ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Prof. Lothar Thiele

# Embedded Systems - HS 2020

## Sample solution to Lab 2

Date : 14.10.2020

## Interrupts, Timers & Debugging

### Goals of this Lab

- Learn the difference between polling and interrupts
- Configure and implement hardware interrupts
- Learn how to debug code running on the microprocessor
- Configure and use hardware timers
- Understand and implement pulse-width modulation (PWM)

### Introduction

Embedded systems can interact with their surrounding environment in many different ways. Whether it is sensing or actuating, there are two basic mechanisms with which actions are initiated: event-triggered and time-triggered. In this lab, we learn how to use interrupts and timers, which are the building blocks to develop event-triggered and time-triggered functionality, respectively. Furthermore, we introduce basic debugging techniques.

A template for Lab 2 is provided as a ZIP file on the lecture's website. Table 1 provides an overview of the project structure. The document *DriverLib Userguide* which is required for this lab is contained in the `lab_documents.zip` file (`lab_documents` > `launchpad` > `msp432driverlib-userguide.pdf`) which can be downloaded from the lecture's website.

## Task 1: Interrupts and Debugging

**Table 1:** Project source files and folders of the `lab2` project. System startup code and linker files are omitted. The last column shows whether a file is modified during this lab.

| File/Folder Name | Description | Modified |
|---|---|---|
| `lab2.h` | Header file | – |
| `libraries` | DriverLib library | – |
| `main.c` | Contains the main() function | Task 1 & 2 |
| `task_1.c` | Template for Task 1 | Task 1 |
| `task_2.c` | Template for Task 2 | Task 2 |
| `uart.c/h` | UART interface initialization and communication | – |

## Interrupts

**Definition** A hardware interrupt is an electronic signal that alerts the microprocessor of an event. An interrupt can be triggered by either an internal peripheral (e.g. timer) or an external device (e.g. button).

**Why Interrupts?** In Lab 1, polling was introduced to detect when a button is pressed. When polling is used, the microprocessor repeatedly checks whether the event has occurred. In the case of a button, the value of the GPIO pin is read to determine whether it is high (released) or low (pressed). Once the button is pressed, the microprocessor immediatelly detects it during its next check, since it is constantly active and is doing nothing but checking this single condition. This behavior is represented by the left diagram in Figure 1. Although polling has a quick reaction time, it consumes an overproportional amount of energy since the system is always in the *active* state. Therefore, polling is only suitable for applications where the arrival time of events is known and/or if a condition has to be checked with a high frequency.

Another alternative is to configure an interrupt on the button's GPIO pin such that when a preconfigured trigger condition is met, an interrupt is generated. With this approach, the microprocessor can enter a low power *sleep* state and be woken up directly through the interrupt. The MSP432 can detect signal changes (e.g. rising or falling edges) to generate such an interrupt. If a GPIO pin is configured to be pulled up, for example, a falling edge would occur when the button is pressed as it pulls the pin to ground. The behavior with interrupts can be seen in the right diagram in Figure 1. It should be noted that although using interrupts can be energy-efficient, it *does* introduce a delay, since the transition from the *sleep* state to the *active* state requires some time. Interrupts are thus better suited to handle asynchronous or infrequent events.

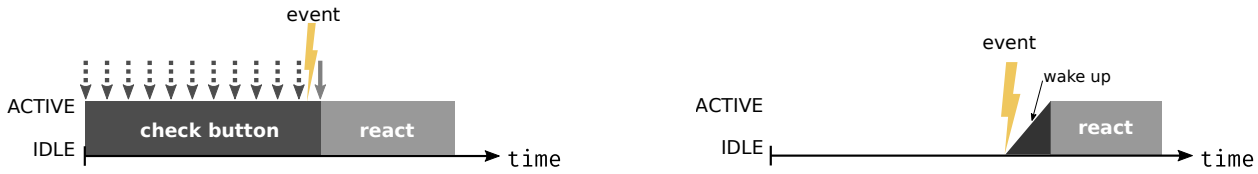


**Figure 1:** Detecting an event with polling (left) and with hardware interrupts (right).

**System States** Microprocessors have multiple states, such as *active* and *sleep*, which are essential for power management. Depending on the application, the transition between these states can be time-triggered or event-triggered. In both cases, an interrupt initiates the transition from the *sleep* to the *active* state. The only difference is the interrupt source (peripheral). In the case of an event trigger, this can be a GPIO pin. For a time trigger, a timer or a real-time clock is typically used. Figure 2 and Figure 3 show the generic state diagrams for these two systems. To enter the *sleep* state, it is sufficient to tell the Power Control Module (PCM) to go into a low power mode; e.g. by calling `PCM_gotoLPM3()`, the chip enters low power mode 3. Depending on the low power mode level, the microprocessor will turn off a set of peripheral and internal functions to save energy. Then, it will halt (sleep) until an interrupt occurs. When the interrupt is received, it will switch to the *active* state and execute the corresponding interrupt service routine (ISR). Afterwards, it will continue with the program execution in the `main()` function.

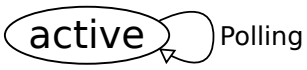


**Figure 2:** Polling: The system typically remains in the *active* state to detect an event.

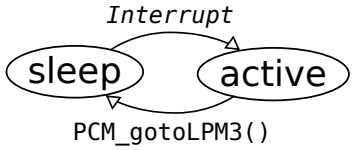


**Figure 3:** Interrupt: The system can remain in the *sleep* state until an event occurs.

**Basic Operation** Once a correctly configured system is deployed, the interrupt flows as depicted in Figure 4. First, an event needs to occur (i.e. a button is pressed) such that a signal change is detected. The interrupt flag (IFG) register has a bit assigned to each configured interrupt peripheral. Each time the interrupt occurs, the corresponding IFG bit is set. Before the microprocessor is informed about the event, the signal must fulfill three conditions. First, the pin must be configured to trigger an interrupt. Secondly, the corresponding interrupt must be enabled. If these conditions hold, the global interrupt enable (GIE) register is tested. Only if global interrupts are enabled, the microprocessor will be preempted and the according interrupt service routine (ISR; basically an assigned function in the program code) be executed. This global switch is useful to temporarily turn off all[1] interrupts in order to prevent the modification of a global variable or to avoid preemption between instructions that belong together logically. The *interrupt vector table* maps the interrupts to the correct ISR.

