香港中文大學
The Chinese University of Hong Kong

CENG2400 Embedded System Design

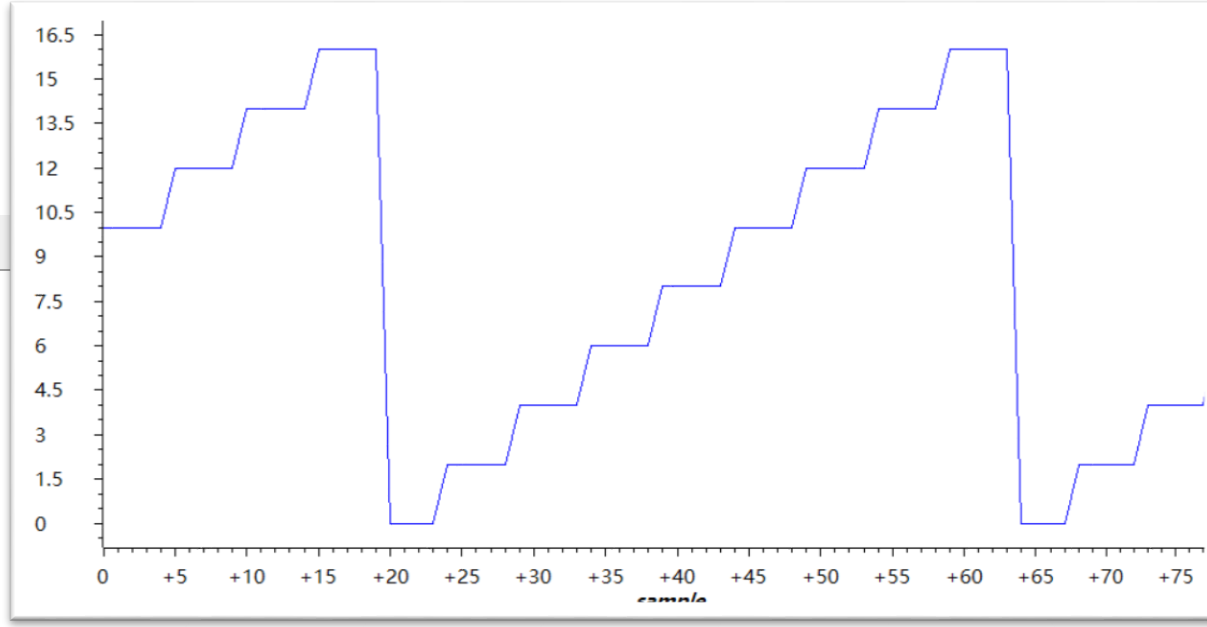
Lecture 03: Concurrency (I)

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!

Review: Lab01 – Code & Results



main.c

```c
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

uint8_t magic_number=0;

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

    while(1)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, magic_number);
        SysCtlDelay(2000000);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
        if(magic_number==16) {magic_number=0;} else {magic_number+=2;}
    }
}
```

Review: Lab01 – API Functions

- **`void SysCtlClockSet(uint32_t ui32Config)`**
  - Sets the clocking of the device.
    - **`ui32Config`** is the required configuration of the device clocking.
- **`void SysCtlDelay(uint32_t ui32Count)`**
  - Provides a small delay. (This function does NOT provide an accurate timing mechanism.)
    - **`ui32Count`** is the number of delay loop iterations to perform.
- **`void SysCtlPeripheralEnable(uint32_t ui32Peripheral)`**
  - Enables a peripheral.
    - **`ui32Peripheral`** is the peripheral to enable.
- **`void GPIOPinTypeGPIOOutput(uint32_t ui32Port, uint8_t ui8Pins)`**
  - Configures pin(s) for use as GPIO outputs.
    - **`ui32Port`** is the base address of the GPIO port.
    - **`ui8Pins`** is the bit-packed representation of the pin(s).
- **`void GPIOPinWrite(uint32_t ui32Port, uint8_t ui8Pins, uint8_t ui8Val)`**
  - Writes a value to the specified pin(s).
    - **`ui32Port`** is the base address of the GPIO port.
    - **`ui8Pins`** is the bit-packed representation of the pin(s).
    - **`ui8Val`** is the value to write to the pin(s).

Review: Lab01 – Explanations

```
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN1|GPIO_PIN2|GPIO_PIN3, n)
```

2.1.4 User Switches and RGB User LED

The Tiva C Series LaunchPad comes with an RGB LED. This LED is used in the preloaded RGB quickstart application and can be configured for use in custom applications.

Two user buttons are included on the board. The user buttons are both used in the preloaded quickstart application to adjust the light spectrum of the RGB LED as well as go into and out of hibernation. The user buttons can be used for other purposes in the user's custom application.

The evaluation board also has a green power LED. Table 2-2 shows how these features are connected to the pins on the microcontroller.

Table 2-2. User Switches and RGB LED Signals

GPIO Pin	Pin Function	USB Device
PF4	GPIO	SW1
PF0	GPIO	SW2
PF1	GPIO	RGB LED (Red)
PF2	GPIO	RGB LED (Blue)
PF3	GPIO	RGD LED (Green)

					G	B	R		
n	PF7	PF6	PF5	PF4	PF3	PF2	PF1	PF0	LED
0	0	0	0	0	0	0	0	0	Off
2	0	0	0	0	0	0	1	0	Red
4	0	0	0	0	0	1	0	0	Blue
6	0	0	0	0	0	1	1	0	Purple (Red + Blue)

