ECE3849
D-Term 2021

Real Time Embedded Systems

Module 2 Part 1

Module 2 Part 1 Overview

- Scheduling Strategies Continued.
    - CPU Load Characterization.
    - Earliest Deadline First Scheduling.

ece3849_int_latency: cpu_load_count()

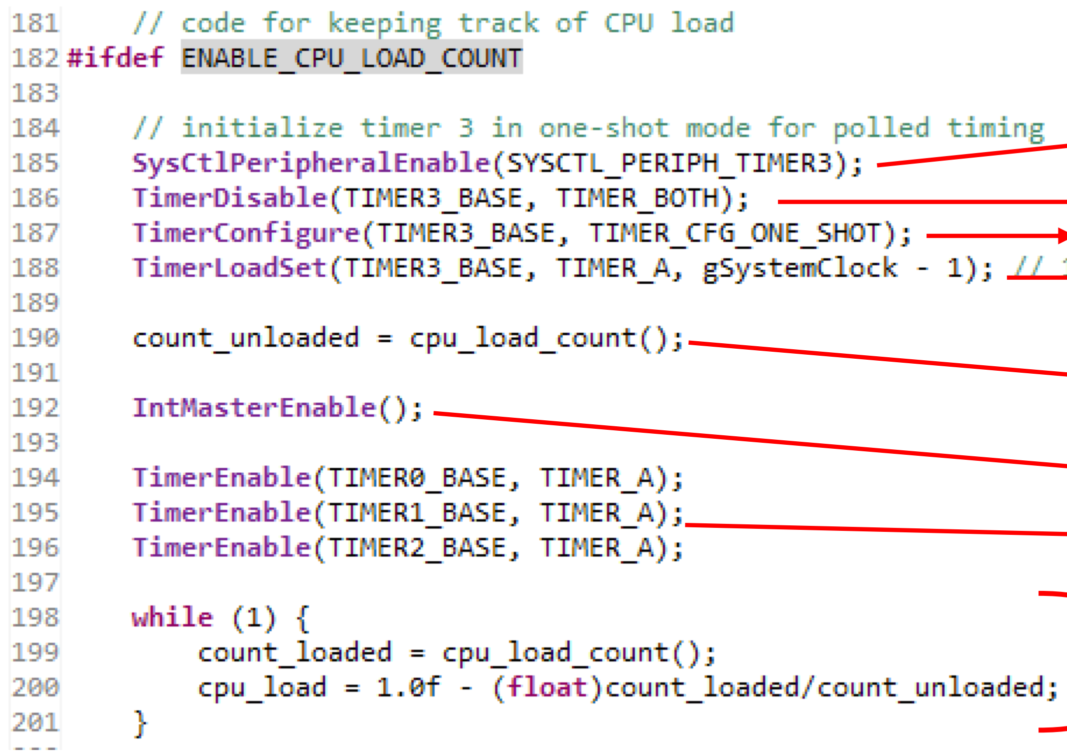
- The ENABLE_CPU_LOAD_COUNT option allows the program to estimate the fraction of time spent in real time tasks / ISR running in the background versus non-real time events running in the foreground.
- Program functionality.
  - TIMER3 initialization.
    - Configures a TIMER3 in a one shot configuration with a duration of 1 second.
    - When enabled the timer will count down to 0 over a 1 second interval.
    - Once at count = 0, it sets its interrupt status bit and holds its count at 0.
  - cpu_load_count() function.
    - Clears the interrupt status bit.
    - Starts the timer.
    - Then sits in a while loop waiting for the timers interrupt status bit to be set.
    - Each time through the while loop, a loop variable, i, is incremented.

ece3849_int_latency: CPU load calculation

- cpu_load_count() returns the number of loops completed in one timer interval = 1 second for this example.
- The number of while loops counted is proportional to the amount of time spent in the foreground.
  - When no interrupts are running,
    - All of the time will be in the foreground and the count will be high.
    - This measurement is take once at the beginning of the program and stored in the count_unloaded variable.
  - When interrupts are enabled,
    - cpu_load_count will be called repeatedly in the foreground loop for low priority tasks.
    - When the function is preempted by background ISRs, it does not count.
    - The resulting count is stored in the count_loaded variable.
  - The fraction of time spent in the foreground can be calculated by count_loaded/count_unloaded.
- CPU Utilization = 1 – count_loaded / count_unloaded.
- This gives us the average CPU utilization.