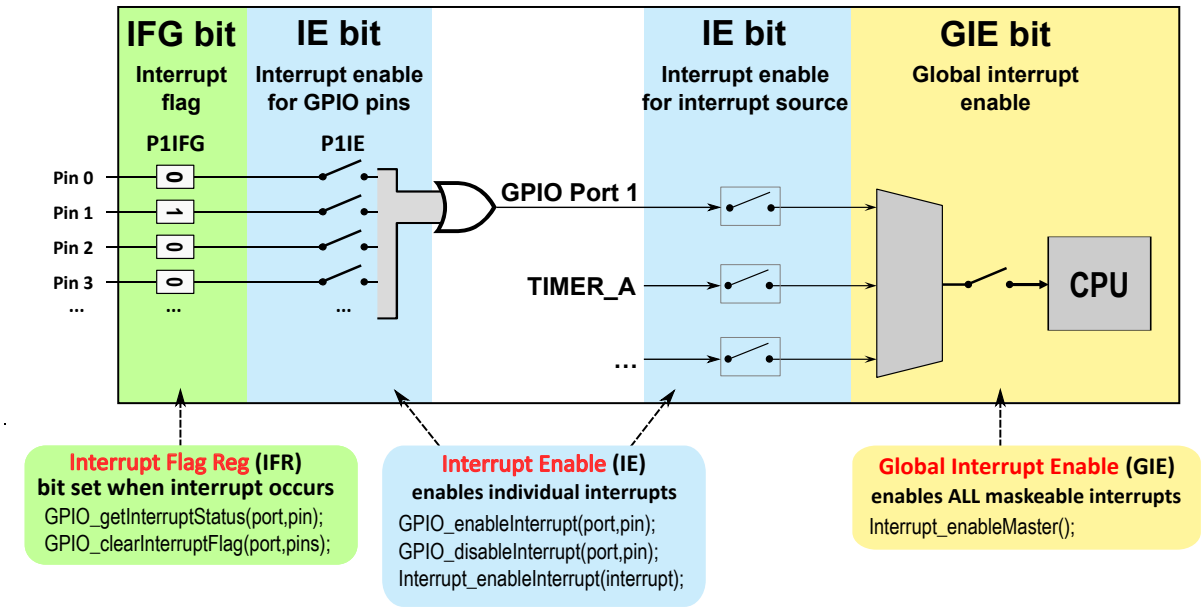


**Figure 4:** Both individual and global interrupts must be enabled for the microprocessor to detect an interrupt.

Depending on the source of the interrupt, either a *dedicated* or a *grouped* interrupt is triggered. For simple peripherals like a comparator, there is only one possible event that generates the interrupt: crossing the comparator's threshold. In the case of other peripherals, like GPIO ports, multiple pins could generate the same interrupt. In these cases, it is necessary to query the pin's interrupt vector register in order to identify the interrupt's exact source. Typically, this is done inside the ISR.

Once an ISR identifies the source of the interrupt, it can react in accordance. Typically, ISRs execute in a privileged mode which can, among other things, mask (i.e. suppress) other interrupts. For this reason, they should be as short as possible and only set application-specific flags to indicate to the microprocessor's main thread that it should execute the corresponding task in response to an interrupt.

Each interrupt type has a priority to uniquely define which interrupt is handled first in case multiple interrupts occur at the same time. Furthermore, it is possible to allow nested interrupts, i.e. that one interrupt interrupts the handling of another interrupt. However, it's not recommended to use this feature since it's complex to program, debug and test software with nested interrupts.

**Configuring an Interrupt** By default, all interrupts are disabled. The following steps are necessary to configure a GPIO interrupt on the MSP432.

[1] Actually, there are a few interrupts that bypass the GIE bit and are therefore not disabled if the GIE bit is set to zero. These interrupts are also called non-maskable interrupts (NMI).

1. Configuring the GPIO pin as an input.
   `GPIO_setAsInputPinWithPullUpResistor()`
2. Selecting the edge (clock signal) which will trigger the interrupt.
   `GPIO_interruptEdgeSelect()`
3. Clearing the pin's interrupt flag. This makes sure that no previous interrupts are handled. (This step is not mandatory, but it's good practice to do so.)
   `GPIO_clearInterruptFlag()`
4. Setting the interrupt enable (IE) bit of the specific GPIO pin (enabling the interrupt in the peripheral).
   `GPIO_enableInterrupt()`
5. Setting the interrupt enable (IE) bit of the corresponding interrupt source (enabling the interrupt in the interrupt controller).
   `Interrupt_enableInterrupt()`
6. Enabling interrupts globally (set global interrupt enable (GIE) bit).
   `Interrupt_enableMaster()`

After these steps, any new event on the pin will generate an interrupt, preempt the microprocessor, and start the execution of the corresponding ISR.

**Defining an Interrupt Service Routine (ISR)** The Interrupt Service Routine (ISR) is the part of code which is executed if the processor detected the corresponding interrupt. In general, a programmer has the flexibility to choose any arbitrary function to be an ISR. There are a few restrictions on the function; since it cannot receive arguments or return values, it can only modify global variables.

The DriverLib provides predefined ISR's that are already associated with an interrupt source. For example, the predefined interrupt handler for Port 1 is `void PORT1_IRQHandler(void)`.

Snippet 1 shows a sample program that uses an infinite loop (line 6) in the main thread and an ISR to detect button presses. In the infinite loop, the program switches to the *sleep* state in order to save energy while waiting for an interrupt to occur (line 7). When an interrupt is received, the processor executes the corresponding ISR (line 15). For all possible interrupt sources, the DriverLib provides a corresponding ISR which is already set up such that it is executed once the interrupt is received. Since the GPIO pins are connected to a grouped interrupt, the first step inside the ISR is to figure out which pin actually triggered the interrupt. For this, the port's interrupt status register is queried by using the DriverLib function `GPIO_getEnabledInterruptStatus(GPIO_PORT)` (line 16). By doing a bitwise AND with the bitmask of a specific GPIO pin `GPIO_PIN`, the pin that triggered the interrupt can be determined (line 17). The DriverLib provides a bitmask for every pin. If the result is true, then the corresponding pin was the source of the interrupt. Once the interrupt source is identified, an application-specific flag can be set (line 18). A global variable is used to communicate this information between the ISR and the main thread (line 1). This way, the main application thread can take the appropriate action (line 9). It is important that the ISR clears the flag of every interrupt handled (line 19). If the flag is not cleared, future interrupts from that source will not be distinguishable. Of course, the application-specific flag needs to be cleared too after executing the associated task (line 10).

## Debugging

Usually, a debug probe in the form of an additional device is connected to the microcontroller in order to debug the microcontroller at runtime. The MSP-EXP432P401R Development Board, used in this lab, features a built-in debug probe. The spatial separation of the debug probe together with the energy trace

```
volatile bool buttonFlag = false;

void main() {
  configureGpioAndInterrupt();

  while(1) {
    PCM_gotoLPM3(); // Go to low power mode and wait for interrupt
    if(buttonFlag == true) {
      togglePin(LED_PORT, LED_PIN);
      buttonFlag = false;
    }
  }
}

void BUTTON_PORT_IRQHandler(void) {
  status = getEnabledInterruptStatus(BUTTON_PORT);
  if(status & GPIO_PIN) {
    buttonFlag = true;
    clearInterrupt(BUTTON_PORT);
  }
}
```

**Snippet 1:** Pseudocode example that uses a pin interrupt to toggle an LED.

component and the programmer from the MSP432 microcontroller is indicated by the dashed white line on the MSP-EXP432P401R. In this lab, we use the Serial Wire Output (SWO) trace which is a set of tools based on ARM hardware components.

