ECE3849
D-Term 2021

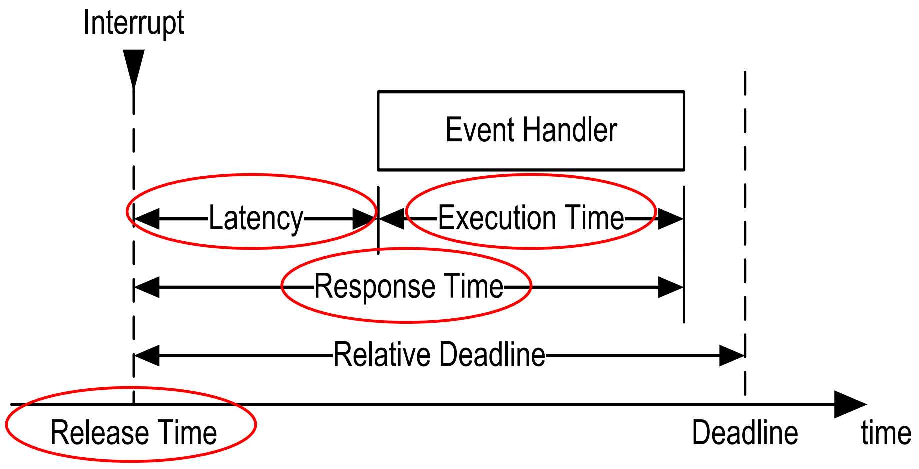
Real Time Embedded Systems

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

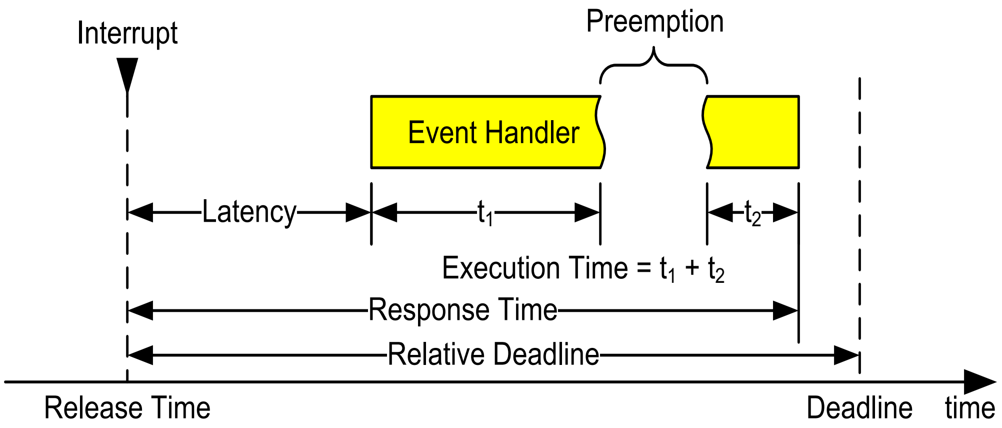
Events without Preemption



- 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.

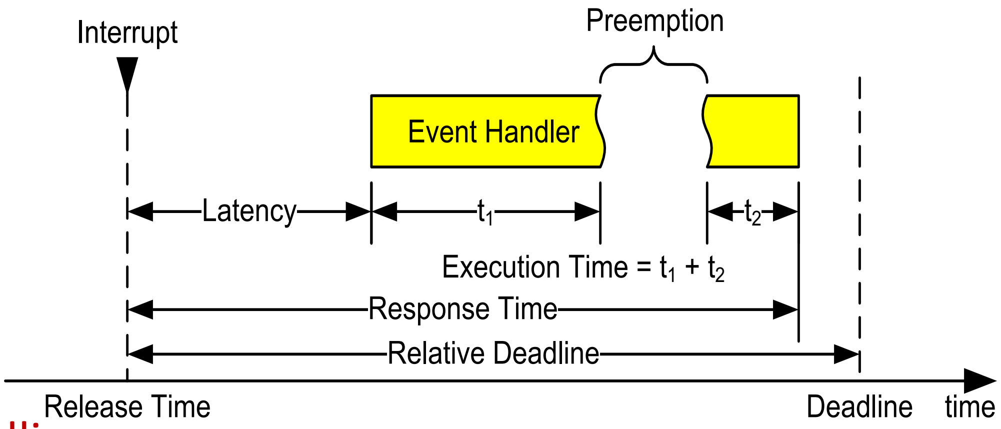
Events with Preemption

- 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.

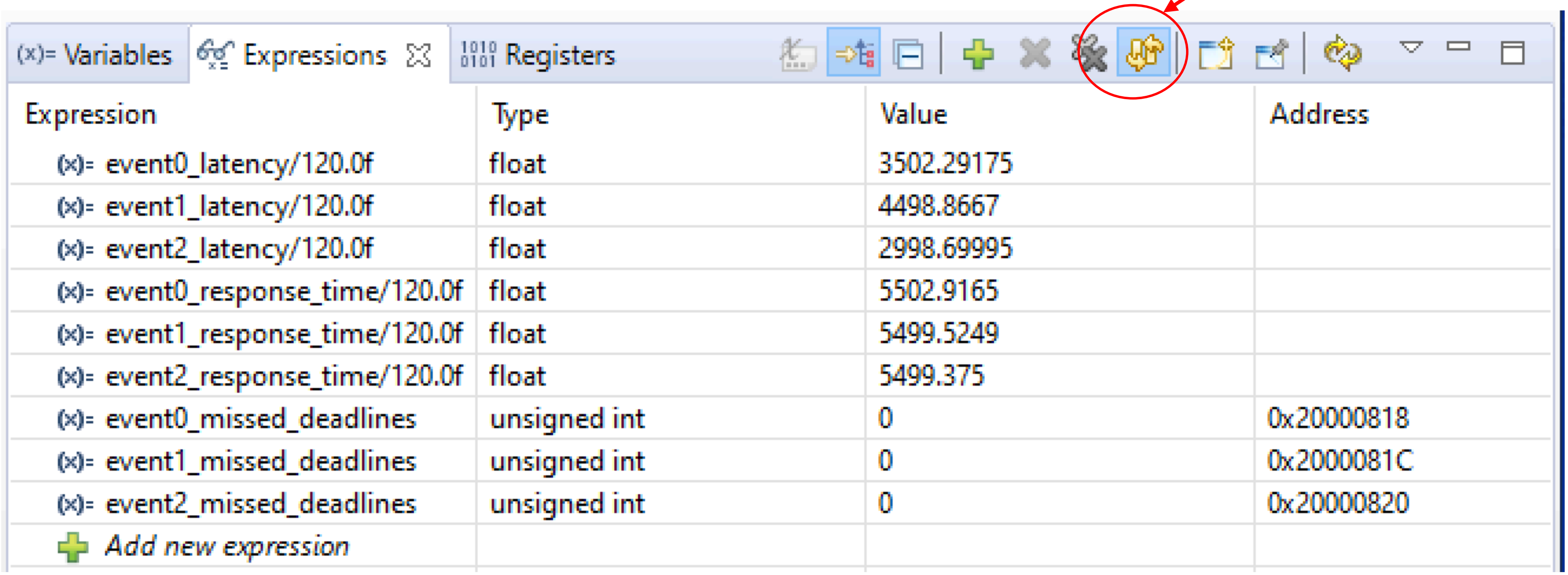
Meeting Deadlines



- 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.

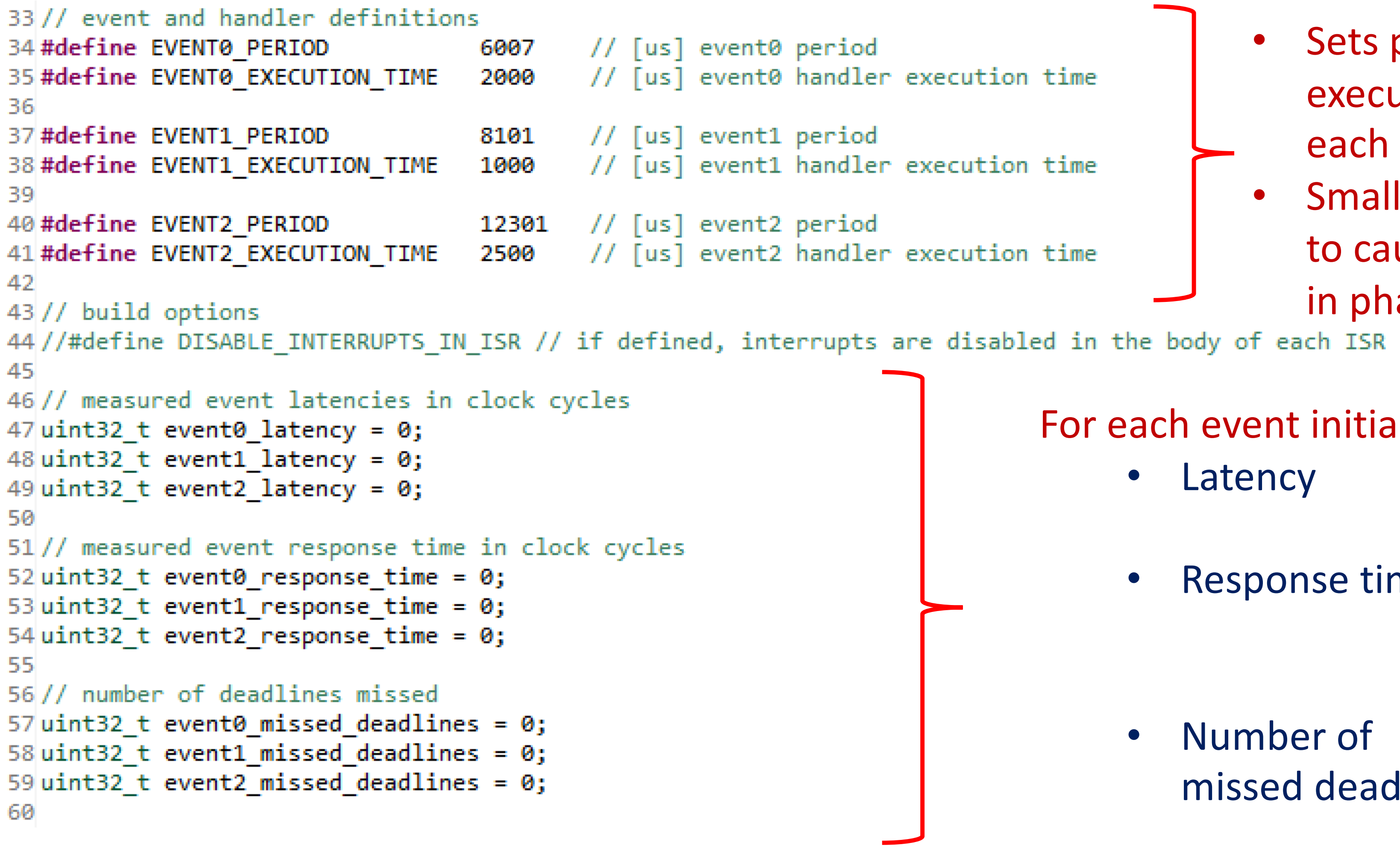
CCS Project: ece3849_int_latency

- 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



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

ece3849_int_latency
Defining Parameters and initializing values



```c
// event and handler definitions
#define EVENT0_PERIOD           6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME   2000    // [us] event0 handler execution time

#define EVENT1_PERIOD           8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME   1000    // [us] event1 handler execution time

#define EVENT2_PERIOD           12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME   2500    // [us] event2 handler execution time

// build options
//#define DISABLE_INTERRUPTS_IN_ISR // if defined, interrupts are disabled in the body of each ISR

// measured event latencies in clock cycles
uint32_t event0_latency = 0;
uint32_t event1_latency = 0;
uint32_t event2_latency = 0;

// measured event response time in clock cycles
uint32_t event0_response_time = 0;
uint32_t event1_response_time = 0;
uint32_t event2_response_time = 0;

// number of deadlines missed
uint32_t event0_missed_deadlines = 0;
uint32_t event1_missed_deadlines = 0;
uint32_t event2_missed_deadlines = 0;
```

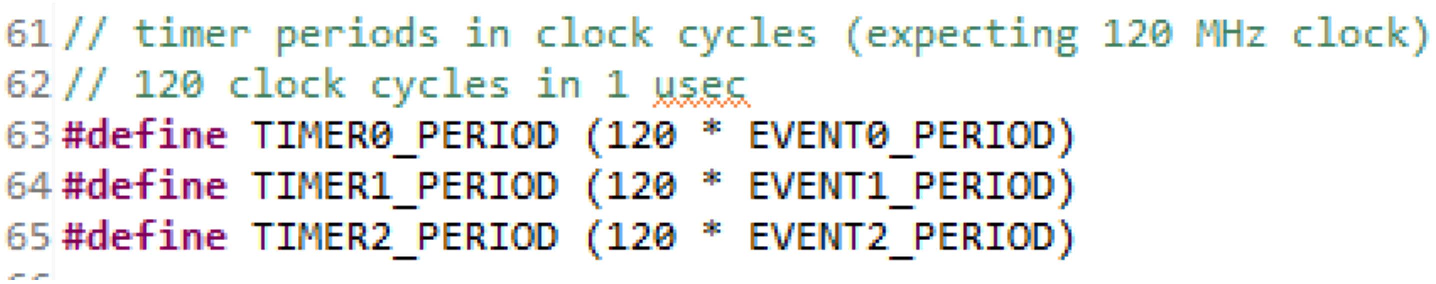
- Sets period and execution time for each event.
- Small value added to cause variation in phase.

For each event initializes

- Latency
- Response time
- Number of missed deadlines

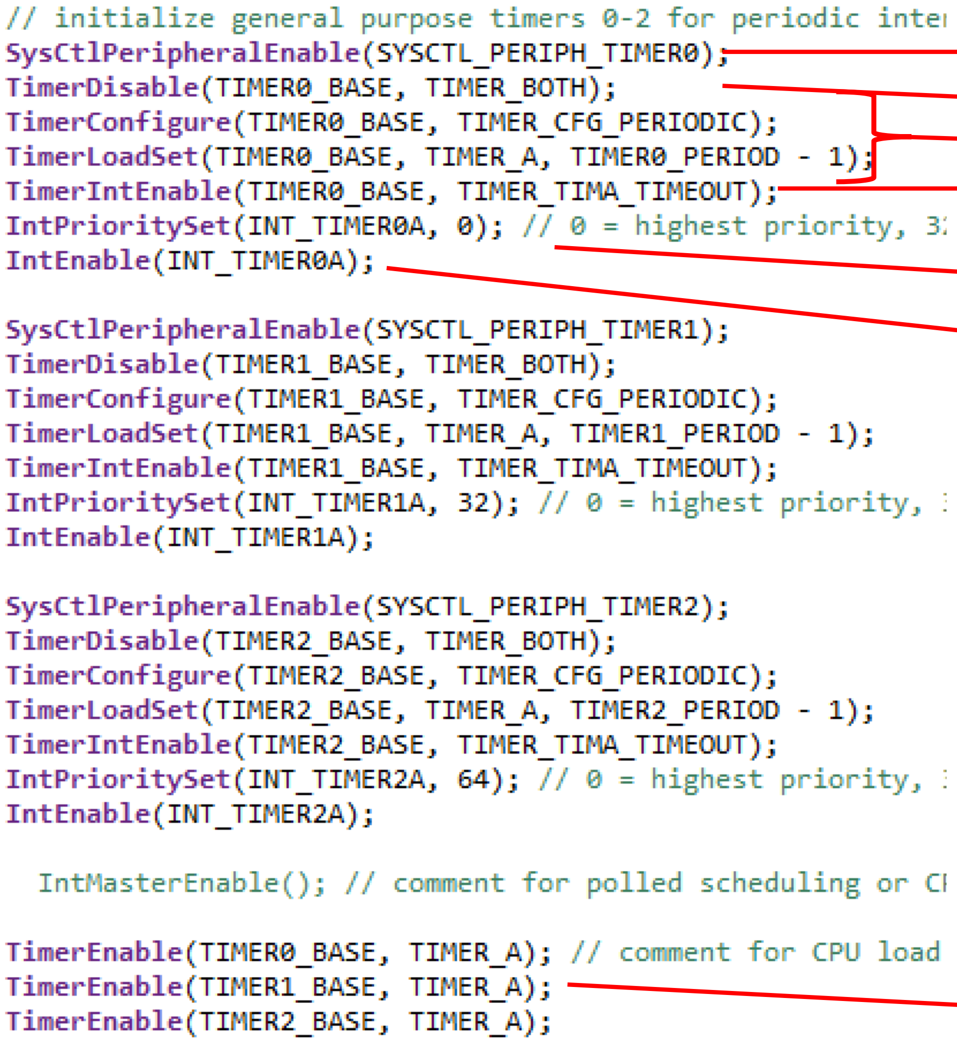
ece3849_int_latency TM4C1294NCPDT Timer Summery

- CORRECTION – Changed from original slides posted.
- There are two General Purpose Timers (GPTM Modules)
  - There are 8 GPTM timer modules.
  - Each GPTM timer has two sub timers - 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.
  - See datasheet for more details.
- This example uses 3 GPTM timers using submodule TimerA.
  - 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.