ece3849_int_latency: CPU load calculation

main()



```
    // code for keeping track of CPU load
#ifdef ENABLE_CPU_LOAD_COUNT

    // initialize timer 3 in one-shot mode for polled timing
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
    TimerDisable(TIMER3_BASE, TIMER_BOTH);
    TimerConfigure(TIMER3_BASE, TIMER_CFG_ONE_SHOT);
    TimerLoadSet(TIMER3_BASE, TIMER_A, gSystemClock - 1); // 1

    count_unloaded = cpu_load_count();

    IntMasterEnable();

    TimerEnable(TIMER0_BASE, TIMER_A);
    TimerEnable(TIMER1_BASE, TIMER_A);
    TimerEnable(TIMER2_BASE, TIMER_A);

    while (1) {
        count_loaded = cpu_load_count();
        cpu_load = 1.0f - (float)count_loaded/count_unloaded;
    }
```

Configures TIMER3.

- Enables TIMER3 peripheral for use.
- Disables operation during configuration.
- Configures the timer for ONE_SHOT mode.
- Loads the count register to 1 second.

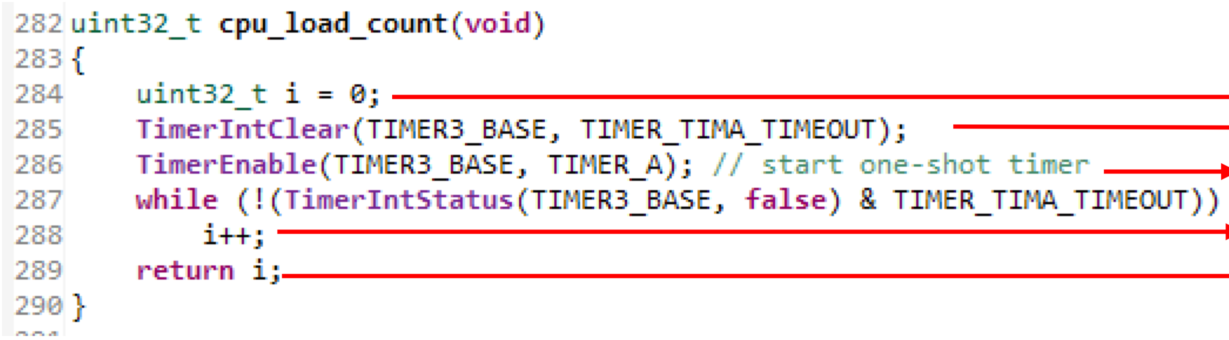
Takes one measurement with interrupt. disabled initialize count_unloaded.

Enables Interrupts.

Starts task counters, to trigger interrupts.

Enters foreground while loop, continually calling cpu_load_count and calculating CPU utilization.