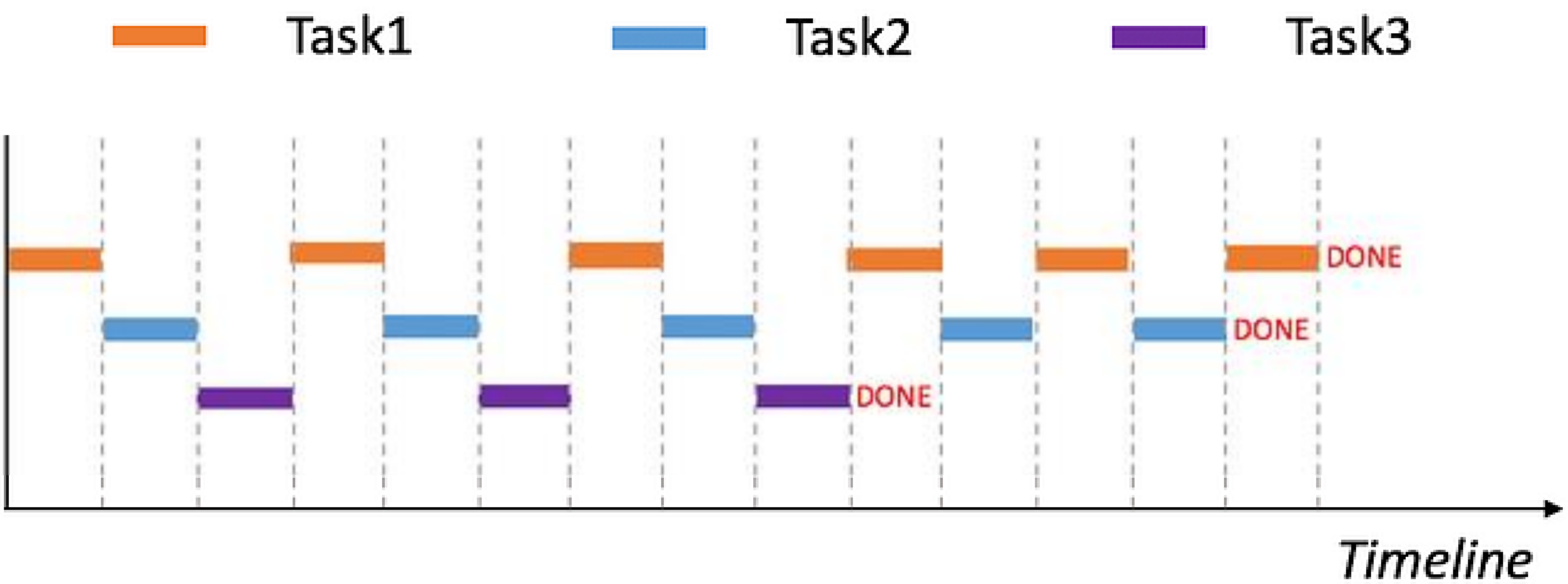
Outline

- **Overview**
  - What is concurrency?
  - Concurrency vs. Parallelism
- **Working Example: LED Flasher**
  - Starter Program
  - Restructured Program
  - FSM-structured Program
- **Preview: Lab02 and Lec04**

Concurrency?

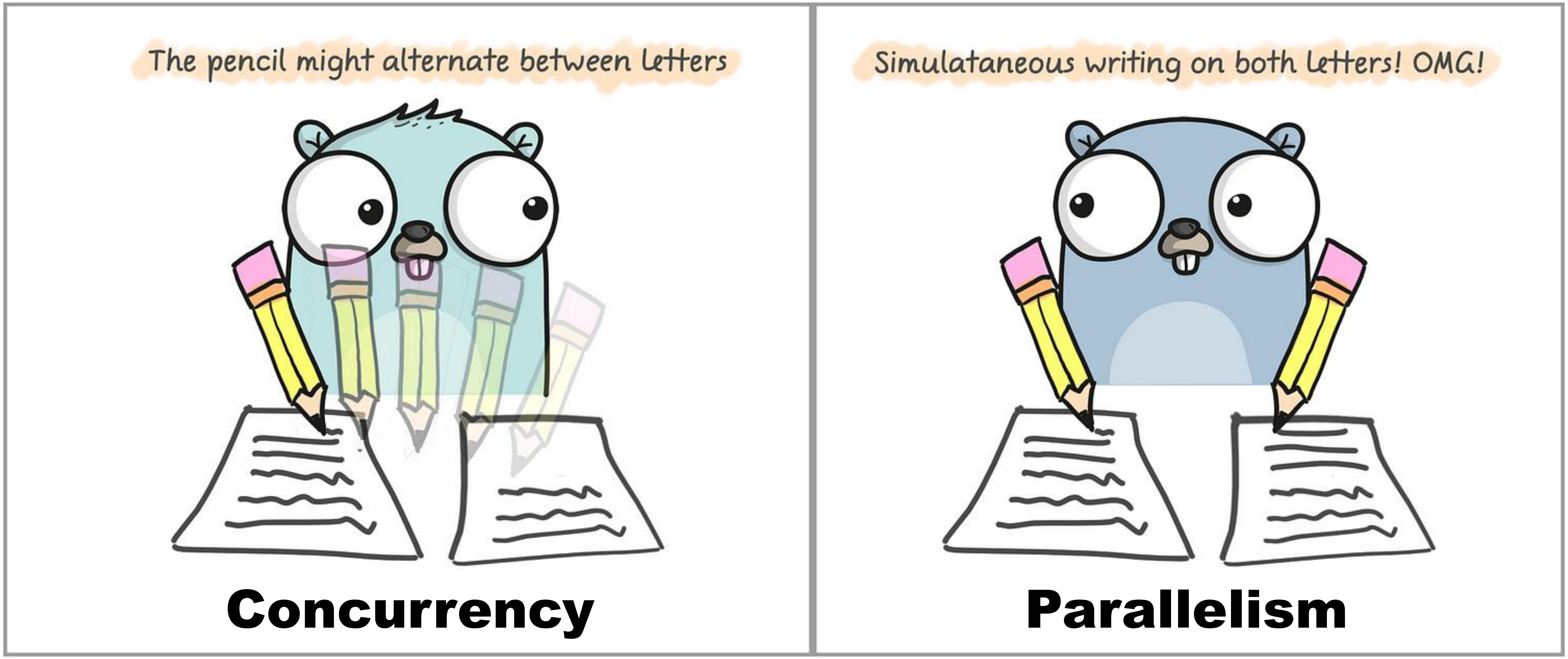
- MCU performs **multiple tasks** (e.g., light on/off LED and read SW) apparently simultaneously, providing the *illusion* of concurrent execution.

