Lecture #15

Interrupt & Cyclic Task Response Timing

18-348 Embedded System Engineering

Philip Koopman

Monday, 14-March-2016

Electrical & Computer ENGINEERING

Carnegie Mellon

777 Flight Control

- **First Boeing "fly by wire" aircraft**
  - Only computer networks between pilot sticks and control surfaces

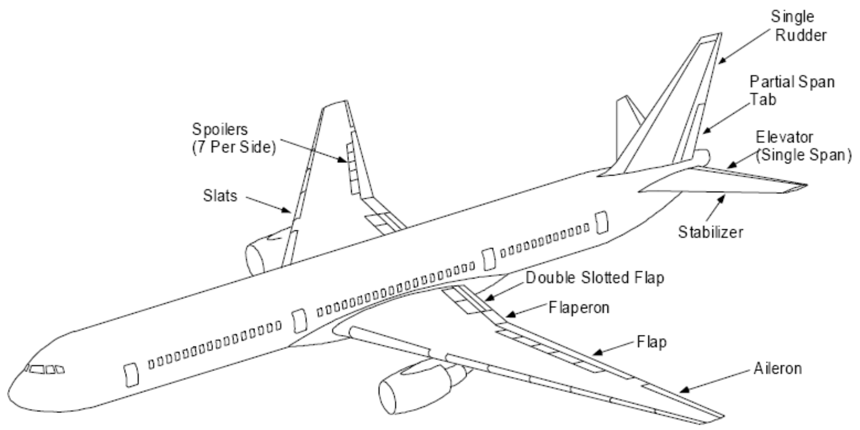
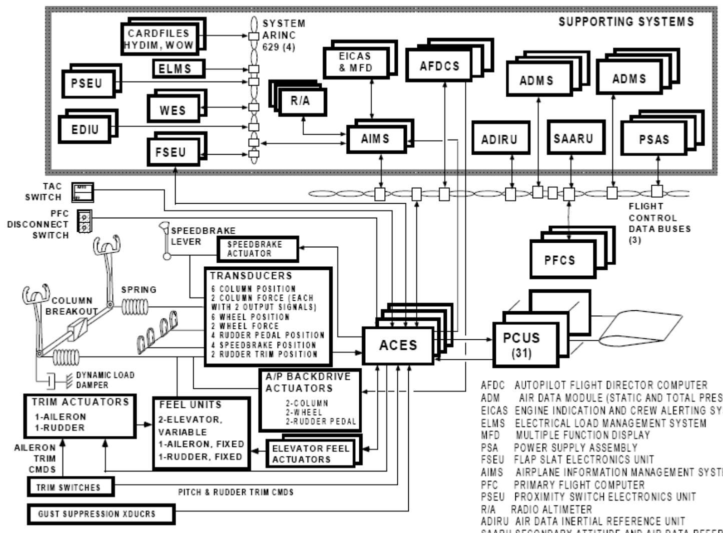


FIGURE 1 777 FLIGHT CONTROL SURFACES

[Yeh98]

777 Triplex Redundancy – 3 PFCs; 3 Networks

- **Note “feel units” to simulate feedback from mechanical flight surfaces**



AFDC	AUTOPILOT FLIGHT DIRECTOR COMPUTER
ADM	AIR DATA MODULE (STATIC AND TOTAL PRESSURES)
EICAS	ENGINE INDICATION AND CREW ALERTING SYSTEM
ELMS	ELECTRICAL LOAD MANAGEMENT SYSTEM
MFD	MULTIPLE FUNCTION DISPLAY
PSA	POWER SUPPLY ASSEMBLY
FSEU	FLAP SLAT ELECTRONICS UNIT
AIMS	AIRPLANE INFORMATION MANAGEMENT SYSTEM
PFC	PRIMARY FLIGHT COMPUTER
PSEU	PROXIMITY SWITCH ELECTRONICS UNIT
R/A	RADIO ALTIMETER
ADIRU	AIR DATA INERTIAL REFERENCE UNIT
SAARU	SECONDARY ATTITUDE AND AIR DATA REFERENCE UNIT
ACE	ACTUATOR CONTROL ELECTRONICS
PCU	POWER CONTROL UNITS, ACTUATORS
HYDIM	HYDRAULIC INTERFACE MODULE
WOW	WEIGHT ON WHEELS
WES	WARNING ELECTRONICS SYSTEM

[Yeh98]

Where Are We Now?

- **Where we’ve been:**
  - Interrupts
- **Where we’re going today:**
  - Looking at the timing of interrupts (and non-preemptive tasks)
- **Where we’re going next:**
  - More Interrupts, Concurrency, Scheduling
  - Analog and other I/O
  - Test #2

Preview

- **How do we organize multiple activities in an application?**
  - Especially if some of them are time sensitive?
- **Cyclic executive**
  - Put everything in one big main loop
- **ISRs only**
  - Use a bunch of ISRs to do all the work
  - Math to compute response time can get a bit hairy
- **Hybrid Main Loop + ISRs**
  - Many real systems are built this way
- **Overall – pay attention to the math**
  - More importantly, the insight behind the math!
  - There is an equation we expect you to really understand

Definition of Concurrency

- **A major feature of computation is providing the *illusion* of multiple simultaneously active computations**
  - Accomplished by switching among multiple computations quickly and frequently
- **Concurrency is when more than one computation is active at the same time**
  - Only one actually runs at a time, but many can be partially executed = “active”
  - ISR active when main program executing
  - Multiple threads active
  - Multiple tasks active
  - … in this course we’re only worried about single-CPU systems …
- **Gives rise to inherent problems**
  - Race conditions – if multiple computations access shared resources
  - Timing problems – if one computation affects timing of another
  - Memory problems – if computations compete for memory space

  - Attempting to fix the above problems leads to other problems, such as:
    - Deadlocks
    - Starvation

How Do You Achieve Concurrency?

- **Many techniques possible**
  - In big systems usually pre-emptive multitasking
  - But in embedded systems many other techniques are used
- **Why not just use a multitasking real time operating system?**
  - Sometimes this is the right choice, but it can be:
  - Too big (memory footprint might not fit on small CPU)
  - Too slow (overhead for task scheduling)
  - Too expensive (runtime license fee of $10 not reasonable on a $0.50 CPU)
  - Too complex (especially to guarantee deterministic timing)
  - Too hard to certify as safe (what if the RTOS has bugs?)
    - Only recently have some Real Time OS implementations been certified “safe”
- **So, let’s see techniques for concurrency and understanding task timing**
  - Today – concentrate on understanding timing of cyclic execs and ISRs

Simplest Approach – Cyclic Executive

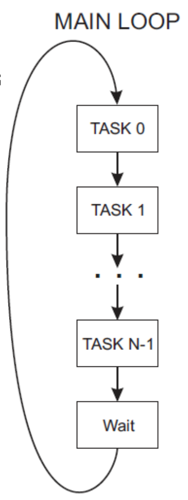
- **Create a main loop that executes each task in turn**
  - Run the loop so fast that all tasks appear to be active
  - Assume one task is catching bytes from the UART/SCI without being over-run by data rate
  - Other tasks just do various computations – really just subroutines in this version
  - No interrupts – only polled operation!

