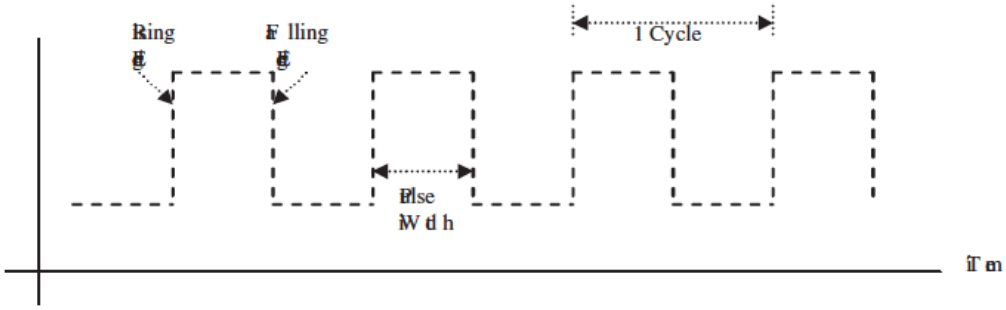
# Fall 2023 - CSCE 438/838: IoT Lab 2

# Clocks, Timers, and Interrupts

Embedded systems often require mechanisms for counting the occurrence of events and for performing tasks at regular intervals. Microcontrollers use two types of hardware to support these functions: 1) Clock: A periodic signal with a certain frequency (clock frequency). 2) Timer: A counter that is updated based on some events, a clock signal, or an external event. Using the clock and timer, we can ensure that events are not missed and that the timing of behavior occurs at regular intervals. Hence, it is important to choose the configurations for clocks and timers accordingly to satisfy the requirements.

## Clock System

Almost every embedded board has an oscillator, a circuit whose sole purpose is generating a repetitive signal. Digital clock generators, or simply clocks, are oscillators that generate signals with a square waveform. The edges of the square waves trigger hardware throughout the device to synchronize the changes in different components. Different components may require oscillators that generate signals of various waveforms, such as sinusoidal, pulsed, sawtooth, and so on, to drive them. In the case of components driven by a digital clock, it is the square waveform. The waveform forms a square because the clock signal is a logical signal that continuously changes from 0 to 1 or 1 to 0. The output of the synchronous sequential circuit is synchronized with that clock.

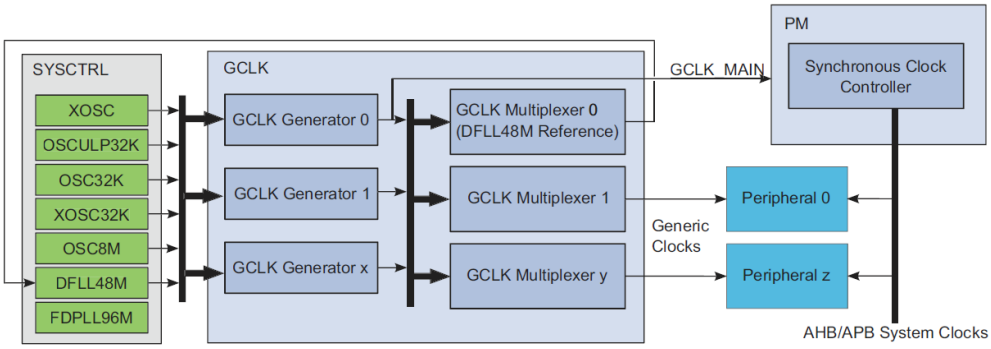



- Clocks for microcontrollers used to be simple. Usually, a crystal with a frequency of a few MHz would be connected to two pins. It would drive the CPU directly and was typically divided by a factor of 2 or 4 for the main bus. Unfortunately, the conflicting demands for high performance and low power mean that most modern microcontrollers have much more complicated clocks, often with two or more sources. In many applications, the MCU spends most of its time in a low-power mode until some event occurs, when it must wake up and handle the event rapidly. It is often necessary to keep track of real-time so that the MCU can wake periodically (e.g., every second or minute) or to time-stamp external events. Therefore, two clocks with quite different specifications are often needed:
  - A fast clock to drive the CPU, which can be started and stopped rapidly to conserve energy but usually need not be particularly accurate.

    - A slow clock that runs continuously to monitor real-time, which must, therefore, use little power and may need to be accurate.

