Embedded Systems CSEN701

Dr. Catherine Elias

Eng. Abdalla Mohamed

Office: C1.213
Mail : abdalla.abdalla@guc.edu.eg

Eng. Youssef Abdelshafy

Office: C2.110
Mail : youssef.abdelshafy@guc.edu.eg

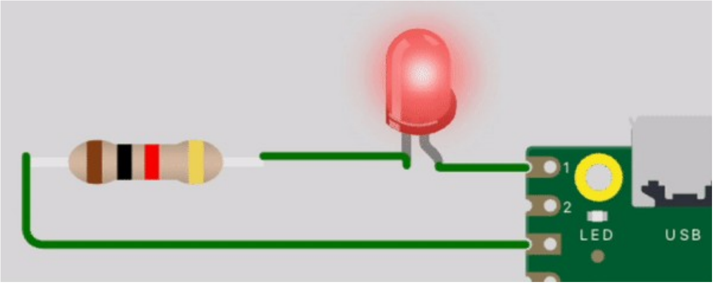
Eng. Mina Wasfy

Office: C4.229
Mail : mina.wasfy@guc.edu.eg

Outline :

- **What are timers & How do they work ?**
- Prescaling and overflow counter
- Timer preipheral in RP 2040
- Alarms in Pico SDK

Within code, we sometimes need a certain functionality to be executed every N seconds, or a certain delay between functionalities (Like blinking a LED).
To execute time related functionalities, there must be a hardware peripherals which notifies the system that the specified time period has passed
These peripherals are called timers.



How do timers work in digital systems

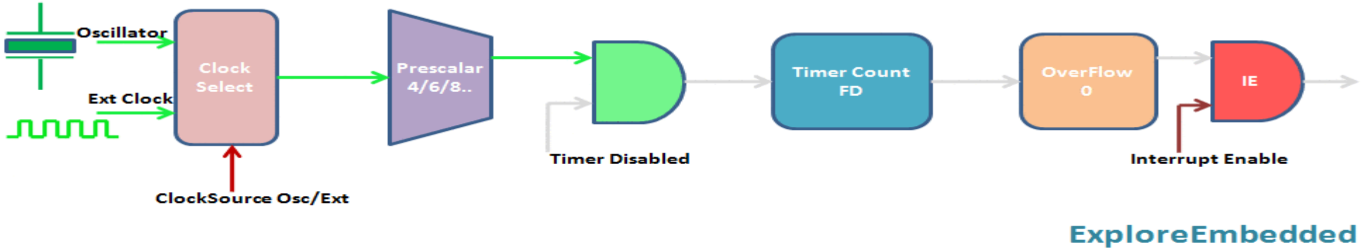
- Timers are composed of registers with specified number of bits that its value automatically **increments**/**decrements** each clock tick.
- Timers can be connected to internal system clock or external clock.
- So if we want to wait for a period of time we will count the number of clock ticks that it takes the system to reach this specified time
- Ex : if a clock tick is generated each 1 microsecond , and we want a delay of 10 microseconds , then we will count 10 ticks (load value).
- Timers can be loaded with a certain value (10 ticks) and **it increments from zero till it reaches the pre-set value**/ **or decrements from the value till zero** thus triggering an **overflow flag** and consequently a **timer interrupt**.

Timers & Overflow

- **Overflow flag** is usually a bit set to one automatically (**hardware_wise**) when the timer overflows its maximum value or reaches the pre-set value.
- The overflow bit triggers the timer interrupt if **enabled**.
- The **overflow bit** must be **cleared** as soon as we enter the **timer_callback_function** / **ISR function**. ISR Interrupt Service Routine

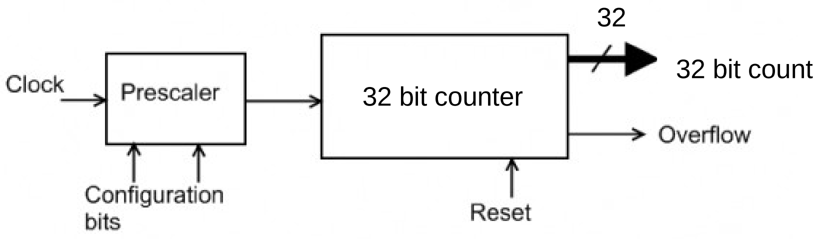
```
PSEUDO CODE

Void ISR_timer() {
CLear_bit(timer_Register,Overflow_flag_bit) ; // clear the flag bit
do the post_delay code :  gpio_put(led_pin,1) ; ( turn on the led after the delay time for example )
printf( " Timer has overflowen and reached the loaded value " ) ;
Timer_load_register = delay_value_to_load ; // setting the new load value for the next overflow
}
```