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

ece3849_int_latency
Configuring the Timer Interrupts



```
// initialize general purpose timers 0-2 for periodic inte
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, 3
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,
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,
IntEnable(INT_TIMER2A);

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

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.

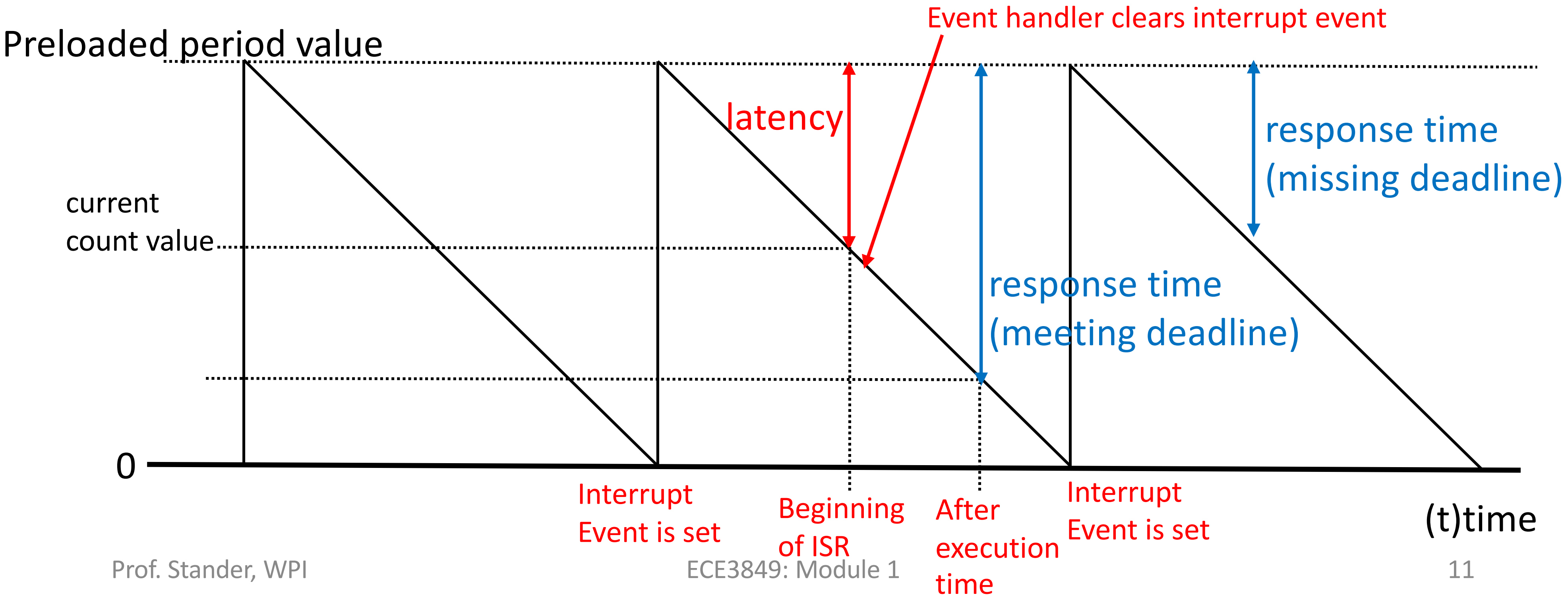
Enables all counters once configuration is complete

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.

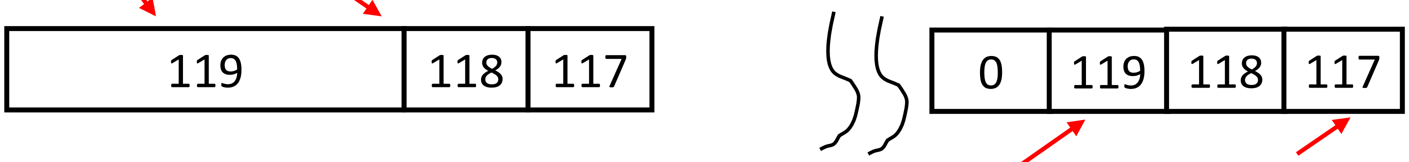
- During configuration the preloaded period value is programmed.
- Timer decrements by 1 every clock.
- It counts down to zero then sets its interrupt status and reloads itself.
  - The release time is when the current count value = preloaded period value.
- Upon entering the event handler the interrupt 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(TIMER0_BASE, TIME_A, 119)
This examples sets timeout count 1usec = 120 clock counts

#2 TimerEnable(TIMER0_BASE, TIMER_A)
Enables counting down – 1 count per CPU clock edge

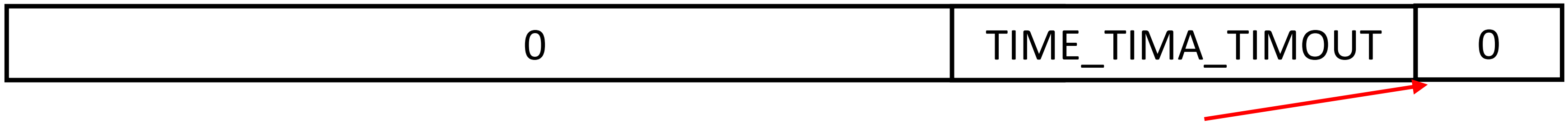


#5 In event handler gets the time
TimerValueGet(TIMER0_BASE, TIMER_A)

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

#6 Time in usec since last interrupt = (119 – TimerValueGet output)/120

#4 TimerIntStatus(TIMER0_BASE, 1)
Reads the interrupt status register.

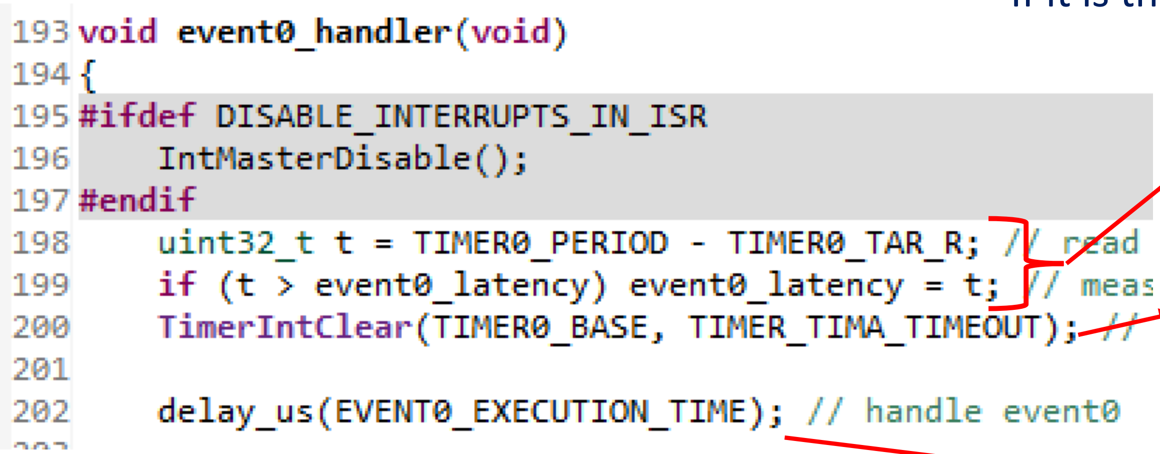


#7 TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT)
Clears Interrupt bit in event handler

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.



```
void event0_handler(void)
{
#ifdef DISABLE_INTERRUPTS_IN_ISR
    IntMasterDisable();
#endif
    uint32_t t = TIMER0_PERIOD - TIMER0_TAR_R; // read
    if (t > event0_latency) event0_latency = t; // meas
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); //

    delay_us(EVENT0_EXECUTION_TIME); // handle event0
```

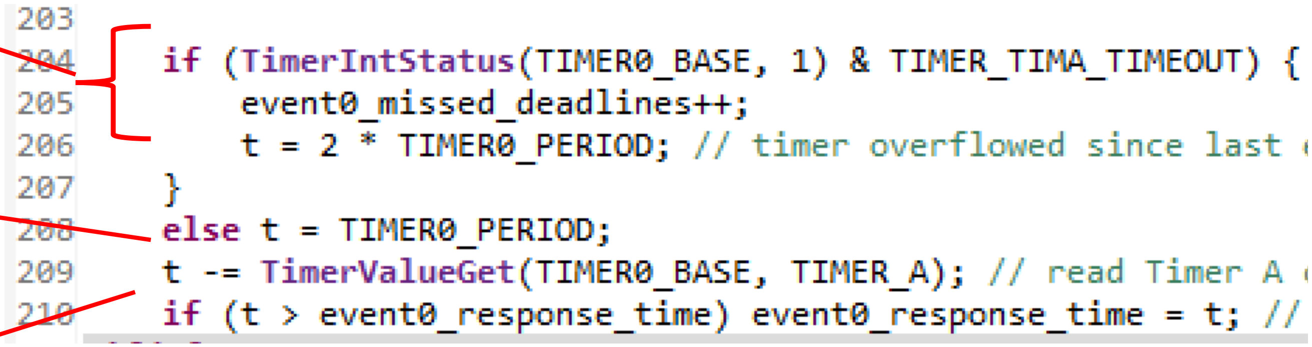
- Clears the interrupt, so it can be triggered again.

- 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.

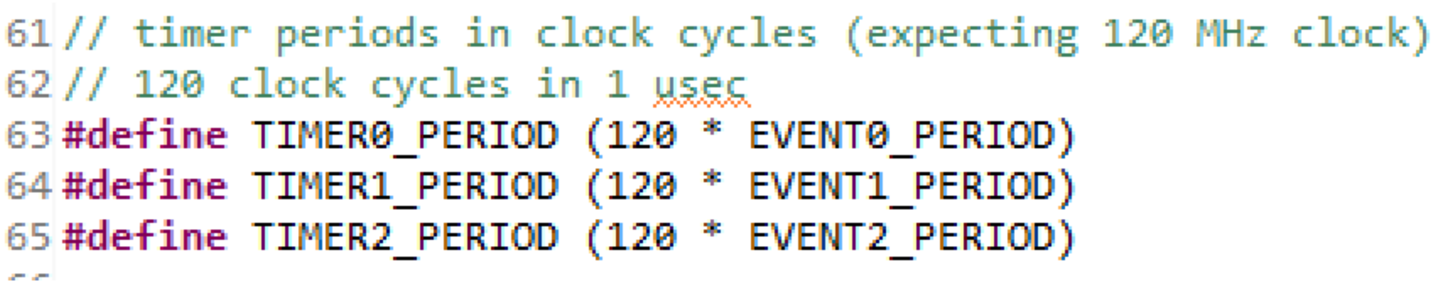


```
if (TimerIntStatus(TIMER0_BASE, 1) & TIMER_TIMA_TIMEOUT) {
    event0_missed_deadlines++;
    t = 2 * TIMER0_PERIOD; // timer overflowed since last
}
else t = TIMER0_PERIOD;
t -= TimerValueGet(TIMER0_BASE, TIMER_A); // read Timer A
if (t > event0_response_time) event0_response_time = t; //
```

- 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.

ece3849_int_latency
TM4C1294NCPDT Timer Summery

- CORRECTION !!!!!!!
- There are two General Purpose Timers (GPTM Modules)
  - There are 8 GPTM timer modules.
  - Each GPTM timer has two sub timers - 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.
  - See datasheet for more details.
- This example uses 3 GPTM timers using submodule TimerA.
  - 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.

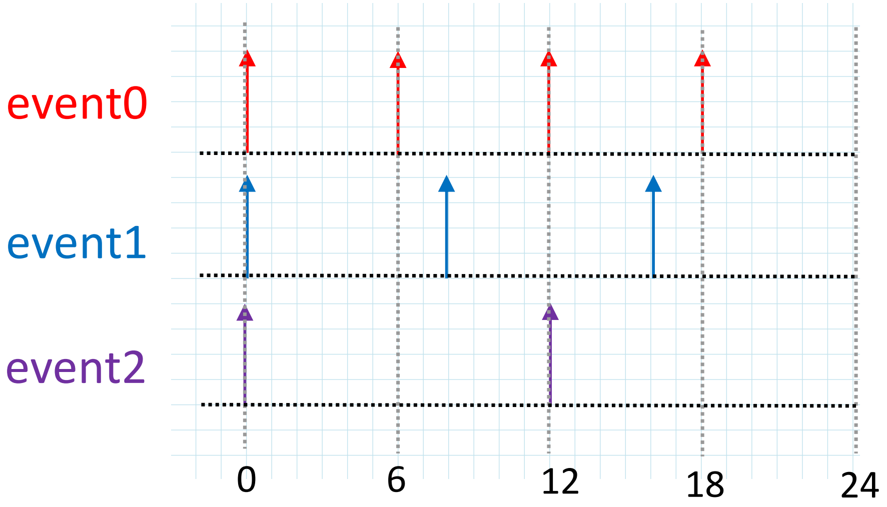


```
// timer periods in clock cycles (expecting 120 MHz clock)
// 120 clock cycles in 1 usec
#define TIMER0_PERIOD (120 * EVENT0_PERIOD)
#define TIMER1_PERIOD (120 * EVENT1_PERIOD)
#define TIMER2_PERIOD (120 * EVENT2_PERIOD)
```

Canonical Real-Time Systems

- System Assumptions / Rules
  1. All Events are periodic.
  2. The relative deadline = period.
     - The event needs to finish before it can be called again.
  3. The events are not phase aligned and can happen at anytime relative to each other.
- Example: There are three events: event0, event1 and event2

Event	Period	Execution Time
event0	6 ms	2 ms
event1	8 ms	1 ms
event2	12 ms	2.5 ms

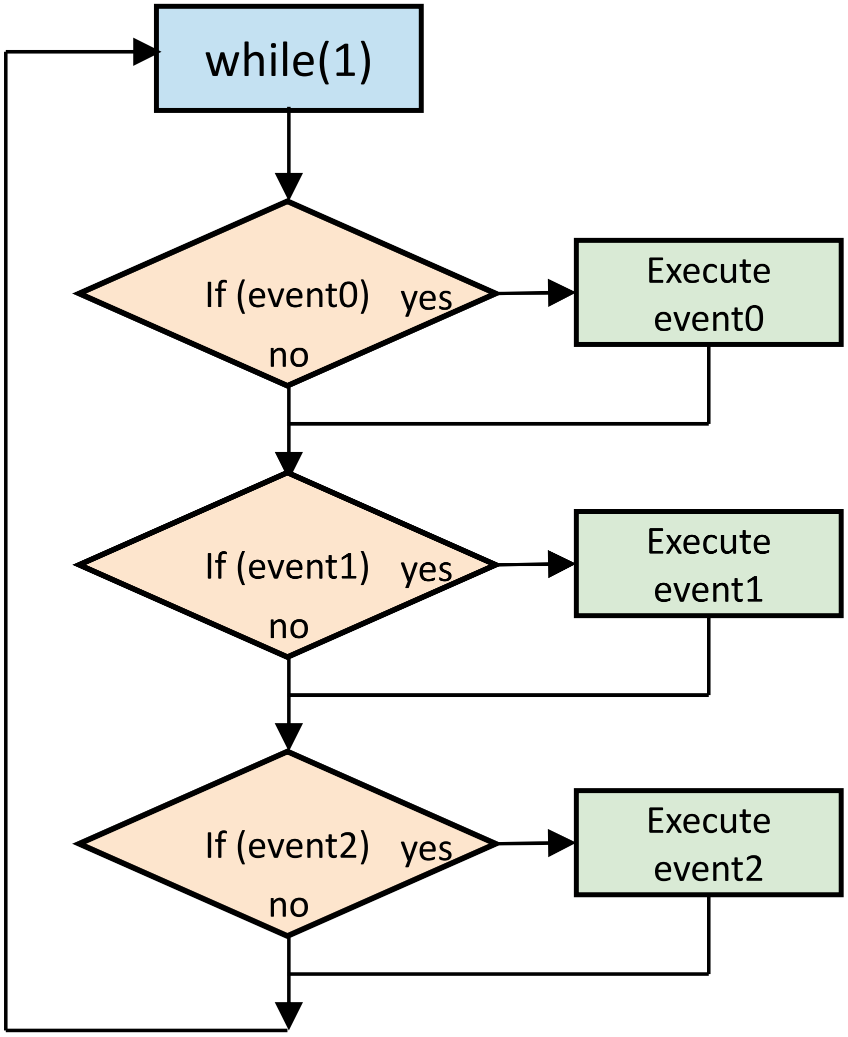


We have three events to service, any concerns?

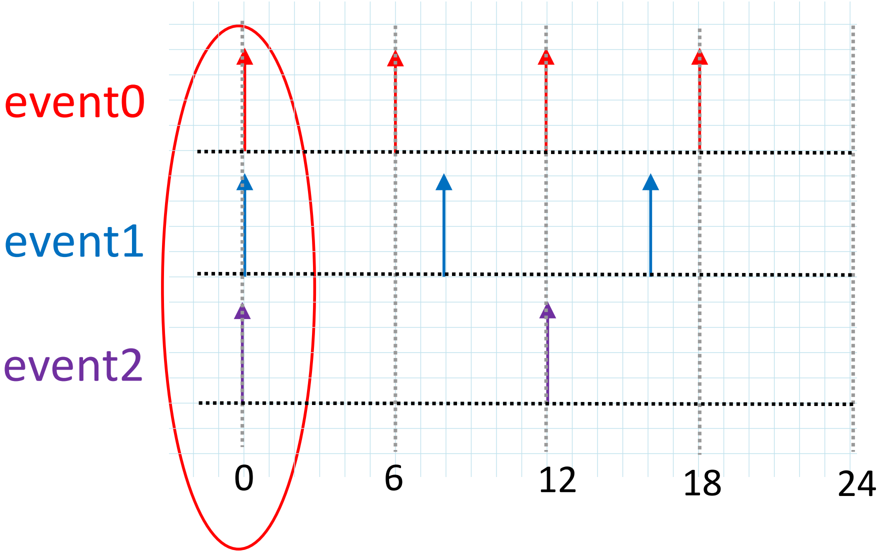
- How do we know if they make deadline?
- Are they preemption? Do we have it? We not have defined a scheduling algorithm.

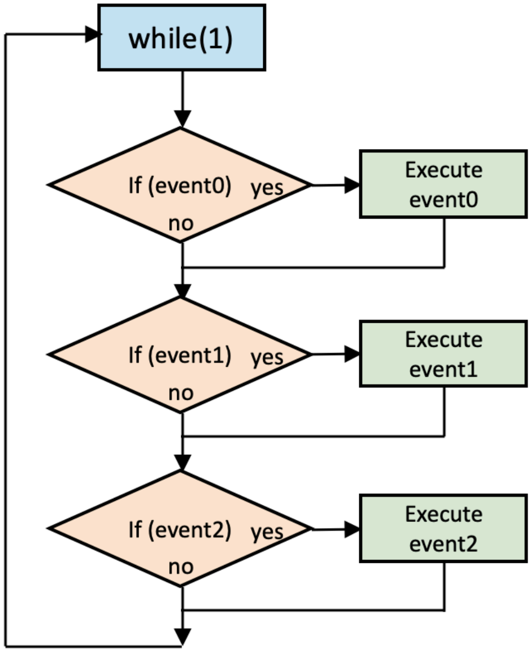
Polling Loops Without Preemption

- Simplest way to run is a polling loop.
  - No priority => no preemption.
  - Each event is checked in sequence.
    - Each event is guaranteed to run. (starvation-free)
- Round-Robin Scheduling
  - The status of each event is checked (polled).
  - If an event occurred, it executes to completion.
  - Then repeats process for the next event.
  - When all events have been checked returns to the beginning



Round Robin Polling

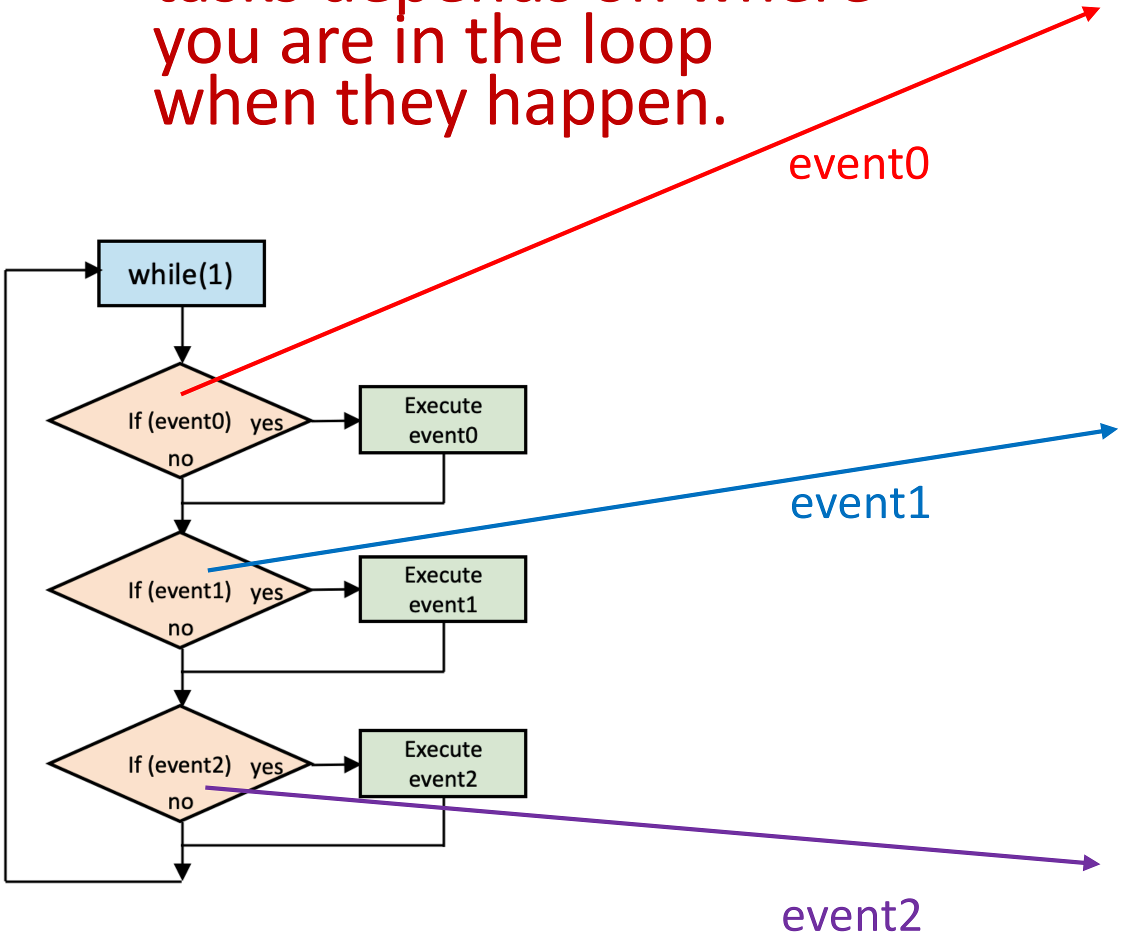


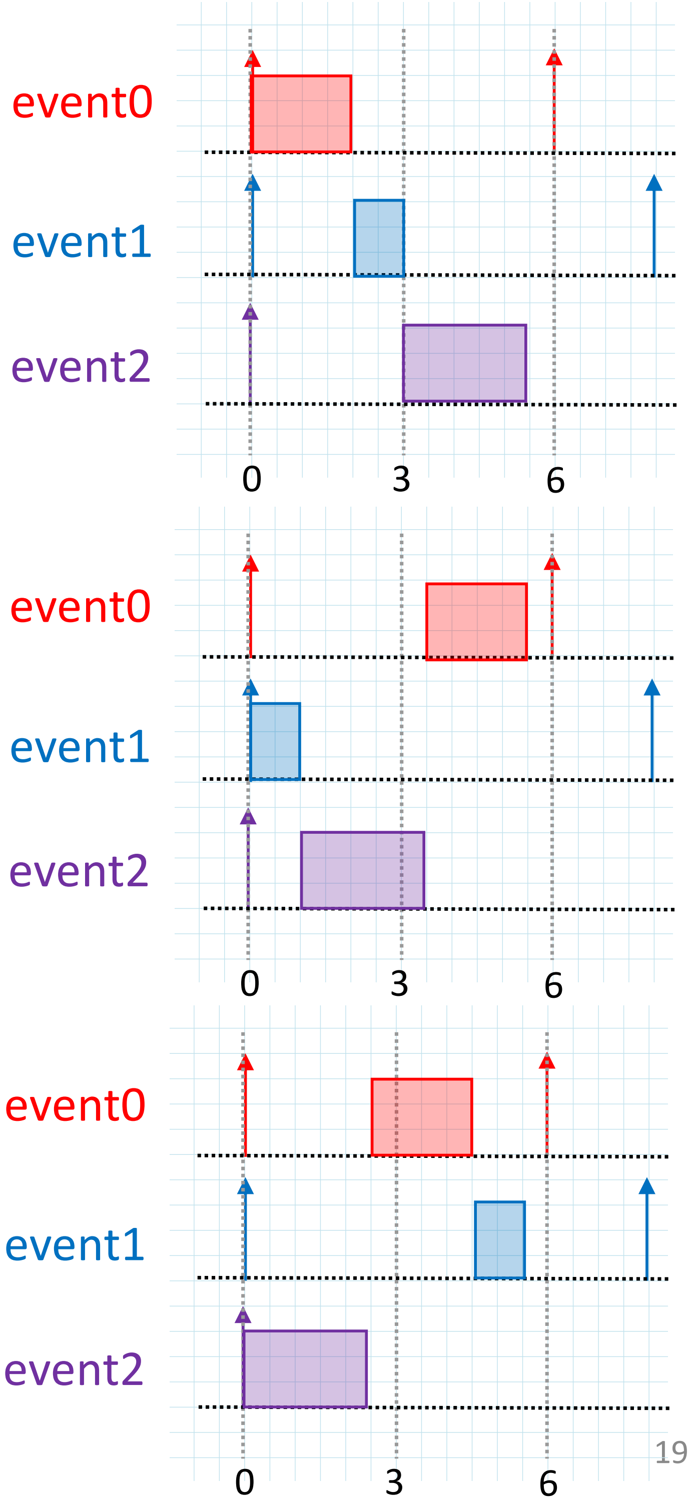


- For the example,
  - All three events occur at relative time 0.
  - Polling loop has been running for a while.
- Which task get serviced first?
  - All equality important no priority?
  - We naturally want to 0 cause it is first, but loop can be any where in its execution.
  - We don’t know.

Round Robin Scheduling

- The order of servicing tasks depends on where you are in the loop when they happen.





Are the tasks schedulable?

Event	Period	Execution Time
event0	6 ms	2 ms
event1	8 ms	1 ms
event2	12 ms	2.5 ms

Remember, we care only about maximum value for these calculations.

