BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS4110 – Computer Design Laboratory

Experiment 06

System Timers and General-Purpose Timer Modules

Date: October 2025

6 System Timers and General-Purpose Timer Modules

Learning Objectives

After completing this experiment, you will be able to:

- Understand the role and limitations of the SysTick timer in the ARM Cortex-M4 core.
- Configure the SysTick timer for periodic interrupts and precise timing.
- Understand the architecture and capabilities of General-Purpose Timer Modules (GPTM).
- Configure GPTM in 16-bit and 32-bit modes for periodic and one-shot operation.
- Calculate timer periods using prescalers and reload values.
- Implement timer-based debouncing for mechanical switches.
- Use multiple timers to control independent timing tasks.

Experiment Overview

Precise timing control is essential in embedded systems — whether blinking an LED, reading sensors at regular intervals, or creating accurate delays. The TM4C123 microcontroller provides two powerful timing mechanisms: the SysTick timer (built into the ARM Cortex-M4 core) and the General-Purpose Timer Module (GPTM) peripherals.

The SysTick timer is a simple 24-bit down-counter designed to provide a consistent time base for operating systems and applications. It is ideal for creating system ticks (e.g., 1 ms intervals) and implementing basic delays.

The GPTM peripherals offer much more flexibility: they support 16-bit and 32-bit operation, prescalers for extended periods, multiple operating modes (periodic, one-shot, PWM, input capture), and independent Timer A and Timer B channels within each module.

In this experiment, you will:

- Configure the SysTick timer to generate periodic interrupts.
- Use SysTick for millisecond timing and software debouncing.
- Understand the GPTM architecture and register structure.
- Configure GPTM in 16-bit and 32-bit periodic modes.
- Calculate timer periods using prescalers and reload values.
- Implement multiple independent timing tasks using different timers.

By the end of this lab, you will understand how to select the appropriate timer for different timing requirements, configure timers for precise intervals, and implement timer-based interrupt-driven applications.

Contents

1 Theoretical Background

1.1 Clock Sources

Unless otherwise noted, we assume the **System Clock (SysClk) = 50 MHz** for all calculations in this chapter.

1.1.1 Clock Setup in Keil µVision5

When you create a project in Keil µVision5 for the TM4C123, the **startup file** (`startup_TM4C123.s`) and **system initialization code** (`system_TM4C123.c`) automatically configure the system clock to 50 MHz. This setup occurs before `main()` is called and:

- Enables the PLL (Phase-Locked Loop) to multiply the crystal oscillator frequency
- Configures the system divider to achieve the target 50 MHz frequency
- Sets the `SystemCoreClock` variable to `50000000` for use in CMSIS functions like `SysTick_Config()`

You can verify the clock configuration by checking the value of `SystemCoreClock` in your code or by examining the `SystemInit()` function in `system_TM4C123.c`.

1.1.2 System Clock on TM4C123

The system clock can be derived from multiple sources:

- **MOSC**: Main oscillator (external crystal, typically 16 MHz or 25 MHz)
- **PIOSC**: Precision internal oscillator (16 MHz ± 1%)
- **PIOSC/4**: Internal oscillator divided by 4 (≈ 4 MHz)
- **LFIOSC**: Low-frequency internal oscillator (∼30 kHz)
- **PLL**: Phase-Locked Loop (can multiply oscillator frequency)

Peripherals, including GPTM, are clocked from the system clock (SysClk) once enabled in the `RCGCTIMER` register. The Keil startup file typically configures the PLL to generate 50 MHz from a 16 MHz crystal.

1.1.3 SysTick Clock Source

The SysTick timer can use one of two clock sources, selected via the `CLKSOURCE` bit in the `CTRL` register:

- **Processor clock** (bit = 1): Same as system clock (50 MHz in our case)
- **Reference clock** (bit = 0): Typically PIOSC/4 (≈ 4 MHz), useful for a stable timebase independent of PLL changes

For this lab, we use the processor clock (50 MHz) for both SysTick and GPTM.

1.2 How Timers Work

Understanding the basic operation of timers will help you configure them correctly and debug timing issues.

