香港中文大學
The Chinese University of Hong Kong

CENG2400 Embedded System Design

Lecture 04: Concurrency (II)

Ming-Chang YANG

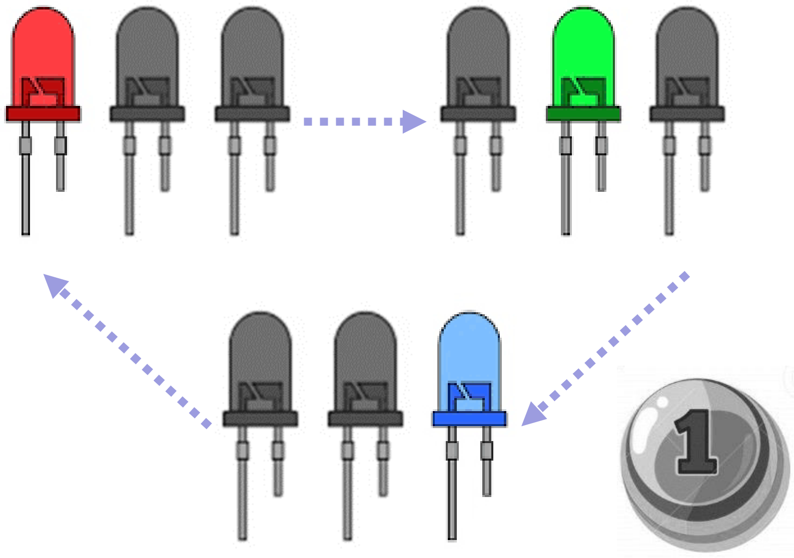
mcyang@cse.cuhk.edu.hk

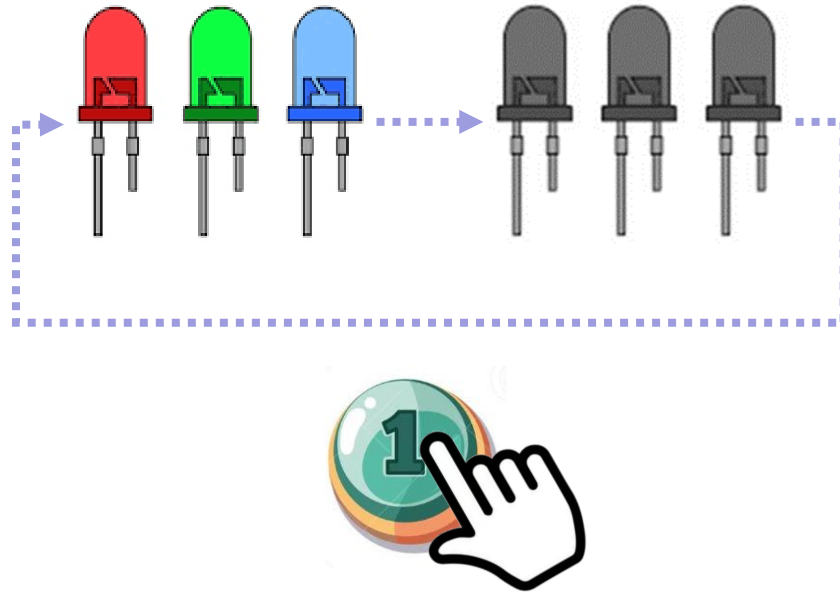
Thanks to Prof. Q. Xu and Drs. K. H. Wong, Philip Leong, Y.S. Moon, O. Mencer, N. Dulay, P. Cheung for some of the slides used in this course!

Outline

- **Review**
  - Working Example: LED Flasher
  - Restructured Program
  - FSM-structured Program
- **Working Example: LED Flasher (Cont'd)**
  - Polling vs. Event-Triggering
    - Interrupt
  - Switch-triggered Program
  - Timer-triggered Program
    - Timer Peripherals
- **Preview: Lab03**

Recall: LED Flasher





Switch 1 Not Pressed: *Display a repeating sequence colors (**red**, then **green**, then **blue**).*

Switch 1 Pressed: *Make the LEDs flash white (**all LEDs on**) and off (**all LEDs off**).*

Switch 2 Pressed: *Use **faster** timing for the RGB sequences and the flashing.*

Recall: Lec03 Concurrency (I)

	Modularity	Responsiveness	CPU Overhead
Starter Program	Poor	Poor	Poor
Restructured Program	Good	Poor	Poor
FSM-structured Program	Good	Better	Poor

- We started with a "**starter program**".
- The "**restructured program**" (which restructured the code into separate tasks) enhanced the modularity.
- The "**FSM-structured program**" further split up the tasks into subtasks, achieving better responsiveness.
- However, they all relied on the "delay" function, making the CPU busy-waiting.

Recall: CPU Overhead

- The µP wastes quite a bit of time in its delay function.
  - This kind of waiting is called busy-waiting and should be avoided except for certain special cases.

```
void Delay(unsigned int time_del) {
    volatile int n;
    while (time_del--) {
        n = 1000;
        while (n--)
            ;
    }
}
```

> **Busy-waiting:** Wasteful method of making a program wait for an event or delay. Program executes test code repeatedly in a tight loop, not sharing time with other parts of program.

Recall: Restructured Program

```
void Flasher(void) { // a simple "scheduler" that repeatedly calls tasks in order
    while (1) {
        ① Task_Read_Switches(); // poll switches to determine mode & delay
        ② Task_Flash();         // only run task when in flash mode
        ③ Task_RGB();           // only run task when NOT in flash mode
    }
}
```

```
#define W_DELAY_SLOW 400
#define W_DELAY_FAST 200
#define RGB_DELAY_SLOW 4000
#define RGB_DELAY_FAST 1000

uint8_t g_flash_LED = 0;// init: RGB mode
uint32_t g_w_delay = W_DELAY_SLOW;
uint32_t g_RGB_delay = RGB_DELAY_SLOW;

① void Task_Read_Switches(void) {
    if (SWITCH_PRESSED(SW1_POS)) {
        g_flash_LED = 1; // flash
    } else {
        g_flash_LED = 0; // RGB
    }
    if (SWITCH_PRESSED(SW2_POS)) {
        w_delay = W_DELAY_FAST;
        RGB_delay = RGB_DELAY_FAST;
    } else {
        w_delay = W_DELAY_SLOW;
        RGB_delay = RGB_DELAY_SLOW;
    }
}
```

```
② void Task_Flash(void) {
    if (g_flash_LED == 1) {
        Control_RGB_LEDs(1, 1, 1);
        Delay(g_w_delay);
        Control_RGB_LEDs(0, 0, 0);
        Delay(g_w_delay);
    }
}
```

```
③ void Task_RGB(void) {
    if (g_flash_LED == 0) {
        Control_RGB_LEDs(1, 0, 0);
        Delay(g_RGB_delay);
        Control_RGB_LEDs(0, 1, 0);
        Delay(g_RGB_delay);
        Control_RGB_LEDs(0, 0, 1);
        Delay(g_RGB_delay);
    }
}
```

Recall: FSM-structured Program

```
void Flasher(void) {
    while (1) {
        ① Task_Read_Switches();
        ② Task_Flash_FSM();
        ③ Task_RGB_FSM();
    }
}
```

```
void Task_RGB_FSM(void) {
    static enum {ST_RED, ST_GREEN, ST_BLUE,
                 ST_OFF} next_state;
    if (g_flash_LED == 0) {
        switch (next_state) {
            case ST_RED:
                Control_RGB_LEDs(1, 0, 0);
                Delay(g_RGB_delay);
                next_state = ST_GREEN;
                break;
            case ST_GREEN:
                Control_RGB_LEDs(0, 1, 0);
                Delay(g_RGB_delay);
                next_state = ST_BLUE;
                break;
            case ST_BLUE:
                Control_RGB_LEDs(0, 0, 1);
                Delay(g_RGB_delay);
                next_state = ST_RED;
                break;
            default:
                next_state = ST_RED;
                break;
        }
    }
}
```

②

```
void Task_Read_Switches(void) {
    if (SWITCH_PRESSED(SW1_POS)) {
        g_flash_LED = 1;
    } else {
        g_flash_LED = 0;
    }
    if (SWITCH_PRESSED(SW2_POS)) {
        g_w_delay = W_DELAY_FAST;
        g_RGB_delay = RGB_DELAY_FAST;
    } else {
        g_w_delay = W_DELAY_SLOW;
        g_RGB_delay = RGB_DELAY_SLOW;
    }
}
```