- We must share the time of µP time among tasks.



Concurrency vs. Parallelism

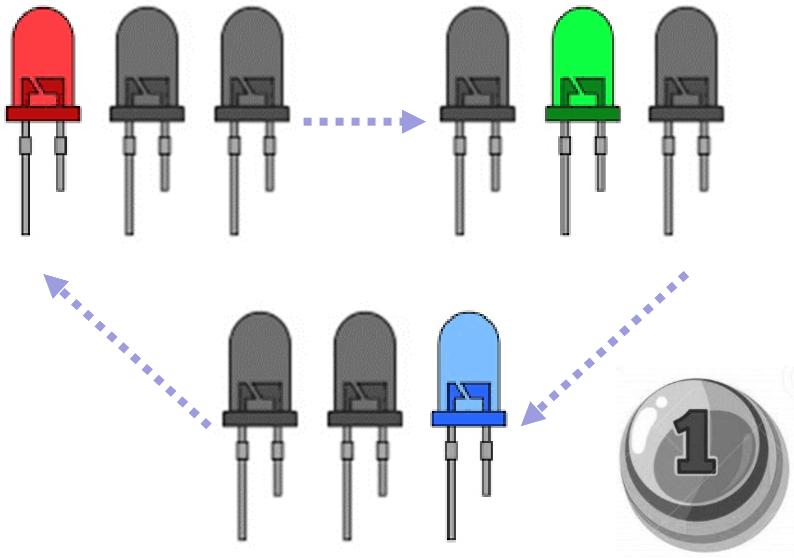
- **Concurrency**: Doing multiple tasks "alternately";
  - Requiring the sharing of time and resources.
- **Parallelism**: Doing multiple tasks "at the same time".
  - Requiring multiple copies of resources.

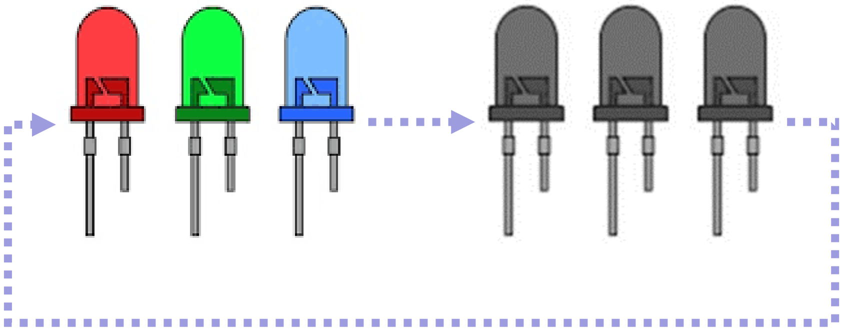


Outline

- **Overview**
  - What is concurrency?
  - Concurrency vs. Parallelism
- **Working Example: LED Flasher**
  - Starter Program
  - Restructured Program
  - FSM-structured Program
- **Preview: Lab02 and Lec04**

Working Example: 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.*

Factors to Consider

- We will start with a starter program and explore how to improve it from the following aspects:

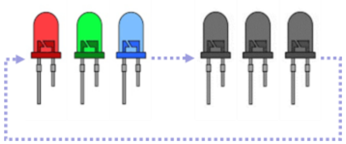
> **Modularity:** Measure of how program is structured to group related portions and separate independent portions

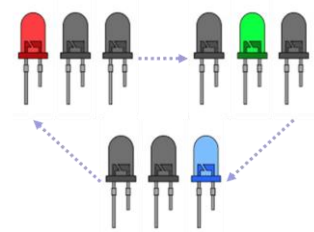
> **Responsiveness:** Measure of how quickly a system responds to an input event

> **CPU Overhead:** Portion of time CPU spends executing code which does not perform useful work for the application

Starter Program: Everything in a Loop

```c
#define W_DELAY_SLOW 400
#define W_DELAY_FAST 200
#define RGB_DELAY_SLOW 4000
#define RGB_DELAY_FAST 1000
void Flasher(void) {
        uint32_t w_delay = W_DELAY_SLOW;
        uint32_t RGB_delay = RGB_DELAY_SLOW;
        while (1) {
                if (SWITCH_PRESSED(SW1_POS)) { // flash white
                        Control_RGB_LEDs(1, 1, 1);
                        Delay(w_delay);
                        Control_RGB_LEDs(0, 0, 0);
                        Delay(w_delay);
                } else { // sequence R, G, B
                        Control_RGB_LEDs(1, 0, 0);
                        Delay(RGB_delay);
                        Control_RGB_LEDs(0, 1, 0);
                        Delay(RGB_delay);
                        Control_RGB_LEDs(0, 0, 1);
                        Delay(RGB_delay);
                }
                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;
                }
        }
}
```



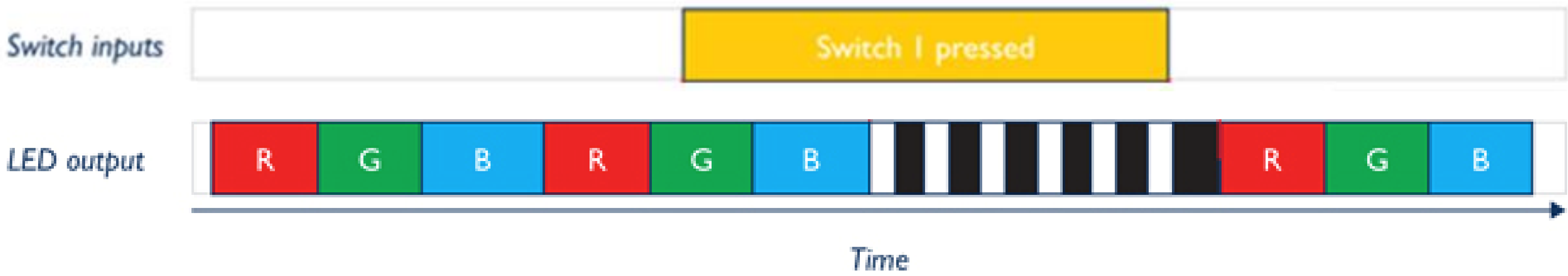


How is it "structured"?

