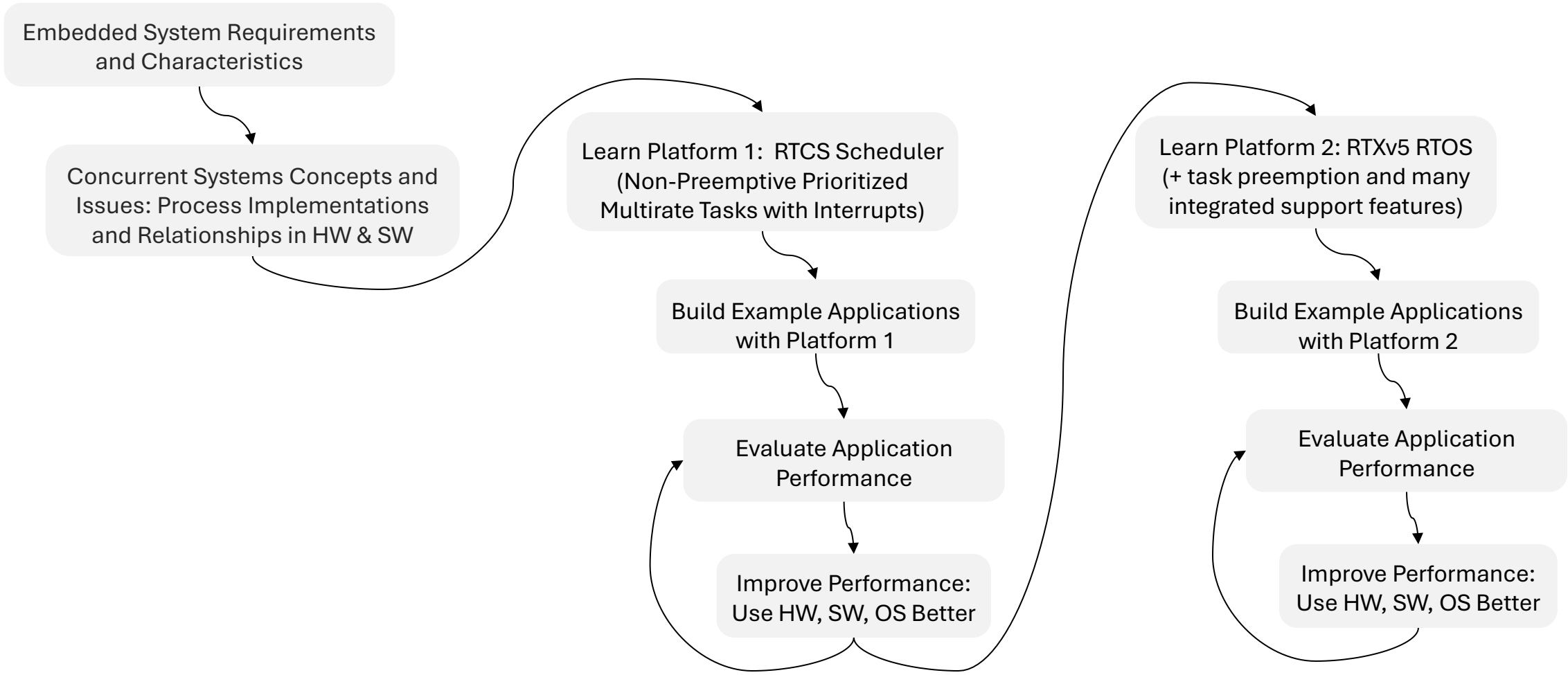
APPLICATION DESIGN BASICS USING RUN-TO-COMPLETION SCHEDULER WITH INTERRUPTS

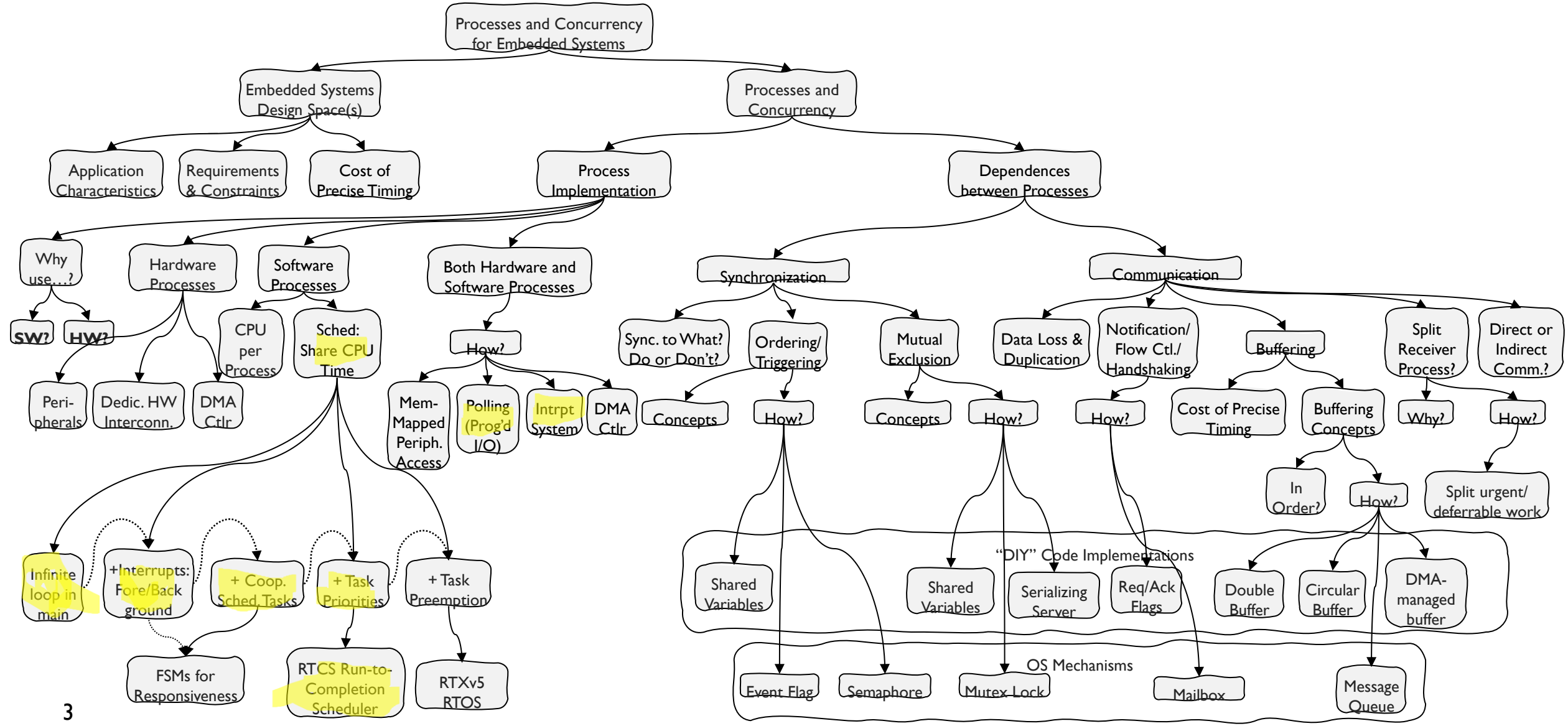
V1

9/11/2025

Where are we in the class?



Extended Topic Map: Class 08



Application Design Overview

- Identify system's **inputs, outputs** and **processes**, and their key connections
- Identify key **hardware, software stages** in each process
  - Initially consider only core peripheral features (later will examine enhancements)
- Analyze each process to find key **synchronization requirements** (e.g. timing, events)
  - What triggers/releases/allows process (e.g. thread's work code) to start execution: ***event or time?***
    - Do A once for every time E1 happens
    - Do B every 100 ms
  - Is there any ***internal*** synchronization (*wait until*)?
    - Do X, wait at least 3.2 ms but no more than 3.8 ms, do Y, Wait for event E, do Z.
- Analyze key **process interactions**
  - Identify and describe ***synchronization*** and ***communication*** **between** processes
    - Triggering, shared data and resources (one-way data flows, other data flows)
  - Identify implicit ***sync.*** required within the ***communication***.
    - Notification of new data? Buffer old data? How much? how to manage buffer? How to handle overrun condition?
- Choose mechanisms to support and implement these requirements and interactions
  - Program structure
  - Interrupt/Scheduler/OS support
  - Algorithms to implement in your code
- Continue with design and implementation
  - Functionality first, then performance
  - Iterate design to improve performance