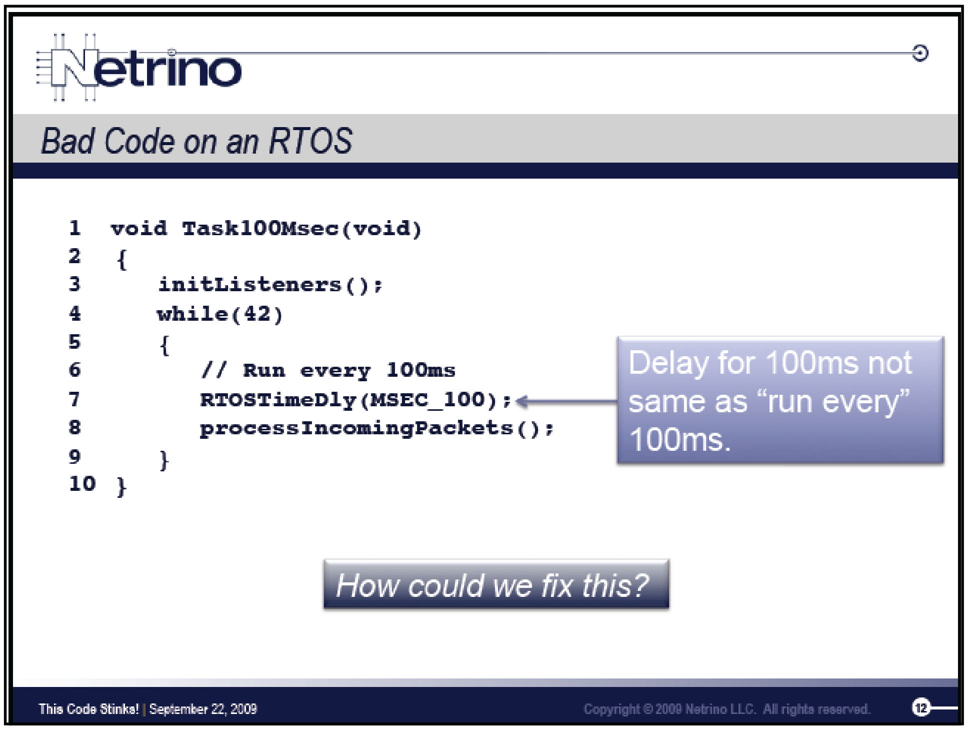
```
// main program loop
for(;;)
{  poll_uart();
   do_task1();
   do_task2();
}
```

- **“Executive”**
  - The main loop is the “executive” directing task execution … a very primitive scheduler

Cyclic Exec Tradeoffs

- **If you run main loop fast enough, implements concurrency**
  - Assume all registers saved/restored within each task
  - Ensure loop executes fast enough for poll_uart() to not miss any bytes
  - Simple timing analysis
    - Hard to get wrong as long as it “simple” and fast enough
  - Frequently used in safety critical applications
    - Timing is pretty much the same every time through loop
      - (assuming tasks are well behaved)
- **Obvious limitations**
  - All tasks have to fit within one sample of I/O
  - All code executed each time through loop, even if not really necessary
  - Have to make code “simple” so timing is easy to understand
- **Can do ad hoc conditional execution, but resist the temptation**
  - It turns into a mess!!! Insist on a “clean” approach; more ideas follow





Netrino

Bad Code on an RTOS

```
void Task100Msec(void)
{
    initListeners();
    while(42)
    {
        // Run every 100ms
        RTOSTimeDly(MSEC_100);
        processIncomingPackets();
    }
}
```

Delay for 100ms not same as “run every” 100ms.

How could we fix this?

Simple Multi-Rate Cyclic Executive

- **What if a single main loop is too slow?**
  - In previous example, all code runs completely each time through loop
  - Possible the UART will get over-run before task1 and task2 complete
  - Solution – break tasks down into self-contained parts
  - Embellishment: “Multi-rate” – some functions called more often than others
- **Notes on example:**
  - Each task part has to finish fast enough to meet minimum UART polling time
  - Each task has to save all its state somewhere (can’t carry live variables across task parts)
  - Can also have lists of pointers to tasks, etc.
    - Actual implementation varies but idea is the same
- **Q: Where should you kick the watchdog?**
- **Q: Why is the “waitForTimer” important?**

```
// main program loop
for(;;)
{  poll_uart();
   do_task1_part1();
   poll_uart();
   do_task1_part2();
   poll_uart();
   do_task1_part3();

   poll_uart();
   do_task2_part1();
   poll_uart();
   do_task2_part2();
   poll_uart();
   do_task2_part3();
   waitForTimer();
}
```

General Multi-Rate Cyclic Exec Tradeoffs

- **More flexible than simple cyclic executive**
  - Execute different tasks at different frequencies as needed
  - But, each task executes an integer number of times per main loop
- **Timing still restrictive**
  - Each task or part of task has to be short enough to finish before fastest task needs to execute again
    - Breaking up a long task into short pieces can be very painful
    - If time for fastest task changes, might have to rewrite code in other tasks
  - Hand-schedule to cover worst case delay between executions of fastest task
- **But, still simple to analyze**
  - Each loop through tasks can be the same as every other loop
  - Worst case is each line in main loop executes exactly once
    - poll_uart() 6 times per loop; everything else once
  - Again – resist urge to do ad hoc adaptive scheduling – always creates a mess!
    - By this, we mean don’t use an “if” to decide whether a task should run

Concept – Latency and Response Time

- **Latency is, generically, the waiting time for something to happen**
  - For real time computing, it's all about latency!
  - Non-interrupts – time between executions of a task (worst case wait)
  - Interrupts – time between interrupt request asserted and ISR executing (worst case wait)
  - "Low" latency = Short wait ("good"); "High" latency = Long wait ("bad")
  - ***Response time*** is more precise – max time until computation *starts* running
- **For simple cyclic execution:**
  - Response time for any task is one time through main loop
- **For multi-rate cyclic exec:**
  - Response time is time between repeated executions of a particular task
    - In this example, six times faster for UART polling than for other tasks
    - In general, depends on how tasks are listed in the main loop
- **What if low latency really only matters for one task, and it is short?**
  - Then use an ISR…