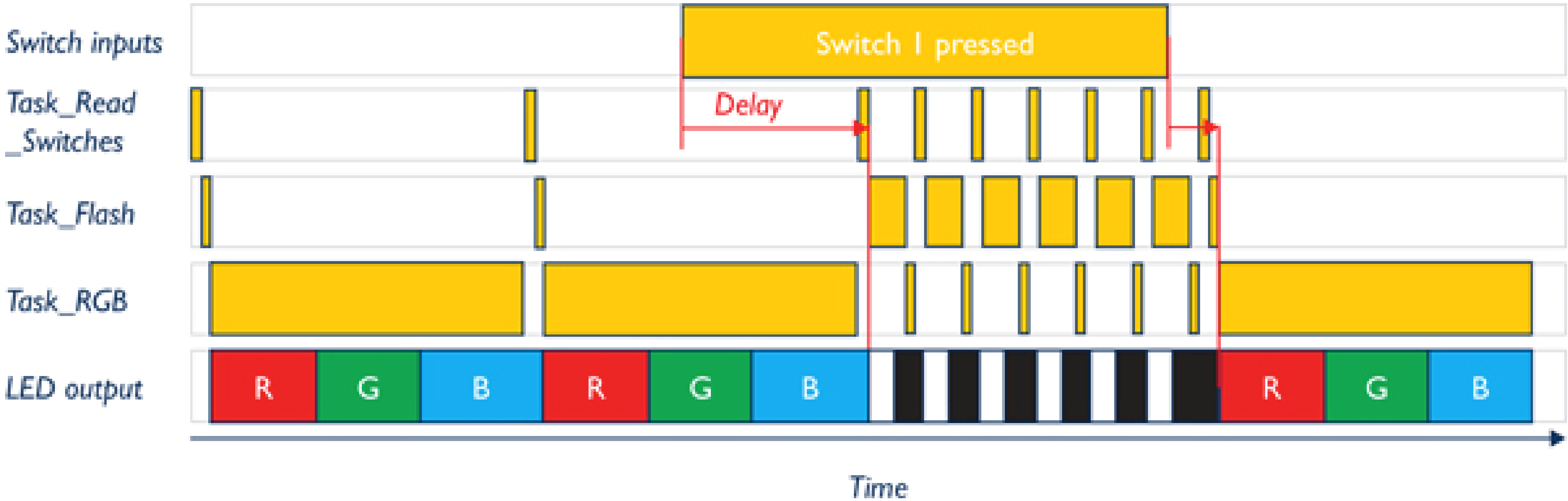
①

```
void Task_Flash_FSM(void) {                    // next_state_w
    static enum {ST_WHITE, ST_BLACK} next_state
                                     = ST_WHITE;
    if (g_flash_LED == 1) {
        switch (next_state) {
            case ST_WHITE:
                Control_RGB_LEDs(1, 1, 1);
                Delay(g_w_delay);
                next_state = ST_BLACK;
                break;
            case ST_BLACK:
                Control_RGB_LEDs(0, 0, 0);
                Delay(g_w_delay);
                next_state = ST_WHITE;
                break;
            default:
                next_state = ST_WHITE;
                break;
        }
    }
}
```

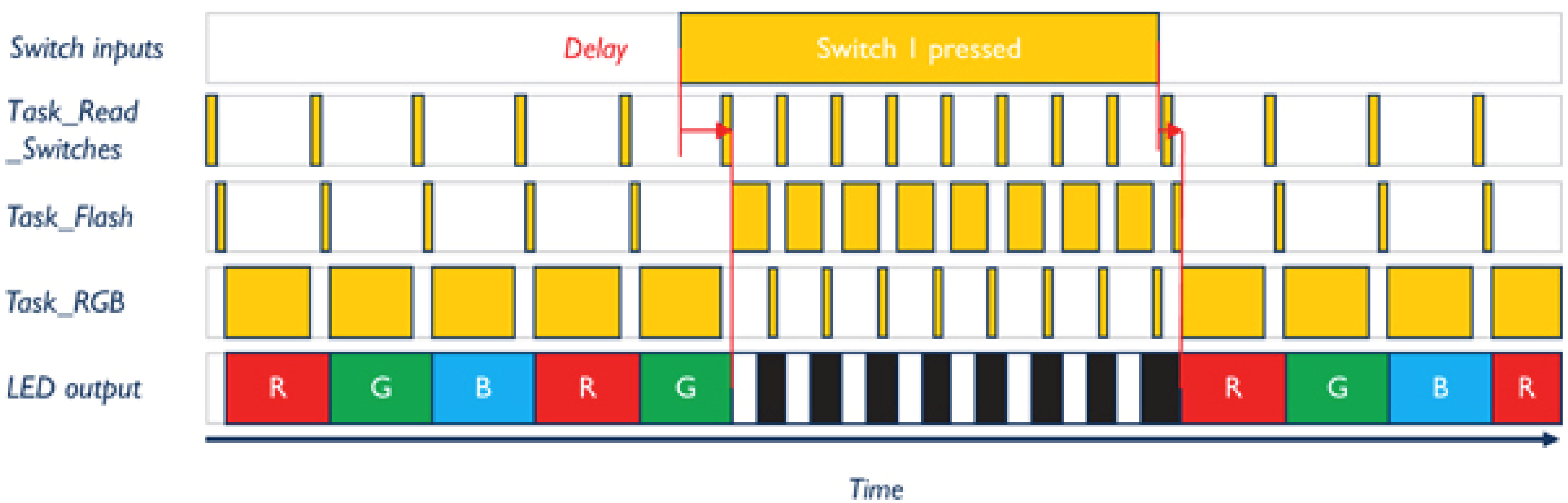
③

Recall: Restructured vs FSM-structured

- **Restructured Program:**



- **FSM-structured Program:**



Outline

- **Review**
  - Working Example: LED Flasher
  - Restructured Program
  - FSM-structured Program
- **Working Example: LED Flasher (Cont'd)**
  - Polling vs. Event-Triggering
    - Interrupt
  - Switch-triggered Program
  - Timer-triggered Program
    - Timer Peripherals
- **Preview: Lab03**

Polling vs. Event-Triggering

- The starter, restructured, and FSM-structured codes are all implemented based on the **polling** approach.
  - They explicitly check to see if processing is needed.

> **Polling:** Scheduling approach in which software repeatedly tests a condition to determine whether to run task code.

- There is an alternative way called **event-triggering**.
  - The processor gets notifications from hardware that a specific event has occurred, and processing is needed.

> **Event-Triggering:** Scheduling approach in which task software runs only when triggered by an event

Why Event-Triggering?

- Software that uses **event-triggering** runs much more efficiently than polling.
  - Why? No time needs to be wasted checking to see if processing is needed.

- Even better, the **event-triggered** approach leads to a much more responsive system.
  - Why? Events are detected much sooner.

- This may allow a much slower processor to be used, saving money, power, and energy.

Key to Event-Triggering: Interrupt

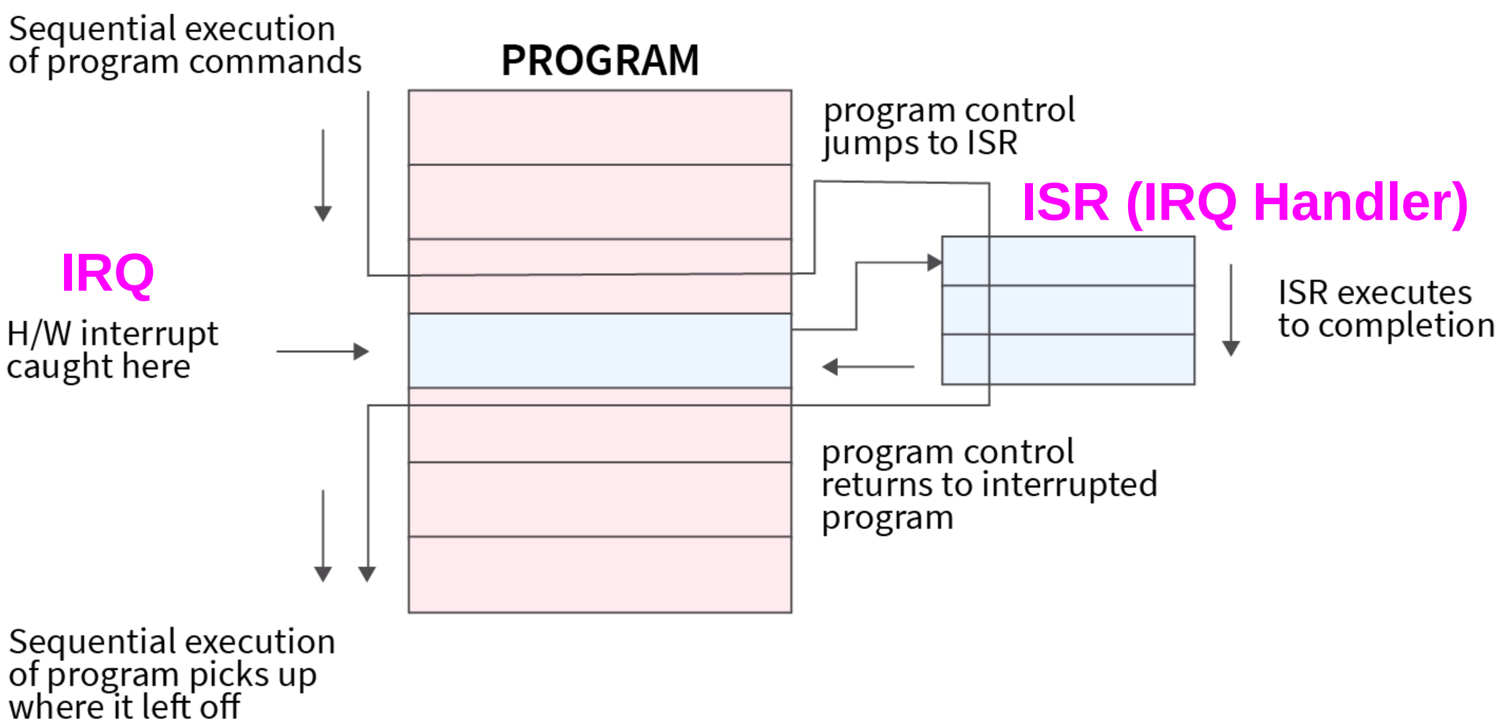
- Peripheral hardware on MCU explicitly supports this event-triggered approach via the **interrupt system**:
  - A peripheral can generate an interrupt request (IRQ) to the processor to indicate that an event has occurred.
  - To service that interrupt request, the processor will execute a special part of the program called an interrupt service routine (ISR) (or interrupt handler).