- The starter program mixes together different activities in a single function.

Spaghetti Code: Poorly-structured code which entangles unrelated features, complicating development & maintenance.

How “responsive” is it?



- **Issue 1**: If we press the switch when the green LED is lit, it does not start flashing immediately until the blue turns off; similarly, releasing the switch also results in a delay.
  - Why? The code only polls the switch between full RGB/flash cycles.

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

- **Issue 2**: If we press the switch briefly during the RGB cycle and release it before the end, it will not detect it.
  - Input events shorter than the RGB/flash cycles may be lost.

How efficient is it w.r.t. "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.

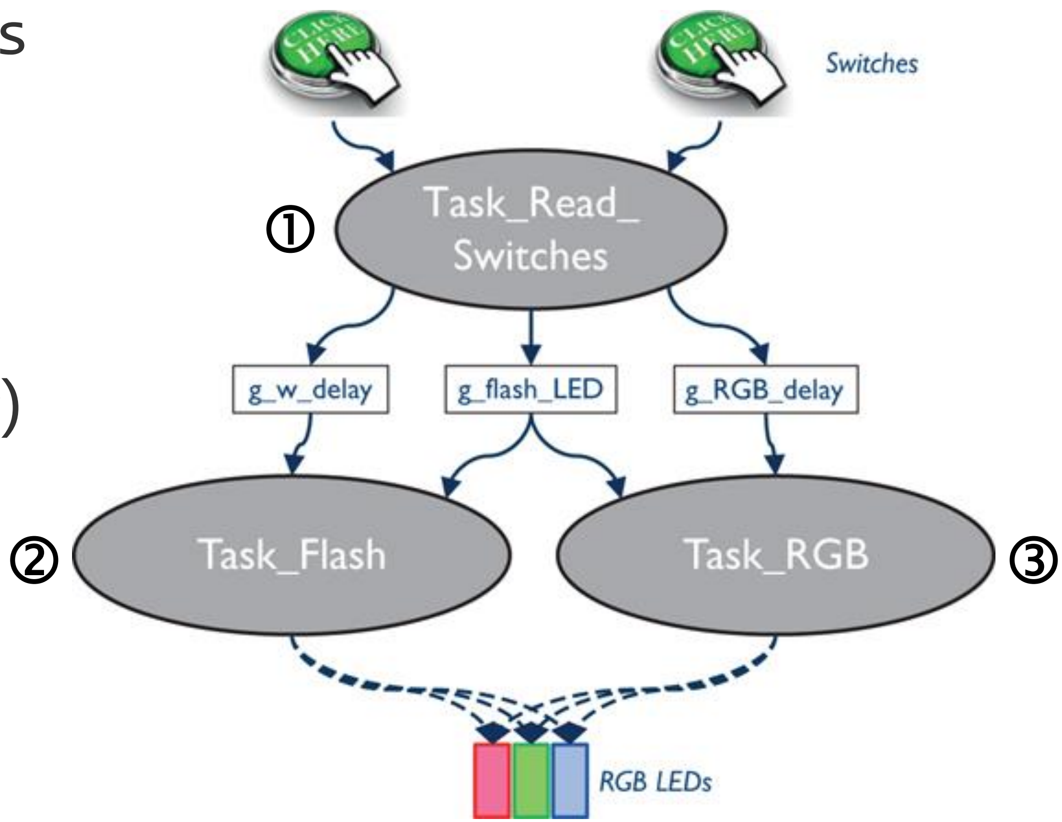
Outline

- **Overview**
  - What is concurrency?
  - Concurrency vs. Parallelism
- **Working Example: LED Flasher**
  - Starter Program
  - Restructured Program
  - FSM-structured Program
- **Preview: Lab02 and Lec04**

Let’s restructure it!

- The code can be restructured into three tasks:
  ① `Task_Read_Switches`
  ② `Task_Flash`
  ③ `Task_RGB`

Note: The tasks (*in ovals*) communicate with each other via the global variables (*rectangles*).



Task: A subroutine that performs a specific activity (or a closely related set of activities).

