

Different types of embedded systems.

- Real-Time Systems
 - Example: Motor controller for a robot
 - Focuses on reliability
 - Lacks advanced features
 - All parts of the system must be well understood.
 - Simplicity is important
 - Runs either on bare hardware or an RTOS.
 - Runs on a microcontroller (usually from on chip memory).
- User interface and/or feature driven
 - Example: Smart phone – click an app and it runs.
 - Advanced features are important.
 - Runs on a traditional OS, such as Linux.
 - Runs on a system-on-chip with off-chip main memory storage.
 - Not well suited for real-time software.

What is real-time software?

- Traditional software
 - Must produce a correct result.
 - Get input -> calculate value -> output data.
- Multitasking software
 - Must ensure concurrently executing software tasks correctly cooperate to produce the desired results.
 - One process captures data.
 - Another process calculates results.
 - Coordination is needed on when each process runs and how they handle shared data.
- Real-Time software
 - Multi-tasking.
 - Must guarantee the correct result.
 - **Must deliver the results on-time.** Deadlines must be met.
 - Requires careful top-level design of the entire system.
 - Typically demands full control of the hardware.
 - Must understand all tasks, the sequence of events, and how to schedule them.

What is an RTOS?

- Supports multitasking.
 - Similar to multithreading under a traditional OS.
- Real-time scheduling of task.
- Inter-task communications and synchronization.
 - Semaphores and related objects.
- Software timers.
- Real-time dynamic memory allocation.
- And that's it!
 - RTOS is often just a library linked to your application.
 - More advanced features of some RTOSSs.
 - File systems
 - Internet connectivity
 - GUI

Fundamental Real-Time Concepts

- Real-time systems have deadlines.
 - A “hard real-time” system fails if it misses just one deadline.
- Timing of a real-time task:
 - Example studying for a test.

Interrupt:

Professor announces
test is a week away.

Latency:

No studying done
for 2 days.

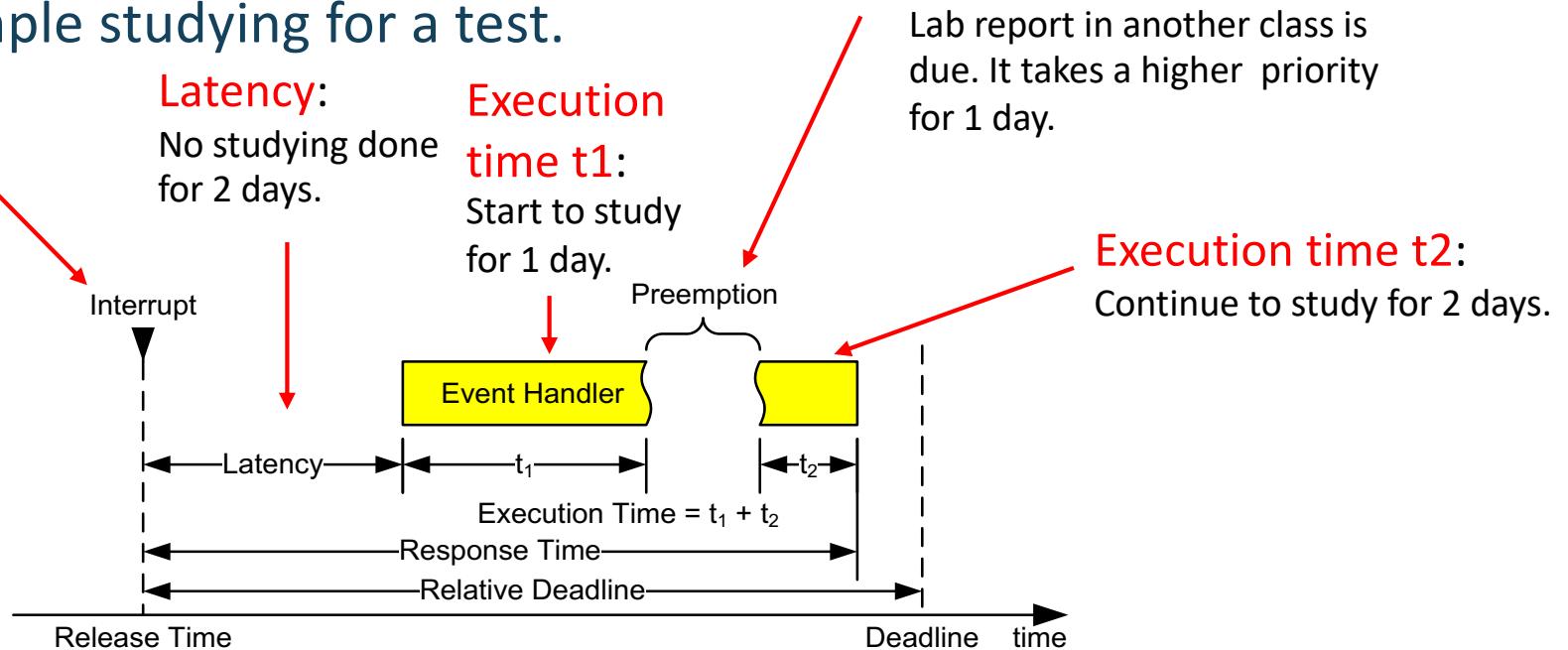
Execution
time t_1 :