cpu_load_count

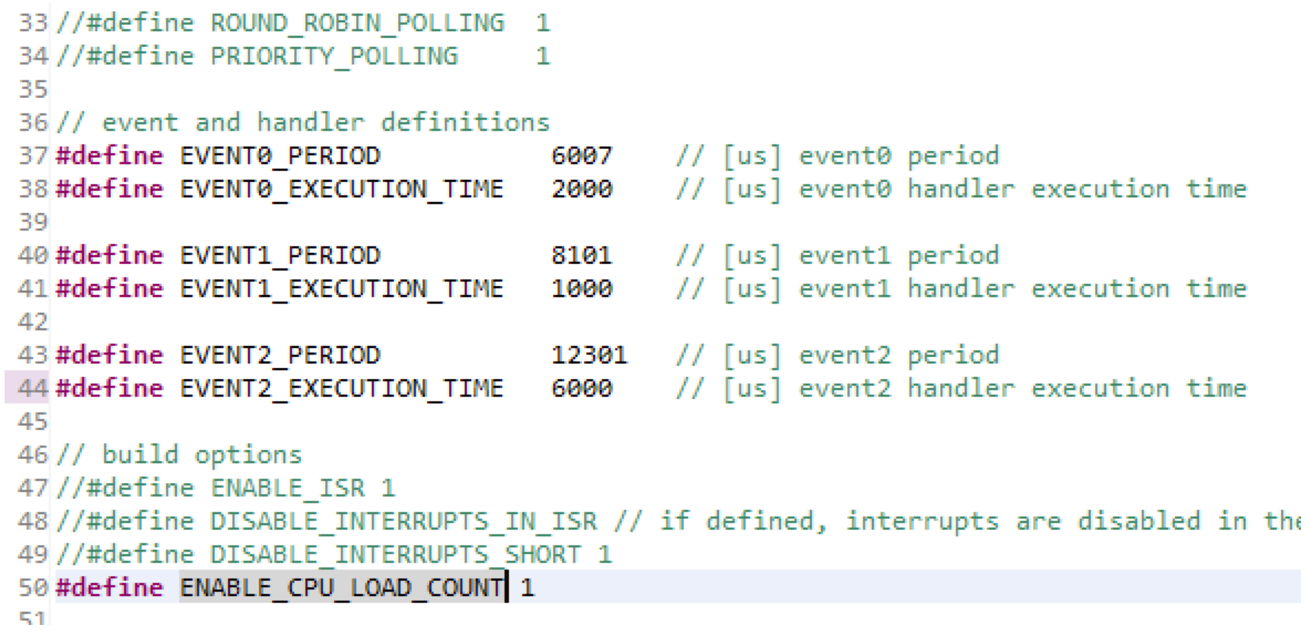


```
uint32_t cpu_load_count(void)
{
    uint32_t i = 0;
    TimerIntClear(TIMER3_BASE, TIMER_TIMA_TIMEOUT);
    TimerEnable(TIMER3_BASE, TIMER_A); // start one-shot timer
    while (!(TimerIntStatus(TIMER3_BASE, false) & TIMER_TIMA_TIMEOUT))
        i++;
    return i;
}
```

- Initializes the loop variable to 0.
- Clears the interrupt status from last time.
- Enables the counter.
- While TIMER3 is counting, it increments i.
- When 1 second is up it returns the i value.

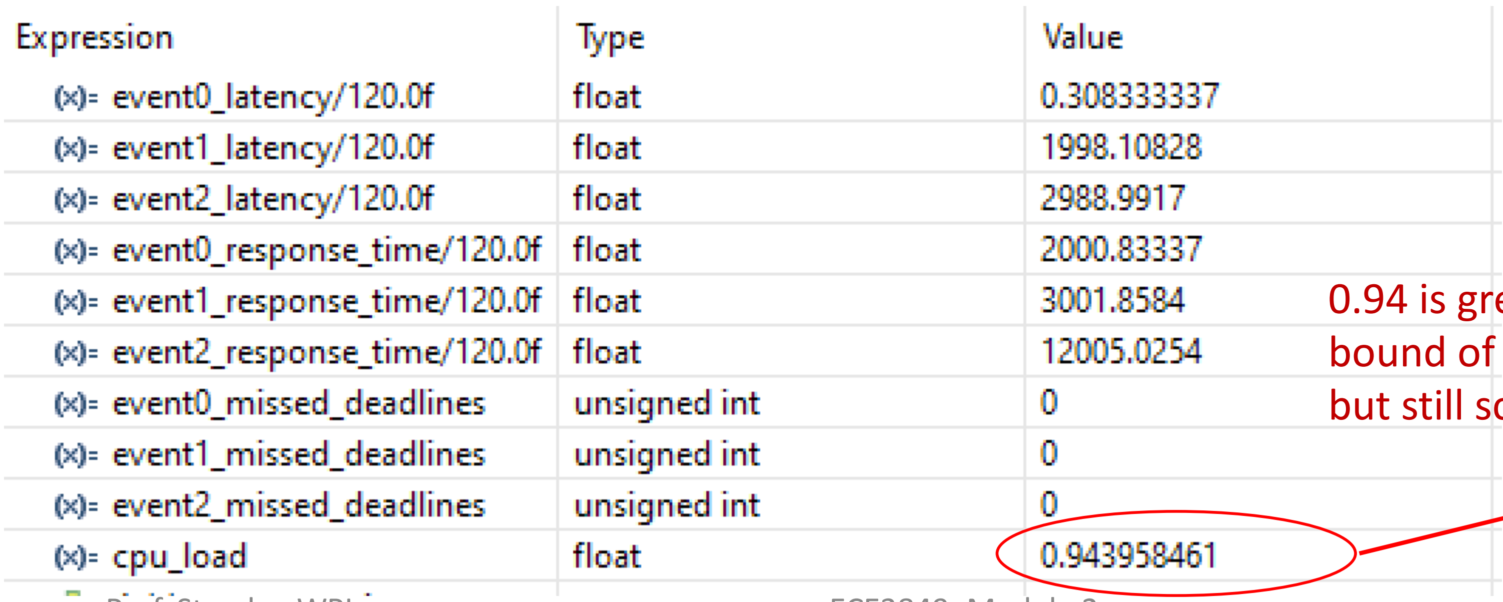
ece3849_int_latency: CPU load calculation

