Queen Mary University of London

Science and Engineering

School of Electronic Engineering and Computer Science
QMUL-BUPT Joint Programme

EBU6475 Microprocessor System Design
EBU5476 Microprocessors for Embedded Computing

Timer Peripherals

References:
Chapter 9.5, The Definitive Guide to ARM®;
Chapter 7, Embedded Systems Fundamentals

Last updated: 18 April 2020

University Program Education Kits

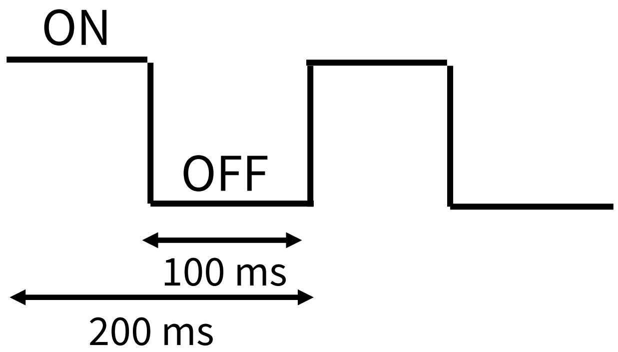
How long does it take to execute an ARM instruction?

- This is not an easy question to answer.
  - Of course we know every microprocessor runs at a certain clock frequency, e.g. STM32-F401RE can run up to 84 MHz.
- Different instructions requires different clock cycles to complete.
  - CPI = Clock Per Instruction
  - For instance, CPI of MOV is 1 (unless move to PC).
- Remember ARM M3/M4 is a pipelined processor so several instructions can be executed in parallel to save time. Therefore CPI is not fixed (but averaged).
- The actual timing of instructions can only be determined when the whole program is ready for simulation or analysis.

Example: Flashing an LED

Let's consider a simple application, we want to flash an LED in a frequency that is visible to human eyes, e.g. 5 Hz.
The LED is connected to a GPIO pin.

Develop a control program
1. initialise output pin
2. turn ON LED
3. wait for 100 ms
4. turn OFF LED
5. wait for 100 ms
6. repeat step 2



How to wait for an accurate time period in our program?

Two ways to create a time delay:
1. use a time delay loop
2. use a hardware timer

Time Delay Loop (Assembly)

Estimate the time taken to execute the following code snippet (in terms of clock cycles):

```
DELAY
    MOV  r0, #250
LOOP
    SUBS r0, #1
    BNE  LOOP
```

Time delay
~= 1 + 250 x (1 + 3)
~= 1001 cycles
This is approximate because of BNE.

How about this?

```
DELAY
    MOV  r0, #250
LOOP
    NOP
    NOP
    NOP
    NOP
    SUBS r0, #1
    BNE  LOOP
```

Time delay
~= 1 + 250 x (1 x 4 + 1 + 3)
~= 2001 cycles

if CPU runs at 80 MHz, this takes approximate 25 µs to execute.

Longer Time Delay

Still quite far from our target 100 ms.
Let's try to write proper functions to calculate.

```
__asm void delay_cycles(unsigned int cycles){
   LSRS r0, #2 // logic shift right 2 bits (/4)
   BEQ  done
loop               // each time it takes 4 cycles
   SUBS r0, #1
   BNE  loop
done
   BX   lr
}
```

Time delay
= 1 + 1 + cycles/4 x (1 + 3) + 2   (assume BEQ not taken)
= cycles + 4

Longer Time Delay (Cont')

The no. of cycles that can be waited is limited by the size of unsigned int = 32 bits in ARM M3/M4, so we need to calculate how many times we need to call **`delay_cycles()`**, depending on the clock frequency (**`SystemCoreClock`** in the code)

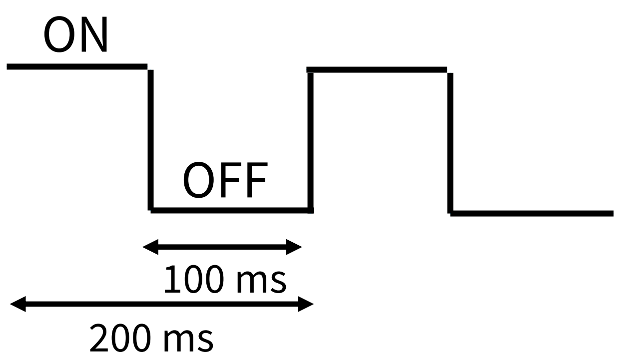
```
void delay_ms(unsigned int ms) {
    unsigned int max_step =
        1000 * (UINT32_MAX / SystemCoreClock);
    unsigned int max_sleep_cycles =
        max_step * (SystemCoreClock / 1000);
    while (ms > max_step) {
        ms -= max_step;
        delay_cycles(max_sleep_cycles);
    }
    delay_cycles(ms * (SystemCoreClock / 1000));
}
```

How would you modify this function to delay for milliseconds (μs) instead?