Cyclic Exec Plus Interrupts

- **Process non-time-critical routines in foreground**
  - Repeated periodically
- **Process one (or a few) time critical functions in background**
  - UART serviced on interrupt instead of polled
  - UART can run at speed independent of other tasks!
  - Other tasks don't have to be broken down into pieces as long as each task can wait for its turn in loop
- **But, it's not a free lunch!**
  - What's the latency for task1?
  - Time to execute whole loop *plus some number* of executions of ISRs

```
// main program loop
for(;;)
{     do_task1();
      do_task2();
}

void interrupt 20
   handle_uart(void)
//-(20*2)-2 = $FFD6 for REI
{  … <service UART/SCI> …
}
```

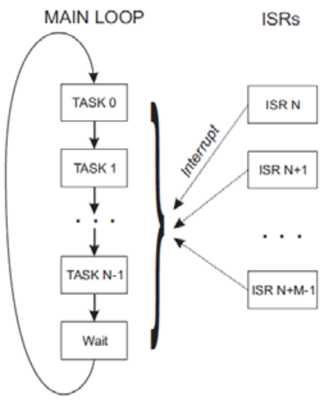
Latency With Interrupts – Simple Version

- **For previous example, latency of handle_uart() is:**
  - Can run back-to-back as many times as needed
  - So, very low latency

- **What's guaranteed worst case latency of do_task1()?**
  - Potentially infinite … if handle_uart() runs back-to-back forever

- **What's expected latency of tasks in main loop?**
  - How many times can UART receive a byte in main loop? call it *N*
  - Worst case execution time of main loop (*simple version*) is:

    execution time of do_task1()
    + execution time of do_task2()
    + ***N*** * execution time of handle_uart()
  - Fortunately, bounded by speed of serial port
    - But, main loop slows down as baud rate goes up, giving time for ***more interrupts*** (this is an essential property of interrupt scheduling; more detail in a few slides)



Latency With Multiple Interrupts – Main Loop

- **There's never just one interrupt in the worst case**
  - What if multiple interrupts can occur?
  - Latency is number of times each interrupt can occur (*simple version*)
    - Assume *M* of ISR1
    - *N* of ISR2
    - *P* of ISR3
    - (in practice could be 10+ different interrupts; but 3 works for an example)

  - Worst case execution time of main loop (*simple incorrect version*) is:

    execution time of do_task1()
    + execution time of do_task2()
    + ***M*** * execution time of ISR1()
    + ***N*** * execution time of ISR2()
    + ***P*** * execution time of ISR3()

  - So worst case for main loop gets worse as interrupts are added
    - What did we mean by "*simple version?*" … we mean that it is actually incorrect – the correct version is more complex

Cyclic+ISR Main Latency – The Correct Version

- **As ISRs execute, time for main loop is extended**
  - As time is extended, there is time for more ISRs to take place
  - As more ISRs take place, time is further extended…
  - Final time is recursive infinite summation
- **Consider this example:**
  - task1 takes 100 msec
  - task2 takes 150 msec
  - ISR1 takes 1 msec; repeats at most every 10 msec
  - ISR2 takes 2 msec; repeats at most every 20 msec
  - ISR3 takes 3 msec; repeats at most every 30 msec
  - How long is worst case main loop execution time (i.e., task1 and task2 latency?)
    - main loop with no ISRs is **250 msec**
    - In 250 msec, could have 26 @ ISR1; 13 @ ISR2; 9 @ ISR3 = 250+79 msec = 329
    - In 329 msec, could have 33 @ ISR1; 17 @ ISR2; 11 @ ISR3 = 250+100 msec = 350
    - In 350 msec, could have 36 @ ISR1; 18 @ ISR2; 12 @ ISR3 = 250+108 msec = 358
    - In 358 msec, could have 36 @ ISR1; 18 @ ISR2; 12 @ ISR3 = 250+108 msec = **358 msec**
      - » (process converges when you get same answer twice in a row)

Cyclic + ISR Main Latency – The Math

- **Given:**
  - Main loop with no ISRs executes in MainLoopOnly
  - ISR$_m$ takes ISRtime$_m$ to execute and runs at most every ISRperiod$_m$

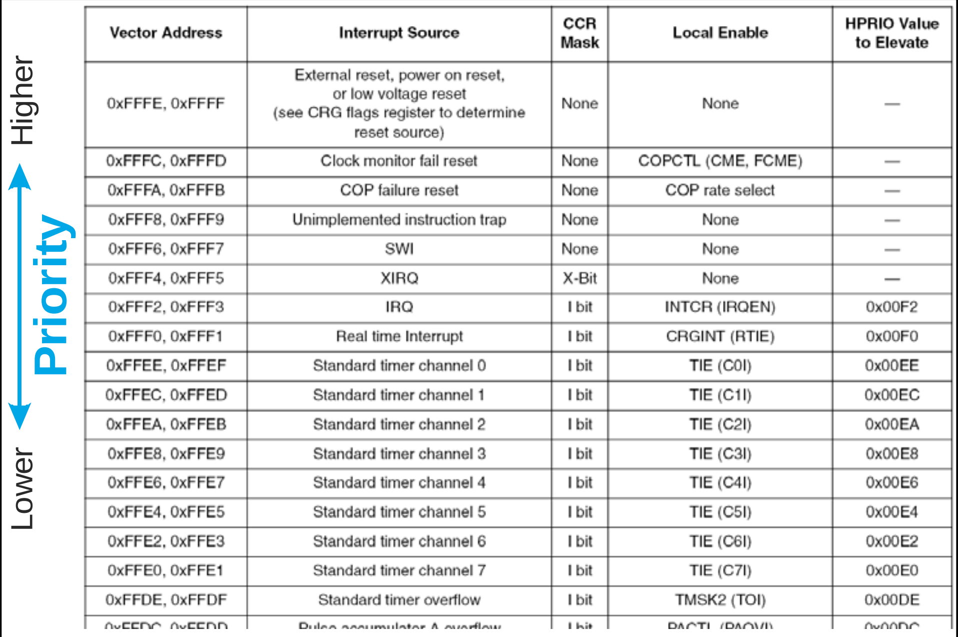
$$MainTime_0 = MainLoopOnly$$

$$MainTime_{i+1} = MainTime_0 + \sum_{\forall ISRs_j} \left\lfloor \frac{MainTime_i}{ISRperiod_j} + 1 \right\rfloor ISRtime_j$$

  - Note that this uses a ***FLOOR FUNCTION*** – not square brackets "[ ]"
  - This is really just the calculation we worked out on the previous slide
- **Worst case main loop execution time is**
  - Take floor of number of times each ISR can execute+1 times execution time
  - This extends main loop latency ….
    … meaning each ISR might be able to execute more times
  - Continue evaluation until latency$_i$ converges to a fixed value
  - This is why we kept saying "easier to evaluate" for non-ISR schedules!