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
    }
}
```

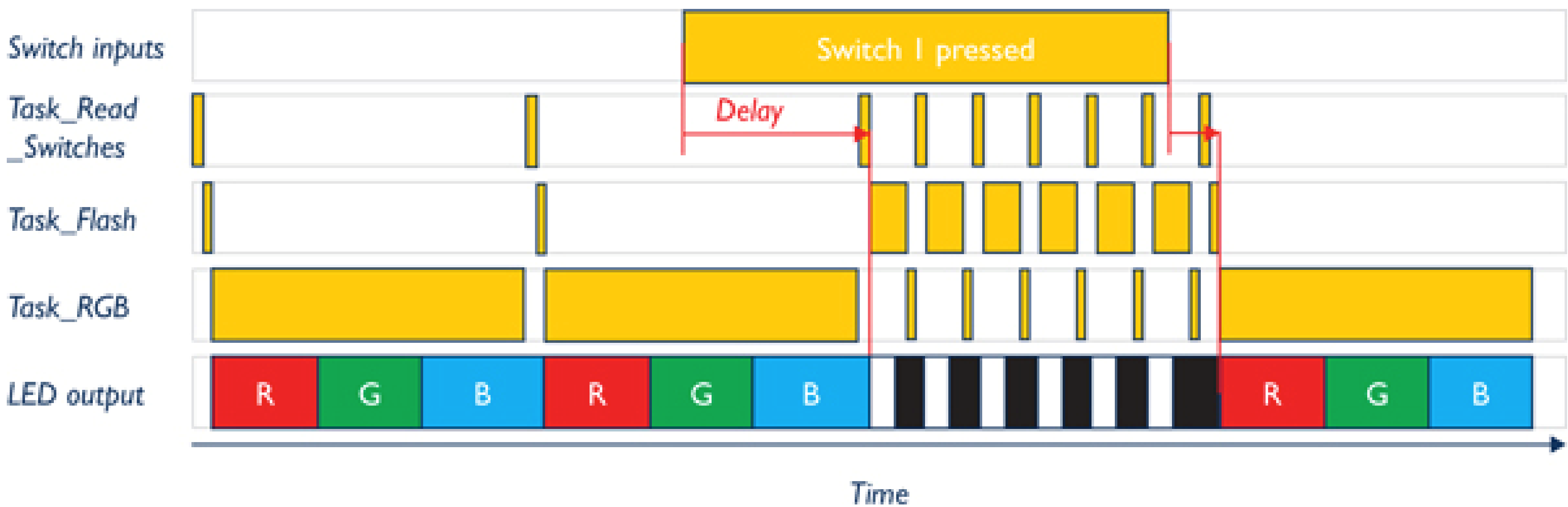
```
#define W_DELAY_SLOW 400
#define W_DELAY_FAST 200
#define RGB_DELAY_SLOW 4000
#define RGB_DELAY_FAST 1000

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);
    }
}
```

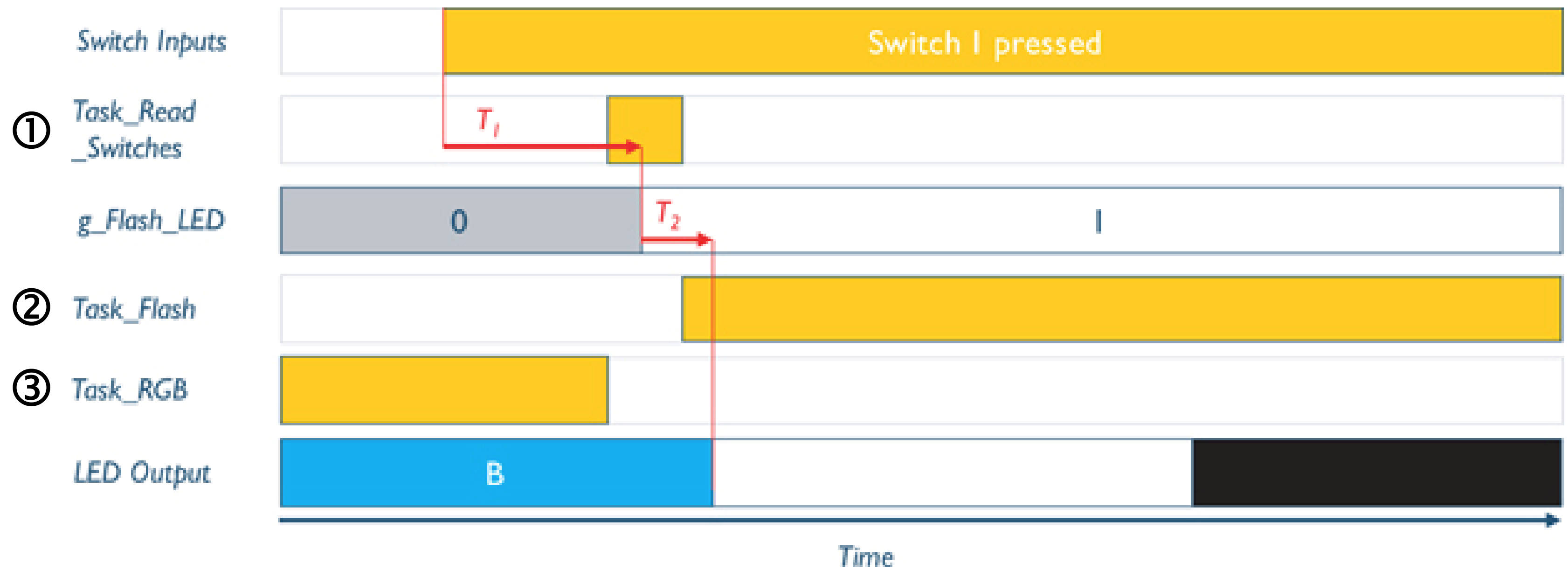
How about it now?

- The program is now structured much better!
  - It isolates the three tasks from each other.
- The responsiveness is no better than the starter program.
  - In fact, it is slightly worse because of the overhead of the scheduler calling the task functions.



- The program still relies on the “delay” function, wasting the time of µP quite a bit.

Why NOT responsive still?



- If we look closer, the delay has two parts:
  - $T1$: Delay between when the switch is pressed (released) and when the variable g_Flash_LED is updated (in Task ①);
  - $T2$: Delay between when the variable g_Flash_LED is updated (in Task ①) and the LED starts flashing (in Task ②).

Class Exercise 3.1

- Consider the "restructured program." Assume there is no time taken to switch between tasks, and that the tasks have the following execution times:

Task or handler	Execution time when in flash mode	Execution time when in RGB mode
Task_Read_Switches	1 ms	1 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.

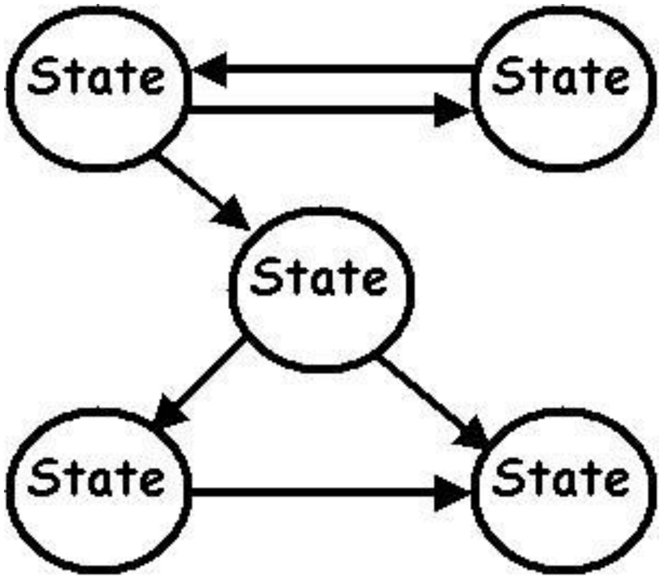
② Describe the sequence that leads to maximum delay between **releasing** the switch and seeing the **LED sequence RGB colors**. Calculate the value of that delay.

Outline

- **Overview**
  – What is concurrency?
  – Concurrency vs. Parallelism
- **Working Example: LED Flasher**
  – Starter Program
  – Restructured Program
  – FSM-structured Program
- **Preview: Lab02 and Lec04**

Using FSM to improve responsiveness

- We can use a structure called the Finite State Machine (FSM) to improve the responsiveness.
  - How? Splitting up the tasks to make them return before it has finished all of its work – This gives more frequent opportunities to run other tasks!



Finite State Machine (FSM): A type of state machine with all states and transitions defined.

State Machine: State-based system model with rules for transitions between states.

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);
        }
}
```

