and driver programming.

Port pin	PCR register	Alternate function <sup>(1)</sup> , (2)	Functions	Peripheral <sup>(3)</sup>	I/O direction <sup>(4)</sup>	Pad speed <sup>(5)</sup>		Pin	
						SRC = 0	SRC = 1	100-pin	144-pin
B[7]	PCR[23]	ALT0 ALT1 ALT2 ALT3 — —	GPIO[23] — — — AN[0] RXD	SIU Lite — — — ADC0 LIN0	Input Only	—	—	29	43
B[8]	PCR[24]	ALT0 ALT1 ALT2 ALT3 — —	GPIO[24] — — — AN[1] ETC[5]	SIU Lite — — — ADC0 eTimer0	Input Only	—	—	31	47
B[9]	PCR[25]	ALT0 ALT1 ALT2 ALT3 —	GPIO[25] — — — AN[11]	SIU Lite — — — ADC0 – ADC1	Input Only	—	—	35	52

Figure 3: Pin list and multiplexing [1]



Every microcontroller has its own individual modules and registers. That is why it is necessary to read the reference manual and to program new drivers when the hardware is changed.

## 2. Development Environment

### 2.1. Evaluation Board

In the practical you will program an industry-standard ECU which is situated in a development rack as shown in Figure 4. In order to fulfill the tasks you will need to read sensor inputs and based on this control actuators. All sensors and actuators (i.e interaction points) are provided on the front panel of the rack which are connected to the corresponding pins of the ECU's microcontroller. The front panel is shown in Figure 5.



Figure 4: Hardware Setup for ASE Practical Unit 1

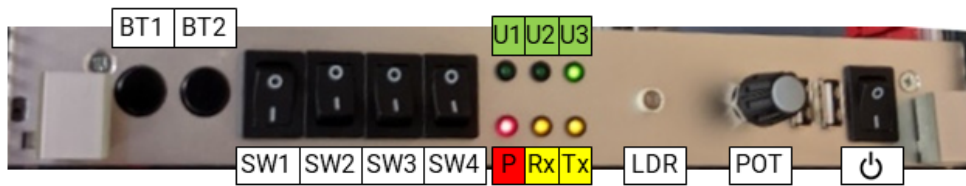


Figure 5: The front panel of the rack with buttons, switches, LEDs, light sensor, potentiometer and power button (from left to right, photo rotated by 90°)

Through the front panel of this rack you can access the sensors and actuators needed for the practical tasks (see schematic on the right). As indicated in Figure 5 the following sensors and actuators are available:

- 2 buttons (BT1 + BT2)
- 4 switches (SW1 to SW4)
- 6 LEDs (P, Rx, Tx, U1, U2, U3)
- 1 light sensor (LDR)
- 1 potentiometer working as a variable resistor<sup>1</sup> (POT)


At the bottom of the front panel there is the power switch. Make sure to **turn the ECU on** when testing it. You can also turn it off and on again in order to **reset** the ECU.

## 2.2. Compiling and Building Your Project

In the practical, we will use the IDE<sup>2</sup> SPC5-Studio from STMicroelectronics<sup>3</sup> which is based on Eclipse<sup>4</sup>. At the practical session the IDE and a template project will already be prepared for you. You should see the IDEs main window as shown in Figure 6. The project files are located in the folder "Unit 1" on the practical PC's desktop.

On the left side of the window you will see a tree view with the structure of the project, i.e. header and source code as well as configuration files. As far as the practical is concerned only the files in the "source" folder need to be edited by you.

The project template's main.c file already provides a basic setup as well as function definitions you need to implement in order to solve the practical tasks.

Click on the build button in the toolbar (the icon is a hammer: ) in order to compile and build the current state of your project. Should there be any compilation errors, they will be shown at the bottom of the window in the "Problems" tab. Before you can run your project on the ECU any build errors have to be resolved.