What About Latency For Interrupts Themselves?

- **Interrupts are usually the high priority, fast-reaction-time routines**
  - With only one ISR, latency is just waiting for interrupt mask to turn off
    - Same ISR might already be running – wait for RTI
    - I flag might be set (SEI) – wait for next CLI
  - But with multiple ISRs in system, it gets more complex
    - Wait for interrupt mask to be turned off
    - Wait for other ISRs to execute
- **Let's take the case of prioritized interrupts**
  - When multiple interrupts are pending, one of them gets priority over others


Higher
Priority
Lower

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
0xFFFE, 0xFFFF	External reset, power on reset, or low voltage reset (see CRG flags register to determine reset source)	None	None	—
0xFFFC, 0xFFFD	Clock monitor fail reset	None	COPCTL (CME, FCME)	—
0xFFFA, 0xFFFB	COP failure reset	None	COP rate select	—
0xFFF8, 0xFFF9	Unimplemented instruction trap	None	None	—
0xFFF6, 0xFFF7	SWI	None	None	—
0xFFF4, 0xFFF5	XIRQ	X-Bit	None	—
0xFFF2, 0xFFF3	IRQ	I bit	INTCR (IRQEN)	0x00F2
0xFFF0, 0xFFF1	Real time Interrupt	I bit	CRGINT (RTIE)	0x00F0
0xFFEE, 0xFFEF	Standard timer channel 0	I bit	TIE (C0I)	0x00EE
0xFFEC, 0xFFED	Standard timer channel 1	I bit	TIE (C1I)	0x00EC
0xFFEA, 0xFFEB	Standard timer channel 2	I bit	TIE (C2I)	0x00EA
0xFFE8, 0xFFE9	Standard timer channel 3	I bit	TIE (C3I)	0x00E8
0xFFE6, 0xFFE7	Standard timer channel 4	I bit	TIE (C4I)	0x00E6
0xFFE4, 0xFFE5	Standard timer channel 5	I bit	TIE (C5I)	0x00E4
0xFFE2, 0xFFE3	Standard timer channel 6	I bit	TIE (C6I)	0x00E2
0xFFE0, 0xFFE1	Standard timer channel 7	I bit	TIE (C7I)	0x00E0
0xFFDE, 0xFFDF	Standard timer overflow	I bit	TMSK2 (TOI)	0x00DE
0xFFDC, 0xFFDD	Pulse accumulator A overflow	I bit	PACTL (PAOVI)	0x00DC

Latency For Prioritized Interrupts

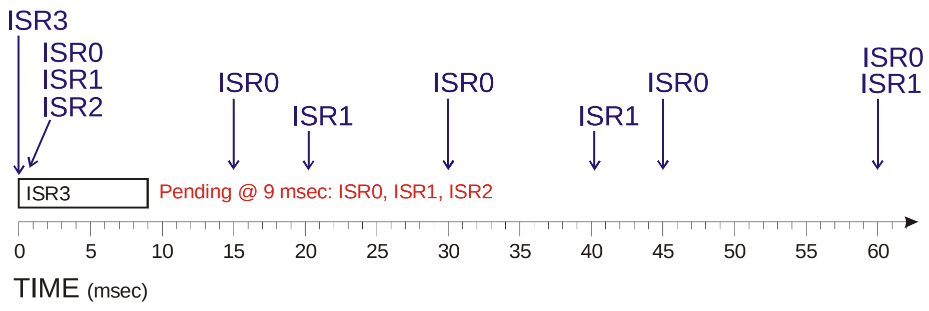
- **Have to wait for other interrupts to execute**
  - One might already be executing with lower priority (have to wait)
    - Or, interrupts might be masked for some other reason ("blocking")
  - All interrupts at higher priority might execute one or more times
  - Worst case – have to assume that every possible higher priority interrupt is queued AND longest possible blocking time (lower priority interrupt)

- **Example, (same as previous situation):**
  - ISR1 takes 1 msec; repeats at most every 10 msec
  - ISR2 takes 2 msec; repeats at most every 20 msec
  - ISR3 takes 3 msec; repeats at most every 30 msec

  - For ISR2, latency is:
    - ISR3 might just have started – 3 msec
    - ISR1 might be queued already – 1 msec
    - ISR2 will run after 3 + 1 = 4 msec
      - » This is less than 10 msec total (period of ISR1), so ISR1 doesn't run a second time

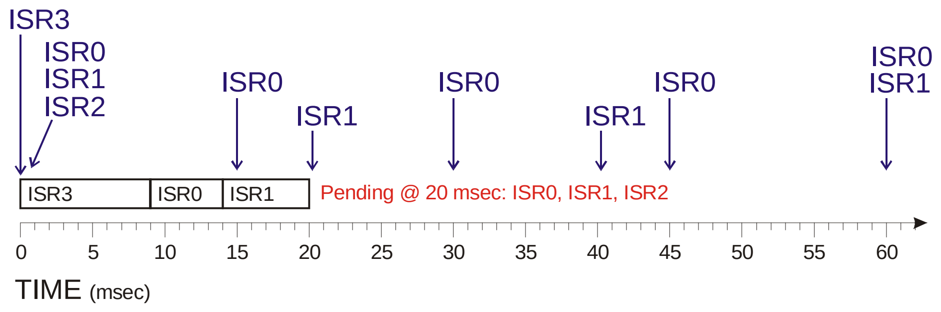
Example – ISR Worst Case Latency

- **Assume following task set (ISR0 highest priority):**
  - ISR0 takes 5 msec and occurs at most once every 15 msec
  - ISR1 takes 6 msec and occurs at most once every 20 msec
  - **ISR2** takes 7 msec and occurs at most once every 100 msec
  - ISR3 takes 9 msec and occurs at most once every 250 msec
  - ISR4 takes 3 msec and occurs at most once every 600 msec