Start to study
for 1 day.

Preemption:

Lab report in another class is
due. It takes a higher priority
for 1 day.

Execution time t_2 :
Continue to study for 2 days.



- If the max (Response Time) < Relative Deadline
 - Success!! You pass the test!

Real-time software challenges

- Keep maximum latency low
 - Preemptive multitasking.
 - Task prioritization.
 - Optimized task scheduler.
- Support for data and resource sharing among tasks
 - Tends to affect latency and response time of unrelated tasks.
- Keep maximum execution time low
 - Optimization (on both software and hardware level).
 - Use of “deterministic” algorithms with predictable maximum execution time.

ECE 3849 Contents

- Real-time scheduling theory
 - How to determine if a real-time system is schedulable.
- Interrupts
 - Low latency.
 - Challenging resource-sharing issues.
- RTOS
 - Resource sharing with less impact on latency / response time.
- Input / Output
 - Time interval measurement and timed I/O.
 - Relaxing the I/O latency requirements.
 - Reduction of CPU load due to I/O.
- CPU architecture, memory
 - Execution time optimization.

ECE3849
D-Term 2021

Real Time Embedded Systems

Module 1 Part 1

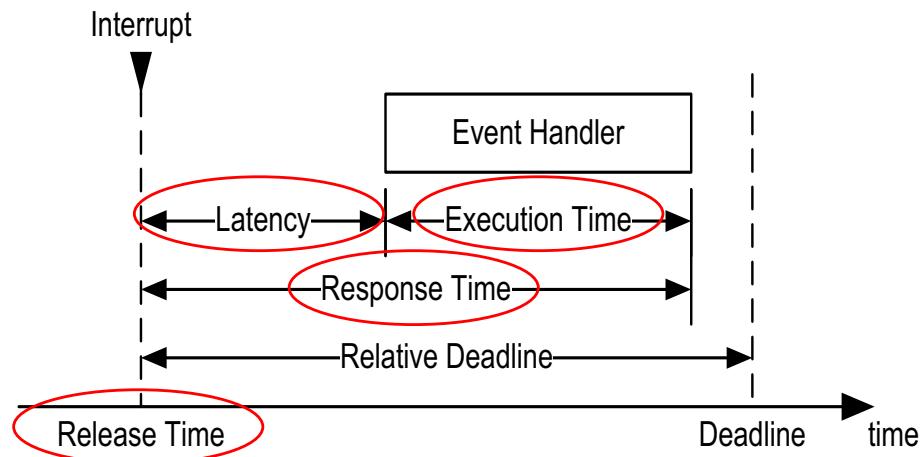
Module 1 Part 1 Overview

- Meeting and Setting deadlines
 - Latency
 - Execution Time
 - Response Time
- Scheduling Strategies
 - Round Robin Polling
 - Priority Polling
 - Interrupts with Preemption
- Rate-Monotonic Scheduling Theory

Events without Preemption

- Simplest case: No preemption

- The event handler / task runs from start to finish without interruption.
- Conditions this occurs under
 - Only one task running.
 - The task is highest priority.
- Event is serviced immediately.



- Release time
 - Hardware notifies the software that an external event has occurred.