In order to prepare for the practical at home, you can use the Virtual ECU software provided to you through Opal. All tasks of Unit 1 can be solved and tested with it. Please read the Virtual ECU documentation on how to set it up. With regards to how they are programmed the hardware ECU used in the practical and the Virtual ECU behave the same. This means you can bring your prepared source code to the practical for immediate testing.

<sup>1</sup> see <https://en.wikipedia.org/wiki/Potentiometer>

<sup>2</sup> Integrated Development Environment

<sup>3</sup> <https://www.st.com/en/development-tools/spc5-studio.html>

<sup>4</sup> <https://www.eclipse.org>

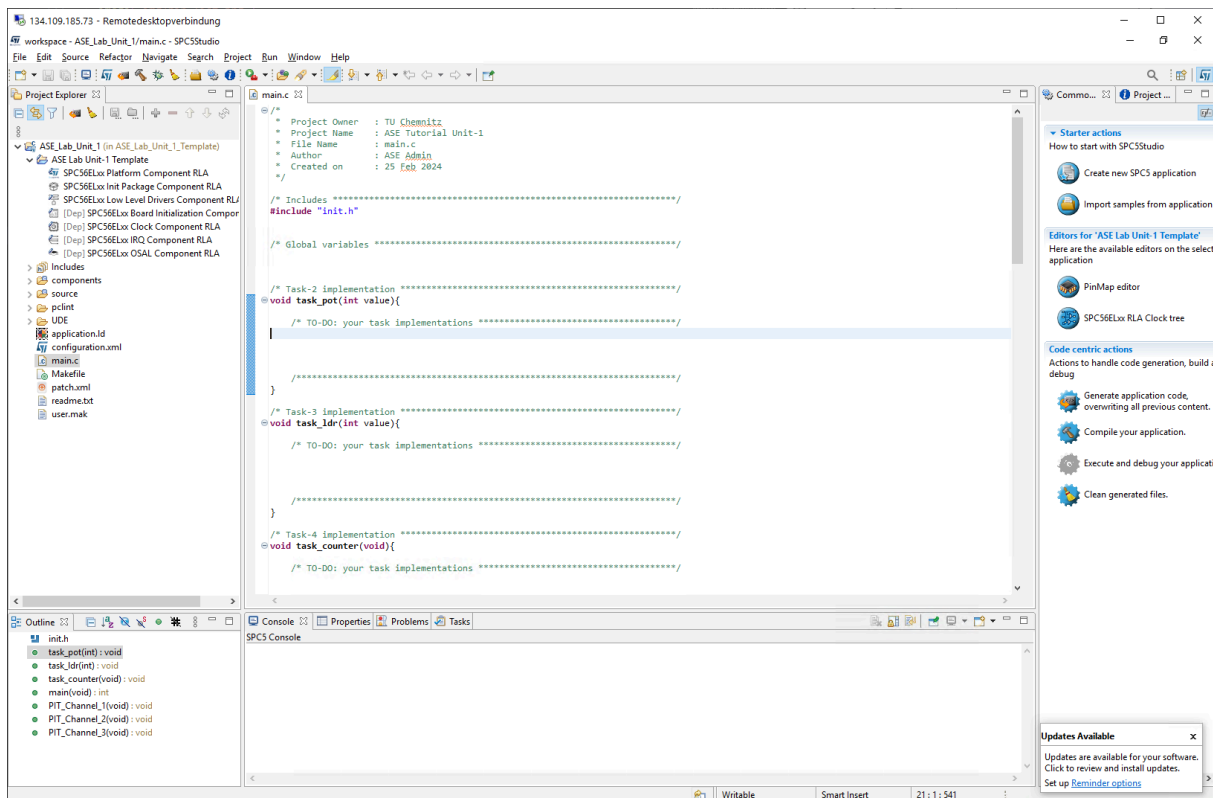


Figure 6: The main window of the SPC5-Studio IDE

## 2.3. Flashing Your Project to the ECU

Once you want to run and test your solution you have to flash it to the ECU. For this a separate flashing tool is needed: "UDE Starterkit 5.2". The flasher should already be opened on the practical PC and automatically catches when you rebuilt the program in SPC5-Studio. The flashing window in UDE looks as shown in Figure 7. Click on "Program all" in this window to flash your project to the ECU. When the process was successful **reset** the ECU by turning it off and on again. Then you can test by interacting with the front panel and observing the output (i.e. lighting of the LEDs).