Interrupt Request (IRQ): Hardware signal indicating that an interrupt is requested

Interrupt Service Routine (ISR): Software routine which runs in response to interrupt request.

How interrupt works? *(more in Lec05)*

- Upon the completion of one "instruction" execution, the processor checks if an IRQ is pending or not.
  - *No?* Continue with the next instruction in the program.
  - *Yes?* Execute the corresponding ISR to handle that IRQ, then resume the suspended program.

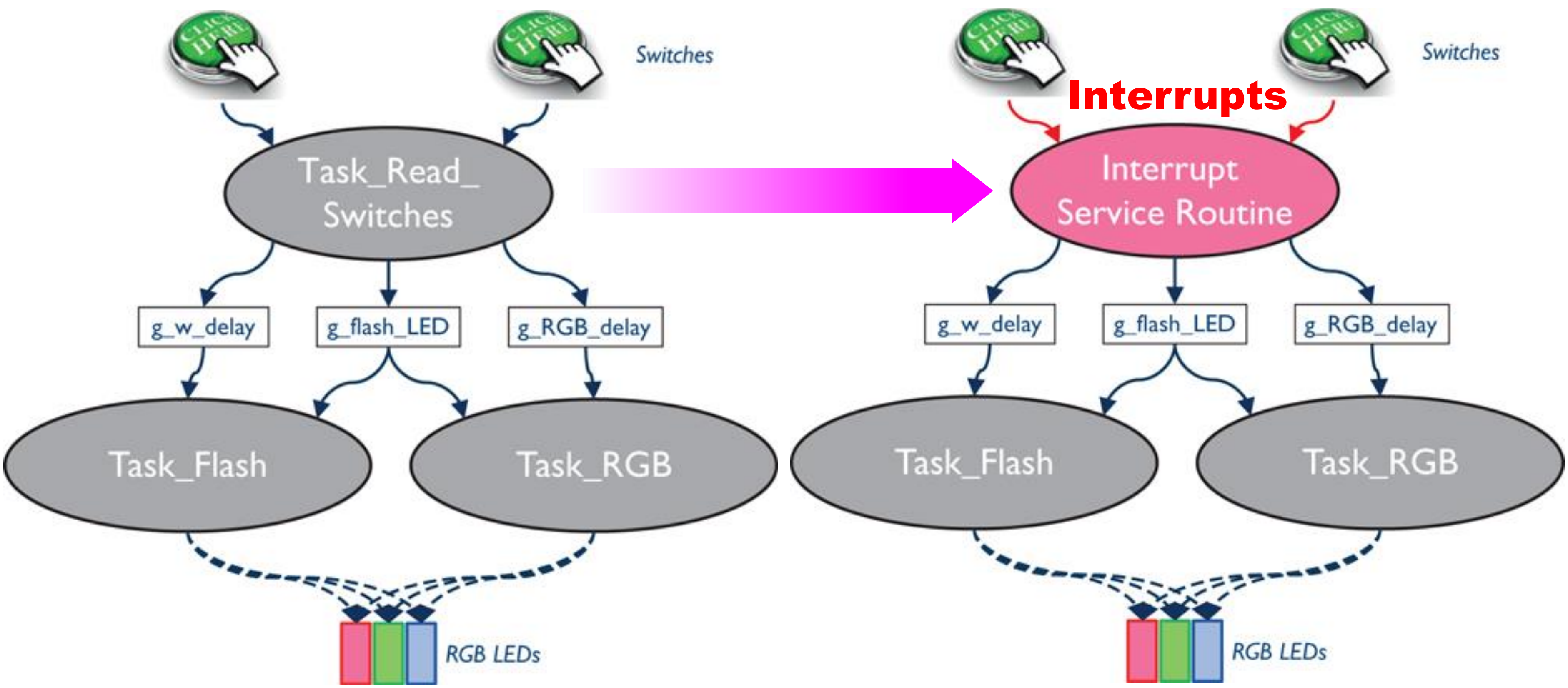


Outline

- **Review**
  - Working Example: LED Flasher
  - Restructured Program
  - FSM-structured Program
- **Working Example: LED Flasher (Cont’d)**
  - Polling vs. Event-Triggering
    - Interrupt
  - Switch-triggered Program
  - Timer-triggered Program
    - Timer Peripherals
- **Preview: Lab03**

Let's make switches interrupt!

- The task of `Read_Switches` is handled by an **ISR**.
  - The **ISR** updates the shared variables (e.g., `g_w_delay`, `g_flash_LED`, `g_RGB_delay`) which are read by other tasks.



Switch-triggered Program (1/2)

- The shared variables are updated in an **IRQ handler**:

```
void Task_Read_Switches(void) {
        if (SWITCH_PRESSED(SW1_POS)) {
                g_flash_LED = 1;
        } else {
                g_flash_LED = 0;
        }
        if (SWITCH_PRESSED(SW2_POS)) {
                w_delay = W_DELAY_FAST;
                RGB_delay = RGB_DELAY_FAST;
        } else {
                w_delay = W_DELAY_SLOW;
                RGB_delay = RGB_DELAY_SLOW;
        }
}
```

```
void PORTD_IRQHandler(void) {
    // Read switches
    if ((PORTD->ISFR & MASK(SW1_POS))) {
    // check if IRQ is triggered by SW1
        if (SWITCH_PRESSED(SW1_POS)) {
            g_flash_LED = 1;
        } else {
            g_flash_LED = 0;
        }
    }
    if ((PORTD->ISFR & MASK(SW2_POS))) {
    // check if IRQ is triggered by SW2
        if (SWITCH_PRESSED(SW2_POS)) {
            g_w_delay = W_DELAY_FAST;
            g_RGB_delay = RGB_DELAY_FAST;
        }   else {
            g_w_delay = W_DELAY_SLOW;
            g_RGB_delay = RGB_DELAY_SLOW;
        }
    }
    // clear status flags
    PORTD->ISFR = 0xffffffff;
    // It is often the responsibility of
    IRQ handler to clear the interrupt,
}   marking that the IRQ is serviced.
```

Note 1: The code needed to configure the port peripheral to request an interrupt and to enable interrupts is not shown.

Note 2: The shared variables should be defined as volatile, indicating to the compiler that they may change unexpectedly.