```
void main (void) {
  <init>;  // pseudo-code in <...>

  while (1) {
    if (<event0 occurred>) {
      <handle event0>;
    }
    if (<event1 occurred>) {
      <handle event1>;
    }
    if (<event2 occurred>) {
      <handle event2>;
    }
  }
}
```

t_{exec0} = 2 ms

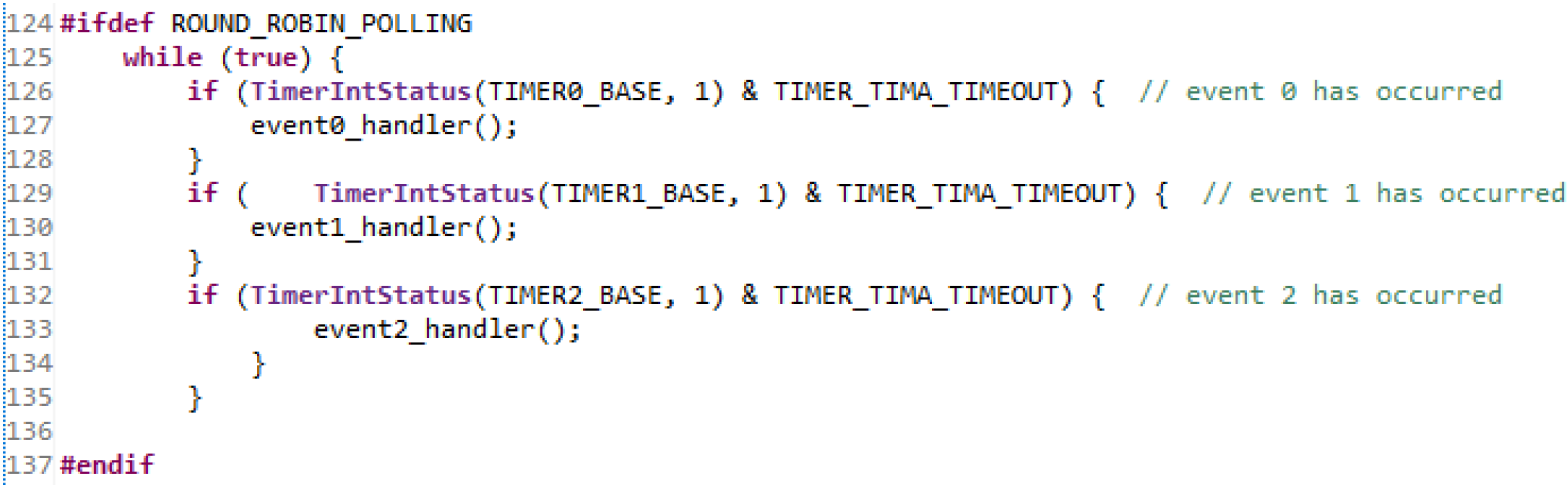
t_{exec1} = 1 ms

t_{exec2} = 2.5 ms

Event	Period	Execution Time	Latency	Response Time	Relative Deadline	Schedulable ?
event0	6 ms	2 ms	1 event 1 + event 2 = 1 + 2.5 =3.5	Latency+ execution max = 3.5 + 2m = 5.5m	5.5 m < 6m	YES
event1	8 ms	1 ms	2 + 2.5 = 4.5m	1 + 4.5 = 5.5	5.5 m < 8m	YES
event2	12 ms	2.5 ms	1 + 2 = 3	5.5	5.5 < 12 m	YES

ece3849_int_latency Round Robin Polling

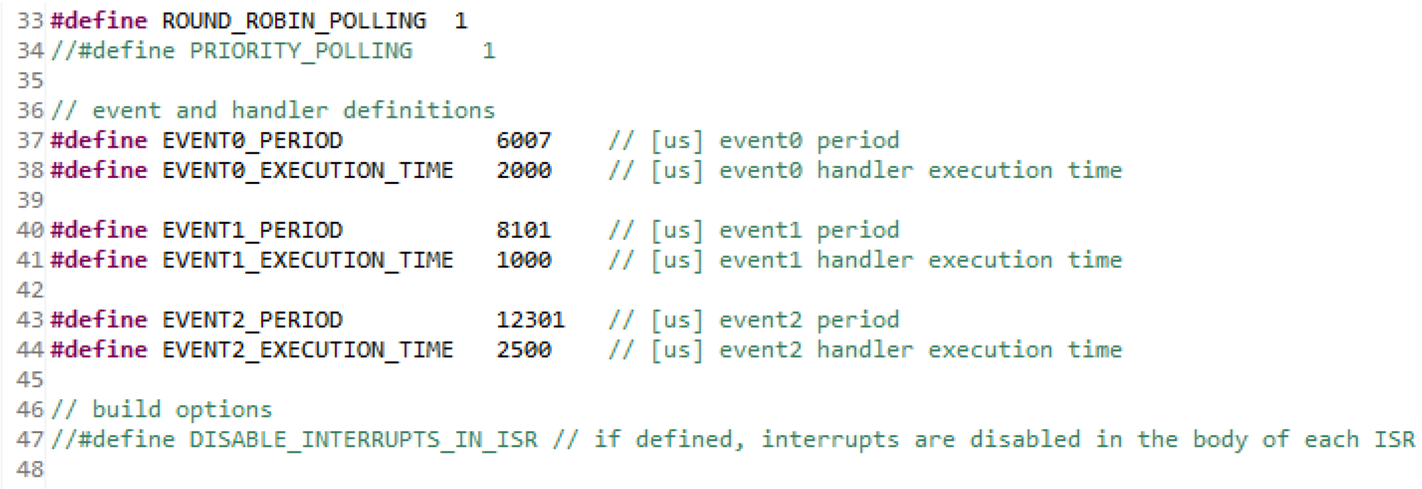
- In the Round Robin Polling option the infinite polling loop contains three if statements.
- The Interrupt status of each event is checked to see if an event occurred.
    - It is masked with the TimerA timeout interrupt bit, TIMER_TIMA_TIMEOUT.
        - This separates the event status from other unrelated interrupts.
    - If the Interrupt for that event occurred, the interrupt handler function is called.



```
#ifdef ROUND_ROBIN_POLLING
    while (true) {
        if (TimerIntStatus(TIMER0_BASE, 1) & TIMER_TIMA_TIMEOUT) {   // event 0 has occurred
            event0_handler();
        }
        if (    TimerIntStatus(TIMER1_BASE, 1) & TIMER_TIMA_TIMEOUT) {   // event 1 has occurred
            event1_handler();
        }
        if (TimerIntStatus(TIMER2_BASE, 1) & TIMER_TIMA_TIMEOUT) {   // event 2 has occurred
                event2_handler();
            }
        }

#endif
```

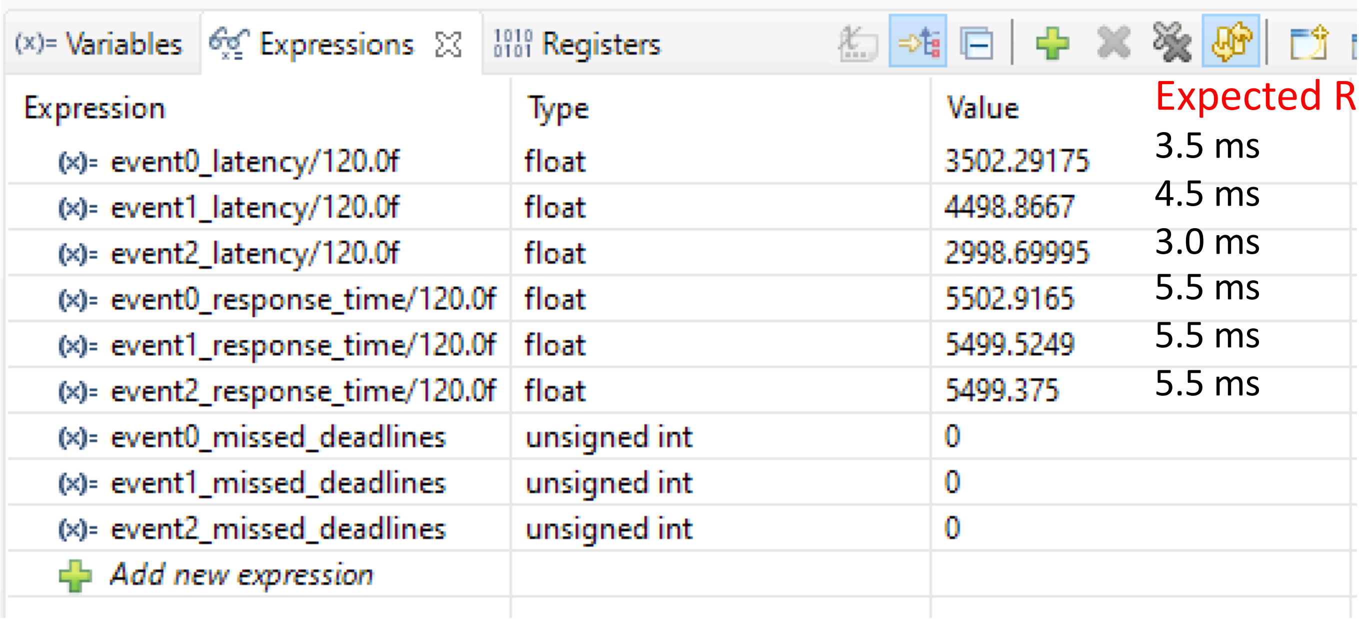
ece3849_int_latency
Example 1: event2 = 2.5

- Run settings



```
#define ROUND_ROBIN_POLLING  1
//#define PRIORITY_POLLING     1

// event and handler definitions
#define EVENT0_PERIOD           6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME   2000    // [us] event0 handler execution time

#define EVENT1_PERIOD           8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME   1000    // [us] event1 handler execution time

#define EVENT2_PERIOD           12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME   2500    // [us] event2 handler execution time

// build options
//#define DISABLE_INTERRUPTS_IN_ISR // if defined, interrupts are disabled in the body of each ISR

```

- Run results



Expression	Type	Value	Expected Results
(x)= event0_latency/120.0f	float	3502.29175	3.5 ms
(x)= event1_latency/120.0f	float	4498.8667	4.5 ms
(x)= event2_latency/120.0f	float	2998.69995	3.0 ms
(x)= event0_response_time/120.0f	float	5502.9165	5.5 ms
(x)= event1_response_time/120.0f	float	5499.5249	5.5 ms
(x)= event2_response_time/120.0f	float	5499.375	5.5 ms
(x)= event0_missed_deadlines	unsigned int	0	
(x)= event1_missed_deadlines	unsigned int	0	
(x)= event2_missed_deadlines	unsigned int	0	
Add new expression			

Example 2: Event 2 = 3.5 ms

- Lets try Round Robin Again

t_{exec0} = 2 ms

t_{exec1} = 1 ms

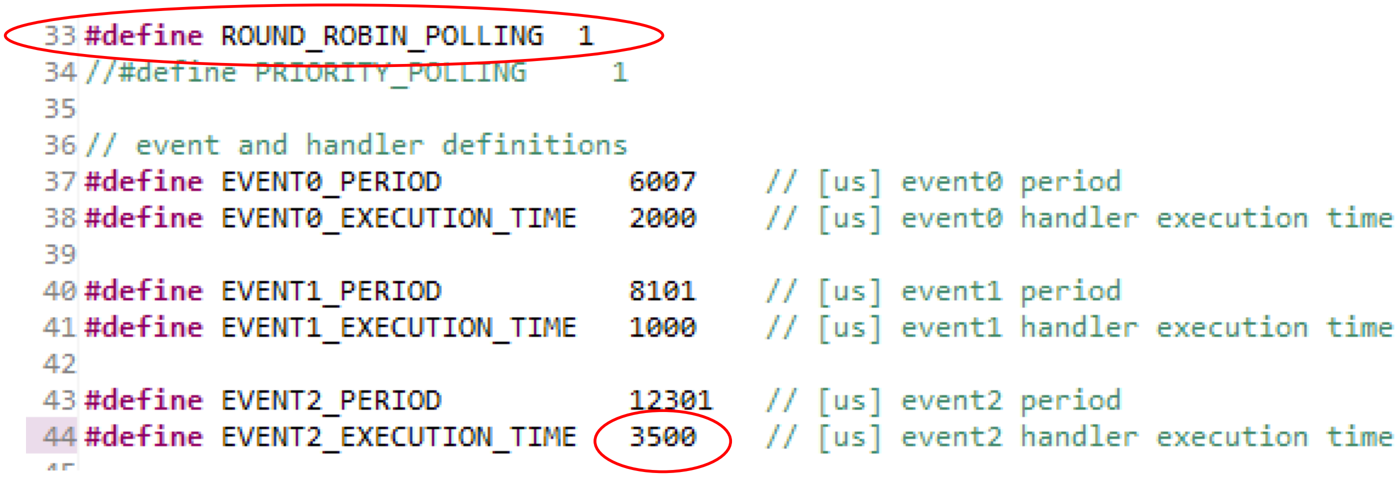
t_{exec2} = 3.5 ms