Example Applications: Key Requirements and Challenges

Application	Providing Functionality					Meeting Performance Requirements			
	Digital Interfacing	Analog Interfacing	Sync and Do: Triggering	Sync and Don't: Sharing & Races	Inter-Process Comm.	Timing Stability	Responsiveness	Reducing Software Overhead	Tolerating Timing Mismatches
Blinky Control Panel	In, Out, PWM								
Quadrature Decoder	In								
Waveform Generator		Out							
Oscilloscope		In							
Serial Comms.	In, Out								
I2C Comms.	In, Out								
LCD Controller	Out								
Touchscreen		In							
SMPS Controller	Out	In							
µSD via SPI Comms.	In, Out								

APPLICATION DESIGN USING RTCS

Checklist for Using RTC Scheduler

1. Add RTCS folder to project directory
2. Add RTCS folder to include search path
3. Link in scheduler tick routine
    1. Add call to tick_timer_intr() to a periodic interrupt.
    2. Add #include “rtcs.h” to that source file too.
4. Create code for your tasks in tasks.[ch]
5. Modify main code to
    1. Add your tasks to scheduler task table
    2. Start up the scheduler

RTCS_DEMO PROGRAM: INDEPENDENT R, G, B LED FLASHING

Example Application (RTCS_Demo)

- Toggle red LED every 500 ms
- Toggle green LED every 490 ms
- Toggle blue LED every 480 ms

$\geq$ GCD = 10ms

- How would you code this without a scheduler?
- With a periodic scheduler, consider greatest common divisor (GCD) of periods

Demo Code for a Task

```
void Task_R(void) {
  static uint8_t LED_On=0;

  PTB->PSOR = MASK(DEBUG_RED_POS);

  if (LED_On)
        PTB->PCOR = MASK(RED_LED_POS);
  else
        PTB->PSOR = MASK(RED_LED_POS);

  LED_On = 1 - LED_On;

  PTB->PCOR = MASK(DEBUG_RED_POS);
}
```