- 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 the
//#define DISABLE_INTERRUPTS_SHORT 1
#define ENABLE_CPU_LOAD_COUNT 1
```

- Run Results

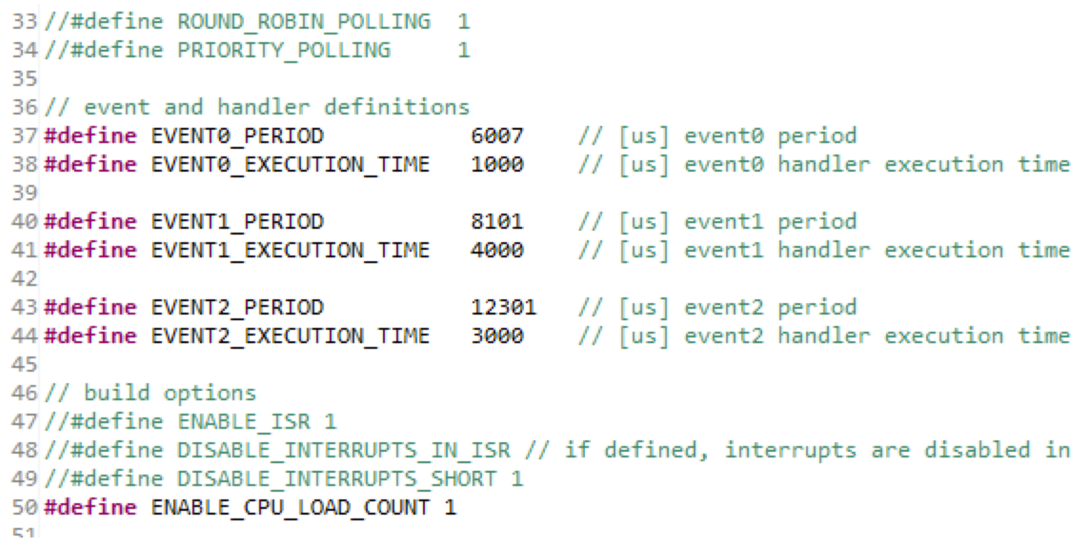


Expression	Type	Value
(x)= event0_latency/120.0f	float	0.308333337
(x)= event1_latency/120.0f	float	1998.10828
(x)= event2_latency/120.0f	float	2988.9917
(x)= event0_response_time/120.0f	float	2000.83337
(x)= event1_response_time/120.0f	float	3001.8584
(x)= event2_response_time/120.0f	float	12005.0254
(x)= event0_missed_deadlines	unsigned int	0
(x)= event1_missed_deadlines	unsigned int	0
(x)= event2_missed_deadlines	unsigned int	0
(x)= cpu_load	float	0.943958461

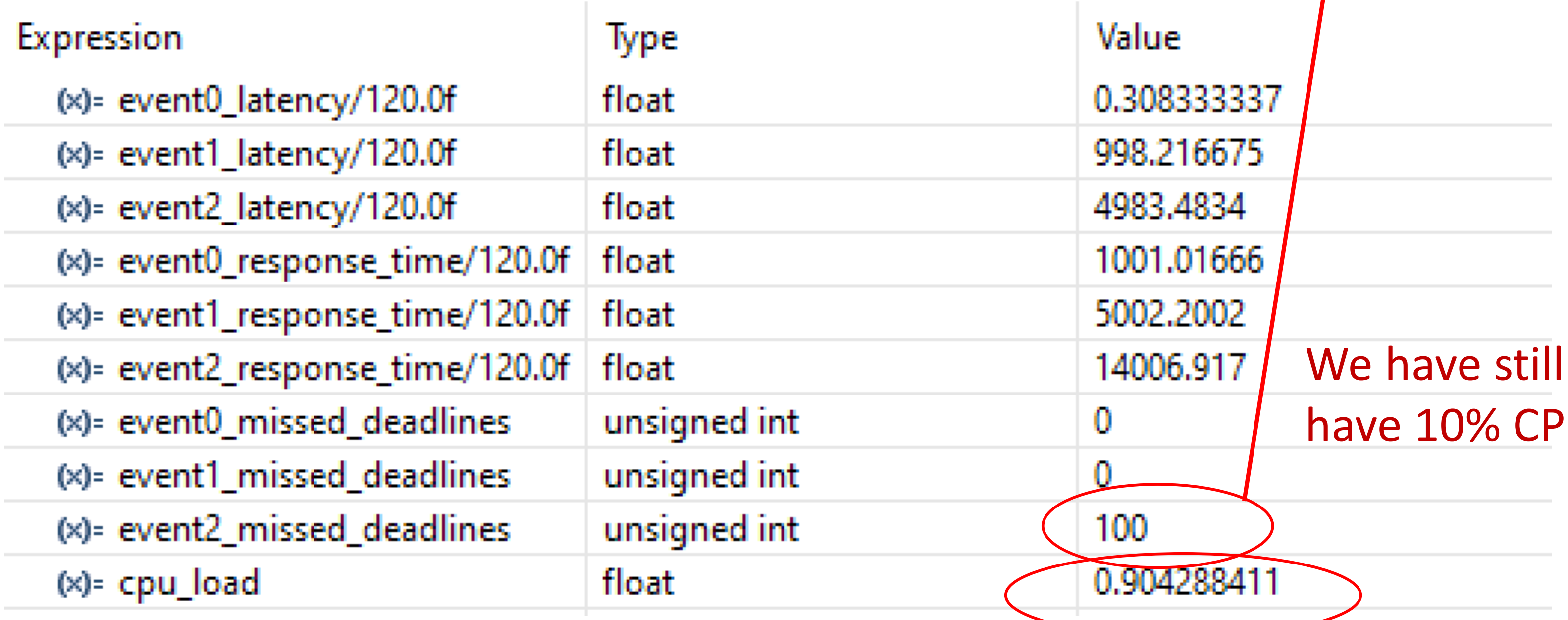
0.94 is greater than the upper bound of 0.7798 for three tasks but still schedulable.

Expected Value:
0.958

ece3849_int_latency: Missing Deadlines



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

// event and handler definitions
#define EVENT0_PERIOD          6007    // [us] event0 period
#define EVENT0_EXECUTION_TIME  1000    // [us] event0 handler execution time

#define EVENT1_PERIOD          8101    // [us] event1 period
#define EVENT1_EXECUTION_TIME  4000    // [us] event1 handler execution time