- Several types of oscillators are used to generate the clock signal. These are the two extreme types, which are also the most common:
    - Crystal: Accurate (the frequency is close to what it says on the can, typically within 1 part in 105) and stable (does not change significantly with time or temperature). Crystals for microcontrollers typically run at either a high frequency of a few MHz to drive the main bus or a low frequency of 32,768 Hz for a real-time clock. The disadvantages are that crystals are expensive and delicate, the oscillator draws a relatively large current, particularly at high frequency, and the crystal is an extra component and may need two capacitors. Crystal oscillators also take a long time to start up and stabilize, often around 105 cycles, an unavoidable side effect of their high stability.
    - Resistor and capacitor (RC): Cheap and quick to start but used to have poor accuracy and stability. The components can be external but are now more likely to be integrated within the MCU. The quality of integrated RC oscillators has improved dramatically in recent years, and the F20xx provides four frequencies calibrated at the factory to within ±1%.

**Clocks on SAMD21 (Section 13 and 14 on Datasheet)**

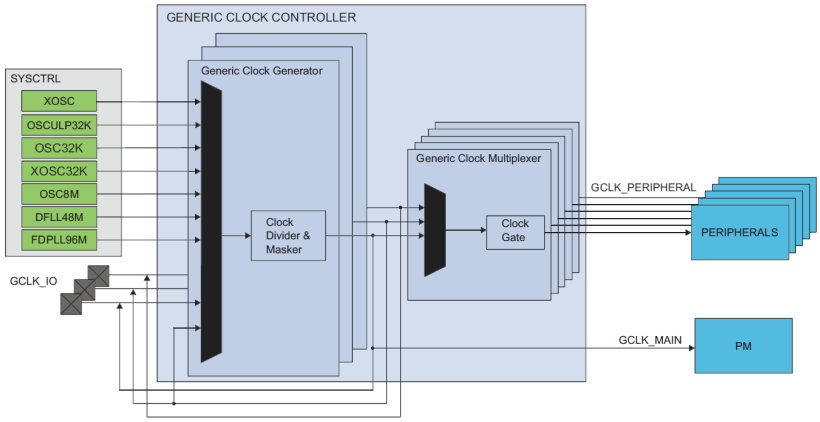



- The clock system on the SAM D21 consists of:
    - Clock sources, controlled by SYSCTRL
        - A Clock source is the base clock signal used in the system. Example clock sources are the internal 8MHz oscillator (OSC8M), External crystal oscillator (XOSC), and the Digital frequency locked loop (DFLL48M).
        - Clock sources (Page 3):

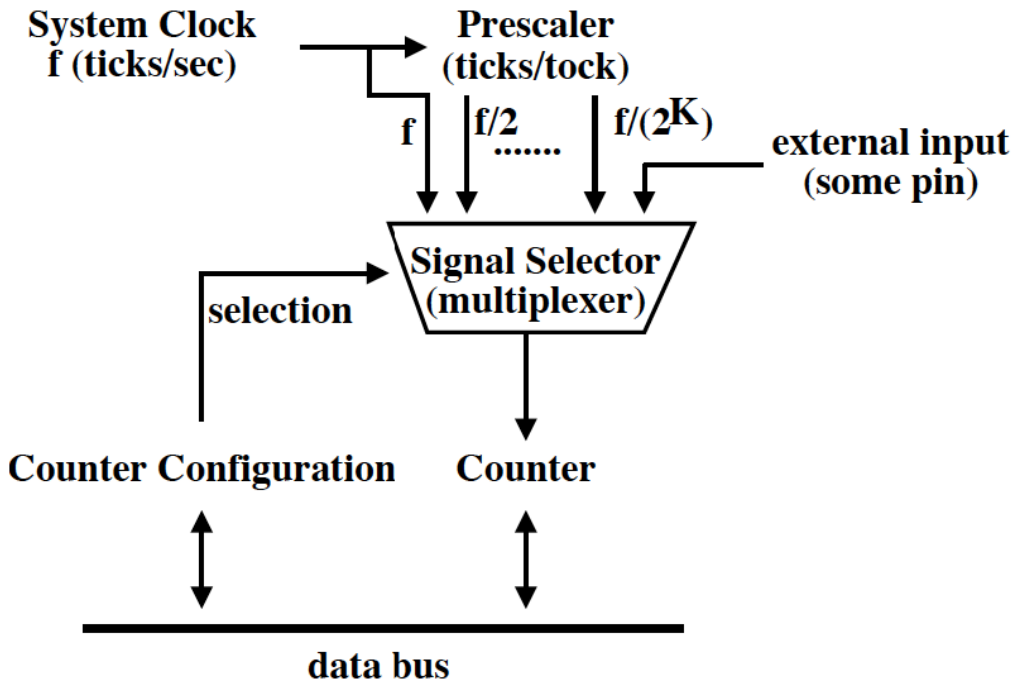
| | |
|---|---|
| Oscillators | 32.768kHz crystal oscillator (XOSC32K)<br>0.4-32MHz crystal oscillator (XOSC)<br>32.768kHz internal oscillator (OSC32K)<br>32kHz ultra-low-power internal oscillator (OSCULP32K)<br>8MHz high-accuracy internal oscillator (OSC8M)<br>48MHz Digital Frequency Locked Loop (DFLL48M)<br>96MHz Fractional Digital Phased Locked Loop (FDPLL96M) |