Flashing an LED: Time Delay Loop

Now we can put together a draft (pseudo code) for our control program.

```
gpio_set_mode(Output);
while (1) {
    gpio_set_pin(HIGH);
    delay_ms(100);
    gpio_set_pin(LOW);
    delay_ms(100);
}
```

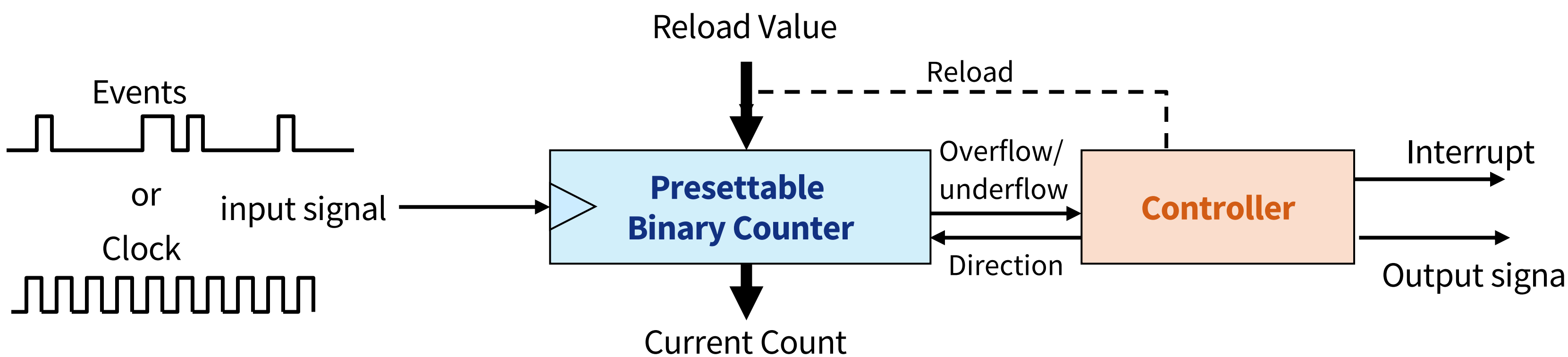


In this solution, the whole processor is doing one and only one thing. Can we utilise the time in running the delay loop for some other more meaningful tasks?

Timer: Concept

- The core of a timer is a digital counter whose value changes by one each time the counter is clocked.
- The faster the clocking rate, the faster the device counts.
    - It is crucial to determine the clocking rate by identifying/choosing the clock source.
- Example: input clock frequency = 10 MHz, then period = 0.1 μs. One count (up or down) represents 0.1 μs.
  How much time has passed if the counter counts from 0 to 6475?

Timer Circuit Hardware

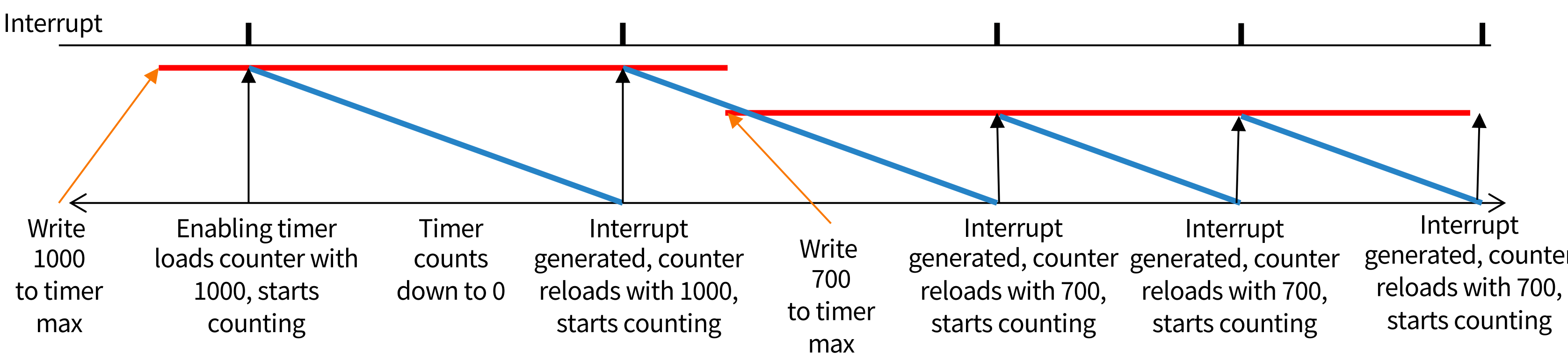


- Based on presettable (i.e. load with a start value) binary counter
- It is enhanced with configurability:
  - Count value can be read and written by the processor
  - Count direction can often be set to up or down
  - Counter's clock source can be selected
    Counter mode: count pulses which indicate events (e.g. odometer pulses)
    Timer mode: clock source is periodic, so counter value is proportional to elapsed time (e.g. stopwatch)
  - Counter's overflow/underflow action can be selected
    Generate interrupt
    Reload counter with special value and continue counting
    Toggle hardware output signal