### Task 1.1: Interrupts with Buttons

The goal of this task is to setup interrupts such that pressing button S1 toggles the green LED and pressing button S2 toggles the blue LED.

- Start the Code Composer Studio (CCS) from the Terminal by typing `ccstudio` and hitting enter.
- Import the template CCS project (`lab2.zip`) into the CCS by performing a right-click inside the project explorer. Use the *Select archive file* option in the Import dialog.
- Make sure that only the `task1()` method is uncommented in the main function in `main.c` file. Use the provided template `task_1.c` to implement the subtasks of task 1.
- Get an overview of the existing GPIO (`GPIO_xxx`) and Interrupt (`Interrupt_xxx`) methods. For this, have a short look in the ***DriverLib Userguide* (p. 117 and 181)**. A selection of relevant methods are listed in the Section *Configuring an Interrupt*.
- First, configure the green and blue LEDs of LED2 as output (Placeholder 1). Also make sure that they are turned off whenever the LaunchPad is reset (Placeholder 5). (If you can't remember the DriverLib commands from Lab1, look up the GPIO_xxx methods in the *DriverLib Userguide*.)
- Configure the GPIO pins that are connected to the buttons S1 and S2 of the Launchpad as input with pull-up resistor (Placeholder 2).
- Setup the interrupts by selecting the edge and by clearing the interrupt flags. Select the edge such that an interrupt is triggered when the button is released (Placeholder 3).
- Enable the interrupts for S1 and S2. For this, enable interrupts for (1) the corresponding GPIO pins (peripheral), (2) the port (interrupt controller) and (3) globally (Placeholder 4).
- Implement the `PORT1_IRQHandler` interrupt service routine (ISR) such that the green or the blue LED of LED2 (RGB LED) is toggled and the corresponding flag variable (`button1Flag` or `button2Flag`) is set to `true`. In the template, we provide Pseudocode 1 in order to help you to structure your code. In addition, we already assigned the content of Port1's status register to the

variable `status`. Use the bit masks `GPIO_PIN1` and `GPIO_PIN4` provided by the DriverLib to figure out which button was pressed. S1 should toggle the green LED whereas S2 should toggle the blue LED.

- Compile and run (debug) your program.
- Press the buttons S1 and S2 and verify the resulting action (LEDs and UART). Verify that the interrupt is triggered when the button is released, not when it is pressed. As in Lab 1, you can use the terminal in the debug view of Code Composer Studio (CCS) to display the UART output (baudrate: 115200).

**Solution for Task 1.1:**
See Snippet 2 and Snippet 3.

```
//// Placeholder 1 (Task 1.1) ////////////////////////////////////////////////
/* Configuring P2.1 (LED2 green) and P2.2 (LED2 blue) as output */
GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN1);
GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN2);
//////////////////////////////////////////////////////////////////////////////

//// Placeholder 2 (Task 1.1) ////////////////////////////////////////////////
/* Configuring P1.1 (Button S1) and P1.4 (Button S2) as an input */
GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);
GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN4);
//////////////////////////////////////////////////////////////////////////////

//// Placeholder 3 (Task 1.1) ////////////////////////////////////////////////
/* Configure interrupts for buttons S1 and S2 */
// Configure P1.1
GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1, GPIO_LOW_TO_HIGH_TRANSITION);
GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
// Configure P1.4
GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN4, GPIO_LOW_TO_HIGH_TRANSITION);
GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN4);
//////////////////////////////////////////////////////////////////////////////

//// Placeholder 4 (Task 1.1) ////////////////////////////////////////////////
/* Enable interrupts for both GPIO pins */
GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1);
GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN4);
/* Enable interrupts on Port 1 */
Interrupt_enableInterrupt(INT_PORT1);
/* Enable interrupts globally */
Interrupt_enableMaster();
//////////////////////////////////////////////////////////////////////////////

//// Placeholder 5 (Task 1.1) ////////////////////////////////////////////////
/* Initially turn LEDs off */
GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);
GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN2);
//////////////////////////////////////////////////////////////////////////////

```

**Snippet 2:** Interrupt setup (Task 1.1).

## Task 1.2: Breakpoints and Stepping Through Code

In this task, you are going to use the IDE's debugger functionality to step through the code while it is being executed on the Launchpad. Furthermore, you will learn how to work with breakpoints. We re-use the code of Task 1.1 as a sample program.

- Start to debug the working program from Task 1.1. Do not press the *Resume* (Start) button yet.

```c
void PORT1_IRQHandler(void)
{
  uint32_t status;

  do {
    /* Get the content of the interrupt status register of Port 1 */
    status = GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);

    if (status & GPIO_PIN1)
    {
      /* Toggle green LED of LED2 */
      GPIO_toggleOutputOnPin(GPIO_PORT_P2, GPIO_PIN1);
      /* Set the flag to inform the while loop that the button has been pressed */
      button1Flag = true;
      GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
    }
    else if (status & GPIO_PIN4)
    {
      /* Toggle blue LED of LED2 */
      GPIO_toggleOutputOnPin(GPIO_PORT_P2, GPIO_PIN2);
      /* Set the flag to inform the while loop that the button has been pressed */
      button2Flag = true;
      GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN4);
    }
  } while(status);
}
```

**Snippet 3:** ISR (Task 1.1).

- In the editor, set a breakpoint on the line which is responsible for toggling the green LED of LED2 in the `task_1.c` file. You can set a breakpoint by double clicking on the line number in the code editor (see Figure 5).
- Now, start the execution of the program with the *Resume* (Start) button in the IDE.
- Press the push button S1 on the Launchpad. Why does the green LED not turn on?
- In the code editor, you see that the program execution is paused and the processor halted at the line with the breakpoint. If you hover the mouse over a variable which is set prior to the line with the breakpoint, you can see the current value of that variable. Hover over the counter variable for counting the interrupts for S1 and note the value.
- Now click the *Step Over* button of the IDE (see Figure 6) multiple times and observe the program execution in the program editor. Compare the value of the counter variable before and after the incrementation of the counter variables. Also check the state of the LED on the LaunchPad.
- The execution can be resumed by pressing the *Resume* button in the IDE (see Figure 6).
- When you are finished with debugging the program, make sure to disable all breakpoints again such that the program runs without suspending in the following tasks.

**Solution for Task 1.2:**
If the program is executed without any breakpoints, a push button press would toggle the corresponding LED. However, with a breakpoint at the line of the command to toggle the LED, the program is execution is suspended just before executing the command. Therefore, the LED is not turned on after pressing a push button. By using either the *Step Over* or *Resume* button in the IDE, the program execution is continued such that the chip toggles the LED. Therefore, the LED is turned on after resuming/stepping over.