1.2.1 Down-Counting Operation

Both SysTick and GPTM timers operate as **down-counters** in their most common configuration (periodic mode):

1. **Load**: A reload value is written to the timer's load register (`LOAD` for SysTick, `TAILR` for GPTM)
2. **Count**: The timer counts down from the reload value to zero, decrementing once per clock cycle (or prescaled cycle)

3. **Timeout**: When the counter reaches zero:
   - The timeout flag is set
   - An interrupt is generated (if enabled and unmasked)
   - The counter automatically reloads from the load register (periodic mode) or stops (one-shot mode)
4. **Clear**: The interrupt handler must clear the timeout flag by writing to the interrupt clear register

1.2.2 Timer Modes

Timers can operate in different modes depending on the application:

- **Periodic Mode**: The timer automatically reloads and continues counting. Ideal for fixed-rate tasks like LED blinking, sampling sensors, or OS ticks.
- **One-Shot Mode**: The timer counts down once and stops. Useful for timeouts, delays, or single events.
- **Capture Mode**: (GPTM only) Captures the counter value when an external event occurs on a GPIO pin. Used for measuring pulse widths or frequencies.
- **PWM Mode**: (GPTM only) Generates pulse-width modulated output signals for motor control, dimming LEDs, etc.

1.2.3 Prescaler (16-bit Mode Only)

In 16-bit mode, GPTM provides an 8-bit prescaler (`TAPR`) that extends the timer range by dividing the input clock:

$$T = \frac{(\text{TAILR} + 1) \times (\text{TAPR} + 1)}{f_{\text{SysClk}}}$$

The prescaler is ignored in 32-bit mode. For example, with `TAPR = 255` (divisor = 256), the effective clock frequency is reduced to $50\,\text{MHz}/256 \approx 195.3\,\text{kHz}$.

1.2.4 Interrupt Handling

Proper interrupt handling is critical for timer-based applications:

1. **Enable the interrupt**: Set the appropriate bit in the timer's interrupt mask register (`IMR`)
2. **Enable in NVIC**: Set the corresponding bit in the NVIC's `ISER` register
3. **Implement the ISR**: Write an interrupt service routine with the correct name (e.g., `SysTick_Handler()`, `TIMER1A_Handler()`)
4. **Clear the flag**: Write 1 to the appropriate bit in the interrupt clear register (`ICR`) to acknowledge the interrupt

Best Practice: Clear the interrupt flag early in the ISR to prevent missing subsequent interrupts:

```
void TIMER1A_Handler(void) {
    TIMER1->ICR = (1 << 0);  // Clear TATOCINT flag immediately
    // Perform minimal work: toggle GPIO, update counters, etc.
}
```

1.3 SysTick Timer

The SysTick timer is a 24-bit down-counter that is part of the ARM Cortex-M4 core. It is present in all Cortex-M processors and provides a standard, simple timing mechanism for operating systems and applications.

1.3.1 SysTick Features

The SysTick timer provides:

- **24-bit down-counter**: Counts from a reload value down to zero.
- **Automatic reload**: When the counter reaches zero, it automatically reloads from the `LOAD` register.
- **Optional interrupt**: Can generate an interrupt when the counter reaches zero.
- **Clock source selection**: Can use the processor clock or an external reference clock.
- **CMSIS support**: The CMSIS library provides a simple `SysTick_Config()` function for easy setup.

The SysTick timer is ideal for creating system ticks (e.g., 1 ms intervals for RTOS scheduling), implementing delays, and measuring time intervals. However, its 24-bit width limits the maximum period at high clock frequencies.

1.3.2 SysTick Registers

The SysTick timer is controlled through three main registers:

SysTick Control and Status Register (STCTRL)

Register: `SysTick->CTRL` — SysTick Control and Status (`0xE000E010`)
The `CTRL` register controls the SysTick timer operation and provides status information.

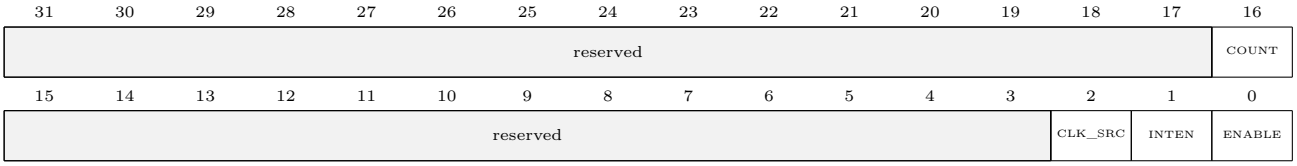

Figure 6.1: SysTick CTRL Register

Key Bits:

- **Bit 0 (ENABLE)**: Enable/disable the SysTick timer
  - 0 = Timer disabled
  - 1 = Timer enabled
- **Bit 1 (INTEN)**: Enable SysTick interrupt
  - 0 = No interrupt when counter reaches zero
  - 1 = Generate interrupt when counter reaches zero
- **Bit 2 (CLK_SRC)**: Clock source selection
  - 0 = External reference clock
  - 1 = Processor clock (typical)
- **Bit 16 (COUNT)**: Counter reached zero since last read (read-only)
  - 0 = Has not counted to zero
  - 1 = Has counted to zero (cleared on read)

SysTick Reload Value Register (STLOAD)

Register: `SysTick->LOAD` — SysTick Reload Value (`0xE000E014`)
The `LOAD` register holds the value that is loaded into the counter when it reaches zero or when the timer is enabled.

Figure 6.2: SysTick LOAD Register — 24-bit Reload Value

RELOAD Field (Bits 23:0): The value to load into the counter. Valid range: `0x000001` to `0xFFFFFF`. Writing zero disables the counter.

SysTick Current Value Register (STCURRENT)

Register: `SysTick->VAL` — SysTick Current Value (`0xE000E018`)
The `VAL` register contains the current value of the SysTick counter.

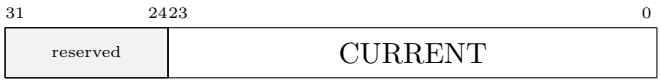

Figure 6.3: SysTick VAL Register — Current Counter Value

CURRENT Field (Bits 23:0): Current counter value. Writing any value clears the counter to zero and clears the `COUNTFLAG` in `CTRL`.

1.3.3 SysTick Timing Calculation

The SysTick timer counts down from the `LOAD` value to zero. The period is determined by:

$$T = \frac{\text{LOAD} + 1}{f_{\text{clock}}}$$

where f_{clock} is the processor clock frequency (typically 50 MHz on the TM4C123).
Example: To generate a 1 ms interrupt at 50 MHz:

$$\text{LOAD} = \frac{T \times f_{\text{clock}}}{1} - 1 = \frac{0.001 \times 50{,}000{,}000}{1} - 1 = 49{,}999$$

Maximum Period: With a 24-bit counter at 50 MHz:

$$T_{\max} = \frac{2^{24}}{50{,}000{,}000} = \frac{16{,}777{,}216}{50{,}000{,}000} \approx 0.335 \text{ seconds}$$

For longer periods, accumulate ticks in software or use the GPTM.

1.3.4 SysTick Configuration with CMSIS

The CMSIS library provides a convenient function to configure the SysTick timer:

```
// Configure SysTick for 1 ms interrupts at SystemCoreClock
SysTick_Config(SystemCoreClock / 1000);  // 50,000 ticks = 1 ms period
```

Listing 6.1: SysTick configuration using CMSIS

This function:

- Sets the `LOAD` register to the specified value minus 1
- Clears the `VAL` register
- Enables the SysTick interrupt in the NVIC
- Enables the SysTick timer with the processor clock source

The SysTick interrupt handler is named `SysTick_Handler()`:

```
void SysTick_Handler(void) {
    // Called every 1 ms
    // Increment global counter, toggle LED, etc.
}
```

1.4 General-Purpose Timer Module (GPTM)

The TM4C123 includes twelve General-Purpose Timer Modules (GPTM): six 16/32-bit timers (TIMER0-TIMER5) and six 32/64-bit wide timers (WTIMER0-WTIMER5). Each module can operate as two independent 16-bit timers or one 32-bit timer.

1.4.1 GPTM Features

GPTM modules provide:

- **16-bit or 32-bit operation**: Configurable timer width
- **Two independent timers**: Timer A and Timer B in each module
- **Multiple operating modes**:
  - One-Shot mode: Timer runs once and stops
  - Periodic mode: Timer reloads automatically
  - Real-Time Clock (RTC) mode: Precise time-keeping
  - PWM mode: Pulse Width Modulation output
  - Input Capture mode: Measure input signal timing
- **Prescaler**: Extends timer range in 16-bit mode (8-bit prescaler for standard timers)
- **Interrupt generation**: Timeout, match, and capture interrupts
- **Trigger and synchronization**: Can trigger ADC conversions or synchronize with other timers

1.4.2 GPTM Architecture

Each GPTM module contains:

- **Timer A**: Independent timer with its own registers
- **Timer B**: Independent timer with its own registers
- **Configuration Register (CFG)**: Selects 16-bit or 32-bit mode
- **Mode Registers (TAMR, TBMR)**: Configure operating mode for each timer
- **Load Registers (TAILR, TBILR)**: Set the reload/start value
- **Prescaler Registers (TAPR, TBPR)**: Extend timer range (16-bit mode only)
- **Control Register (CTL)**: Enable/disable timers and configure behavior
- **Interrupt Registers (IMR, RIS, MIS, ICR)**: Manage interrupts

In 32-bit mode, Timer A operates as a full 32-bit timer, and Timer B is not available. In 16-bit mode, both Timer A and Timer B operate independently as 16-bit timers.

1.4.3 GPTM Configuration Registers

GPTMCFG — Timer Configuration Register

Register: `TIMERx->CFG` — GPTM Configuration (`Base + 0x000`)
The `CFG` register selects the timer width (16-bit or 32-bit mode).

Figure 6.4: GPTMCFG Register — Timer Width Configuration

GPTMCFG Field (Bits 2:0):

- `0x0`: For a 16/32-bit timer, this value selects the 32-bit timer configuration. For a 32/64-bit wide timer, this value selects the 64-bit timer configuration.
- `0x1`: For a 16/32-bit timer, this value selects the 32-bit real-time clock (RTC) counter configuration. For a 32/64-bit wide timer, this value selects the 64-bit real-time clock (RTC) counter configuration.
- `0x2-0x3`: Reserved
- `0x4`: For a 16/32-bit timer, this value selects the 16-bit timer configuration. For a 32/64-bit wide timer, this value selects the 32-bit timer configuration. The function is controlled by bits 1:0 of **GPTMTAMR** and **GPTMTBMR**.
- `0x5-0x7`: Reserved

GPTMTAMR — Timer A Mode Register

Register: `TIMERx->TAMR` — GPTM Timer A Mode (`Base + 0x004`)
The `TAMR` register configures the operating mode for Timer A.

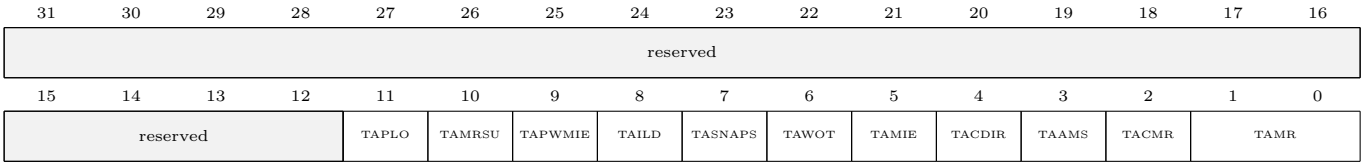

Figure 6.5: GPTMTAMR Register — Timer A Mode Configuration

Key Fields:

- **Bits 1:0 (TAMR)**: Timer A Mode
  - `0x1`: One-Shot mode
  - `0x2`: Periodic mode
  - `0x3`: Capture mode
- **Bit 4 (TACDIR)**: Timer A Count Direction (0 = down, 1 = up)

For this experiment, we will only be using One-Shot and Periodic modes. For more details on other fields, refer to the datasheet.

GPTMTAILR — Timer A Interval Load Register

Register: `TIMERx->TAILR` — GPTM Timer A Interval Load (`Base + 0x028`)
The `TAILR` register sets the start/reload value for Timer A.

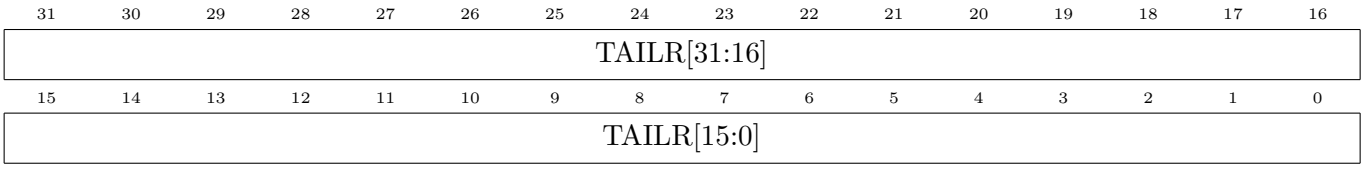

Figure 6.6: GPTMTAILR Register — Timer A Interval Load Value

TAILR Field:

- **Count-down mode**: Specifies the starting count value loaded into the timer
- **Count-up mode**: Sets the upper bound for the timeout event
- **32-bit mode**: Full 32-bit register (upper 16 bits correspond to GPTMTBILR contents)
- **16-bit mode**: Only bits [15:0] are used (upper 16 bits read as 0, no effect on GPTMTBILR)
- **64-bit Wide Timer mode**: Contains bits [31:0] of the 64-bit count (GPTMTBILR contains bits [63:32])

GPTMTAPR — Timer A Prescaler Register

Register: `TIMERx->TAPR` — GPTM Timer A Prescaler (`Base + 0x038`)
The `TAPR` register extends the timer range in 16-bit mode by providing an 8-bit prescaler.

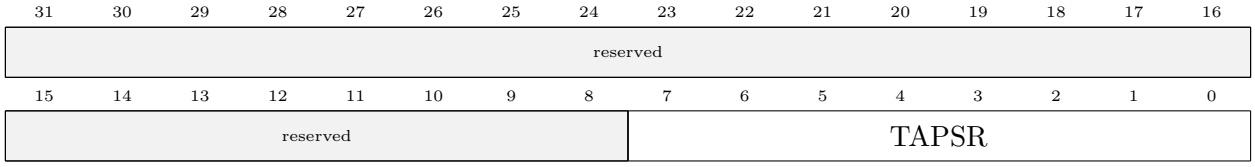

Figure 6.7: GPTMTAPR Register — Timer A Prescaler (16-bit mode only)

TAPSR Field (Bits 7:0):

- Prescaler value: `0x00` to `0xFF` (0 to 255)
- Only used in 16-bit mode; ignored in 32-bit mode
- Effective divisor: `TAPSR + 1`

GPTMCTL — Timer Control Register

Register: `TIMERx->CTL` — GPTM Control (`Base + 0x00C`)
The `CTL` register enables/disables timers and configures their behavior.

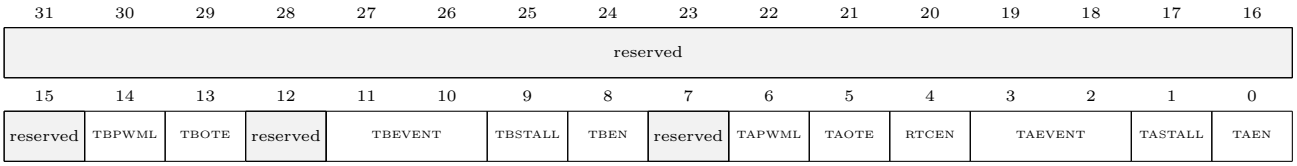

Figure 6.8: GPTMCTL Register — Timer Enable and Control

Key Control Functions:

- **Timer Enable Control**: TAEN (bit 0) and TBEN (bit 8) independently enable/disable Timer A and Timer B
- **Output Trigger Control**: TAOTE (bit 5) and TBOTE (bit 13) enable timers to trigger external peripherals (ADC, other timers, etc.)

- **Independent Operation**: Each timer can be controlled separately, allowing flexible dual-timer configurations

GPTMIMR — Interrupt Mask Register

Register: `TIMERx->IMR` — GPTM Interrupt Mask (`Base + 0x018`)
The `IMR` register enables or disables (masks/unmasks) timer interrupts.

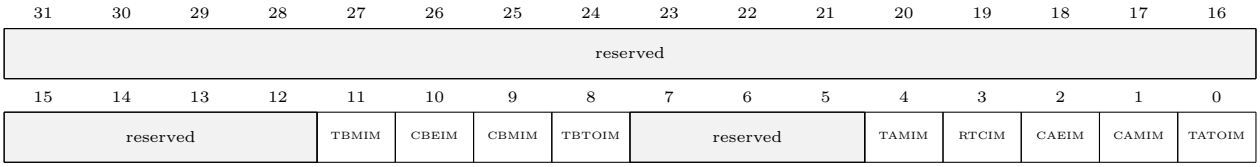

Figure 6.9: GPTMIMR Register — Interrupt Mask

Key Bits:
- **Bit 0 (TATOIM)**: Timer A Timeout Interrupt Mask (0 = masked, 1 = enabled)
- **Bit 8 (TBTOIM)**: Timer B Timeout Interrupt Mask (0 = masked, 1 = enabled)

GPTMICR — Interrupt Clear Register

Register: `TIMERx->ICR` — GPTM Interrupt Clear (`Base + 0x024`)
Writing '1' to a bit in this register clears the corresponding interrupt flag.