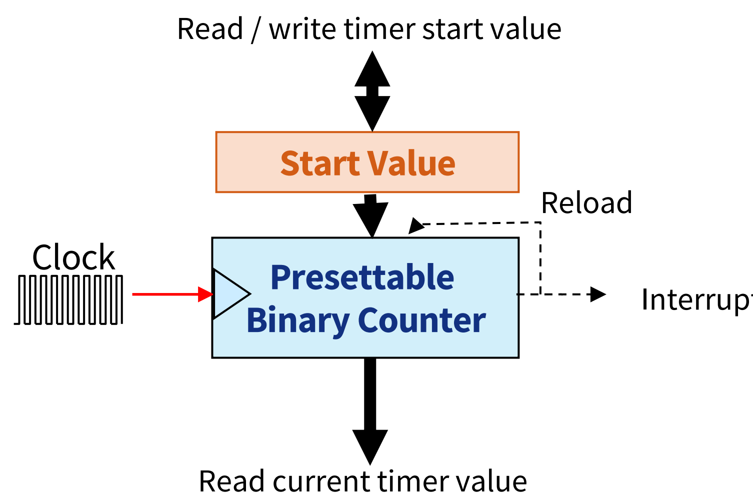
Common Timer Peripherals

Peripherals	Descriptions
Interrupt/SysTick	Part of CPU core’s peripherals Can generate periodic interrupt Can trigger DMA (direct memory access) transfers
Pulse Width Modulation (PWM)	Connected to I/O pins, has input capture and output compare support Can generate PWM signals Can generate interrupt requests
Low-Power Timer	Can operate as timer or counter in all power modes Can wake up system with interrupt Can trigger hardware
Real-time Clock (RTC)	Powered by external 32.768 kHz crystal Tracks elapsed time (seconds) in register Can set alarm Can generate 1 Hz output signal and/or interrupt Can wake up system with interrupt

Interrupt Timer



- Load start value from register
- Counter counts down with each clock pulse
- When timer value reaches zero
  - Generates interrupt
  - Reloads timer with start value



Calculating Start Value

- Goal: generate an interrupt every *T* seconds
- Start value = round(*T* x Freq) - 1
  - We have to round the value since register keeps an integer, not a real number
  - Rounding provides closest integer to desired value, resulting in minimum timing error.
- Example 1: interrupt every 137.41 ms, assuming clock frequency 24 MHz
  - 137.41 ms x 24 MHz – 1 = 3297839 (happens to be integer)
- Example 2: interrupt with a frequency of 88 Hz with a 56 MHz clock
  - round((1/88 Hz) x 46 MHz) - 1 = 522726
  - actual frequency = 88.0000004591 Hz (very small error)

Example: Stopwatch

- Measure time with 100 µs resolution
- Display elapsed time, updating screen every 10 ms
- Controls - switch/button S1: toggle start/stop
- Use interrupt timer
  - Counter decrements from start value every 100 µs
    - Set to timer to expire every 100 µs
    - Calculate start value,
      e.g. at 24 MHz = round(100 µs x 24 MHz)-1 = 2399
  - LCD Update every 10 ms
    - Update LCD every *N*-th ISR
    - N = 10 ms/100 µs = 100
    - Don't update LCD in ISR! Too slow.
    - Instead set flag in ISR, poll it in main loop