### Task 1.3: Interrupt Trace [Optional Task]

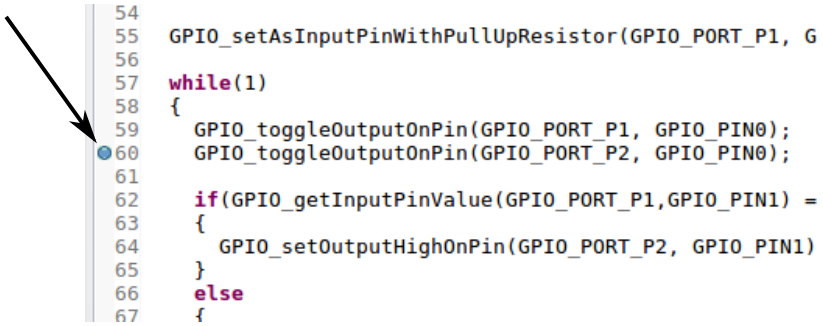


**Figure 5:** Setting a breakpoint by double clicking the line number in the CCS editor.

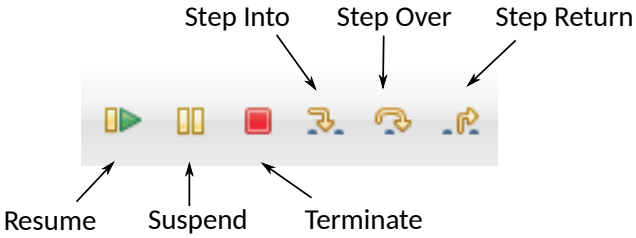


**Figure 6:** Buttons for debugging.

In this task, you will record a trace of all interrupts that occur while the program is executed on the MSP-EXP432P401R. The trace contains the source of the interrupts and the timestamps when the interrupts happened. As a sample program, we use the program of Task 1.1.

- Again use the working program from Task 1.1.
- Start to debug the program, but do not yet press the *Resume* (Start) button.
- Start to collect an interrupt trace by selecting `Tools > Code Analysis > Event Analysis (hardware events and messaging)` from the menubar (see Figure 7). When selecting *Event Analysis (hardware events and messaging)*, a configuration window will pop up[2]. You should select `Exception Profile` and keep the other default configuration (e.g. `SWO Trace`). Also note that `COM Port` should be valid, and just click `OK` (see Figure 8).
  Note: The *Tools* menu is only available in the debug view of CCS.
- Click `Start Capture` in the *Event Analysis* window to start capture the hardware events (also interrupts here). Then, use the *Resume* button in the IDE to start the program. It is very possible that your CCS as well as VM Virtual Box will crush several times.
- Now generate some interrupts by pressing the buttons of the Launchpad.
- Pause the execution of the program with the *Suspend* button in the IDE (see Figure 6).
- Select `Exception Profile` in the *Event Analysis* window (right bottom corner of the IDE). You should now have a new plot that contains the recorded interrupt trace data (see Figure 9):
  - **Event Type** All the timestamps and the type (e.g. enter, exit) of the interrupt events can be viewed by hovering the mouse over the corresponding events. Usually, you need to zoom out in order to see events.
  - **Exception Statistics** Displays statistic informations of all events, e.g. name (PORT1), Occurence Count (13), etc., see Figure 9.

[2]If the configuration window does not pop up, close CCS and start it again by executing the `ccstudio` command in the terminal

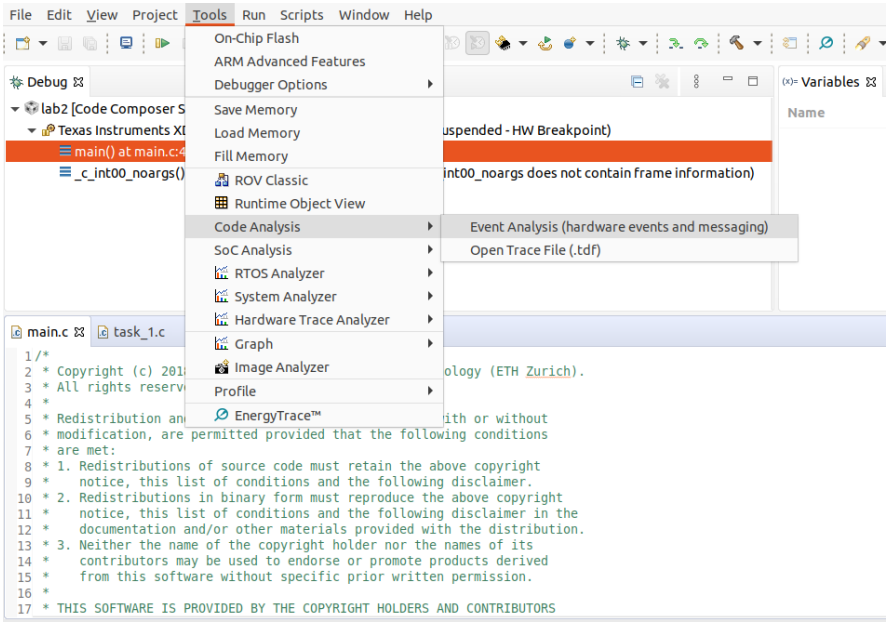


**Figure 7:** Code Analysis Menu Option.

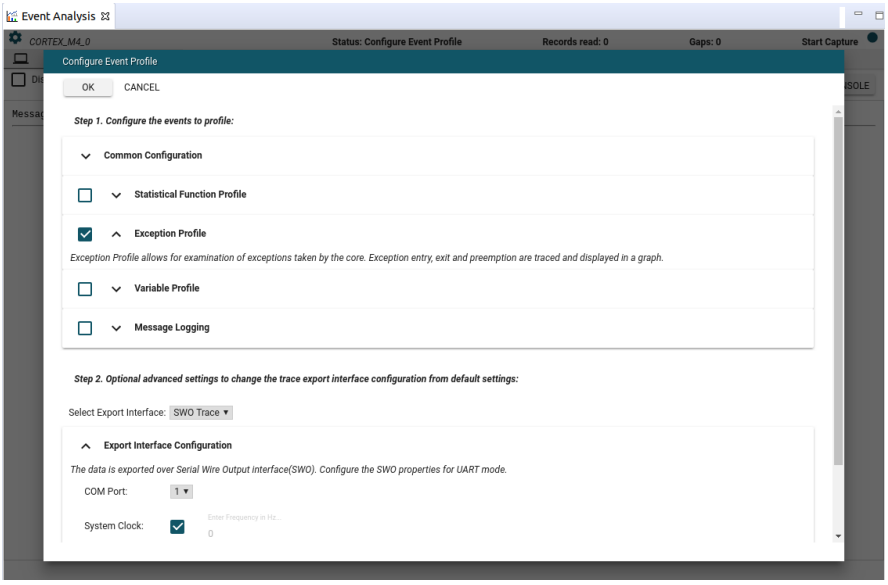


**Figure 8:** Event Analysis Configuration.

---

# Task 2: Timers and PWM

## Timers

In order to start time-triggered actions after specific time intervals, microcontrollers usually feature one or multiple types of timers. The different types differ for example in their power consumption, counter register width (typically 16 or 32 bits) and feature set. For some types of timers, multiple identical instances are available. In this lab, we only use Timer_A which is highlighted in Figure 10. Timer_A0