- Latency
 - The time between when the event / interrupt happens and when the event handler / task starts to execute.
 - In this simple case depends on the microcontroller being used and the time for the software to context switch.

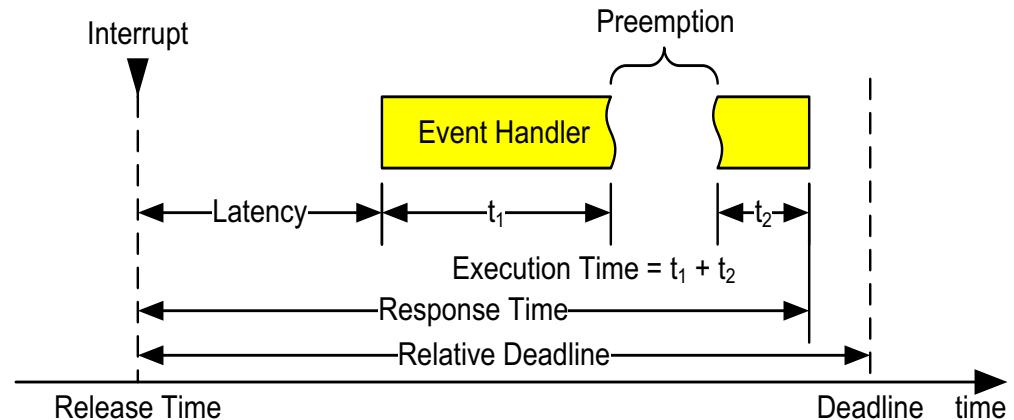
- Execution time depends on the event handler code and the hardware running the code.

- Fast hardware => faster execution time.
- Software should always optimize for the worst case.
- For scheduling hard deadlines, the average has little importance.

- Response time = Maximum Latency + Maximum Execution Time.

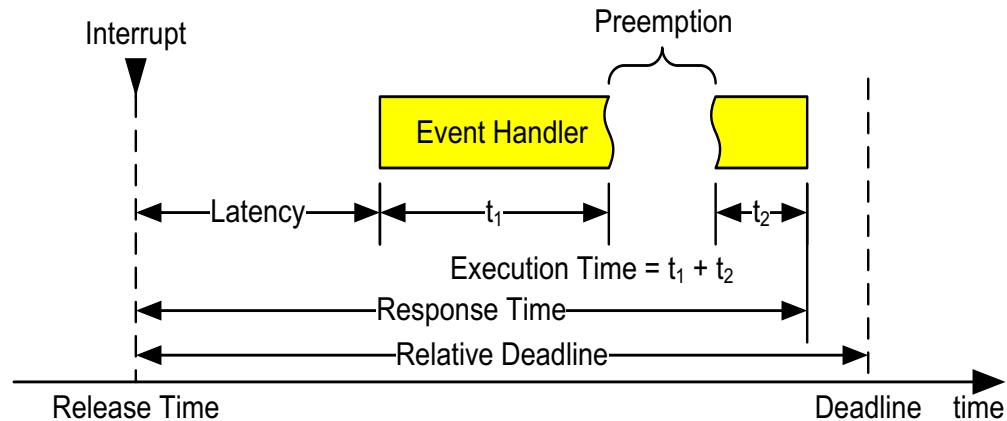
Events with Preemption

- There are multiple tasks
 - Each task has a priority.
 - Higher priority tasks can interrupt lower priority task.



- Release time
 - Same as before: Hardware notifies the software that an external event has occurred.
- Latency
 - Lower priority tasks may have to wait for higher priority tasks to complete before it starts.
 - Best case it starts immediately.
 - **Worst case** has to wait for other tasks to complete.
- Execution time now depends not only the event handler / task but also execution times of higher priority tasks.
 - Best case it executes without preemption, start to finish.
 - **Worst case** has to wait for all other higher priority task to complete.
- **Response time = Maximum Latency + Maximum Execution time.**