    - Generic Clock Controller (GCLK), which controls the clock distribution system, is made up of:

- `Generic Clock generators:` A programmable prescaler that can use any of the system clock sources as its source clock. The Generic Clock Generator 0, also called GCLK_MAIN, is the clock feeding the Power Manager used to generate synchronous clocks.
- `Generic Clocks`: Typically, the clock input of a peripheral on the system. The generic clocks, through the Generic Clock Multiplexer, can use any of the Generic Clock generators as its clock source. Multiple instances of a peripheral will typically have a separate generic clock for each instance. The DFLL48M clock input (when multiplying another clock source) is generic clock 0.




## Timer/Counter (TC)



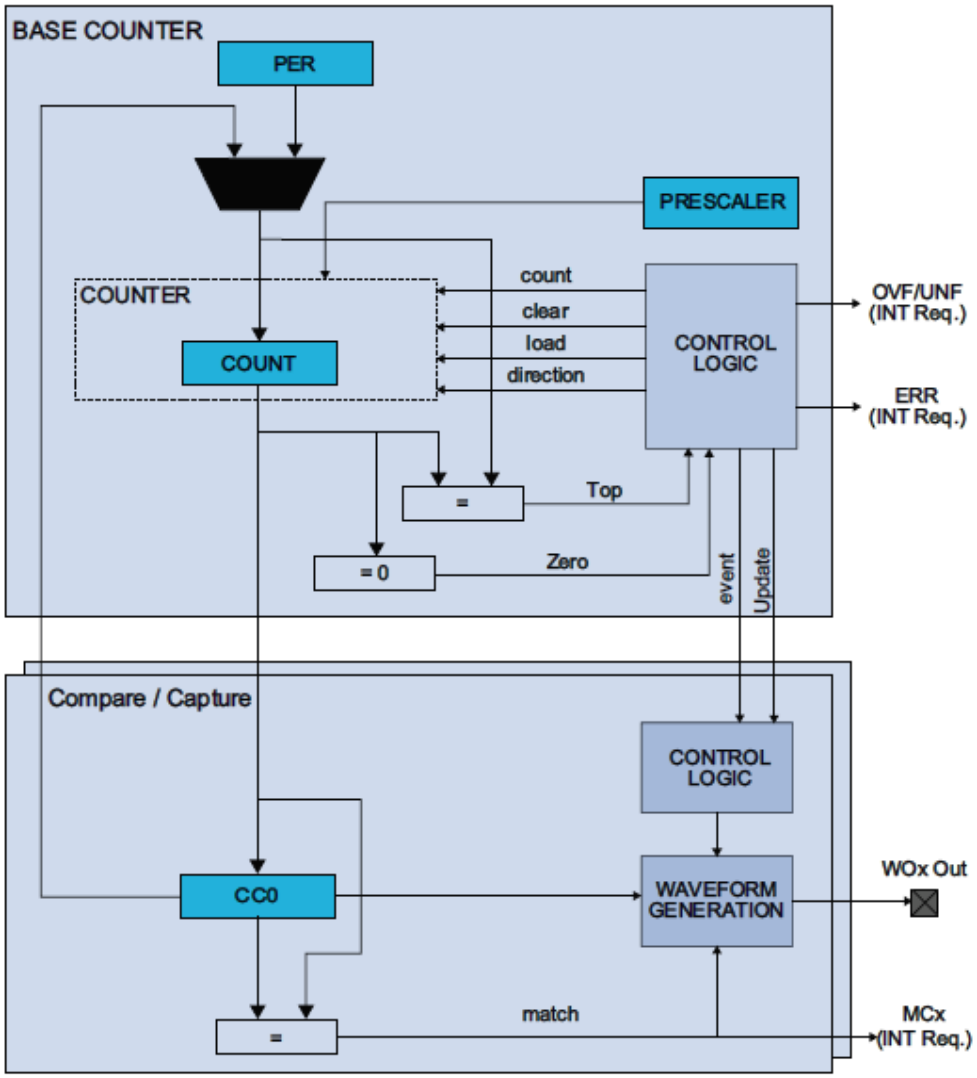
**Concepts** Timers: Most microcontrollers have at least one timer because of the wide range of functions that they provide:

- The time at which transitions occur on an input can be recorded. This may be used to deduce the speed of a bicycle, for instance, if the input is driven by a sensor that gives a pulse every time the wheel completes a revolution.
- Outputs can be driven on and off automatically at a specified frequency. This is used for pulse-width modulation to control the speed of the motor in a washing machine.
- They provide a regular “tick” that can be used to schedule tasks in a program. Many programs are awakened periodically by the timer to perform some action—measure the temperature and transmit it to a base station, for example—then go to sleep (enter a low-power mode) until awakened again. This conserves power, which is vital in battery-powered applications.

**Components of a timer:**

1. **Clock Source**: The counter can be incremented at regular intervals using the system clock source (or a derivative thereof).
2. **Counter**: A counter counts the number of times the detector signal transitions from high to low. A counter is a register.
3. **Prescaler**: Prescalers are implemented as counters in and of themselves. By “tapping into” the prescaler counter at different bits, we can divide the system clock by a range of different divisors.
4. **Signal Selector**: Signal Selector is implemented as a multiplexer, selecting one of the available event sources to be provided to the counter. Which event source is selected is determined by the configuration of the counter.

**Timers on SAMD21 (Section 29 on Datasheet)**




Features of TC on SAMD21

- Selectable configuration
- 8-, 16- or 32-bit TC, with compare/capture channels
- Waveform generation
  - Frequency generation
  - Single-slope pulse-width modulation
- Input capture
  - Event capture
  - Frequency capture
  - Pulse-width capture
- One input event
- Interrupts/output events on:
  - Counter overflow/underflow
  - Compare match or capture
- Internal prescaler

**Clock selection**(Table 14-4) Timers share input clock signals, which should be noticed when selecting the frequency for your timer.

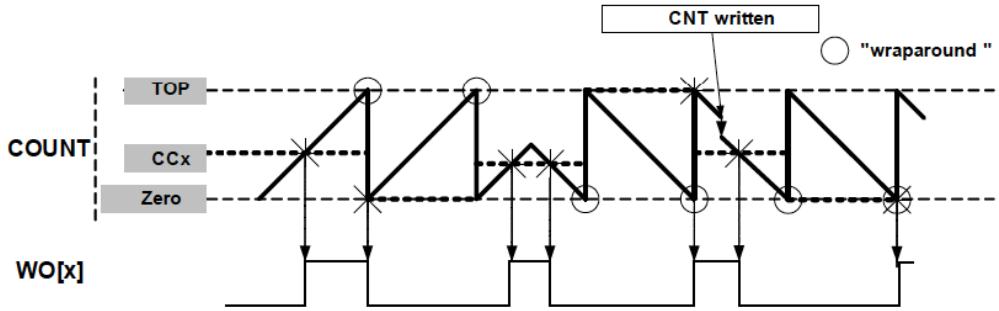
| | | |
|---|---|---|
| 0x1A | GCLK_TCC0, GCLK_TCC1 | TCC0,TCC1 |
| 0x1B | GCLK_TCC2, GCLK_TC3 | TCC2,TC3 |
| 0x1C | GCLK_TC4, GCLK_TC5 | TC4,TC5 |
| 0x1D | GCLK_TC6, GCLK_TC7 | TC6,TC7 |

**Timer Waveform Generation Operation/Frequency Operation Mode** The time can be used to generate a new frequency signal called **Waveform Generation Operation**. Also, the timer value does not increment indefinitely; there are two modes to control the frequency operation of the timer, where the top value of the time can be configured:

1. **Normal Frequency Operation (**NFRQ**)**

When NFRQ is used, the waveform output (WO[x]) toggles every time **CCx and the counter are equal**, and the interrupt flag corresponding to that channel will be set. The top value is the maximum value that the timer allows. For example, the top value of the 16-bit mode is 2^16 = 65536. We don’t need to configure CCx in this lab, and using the counter value is sufficient.