**Figure 9:** Exemplary trace of interrupts.

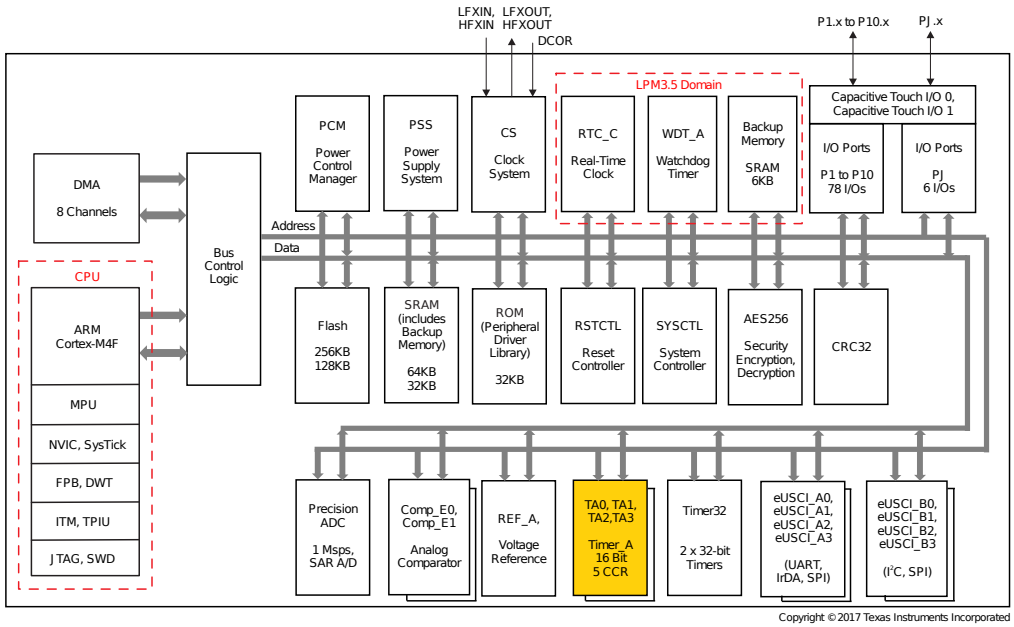


**Figure 10:** Timer_A is an internal peripheral of the MSP432P401x.
(source: `lab_documents.zip` > `launchpad` > `msp432p401r-datasheet.pdf`, p. 3)

(abbreviated TA0) is the first instance (instance 0) of Timer_A.

**Basic Operation** The structure of a timer is depicted in Figure 11. Its main component is a counter. A counter is a register whose value is incremented with every rising (or falling) edge of the clock input. The timer can be configured to trigger an interrupt every time the counter register rolls over (i.e. when it resets to 0 after reaching the highest counter value). Furthermore, it is possible to select different clock sources and to divide the clock frequency by a constant factor. This is useful to achieve larger intervals between roll-overs without increasing the size of the counter register. However, this reduces the granularity since the counter value is incremented less frequently.

Timer_A in the MSP432 can be in one of four modes:

- **Halted:** The timer is stopped and the counter value is not incremented.
- **Continuous:** The counter is continuously incremented and the counter value is reset to 0 only if the counter register rolls over.

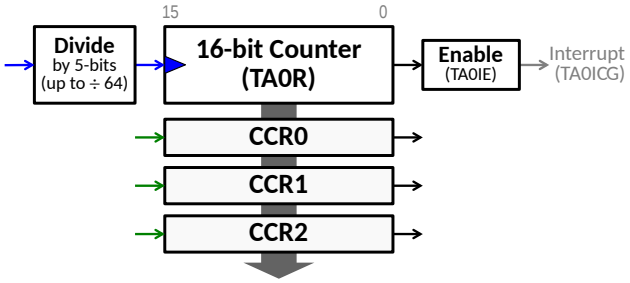


**Figure 11:** The structure of a timer.

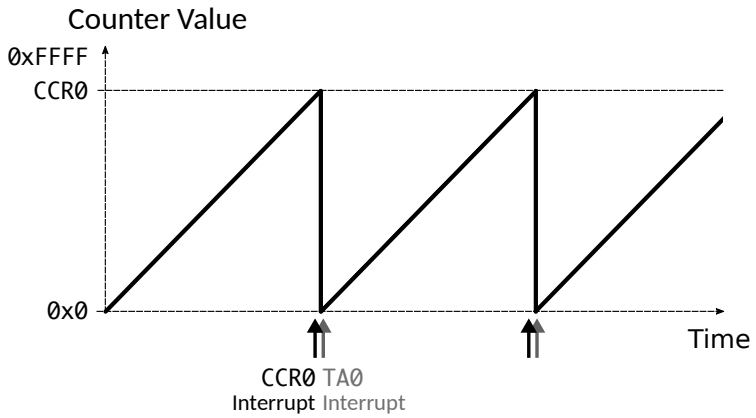


**Figure 12:** A timer in UP-mode and the corresponding interrupts.

- **Up:** The Up-mode is visualized in Figure 12. This is the mode which is used in this lab. In this mode, the counter value is continuously incremented up to a value defined in the first capture and compare register (CCR0). The counter is reset to 0 each time the value in CCR0 is reached. In this mode, two separate interrupts are generated which are only 1 cycle apart (if enabled). First the CCR0 register generates an interrupt when the maximum value has been reached (CCR0 Interrupt). One cycle later, the timer generates an interrupt because the timer is reset to 0 (TA0 Interrupt).
- **Up/Down:** The counter value is incremented up to a value defined in the first capture and compare register (CCR0). Every time the counter reaches the value in the CCR0, the counting direction is reversed and the counter is counting down. The counting direction is also reversed if the counter reaches 0. The counter therefore alternately counts up and down.

**Capture and Compare Features** Most timers provide the following two features:

- **Input capture:** As depicted in Figure 13, the counter value is copied to a separate register when an event at one of the CCR inputs occurs. The same event that causes the timer value to be captured can also trigger new timer events such as a timer interrupt, sending a signal to a peripheral or modifying a pin.
- **Output compare:** Figure 14 shows the output compare feature. At every incrementation of the counter register, the counter value is compared to the compare register. Certain conditions (e.g. equality) can trigger an event such as a timer interrupt, sending a signal to a peripheral or modifying a pin.