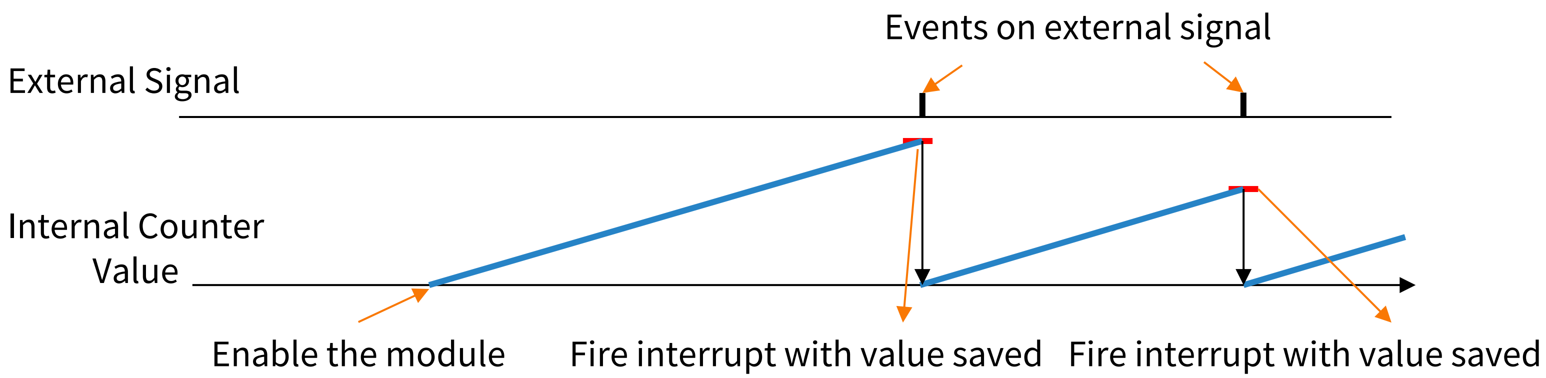
Timer / PWM Module (TPM)

- Core Counter
  - Clock options - external or internal
  - Prescaler to divide clock
  - Can reload with set value, or overflow and wrap around
- Multiple channels - several modes
  - Capture Mode: Capture timer's value when input signal changes
  - Output Compare: Change an output signal when timer reaches certain value
  - PWM: Generate pulse-width-modulated signal. Width of pulse is proportional to specified value.
- Possible triggering of interrupt, hardware trigger on overflow
- One I/O pin per channel

Major Channel Modes

- Input Capture Mode
  - Capture timer's value when input signal changes - rising edge, falling edge, or both
  - This mode answers your question:
    "How long after I started the timer did the input change?"
    so it effectively measure the time difference.
- Output Compare Mode
  - Modify output signal when timer reaches specified value
    Set, clear, pulse, toggle (invert)
  - Make a pulse of specified width
  - Make a pulse after specified delay
- Pulse Width Modulation
  - Make a series of pulses of specified width and frequency

Input Capture Mode



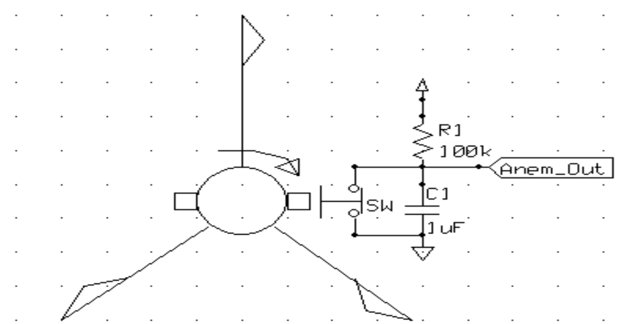
- I/O pin operates as input on edge
- When valid edge is detected on pin…
  - Current value of counter is stored
  - Interrupt is called

Example: Wind Speed Indicator (Anemometer)

- Rotational speed (and pulse frequency) is proportional to wind velocity
- Two measurement options:
  - Frequency
  - Width
- Can solve for wind velocity *v*

$$v_{wind} = \frac{K * f_{clk}}{N_{anemometer}}$$

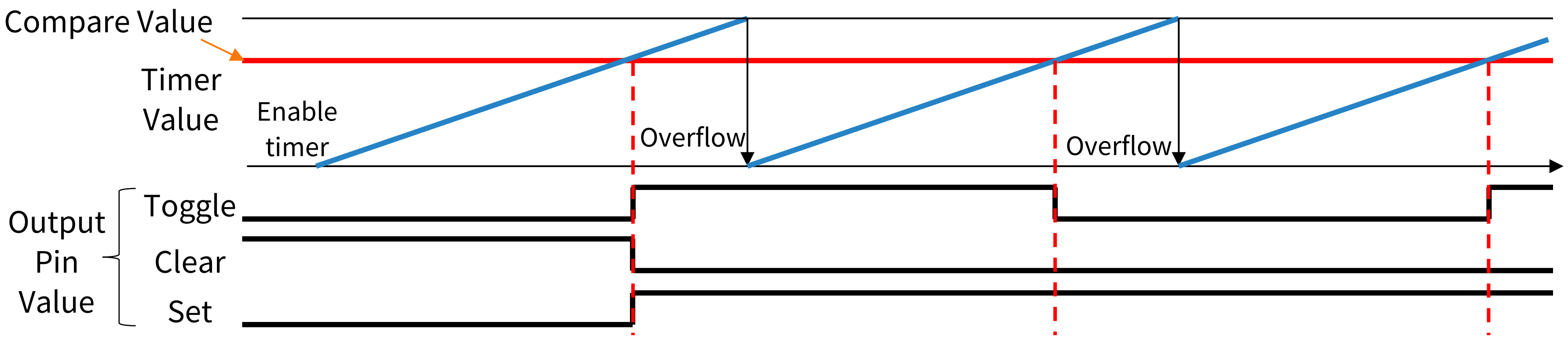
- How can we use the timer for this?
  Use Input Capture Mode to measure period of input signal