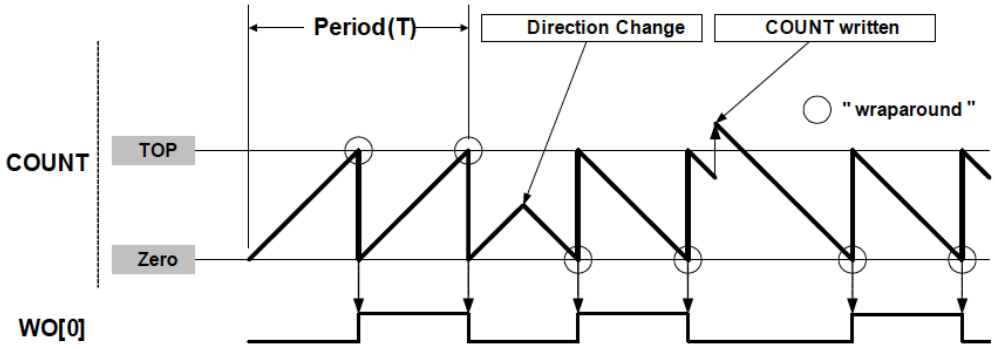
**Figure 29-4. Normal Frequency Operation**




2. **Match Frequency Operation (**MFRQ**)**

When MFRQ is used, the value in **CC0 will be used as the top value**, and WO[0] will toggle on every overflow/underflow.

**Figure 29-5. Match Frequency Operation**




These behaviors are controlled by timer control (**Control A**) register [Page 628]:

### 29.8.1 Control A

**Name:** CTRLA
**Offset:** 0x00
**Reset:** 0x0000
**Property:** Write-Protected, Enable-Protected, Write-Synchronized

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|
| | | | PRESCSYNC[1:0] | | RUNSTDBY | PRESCALER[2:0] | | |
| Access | R | R | R/W | R/W | R/W | R/W | R/W | R/W |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | | WAVEGEN[1:0] | | | MODE[1:0] | | ENABLE | SWRST |
| Access | R | R/W | R/W | R | R/W | R/W | R/W | R/W |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The configurations available are specified:

- **Bits 6:5 – WAVEGEN[1:0]: Waveform Generation Operation**
  These bits select the waveform generation operation. They affect the top value, as shown in "Waveform Output Operations" on page 616. It also controls whether frequency or PWM waveform generation should be used. How these modes differ can also be seen from "Waveform Output Operations" on page 616.
  These bits are not synchronized.

**Table 29-7. Waveform Generation Operation**

| Value | Name | Operation | Top Value | Waveform Output on Match | Waveform Output on Wraparound |
|---|---|---|---|---|---|
| 0x0 | NFRQ | Normal frequency | PER(1)/Max | Toggle | No action |
| 0x1 | MFRQ | Match frequency | CC0 | Toggle | No action |

## Interrupts

Execution of a program usually proceeds predictably, but there are two classes of *exceptions* to this rule: interrupts and resets:

- Interrupts: Usually generated by hardware (although they can be initiated by software) and often indicate an event that needs an urgent response has occurred. For instance, a packet of data might have been received and needs to be processed before the next packet arrives. The processor stops what it was doing, stores enough information (the contents of the program counter and status register) for it to resume later on, and executes an interrupt service routine (ISR). It returns to its previous activity when the ISR has been completed. Thus, an ISR is a subroutine called by hardware (at an unpredictable time) rather than software. A second use of interrupts is to wake the processor from a low-power state.
- Resets: Again, usually generated by hardware when power is applied, or something catastrophic has happened and regular operation cannot continue. This can happen accidentally if the watchdog timer is not disabled, which is easy to forget. A reset causes the device to (re)start from a well-defined state.

At least four of the ten functions from the list of device driver functionality introduced at the start of this chapter are supported by interrupt-handling device drivers, including:

- Interrupt-Handling *Startup*, initialization of the interrupt hardware (i.e., interrupt controller, activating interrupts, etc.) upon power-on or reset.
- Interrupt-Handling *Shutdown*, configuring interrupt hardware (i.e., interrupt controller, deactivating interrupts, etc.) into its power-off state.
- Interrupt-Handling *Disable*, allowing other software to disable active interrupts on the fly (not allowed for Non-Maskable Interrupts (NMIs), which are interrupts that cannot be disabled).
- Interrupt-Handling *Enable*, allowing other software to enable inactive interrupts on-the-fly.

and one additional function unique to interrupt-handling:

- Interrupt-Handler *Servicing*, the interrupt-handling code itself, is executed after the interruption of the main execution stream (this can range in complexity from a simple non-nested routine to nested and/or reentrant routines).