Timer Block Diagram



Timers & Overflow

- A 32 bit up timer can count up to (2^32 -1) **4300 million Ticks (max load value)**.
- Considering an internal CLK of 1 MHZ 1/10^6 = 1 microsecond, each **tick = 1 us**.
- The maximum delay value before the overflow is **4294967296 us = 4300 seconds corresponding to about 70 mins.**
- Any delay <70 mins can have a corresponding load value , for example a 10 mins delay will be equal to (10 * 60 * 10^6) = **600 million us / ticks ,** so simply this value will be pre-set and the **overflow flag** will be triggered followed by the timer interrupt.
- We can set periodic ISR functions by repeating it using timer delays.



Outline :

- ◎ What are timers & How do they work ?
- ◎ **Prescaling and overflow counter**
- ◎ Timer preipheral in RP2040
- ◎ Alarms in Pico SDK

What if we need a longer delay like 100 mins ?!

a) CLK prescaling :

- Hardware solution (if optional in the architecture)
- Multiplies the maximum time by pre-scaling the timer CLK .
- Instead of incrementing per each CLK tick, the timer can be adjusted to increment every 8 CLK ticks or (chosen 2^n) 32,64,512,1024 ticks depending on the configurations.
- A configuration of bit is set to enable a certain CLK pre-scaling in AVR timers .
- If the **0 1 0** option is set , the same 32 timer can count up to **8 * (4300 million ticks**) = **560 minutes**.
- 100 mins = 100*60*10^6 (number of microsecond ticks) / 8 (pre-scaling increment value) = **750 million** (value to be loaded).

Timer Control Register

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/counter stopped)
0	0	1	clk_{TOS}/(No prescaling)
0	1	0	clk_{TOS}/8 (From prescaler)
0	1	1	clk_{TOS}/32 (From prescaler)
1	0	0	clk_{TOS}/64 (From prescaler)
1	0	1	clk_{TOS}/128 (From prescaler)
1	1	0	clk_{TOS}/256 (From prescaler)
1	1	1	clk_{TOS}/1024 (From prescaler)

COMP3221/9221: Microprocessors and Embedded Systems 24

What if we need a longer delay like 150 mins ?!

b) Overflow counter : **Imagine that the timer is a cup of water with a certain capacity of 70 ml so if we need to get 150 ml of water we will refill it to max twice (over_flows twice) and the third cup will be of 10 ml to complete 150 ml.**

- Software Solution
- Assigns a counter for how many times the timer needs to overflow
- The **over_flow counter** increments in the ISR
- No CLK prescaling

```
PSEUDO CODE
int overflow_couter = 0 ;
int oveflow_limit = 2 ;     // delay time is 150 mins , my max is 70 mins so 150/2 ( delay / (max_load) = 2  .
int remaining_delay_value = value of 10 mins ( delay % max_load)  ;
Void ISR_timer() {   // needs to oveflow twice then another 10 mins are remaining  ( delay % max_load) .
over_flow_counter ++
CLear_bit(timer_Register,Overflow_flag_bit) ; // clear the flag bit
Timer_load_register = delay_value_to_load ; // setting the new load value for the next overflow ( in this case is the maximum)
if ( over_flow_counter == over_flow_limit ) { // check it overflows twice before entering the if
    Timer_load_register = remaining_delay_value ; ( reload the remaining 10 mins)
    }

if ( over_flow_counter == 3 ) {  // (3 = overflow_limit + 1)
    over_flow_counter = 0 ;  // reset the counter
    do the 150_min delay code : gpio_put(led,true) ;
    printf( " delay of 150 mins " ) ;
}

}
```

Outline :

- What are timers & How do they work ?
- Prescaling and overflow counter
- **Timer preipheral in RP 2040**
- Alarms in Pico SDK

Lets interact with the Timer peripheral in RP2040 :

1. **We visit the datasheet : https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf**
2. **Navigate to Timer section 4.6 to view the timer Architecture.**

4.6. Timer

4.6.1. Overview

The system timer peripheral on RP2040 provides a global microsecond timebase for the system, and generates interrupts based on this timebase. It supports the following features:

- A single 64-bit counter, incrementing once per microsecond
- This counter can be read from a pair of latching registers, for race-free reads over a 32-bit bus.
- Four alarms: match on the lower 32 bits of counter, IRQ on match.