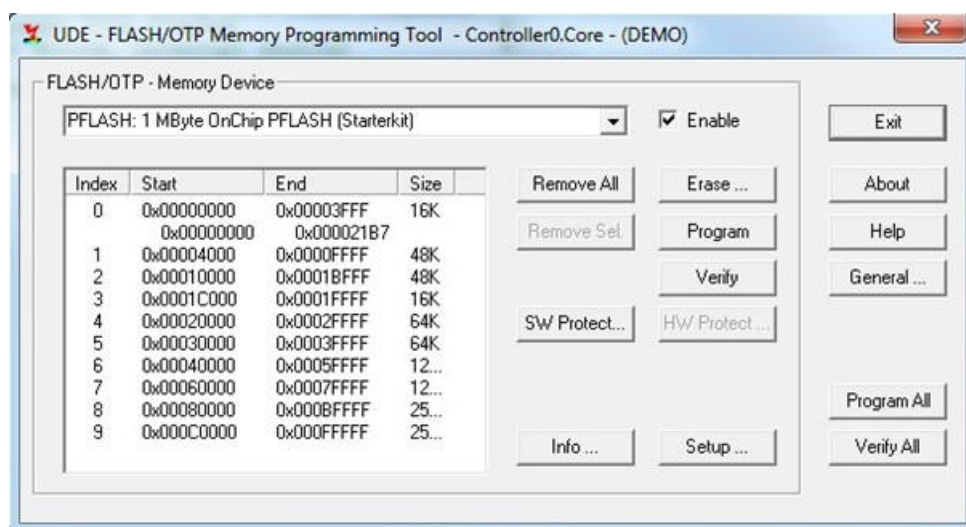


Figure 7: The flashing interface





Should you have any problems with flashing, the practical supervisors will guide you through the flashing process when you inform them.

### 3. Tasks

The following sections describe the tasks you should solve. Only continue to the next task once your current implementation is working. In general, each task will comprise of applying the correct configuration to the ECU and then implementing the described functionality. In the end, all 3 tasks should run on the ECU. The active task can be selected using the front panel switches.

#### 3.1. Pin Configuration

In the tasks you will need to configure pins in order to be able to access the ECU's peripherals (e.g. switches, LEDs). In order to configure pins for sensors (i.e. input pins) and actuators (i.e. output pins) refer to the provided document **pin\_muxing.pdf** to identify the correct PCR register associated with each port pin. Identify the corresponding IO port pin mapping of the used evaluation board from Table 2. For example, Rx LED corresponds to port pin D[9], and the PCR register number for this LED from the additional document is 57. In order to configure a pin:

- Extend the function "SIU\_Init()" in file "siu.c":
  - To configure a digital pin as **input** (e.g. for a button or switch) use **SIU.PCR[Y] = 0x100** where Y is the corresponding PCR number you looked up.
  - Likewise, to configure a digital pin as **output** use the same syntax but provide **0x200** as value.
  - For analog inputs (e.g. potentiometer and light sensor), write **0x2500** to the corresponding register.

To **set** an output pin to a specific value (e.g. to set an LED's state) you have to write it to the corresponding register: `SIU.GPDO[x].R = <value>;` where x is the corresponding PCR number. GPDO stands for General Purpose Digital **Output**.

To **read** the value of an input pin (e.g. to check the state of a switch) you have to access the corresponding register: `value = SIU.GPDI[x].R` where x is the corresponding PCR number. GPDI stands for General Purpose Digital **Input**.