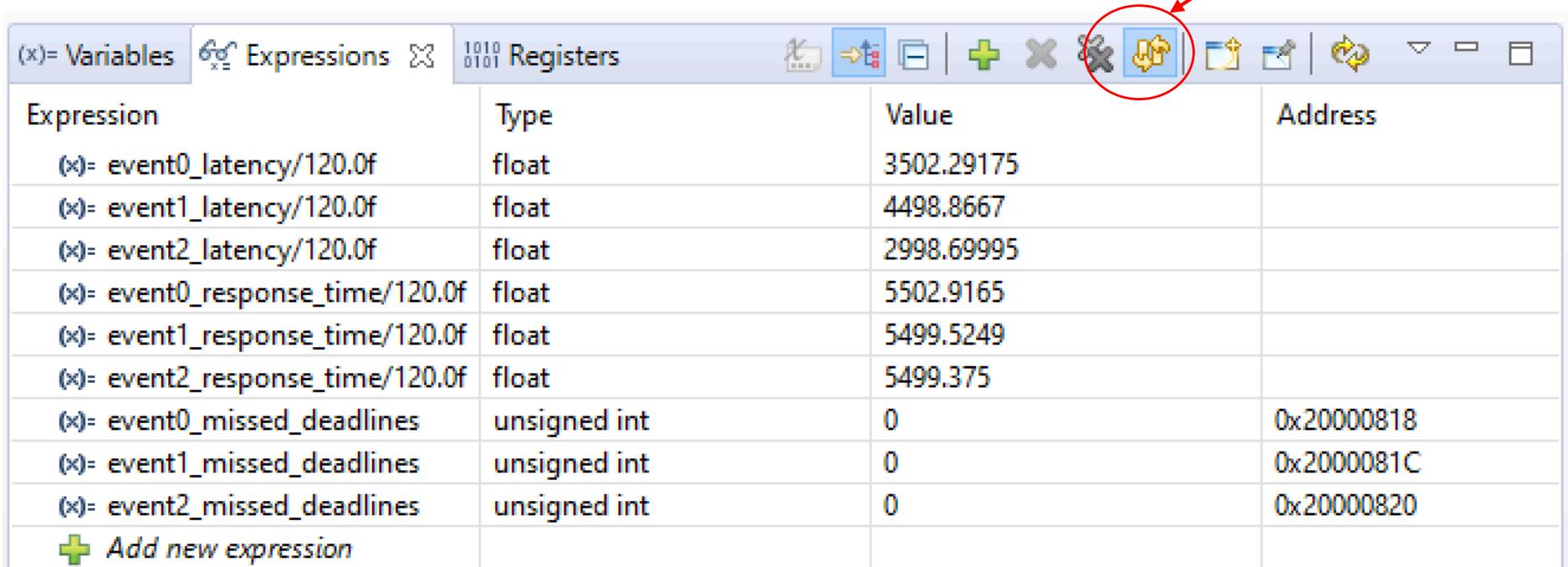
Meeting Deadlines



- The software must respond by the deadline.
 - Some deadlines are absolute, it must happen by 1:00 pm today.
 - Relative deadlines are defined relative to the release time.
 - When considering how to schedule a task, relative deadlines are important.
 - For periodic events the relative deadline equals the period of the event, as task must complete before the next event occurs.
- Relative deadline
 - Hard deadline – system fails if it misses any deadline . Some Examples?
 - Power cycle indicator (depends on the consequence)?
 - Thrusters on Mars rover when landing – come down too fast and loose the rove.
 - Soft deadline – systems tolerate if it misses some deadlines – Some Examples?
 - Non-critical sensor reading, home thermometer? Lab submissions – only because late deduction.
- A task is “schedulable” if the Maximum Response Time < Relative Deadline
- What might cause the response time to vary?
 - Performance throttling of hardware , slows down? Sequence of tasks change the average.
 - Conditional equations: if / else. If takes 1 msec, else 10 msec.