Set (1) a debug output bit to see on scope/logic analyzer when task starts running

Clear (0) the debug output bit to see when task stops running

Demo Scheduler Start-Up

```
int main (void) {
  Init_Debug_Signals();
  Init_RGB_LEDs();

  RTCS_Init(100); // 100 Hz timer ticks
  RTCS_Add_Task(Task_R, 0, 50);
  RTCS_Add_Task(Task_G, 1, 49);
  RTCS_Add_Task(Task_B, 2, 48);

  RTCS_Run_Scheduler();  // This call never returns
}
```

RGB/FLASHER PROGRAM WITH RTCS

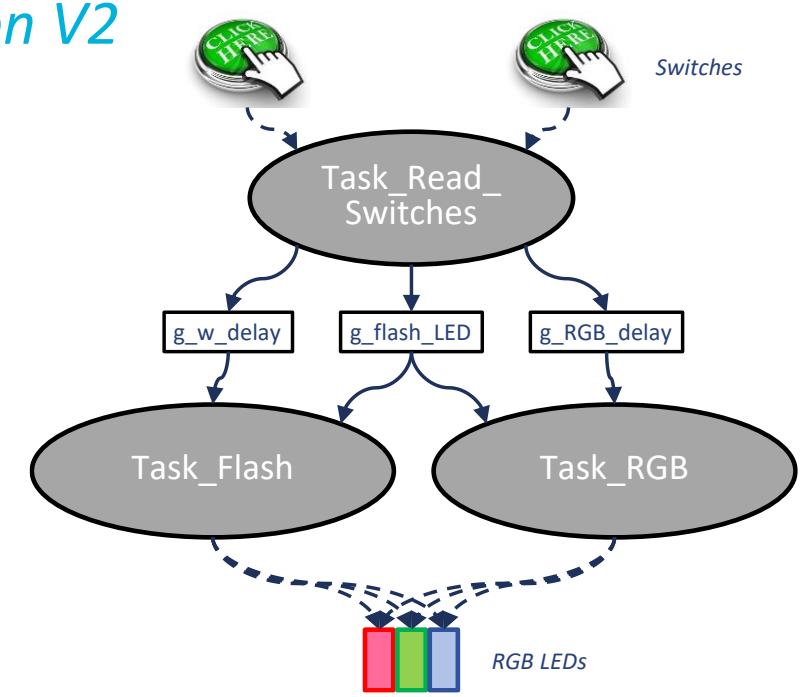
RGB/Flasher Upgrade to RTCS

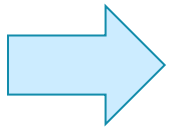
- **ESF** textbook Chapter 3 example
  - LED behavior: flash (White/Off) or sequence (R/G/B)
  - LED timing: slow or fast
- How to port the program to use the RTCS?
- Version 1
  - Simple port, start with switch polling version (V2)
  - Uses a few RTCS features
- Version 2
  - Better port, start with switch interrupt version (V3)
  - Takes advantage of more RTCS features
- Code for both is in course repository in RTCS folder