TPM Capture Mode for Anemometer

- Configuration
  - Set up module to count at given speed from internal clock
  - Set up channel for input capture on rising edge
- Operation: Repeat
  - First interrupt - on rising edge
    - Reconfigure channel for input capture on falling edge
    - Clear counter, start it counting
  - Second interrupt - on falling edge
    - Read capture value, save for later use in wind speed calculation
    - Reconfigure channel for input capture on rising edge
    - Clear counter, start it counting

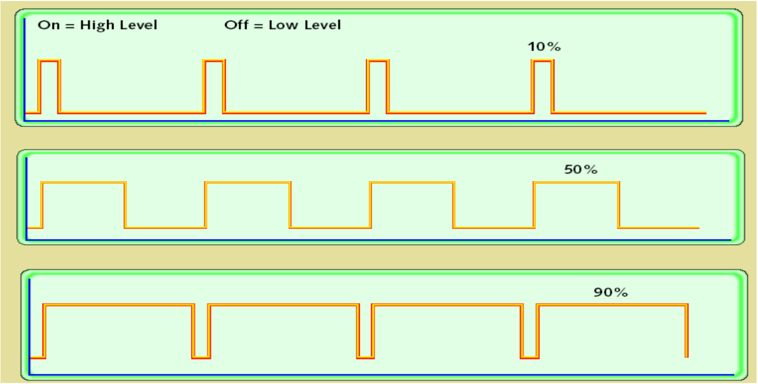
Output Compare Mode



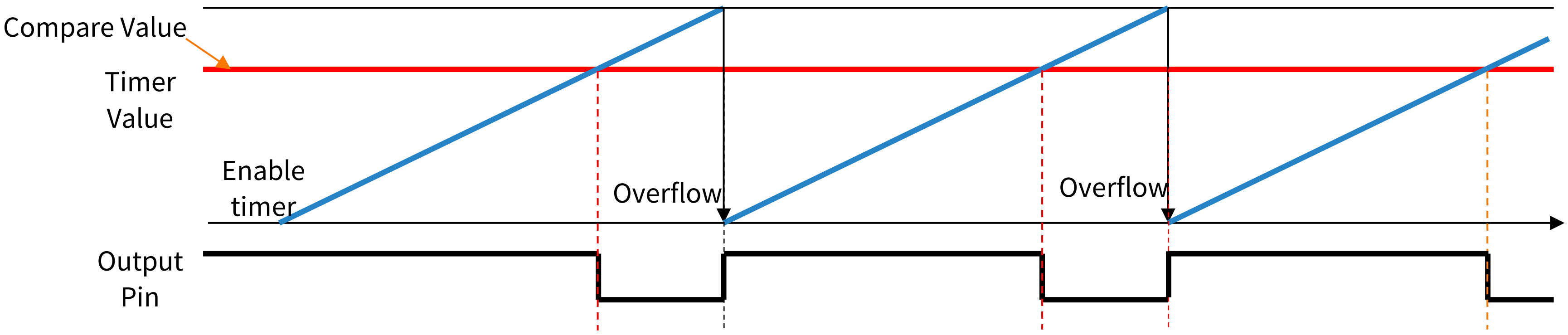
- Action on match
  - Toggle
  - Clear
  - Set
- When counter matches value …
  - Output signal is generated
  - Interrupt is called (if enabled)

Pulse Width Modulation (PWM)

- Digital power amplifiers are more efficient and less expensive than analog power amplifiers
  - Applications: motor speed control, light dimmer, switch-mode power conversion
  - Load (motor, light, etc.) responds slowly, averages PWM signal
- Digital communication is less sensitive to noise than analog methods
  - PWM provides a digital encoding of an analog value
  - Much less vulnerable to noise
- PWM signal characteristics
  - Fixed modulation frequency f_{mod} how many pulses occur per second
  - Period: $1 / f_{mod}$
  - On-time: amount of time that each pulse is on (asserted)
  - Duty-cycle: on-time/period
  - Adjust on-time (hence duty cycle) to represent the analog value