CCS Project: ece3849_int_latency

- Program: `ece3849_int_latency`
 - Generates three events: event0, event1 and event2
 - Each event has a
 - Period specified in usec
 - Execution time specified in usec
 - Measures the latency and execution time of each event in clock ticks.
 - The clock ticks are divided by 120 to convert to usec, because the clock is running at 120 MHz.
 - Increments a counter if any deadlines are missed.
- Results are viewed by adding expressions and using continuous refresh



The screenshot shows the CCS (Code Composer Studio) interface with the 'Expressions' tab selected. A red circle highlights the 'Continuous Refresh' button (a blue icon with a circular arrow) in the toolbar above the table. The table lists various expressions and their values:

| Expression | Type | Value | Address |
|---|--------------|------------|------------|
| <code>(x)= event0_latency/120.0f</code> | float | 3502.29175 | |
| <code>(x)= event1_latency/120.0f</code> | float | 4498.8667 | |
| <code>(x)= event2_latency/120.0f</code> | float | 2998.69995 | |
| <code>(x)= event0_response_time/120.0f</code> | float | 5502.9165 | |
| <code>(x)= event1_response_time/120.0f</code> | float | 5499.5249 | |
| <code>(x)= event2_response_time/120.0f</code> | float | 5499.375 | |
| <code>(x)= event0_missed_deadlines</code> | unsigned int | 0 | 0x20000818 |
| <code>(x)= event1_missed_deadlines</code> | unsigned int | 0 | 0x2000081C |
| <code>(x)= event2_missed_deadlines</code> | unsigned int | 0 | 0x20000820 |
| + Add new expression | | | |

ece3849_int_latency

Defining Parameters and initializing values

```
33 // event and handler definitions
34 #define EVENT0_PERIOD          6007    // [us] event0 period
35 #define EVENT0_EXECUTION_TIME  2000    // [us] event0 handler execution time
36
37 #define EVENT1_PERIOD          8101    // [us] event1 period
38 #define EVENT1_EXECUTION_TIME  1000    // [us] event1 handler execution time
39
40 #define EVENT2_PERIOD          12301   // [us] event2 period
41 #define EVENT2_EXECUTION_TIME  2500    // [us] event2 handler execution time
42
43 // build options
44 // #define DISABLE_INTERRUPTS_IN_ISR // if defined, interrupts are disabled in the body of each ISR
45
46 // measured event latencies in clock cycles
47 uint32_t event0_latency = 0;
48 uint32_t event1_latency = 0;
49 uint32_t event2_latency = 0;
50
51 // measured event response time in clock cycles
52 uint32_t event0_response_time = 0;
53 uint32_t event1_response_time = 0;
54 uint32_t event2_response_time = 0;
55
56 // number of deadlines missed
57 uint32_t event0_missed_deadlines = 0;
58 uint32_t event1_missed_deadlines = 0;
59 uint32_t event2_missed_deadlines = 0;
60
```

- Sets period and execution time for each event.
- Small value added to cause variation in phase.

For each event initializes

- Latency
- Response time
- Number of missed deadlines

ece3849 int latency

TM4C1294NCPDT Timer Summery

- There are two General Purpose Timers (GPTM Modules)
 - TimerA and B can be used individually in 16-bit mode or together for 32-bit counts.
 - They can count up or down.
 - They can be a one shot timer or a periodic timer that reloads itself when the time is reached.
 - Each timer can configure up to 8 interval timers.
 - See datasheet for more details.
- This example uses 3 interval timers of the TimerA module.
 - Each event uses an interval timer with an interrupt output to trigger the event handler task.
- The the timer units are in number of CPU clocks.
 - The CPU is running at 120 MHz.
 - There are a 120 clocks in 1 usec.

```
61 // timer periods in clock cycles (expecting 120 MHz clock)
62 // 120 clock cycles in 1 usec
63 #define TIMER0_PERIOD (120 * EVENT0_PERIOD)
64 #define TIMER1_PERIOD (120 * EVENT1_PERIOD)
65 #define TIMER2_PERIOD (120 * EVENT2_PERIOD)
```