#define EVENT2_PERIOD          12301   // [us] event2 period
#define EVENT2_EXECUTION_TIME  3000    // [us] event2 handler execution time

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



Expression	Type	Value
event0_latency/120.0f	float	0.308333337
event1_latency/120.0f	float	998.216675
event2_latency/120.0f	float	4983.4834
event0_response_time/120.0f	float	1001.01666
event1_response_time/120.0f	float	5002.2002
event2_response_time/120.0f	float	14006.917
event0_missed_deadlines	unsigned int	0
event1_missed_deadlines	unsigned int	0
event2_missed_deadlines	unsigned int	100
cpu_load	float	0.904288411

We are missing deadlines.

But we are only using 90% of the time?

We have still have 10% CPU time to spare.

RMS Graphical Model

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:

How might change our scheduling algorithm to meet our deadlines?

- ?

RMS Graphical Model

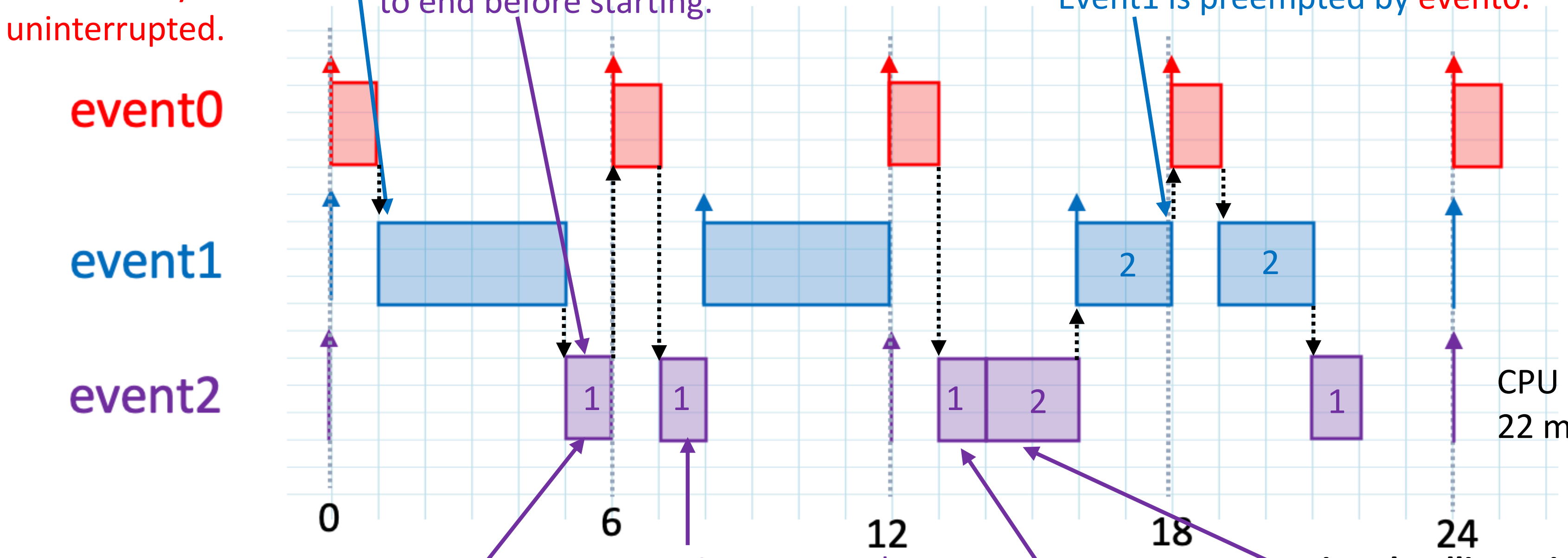
Event	Period	Execution Time	Priority	Latency	Response Time (Latency + ?)	Relative Deadline (Period)	Schedulable ? YES = response < deadline
event0	6 ms	1 ms	high	0 ms	1 ms	6 ms	yes
event1	8 ms	4 ms	mid	1 ms	5 ms	8 ms	yes
event2	12 ms	3 ms	low	5 ms	14 ms	12 ms	NO