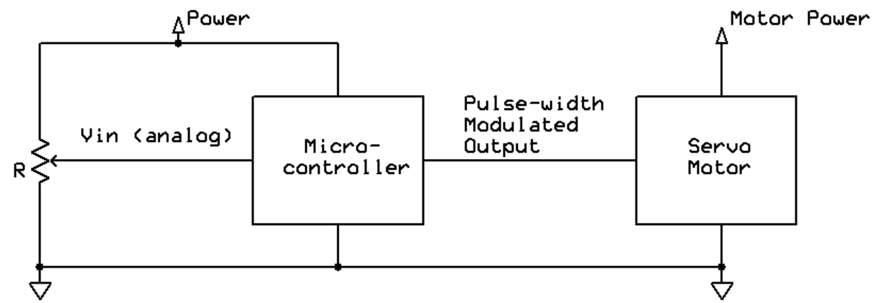
PWM Mode

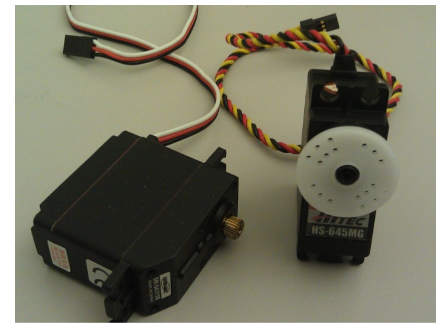


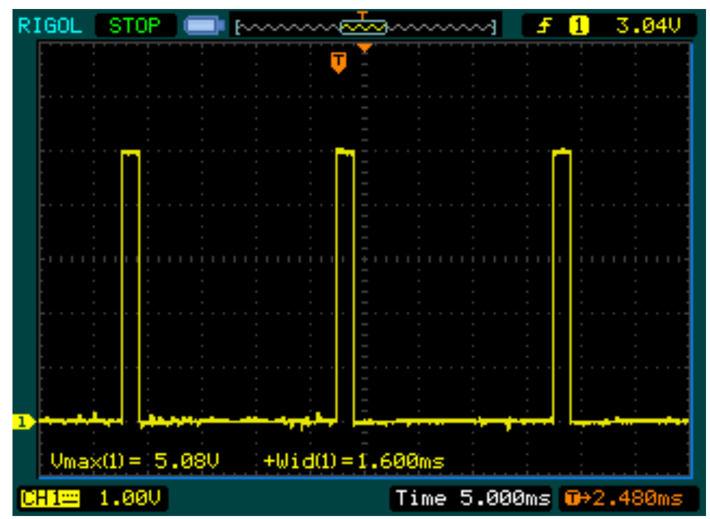
- PWM duty cycle proportional to compare value
  - Period = max timer value
  - On-time = compare value

$$Duty\ Cycle = \frac{Compare\ Value}{Max\ Value} \cdot 100\%$$

PWM to Drive Servo Motor

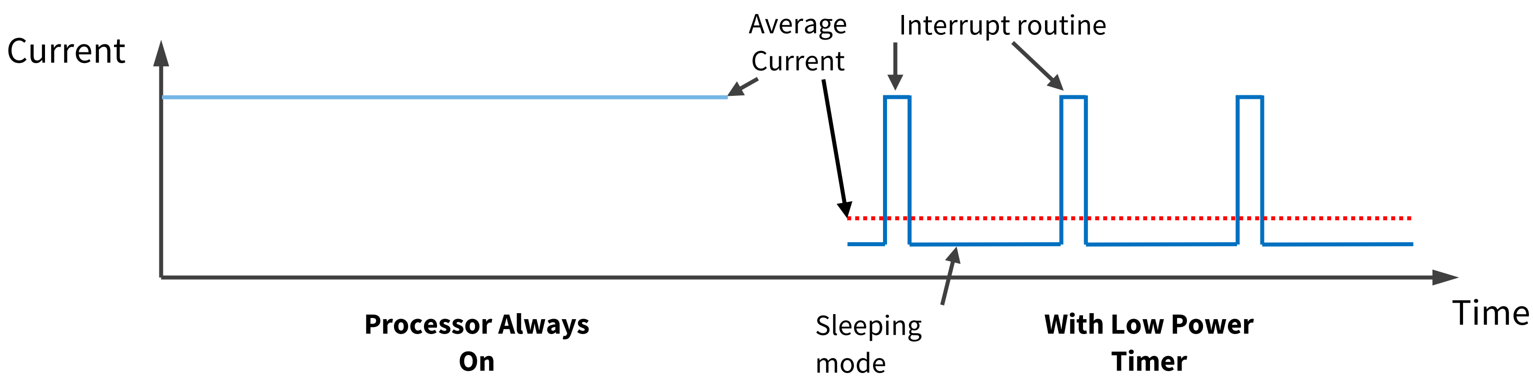






- Servo PWM signal
  - 20 ms period
  - 1 to 2 ms pulse width

Low Power Timer



- Features
  - Count time or external pulses
  - Generate interrupt when counter matches compare value
  - Interrupt wakes MCU from any low power mode
- Current draw can be reduced to microamps or even nanoamps!
- Use the **`WFI`** (Wait For Instruction) instruction (**`__WFI()`** in C)
  - Puts CPU in low power mode until interrupt request

Programming SysTick and Interrupt in C (with CMSIS)

The basic accesses to SysTick registers and developing a simple driver for SysTick

Why have a SysTick timer?