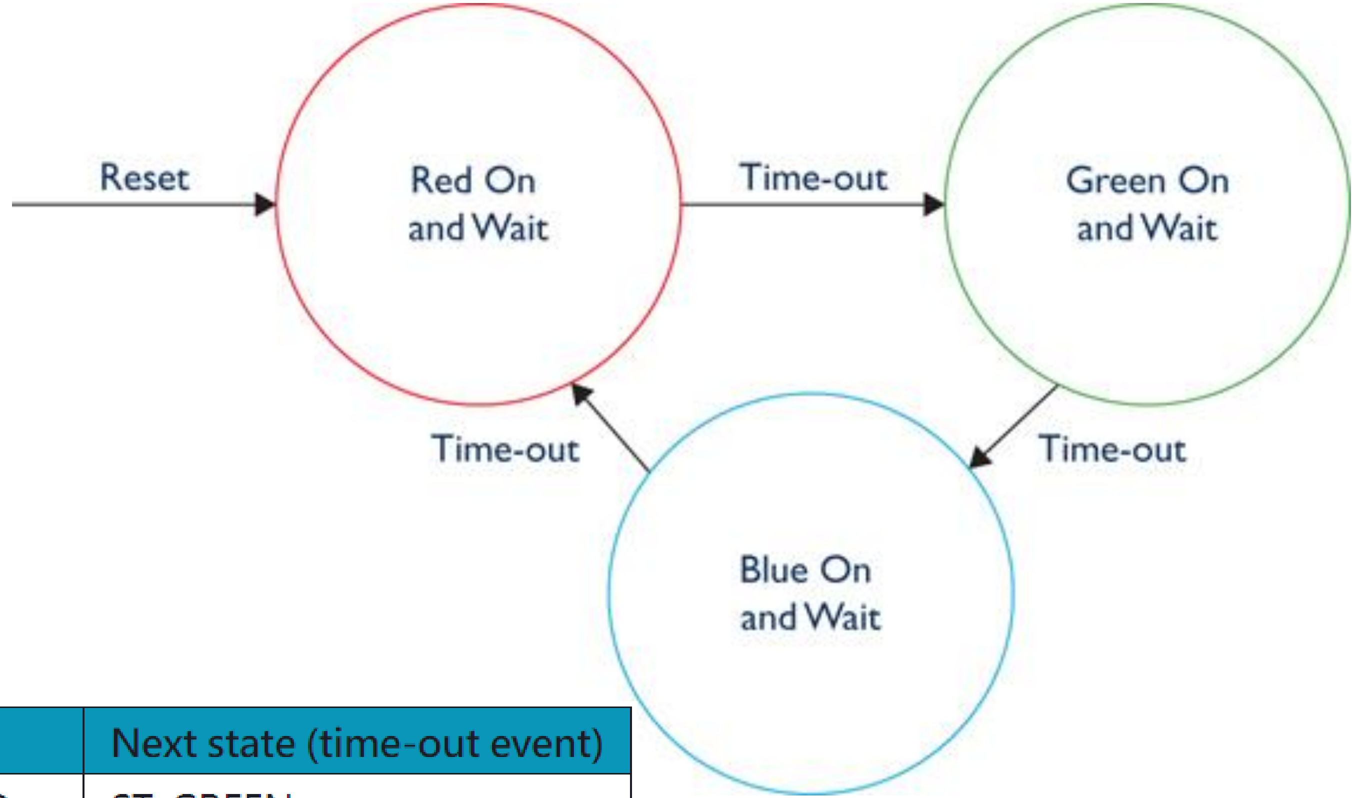
FSM-structured Task_RGB (1/2)

```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;
            default:
                next_state = ST_RED;
                break;
        }
    }
}
```

```
/* the (static) state variable
is to track the next state to
execute, and it is declared as
an enumerated type to make the
code easier to read */

/* the switch statement selects
which code to execute based on
the value of next_state */

/* each case statement contains
the code for one state and may
update the state variable for
future calls to Task_RGB_FSM */
```

FSM-structured Task_RGB (2/2)

- The below state transition diagram and table describe how the state machine operates:



State	Action	Next state (time-out event)
ST_RED	Light red LED	ST_GREEN
ST_GREEN	Light green LED	ST_BLUE
ST_BLUE	Light blue LED	ST_RED