```
void main (void) {
  <init>;  // pseudo-code in <...>

  while (1) {
    if (<event0 occurred>) {
      <handle event0>;
    }
    if (<event1 occurred>) {
      <handle event1>;
    }
    if (<event2 occurred>) {
      <handle event2>;
    }
  }
}
```

Event	Period	Execution Time	Latency (sum of other events execution times)	Response Time (Latency + execution time)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	2 ms	1 + 3.5 m = 4.5	4.5 + 2 = 6.5	6.5 > 6	NO
event1	8 ms	1 ms	2 + 3.5 = 5.5	1 + 5.5 = 6.5	6.5 < 8	YES
event2	12 ms	3.5 ms	2+1 = 3	3.5 + 3 = 6.5	6.5 < 12	YES

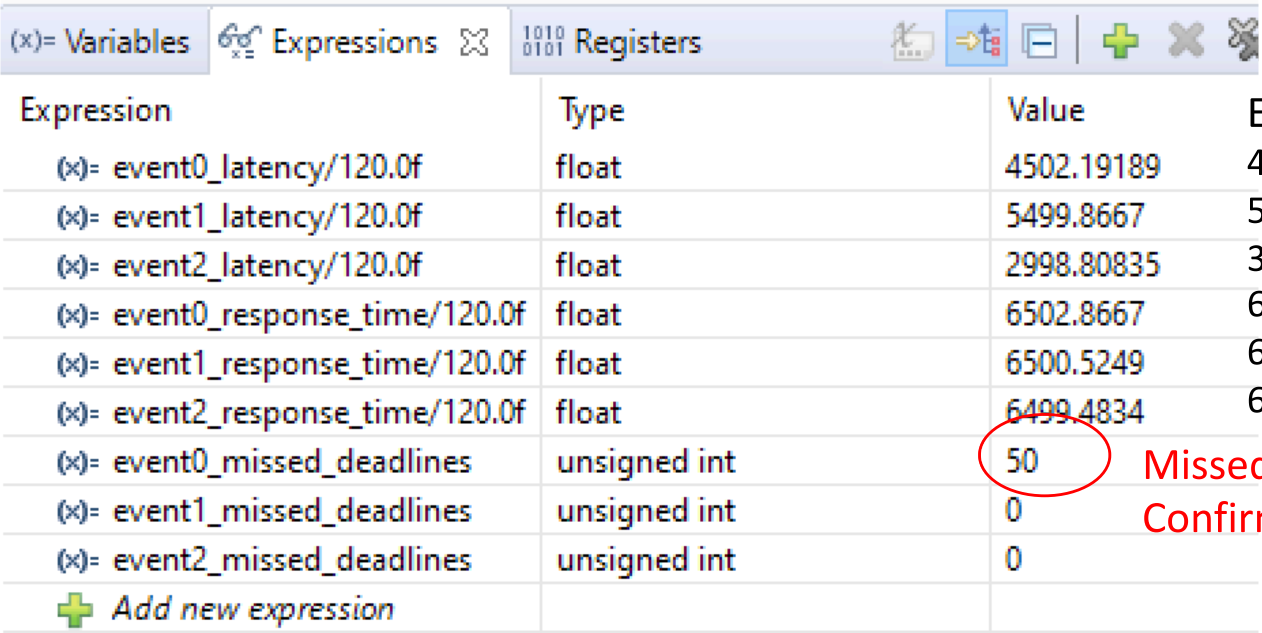
ece3849_int_latency
Example 2: event2 = 3.5 ms

- Run settings



```
#define ROUND_ROBIN_POLLING  1
//#define PRIORITY_POLLING     1

// event and handler definitions
#define EVENT0_PERIOD           6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME   2000    // [us] event0 handler execution time

#define EVENT1_PERIOD           8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME   1000    // [us] event1 handler execution time

#define EVENT2_PERIOD           12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME   3500    // [us] event2 handler execution time
```

- Run results



Expression	Type	Value
(x)= event0_latency/120.0f	float	4502.19189
(x)= event1_latency/120.0f	float	5499.8667
(x)= event2_latency/120.0f	float	2998.80835
(x)= event0_response_time/120.0f	float	6502.8667
(x)= event1_response_time/120.0f	float	6500.5249
(x)= event2_response_time/120.0f	float	6499.4834
(x)= event0_missed_deadlines	unsigned int	50
(x)= event1_missed_deadlines	unsigned int	0
(x)= event2_missed_deadlines	unsigned int	0
Add new expression		

Expected Results
4.5 ms
5.5 ms
3.0 ms
6.5 ms
6.5 ms
6.5 ms

Missed Deadlines Confirmed!

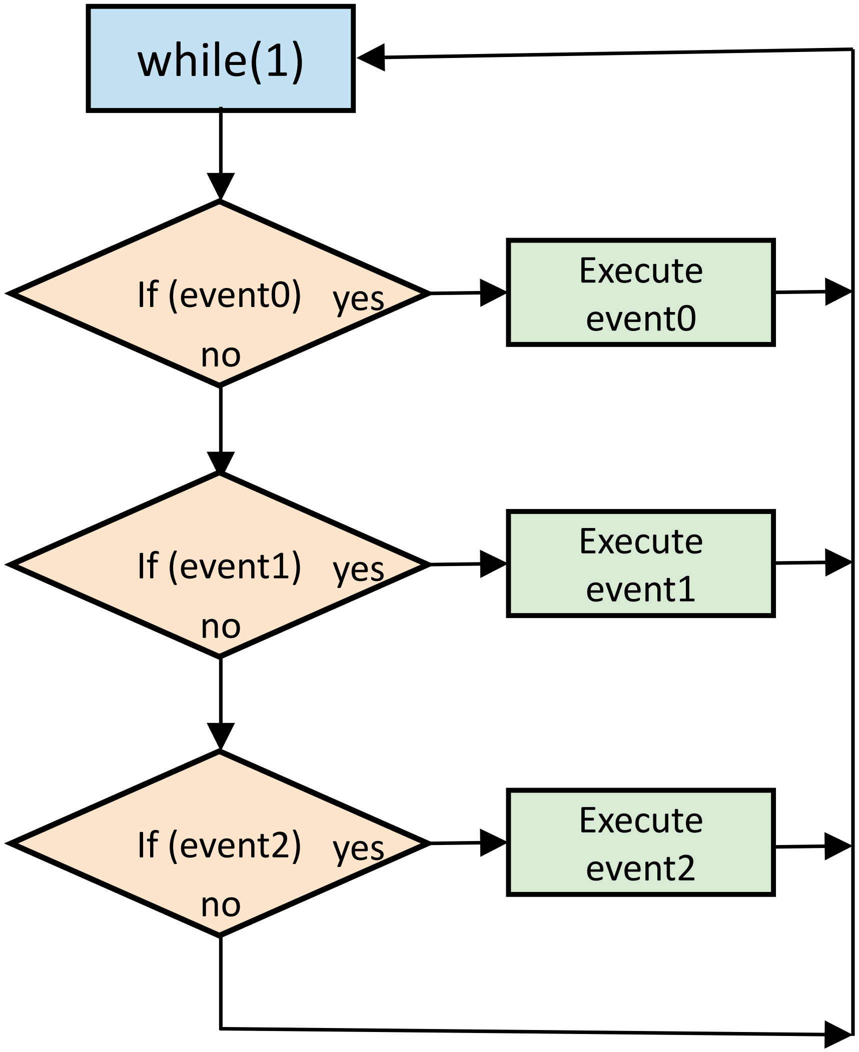
Example 2, event2 = 3.5 ms Round Robin is not schedulable

- The execution time of all the events factor into the response time of ALL the events.

- What is the bottleneck in this case?
  - Short event has the most problems?
- How might we schedule differently?
  - Recheck our requirements and see if we could the period long?
  - Different polling – prioritize shortest first.

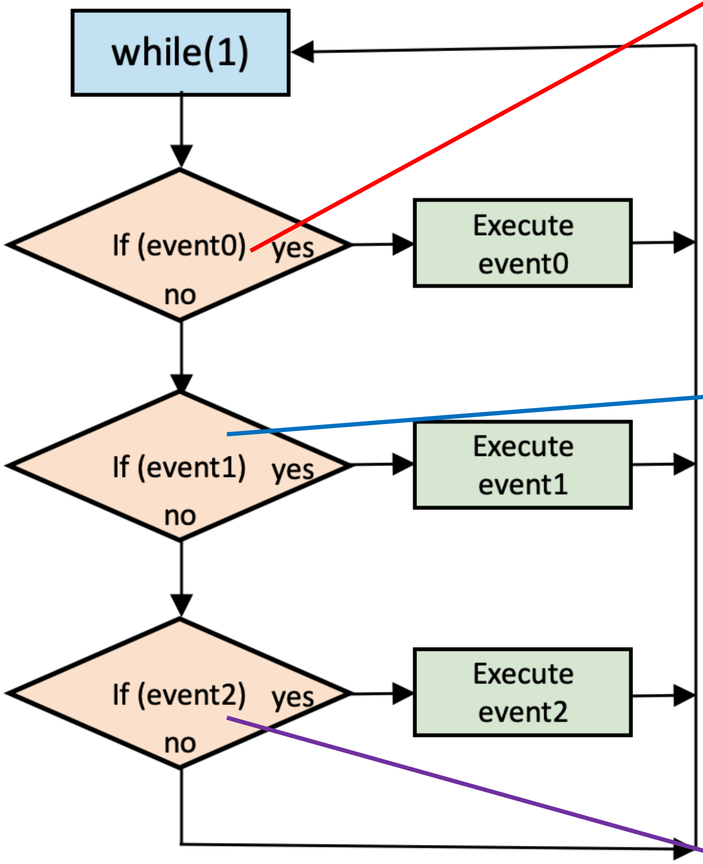
Priority Polling: Shortest deadline first

- event0 will always be serviced.
- Only if event0 is not waiting will event 1 be serviced.
- Only if event0 and event1 are not waiting will event 2 be serviced.

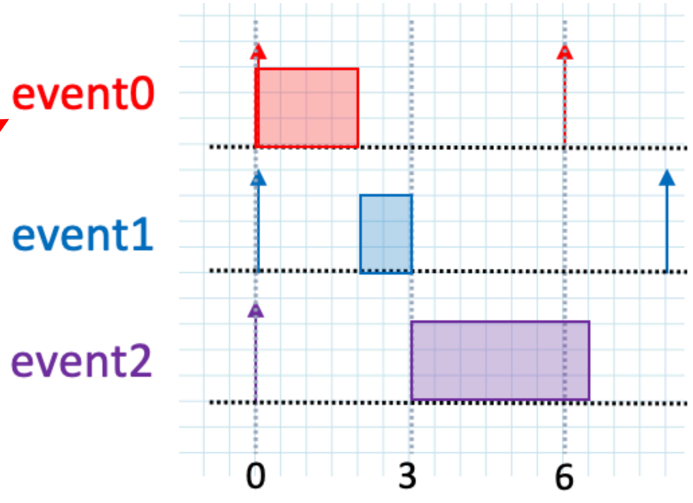


- What is likely to happen if event0 takes most of the CPU time?
  - Possibly 1 or 2 is starved?

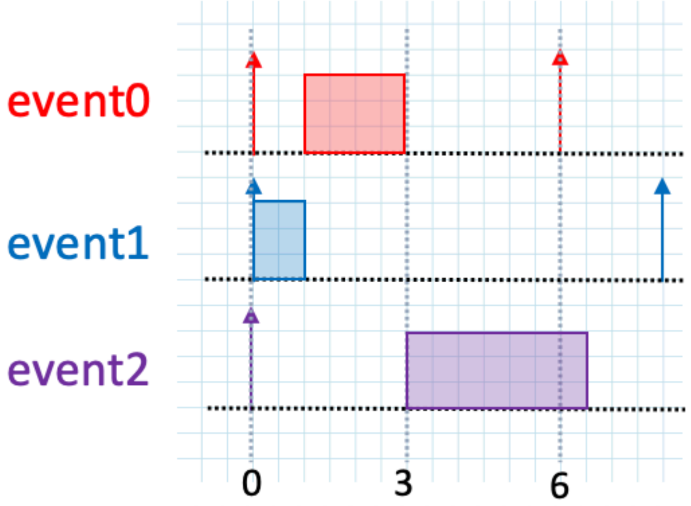
Priority Polling: event0 latency



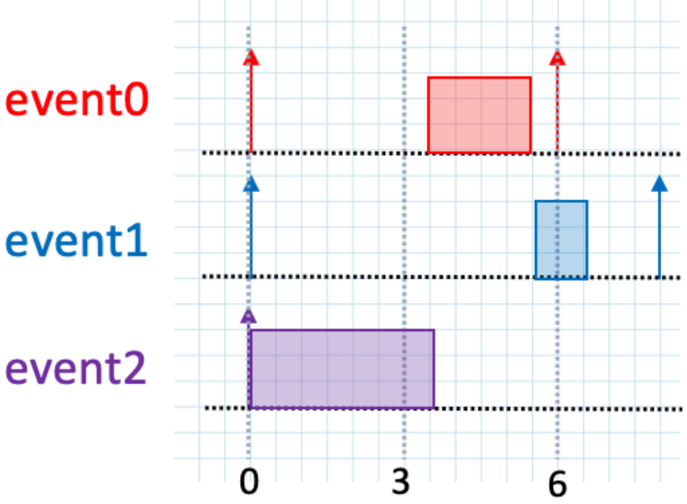
event0 latency = 0



event0 Latency = event1 execution time



event0 latency = Event2 execution time



- Max event0 latency = max(event1 execution time , event2 execution time)

Priority Polling

```
void main (void) {
  <init>;

  while (1) {
    if (<event0 occurred>) {
      <handle event0>;
    }
    else if (<event1 occurred>) {
      <handle event1>;
    }
    else if (<event2 occurred>) {
      <handle event2>;
    }
  }
}
```

t_{exec0} = 2 ms

t_{exec1} = 1 ms

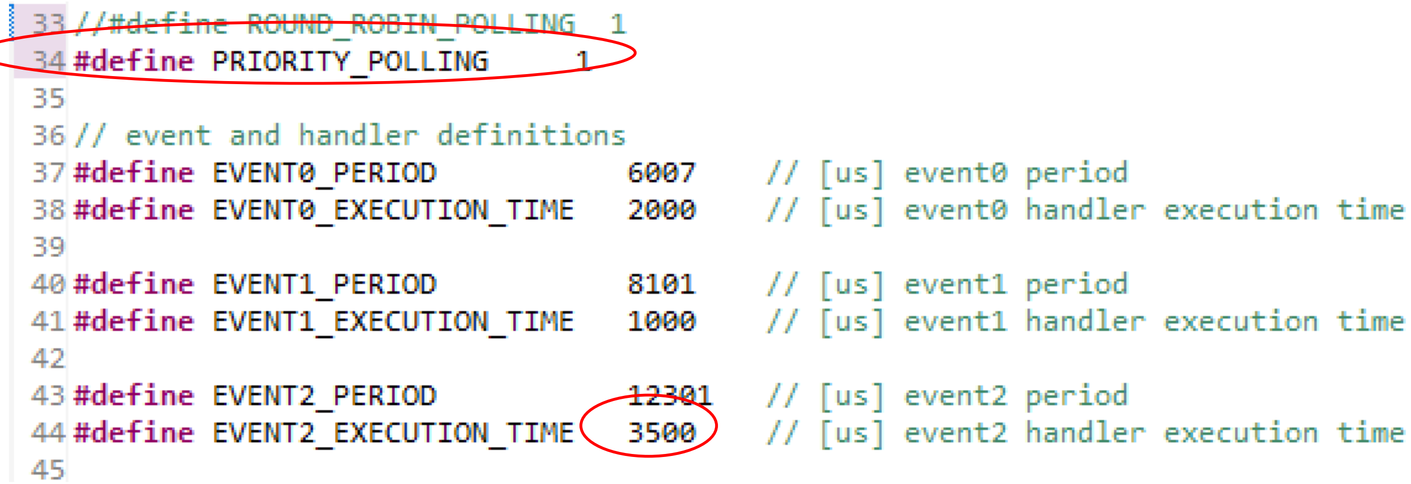
t_{exec2} = 3.5 ms

Event	Period	Execution Time	Latency	Response Time (Latency + execution time)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	2 ms	3.5 m (max of 1 OR 3.5)	5.5	5.5 < 6	YES
event1	8 ms	1 ms				
event2	12 ms	3.5 ms				

- What will happen if event0 period gets shorter?
  - ?

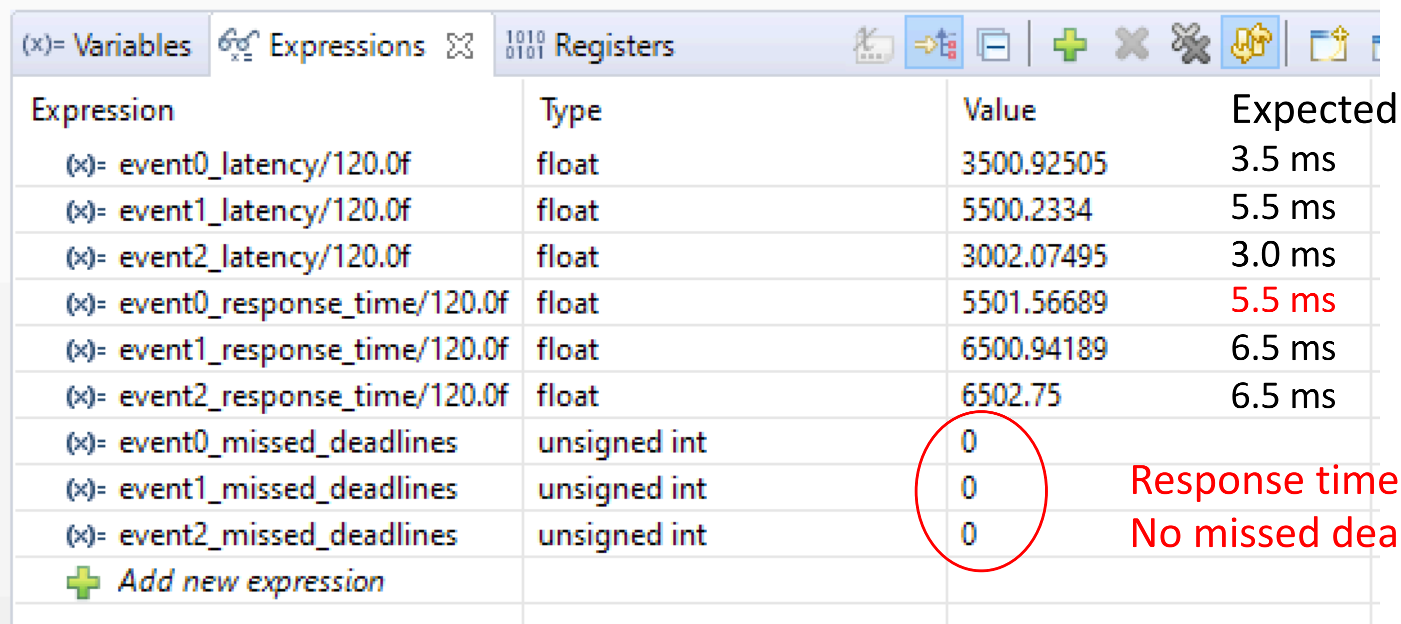
ece3849_int_latency
Example 2: event2 = 3.5, Priority Polling

- Run settings