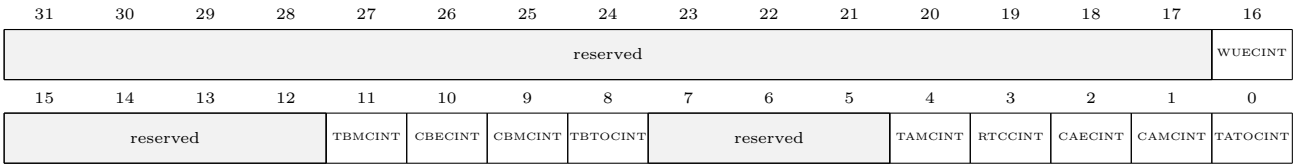

Figure 6.10: GPTMICR Register — Interrupt Clear

1.4.4 GPTM Timing Calculations

The timer period depends on the operating mode and configuration:

32-bit Mode (No Prescaler)

In 32-bit mode, the period is:

$$T = \frac{\text{TAILR} + 1}{f_{\text{clock}}}$$

At 50 MHz, the maximum period is:

$$T_{\max} = \frac{2^{32}}{50{,}000{,}000} = \frac{4{,}294{,}967{,}296}{50{,}000{,}000} \approx 85.9 \text{ seconds}$$

16-bit Mode (With Prescaler)

In 16-bit mode, the prescaler extends the timer range:

$$T = \frac{(\text{TAILR} + 1) \times (\text{TAPR} + 1)}{f_{\text{clock}}}$$

At 50 MHz, with maximum values (`TAILR = 0xFFFF`, `TAPR = 0xFF`):

$$T_{\max} = \frac{65{,}536 \times 256}{50{,}000{,}000} = \frac{16{,}777{,}216}{50{,}000{,}000} \approx 0.335 \text{ seconds}$$

Example: To generate a 500 ms interrupt in 16-bit mode at 50 MHz:

$$\text{TAILR} \times \text{TAPR} = T \times f_{\text{clock}} = 0.5 \times 50{,}000{,}000 = 25{,}000{,}000$$

Choose `TAPR = 255` (prescaler divisor = 256):

$$\text{TAILR} = \frac{25{,}000{,}000}{256} - 1 = 97{,}656 - 1 = 97{,}655 \text{ (exceeds 16-bit range)}$$

This period cannot be achieved in 16-bit mode. Use 32-bit mode instead:

$$\text{TAILR} = 25{,}000{,}000 - 1 = 24{,}999{,}999$$

1.4.5 GPTM Interrupt Numbers

Each GPTM module has a unique interrupt number for Timer A and Timer B; you can find these in the TM4C123 datasheet or in the device headers. The CMSIS-compliant header `tm4c123gh6pm.h` defines these as `IRQn_Type` enumerations (e.g., `TIMER1A_IRQn`), which work directly with the NVIC helper functions.

Enabling an interrupt (CMSIS).

Use `NVIC_EnableIRQ(IRQn_Type irqn)`; CMSIS handles the correct ISER register and bit index for you.

```
#include "TM4C123.h"

// Enable TIMER1A interrupt:
NVIC_EnableIRQ(TIMER1A_IRQn);

// Enable multiple GPTM interrupts:
NVIC_EnableIRQ(TIMER0A_IRQn);
NVIC_EnableIRQ(TIMER1A_IRQn);
NVIC_EnableIRQ(TIMER2A_IRQn);
```

Setting interrupt priority.

Use `NVIC_SetPriority(IRQn_Type irqn, uint32_t priority)` before enabling the interrupt. On Cortex-M4 (TM4C123), priorities are typically 0-7 (0 = highest urgency).

```
// Assign priorities (lower number = higher priority)
NVIC_SetPriority(TIMER0A_IRQn, 3);
NVIC_SetPriority(TIMER1A_IRQn, 4);
NVIC_SetPriority(TIMER2A_IRQn, 5);
```

```
// Then enable them
NVIC_EnableIRQ(TIMER0A_IRQn);
NVIC_EnableIRQ(TIMER1A_IRQn);
NVIC_EnableIRQ(TIMER2A_IRQn);
```

1.5 Configuration Workflow

1.5.1 SysTick Configuration Steps

To configure the SysTick timer:

1. Calculate the reload value: `LOAD = (Period × SystemCoreClock) - 1`
2. Load the value into `SysTick->LOAD`
3. Clear the current value: `SysTick->VAL = 0`
4. Configure `SysTick->CTRL`:
   - Set bit 0 (ENABLE) to enable the timer
   - Set bit 1 (TICKINT) to enable interrupts
   - Set bit 2 (CLKSOURCE) to use processor clock
5. Implement `SysTick_Handler()` to handle interrupts

Or use the CMSIS function:

```
SysTick_Config(SystemCoreClock / 1000);  // 1 ms period
```

1.5.2 GPTM Configuration Steps (Periodic Mode)

To configure a GPTM timer in periodic mode:

1. Enable the timer clock in `SYSCTL->RCGCTIMER`
2. Wait for clock stabilization (3 NOP instructions or check ready bit)
3. Disable the timer: `TIMERx->CTL = 0`
4. Select timer width: `TIMERx->CFG` (`0x00` for 32-bit, `0x04` for 16-bit)
5. Configure mode: `TIMERx->TAMR = 0x02` (periodic mode, count down)
6. Set prescaler (16-bit mode only): `TIMERx->TAPR = value`
7. Set reload value: `TIMERx->TAILR = value`
8. Clear interrupt flag: `TIMERx->ICR = 0x01`
9. Enable timeout interrupt: `TIMERx->IMR |= 0x01`
10. Enable interrupt in NVIC: `NVIC->ISER[n] |= (1 « bit)`
11. Enable the timer: `TIMERx->CTL |= 0x01`
12. Implement the ISR (e.g., `TIMER1A_Handler()`)

Register	Purpose	Typical Value
`RCGCTIMER`	Enable timer clock	`Set bit for TIMERx`
`TIMERx->CFG`	Select width (16-bit vs 32-bit)	`0x00 (32-bit) or 0x04 (16-bit)`
`TIMERx->TAMR`	Set mode	`0x02 (periodic, count down)`
`TIMERx->TAILR`	Set reload value	`From desired period`
`TIMERx->TAPR`	Prescaler (extends range in 16-bit mode)	`0-255 (16-bit only)`
`TIMERx->CTL`	Enable timer	`TAEN = 1 (bit 0)`
`TIMERx->IMR`	Unmask interrupt	`TATOIM = 1 (bit 0)`
`TIMERx->ICR`	Clear interrupt flag (write-1-to-clear)	`TATOCINT = 1 (bit 0)`

Table 6.1: GPTM Configuration Register Summary

2 Procedure

2.1 Examples

The following examples demonstrate SysTick and GPTM configuration and usage.

2.1.1 Example 1 — Millisecond Counter with SysTick Timer

This example configures the SysTick timer to generate interrupts every 100 ms and toggles the green LED (PF3) in the interrupt handler.

```
#include "TM4C123.h"

#define GREEN_LED 0x08          // PF3 - Green LED

volatile uint32_t systick_counter = 0;  // Global counter for SysTick interrupts

int main(void)
{
  SYSCTL->RCGCGPIO |= (1<<5);     // Enable clock to GPIOF

  __asm__("NOP");
  __asm__("NOP");
  __asm__("NOP");

  GPIOF->DIR |= GREEN_LED;       // Set green LED as output pin
  GPIOF->DEN |= GREEN_LED;       // Enable digital function for green LED


  SysTick->LOAD = 5000000 - 1;       // Set reload value for 1ms
  SysTick->VAL = 0;            // Clear current value
  SysTick->CTRL = 0x07;          // Enable SysTick with processor clock and interrupt

  while(1)                 // Main loop
  {
  }
}

void SysTick_Handler(void)
{
  systick_counter++;           // Increment counter every 1ms
  GPIOF->DATA ^= GREEN_LED;      // Toggle green LED
}
```

Listing 6.2: SysTick timer example — LED blink every 100ms

Explanation:

- `SysTick->LOAD = 5000000 - 1;` sets the reload value for 100 ms at 50 MHz.
- `SysTick->VAL = 0;` clears the current counter value.
- `SysTick->CTRL = 0x07;` enables the timer with processor clock and interrupt.
- `SysTick_Handler()` is called every 100 ms and toggles the LED.
- `systick_counter` tracks the number of interrupts (can be used for longer timing).