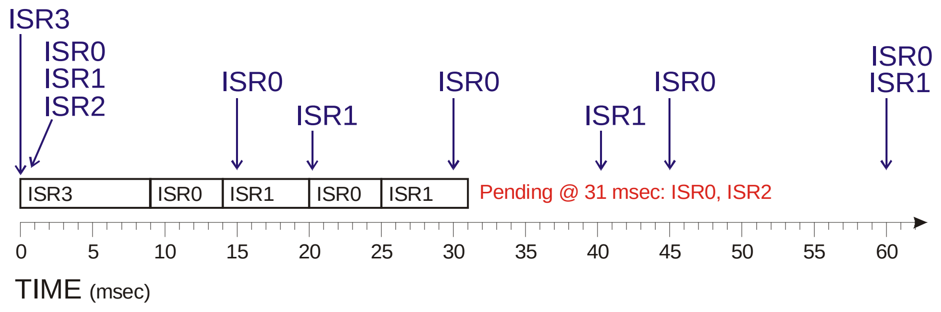


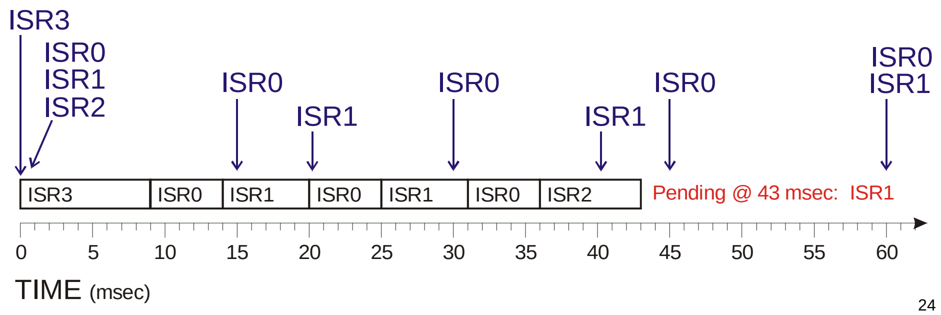
Will ISR2 Execute Within 50 msec?

- **Worst Case is ISR3 runs just before ISR2 can start**
  - Why this one? – has longest execution time of everything lower than ISR2
- **Then ISR0 & ISR1 go because they are higher priority**
  - But wait, they retrigger by 20 msec – so they are pending *again*



ISR0 & ISR1 Retrigger, then ISR2 goes





ISR Latency – The Math

- **In general, higher priority interrupts might run multiple times!**
  - Assume N different interrupts sorted by priority (0 is highest; N-1 is lowest)
  - Want latency of interrupt *m*

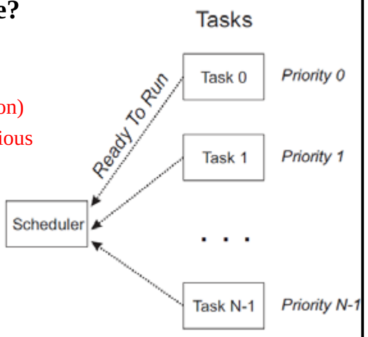
$$ilatency_0 = 0$$

$$ilatency_{i+1} = \max_{j>m}\left(ISRtime_j\right) + \sum_{\forall ISRs_{j<m}} \left\lfloor \frac{ilatency_i}{ISRperiod_j} + 1 \right\rfloor ISRtime_j$$

  - Very similar to equation for main loop
    - What it's saying is true for anything with preemption plus initial blocking time:
    1. You have to wait for <u>*one worst-case*</u> task at same or lower priority to complete
    2. You always have to wait for all tasks with higher priority, sometimes repeated

Another Approach – Everything in Interrupts

- **What if everything in our system is time sensitive?**
  - Another way to organize things is put everything in interrupts
    - ***<u>You don't really want to do this!!!</u>*** (we'll see why soon)
    - BUT, it gives insight into the scheduling math and various options



```
…set up interrupts here…
// main program loop
for(;;)
{ // could just do nothing!
}
// interrupt priority is in device order (#20 is ISR0)
void interrupt 20 handle_device0(void) { …… }
void interrupt 21 handle_device1(void) { …… }
void interrupt 22 handle_device2(void) { …… }
void interrupt 23 handle_device3(void) { …… }
   …
```

General Latency For Prioritized Tasks

- **This is for the non-preemptive case (tasks can't be pre-empted)**
  - True of interrupts that don't clear the I bit
  - True of main loop as well – it is effectively the lowest priority task (task *N*)

- **Notation:**
  - Each task is numbered *i*; *i=0 is highest priority; i=N-1 is lowest*
  - You know how long each task takes to execute (at least in worst case) – C_i
  - You know period of interrupt arrival (worst case) – P_i
  - Interrupts are never disabled by main program
  - Interrupts are non-preemptive (once an ISR starts, it runs to completion)

$$R_{i,0} = \max_{i<j<N}(C_j) \qquad ;i < N-1$$

$$R_{i,k+1} = R_{i,0} + \sum_{m=0}^{m=i-1}\left(\left\lfloor \frac{R_{i,k}}{P_m} + 1 \right\rfloor C_m\right) \qquad ;i > 0$$

  - R_i is response time time until *i starts execution*, same as previous latency equation; just cleaner notation

Example Response Time Calculation

- **What's the Response Time for task 2?**
  - Note: N=4 (tasks 0..3)
  - Have to wait for task 3 to finish
    - (longest execution time)
  - Have to wait for two execution of task 0
  - Have to wait for one execution of task 1

Task# *i*	Period (P_i)	Execution Time (C_i)
0	8	1
1	12	2
2	20	3
3	25	6

$$R_{2,0} = \max_{2<j<4}(C_j) = C_3 = 6$$

$$R_{2,1} = R_{2,0} + \sum_{m=0}^{m=1}\left(\left\lfloor \frac{R_{i,0}}{P_m} + 1 \right\rfloor C_m\right) = 6 + \left(\left\lfloor \frac{6}{8} + 1 \right\rfloor 1\right) + \left(\left\lfloor \frac{6}{12} + 1 \right\rfloor 2\right) = 6 + 1 + 2 = 9$$

$$R_{2,2} = R_{2,0} + \sum_{m=0}^{m=1}\left(\left\lfloor \frac{R_{i,1}}{P_m} + 1 \right\rfloor C_m\right) = 6 + \left(\left\lfloor \frac{9}{8} + 1 \right\rfloor 1\right) + \left(\left\lfloor \frac{9}{12} + 1 \right\rfloor 2\right) = 6 + 2 + 2 = 10$$

$$R_{2,\infty} = 10$$

Math Differences For Combined System

- **"combined" (informal term) = "interrupts + main loop"**
- **Back to the cyclic executive plus ISRs**
  - Main loop can be pre-empted (interrupted) by ISRs – consider this task N
  - ISRs don't have to wait for main loop completion…
    … but main loop does have to wait for ISRs!
- **Math for Response time**
  - ISR math – almost unchanged – but now have to worry about blocking time *B*
    - Main loop has to finish current instruction (what if it is a multiply instruction?)
    - Main loop might have interrupts disabled; B = maximum time for this to happen

$$\boxed{\begin{aligned} R_{i,0} &= \max\left[\max_{i<j<N}(C_j), B\right] &&; i < N-1 \\ R_{i,k+1} &= R_{i,0} + \sum_{m=0}^{m=i-1}\left(\left\lfloor \frac{R_{i,k}}{P_m} + 1\right\rfloor C_m\right) &&; i > 0 \end{aligned}}$$

Back To Main Loop Response Time…

- **Response time for main loop is time to complete a cycle**
  - If data changes just after "do_task1()" starts executing, have to assume wait until next start of "do_task1()" to do the new computation
  - In general, if we assume main loop is task N, response time is one main loop

$$\boxed{\begin{aligned} R_{N,0} &= C_N \\ R_{N,k+1} &= R_{N,0} + \sum_{m=0}^{m=N-1}\left(\left\lfloor \frac{R_{N,k}}{P_m} + 1\right\rfloor C_m\right) \end{aligned}}$$

  - This is same equation as earlier, but with cleaned up notation

Back To The Big Picture

- **We've been building up a framework for**
  … non-preemptive scheduling …
  - Tasks run to completion; also called **cooperative task scheduling**
  - When one task completes, task at next higher priority executes
  - Any time you have ISRs, probably this is the type of scheduling you need to know!