Switch-triggered Program (2/2)

- The main scheduler no longer needs to call `Task_Read_Switches`:

```
void Flasher(void) {
        while (1) {
                Task_Read_Switches();
                Task_Flash();
                Task_RGB();
        }
}
```

```
void Flasher(void) {
        while (1) {
                Task_Flash();
                Task_RGB();
        }
}
```

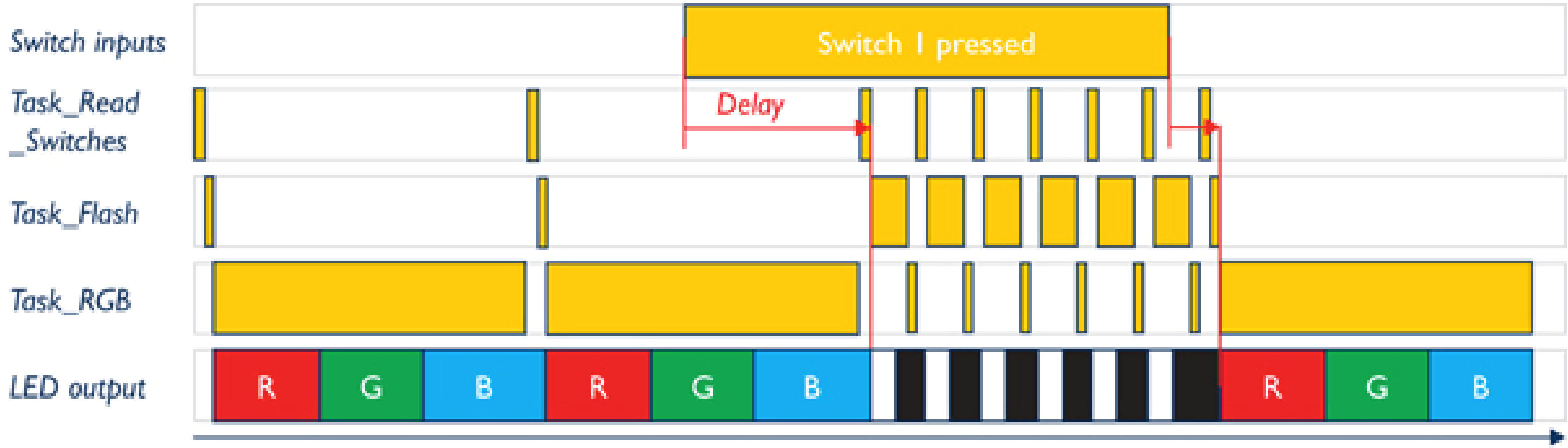
- Other two tasks (`Task_Flash` & `Task_RGB`) remain unchanged:

```
void Task_Flash(void) {
        if (g_flash_LED == 1) {
                Control_RGB_LEDs(1, 1, 1);
                Delay(g_w_delay);
                Control_RGB_LEDs(0, 0, 0);
                Delay(g_w_delay);
        }
}
```

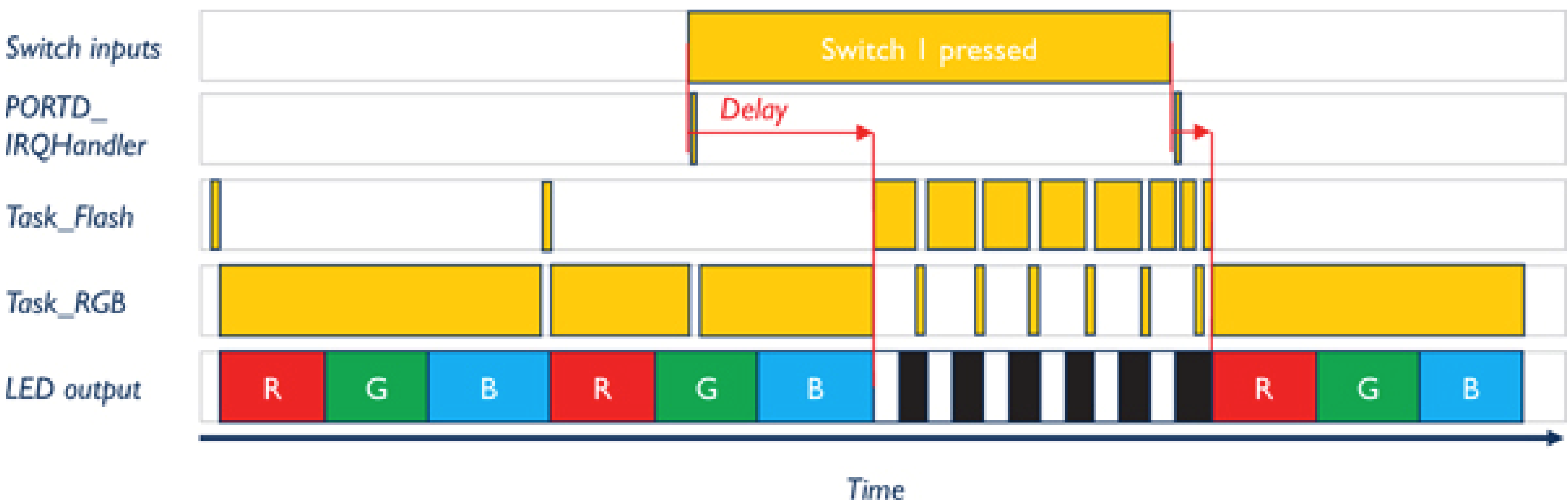
```
void Task_RGB(void) {
        if (g_flash_LED == 0) {
                Control_RGB_LEDs(1, 0, 0);
                Delay(g_RGB_delay);
                Control_RGB_LEDs(0, 1, 0);
                Delay(g_RGB_delay);
                Control_RGB_LEDs(0, 0, 1);
                Delay(g_RGB_delay);
        }
}
```

How good is it now? (1/2)

- **Restructured Program:**

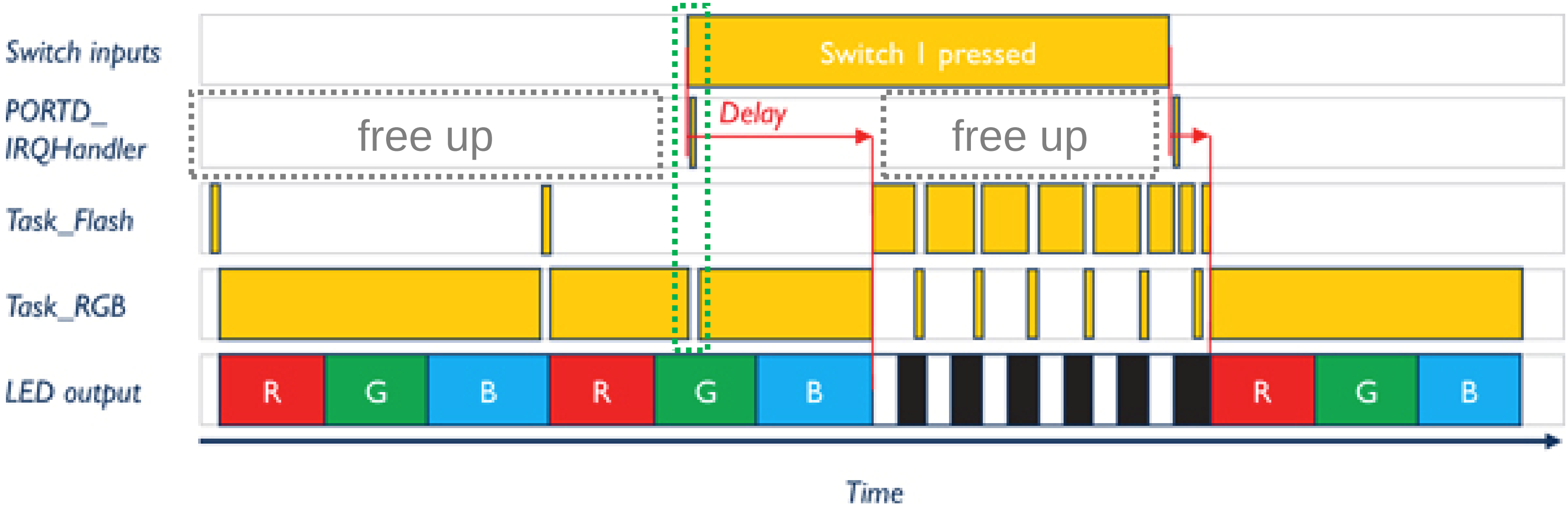


- **Switch-triggered Program:**



How good is it now? (2/2)

- **Switch-triggered Program**:



- **Responsiveness**: MCU can respond to interrupts quickly.
  - The delay between switch pressing to variable updating is reduced.
  - Recall: We may reduce the time switching to different tasks by FSM.
- **CPU Overhead**: The to read switches now executes only when needed (in the IRQ handler).
  - This frees up the time of μP on checking switches.
  - The code still relies on the “delay” function, making μP busy-waiting.

Class Exercise 4.1

- Consider the "switch-triggered program." Assume the IRQ handler starts executing as soon as the switch changes. Also assume there is no time taken to switch between tasks or the handler, and that the tasks and handler have the following execution times:

Task or handler	Execution time when in flash mode	Execution time when in RGB mode
PORTD_IRQ_Handler	0.01 ms	0.01 ms
Task_Flash	100 ms	1 ms
Task_RGB	1 ms	1000 ms

① Describe the sequence that leads to maximum delay between **pressing** the switch and seeing the **LED flash**. Calculate the value of that delay.