**SAMD21 Nested Vector Interrupt Controller** Built-in functions:

- NVIC_DisableIRQ(TCx_IRQn);
- NVIC_ClearPendingIRQ(TCx_IRQn);
- NVIC_SetPriority(TCx_IRQn, 0);
- NVIC_EnableIRQ(TCx_IRQn);

Interrupt service handler:

- void TCx_Handler()

Interrupt priority in SAMD21.

- NMI: non-maskable interrupt
- Note WDT and TC

| Peripheral Source | NVIC Line |
|---|---|
| EIC NMI – External Interrupt Controller | NMI |
| PM – Power Manager | 0 |
| SYSCTRL – System Control | 1 |
| WDT – Watchdog Timer | 2 |
| RTC – Real Time Counter | 3 |
| EIC – External Interrupt Controller | 4 |
| NVMCTRL – Non-Volatile Memory Controller | 5 |
| DMAC - Direct Memory Access Controller | 6 |
| USB - Universal Serial Bus | 7 |
| EVSYS – Event System | 8 |
| SERCOM0 – Serial Communication Interface 0 | 9 |
| SERCOM1 – Serial Communication Interface 1 | 10 |
| SERCOM2 – Serial Communication Interface 2 | 11 |
| SERCOM3 – Serial Communication Interface 3 | 12 |
| SERCOM4 – Serial Communication Interface 4 | 13 |
| SERCOM5 – Serial Communication Interface 5 | 14 |
| TCC0 – Timer Counter for Control 0 | 15 |
| TCC1 – Timer Counter for Control 1 | 16 |
| TCC2 – Timer Counter for Control 2 | 17 |

| Peripheral Source | NVIC Line |
|---|---|
| TC3 – Timer Counter 3 | 18 |
| TC4 – Timer Counter 4 | 19 |
| TC5 – Timer Counter 5 | 20 |
| TC6 – Timer Counter 6 | 21 |
| TC7 – Timer Counter 7 | 22 |
| ADC – Analog-to-Digital Converter | 23 |
| AC – Analog Comparator | 24 |
| DAC – Digital-to-Analog Converter | 25 |
| PTC – Peripheral Touch Controller | 26 |
| I2S - Inter IC Sound | 27 |

**SAMD21 Timer interrupts** The TC has the following interrupt sources:

- **Overflow/Underflow: OVF.** This asynchronous interrupt can be used to wake the device from any sleep mode.
- **Compare or Capture Channel: MCx**. This asynchronous interrupt can be used to wake the device from any sleep mode.
- **Capture Overflow Error: ERR**. This asynchronous interrupt can be used to wake the device from any sleep mode.
- **Synchronization Ready: SYNCRDY**. This asynchronous interrupt can be used to wake the device from any sleep mode.

### Development procedure

### The procedure for setting up the TC

1. Find out the timer functions from the datasheet Section 29.6
2. Find the registers for controlling the corresponding functions from the datasheet Section 29.7.
3. Map the register address to code (Done by Arduino Core library)
4. Configure the related registers (See the example code in the following Section)
    1. Initialization
    2. Enabling, Disabling and Resetting

# Examples

## Simple Timer using millis()

Using **delay**() to control timing is one of the first things you learn when experimenting with the Arduino. Timing with delay() is simple and straightforward, but it does cause problems down the road when you want to add additional functionality. The problem is that **delay**() is a "busy wait" that monopolizes the processor.

The following example shows how to blink an LED without **delay**(). This example uses the Arduino built-in **millis**() function. This function returns the number of milliseconds since the Arduino board began running the program. This number will overflow (go back to zero) after approximately 50 days. The **millis**() function implements a simple **timer**, which we will learn more about later.

```
// LED definitions in the datasheet
// D13 (PIN_LED_13): Blue
// TX (PIN_LED_TXL): Green
// RX (PIN_LED_RXL): Yellow

// Variables will change:
long previousMillis = 0;        // will store the last time LED was updated

// the following variable is a long because the time, measured in milliseconds,
// will quickly become a bigger number than can be stored in an int.
long interval = 1000;           // interval at which to blink (milliseconds)

void setup() {
  // set the digital pin as output:
  pinMode(PIN_LED_13, OUTPUT);
}

void loop()
{
  // here is where you'd put code that needs to be running all the time.

  unsigned long currentMillis = millis();

  if(currentMillis - previousMillis > interval) {
```

```
    // save the last time you blinked the LED
    previousMillis = currentMillis;

    // Do something here

  }
}
```