The peripherals which need to be configured will be mentioned in the task descriptions. Table 2 shows the available peripherals, their corresponding pin mapping as well as what their values mean.

#### 3.2. Task 1 - Reading and Displaying the Potentiometer Value

The goal of this task is to display the current reading of the potentiometer (i.e. the voltage value the user set by the rotating the potentiometer's shaft) through the front panels LEDs. This means the potentiometer value (0 - 4095) has to be mapped to the 6 available LEDs as shown in Table 3. In addition to this, the functionality should only be active when **Switch 1 is ON**, otherwise all LEDs should be off. Consequently, you should proceed like this:

1. Configure the peripherals needed for this task (see Section 3.1):
  - All LEDs as output
  - SW1 as input
  - POT as analog input



Table 2: IO Port Pin Mapping

Peripheral	Description	Type	MCU Pin Name	Values
BT1	Button 1	Input	D[12]	1 - if pressed, 0 - not pressed
BT2	Button 2	Input	D[14]	1 - if pressed 0 - not pressed
SW1	Switch 1	Input	D[4]	1 - if turned on, 0 - turned off
SW2	Switch 2	Input	D[5]	1 - if turned on, 0 - turned off
SW3	Switch 3	Input	D[6]	1 - if turned on, 0 - turned off
SW4	Switch 4	Input	D[7]	1 - if turned on, 0 - turned off
P	Red LED	Output	D[8]	1 - on, 0 - off
Rx	Yellow LED	Output	D[9]	1 - on, 0 - off
Tx	Yellow LED	Output	D[10]	1 - on, 0 - off
U1	Green LED	Output	D[11]	1 - on, 0 - off
U2	Green LED	Output	C[11]	1 - on, 0 - off
U3	Green LED	Output	A[6]	1 - on, 0 - off
POT	Potentiometer	Analog input	E[2]	Current value = number between 0 (turned all to the left) and 4095 (turned all to the right)
LDR	Light sensor	Analog input	C[0]	Current value = number between 0 (bright) and 4095 (dark)

- Implement the mapping to LEDs in the function **task\_pot(int value)** in the project template's main.c file
- The task\_pot function needs to be called inside the infinite loop of the **main(void)** function which is the main entry of the program
  - Supply the current value of the potentiometer as its parameter
  - You can read it using `int pot_value = (ADC0.CDR[5].R & 0x0000FFFF);`
- The task\_pot function should only be called if SW1 is ON

### 3.3. Task 2 - Reading and Displaying the Light Sensor Value

Once Task 1 is working you should extend your solution to include the following:

Table 3: Mapping of analog input value to LED states

Value ( $x$ )	U1	U2	U3	Tx	Rx	P
$x < 682$	0	0	0	0	0	1
$682 \leq x < 1364$	0	0	0	0	1	1
$1364 \leq x < 2046$	0	0	0	1	1	1
$2046 \leq x < 2728$	0	0	1	1	1	1
$2728 \leq x < 3410$	0	1	1	1	1	1
$x \geq 3410$	1	1	1	1	1	1

The goal of this task is to display the current reading of the light sensor through the front panels LEDs. In order to change the sensor reading you both cover the sensor with something and use your phone's flashlight to illuminate it. As in Task 1 the light sensor value (0 - 4095) has to be mapped to the 6 available LEDs as shown in Table 3. In addition to this, the functionality should only be active when **Switch 1 is OFF** and **Switch 2 is ON**, otherwise all LEDs should be off. Consequently, you should proceed like this:

1. Configure the additional peripherals needed for this task (see Section 3.1):
  - SW2 as input
  - LDR as analog input
2. Implement the mapping to LEDs in the function **task\_ldr(int value)** in the project template's main.c file or re-use the **task\_pot** function in a smart way
3. The task\_ldr function needs to be called inside the infinite loop of the **main(void)** function which is the main entry of the program
  - Supply the current value of the light sensor as its parameter
  - You can read it using `int ldr_value = (ADC1.CDR[3].R & 0x0000FFFF);`
4. The task\_ldr function should only be called if SW1 is OFF and SW2 is ON

If everything is correct it should be possible to switch between the functionalities of Task 1 and Task 2 by setting Switch 1 and 2 appropriately while the ECU is running.