FSM-structured Task_Flash

- `Task_Flash_FSM` can be similarly structured (as FSM):

```
void Task_Flash_FSM(void) {
    static enum {ST_WHITE, ST_BLACK} next_state = ST_WHITE;

    if (g_flash_LED == 1) { // Only run task when in flash mode
        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;
        }
    }
}
```

```
/* the (static) state
variable is to track
the next state to
execute */

/* the switch statement
selects which code to
execute based on the
value of next_state */

/* each case statement
contains the code for
one state */
```

Put Together: 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;
        }
}
```

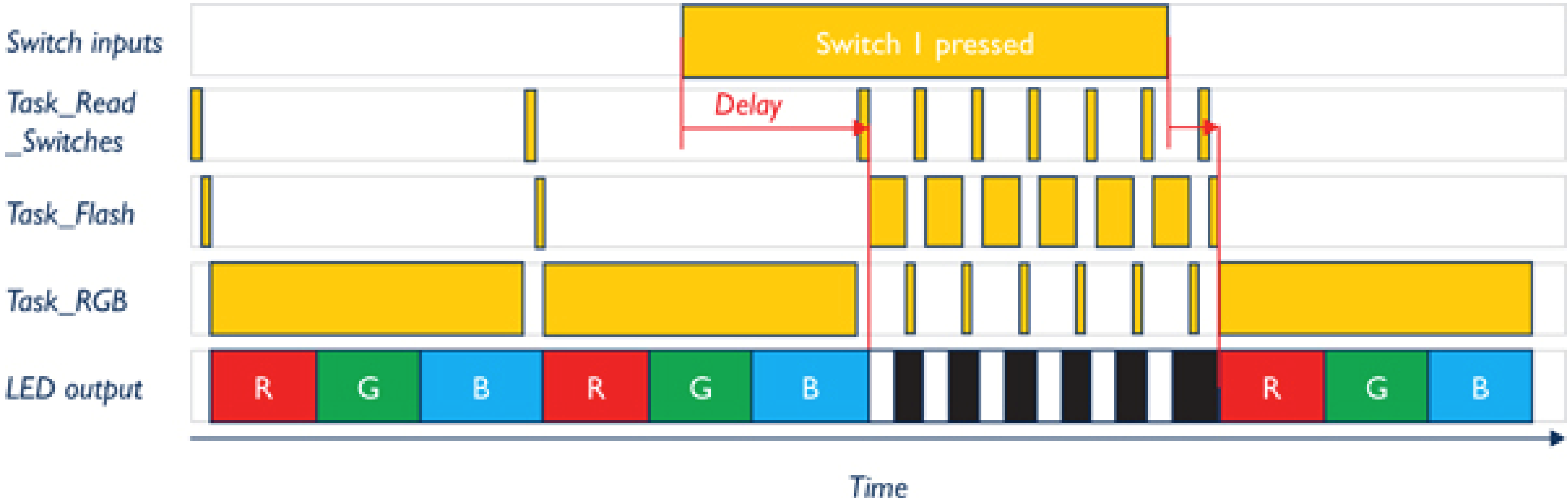
①