4.6.1.1. Other Timer Resources on RP2040

The system timer is intended to provide a global timebase for software. RP2040 has a number of other programmable counter resources which can provide regular interrupts, or trigger DMA transfers.

- The PWM (Section 4.5) contains 8× 16-bit programmable counters, which run at up to system speed, can generate interrupts, and can be continuously reprogrammed via the DMA, or trigger DMA transfers to other peripherals.
- 8× PIO state machines (Chapter 3) can count 32-bit values at system speed, and generate interrupts.
- The DMA (Section 2.5) has four internal pacing timers, which trigger transfers at regular intervals.
- Each Cortex-M0+ core (Section 2.4) has a standard 24-bit SysTick timer, counting either the microsecond tick (Section 4.7.2) or the system clock.

Timer in RP 2040

Alarms | Systick | PWM | PIO | DMA | RTC

Alarms overview in RP 2040

Monotonic system counter counting from 0 till 2^64
thousands of years. (64 bit register)
Increments every microsecond .

64 bit

32 bit counter TIMEH (Highest 32-bit of the system counter)

32 bit

32 bit

32 bit counter TIMEL (lowest 32-bit of the system counter)

32 bit counter ALARM0

32 bit counter ALARM1

32 bit counter ALARM2

32 bit counter ALARM3

4.6.2. Counter

The timer has a 64-bit counter, but RP2040 only has a 32-bit data bus. This means that the TIME value is accessed through a pair of registers. These are:

- TIMEHW and TIMELW to write the time
- TIMEHR and TIMELR to read the time

These pairs are used by accessing the lower register, L, followed by the higher register, H. In the read case, reading the L register latches the value in the H register so that an accurate time can be read. Alternatively, TIMERAWH and TIMERAWL can be used to read the raw time without any latching.

CAUTION

> While it is technically possible to force a new time value by writing to the TIMEHW and TIMELW registers, programmers are discouraged from doing this. This is because the timer value is expected to be monotonically increasing by the SDK which uses it for timeouts, elapsed time etc.

Alarms overview in RP 2040

Monotonic system counter counting from 0 till 2^64
thousands of years. (64 bit register)
Increments every microsecond .

64 bit

32 bit counter TIMEH (Highest 32-bit of the system counter)

32 bit

32 bit

32 bit counter TIMEL (lowest 32-bit of the system counter)

Alarms match on (compare their target value with) with **TIMEL** register of the main counter.

32 bit counter ALARM0

32 bit counter ALARM1

32 bit counter ALARM2

32 bit counter ALARM3

4.6.3. Alarms

The timer has 4 alarms, and outputs a separate interrupt for each alarm. The alarms match on the lower 32 bits of the 64-bit counter which means they can be fired at a maximum of 2^{32} microseconds into the future. This is equivalent to:

- $2^{32} \div 10^6$: ~4295 seconds
- $4295 \div 60$: ~72 minutes

NOTE

This timer is expected to be used for short sleeps. If you want a longer alarm see Section 4.8.

To enable an alarm:

- Enable the interrupt at the timer with a write to the appropriate alarm bit in INTE: i.e. `(1 << 0)` for `ALARM0`
- Enable the appropriate timer interrupt at the processor (see Section 2.3.2)
- Write the time you would like the interrupt to fire to ALARM0 (i.e. the current value in TIMERAWL plus your desired alarm time in microseconds). Writing the time to the `ALARM` register sets the ARMED bit as a side effect.

Once the alarm has fired, the ARMED bit will be set to `0`. To clear the latched interrupt, write a `1` to the appropriate bit in INTR.

32 bit INTE

32 bit ARMED

32 bit counter INTR

- **INTE, ARMED, INTR** registers are 32 bit registers, only the first 4 bits are accessible (for each alarm register).