- Cortex-M processors have a small integrated timer called the SysTick (System Tick) timer.
  - Part of NVIC, generating SysTick interrupt.
- SysTick timer is a simple decrement 24-bit timer.
  - either on processor clock frequency, or
  - a reference clock frequency (e.g. on-chip clock source)
- It is common in modern OS that we need a periodic interrupt to execute the OS kernel.
- Even without an OS, SysTick can be used for periodic interrupt generation, delay generation, or timing measurement.

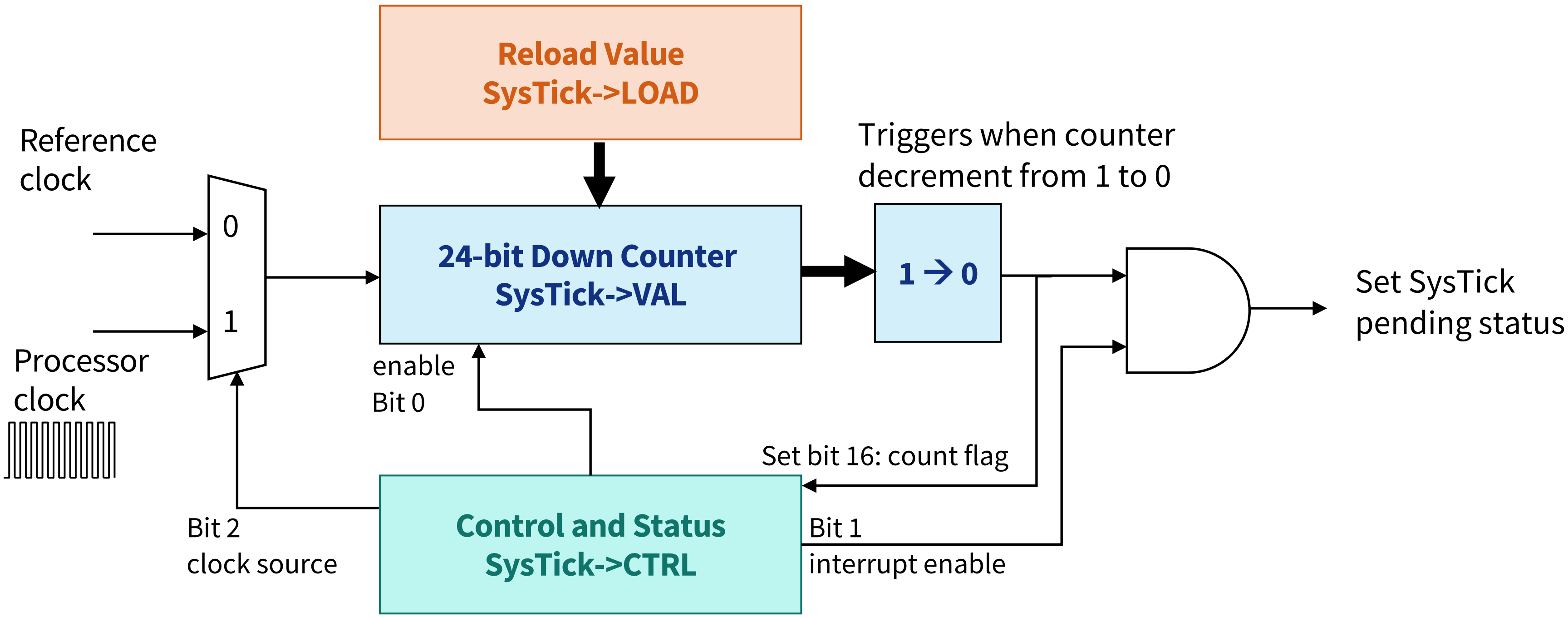
Operations of the SysTick Timer

- SysTick timer has 4 registers.
  - The data structure SysTick is defined in CMSIS to access them easily.
- SysTick is a 24-bit decrement (counts down) counter using either processor's clock or a reference clock.
- It is enabled at bit 0 of CTRL. When it reaches zero, it will load from VAL and continue.