RGB/FLASHER PROGRAM WITH RTCS

RTCS_RGB_Flasher_1

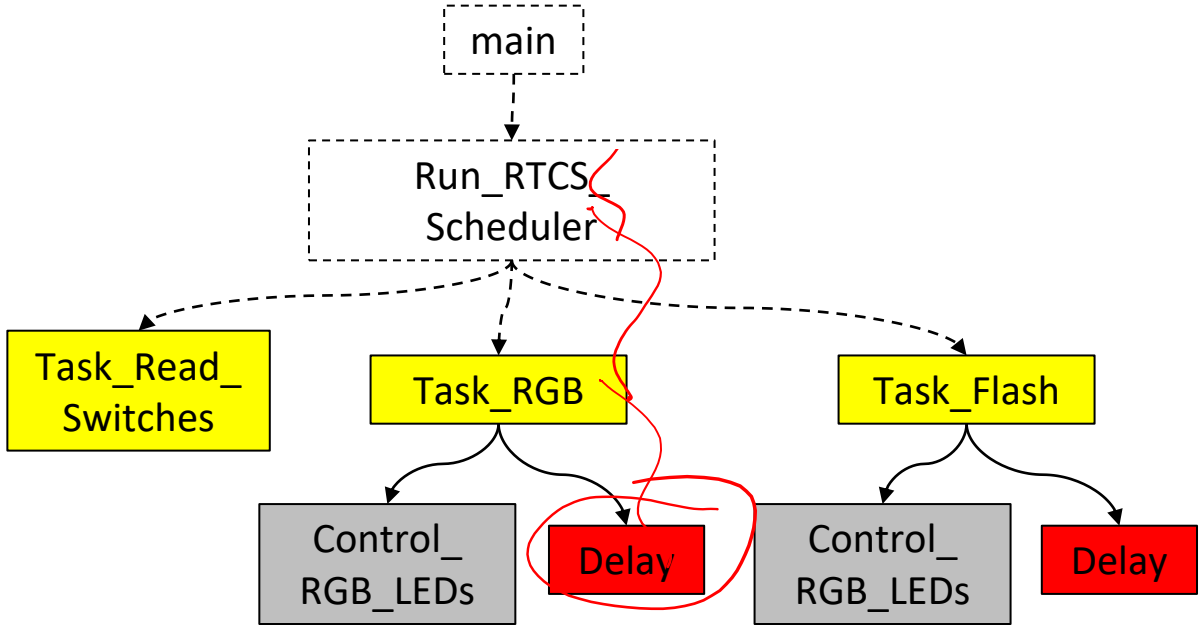
Based on V2

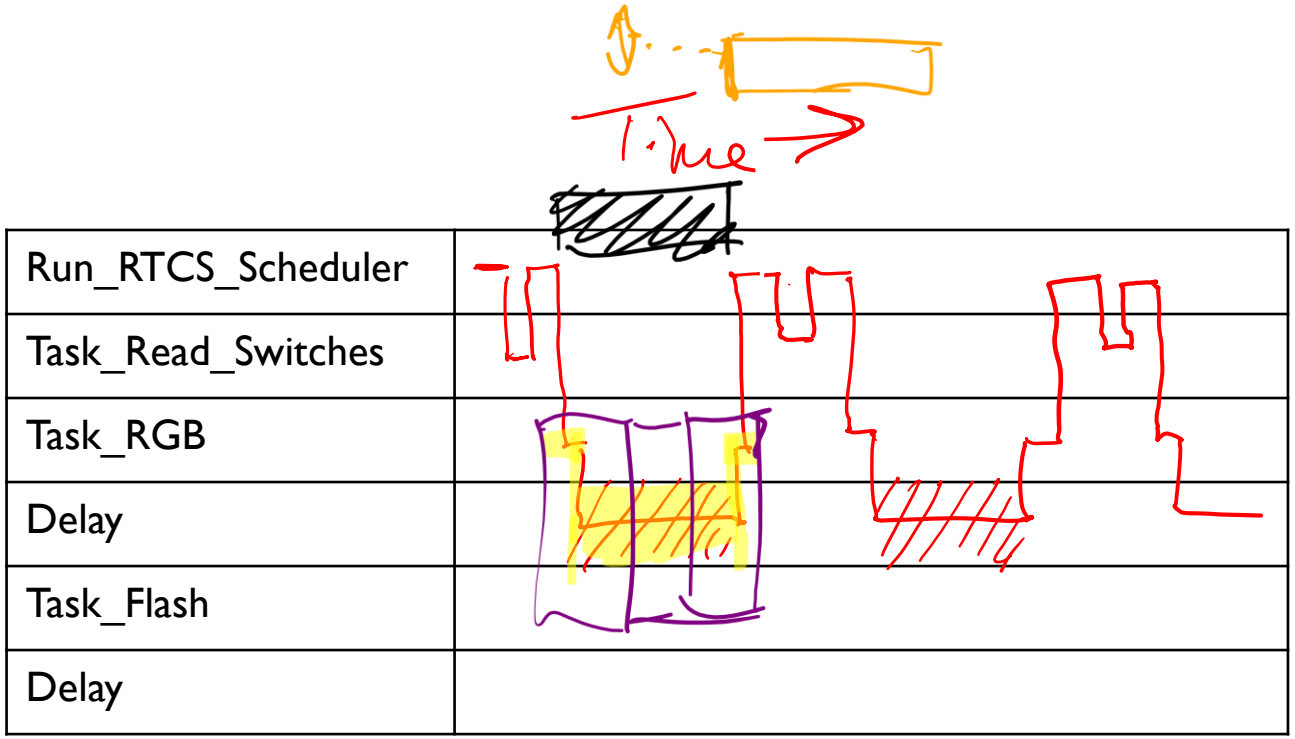




- Switches polled by periodic Task_Read_Switches
  - ***Enables*** and ***disables*** tasks
  - Modifies global delay variables
- LEDs
  - Task_RGB and Task_Flash control LED timing using a busy-wait Delay function

Where is the Idle Time?