- To enable the interrupt associated with a specific alarm register, set the corresponding bit to it in INTE register to 1

32 bit INTE

- Once an alarm is armed, the corresponding bit to it in ARMED register is set to 1
- Once an alarm is fired, the corresponding bit to it in ARMED register is set to 0

32 bit ARMED

- To clear interrupt, set the corresponding bit to it in INTR register is set to 1

32 bit counter INTR

Alarms

		3	2	1	0
INTE	Not accessible	1	0	0	1
ARMED	Not accessible	0	0	0	1
INTR	Not accessible	1	0	0	0

Alarm 0 is enabled and armed.

Alarm 3 is enabled and has just fired and got cleared and ready to be armed again .

To enable an alarm:

- Enable the interrupt at the timer with a write to the appropriate alarm bit in INTE: i.e. (1 << 0) for ALARM0
- Enable the appropriate timer interrupt at the processor
- Write the time you would like the interrupt to fire to ALARM0 (i.e.).
- Writing the time to **the current value in TIMERAWL plus your desired alarm time in microseconds** the ALARM register sets the ARMED bit as a side effect
- When the alarm fires it disarms itself automatically by setting the ARMED bit to zero .

IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source
0	TIMER_IRQ_0	6	XIP_IRQ	12	DMA_IRQ_1	18	SPI0_IRQ	24	I2C1_IRQ
1	TIMER_IRQ_1	7	PIO0_IRQ_0	13	IO_IRQ_BANK0	19	SPI1_IRQ	25	RTC_IRQ
2	TIMER_IRQ_2	8	PIO0_IRQ_1	14	IO_IRQ_QSPI	20	UART0_IRQ		
3	TIMER_IRQ_3	9	PIO1_IRQ_0	15	SIO_IRQ_PROC0	21	UART1_IRQ		
4	PWM_IRQ_WRAP	10	PIO1_IRQ_1	16	SIO_IRQ_PROC1	22	ADC_IRQ_FIFO		
5	USBCTRL_IRQ	11	DMA_IRQ_0	17	CLOCKS_IRQ	23	I2C0_IRQ		

Low level code
Don't memorize

```
// Use alarm 0
#define ALARM_NUM 0
#define ALARM_IRQ TIMER_IRQ_0

// Alarm interrupt handler
static volatile bool alarm_fired;

static void alarm_irq(void) {
    // Clear the alarm irq
    hw_clear_bits(&timer_hw->intr, 1u << ALARM_NUM);

    // Assume alarm 0 has fired
    printf("Alarm IRQ fired\n");
    alarm_fired = true;
}

static void alarm_in_us(uint32_t delay_us) {
    // Enable the interrupt for our alarm (the timer outputs 4 alarm irqs)
    hw_set_bits(&timer_hw->inte, 1u << ALARM_NUM);
    // Set irq handler for alarm irq
    irq_set_exclusive_handler(ALARM_IRQ, alarm_irq);
    // Enable the alarm irq
    irq_set_enabled(ALARM_IRQ, true);
    // Enable interrupt in block and at processor

    // Alarm is only 32 bits so if trying to delay more
    // than that need to be careful and keep track of the upper
    // bits
    uint64_t target = timer_hw->timerawl + delay_us;

    // Write the lower 32 bits of the target time to the alarm which
    // will arm it
    timer_hw->alarm[ALARM_NUM] = (uint32_t) target;
}

int main() {
    stdio_init_all();
    printf("Timer lowlevel!\n");

    // Set alarm every 2 seconds
    while (1) {
        alarm_fired = false;
        alarm_in_us(1000000 * 2);
        // Wait for alarm to fire
        while (!alarm_fired);
    }
}
```

Outline :

- What are timers & How do they work ?
- Prescaling and overflow counter
- Timer preipheral in RP 2040
- **Alarms in Pico SDK**

Lets interact with the Timer peripheral in RP2040 using Pico SDK Hardware APIs :

1. **We visit the website : https://www.raspberrypi.com/documentation/pico-sdk**
2. **Navigate to Hardware timer .**

Functions for adding alarms :

- **add_alarm_in_ms()** :

Takes parameters of the delay time in milliseconds and the callback function thus setting an alarm from the four and assigning it to call the callback function upon firing.