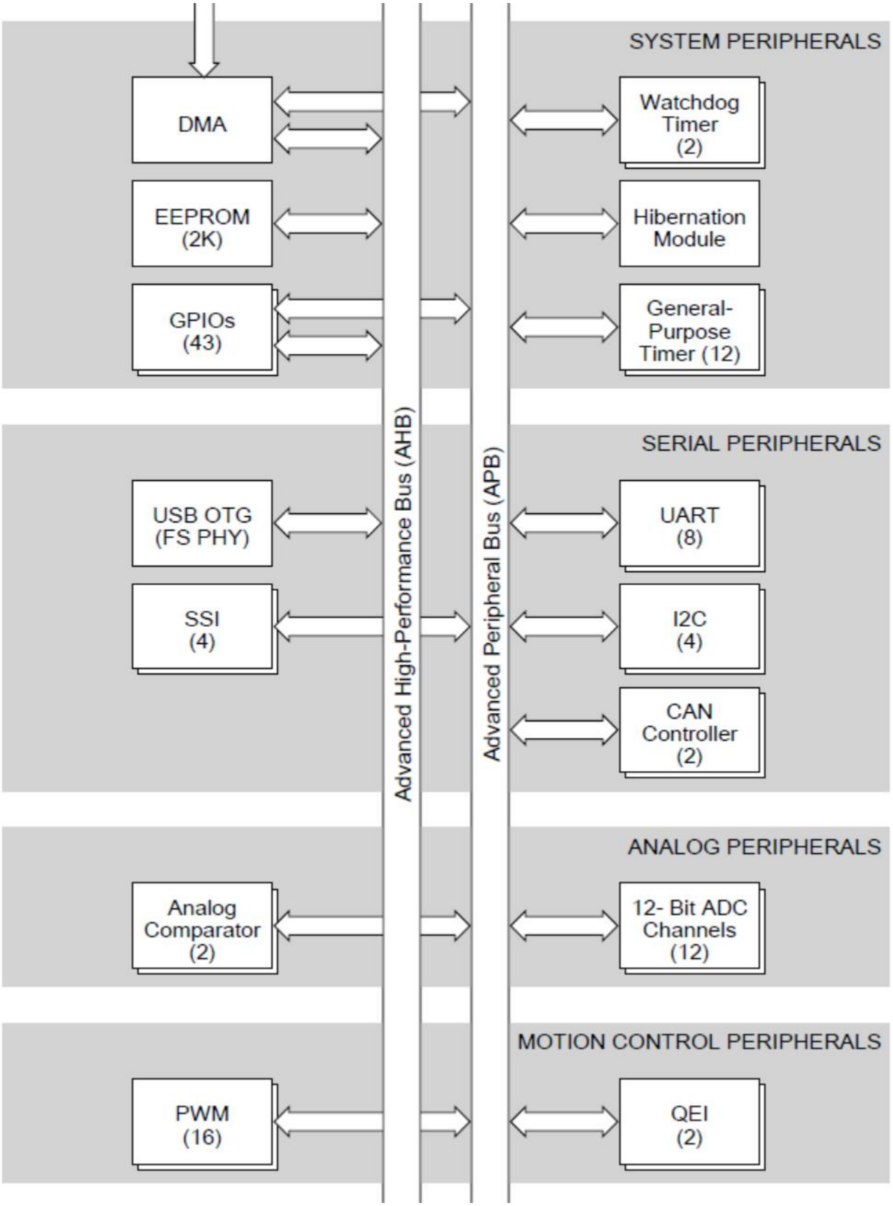
② Describe the sequence that leads to maximum delay between **releasing** the switch and seeing the **LED sequence RGB colors**. Calculate the value of that delay.

Outline

- **Review**
  - Working Example: LED Flasher
  - Restructured Program
  - FSM-structured Program
- **Working Example: LED Flasher (Cont'd)**
  - Polling vs. Event-Triggering
    - Interrupt
  - Switch-triggered Program
  - Timer-triggered Program
    - Timer Peripherals
- **Preview: Lab03**

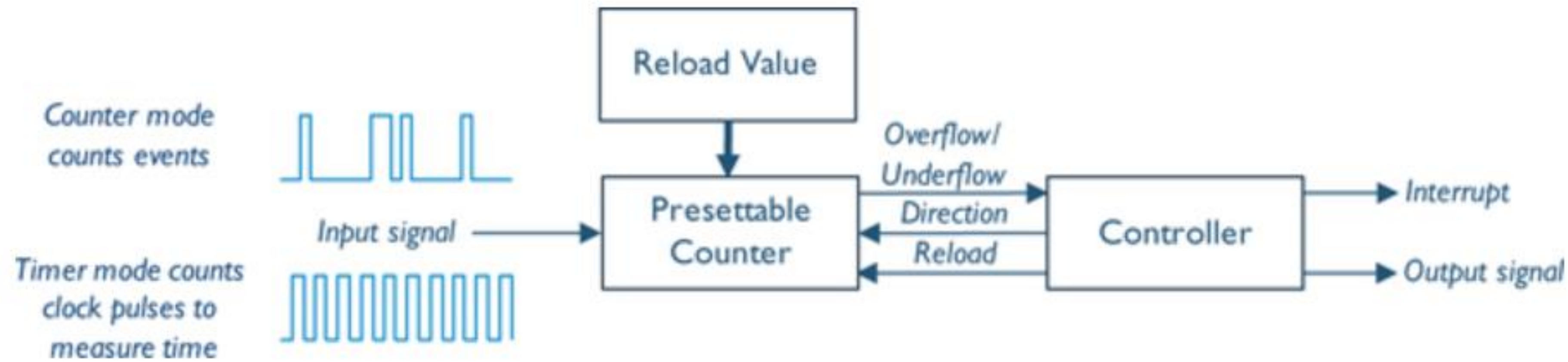
Let’s ask the hardware for help!

- Recall: There are specialized hardware peripherals in the MCU.
  - They can offload and accelerate specific tasks from the µP.
  - They can execute independently of the µP.
  - Note: The µP may be capable of doing these tasks in software, but with higher complexity or less accuracy.



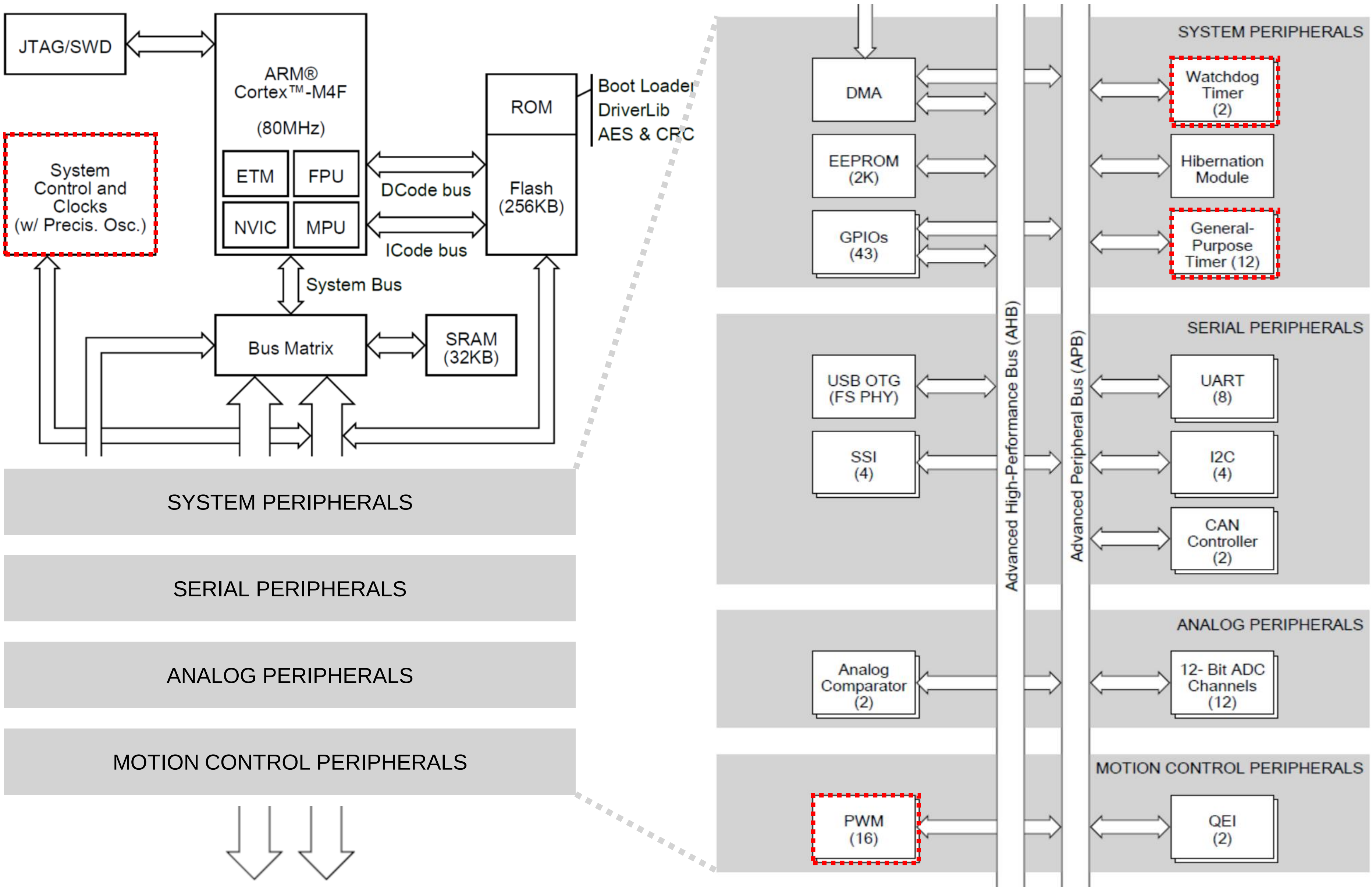
Timer Peripheral

- One common peripheral is called a **timer**.
  - At its heart, it is a counter circuit that counts *how many pulses it receives*.
    - A transition on input may represent an event or a fixed time interval.
    - The transition causes the counter to change (i.e., increment by one).