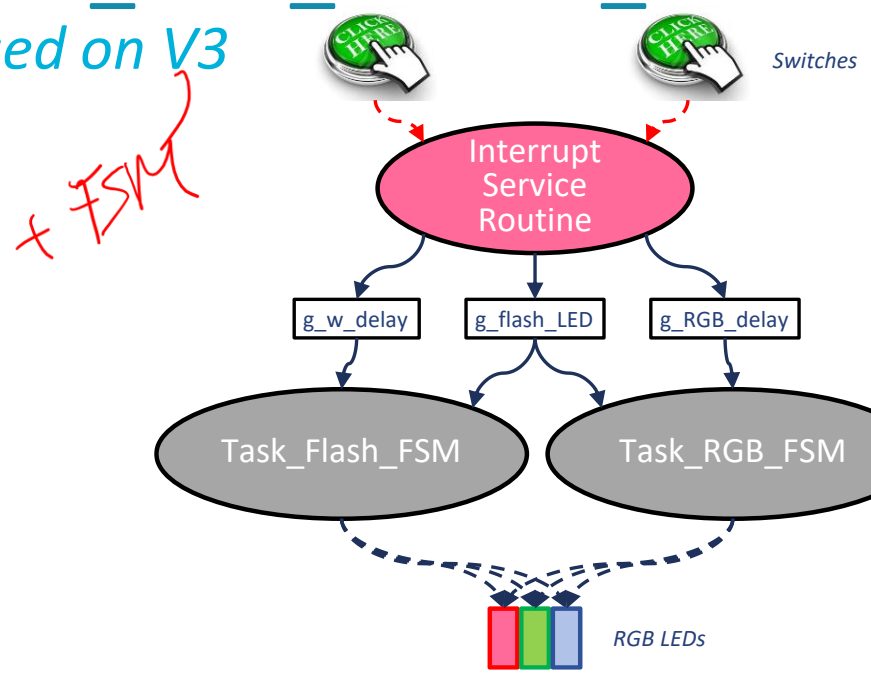


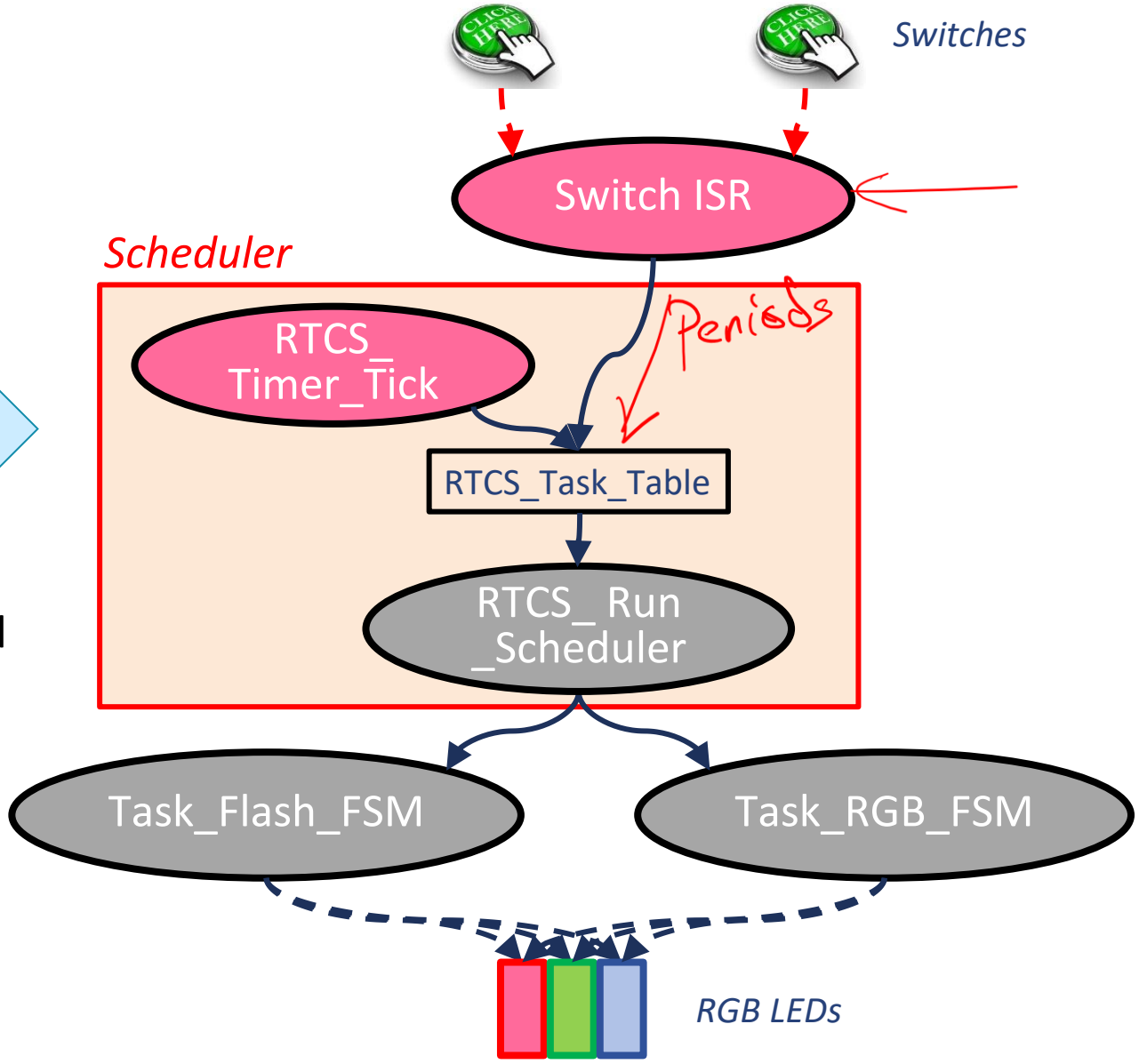


- Analysis
  - Examine call graph
  - Consider code execution timeline
- Idle time is buried within task call to in busy-wait Delay function
  - Not available to scheduler