### 3.4. Task 3 - Implementing a Binary Counter

Once Task 1 and Task 2 are working you should extend your solution to include the following:

The goal of this task is to display a binary counter on the green LEDs (U1, U2 and U3) of the front panel. Consequently, as the 3 LEDs act as 3 bits we can display the numbers 0 to 7 of them ( $2^3 - 1$ ). **Every second** the value of the counter should be either increased or decreased based on the current mode. The current mode can be changed by using the buttons BT1 and BT2. If the user presses BT1 the counter should count up (counter value increases every second) whereas if the user presses BT2 the counter should count down (counter value decreases every second) from thereon. The default mode after starting the ECU should be counting up. The counter should **wrap around**, e.g. after exceeding 7 it should begin at 0 again and when the value would be below 0 it should continue with 7.



In order to execute timed operations independently from the code in the main function, on microcontrollers we can use so-called Periodic Interrupt Timers (PIT). Basically, these are separate components which are given a specific time after which they will issue an interrupt request (IRQ)<sup>5</sup>. Based on this the microcontroller will **pause** the currently executing code and jump to a previously registered interrupt service routine (ISR). In most cases, this will be a separate function in the source code which then gets executed. Since the normal operation of the microcontroller is paused while handling the IRQ, it is important that the code in the ISR runs as fast as possible so that the program can continue its normal execution (i.e. no long-running loops or sleeping). So in general, a PIT can be used to run a specific function after a pre-defined amount of time automatically<sup>6</sup>.

<sup>5</sup><https://en.wikipedia.org/wiki/Interrupt>

<sup>6</sup>Think of JavaScript's **setTimeout** function, only in hardware

The ECU used in the practical has 3 hardware timers (called timer channel 1, timer channel 2 and timer channel 3) which are available to you. They can be used in parallel and independent of each other. The ISRs are have been predefined for you in the project template and are called PIT\_Channel\_1, PIT\_Channel\_2 and PIT\_Channel\_3 accordingly.

Based on the description you should proceed like this to complete the task:

1. Configure the additional peripherals needed for this task (see Section 3.1):
  - BT1 and BT2 as input
  - SW3 and SW4 as input
2. Choose a timer channel and configure it
  - use the function `PIT_ConfigureTimer(timer_channel, interval)` to set the time in milliseconds after which the corresponding timer's ISR will be called
  - use the function `PIT_StartTimer(timer_channel)` to start the timer, e.g. its ISR will be called after is interval elapses
  - use the function `PIT_StopTimer(timer_channel)` to stop the timer, so that no more interrupts will be raised
3. Implement code for setting the LEDs based on the current counter value `task_counter()`
  - for this the counter value has to be converted to a 3-bit binary number as shown in Table 4
4. The `task_counter` function should only be called if SW1 is OFF, SW2 is OFF and SW3 is ON
  - However, the counter value itself should only be updated (increased or decreased) when SW4 is ON

If everything is correct it should be possible to switch between the functionalities of Task 1, Task 2 and Task 3 by setting Switch 1, 2 and 3 appropriately while the ECU is running.



The ISRs `PIT_Channel_X` functions are special routines. **You must not call them yourself in your code** as they will be called by the microcontroller after an IRQ has occurred.

Table 4: Binary counter function

Counter value	U1	U2	U3
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

## Bibliography

- [1] STMicroelectronics, "SPC56EL60x Family Reference Manual." [Online]. Available: <https://www.st.com/en/automotive-microcontrollers/spc56el60l5.html>