    - Timer peripherals can therefore count events, measure elapsed time, generate events at fixed times, generate waveforms, or measure pulse widths and frequencies.
    - Other hardware is often added to make it even more flexible.

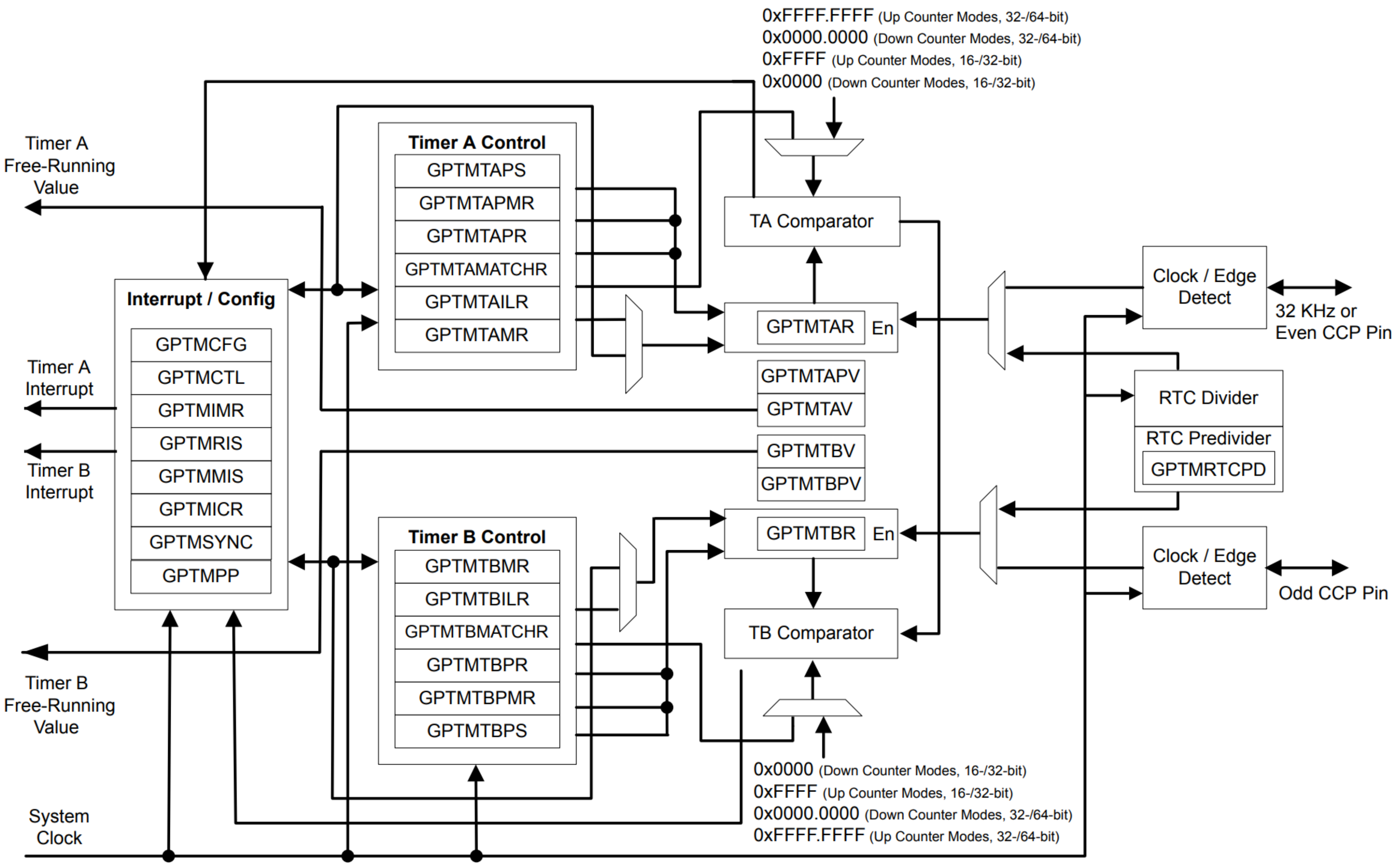
Timer Peripherals on TM4C123GH6PM (1/2)



Timer Peripherals on TM4C123GH6PM (2/2)

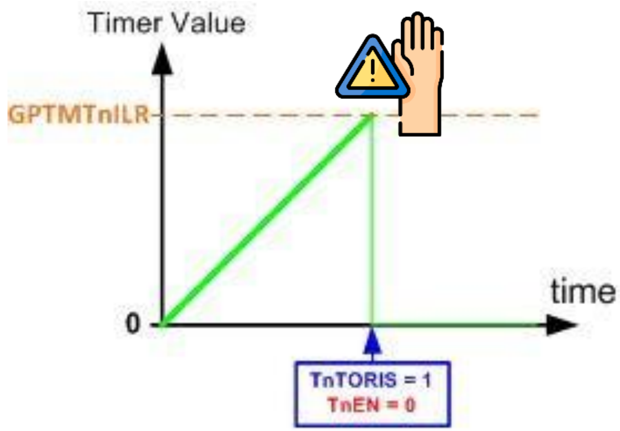
- **SysTick** (§3.3 in MCU Data Sheet; §28 in TivaWare Library)
  - Provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism.
- **General-Purpose Timers** (§11 in MCU Data Sheet; §29 in TivaWare Library)
  - Provides programmable timers to count or time events.
  - There are six 16/32-bit GPTM & six 32/64-bit Wide GPTM, each module (M) provides two timers/counters (Timer A/B).
    - **Modes**: one-shot, periodic, prescaler, real-time, count-up/down, etc.
- **Watchdog Timers** (§12 in MCU Data Sheet; §33 in TivaWare Library)
  - Used to regain control when a system has failed.
  - There are two Watchdog Timer Modules (Watchdog Timer 0/1).
- **Pulse Width Modulator** (§20 in MCU Data Sheet; §21 in TivaWare Library)
  - Used for digitally encoding analog signal levels.
  - There are two PWM modules, each with four PWM generator blocks and a control block.