```
//#define ROUND_ROBIN_POLLING  1
#define PRIORITY_POLLING        1

// event and handler definitions
#define EVENT0_PERIOD           6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME   2000    // [us] event0 handler execution time

#define EVENT1_PERIOD           8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME   1000    // [us] event1 handler execution time

#define EVENT2_PERIOD           12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME   3500    // [us] event2 handler execution time

```

- Run results

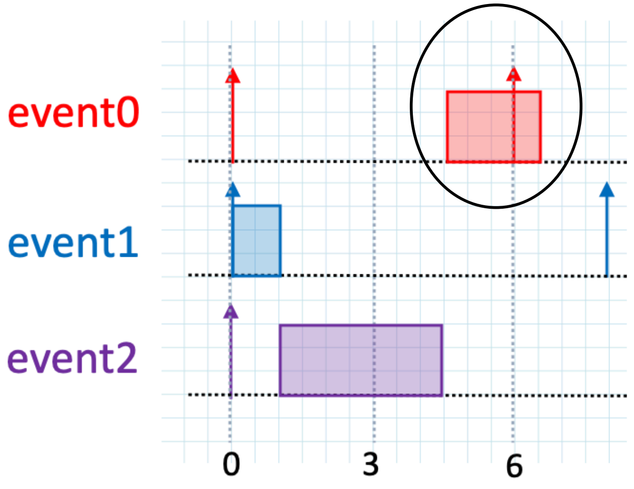


Expression	Type	Value	Expected Results
(x)= event0_latency/120.0f	float	3500.92505	3.5 ms
(x)= event1_latency/120.0f	float	5500.2334	5.5 ms
(x)= event2_latency/120.0f	float	3002.07495	3.0 ms
(x)= event0_response_time/120.0f	float	5501.56689	5.5 ms
(x)= event1_response_time/120.0f	float	6500.94189	6.5 ms
(x)= event2_response_time/120.0f	float	6502.75	6.5 ms
(x)= event0_missed_deadlines	unsigned int	0	
(x)= event1_missed_deadlines	unsigned int	0	
(x)= event2_missed_deadlines	unsigned int	0	
Add new expression			

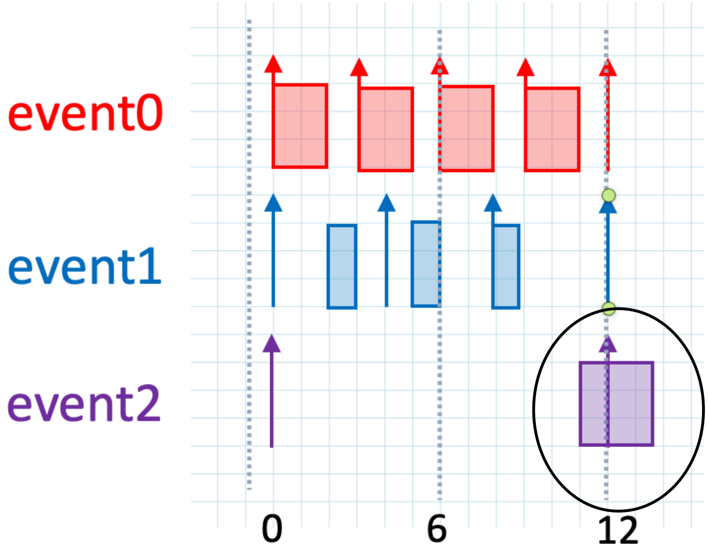
Response time lower
No missed deadlines!

Round Robin vs. Priority Polling

- Round Robin Scheduling
  - Pro: Will guarantee each event gets processed.
  - Con: Response time is limited by the sum of the execution times of all events.
    - May cause events with short deadlines to miss their deadline.



- Prioritize Short Deadline First
  - Pro: Short dead lines will be met.
  - Con: Starvation
    - Events with long deadlines may never get serviced if higher priority tasks occur at a fast enough rate.
  - Figuring out maximum latency for low priority tasks complicated.



Example 3: Event 2 = 6 msec

- Lets try Priority Polling again

t_{exec0} = 2 ms

t_{exec1} = 1 ms

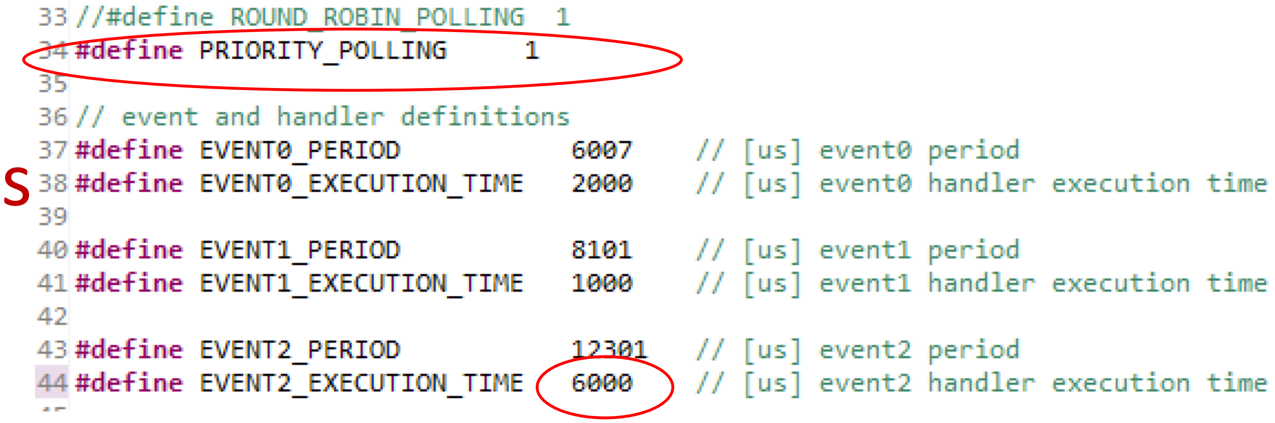
t_{exec2} = 6.0 ms

```
void main (void) {
  <init>;

  while (1) {
    if (<event0 occurred>) {
      <handle event0>;
    }
    else if (<event1 occurred>) {
      <handle event1>;
    }
    else if (<event2 occurred>) {
      <handle event2>;
    }
  }
}
```

Event	Period	Execution Time	Latency	Response Time (Latency + execution time)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	2 ms				
event1	8 ms	1 ms				
event2	12 ms	6 ms				

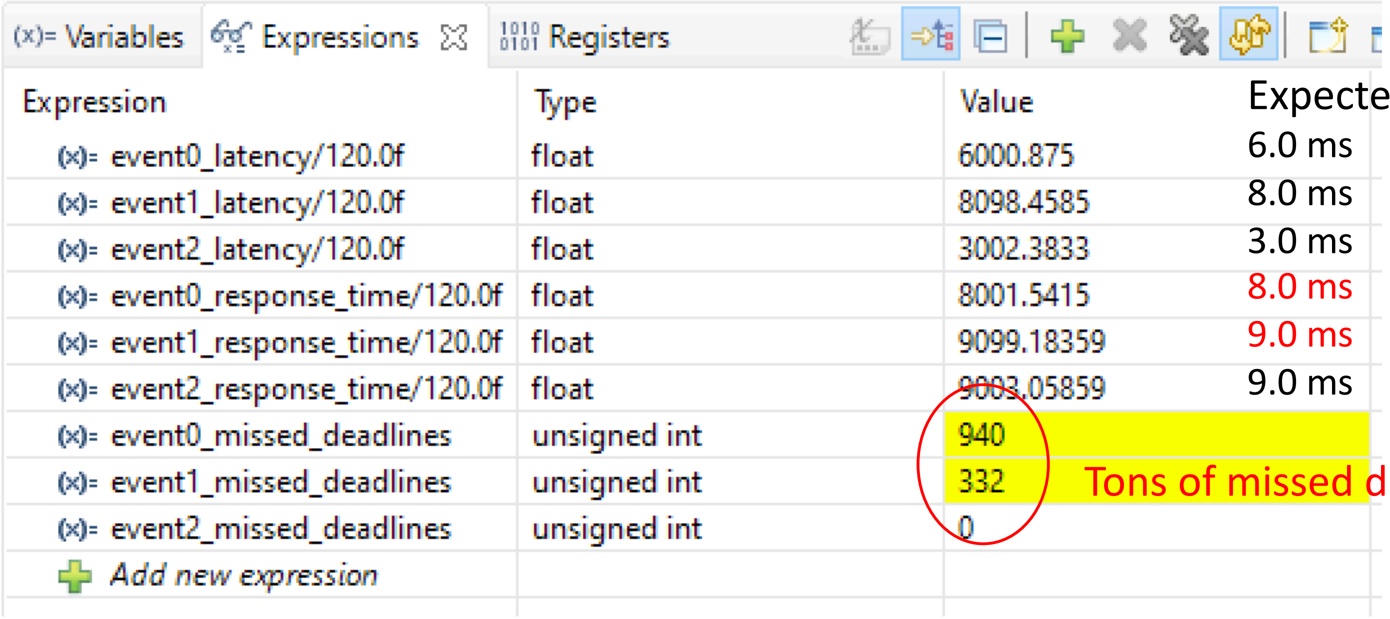
ece3849_int_latency
Example 3: event2 = 6.0, Priority Polling

- Run settings



```
//#define ROUND_ROBIN_POLLING  1
#define PRIORITY_POLLING     1

// event and handler definitions
#define EVENT0_PERIOD           6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME   2000    // [us] event0 handler execution time

#define EVENT1_PERIOD           8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME   1000    // [us] event1 handler execution time

#define EVENT2_PERIOD           12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME   6000    // [us] event2 handler execution time
```

- Run results

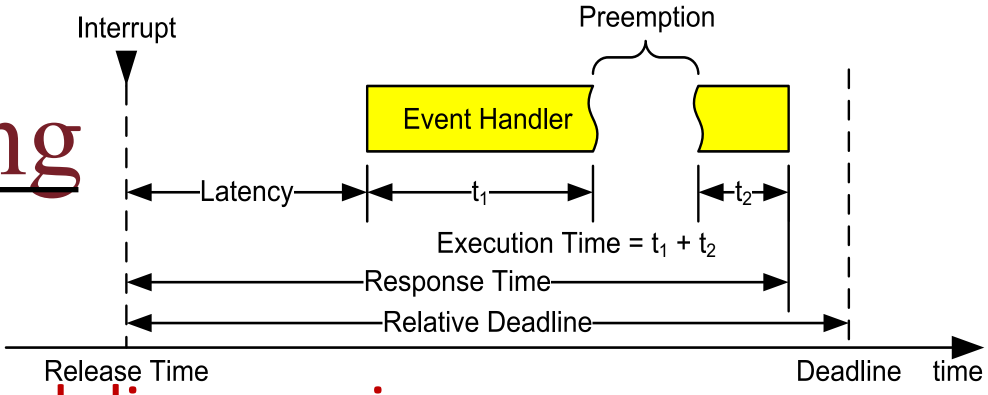


Expression	Type	Value	Expected Results
(x)= event0_latency/120.0f	float	6000.875	6.0 ms
(x)= event1_latency/120.0f	float	8098.4585	8.0 ms
(x)= event2_latency/120.0f	float	3002.3833	3.0 ms
(x)= event0_response_time/120.0f	float	8001.5415	8.0 ms
(x)= event1_response_time/120.0f	float	9099.18359	9.0 ms
(x)= event2_response_time/120.0f	float	9003.05859	9.0 ms
(x)= event0_missed_deadlines	unsigned int	940	
(x)= event1_missed_deadlines	unsigned int	332	
(x)= event2_missed_deadlines	unsigned int	0	
Add new expression			

Tons of missed deadlines!

- What strategies can we try next to meet all the deadlines?
  - ?

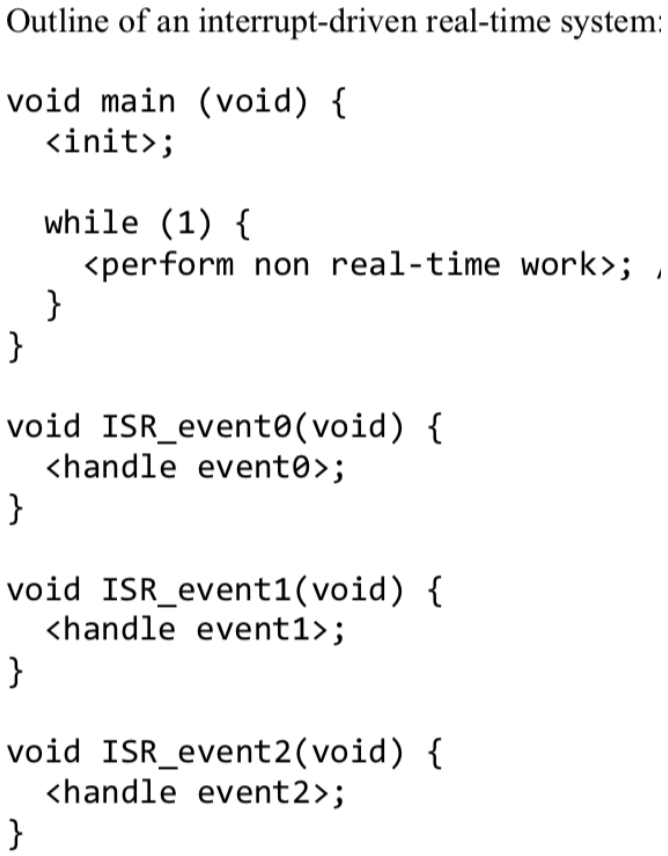
Preemptive scheduling



- The simplest preemptive scheduling requires an interrupt controller.
  - The Cortex-M4 is designed for real time systems and supports an interrupt-driven strategy.
- The infinite while loop runs non-critical tasks in the foreground.
- Interrupt controller automatically calls an Interrupt Service routine (ISR) as soon as an event occurs.
  - The foreground tasks are preempted when ISRs are called.
  - ISR’s are said to run in the background.
  - Use of foreground and background is arbitrary and sometimes swapped.
- Higher priority ISRs can interrupt / preempt lower priority ones.

Interrupt-Driven Design

- Beginning of main
  - Initialize Interrupts and hardware.
  - While loop (in foreground)
    - Performs low priority non-interrupt driven tasks.
    - Or it just loops if nothing to do.
  - ISR_event0 – highest priority ISR
  - ISR_event1 – Mid priority ISR
  - ISR_event2 – Lowest priority ISR



Outline of an interrupt-driven real-time system:

```
void main (void) {
  <init>;

  while (1) {
    <perform non real-time work>;
  }
}

void ISR_event0(void) {
  <handle event0>;
}

void ISR_event1(void) {
  <handle event1>;
}

void ISR_event2(void) {
  <handle event2>;
}
```

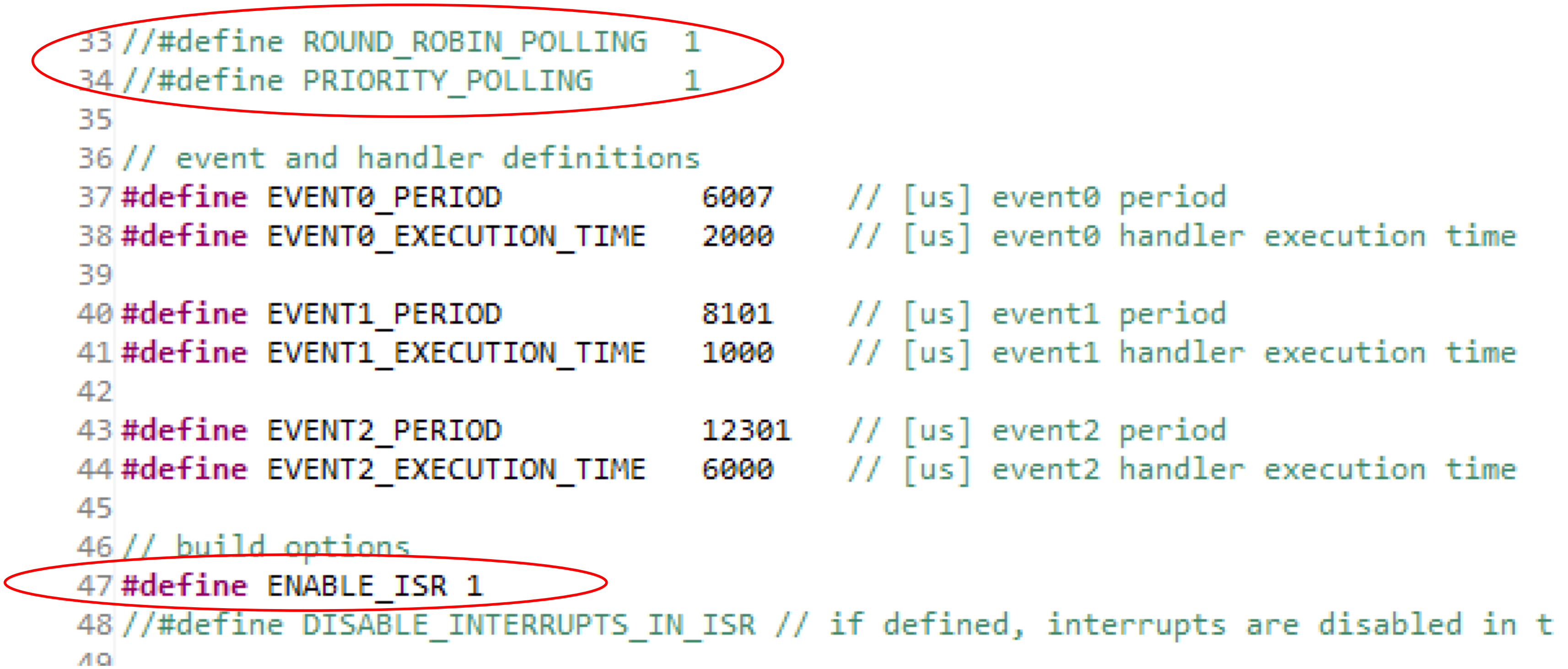
- Interrupt controller automatically calls ISR routines to operate in the background

Prioritization of Interrupts

- Higher priority ISR can preempt / interrupt lower priority ISRs.
- How should we prioritize the interrupts?
  - Shorter periods should have higher priority.
  - Smaller deadlines
- Once setup it is possible to turn on and off preemption.
  - When preemption is off, the system behaves like a priority polling design.
  - But we need to respond more quickly than this.

ece3849_int_latency: Enable Interrupts

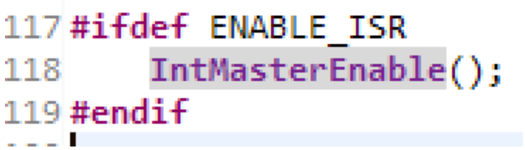
- Comment out polling option and enable ISR.