- **Scheduling summary for response time R_i**
  - You always have to wait for ***one*** initial blocking period
    - Often is the longest execution lower-priority task
    - Could be something else that sets interrupt mask flag
  - You have to wait for ***all*** higher priority tasks
    - And, even worse, some might execute multiple times!
  - Assumptions!
    - System doesn't get overloaded – task *m* completes before next time task *m* executes
    - Tasks are periodic and you know the worst-case period P_i
    - You know the worst-case compute time for each task C_i
    - You're willing to schedule for the worst case, perhaps leaving CPU idle in other cases

Why Do We Need More Than This?

- **Cyclic Exec can be enough**
  - Mostly used when CPU is so fast, everything can be run faster than external world changes

- **Background task plus ISRs commonly used**
  - Works as long as each ISR can be kept ***short***
  - Works as long as everything that needs to be "fast" can be put in ISR

- **But, here's the rub – Low Priority ISRs and Blocking Time**
  - Response time dominated by longest ISR, even if ***low priority***
  - Response time dominated by I mask being set in main program ("blocking")
  - So this only really works if interrupts are short – and main program can be slow

  - Problem if you need a complex ISR!
  - Problem if you need to disable interrupts!

  - But for now, let's look at how people usually make this work

Real Time System Pattern – Main Plus ISR

- **ISR does minimum possible work to service interrupt**
  - Main program loop processes data later, when there is time

```
// main program loop
for(;;)
{ <detailed service for device 0>
  <detailed service for device 1>
   …
  <detailed service for device N-1>
  <other background tasks>
}
// interrupt priority is in device order (#20 is ISR0)
void interrupt 20 handle_device0(void) { …… }
void interrupt 21 handle_device1(void) { …… }
…
void interrupt 23 handle_device<N-1>(void) { …… }
```

Example – Keeping Time Of Day

- **System might need time of day in hours, minutes, seconds**
  - Naïve approach – do the computation in the ISR
    - Requires division and modular arithmetic
    - The problem is that this slows ISR, increasing max response time
  - Here’s the “big-ISR” approach
    - (we’re going to ignore setup for TOI – you’ve seen this before)

```
// current time
volatile uint64 timer_val; // assume initialized to current time
volatile uint8  seconds, minutes, hours;
volatile uint16 days;

void interrupt 16 timer_handler(void) // TOI
{ TFLG2 = 0x80;
  timer_val += 0x10C6;  // 16 bits fraction; 48 bits intgr
   seconds = (timer_val>>16)%60;
   minutes = ((timer_val>>16)/60)%60;
   hours =   ((timer_val>>16)/(60*60))%24;
   days =    (timer_val>>16)/(60*60*24);
}
```

Keeping The Time Of Day ISR “Skinny”

```
volatile uint64 timer_val; // assume initialized to current time
volatile uint8  seconds, minutes, hours;
volatile uint16 days;

void main(void)
{ … initialization …
   for(;;)
   { update_tod();
     do_task1();
     do_task2();
   }
}

void update_tod()
{  DisableInterrupts(); // avoid concurrency bug
   timer_temp = timer_val>>16;
   EnableInterrupts();
   seconds = (timer_temp)%60;
   minutes = ((timer_temp)/60)%60;
   hours =   ((timer_temp)/(60*60))%24;
   days =    (timer_temp)/(60*60*24);
}

void interrupt 16 timer_handler(void) // TOI
{ TFLG2 = 0x80;
  timer_val += 0x10C6;  // 16 bits fraction; 48 bits intgr
}   // blocking time of ISR no longer includes division operations!
```

Want this here instead of at end of subroutine to minimize Blocking Time B

Skinny ISRs

- **General idea**
  - Move everything you can to a periodically run main routine
  - Keep only the bare minimum in the ISR
  - Usually amounts to storing info somewhere for main loop to process later
- **Advantages:**
  - Reduces blocking time of that ISR, improving response time
- **Disadvantages; issues:**
  - It only takes ***ONE*** long ISR to give bad blocking time for whole system!
    - So *all* the ISRs have to be skinny!
  - It feels like more work than writing long ISRs
    - (if you think that is work, try debugging a system with random timing failures!)

Deprecated Alternative – ISRs with CLI

- **If you have a long ISR, why not just re-enable interrupts?**

```
void interrupt 16 timer_handler(void) // TOI
{ TFLG2 = 0x80;
  timer_val += 0x10C6;  // 16 bits fraction; 48 bits intgr
#asm
  CLI  ; re-enable interrupts   ** BAD IDEA! **
#endasm
  seconds = (timer_val>>16)%60;
  minutes = ((timer_val>>16)/60)%60;
  hours =   ((timer_val>>16)/(60*60))%24;
  days =    (timer_val>>16)/(60*60*24);
}
```

- **What does this do?**
  - CLI – enables interrupts (same as EnableInterrupt() call)
  - In GCC use keyword volatile – tells compiler "don't move this instruction around"!!!

Why Is CLI A Really Bad Idea?

- **What it does if you are careful:**
  - Re-enables interrupts while ISR is still executing
  - RTI re-re-enables interrupts (so this still works OK)
  - Blocking time is now from start of ISR until CLI executes – not whole ISR
  - So, it is as if you had a shorter ISR
  - Makes sure that TOD is updated immediately, even in middle of main loop
- **So why is it a problem?** http://betterembsw.blogspot.com/2014/01/do-not-re-enable-interrupts-in-isr.html
  - Some current systems use just this approach, but it's a bad idea
  - Problem 1: what if interrupt re-triggers before end of ISR?
    - Need to make ISR re-entrant (more on this later) – notoriously easy to get wrong
    - If ISR can occur in bursts, overflowing stack
  - Problem 2: what if ISR is changing memory locations used by another ISR?
    - Very tricky to debug if multiple ISRs fight over resources and can be interrupted … and designers miss this kind of thing because ISRs aren't in main flow of code
  - Problem 3: causes priority inversion if lower priority interrupt hits
    - Lower priority ISR completes before higher priority ISR!
  - Bottom line – this approach has bitten designers too often; ***avoid it***