RTCS_RGB_Flasher_2

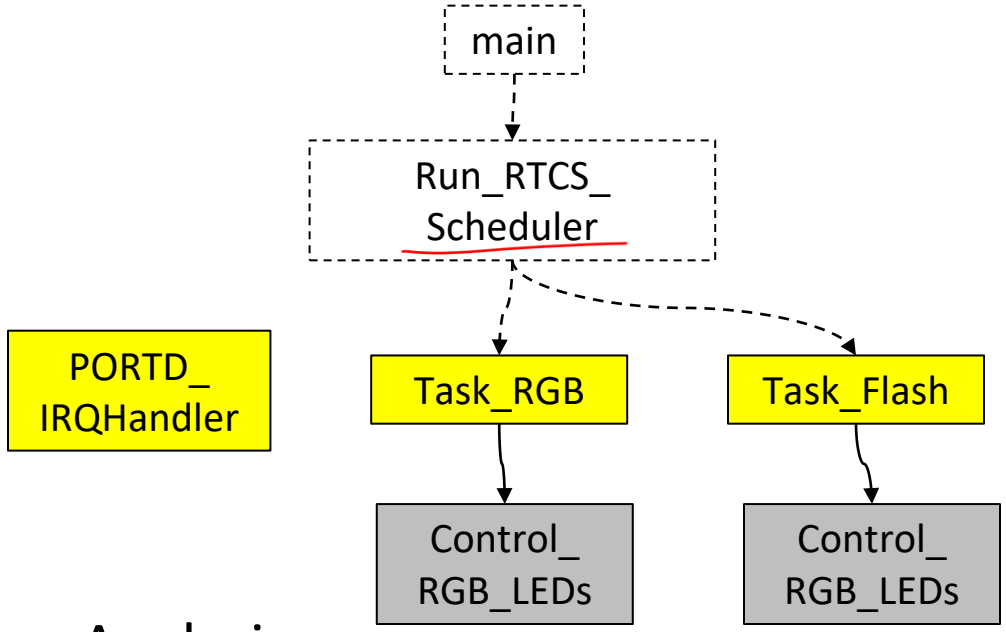
Based on V3

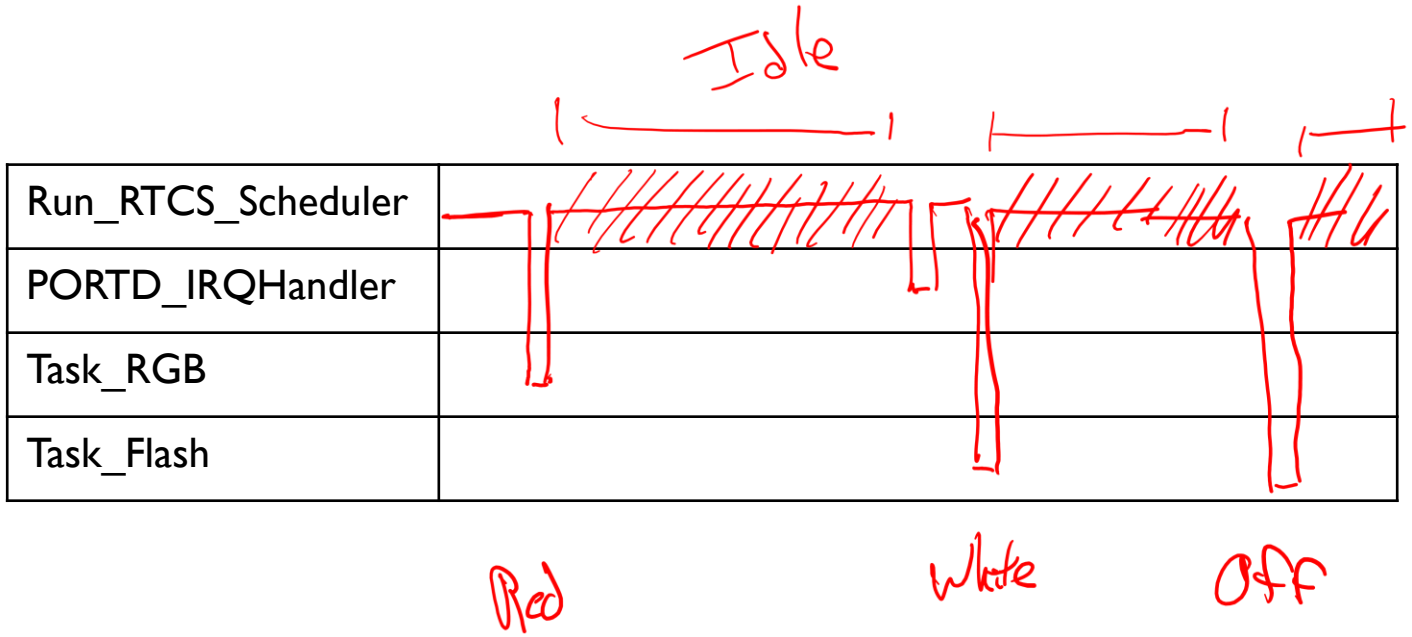




- Switches trigger interrupts when pressed or released
  - Handled by ISR PORTD_IRQHandler
  - ISR enables, disables tasks, and changes task periods. No more g_*_delay global variables!
- LEDs
  - Task_RGB and Task_Flash broken into FSMs (one state per color)
  - Scheduler controls LED timing (via task periods, changed by PORTD_IRQHandler)

Where is the Idle Time?