Usually, one register of a timer can only be configured to provide a single functionality: capturing the timer value or comparing a value. Such a register is therefore called *Capture and Compare Register (CCR)*. Oftentimes, a timer has multiple capture and compare registers which are labeled CCRx, where

x is the instance of the CCR. Each CCR register has its corresponding pin. In capture mode, this pin can be used as an input. In compare mode, the pin represents an output and its behavior (*compare output mode*) can be configured in many ways (for more details see Figure 17-12 in the MSP432 Reference Document `lab_documents.zip` > `launchpad` > `msp432p4xx-reference.pdf`).

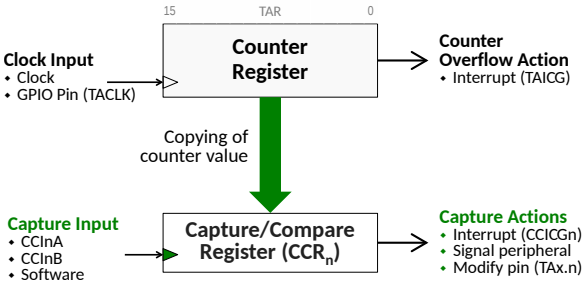


**Figure 13:** Input capture functionality of a timer.

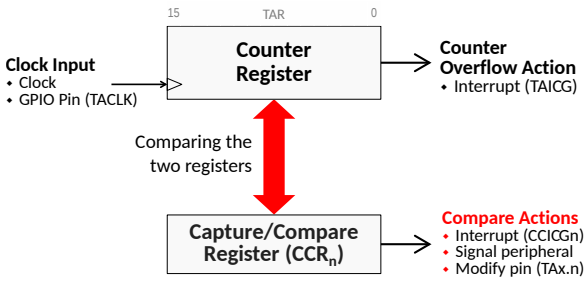


**Figure 14:** Output compare functionality of a timer.

**Configuring a Timer** The following steps are required to configure and start a timer in output compare mode:

1. Setting up the counter (e.g. in Up-mode):
   `Timer_A_configureUpMode()`
   - Selecting a clock source.
   - Selecting a divider to reduce the clock frequency.
   - Enabling/disabling interrupts when the counter rolls over (i.e. when it resets to 0).
   - Clearing the counter value in the counter register (if required).
2. Setting up capture and compare registers (CCRs):
   `Timer_A_initCompare()`
   - Selecting the compare-to value (i.e. selecting the intervals).
   - Configuring the output compare mode.
   - Enabling/disabling the interrupt on compare.
3. Clearing interrupt flags (if any timer interrupt has been enabled):
   `Timer_A_clearCaptureCompareInterrupt()` and/or
   `Timer_A_clearInterruptFlag()`
4. Starting the timer:
   `Timer_A_startCounter()`

## Task 2.1: Periodic Execution

In this task, you will make use of the hardware timer Timer_A of the MSP432 to periodically generate interrupts. The interrupt is used to toggle LED1.

- Make yourself familiar with the Timer (`Timer_xxx`) functions in the ***DriverLib Userguide*** on **page 312**.
- Make sure that only the `task2()` method is uncommented in the `main.c` file.

- Use the `upConfigA0` struct in the template `task_2.c` to configure Timer_A0 in Up-mode. The available arguments are listed in the *DriverLib Userguide* (DriverLib method `Timer_A_configureUpMode()`). Use the following settings:
  - Use the SMCLK clock source to drive the timer.
  - Determine the divider and the CCR0 value of Timer_A0 such that the timer generates one interrupt every second. (Hint: SMCLK runs at a frequency of 3 MHz.)
  - Disable timer roll-over interrupts.
  - Use the CCR0 compare interrupt to trigger the toggling of the LED (enable the interrupt of the peripheral).
  - Clear the value in the counter register.

  CCR0 is special as it is used for setting up the timer in Up-mode. Therefore, it is configured inside the same struct which is used to configure the Up-mode of the timer. CCR0 is the only register that can be configured this way when using the DriverLib. For other CCRs, the compare mode needs to be configured separately (e.g. with `Timer_A_initCompare()`).
- Configure LED1 as output (Placeholder 1).
- Configure Timer_A0 in Up-mode by using the DriverLib method `Timer_A_configureUpMode()` (Placeholder 2). Use the previously defined struct `upConfigA0` and pass it by reference, i.e. use the `&` operator.
- Clear the interrupt flag of the CCR0 by using the DriverLib method `Timer_A_clearCaptureCompareInterrupt()` (Placeholder 3).
- Enable the interrupt from the timer in the interrupt controller (set the interrupt enable (bit) of the corresponding interrupt source (`INT_TA0_0`) with `Interrupt_enableInterrupt()`).
- Enable interrupts globally (Placeholder 3).
- Use the DriverLib method `Timer_A_startCounter()` to start the timer in Up-mode (Placeholder 3).
- In the interrupt service routine `TA0_0_IRQHandler`, toggle the LED1 (Placeholder 5) and clear the interrupt flag of CCR0 (Placeholder 9). For the second part, you can use the DriverLib method `Timer_A_clearCaptureCompareInterrupt()`.
- Build and run (debug) your program.
- Verify that LED1 is toggled every second.

**Solution for Task 2.1:**
In general, the following equation must hold:

$$T = \frac{\#\text{ticks} \cdot \text{divider}}{f_c}$$

$T$ is the requested timer interrupt period, #ticks is the number of timer ticks (counter increments), divider is the divider value for dividing the clock frequency $f_c$. We know that $T = 1.0\,\text{sec}$ and $f_c = 3\,\text{MHz}$. The goal is now to find a combination of values $\{\#\text{ticks}, \text{divider}\}$ s.t. the above equation holds.

- Since we only have a finite number of possible divider values, we first need to find the closest one.
- We use the maximum number of counter increments (ticks) which is $2^{16} - 1$ since we are using a 16-bit counter.
- divider $\approx \frac{T \cdot f_c}{\#\text{ticks}} = \frac{1.0 \cdot 3 \cdot 10^6}{2^{16}-1} \approx 45.8$
- Since we used the maximum #tick value, we need to use a divider value larger than $45.8$: only the values 48, 56 and 64 are possible.

- By solving the equation for #ticks, we can calculate the corresponding #ticks values: $\#\text{ticks} = \frac{T \cdot f_c}{\text{divider}}$

The possible solutions for this task are:

- $\{\text{divider} = 48, \text{CCR0} = 62'500\}$
- $\{\text{divider} = 56, \text{CCR0} = 53'571\}$
- $\{\text{divider} = 64, \text{CCR0} = 46'875\}$

Both the divider and CCR0 value need to be set in the `upConfigA0` struct. In the sample solution, the macro `TA0_CCR0` is defined to be $46'875$ (maximal divider chosen to also enable $T > 1.0\,\text{sec}$ in the future).