```
//#define ROUND_ROBIN_POLLING  1
//#define PRIORITY_POLLING     1

// event and handler definitions
#define EVENT0_PERIOD           6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME   2000    // [us] event0 handler execution time

#define EVENT1_PERIOD           8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME   1000    // [us] event1 handler execution time

#define EVENT2_PERIOD           12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME   6000    // [us] event2 handler execution time

// build options
#define ENABLE_ISR 1
//#define DISABLE_INTERRUPTS_IN_ISR // if defined, interrupts are disabled in t
```

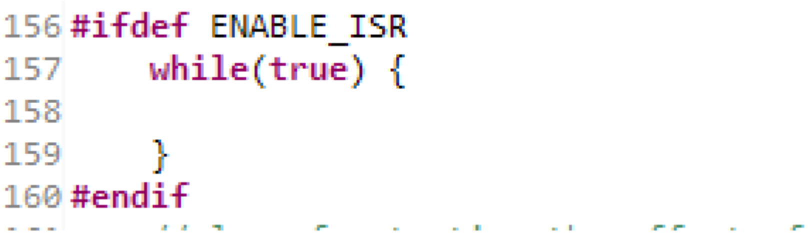
ece3849_int_latency: Enable Interrupts

- ENABLE_ISR allows the IntMasterEnable function to be called.



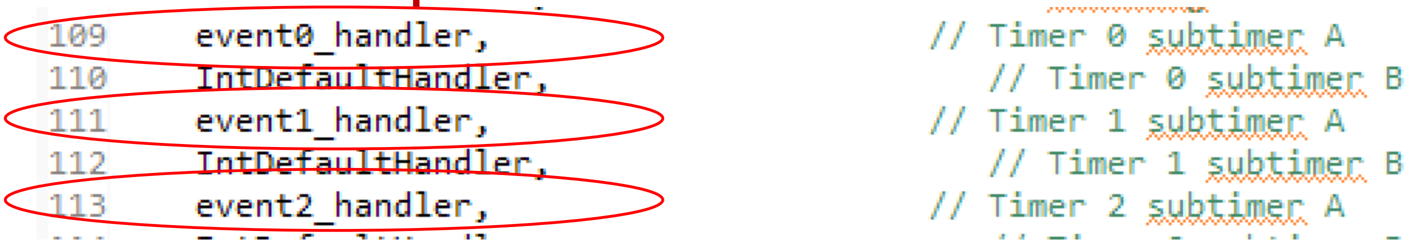
```
#ifdef ENABLE_ISR
    IntMasterEnable();
#endif
```

- Creates a while loop with nothing in it.



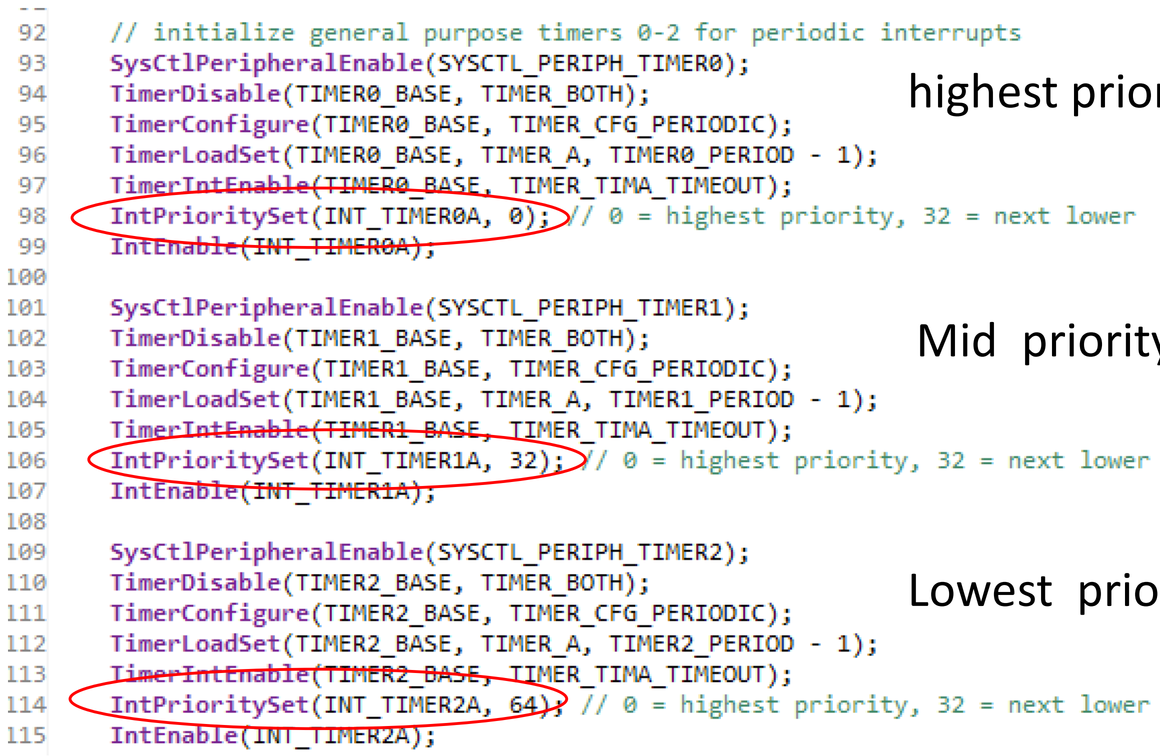
```
#ifdef ENABLE_ISR
    while(true) {

    }
#endif
```

- In the vector table in tm4c1294ncpdt_startup_css.c the event handler functions are assigned to the Timer A interrupts



```
    event0_handler,                     // Timer 0 subtimer A
    IntDefaultHandler,                      // Timer 0 subtimer B
    event1_handler,                     // Timer 1 subtimer A
    IntDefaultHandler,                      // Timer 1 subtimer B
    event2_handler,                     // Timer 2 subtimer A
```

ece3849_int_latency: Setting Priority



```
    // initialize general purpose timers 0-2 for periodic interrupts
    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, 32 = next lower
    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, 32 = next lower
    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, 32 = next lower
    IntEnable(INT_TIMER2A);
```

highest priority = lowest number = 0

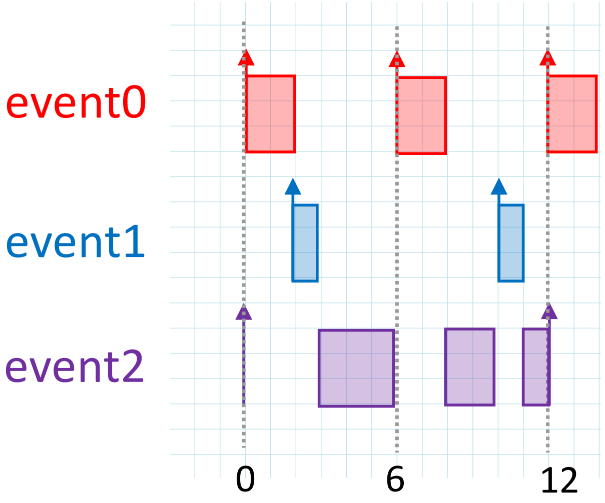
Mid priority =32

Lowest priority = highest number = 64

- The priority value register set by IntPrioritySet function is an 8-bit value.
- However, only the 3 most significant bits of the value are used for interrupt priority.
  - Valid values are 0, 32, 64, 96, 128, 160, 192, 224 (multiples of 32)

Example 3: Event 2 = 6 msec

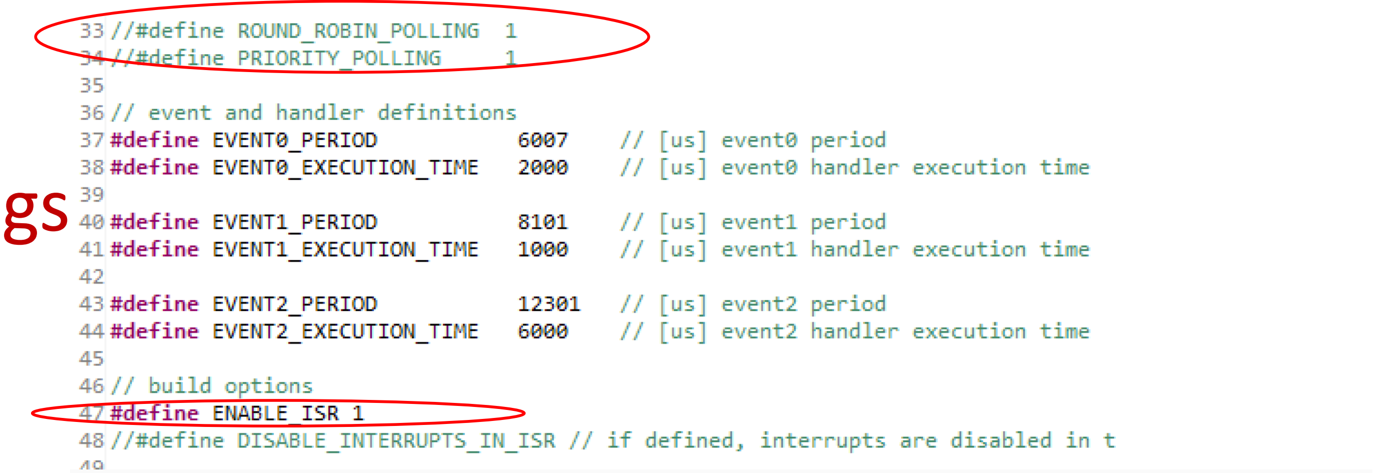
- Preemptive Interrupt Driven Design
  - Interrupt controller calls ISR immediately on event0.
    - There will be a small delay to call ISR.
  - Higher priority events interrupt/preempt lower priority events.
    - Higher priority events may interrupt multiple times.
    - Events do not have fixed phase and can happen anytime relative to each other.



Event	Period	Execution Time	Latency	Response Time (Latency + ?)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	2 ms				
event1	8 ms	1 ms				
event2	12 ms	6 ms				

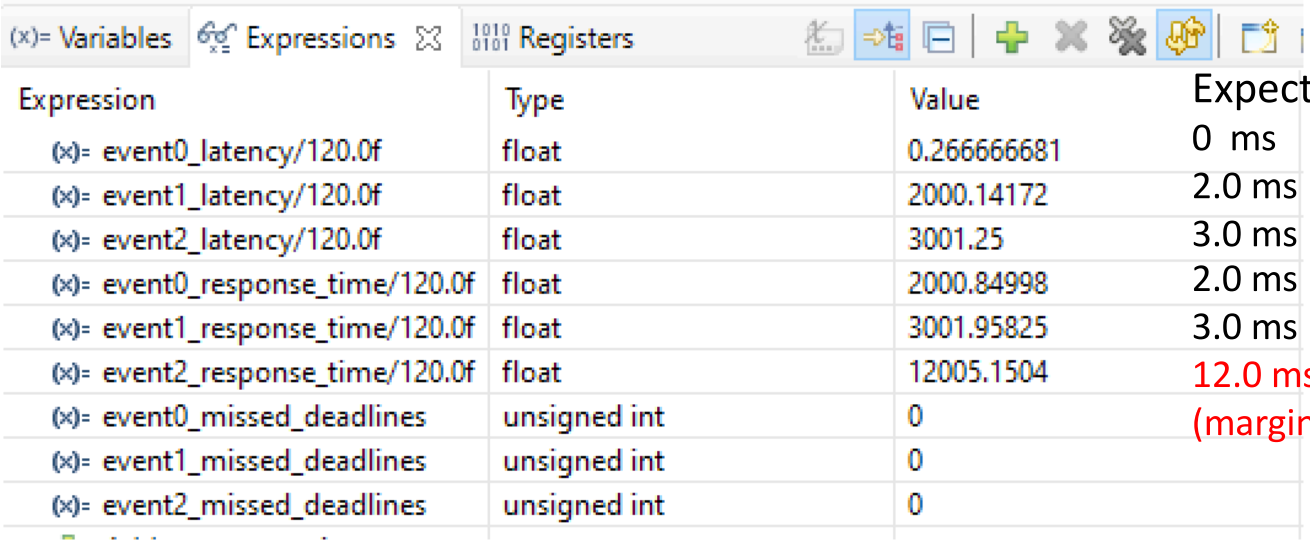
ece3849_int_latency
Example 3: event2 = 6.0, Interrupt Driven

- Run settings



```
//#define ROUND_ROBIN_POLLING  1
//#define PRIORITY_POLLING     1

// event and handler definitions
#define EVENT0_PERIOD          6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME  2000    // [us] event0 handler execution time

#define EVENT1_PERIOD          8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME  1000    // [us] event1 handler execution time

#define EVENT2_PERIOD          12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME  6000    // [us] event2 handler execution time

// build options
#define ENABLE_ISR 1
//#define DISABLE_INTERRUPTS_IN_ISR // if defined, interrupts are disabled in t
```

- Run results



Expression	Type	Value	Expected Results
event0_latency/120.0f	float	0.266666681	0 ms
event1_latency/120.0f	float	2000.14172	2.0 ms
event2_latency/120.0f	float	3001.25	3.0 ms
event0_response_time/120.0f	float	2000.84998	2.0 ms
event1_response_time/120.0f	float	3001.95825	3.0 ms
event2_response_time/120.0f	float	12005.1504	12.0 ms (marginal)
event0_missed_deadlines	unsigned int	0	
event1_missed_deadlines	unsigned int	0	
event2_missed_deadlines	unsigned int	0	

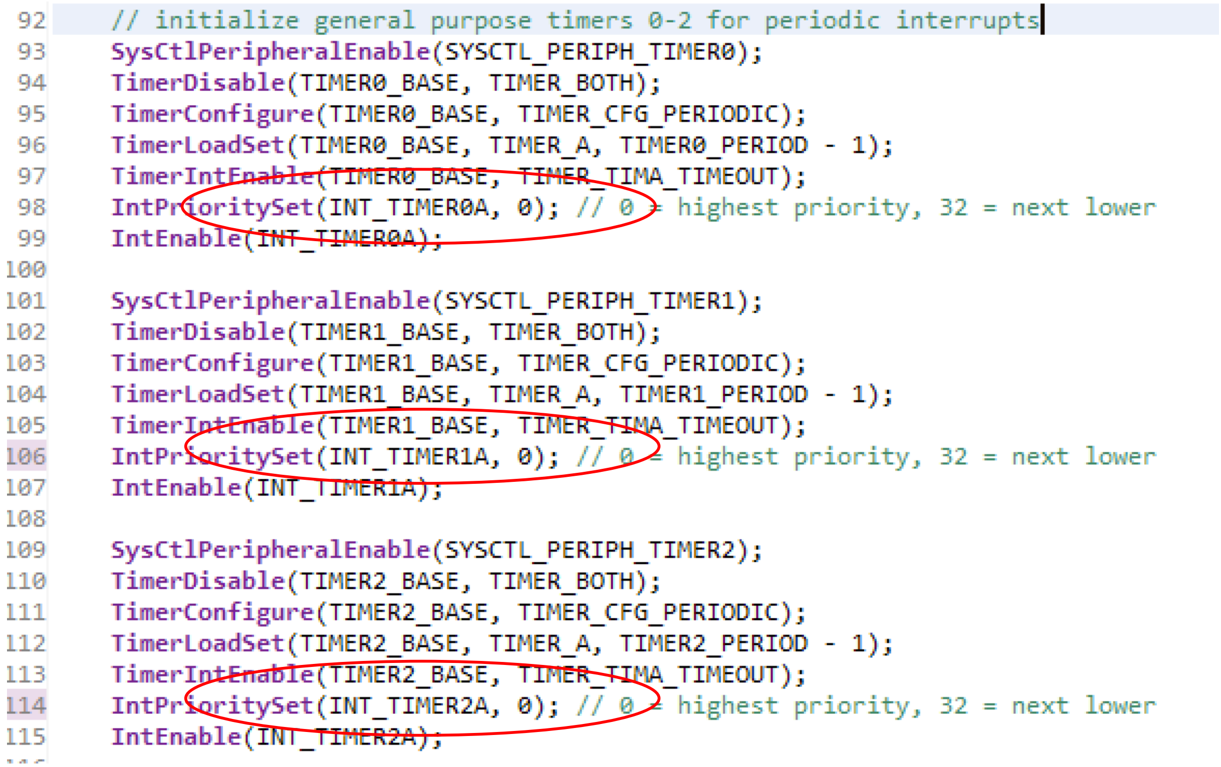
- Why is the event0 latency not 0?

Latency of Highest Interrupt

- If ISR are enabled, the latency of the highest priority interrupt will be small but not zero.
  - The current operation needs to complete.
  - The context of the CPU state is saved.
  - The first instruction of the ISR is fetched.

- Also in this the ece3849_int_latency program several operations were executed before the latency was calculated.
  - The time was read to calculate the latency.
  - This would normally not be considered part of latency.

ece3849_int_latency
Example 3: event2 = 6.0, Interrupt Driven, all the same priority

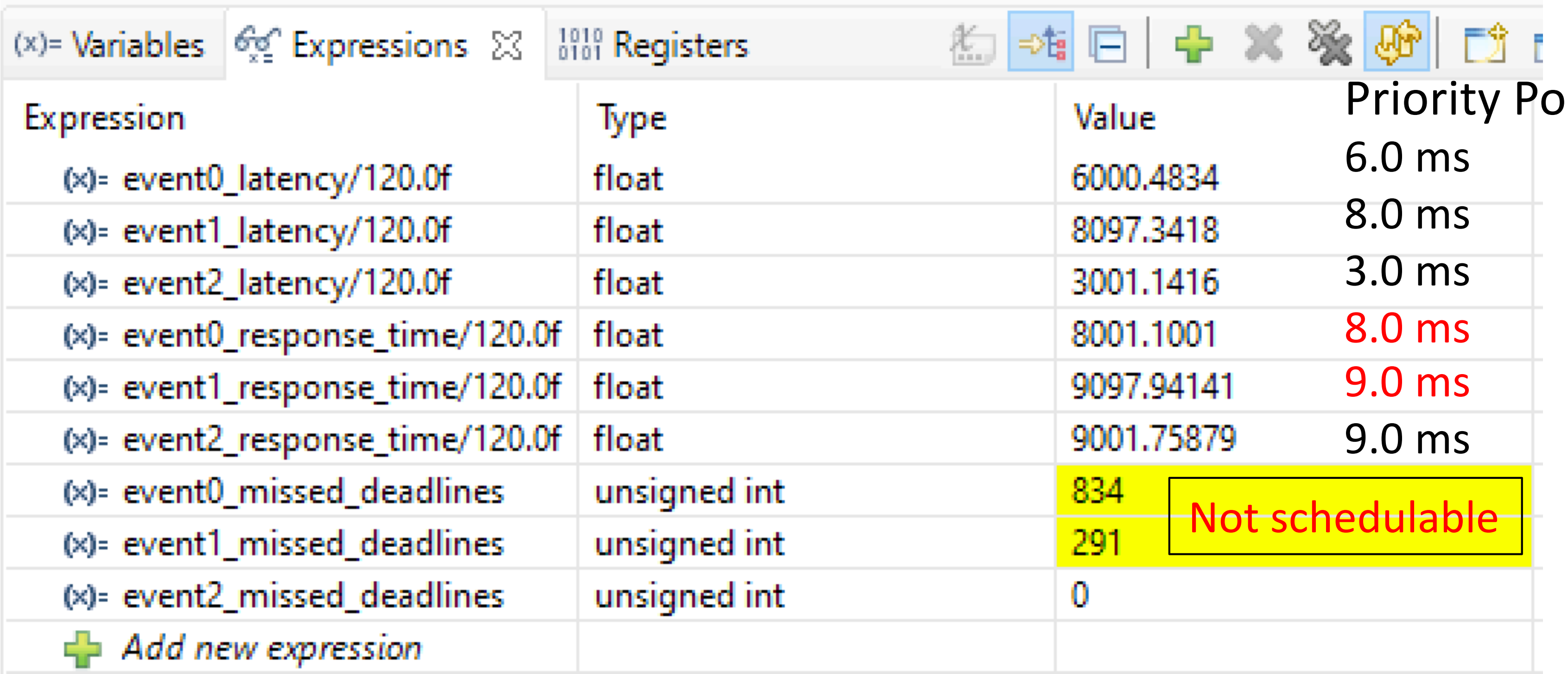