- Analysis
  - Examine call graph
  - Consider code execution timeline
- Idle time
  - Available to scheduler!

How to Change Program Components?

- Tasks
  - Get rid deciding whether to run and busy wait delay
- Switch ISR
  - Update scheduler's task table (not g_*) based on desired behavior
- Variables
  - g_w_delay, g_flash_LED, g_RGB_delay aren't needed any more since scheduler will take care of them

Task Changes

```
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;
        }
    }
}
```

- Eliminate run test
- Eliminate delay loop calls, as scheduler will provide delays
- Similar changes for Task_RGB_FSM

Switch ISR Changes

- Variables g_w_delay, g_flash_LED, g_RGB_delay aren't needed any more since scheduler will take care of their function
  - Use RTCS interface functions to update scheduling information
  - Code will update scheduler's data to "Make it so"
- If SW1 pressed
  - Enable Task_Flash_FSM, request to run it and disable Task_RGB_FSM
  - Else do opposite
- Update task periods Task_Flash_FSM and Task_RGB_FSM based on if SW2 is pressed
- Set, clear debug bit to show start and end of ISR on scope/logic analyzer

```
void PORTD_IRQHandler(void) {
  PTB->PSOR = MASK(DEBUG_ISR_POS);
  // Read switches
  if ((PORTD->ISFR & MASK(SW1_POS))) {
    if (SWITCH_PRESSED(SW1_POS)) { // flash white
      RTCS_Enable_Task(Task_Flash_FSM, 1);
      RTCS_Release_Task(Task_Flash_FSM);
      RTCS_Enable_Task(Task_RGB_FSM, 0);
    } else {
      RTCS_Enable_Task(Task_Flash_FSM, 0);
      RTCS_Enable_Task(Task_RGB_FSM, 1);
      RTCS_Release_Task(Task_RGB_FSM);
    }
  }
  if ((PORTD->ISFR & MASK(SW2_POS))) {
    if (SWITCH_PRESSED(SW2_POS)) {
      RTCS_Set_Task_Period(Task_Flash_FSM, W_DELAY_FAST);
      RTCS_Set_Task_Period(Task_RGB_FSM, RGB_DELAY_FAST);
    } else {
      RTCS_Set_Task_Period(Task_Flash_FSM, W_DELAY_SLOW);
      RTCS_Set_Task_Period(Task_RGB_FSM, RGB_DELAY_SLOW);
    }
  }
  // clear status flags
  PORTD->ISFR = 0xffffffff;
  PTB->PCOR = MASK(DEBUG_ISR_POS);
}
```

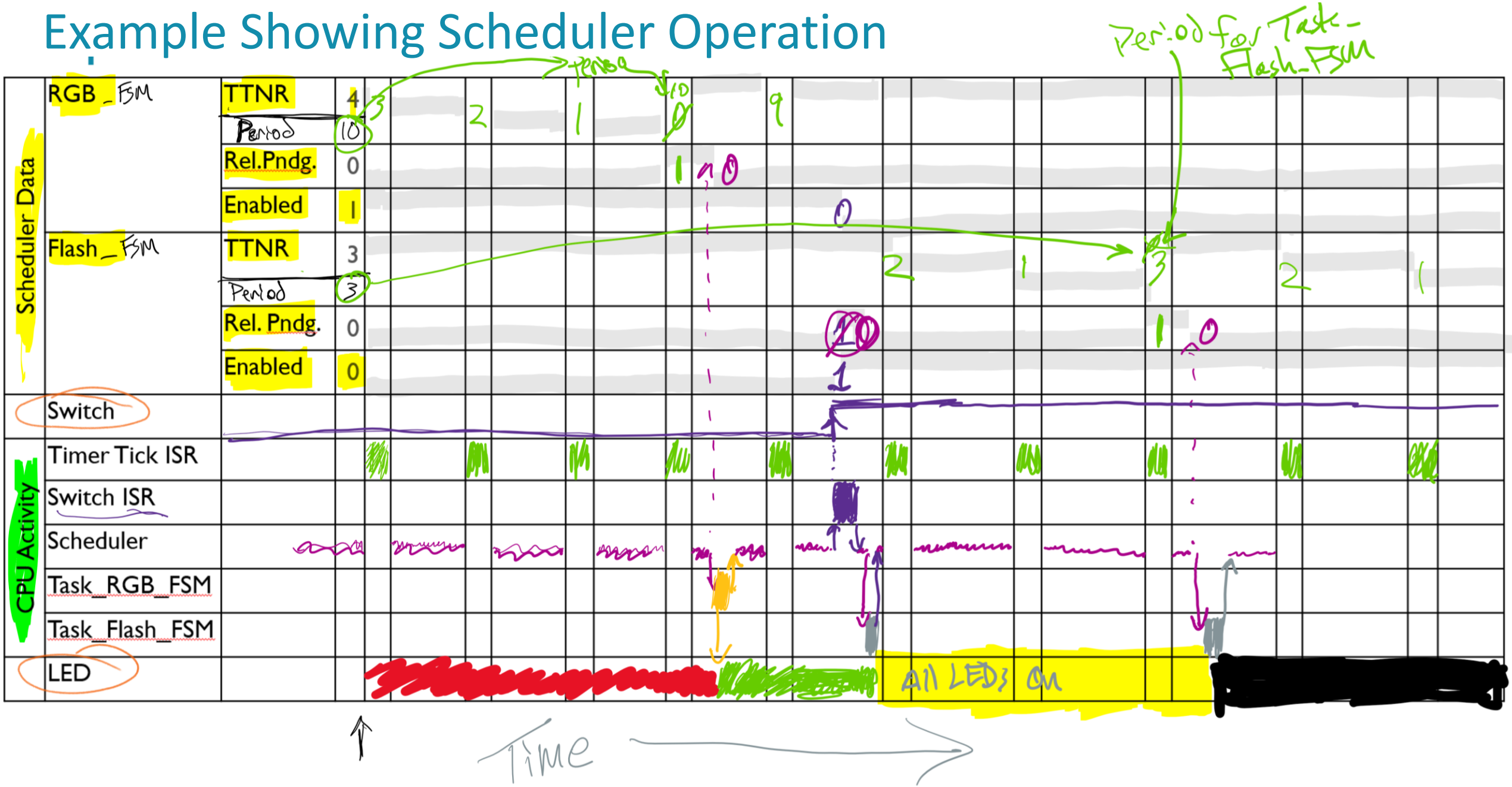
Scheduler Set-Up

```
int main (void) {
  Init_Debug_Signals();
  Init_RGB_LEDs();
  Init_Interrupts();

  RTCS_Init(100); // 100 Hz timer ticks

  RTCS_Add_Task(Task_Flash_FSM, 0, W_DELAY_SLOW);
  RTCS_Enable_Task(Task_Flash_FSM, 0);

  RTCS_Add_Task(Task_RGB_FSM, 1, RGB_DELAY_SLOW);
  RTCS_Enable_Task(Task_RGB_FSM, 1);

  RTCS_Run_Scheduler();  // This call never returns
}
```