GPTM Block Diagram

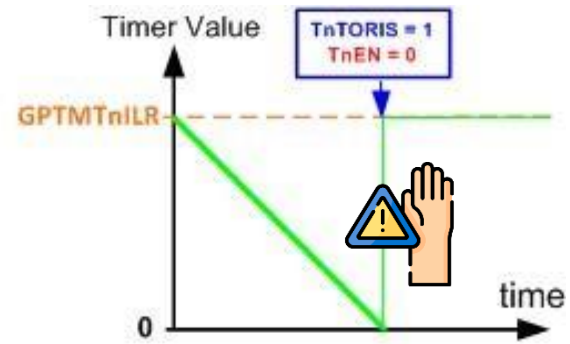


One-shot Mode vs. Periodic Mode

- **One-Shot Mode**: Trigger an event only once

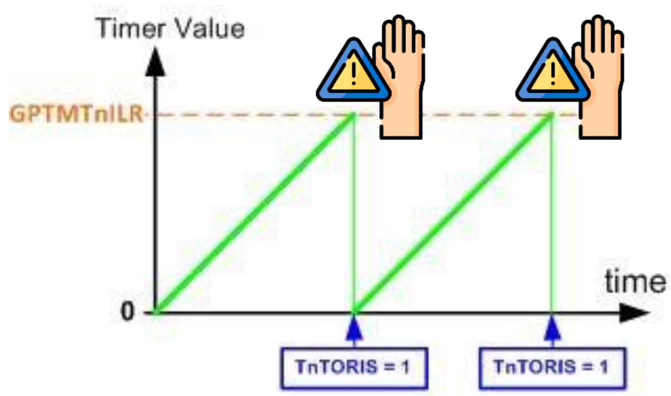


(a) up counting

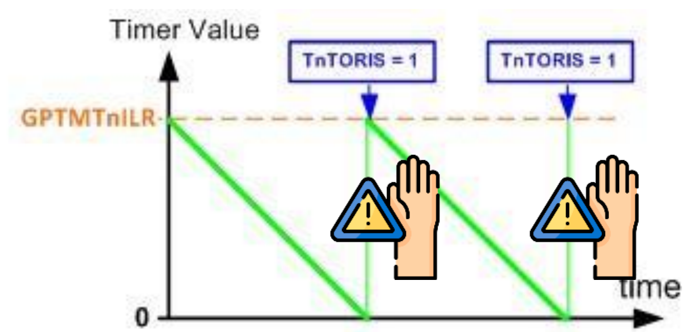


(b) down counting

- **Periodic Mode**: Trigger events at regular intervals



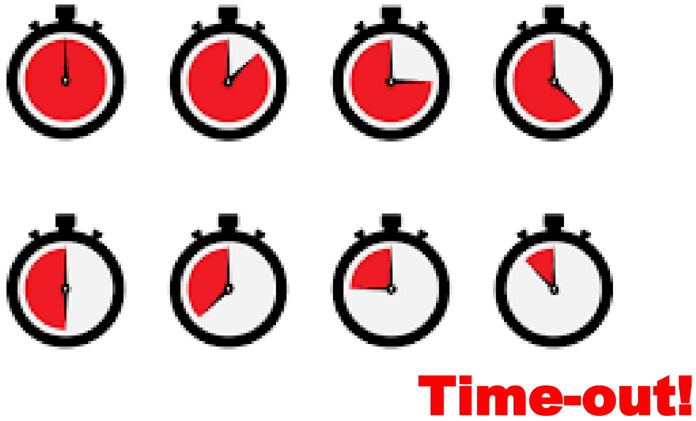
(a) up counting

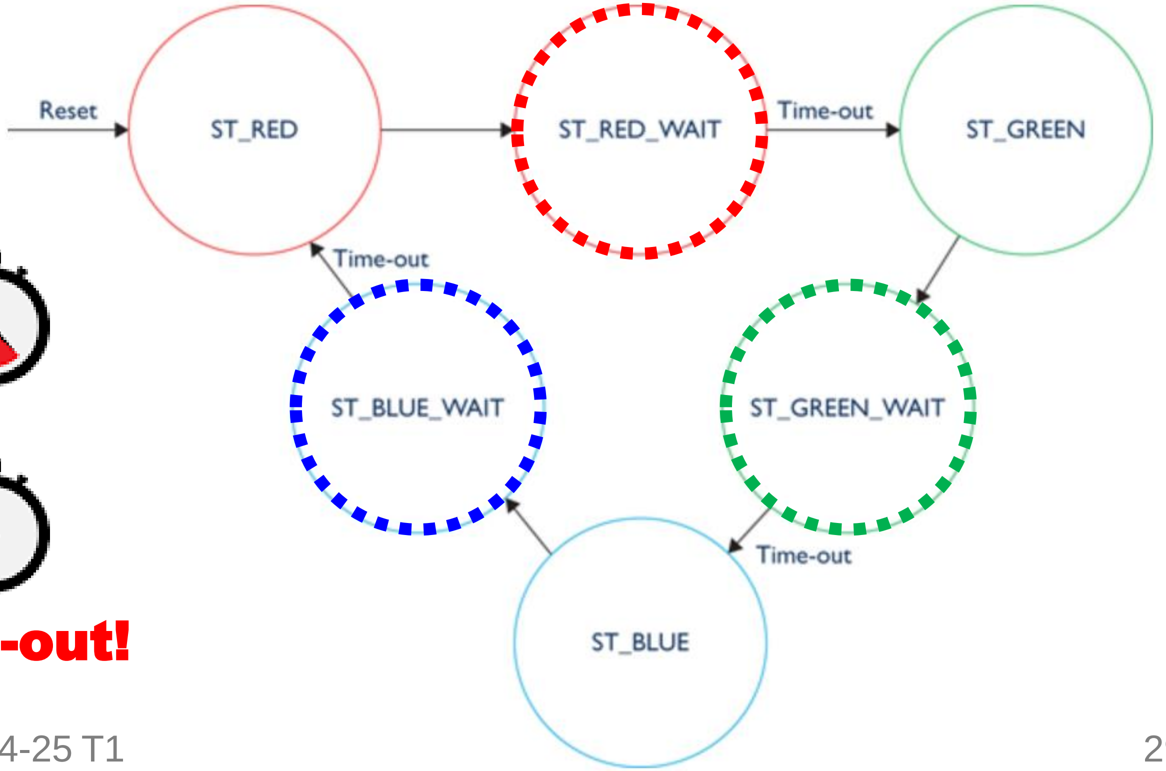


(b) down counting

Let's use timer to replace the "delay"!

- Three **new states** (`ST_R/G/B_WAIT`) are introduced into the FSM to wait for each color cycle to complete.
  - FSM does not advance past the `_WAIT` state until time-out, no matter how many times `Task_RGB` has been called.
- **One-shot timer** generates an IRQ upon time-out.





Timer-triggered Program

- The “delay” function is offloaded from the µP to timer!

```c
void Task_RGB_FSM(void) {
    static enum {ST_RED, ST_GREEN,
                 ST_BLUE, ST_OFF} next_state;
    if (g_flash_LED == 0) {
        switch (next_state) {
            case ST_RED:
                Control_RGB_LEDs(1, 0, 0);
                Delay(g_RGB_delay);
                next_state = ST_GREEN;
                break;
            case ST_GREEN:
                Control_RGB_LEDs(0, 1, 0);
                Delay(g_RGB_delay);
                next_state = ST_BLUE;
                break;
            case ST_BLUE:
                Control_RGB_LEDs(0, 0, 1);
                Delay(g_RGB_delay);
                next_state = ST_RED;
                break;
```

```c
void Task_RGB_FSM_Timer(void) {
    static enum {ST_RED, ST_RED_WAIT,
                 ... , ST_OFF} next_state;
    if (g_flash_LED == 0) {
        switch (next_state) {
            case ST_RED:
                Control_RGB_LEDs(1, 0, 0);
                /* START TIMER */
                next_state = ST_RED_WAIT;
                break;
            case ST_RED_WAIT:
                if (/* TIMEOUT */) {
                    next_state = ST_GREEN;
                }
                break;
```

(omitted: similar to ST_RED)

(omitted: similar to ST_RED)

- Note that an IRQ handler is also needed to handle the “time-out” interrupt triggered by the one-shot timer.

Timer-triggered Program for Tiva

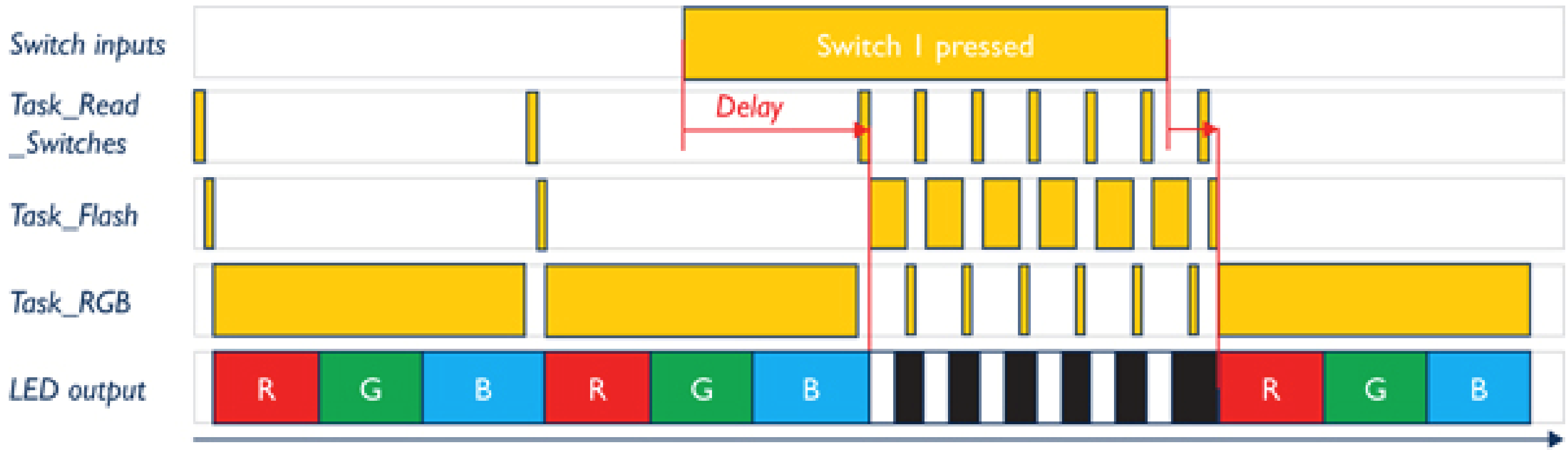
```
#define RGB_DELAY 1500 // unit: ms
uint8_t g_flash_LED = 0; // 0: RGB; 1: flash
bool timer1finish = 1; // if time-out: set to 1
int main(void) { // Flasher()
  Initialization();
  while (1) {
    Read_Switches_Timer();
    RGB_Timer();
    Flash_Timer();
  }
}
void Timer1IntHandler(void) {
  TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
  // MUST clear the interrupt request
  timer1finish = 1; // set time-out flag
}
void Read_Switches_Timer(void) {
  if ( GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4) )
    g_flash_LED = 0;
  else
    g_flash_LED = 1;
}
```

```
void RGB_Timer(void) {
  if ( g_flash_LED == 0 ) {
    static enum { R, R_wait, G, G_wait, B, B_wait }
                  next_state;
    switch (next_state) {
      case R:
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 |
        GPIO_PIN_2 | GPIO_PIN_3, GPIO_PIN_1); // red
        timer1finish = 0; // clear time-out flag
        TimerEnable(TIMER1_BASE, TIMER_A); // start
                                              timer
        next_state = R_wait;
        break;
      case R_wait:
        if ( timer1finish ) // check time-out flag
          next_state = G;
      break;
      ...
    }
  }
}
void Flash_Timer(void) { ... }
```