Address	CMSIS-Core Symbol	Register
0xE000E010	SysTick->CTRL	SysTick Control and Status Register
0xE000E014	SysTick->LOAD	SysTick Reload Value Register
0xE000E018	SysTick->VAL	SysTick Current Value Register
0xE000E01C	Systick->CALIB	SysTick Calibration Register

Note: CALIB can be ignored since CMSIS 1.2.

SysTick Timer: Block Diagram



Using the SysTick Timer (CMSIS)

- The easiest way to generate a period SysTick interrupt is to use this CMSIS-Core fuction:
  `uint32_t SysTick_Config(uint32_t ticks);`
- The function sets the interrupt interval to ticks, enables the counter using processor clock and enables the SysTick exception with lowest priority.
- Example: if you want to trigger a SysTick exception of 1 kHz,
  `SysTick_config(SystemCoreClock / 1000);`
  Then **`SysTick_Handler(void)`** is triggered at a rate of 1 kHz.

Reference: https://www.keil.com/pack/doc/CMSIS/Core/html/group__SysTick__gr.html

Writing to SysTick registers

If you want to use the reference clock source or not trigger interrupt, you can write directly to the SysTick registers. Recommended procedure:

1. Disable the SysTick timer by writing 0 to SysTick->CTRL. (Just in case it is enabled previously)
2. Write the new reload value to SysTick->LOAD. The reload value should be (interval value – 1).
3. Write to the SysTick Current Value register SysTick->VAL with any value to clear the current value to 0.
4. Write to the SysTick Control and Status register SysTick->CTRL to start the SysTick timer.

Writing to SysTick registers (Code)

A simple C example of polling SysTick value for timed delay.

```
SysTick->CTRL = 0;      // stop SysTick
SysTick->LOAD = 0xFF;   // count 255+1=256 cycles
SysTick->VAL  = 0;
SysTick->CTRL = 5;
// wait until count flag is set
while ((SysTick->CTRL & 0x00010000) == 0);
SysTick->CTRL = 0;      // stop SysTick
```

What is the delay if the processor is running at 56 MHz?
How accurate is your calculation?

Building Driver for SysTick

- Setup timer to trigger interrupts at every (timestamp) µs
  - **`timer_init(timestamp);`**
- Set interrupt handler
  - **`timer_set_callback(timer_isr);`**
- Enable SysTick (start)
  - **`timer_enable();`**
- Disable SysTick (stop)
  - **`timer_disable();`**

Setup SysTick: timer_init()

```
void timer_init(uint32_t timestamp) {
  uint32_t tick_us = (SystemCoreClock)/1e6;
  tick_us = tick_us*timestamp;
  SysTick_Config(tick_us);
  //NVIC_SetPriority(SysTick_IRQn, 3);
}
```

Explanation:

- Calculate how many cycles for each millisecond
- Multiply with timestamp to get required interval (in cycles)
- Configure the interval using **`SysTick_Config()`**
- **Change interrupt priority if needed**

Enable/Disable Timer

Start and stop the timer by setting/clearing the right bits

```
void timer_enable(void) {
  SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                   SysTick_CTRL_TICKINT_Msk   |
                   SysTick_CTRL_ENABLE_Msk;
}
void timer_disable(void) {
  SysTick->CTRL  &= ~SysTick_CTRL_ENABLE_Msk;
}
```

Explanation:

- start: processor clock, enable interrupt and enable timer
- stop: disable timer (clear bit 0)
- All bit masks (e.g. SysTick_CTRL_CLKSOURCE_Msk) are defined for compatibility (and no need to remember exact bit no.)

Enable/Disable Timer

Set up a pointer to function for the interrupt service routine

```
static void (*timer_callback)(void) = 0;
void timer_set_callback(void (*callback)(void)) {
  timer_callback = callback;
}

void SysTick_Handler(void){
  timer_callback();
}
```

Explanation:

- User provides a pointer to the callback function (the actual ISR) by calling timer_set_callback().
- When timer interrupt happens, SysTick_Handler() is invoked, it then accesses the pointer to the callback and calls the function.

Example: Flashing an LED (Timer)

Let's revisit our example using timer interrupt instead.

```
void toggle_led(void){
    gpio_toggle(LED_PIN);
}
void main(void) {
  timer_init(100000); // 100 ms = 100000 us
  timer_set_callback(toggle_led);
  timer_enable();
  __enable_irq();
  while (1)
    __WFI();
}
```

- __enable_irq(): The function enables interrupts and all configurable fault handlers by clearing PRIMASK.
- __WFI() suspends execution until one of the following events occurs (put in low power mode). But the while loop can be replaced by other tasks to be run in parallel.