Analyze Resulting Performance with Logic Analyzer

- Measure the responsiveness…
  - For switch 1 (Flash/RGB)
  - For switch 2 (Fast/Slow)
- How much idle time?

Example Showing Scheduler Operation



SLIDES FOR LATER

Scheduler Feature Summary

- To be added…

1. Triggering

- Two basic triggering options
  - Time-triggered: periodic thread runs every X ms. (Could have aperiodic time triggers too)
    - Structure the thread as an infinite loop with an OS ***delay*** call or wait for next interval
  - Event-triggered: Thread can run after event Y happens
    - Use an OS-provided synchronization primitive (event, semaphore etc.) to signal the event has occurred
    - Structure the thread as an infinite loop with an OS ***wait*** call
    - Will need the triggering thread(s) to use the OS ***signal*** call
- Allow multiple pending trigger requests?
  - What if periodic 2 ms thread hasn't gotten to run for 7 ms?
  - What if 5 events have occurred but thread hasn't run yet?

2. Messages

- Does the thread pass data to another thread (one-way, only one writer)?
  - If so, use OS-provided message primitives
- How much information is to be passed?
  - Event: something (implicitly defined) has happened
  - Data + event = message: something has happened, and here are the explicit details
- Does handshaking matter?
  - Does sender care if receiver has gotten the notification?
- Allow multiple pending messages, or just use the last one?
  - How much (if any) information buffering is needed? *Depends on how many events can occur before other thread can service them.*
  - Single item:
    - OS: use event or mailbox
  - Multiple items:
    - OS: use queue

3. Shared Data and Resources

- Do multiple threads need to update shared data or a common resource?
- ***Sharing data*** in a ***preemptive system*** (threads, ISRs) introduces risks
  - If **data updates are not atomic** (are divisible and can be interrupted)
    - Anything which takes multiple instructions to modify (anything in memory!)
    - Multiword variable (long int, float, double)
    - Multiword structure or object
  - If **multiple threads can write** to the same data variable

- Protect the object
  - Reduce or disable preemption manually
  - Use algorithm for protection: access flag, double buffering, etc.
  - Apply architectural solution (e.g. server)
  - Use OS-provided support: mutex

POWER AND ENERGY MANAGEMENT

Automatically Saving Power & Energy when Idle

- Scheduler **knows** if system is idle (no tasks ready to execute)
  - So it can put processor into low-power mode
- Any enabled IRQ will wake up MCU, run ISR
- After ISR, scheduler resumes running (and perhaps sleeping!)

- Special case: multiple tasks may be ready to run
  - Break out of for loop after completing one task in order to restart at top of priority table
  - There may still be other tasks (lower priority) with runs requested
  - Add variable to count how many tasks were run in this while loop iteration, use this to sleep

```
void RTCS_Run_Scheduler(void) {
/* Loop forever */
  while (1) {
    tasks_run = 0
    /* Check each task */
    for (i=0 ; i<RTCS_MAX_TASKS ; i++) {
       if task i is ready
              run task i
              tasks_run++;
              break;
    } // at end of for loop
    if tasks_run == 0
       // go to sleep
       __wfi()
  } // end of while loop
}
```