ece3849 int latency

Configuring the Timer Interrupts

```
// initialize general purpose timers 0-2 for periodic inter
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerDisable(TIMER0_BASE, TIMER_BOTH);
TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
TimerLoadSet(TIMER0_BASE, TIMER_A, TIMER0_PERIOD - 1);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
IntPrioritySet(INT_TIMER0A, 0); // 0 = highest priority, 31 = lowest
IntEnable(INT_TIMER0A);

SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
TimerDisable(TIMER1_BASE, TIMER_BOTH);
TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
TimerLoadSet(TIMER1_BASE, TIMER_A, TIMER1_PERIOD - 1);
TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
IntPrioritySet(INT_TIMER1A, 32); // 0 = highest priority, 31 = lowest
IntEnable(INT_TIMER1A);

SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
TimerDisable(TIMER2_BASE, TIMER_BOTH);
TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC);
TimerLoadSet(TIMER2_BASE, TIMER_A, TIMER2_PERIOD - 1);
TimerIntEnable(TIMER2_BASE, TIMER_TIMA_TIMEOUT);
IntPrioritySet(INT_TIMER2A, 64); // 0 = highest priority, 31 = lowest
IntEnable(INT_TIMER2A);

IntMasterEnable(); // comment for polled scheduling or CI

TimerEnable(TIMER0_BASE, TIMER_A); // comment for CPU load
TimerEnable(TIMER1_BASE, TIMER_A);
TimerEnable(TIMER2_BASE, TIMER_A);
```

- Enables the Timer0 peripheral to be used.
- Disables operation during configuration.
- Counts down from the TIMERx_PERIOD
- Enables the timer interrupt to trigger when it reaches 0.
- Sets the priority of the interrupt.
- Enables the CPU to respond to the timer interrupt.

These function calls are documented in in the TivaWare Peripheral Driver Library User's Guide.

Enables all counters once configuration is complete

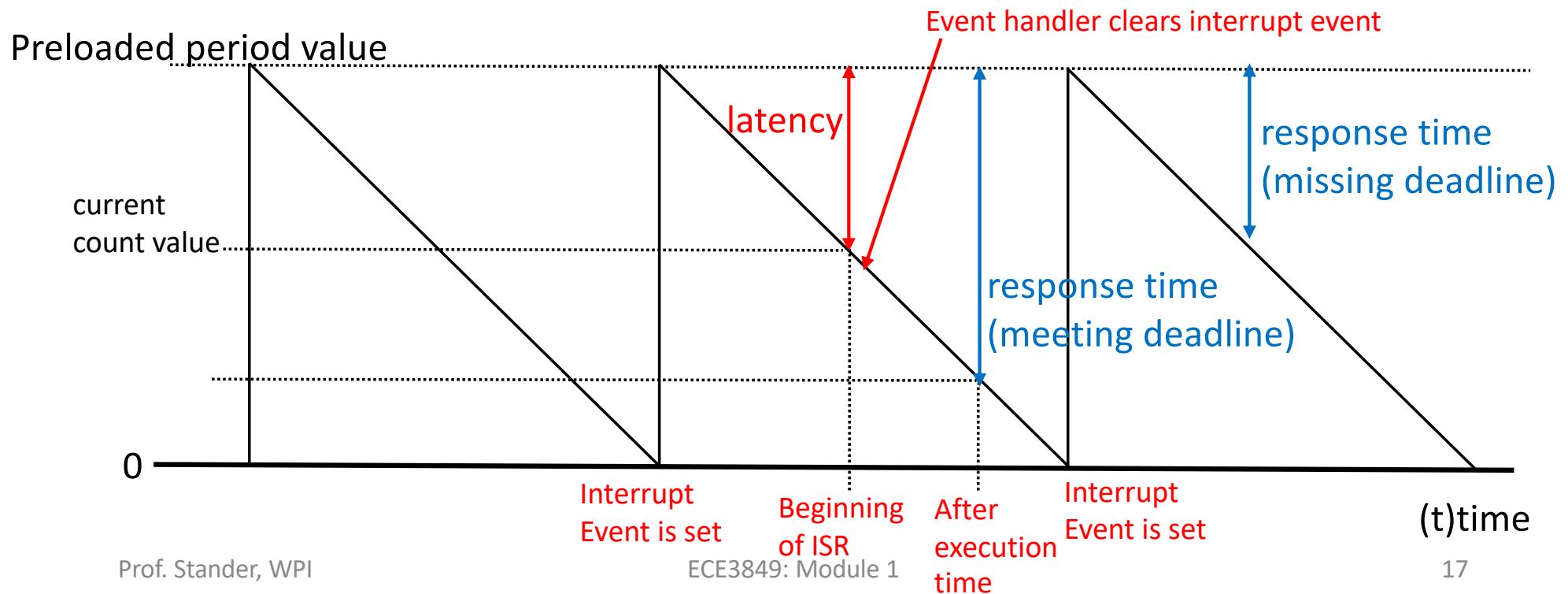
ece3849_int_latency

Infinite while loop

- There are several different scheduling modes we will explore.
 - Each will have different behaviors but all will have an infinite while loop.
- While in the loop, the timers control when events happen.
 - When an event happens an interrupt bit is set.
 - An event handler routine is called to service the event.
- The event handler does the following.
 - Reads the current time.
 - Clears the interrupt status register.
 - Calculates the latency.
 - Waits for the execution time.
 - Calculates the response time.