## Simple Timer using Timer Interrupt

As we already learned timer and timer interrupt, this example shows you how to use the timer to execute a periodic task. To comprehend the timer's registers, you should check the code with the corresponding description in the datasheet.
The general idea of this program is the same as we learned from Lab 1.
Note that more registers and register bits must be configured for the timer than the watchdog timer.

```
// LED definitions in the datasheet
// D13 (PIN_LED_13): Blue
// TX (PIN_LED_TXL): Green
// RX (PIN_LED_RXL): Yellow

#define CPU_HZ 48000000
#define TIMER_PRESCALER_DIV 1024

void startTimer(int frequencyHz);
void setTimerFrequency(int frequencyHz);
void TC3_Handler();

bool isLEDOn = false;

void setup() {
  SerialUSB.begin(9600);
  // while(!SerialUSB);
  pinMode(PIN_LED_13, OUTPUT);
  startTimer(1);
}

void loop() {}

void setTimerFrequency(int frequencyHz) {
  int compareValue = (CPU_HZ / (TIMER_PRESCALER_DIV * frequencyHz)) - 1;
  TcCount16* TC = (TcCount16*) TC3;
  // Make sure the count is in a proportional position to where it was
  // to prevent any jitter or disconnect when changing the compare value.
  TC->COUNT.reg = map(TC->COUNT.reg, 0, TC->CC[0].reg, 0, compareValue);
  TC->CC[0].reg = compareValue;
  SerialUSB.println(TC->COUNT.reg);
  SerialUSB.println(TC->CC[0].reg);
  while (TC->STATUS.bit.SYNCBUSY == 1);
}
```

```
void startTimer(int frequencyHz) {
  REG_GCLK_CLKCTRL = (uint16_t) (GCLK_CLKCTRL_CLKEN | GCLK_CLKCTRL_GEN_GCLK0 |
GCLK_CLKCTRL_ID_TCC2_TC3) ;
  while ( GCLK->STATUS.bit.SYNCBUSY == 1 ); // wait for sync

  TcCount16* TC = (TcCount16*) TC3;

  TC->CTRLA.reg &= ~TC_CTRLA_ENABLE; //Disable timer
  while (TC->STATUS.bit.SYNCBUSY == 1); // wait for sync

  // Use the 16-bit timer
  TC->CTRLA.reg |= TC_CTRLA_MODE_COUNT16;
  while (TC->STATUS.bit.SYNCBUSY == 1); // wait for sync

  // Use match mode so that the timer counter resets when the count matches the
compare register
  TC->CTRLA.reg |= TC_CTRLA_WAVEGEN_MFRQ;
  while (TC->STATUS.bit.SYNCBUSY == 1); // wait for sync

  // Set prescaler to 1024
  TC->CTRLA.reg |= TC_CTRLA_PRESCALER_DIV1024;
  while (TC->STATUS.bit.SYNCBUSY == 1); // wait for sync

  setTimerFrequency(frequencyHz);

  // Enable the compare interrupt
  TC->INTENSET.reg = 0;
  TC->INTENSET.bit.MC0 = 1;

  NVIC_EnableIRQ(TC3_IRQn);

  TC->CTRLA.reg |= TC_CTRLA_ENABLE;
  while (TC->STATUS.bit.SYNCBUSY == 1); // wait for sync
}

void TC3_Handler() {
  TcCount16* TC = (TcCount16*) TC3;
  // If this interrupt is due to the compare register matching the timer count
  // we toggle the LED.
  if (TC->INTFLAG.bit.MC0 == 1) {
    TC->INTFLAG.bit.MC0 = 1;
    // Write callback here!!!
    digitalWrite(PIN_LED_13, isLEDOn);
    isLEDOn = !isLEDOn;
  }
}
```

## Multi-tasking with Timer

The examples show how to control your microcontroller to do periodic tasks. Now, we are ready to scale up - multi-tasking on the Sparkfun Pro RF. This will be your lab

assignment. The basic concept is to do two (or more) tasks simultaneously at different periods without delaying any tasks.

# Assignment: Multi-tasking

In this assignment, design a multi-tasking system with 3 different approaches. This lab assignment will help you get a better understanding of the following:

1. How to multi-tasking on embedded systems,
2. How to configure and operate the timers on your microcontroller.

## Requirements:

- Toggling two LEDs
  - Turn on the yellow LED for 1 second, and turn it off for 1 second.
  - Turn on the blue LED for 0.5 seconds, and turn it off for 0.5 seconds.
  - Print out a message every time an LED changes its state:
    - “Yellow LED is on/off.”
      - “Blue LED is on/off.”