2.1.2 Example 2 — Maximum 16-bit Delay with GPTM

This example configures TIMER1 in 16-bit periodic mode with maximum prescaler to achieve the longest possible delay and toggles the blue LED (PF2).

```
#include "TM4C123.h"

#define BLUE_LED 0x04


int main(void)
{
  SYSCTL->RCGCGPIO |= (1<<5);     // Enable clock to GPIOF
  SYSCTL->RCGCTIMER |= (1<<1);    // Enable clock to Timer1

  __asm__("NOP");           // Wait for clock stabilization
  __asm__("NOP");           // Wait for clock stabilization
  __asm__("NOP");           // Wait for clock stabilization


  GPIOF->DIR |= BLUE_LED;       // Set blue LED as output pin
  GPIOF->DEN |= BLUE_LED;       // Enable digital function for blue LED

  TIMER1->CTL = 0;              // Disable the timer
  TIMER1->CFG = 0x4;            // Choose 16-bit mode
  TIMER1->TAMR = 0x02;            // Periodic mode
  TIMER1->TAPR = 256  - 1;      // Set prescaler value
  TIMER1->TAILR = 65536 - 1;      // Set initial reload value
  TIMER1->ICR = 0x1;              // Clear any prior interrupts
  TIMER1->IMR |=(1<<0);       // Enable timeout interrupt
  NVIC->ISER[0] |= (1<<21);     // Enable Timer1A interrupt in NVIC
  TIMER1->CTL |= 0x01;              // Enable the timer

    while(1)              // Main loop
    {
    }
}


void TIMER1A_Handler()
{
  if(TIMER1->MIS & 0x1) {       // Check if timer timeout interrupt occurred
    GPIOF->DATA  ^= BLUE_LED;   // Toggle blue LED
    TIMER1->ICR = 0x1;        // Clear timer interrupt flag
  }
}
```

Listing 6.3: GPTM Timer1A example — 16-bit mode with prescaler

Explanation:

- `SYSCTL->RCGCTIMER |= (1<<1);` enables clock to TIMER1.
- `TIMER1->CFG = 0x4;` selects 16-bit mode.
- `TIMER1->TAMR = 0x02;` configures periodic mode (count down).
- `TIMER1->TAPR = 256 - 1;` sets prescaler to maximum (divisor = 256).
- `TIMER1->TAILR = 65536 - 1;` sets interval load to maximum (65536).
- Period: $T = \frac{65536 \times 256}{50{,}000{,}000} \approx 0.335$ seconds.
- `NVIC->ISER[0] |= (1<<21);` enables TIMER1A interrupt (IRQ 21).

- `TIMER1A_Handler()` toggles the blue LED and clears the interrupt flag.

2.2 Tasks

2.2.1 Task 1 — Debouncing a Push Button with SysTick

Modify the SysTick example so that the RED LED (PF1) toggles only when SW1 (PF4) is pressed and properly debounced using the SysTick timer.

Requirements:

- Use `SysTick_Config(SystemCoreClock/1000)` to generate a 1 ms tick.
- Implement interrupt-driven GPIO input for SW1 (PF4) as in Experiment 5.
- In the GPIO ISR, use a global millisecond counter to implement 30 ms debouncing.
- Only toggle the RED LED if at least 30 ms have elapsed since the last press.

Hint: Declare global variables:

```
volatile uint32_t global_ms = 0;
volatile uint32_t last_press_ms = 0;
```

In `SysTick_Handler()`, increment `global_ms`. In `GPIOF_Handler()`, check:

```
if ((global_ms - last_press_ms) >= 30) {
    GPIOF->DATA ^= (1 << 1);  // Toggle RED LED
    last_press_ms = global_ms;
}
```

2.2.2 Task 2 — Multiple Blinking LEDs with GPTM

Use three GPTM timers (TIMER0A, TIMER1A, TIMER2A) to blink the three LEDs at different rates:

- **RED LED (PF1)**: Blink every 250 ms (use TIMER0A)
- **BLUE LED (PF2)**: Blink every 500 ms (use TIMER1A)
- **GREEN LED (PF3)**: Blink every 1000 ms (use TIMER2A)

Requirements:

- Configure each timer in 32-bit periodic mode.
- Calculate the `TAILR` values for each period at 50 MHz.
- Enable interrupts for each timer in the NVIC.
- Implement separate ISRs (`TIMER0A_Handler()`, `TIMER1A_Handler()`, `TIMER2A_Handler()`).
- Each ISR should toggle its corresponding LED and clear the interrupt flag.