```
    // initialize general purpose timers 0-2 for periodic interrupts
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerDisable(TIMER0_BASE, TIMER_BOTH);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
    TimerLoadSet(TIMER0_BASE, TIMER_A, TIMER0_PERIOD - 1);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    IntPrioritySet(INT_TIMER0A, 0); // 0 = highest priority, 32 = next lower
    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, 0); // 0 = highest priority, 32 = next lower
    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, 0); // 0 = highest priority, 32 = next lower
    IntEnable(INT_TIMER2A);
```

- What happens if we make them all the same priority?
  - ?

ece3849_int_latency Example 3: event2 = 6.0, Interrupt Driven, all the same priority

- Interrupts with the same software priority are still given priority in the hardware through the vector table.
- If all events have same software priority, system behaves the same as in priority polling design.

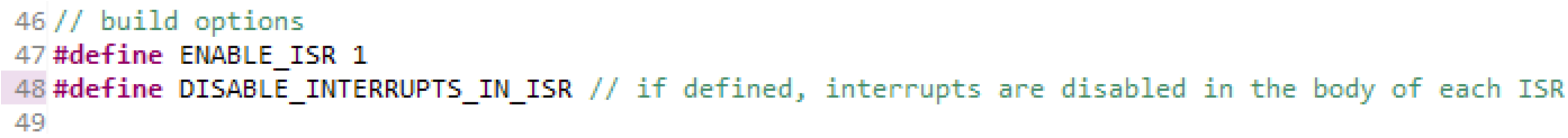


Expression	Type	Value	Priority Polling
(x)= event0_latency/120.0f	float	6000.4834	6.0 ms
(x)= event1_latency/120.0f	float	8097.3418	8.0 ms
(x)= event2_latency/120.0f	float	3001.1416	3.0 ms
(x)= event0_response_time/120.0f	float	8001.1001	8.0 ms
(x)= event1_response_time/120.0f	float	9097.94141	9.0 ms
(x)= event2_response_time/120.0f	float	9001.75879	9.0 ms
(x)= event0_missed_deadlines	unsigned int	834	
(x)= event1_missed_deadlines	unsigned int	291	
(x)= event2_missed_deadlines	unsigned int	0	
Add new expression			

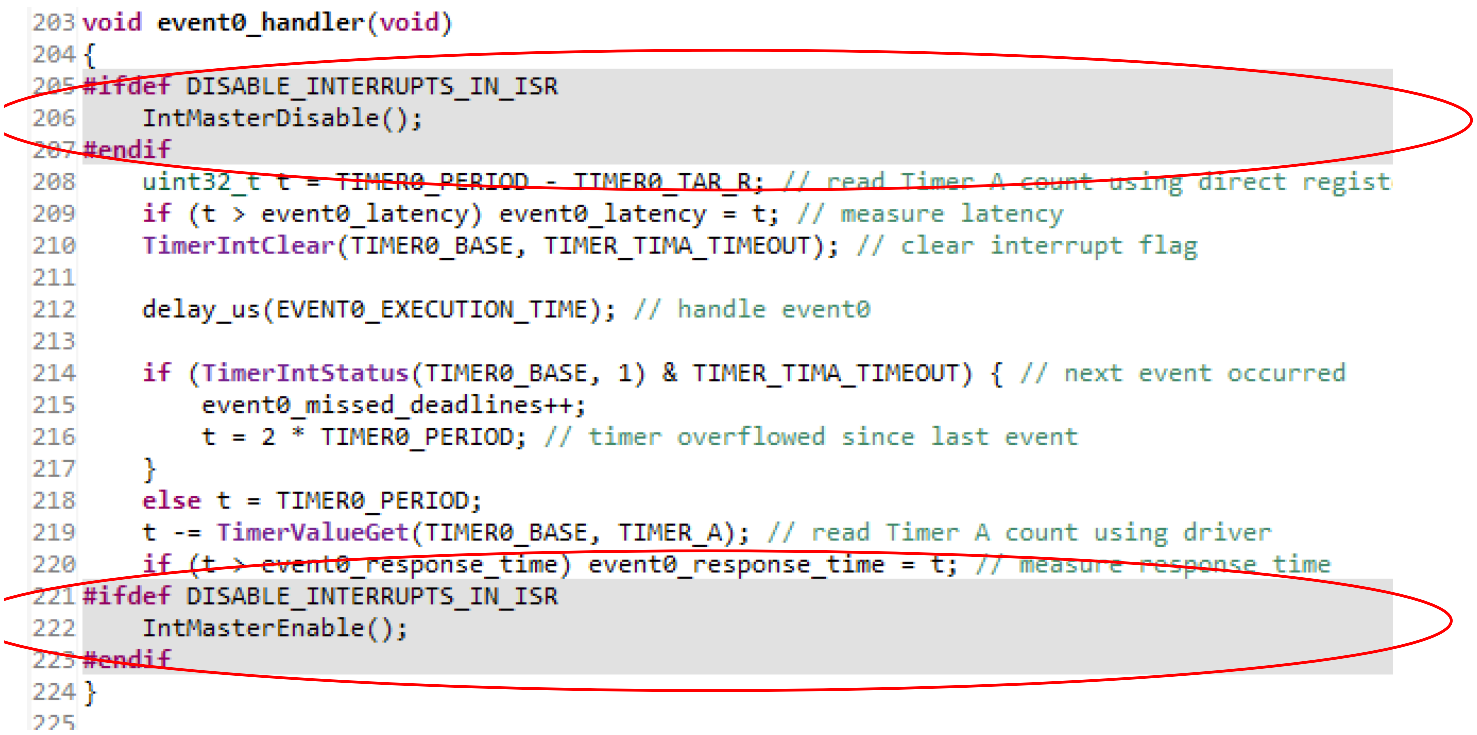
Not schedulable

Disabled Interrupts in the ISR

- In ece3849_int_latency, the interrupts can be disabled in the ISR routines.



```
// build options
#define ENABLE_ISR 1
#define DISABLE_INTERRUPTS_IN_ISR // if defined, interrupts are disabled in the body of each ISR

```



```
void event0_handler(void)
{
#ifdef DISABLE_INTERRUPTS_IN_ISR
    IntMasterDisable();
#endif
    uint32_t t = TIMER0_PERIOD - TIMER0_TAR_R; // read Timer A count using direct regist
    if (t > event0_latency) event0_latency = t; // measure latency
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // clear interrupt flag

    delay_us(EVENT0_EXECUTION_TIME); // handle event0

    if (TimerIntStatus(TIMER0_BASE, 1) & TIMER_TIMA_TIMEOUT) { // next event occurred
        event0_missed_deadlines++;
        t = 2 * TIMER0_PERIOD; // timer overflowed since last event
    }
    else t = TIMER0_PERIOD;
    t -= TimerValueGet(TIMER0_BASE, TIMER_A); // read Timer A count using driver
    if (t > event0_response_time) event0_response_time = t; // measure response time
#ifdef DISABLE_INTERRUPTS_IN_ISR
    IntMasterEnable();
#endif
}

```

Interrupts Globally disabled at start of ISR

Interrupts Globally re-enabled at end of ISR

Effect of Disabled Interrupts in the ISR

- Under what conditions is it important to disable interrupts?

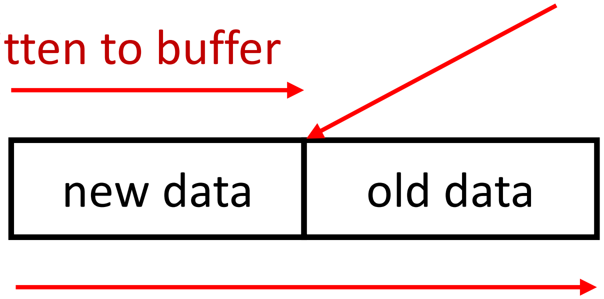
- What happens to the response time if we do this?
  - ?

Disabling Interrupts / Shared Data

- If ISRs are disabled, the latency will be the longest time they are disabled for.
    - In the ece3849_int_latency program, this was the entire execution time of each ISR.
- We may globally disable interrupts to protect access to shared data and resources.
    - event1 is filling up a buffer.
    - event0 is reading out of the buffer.
    - If event0 preempts event1, then all the data will not get written and the data read out will be a combination of old and new.

#2 Read event preempts the write event.

#1 New data being written to buffer



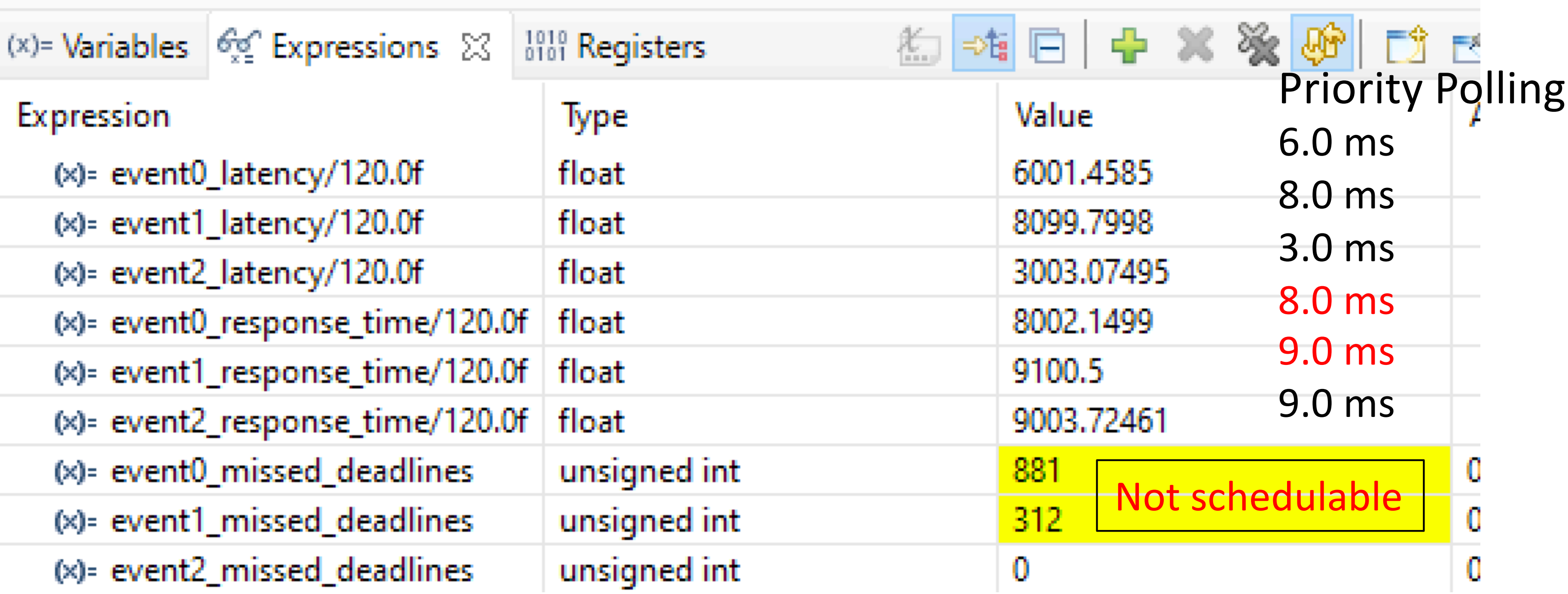
new data

old data

#3 Read event reads both old and new data causing it to be invalid

ece3849_int_latency
Example 3: event2 = 6.0, Interrupt Driven but disabled in ISR

- If Interrupts are disabled during the entire ISR, no preemption can happen and performance reverts back to priority polling.

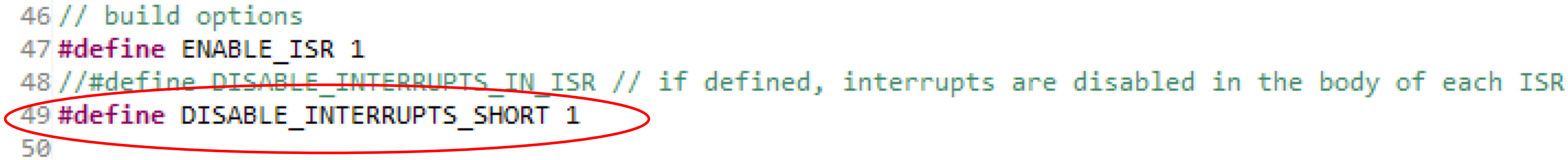


Expression	Type	Value	Priority Polling	
(x)= event0_latency/120.0f	float	6001.4585	6.0 ms	
(x)= event1_latency/120.0f	float	8099.7998	8.0 ms	
(x)= event2_latency/120.0f	float	3003.07495	3.0 ms	
(x)= event0_response_time/120.0f	float	8002.1499	8.0 ms	
(x)= event1_response_time/120.0f	float	9100.5	9.0 ms	
(x)= event2_response_time/120.0f	float	9003.72461	9.0 ms	
(x)= event0_missed_deadlines	unsigned int	881		0
(x)= event1_missed_deadlines	unsigned int	312		0
(x)= event2_missed_deadlines	unsigned int	0		0

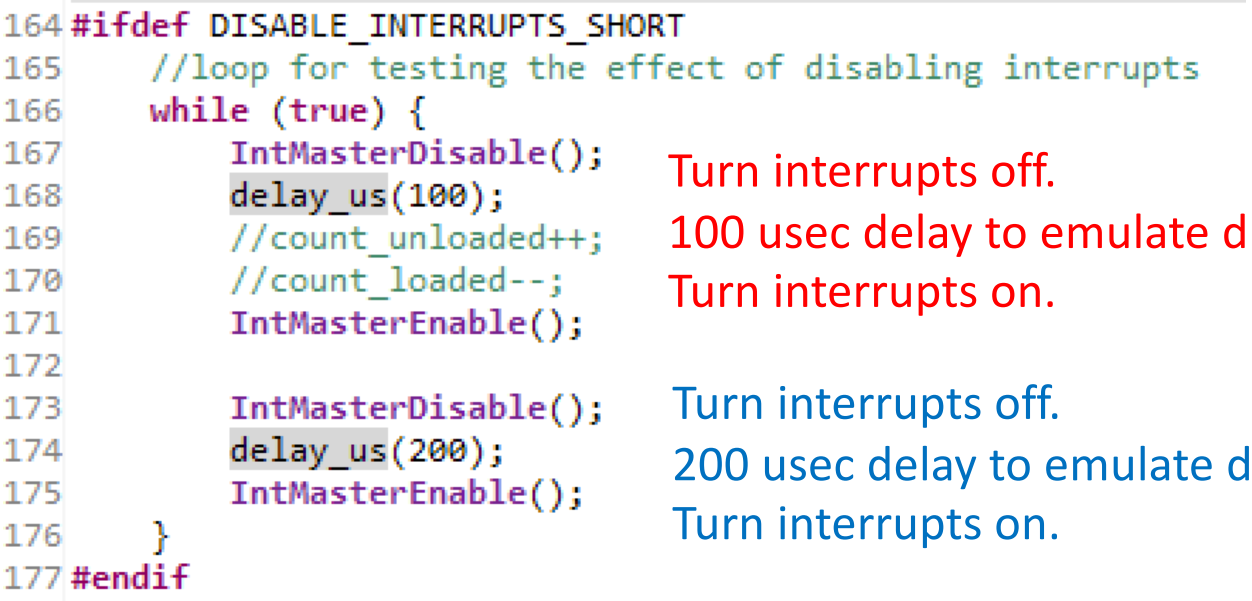
Not schedulable

Disabling Interrupts for short periods

- Maximum latency is affected by disabling interrupts.
  - It is important to minimize the time interrupts are disabled.
  - Only disable them during critical time to avoid shared data problems. Example…
    - Disable interrupts only when data being written or read to the buffer.
    - Enable at all other times.
- ece3849 int latency simulation (what do you think will happen?)