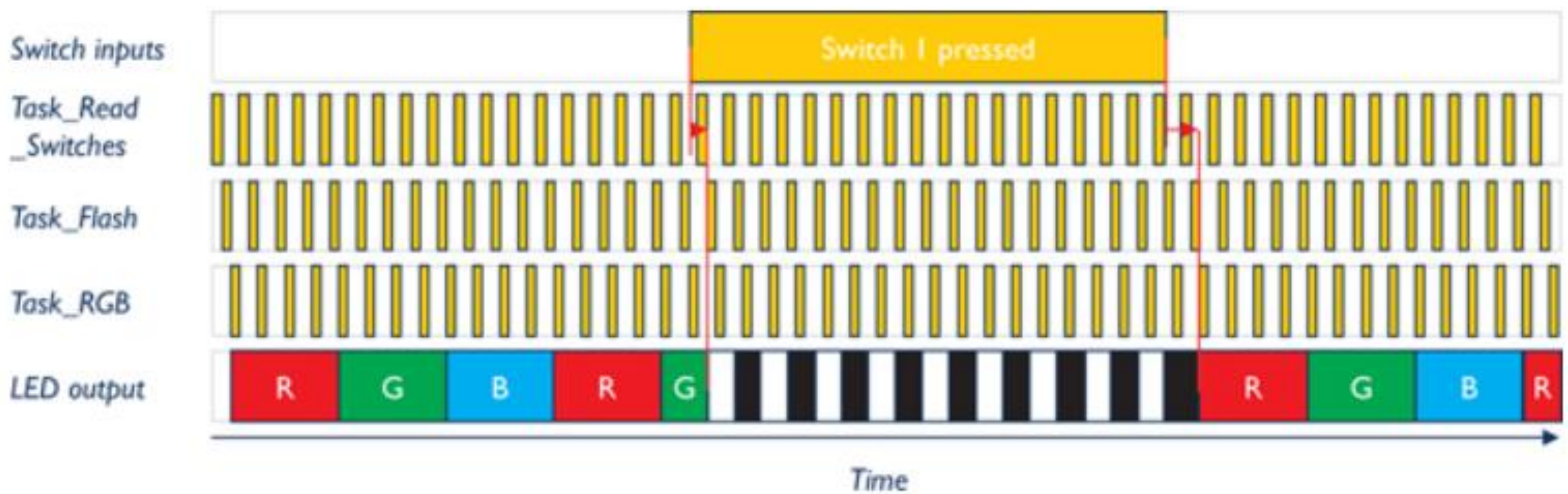
```
void Initialization(void) {
    SysCtlClockSet(SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);
    // configure GPIO
    ...
    // configure Timer 1A (note: there are six timers (namely Timer 0/1/2/3/4/5), each has Timer A/B)
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1); // enable Timer 1 peripheral
    TimerConfigure(TIMER1_BASE, TIMER_CFG_ONE_SHOT); // configure Timer 1 to be count-down one-shot timer
    IntEnable(INT_TIMER1A); // enable the interrupt for Timer 1A
    TimerIntRegister(TIMER1_BASE, TIMER_A, Timer1IntHandler);
    TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT); // enable the interrupt source for Timer 1A
    TimerLoadSet(TIMER1_BASE, TIMER_A, (RGB_DELAY) * SysCtlClockGet()/1000); // set the timer value
    // enable the processor interrupt
    IntMasterEnable();
}
```

How good is it now? (1/2)

- **Restructured Program**:

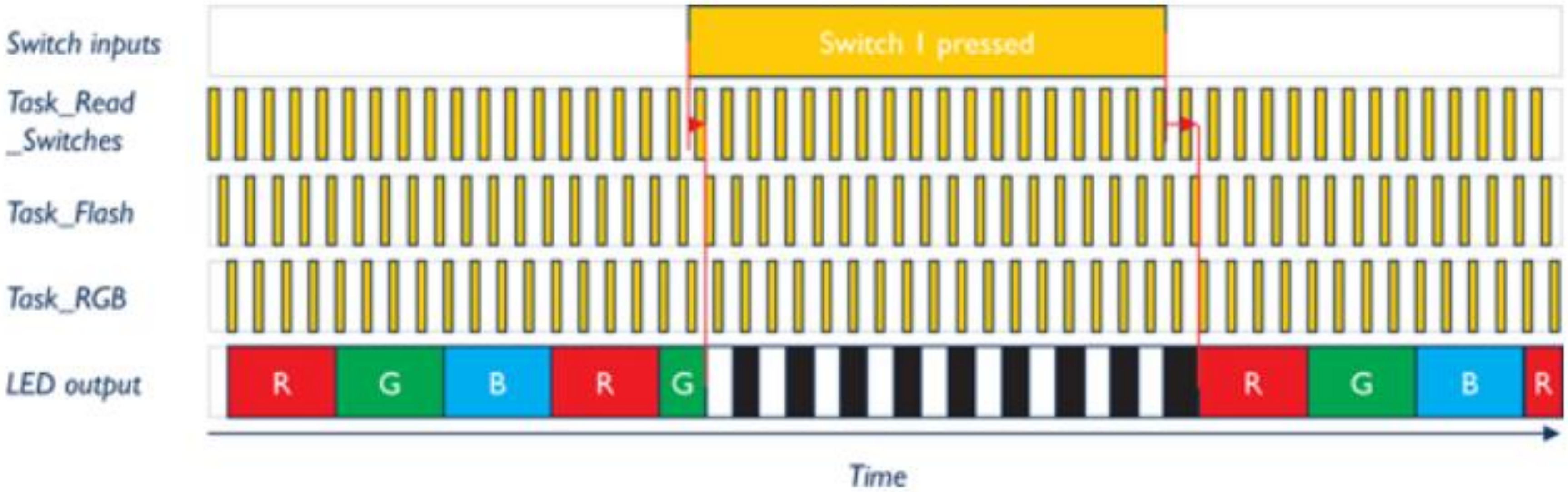


- **Timer-triggered Program**:



How good is it now? (2/2)

- **Timer-triggered Program**:



  - **Responsiveness**: The delays are very small because all calls to the three tasks complete very quickly (by the µP).
  - **CPU Overhead**: The hardware timer tracks the delay (rather than a software function executing on the µP).
    - This releases the µP from busy-waiting, spending time on other (useful) tasks to improve the responsiveness.

Outline

- **Review**
  - Working Example: LED Flasher
  - Restructured Program
  - FSM-structured Program
- **Working Example: LED Flasher (Cont’d)**
  - Polling vs. Event-Triggering
    - Interrupt
  - Switch-triggered Program
  - Timer-triggered Program
    - Timer Peripherals
- **Preview: Lab03**

Preview: Lab03

- "Timer-triggered program" is provided but incomplete.
  - **Job #1**: Complete the implementation for `Task_Flash` by using a different timer (e.g., `Timer 0A`).
- Combining "switch-triggered" and "timer-triggered" might further reduce the CPU overhead.
  - **Job #2**: Turn `Task_Read_Switch` into event-triggering (i.e., reading switches and updating variables in an IRQ handler).

	Modularity	Responsiveness	CPU Overhead
Starter	Poor	Poor	Poor
Restructured	Good	Poor	Poor
FSM-structured	Good	Better	Poor
Switch-triggered	Good	Better	Better
Timer-triggered	Good	Good	Good

Summary

- **Review**
  - Working Example: LED Flasher
  - Restructured Program
  - FSM-structured Program
- **Working Example: LED Flasher (Cont’d)**
  - Polling vs. Event-Triggering
    - Interrupt
  - Switch-triggered Program
  - Timer-triggered Program
    - Timer Peripherals
- **Preview: Lab03**

Important References

- Tiva C Series TM4C123G LaunchPad Evaluation Kit User's Manual

- Tiva™ C Series TM4C123GH6PM Microcontroller Data Sheet datasheet (Rev. E)

- TivaWare™ Peripheral Driver Library for C Series User's Guide (Rev. E)