The code for Task 2.1 is listed in Snippet 4 and Snippet 5.

```
/* Timer_A0 up mode configuration parameters */
const Timer_A_UpModeConfig upConfigA0 =
{
  TIMER_A_CLOCKSOURCE_SMCLK,           // SMCLK clock source (3MHz)
  TIMER_A_CLOCKSOURCE_DIVIDER_64,      // SMCLK/64 = 46.875kHz
  TA0_CCR0,                            // Value in CCR0, NOTE: limited to 16 bit!
  TIMER_A_TAIE_INTERRUPT_DISABLE,      // Disable timer roll-over interrupt
  TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,  // Enable CCR0 interrupt
  TIMER_A_DO_CLEAR                     // Clear value in the timer counter register
};

//// Placeholder 1 (Task 2.1) ////////////////////////////////////////////////
/* Configuring P1.0 (LED1) as output */
GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0);
//////////////////////////////////////////////////////////////////////////////

//// Placeholder 2 (Task 2.1) ////////////////////////////////////////////////
/* Configuring Timer_A0 for Up mode */
Timer_A_configureUpMode(TIMER_A0_BASE, &upConfigA0);
//////////////////////////////////////////////////////////////////////////////

//// Placeholder 3 (Task 2.1) ////////////////////////////////////////////////
/* Clear interrupt */
Timer_A_clearCaptureCompareInterrupt(TIMER_A0_BASE,
                                     TIMER_A_CAPTURECOMPARE_REGISTER_0);
/* Enable interrupt */
Interrupt_enableInterrupt(INT_TA0_0);
/* Enable interrupts globally */
Interrupt_enableMaster();
/* Start the Timer_A0 */
Timer_A_startCounter(TIMER_A0_BASE, TIMER_A_UP_MODE);
//////////////////////////////////////////////////////////////////////////////
```

**Snippet 4:** Timer configuration for generating periodic interrupts (Task 2.1).

### Task 2.2: Pulse-Width Modulation Using a Timer

In this task, you will see how Pulse Width Modulation (PWM) works and how we can make use of Timer_A on the MSP432 to generate a PWM signal (depicted in Figure 15).

The idea of Pulse Width Modulation (PWM) is to switch on and off a consumer at a high frequency. In this task, we use an LED as a consumer. The amount of ON-time relative to the amount of OFF-time (also called duty-cycle) determines how much power is delivered to the LED. Because the LED is switched on and off at a high frequency, the human eye cannot distinguish between on and off and only perceives a constant intensity level. The brightness of the LED corresponds to the average delivered power ($P_{\text{avg}}$)

```c
void TA0_0_IRQHandler(void)
{
  //// Placeholder 5 (Task 2.1) ////////////////////////////////////////////
  /* Toggle LED1 */
  GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
  //////////////////////////////////////////////////////////////////////////

  //// Placeholder 9 (Task 2.1) ////////////////////////////////////////////
  /* Clear interrupt of timer A0 */
  Timer_A_clearCaptureCompareInterrupt(TIMER_A0_BASE,
                                       TIMER_A_CAPTURECOMPARE_REGISTER_0);
  //////////////////////////////////////////////////////////////////////////
}

```

**Snippet 5:** ISR (Task 2.1).

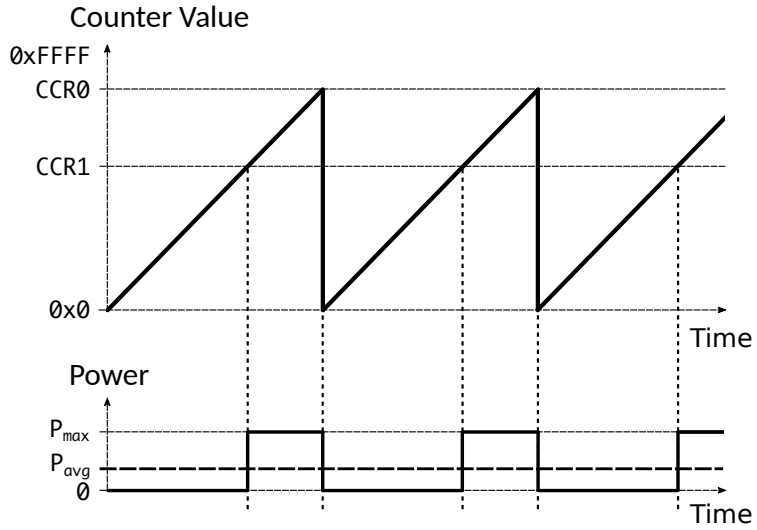


**Figure 15:** Generation of a PWM signal with a timer in UP-mode.

and therefore to the duty-cycle. This means that it is possible to set the brightness of the LED by a parameter in software. The same principle can also be used to drive other consumers such as motors.

Compared to the previous task, we do not use the timer to generate an interrupt, but we use the output of the output compare functionality. By default, this output is connected to pin P2.4, however the template `task_2.c` makes sure that the output of the CCR1 of the second instance of Timer_A (`TA1`) is mapped to port P2.0 (red LED of LED2).

- Continue to use template `task_2.c`.
- Use the `upConfigA1` struct in the template `task_2.c` to configure Timer_A1 in Up-mode. The available arguments are listed in the *DriverLib Userguide* (DriverLib method `Timer_A_configureUpMode()`). Use the following settings:
  - Use the SMCLK clock source to drive the timer.
  - No divider should be used (`DIVIDER_1`).
  - Use `PWM_CCR0` which is provided in the template as the value for CCR0. With this, the timer is configured to have a period of $128$ (`PERIOD`$=128$) clock cycles. Because the timer starts counting at $0$, the maximum counter value is equal to 127 (`PWM_CCR0`$=127$).
  - Disable timer roll-over interrupts.
  - Disable CCR0 compare interrupt.

  - Clear the value in the counter register.

- Use the `compareConfigA1` struct in the template `task_2.c` to configure the output compare functionality of Timer_A1. The available arguments are listed in the *DriverLib Userguide* (DriverLib method `Timer_A_initCompare()`). Use the following settings:

  - Use the CCR1 as the compare register.
  - Disable the corresponding CCR1 interrupt.
  - Use the toggle-reset output mode.
  - Initialize the CCR1 register with the value $c_1 = 125$.

  In the *toggle-reset* mode, the output is toggled whenever the counter reaches the value in the CCR1 and the output is reset (set to 0 V) whenever the counter reaches the count-to value (CCR0). This process is depicted in Figure 15. Further modes (which are not used in this lab) are listed in the *Output Modes* Section of the MSP432P4xx Reference Manual (Section 17.2.5.1, p. 613).

- Configure the output compare functionality of Timer_A1 (Placeholder 4). For this, use the DriverLib method `Timer_A_initCompare()` in addition with the previously defined struct `compareConfigA1`. Make sure to pass the struct by reference, i.e. use the `&` operator.
- Configure the Timer_A1 in Up-mode by using the DriverLib method `Timer_A_configureUpMode()` (Placeholder 4). Use the previously defined struct `upConfigA1` and pass it by reference, i.e. use the `&` operator.
- Use the DriverLib method `Timer_A_startCounter()` to start Timer_A1 in Up-mode (Placeholder 4).
- Build and run (debug) the program.
- Observe the intensity of the red LED of LED2 (RGB LED).
- Set the compare value (CCR1) defined in the `compareConfigA1` to $c_2 = 20$.
- Verify that the brightness of the red LED of LED2 changes as expected.
- What is the relation between the value in `CCR1`, the duty-cycle and the intensity of the LED (qualitative relation only)? Calculate the ON-time (in %) for both `CCR1` values ($c_1$ and $c_2$).