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.

Earliest Deadline First Scheduling

- For the previous example Period = [6,8,12], Execution time = [1,4,3].
- The CPU utilization is
  - U = 1/6 + 4/8 +3/12 = 0.9167 = 91.67%
  - We still have extra CPU cycles to use: 91.67 % < 100 %
- To meet the deadline we need to throw out the fixed priority condition in the RMS Theory.
- The other conditions are still met
  - All real-time tasks are periodic with fixed periods.
  - Higher priority tasks preempt lower priority ones.
  - Execution time per event is fixed for each task.
  - Task switching overhead is negligible.
  - Task do to synchronize with each other.
- Earliest Deadline First (EDF) Scheduling
  - Uses Dynamic Priority
  - Looks at which deadline is coming first and chooses that task to execute.
  - The closer to the deadline, the higher the priority.
  - System needs to know the interrupt periods and keep track of them.

Earliest Deadline First Scheduling

- Tasks with the same deadline, have the same priority.
  - Tasks with the same priority will not preempt each other for this analysis.
  - In general system there will be a variety of choices on how to handle equal priority interrupts.

Event	Period	Execution Time	Latency	Response Time (Latency + ?)	Schedulable ? YES = response < deadline
event0	6 ms	1 ms			
event1	8 ms	4 ms			
event2	12 ms	3 ms			

Earliest Deadline First Scheduling

Event	Period	Execution Time	Latency	Response Time (Latency + ?)	Schedulable ? YES = response < deadline
event0	6 ms	1 ms	3 ms	4 ms	Yes
event1	8 ms	4 ms	1 ms	5 ms	Yes
event2	12 ms	3 ms	5 ms	8 ms	Yes

Utilization = 22/24 = 91.67%

#1 event0 has earliest = 6 , event1 = 8 , event2 = 12

#2 event1 has earliest = 8, event0 nothing pending, event2 = 12

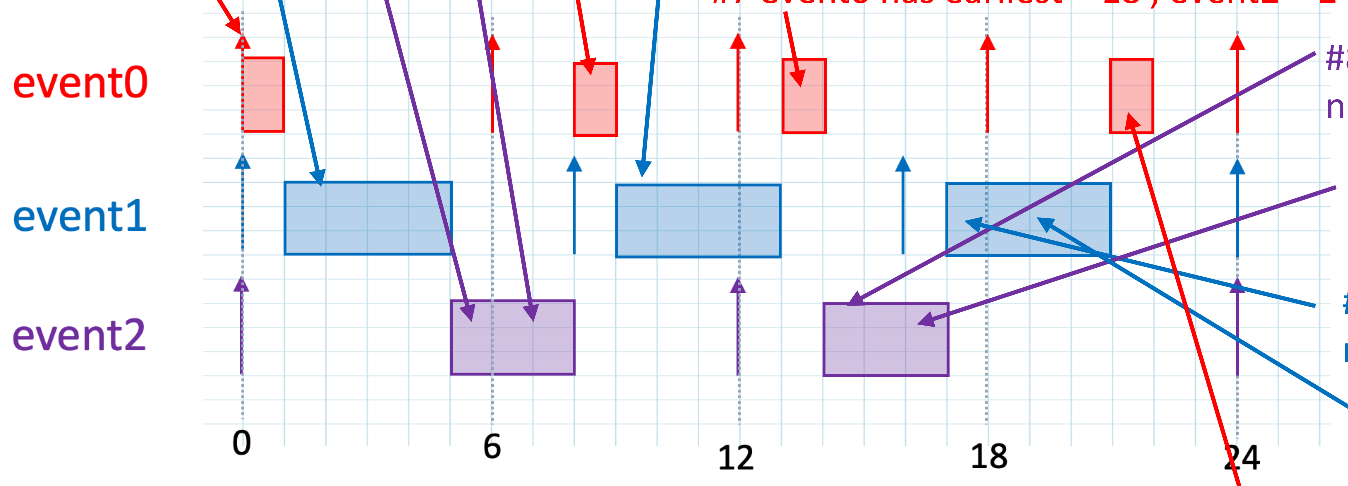
#3 event2 has earliest = 12, event0 & 2 nothing pending

#4 event2 and 0 = 12, they are equal complete event2 (no preemption)

#5 event0 has earliest = 12 , event1 = 16 , event2 nothing pending

#6 event1 has earliest = 16, event0 & 2 nothing pending

#7 event0 has earliest = 18 , event2 = 24, event1 nothing pending



#8 event2 has earliest = 24, nothing pending on 0 & 1

#9 event2 & 1 has earliest = 24, equal no preemption.

#10 event1 has earliest = 24, nothing pending on 0 or 2

#11 event1 & 0 = 24, equal no preemption.

#12 event0 has earliest = 24 , nothing pending on 1 or 2

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).