- **add_alarm_in_us() :**

same as the previous function but in microseconds .

add_alarm_in_ms

```
static alarm_id_t add_alarm_in_ms (uint32_t ms, alarm_callback_t callback, void * user_data, bool fire_if_past)
[inline], [static]
```

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`ms`	the delay (from now) in milliseconds when (after which) the callback should fire
`callback`	the callback function
`user_data`	user data to pass to the callback function
`fire_if_past`	if true, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

Example 1 :

Setting an alarm to do some functionality after 2 seconds

Add the post-delay code in the call back function .

```c
#include <stdio.h>
#include "pico/stdlib.h"

volatile bool alarm_fired = false;

int64_t alarm_callback() {
    printf("Alarm fired!\n");
    alarm_fired = true; // Set the flag when the alarm fires
    return 0; // Do not reschedule
}

int main() {
    stdio_init_all();

    // Set an alarm to fire after 2000 milliseconds (2 seconds)
    add_alarm_in_ms(2000, alarm_callback, NULL, false);

    // Main loop
    while (!alarm_fired) {
        sleep_ms(100); // Sleep for 100 milliseconds to reduce CPU usage
    }

    return 0; // Program complete
}
```

add_alarm_in_ms

`static alarm_id_t add_alarm_in_ms (uint32_t ms, alarm_callback_t callback, void * user_data, bool fire_if_past) [inline], [static]`

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`ms`	the delay (from now) in milliseconds when (after which) the callback should fire
`callback`	the callback function
`user_data`	user data to pass to the callback function
`fire_if_past`	if true, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

Returns

0 if the alarm time passed before or during the call and fire_if_past was false

Returns

<0 if there were no alarm slots available, or other error occurred

Lets interact with the Timer peripheral in RP2040 using Pico SDK Hardware APIs :

1. We visit the website : https://www.raspberrypi.com/documentation/pico-sdk
2. Navigate to Hardware timer .

Predefined struct in the API : Repeating_timer

```
struct repeating_timer {
    uint32_t delay;              // Delay between timer callbacks (in microseconds)
    alarm_id_t id;              // ID of the alarm associated with this timer
    uint32_t last_time;         // Last time this timer was called
    void *user_data;            // Optional user data for the callback
};
```

Functions associated with the repeating_timer struct :

```
static bool add_repeating_timer_us (int64_t delay_us, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
```
Add a repeating timer that is called repeatedly at the specified interval in microseconds.

```
static bool add_repeating_timer_ms (int32_t delay_ms, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
```
Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

```
bool cancel_repeating_timer (repeating_timer_t *timer)
```
Cancel a repeating timer.

Lets interact with the Timer peripheral in RP2040 using Pico SDK Hardware APIs :

1. We visit the website : https://www.raspberrypi.com/documentation/pico-sdk
2. Navigate to Hardware timer .

Function Documentation

add_repeating_timer_ms

```
static bool add_repeating_timer_ms (int32_t delay_ms, repeating_timer_callback_t callback, void * user_data,
repeating_timer_t * out) [inline], [static]
```

Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

`delay_ms`	the repeat delay in milliseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1 microsecond
`callback`	the repeating timer callback function
`user_data`	user data to pass to store in the repeating_timer structure for use by the callback.
`out`	the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

false if there were no alarm slots available to create the timer, true otherwise.

Example 2
Repeat a logic every 1 second 3 times

1. Create an instance of **struct repeating_timer** □ timer (variable name)

2. Implement a **call back function** with the periodic logic , however the function should have a bool return , as if it return true it will continue repeating and if you returned false it will not be repeated

3. Use **add_repeating_timer_ms()** and add your paramaters :
   delay_time □1000
   call_back function name
   Null □ ignored parameter
   address of your repeating timer □ &timer

```c
#include <stdio.h>
#include "pico/stdlib.h"

volatile int repeat_count = 0;  // Counter for how many times the timer has fired

bool repeating_timer_callback(struct repeating_timer *t) {
    repeat_count++;
    printf("Repeating timer fired! Count: %d\n", repeat_count);

    if (repeat_count >= 3) {
        cancel_repeating_timer(t);  // Stop the timer after 3 fires
        printf("Repeating timer stopped after 3 fires.\n");
    }

    return true;  // Continue repeating
}

int main() {
    stdio_init_all();

    struct repeating_timer timer;
    add_repeating_timer_ms(1000, repeating_timer_callback, NULL, &timer);  // Fire every 1

    // Main loop
    while (repeat_count < 3) {  // Loop until the timer has fired 3 times
        sleep_ms(100);  // Sleep for 100 ms to reduce CPU usage
    }

    return 0;  // Program complete
}
```

THANK YOU