Review

- **Cyclic executive**
  - Put everything in one big main loop – OK if loop is fast and external world is slow
  - Scatter high-frequency tasks repeatedly throughout mainloop
  - Response time for cyclic exec – wait for loop to go all the way around
- **ISRs only**
  - Prioritized ISR response time includes: execute worst case blocking task, plus possibly multiple instances of higher priority ISRs
- **Hybrid Main Loop + ISRs**
  - Pretty much the same math, with main loop as task N
  - Avoid CLI in an ISR if possible – it's the Dark Side Of The Force
- **Overall – yes, we expect you to know these equations on your own!**
  - If you know the principles, the equations follow, but memorize if you have to
  - These equations are a really Good Thing to put on your test notes sheet

These equations are important:

$$R_{i,0} = \max\left[\max_{i<j<N}\left(C_j\right), B\right] \qquad ; i < N-1$$

$$R_{i,k+1} = R_{i,0} + \sum_{m=0}^{m=i-1}\left(\left\lfloor \frac{R_{i,k}}{P_m} + 1 \right\rfloor C_m\right) \qquad ; i > 0$$

$$R_{N,0} = C_N$$

$$R_{N,k+1} = R_{N,0} + \sum_{m=0}^{m=N-1}\left(\left\lfloor \frac{R_{N,k}}{P_m} + 1 \right\rfloor C_m\right)$$

Worst Case Interrupt Response Time
Draft, Fall 2007

Philip Koopman
Carnegie Mellon University

1. Overview:

Interrupt Service Routines (ISRs) are commonly used to provide fast response times to external events or timed events. Because the point of providing fast response is to meet deadlines, it is important to know the worst case execution time of multiple concurrent interrupts competing for processor resources. The usual scheduling theory math doesn't work that well for this case because most scheduling theory assumes preemptive task switching, while ISRs are usually written to be non-preemptive (i.e., interrupts remain masked while the ISR is running). This is an instance of the more general problem of determining the maximum response time for a prioritized, non-preemptive tasking environment.

2. Importance:

If only a single interrupt is used in a system, determining interrupt service latency is relatively easy. However, if multiple prioritized interrupts can occur, then some will be serviced quickly, and others will be serviced more slowly. There will be some worst-case situation in which lower priority interrupts will have to wait for one (or more) executions of all higher priority interrupts. Ensuring that the worst case latency of lower priority interrupts is fast enough to meet real time requirements is an important analysis issue. Unfortunately, it is often difficult or impossible to create worst-case situations in a testing situation, so analytic approaches must be used to make sure that testing doesn't miss a particularly bad timing problem.

The ISR response scenario is an instance of the more general situation of a prioritized cooperative scheduling tasking system. In such a system each task has a priority, but tasks run to completion (i.e., tasks are non-preemptive).

3. Graphical Approach

For this discussion, we assume that there is a collection of prioritized tasks that needs to be executed periodically. Those tasks could be ISRs, threads, processes, or any mixture of the above so long as there is a static total ordering of priority across all tasks (i.e., fixed task priority). This would be the case, for example, for prioritized ISRs which keep interrupts masked while executing, deferring any higher priority interrupt servicing until the currently executing ISR has completed. This is usually how prioritized interrupts are executed. (The exception is when a software developer explicitly re-enables interrupts during an ISR, but that is usually bad practice.)

First, let's work out an example graphically to understand what is involved. Consider the below example task set including prioritized interrupts, execution times, and periods:

ISR0 takes 5 msec to execute and occurs at most once every 15 msec
ISR1 takes 6 msec and occurs at most once every 20 msec
ISR2 takes 7 msec and occurs at most once every 100 msec
ISR3 takes 9 msec and occurs at most once every 250 msec
ISR4 takes 3 msec and occurs at most once every 600 msec

where ISR0 is the highest priority and ISR4 is the lowest priority. No ISR can preempt any other ISR. When an ISR completes execution, the highest priority ISR that is ready to execute will execute next. We assume that any underlying tasks don't disable interrupts. There are a number of other assumptions we are making to simplify this analysis, but those will be discussed later in the analytic approach section.

The first question we want to ask is, *what is the worst case latency for ISR2*? For example if ISR2 must complete within 50 msec of the time the interrupt is first requested, is there a case where that won't happen?

The problem is that ISR2 is not the only task running – other ISRs are competing for processor resources. A bad case is when another *lower* priority task than ISR2 has just started to run when ISR2 is triggered for execution. In particular, the worst case is when the task with the longest running time having lower priority than ISR2 has just started to run. For this task set that is ISR3 (ISR3 and ISR4 are both lower priority than ISR2, but ISR3 has a much longer run time of 9 msec compared to the 3 msec run time of ISR4).

Why did we pick a lower priority instead of a higher priority interrupt to start with? The reason is that in the worst case, the CPU will be unavailable for a while when an interrupt arrives, causing other interrupts to pile up before the one we are interested even has a chance to compete for CPU time. Because no interrupt with lower priority will execute after ISR2 becomes ready to run, selecting one with a lower priority adds more work to the tasks that must be completed before ISR2 can be started. We'll take all the higher priority interrupts into account shortly. But once ISR2 is ready to run, no lower priority interrupt can run, so only one such low priority interrupt need be considered, and the longest execution time one is the worst case.

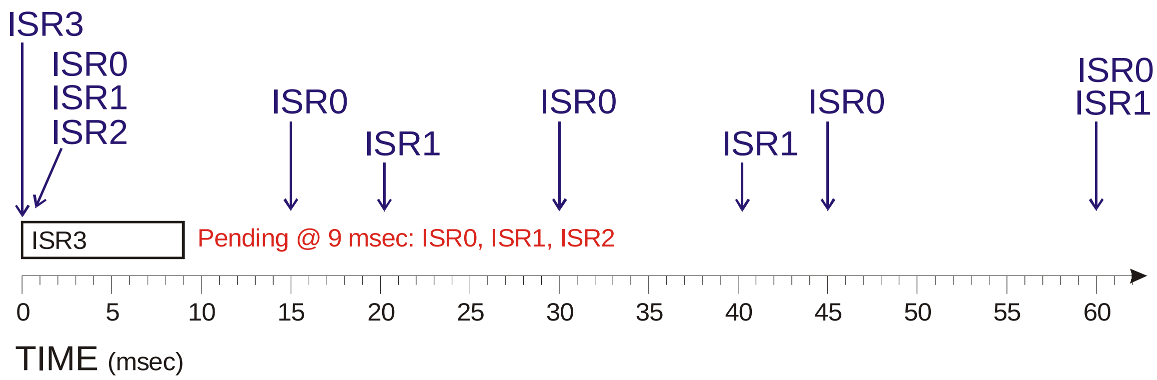


Figure 1. ISR3 Executes before ISR0, ISR1, and ISR2 are triggered.