```
// build options
#define ENABLE_ISR 1
//#define DISABLE_INTERRUPTS_IN_ISR // if defined, interrupts are disabled in the body of each ISR
#define DISABLE_INTERRUPTS_SHORT 1

```

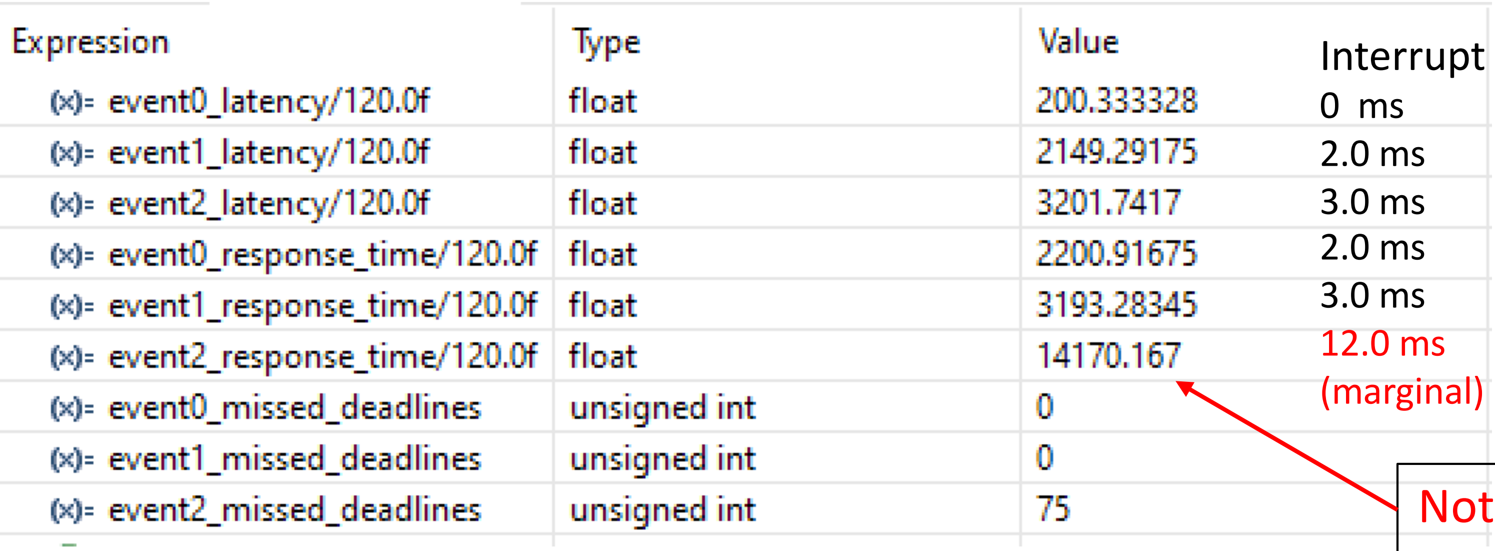


```
#ifdef DISABLE_INTERRUPTS_SHORT
    //loop for testing the effect of disabling interrupts
    while (true) {
        IntMasterDisable();
        delay_us(100);
        //count_unloaded++;
        //count_loaded--;
        IntMasterEnable();

        IntMasterDisable();
        delay_us(200);
        IntMasterEnable();
    }
#endif
```

Turn interrupts off.
100 usec delay to emulate disabling during data writing.
Turn interrupts on.

Turn interrupts off.
200 usec delay to emulate disabling during data reading.
Turn interrupts on.

DISABLE_INTERRUPTS_SHORT Results

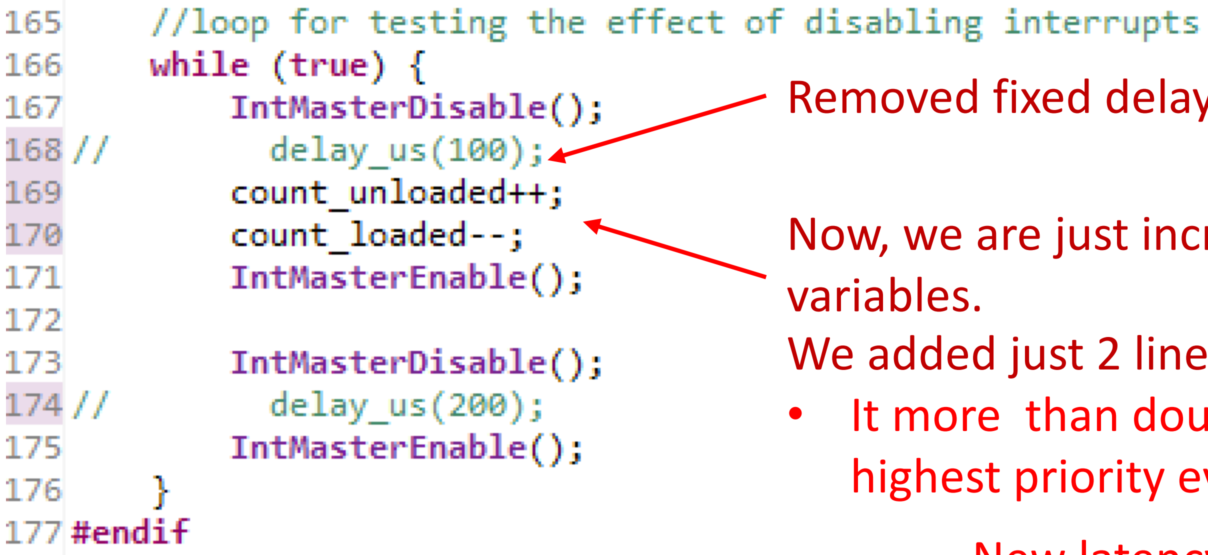


Expression	Type	Value	Interrupt Enable results
(x)= event0_latency/120.0f	float	200.333328	0 ms
(x)= event1_latency/120.0f	float	2149.29175	2.0 ms
(x)= event2_latency/120.0f	float	3201.7417	3.0 ms
(x)= event0_response_time/120.0f	float	2200.91675	2.0 ms
(x)= event1_response_time/120.0f	float	3193.28345	3.0 ms
(x)= event2_response_time/120.0f	float	14170.167	12.0 ms (marginal)
(x)= event0_missed_deadlines	unsigned int	0	
(x)= event1_missed_deadlines	unsigned int	0	
(x)= event2_missed_deadlines	unsigned int	75	

Not schedulable
14.2 > 12 msec

- Effect of disabling interrupts for short times.
  - Latency was increased by the maximum disabled time
    - Max(100 usec for write, 200 usec for read) = 200 usec.
  - All Latencies increased by 200 usec.
- Event2 now solidly not schedulable.
  - The latency caused an additional event0 event to preempt event2 adding 2 msec more to its execution time.
  - Designs without margin can be pushed from working to failing miserably by adding even tiny delays.

Even small changes make a big difference.



```
    //loop for testing the effect of disabling interrupts
    while (true) {
        IntMasterDisable();
//          delay_us(100);
        count_unloaded++;
        count_loaded--;
        IntMasterEnable();

        IntMasterDisable();
//          delay_us(200);
        IntMasterEnable();
    }
#endif
```

Removed fixed delays.

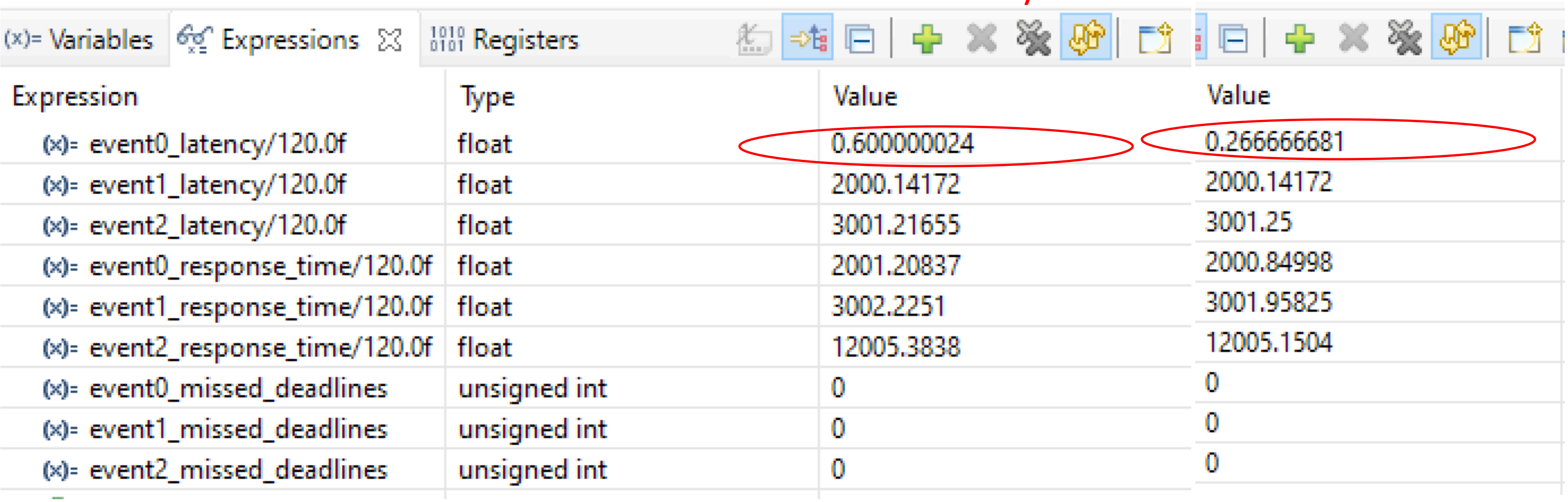
Now, we are just incrementing and decrement to variables.

We added just 2 lines of code.

- It more than doubled the latency of the highest priority event, 600 nsec >> 266 nsec.

New latency

Original Latency



Expression	Type	Value	Value
(x)= event0_latency/120.0f	float	0.600000024	0.266666681
(x)= event1_latency/120.0f	float	2000.14172	2000.14172
(x)= event2_latency/120.0f	float	3001.21655	3001.25
(x)= event0_response_time/120.0f	float	2001.20837	2000.84998
(x)= event1_response_time/120.0f	float	3002.2251	3001.95825
(x)= event2_response_time/120.0f	float	12005.3838	12005.1504
(x)= event0_missed_deadlines	unsigned int	0	0
(x)= event1_missed_deadlines	unsigned int	0	0
(x)= event2_missed_deadlines	unsigned int	0	0

Benchmarks and Metrics

- The latency of the highest priority task is a standard benchmark of a real time system.
  - It has the smallest latency, nothing can do better.
- Definitions of real-time metrics
  - Interrupt Latency = From hardware event to running the first instruction of the ISR.
  - Interrupt Response Time = Interrupt Latency + Context Saving Time
    - Context saving time is the time to save the contents of the CPU state.
  - ISR response time = Interrupt response time + maximum ISR execution time
  - Interrupt recovery time = Time to restore the context upon returning from the ISR.

Summary of Scheduling Strategies

- System assumptions
  - All events are periodic.
  - The relative deadline = period of the event.
  - Events are not phase aligned.
- Loops without preemption
  - The best option to keep things simple.
  - Round-Robin Polling
    - No priority, all events are equal.
    - Every event has an opportunity to execute.
    - Difficult to meet deadline for tasks with small periods.
  - Priority Polling
    - Prioritizes execution of shortest deadline first.
    - Minimizes latency of high priority tasks.
    - If high priority tasks are too frequent or have long execution times, this may cause low priority tasks to starve or miss their deadlines.

Summary of Scheduling Strategies

- Preemptive strategy with interrupts
  - Events with the shortest period are prioritized the highest.
  - They provide the lowest possible latencies for the highest priority events.
    - Lower priority events still suffer from starvation and are at risk for missing their deadlines.
  - Adds complexity requiring an interrupt controller to trigger events.
  - Higher priority tasks preempt lower priority tasks.
    - Evaluation of the maximum response times becomes difficult.
  - Shared data and resources may be required by multiple ISRs at the same time.
    - This can be addressed by disabling interrupts for very short periods of time.

Looking forward

- We must ensure low enough latency.
  - What we have looked at so far.
    - Interrupts reduce latency but must be prioritized.
    - Minimizing the time interrupts are disabled is critical.
  - Future steps
    - RTOS will provide more advanced scheduling options.
- We must ensure low enough execution time.
  - Profiling code to determine actual performance.
  - Writing code with deterministic execution time.
  - Optimizing code for worst-case performance.
  - Priority inheritance (mutex semaphores).
  - Understanding the hardware that runs the software.
  - Analyzing software performance at the instruction level.
  - Optimizing worst-case memory performance (caching).

Better Analysis Methods

- When we introduced interrupts, response time of lower priority events becomes difficult to calculate.
  - Often they are preempted multiple times?
  - What if conditions change and event 1 can be preempted multiple times too.

Event	Period	Execution Time	Latency	Response Time (Latency + ?)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	2 ms	0 ms	2 ms	6 ms	Yes
event1	8 ms	1 ms	2 ms	3 ms	8 ms	Yes
event2	12 ms	6 ms	3 ms	12 ms Tricky to calc	12 ms	Maybe, marginal

- We need a more robust method of calculating response time to guarantee we have thought of the worst case scenario.

Rate-Monotonic Scheduling Theory

- Rate-Monotonic Scheduling (RMS) Theory provides a more reliable way to verify the scheduling.
- The system must satisfy the following conditions.
  - All real-time tasks (events) are periodic with fixed periods.
    - Relative deadline = event period.
  - Fixed priorities are assigned to events according to period.
    - Shortest period = higher priority.
  - Higher priority tasks preempt lower priority ones.
  - Execution time per event is fixed for each task.
    - This is accomplished by using the maximum value.
  - Task switching overhead is negligible.
  - Tasks do not synchronize with each other
    - No blocking or disabling interrupts.

RMS – Theorem

- Theorem: “The first deadline is the hardest”
- If a task meets its first deadline when all tasks are released at the same time, then it will meet all future ones.
  - Worst case when all the tasks happen at once..
- Look at each task
  - Does it meet its first deadline?
  - If yes, it is schedulable for all conditions.
- Lets use a graphical model.

Task	Period	Execution Time	Priority	Latency	Response Time	Schedulable?
task0	6 ms	2 ms				
task1	8 ms	1 ms				
task2	12 ms	6 ms				

Unrolled schedule:

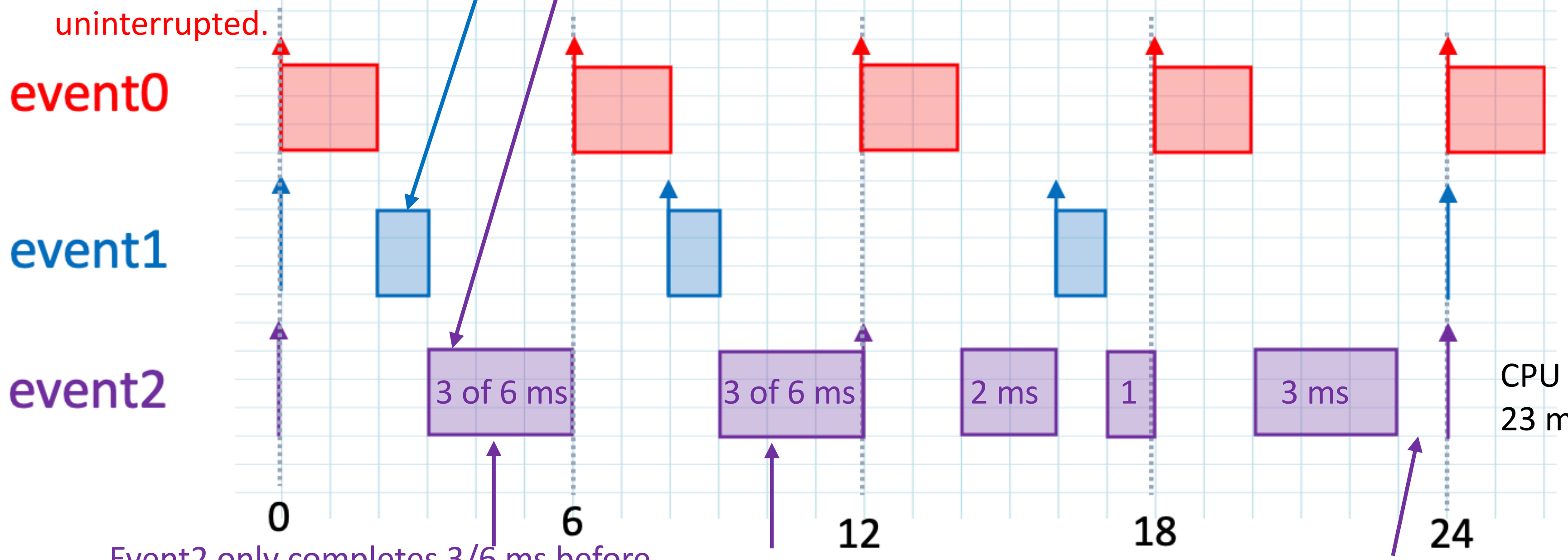
RMS Graphical Model

Event	Period	Execution Time	Priority	Latency	Response Time (Latency + ?)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	2 ms	High	0 ms	2 ms	6 ms	Yes
event1	8 ms	1 ms	Mid	2 ms	3 ms	8 ms	Yes
event2	12 ms	6 ms	Low	3 ms	12 ms	12 ms	Yes, marginal

Event0 happens like clock work, starts immediately uninterrupted.

Event1 must wait for event0 to end before starting.

Event2 must wait for event0 and 1 to end before starting.



CPU is running tasks 23 msec of 24 msec.

Event2 only completes 3/6 ms before event0 and then 1 preempts it.

Event2 resumes when event 0 and 1 are complete.

First deadline 0 ms margin. Second deadline easier has 1 msec.

Task	Period	Execution Time	Priority	Latency	Response Time	Schedulable?
task0	6 ms	1 ms				
task1	8 ms	4 ms				
task2	12 ms	3 ms				

Unrolled schedule:

RMS Graphical Model

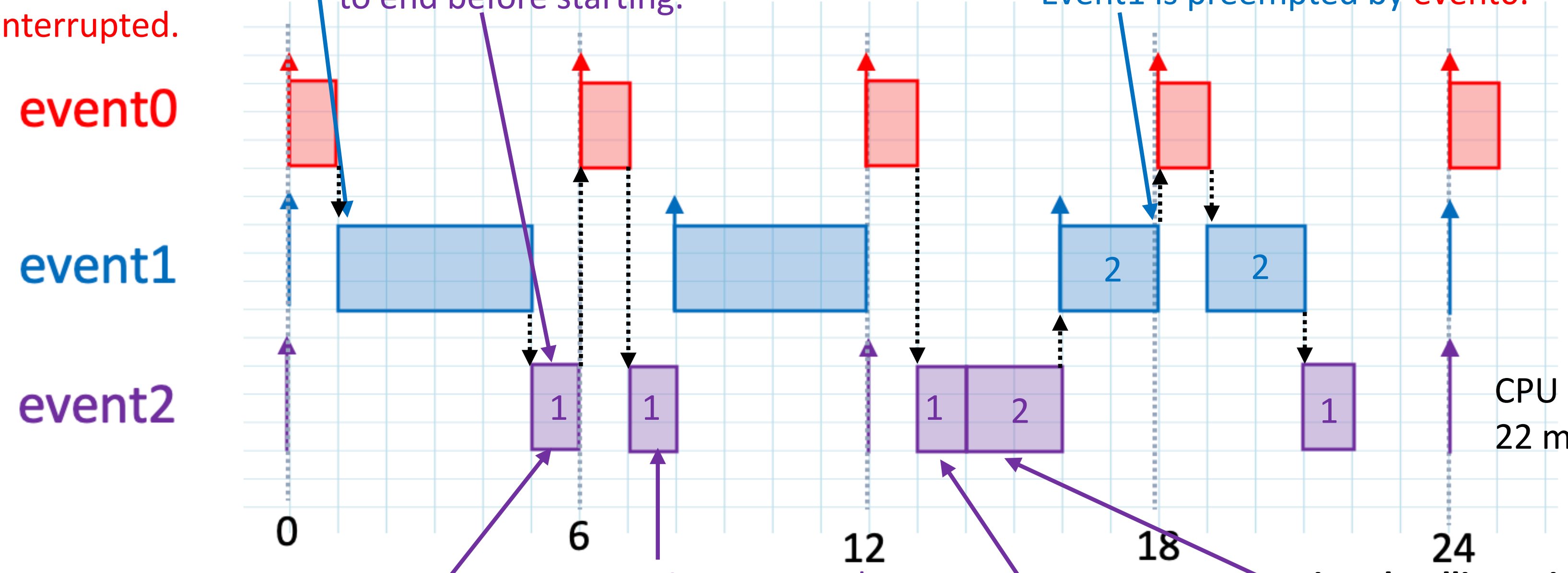
Event	Period	Execution Time	Priority	Latency	Response Time (Latency + ?)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	1 ms	high	0 ms	1 ms	6 ms	yes
event1	8 ms	4 ms	mid	1 ms	5 ms	8 ms	yes
event2	12 ms	3 ms	low	5 ms	14 ms	12 ms	NO

Event0 happens like clock work, starts immediately uninterrupted.

Event1 must wait for event0 to end before starting.

Event2 must wait for event0 and 1 to end before starting.

Event1 is preempted by event0.



CPU is running tasks 22 msec of 24 msec.

First deadline missed by 2 msec.

Event2 only completes 1/6 ms before event0 and then 0 preempts it.

Event2 resumes when event 0 is complete. Event 1 preempts it.

Event2 resumes when event 0 is complete. Next event2 is waiting to start and is preempted by event1.