**Solution for Task 2.2:**
In the Up-mode, the counter periodically counts up to the value stored in `CCR0` ($=$ PERIOD $- 1 = 127$) and resets to $0$. With the output mode *toggle-reset*, the output is set to HIGH each time the counter reaches the value which is stored in `CCR1` and is set to LOW (i.e. reset) when the counter reaches the `CCR0` value. This results in the following relation for the ON-time:

$$\text{ON-time [\%]} = \frac{\text{PERIOD - (CCR1+1)}}{\text{PERIOD}}$$

The ON-time which correspond to $c_1$ and $c_2$ are listed in Table 2.

The code for Task 2.2 is listed in Snippet 6.

**Table 2:** ON-time calculation for Task 2.2.

| | CCR1 | ON-time |
|---|---|---|
| $c_1$ | 125 | $\frac{128-(125+1)}{128} = 1.56\%$ |
| $c_2$ | 20 | $\frac{128-(20+1)}{128} = 83.59\%$ |

```c
/* Timer_A1 up mode configuration parameters */
const Timer_A_UpModeConfig upConfigA1 =
{
  TIMER_A_CLOCKSOURCE_SMCLK,              // SMCLK clock source
  TIMER_A_CLOCKSOURCE_DIVIDER_1,          // SMCLK/1 = 3MHz
  PWM_CCR0,                               // Value in CCR0
  TIMER_A_TAIE_INTERRUPT_DISABLE,         // Disable Timer interrupt
  TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE,    // Disable CCR0 interrupt
  TIMER_A_DO_CLEAR                        // Clear value
};

/* Timer_A1 compare configuration parameters */
Timer_A_CompareModeConfig compareConfigA1 =
{
  TIMER_A_CAPTURECOMPARE_REGISTER_1,          // Use CCR1 as compare register
  TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,   // Disable CCR interrupt
  TIMER_A_OUTPUTMODE_TOGGLE_RESET,            // Toggle-reset output mode
  125                                         // Compare value (CCR1)
};

//// Placeholder 4 (Task 2.2) ////////////////////////////////////////////
/* Configure the compare functionality of Timer_A1 */
Timer_A_initCompare(TIMER_A1_BASE, &compareConfigA1);
/* Configure Timer_A1 for Up Mode */
Timer_A_configureUpMode(TIMER_A1_BASE, &upConfigA1);
/* Start counter of Timer_A1 */
Timer_A_startCounter(TIMER_A1_BASE, TIMER_A_UP_MODE);
//////////////////////////////////////////////////////////////////////////
```

**Snippet 6:** Timer configuration for generating PWM signals (Task 2.2).

### Task 2.3: Breathing LED [Optional Task]

We can combine the two use cases of the timer learned in the last two tasks (Task 2.1 and Task 2.2) such that the intensity of the LED changes over time. We use Timer_A1 to generate the PWM signal and use Timer_A0 to periodically generate an interrupt. Each execution of the ISR decreases the CCR1 value of Timer_A1 by a small value until the value reaches 0. Then, the ISR increases the CCR1 again until it reaches `PWM_CCR0`. This process is repeated in order to implement the breathing behavior of the LED.

- Continue to use the template `task_2.c`.
- Change the period of Timer_A0 to 500 cycles (use the provided macro `TA0_CCR0_BREATH`). Use the largest divider (`DIVIDER_64`).
- In every interrupt, update the `pwmCompareVal` variable (Placeholder 6). Depending on the current direction (variable `goUp`), increase or decrease the compare value by `COMPARE_VAL_INC_STEP`.
- Invert the direction of update `goUp` such that the compare value repeatedly increases up to `PWM_CCR0` and decreases down to 0 again (Placeholder 7).
- Update the compare value of Timer_A1 with the current value in `pwmCompareVal` (Placeholder 8). For this, you can use the DriverLib method `Timer_A_setCompareValue()`.
- Build and run (debug) your program.
- Verify that the brightness of the LED changes over time.
- [BONUS] Modify the update of the CCR1 register such that the duty cycle values represent a sine-shaped curve over time (Hint: You can use the `sin()` function and the `M_PI` macro for $\pi$).

**Solution for Task 2.3:**
The code for implementing the breathing LED is listed in Snippet 7. In the sample solution, the macro `TA0_CCR0` is defined to be `TA0_CCR0_BREATH=499`.

```
void TA0_0_IRQHandler(void)
{
  /* Definitions for Task 2.3 */
  static int16_t pwmCompareVal = PWM_CCR0;    // Current duty cycle value
  static bool goUp = false;                   // Flag to indicate whether compare
                                              // value is currently increased or decreased

  //// Placeholder 6 (Task 2.3) ////////////////////////////////////////////
  /* Update compare value variable */
  if (goUp)
  {
    pwmCompareVal += COMPARE_VAL_INC_STEP;
  }
  else
  {
    pwmCompareVal -= COMPARE_VAL_INC_STEP;
  }
  //////////////////////////////////////////////////////////////////////////

  //// Placeholder 7 (Task 2.3) ////////////////////////////////////////////
  /* Invert the direction of increase/decrease of the duty cycle */
  if (pwmCompareVal <= 0)
  {
    goUp = !goUp;
    pwmCompareVal = 0;
  }
  else if(pwmCompareVal >= PWM_CCR0)
  {
    goUp = !goUp;
    pwmCompareVal = PWM_CCR0;
  }
  //////////////////////////////////////////////////////////////////////////

  //// Placeholder 8 (Task 2.3) ////////////////////////////////////////////
  /* Update the compare value of Timer A1 (linear increase/decrease) */
  uint16_t pwmCompareValCurr = pwmCompareVal;
  /* BONUS: sine-shaped increase/decrease */
  // uint16_t pwmCompareValCurr =
  //   (uint16_t) (sin((double)(pwmCompareVal)/PWM_CCR0*0.5*M_PI)*PWM_CCR0);
  Timer_A_setCompareValue(TIMER_A1_BASE,
                          TIMER_A_CAPTURECOMPARE_REGISTER_1,
                          pwmCompareValCurr);
  //////////////////////////////////////////////////////////////////////////

  //// Placeholder 9 (Task 2.1) ////////////////////////////////////////////
  /* Clear interrupt of timer A0 */
  Timer_A_clearCaptureCompareInterrupt(TIMER_A0_BASE,
                                       TIMER_A_CAPTURECOMPARE_REGISTER_0);
  //////////////////////////////////////////////////////////////////////////
}

```

**Snippet 7:** ISR for the breathing LED (Task 2.3).