```
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;
        }
    }
}
```

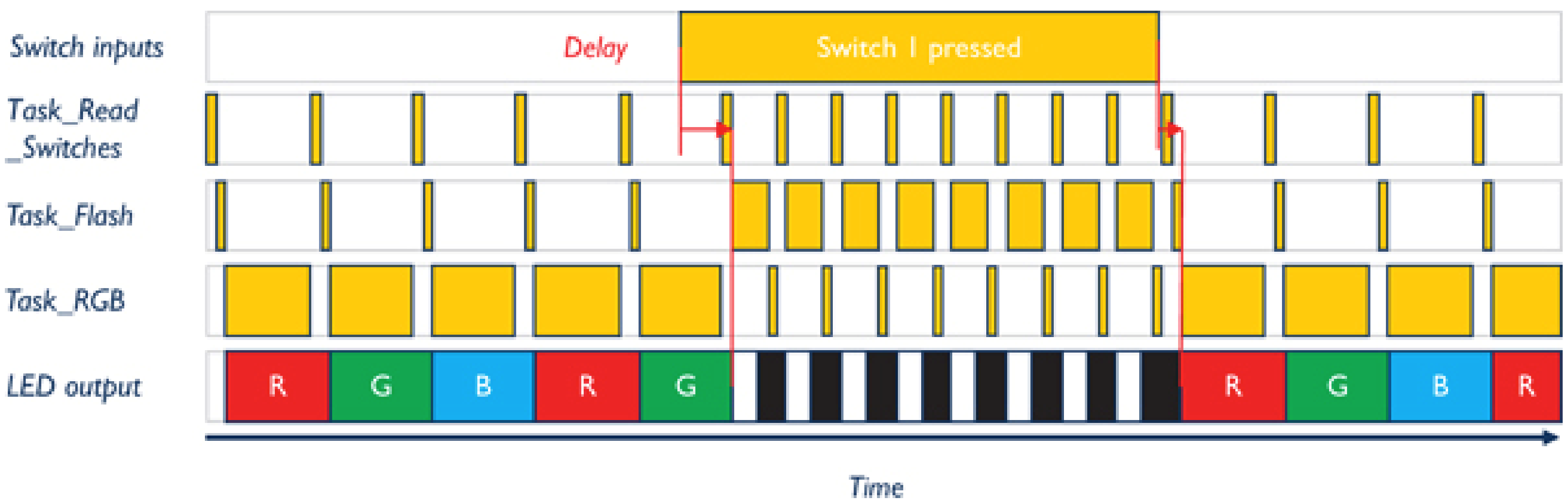
③

How "responsive" is it now? (1/2)

- **Restructured Program:**

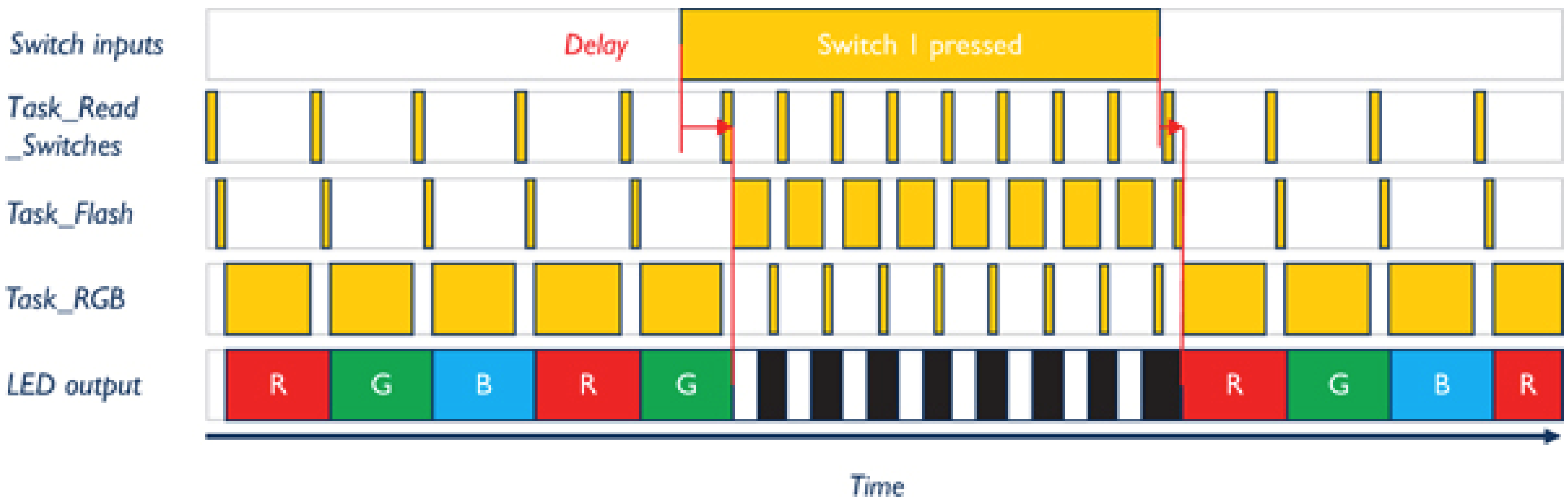


- **FSM-structured Program:**



How “responsive” is it now? (2/2)

- **FSM-structured Program**:



  – The responsiveness is much better!
    - The flashing starts after the current stage of the sequence (green here), rather than the last stage (blue).

  – **Not responsive enough?** We could split up the delay (task) into more states to further reduce the response time.

Class Exercise 3.2

- Consider the “FSM-structured program.” Assume there is no time taken to switch between tasks, and that the tasks have the following execution times:

Task	Execution time when in flash mode	Execution time when in RGB mode
Task_Read_Switches	1 ms	1 ms
Task_Flash	34 ms	1 ms
Task_RGB	1 ms	334 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.

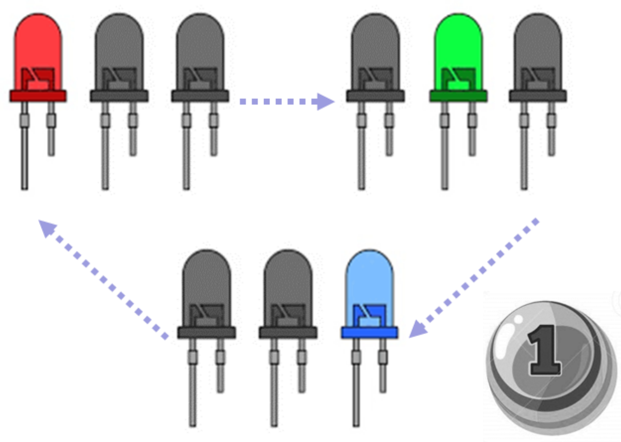
Outline

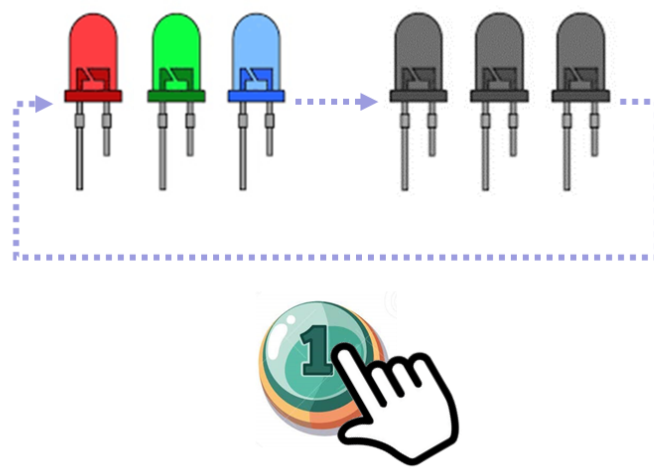
- **Overview**
  - What is concurrency?
  - Concurrency vs. Parallelism
- **Working Example: LED Flasher**
  - Starter Program
  - Restructured Program
  - FSM-structured Program
- **Preview: Lab02 and Lec04**

Preview: Lab02

- The codes in the slides cannot directly run on our Tiva™ LaunchPad.

→ *Let's make them work in* ***Lab02****!*





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.*~~

(IGNORE SWITCH 2)

Preview: Lec04 Concurrency (II)

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

- The “FSM-structured program” is well-structured and exhibits much better responsiveness.
- However, it still relies on the “delay” function, making the CPU busy-waiting.

→ *We will study how to use “interrupts” and “hardware” (e.g., Timer) to save CPU time in **Lec04**!*

Summary

- **Overview**
  - What is concurrency?
  - Concurrency vs. Parallelism
- **Working Example: LED Flasher**
  - Starter Program
  - Restructured Program
  - FSM-structured Program
- **Preview: Lab02 and Lec04**

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)