Calculating latency and response time.

- During configuration the preloaded period value is programmed.
- Timer decrements by 1 every clock.
- It counts down to zero then sets its interrupt status and reloads itself.
 - The release time is when the current count value = preloaded period value.
- Upon entering the event handler the interrupt event is cleared.
- Time_since_interrupt = timer_period – current_timer_count
 - Latency = preloaded period value – current timer value at the start of the interrupt.
 - Response time = preloaded period value – current timer value after the execution time delay.
 - If deadline is missed, the interrupt event is set.
 - We account for this by adding an additional preloaded period value.



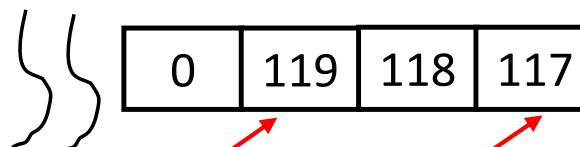
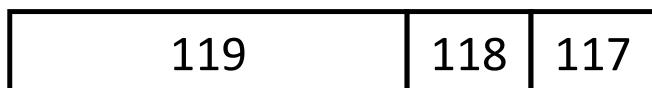
Timer Functional Calls Used

#1 TimerLoadSet(TIMERO_BASE, TIME_A, 119)

This examples sets timeout count 1usec = 120 clock counts

#2 TimerEnable(TIMERO BASE, TIMER A)

Enables counting down – 1 count per CPU clock edge



#3 Interrupt bit is set when timer reach 0 and then rolls back to pre-set value.

#4 TimerIntStatus(TIMERO BASE, 1)

Reads the interrupt status register.



#7 TimerIntClear(TIMERO_BASE, TIMER_TIMA_TIMEOUT)
Clears Interrupt bit in event handler

ece3849_int_latency: Calculating Latency

- Each event handler is similar but accessing different hardware registers and updates different variables.

- Calculating Latency

- TIMER0_TAR_R is the hardware register containing the current time.
 - Latency is the difference between the period and the current time.
 - If it is the largest latency, then it updates the value.

```
193 void event0_handler(void)
194 {
195 #ifdef DISABLE_INTERRUPTS_IN_ISR
196     IntMasterDisable();
197 #endif
198     uint32_t t = TIMER0_PERIOD - TIMER0_TAR_R; // read
199     if (t > event0_latency) event0_latency = t; // meas
200     TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); //
201
202     delay_us(EVENT0_EXECUTION_TIME); // handle event0
203 }
```

- Clears the interrupt, so it can be triggered again.
 - Sits in a loop for the execution time
- **TIMER0_TAR_R** is defined in tm4c1294ncpdt.h.
 - It defines register names that map to hardware register addresses.
- Hardware Registers defined in the **TM4C129NCPDT Microcontroller Datasheet**.

ece3849_int_latency

Calculating Response Time

- Handling a missed deadline
 - If the deadline was exceeded the counter rolls over and a new interrupt is generated.
 - The missed deadlines value is incremented
 - An extra period is added to the start count due to the counter rolling over.

- If the deadline was not missed, the original count is just the period.

```
203  
204 if (TimerIntStatus(TIMER0_BASE, 1) & TIMER_TIMA_TIMEOUT) {  
205     event0_missed_deadlines++;  
206     t = 2 * TIMER0_PERIOD; // timer overflowed since last interrupt  
207 }  
208 else t = TIMER0_PERIOD;  
209 t -= TimerValueGet(TIMER0_BASE, TIMER_A); // read Timer A value  
210 if (t > event0_response_time) event0_response_time = t; //
```

- The execution time is the period start count – the current time value
 - `TimerValueGet` function is equivalent in function accessing `TIMER0_TAR_R` directly but uses the API.
- If the response time is the maximum value the response time is updated.