## Tasks

- **Task 1** (30 points):
  - Approach 1: **Multi-tasking using millis()**
  - Hint: Follow the approach given in the example.
- **Task 2** (30 points):
  - Approach 2: **Using two timer interrupts (TC3 and TC4)**
    - **TC3 and TC4 must be in the Match Frequency Operation mode**
    - **In void startTimer(int frequencyHz), reduce the length of the function by merging the lines that can be combined into one line.**
    - **Report how to configure the CC[0].reg register.**
    - **Use the overflow interrupt (OVF) of the TC3 and TC4, not the match interrupt (MC0)**
  - Hints:
    - Use the match mode for the timers, following the example
    - Clock sources for TC3 and TC4 must be configured differently, with the following code to configure TC4. REG_GCLK_CLKCTRL = (uint16_t) (GCLK_CLKCTRL_CLKEN | GCLK_CLKCTRL_GEN_GCLK0 | GCLK_CLKCTRL_ID_TC4_TC5) ;
- **Task 3** (40 points):
  - Approach 3: **Using one timer unit**
    - **TC3 must be in the Normal Frequency Operation mode, not the match mode.**
    - **TC3 must be running in COUNT8 mode, not the COUNT16 mode like in the example**
    - **Use generic clock generator 2 to supply a 1024Hz clock source to the timer (Similar to Lab 1 WDT example)**
  - Hints:
    - You must use the normal mode of your timer.

        - Figure out the appropriate value for the prescaler to divide the frequency further
        - COUNT8 mode has a special property that may be helpful

- **Task 4** (20 points) (838 Students -> Mandatory; 438 Students -> Bonus):
    - Approach 4: **Using one timer unit**
        - **TC3 must be in the Match Frequency Operation mode**
        - **Report how to configure the CC[0].reg register.**
    - Hints:
        - You must use the match mode of your timer.
        - Check the datasheet and do the math to decide how to configure your registers. You can reuse the clock configuration given in the example.

## Questions:

1. Compare Approach 1 and Approach 2, using or not using the timer. Which one is the better approach for developing reliable embedded system software?

## Report format

- The requirements for each task
- Development plan
    - The procedure of solving the problem and the design of your system
    - Report the configurations used for meeting each requirement in each task in a table, including:

| Register name | Register function | Register value |
|---|---|---|
| | | |
| | | |

- Development Process
    - Record your development process
        - **Acknowledge any resources that you found and helped you with your development (open-source projects/forum threads/books)**
        - Record the software/hardware bugs/pitfalls you had and your troubleshooting procedure.
        - Code snippets for each function you develop

- Test Plan
    - What to test?
        - Each requirement should be tested, e.g., Blinking and printing message
    - What levels of tests?
        - Unit/module level and system level, e.g., For task 2, test two timer modules separately and try them when both are enabled
    - Test results of each task should be recorded in a table. Some example tests are shown in the following table but are not limited to these.

| Component | Test | Result | Comment |
|---|---|---|---|
| Timer | Normal frequency mode configuration test | fail | Record your observation when you see it fails |
| | | | |

| | | | |
|---|---|---|---|
| Timer | Normal frequency mode configuration test | pass | How did you troubleshoot and pass the test |
| Timer | Match frequency mode interrupt test | pass | |
| LED | | not yet run | |
| System | System test | pass | |
| | | | |
| | | | |

- Results
  - Figures in the report:
    - Screenshots that show you complete the required functions (serial message and Arduino IDE warning)
    - Pictures that show you complete the required functions if necessary
  - Answer the questions in the assignment
  - The entire program (Arduino sketch) in the appendix

**Submission Instructions:**

1. Submit your lab on Canvas on or before the deadline (Sep 8th, 8:29 am)
2. Your submission should include one single PDF explaining everything that was asked in the tasks and screenshots, if any
3. Your submission should also include all the code that you have worked on with proper documentation
4. Failing to follow the instructions will make you lose points

# Reference


1. https://gist.github.com/jdneo/43be30d85080b175cb5aed3500d3f989
2. https://learn.adafruit.com/multi-tasking-the-arduino-part-1?view=all
3. Noergaard, Tammy. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012.
4. Davies, John H. *MSP430 microcontroller basics*. Elsevier, 2008.