Now we have a situation where ISR3 might get to run before ISR2 by just beating it to the CPU. Beyond that, it is possible that every higher priority interrupt than ISR2 starts just after ISR3 starts, but before ISR3 ends, so they could also get to run before ISR2 as well. Figure 1 shows this situation, with ISR3 starting to run, followed quickly by ISR0, ISR1, and ISR2 being triggered. The times at which ISR0 and ISR1 can be retriggered are also shown in Figure 1, since as we will find out they might have to be serviced one or more times before ISR2 finally gets a chance to run.

When ISR3 finishes executing at 9 msec, all three of ISR0, ISR1, and ISR2 are pending. Since ISR0 is the highest priority task, it goes first, followed by ISR1. While ISR1 is executing, ISR0 is triggered a second time at 15 msec, and ISR1 is triggered again at time 20 msec. Neither of these events disturbs the execution of ISR1 since it is non-preemptable (interrupts are masked while executing ISRs). This leads to a situation at time 20 where all three of ISR0, ISR1, and ISR2 are still pending (Figure 2).

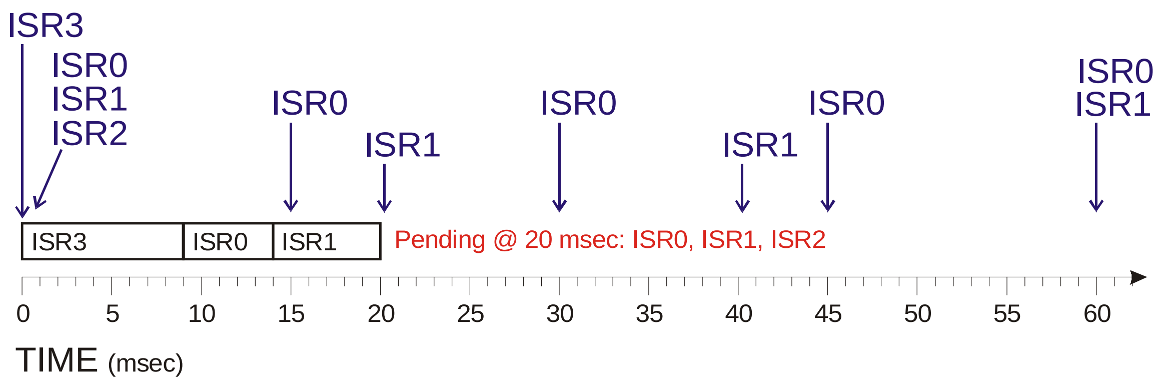


Figure 2. By the time ISR0 and ISR1 run once, they have been retriggered.

At 20 msec, ISR0 is the highest pending task, so it executes again, and is again followed by ISR1, at 25 msec. ISR0 then retriggers at 30 msec, but ISR1 has not yet triggered again. So at 31 msec when ISR1 ends, only ISR0 and ISR2 are pending (Figure 3).

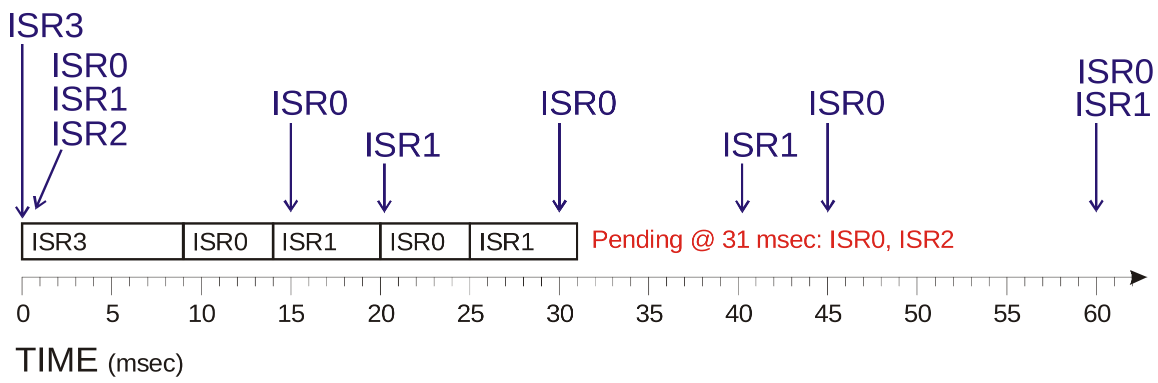


Figure 3. At 31 msec, ISR0 has retriggered and ISR2 is still pending, but it is not time for ISR1 yet.

At 31 msec ISR0 is still the highest priority interrupt pending, so it runs until 36 msec (Figure 4).

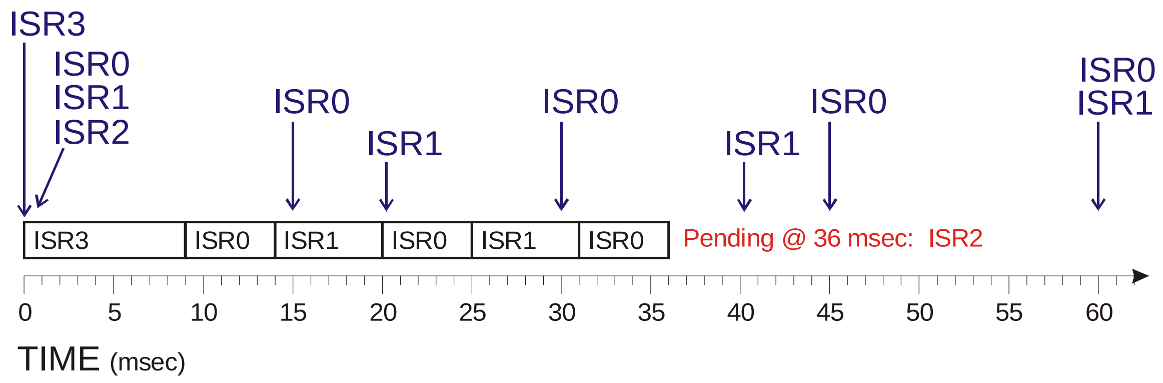


Figure 4. At 31 msec, ISR0 and ISR2 are pending, so ISR0 runs again.

At 36 msec, ISR0 has completed execution and is no longer pending (it won't be triggered again until 45 msec). Moreover, ISR1 is not due to run until 40 msec. This leaves ISR2 as the only task pending, so it starts execution and runs until 43 msec. By 43 msec ISR1 has retriggered, so it starts running (Figure 5), but does not interfere with the completion of ISR2 because interrupts are non-preemptable once started. Thus, in the worst case, ISR2 completes at 43 msec after it is triggered.

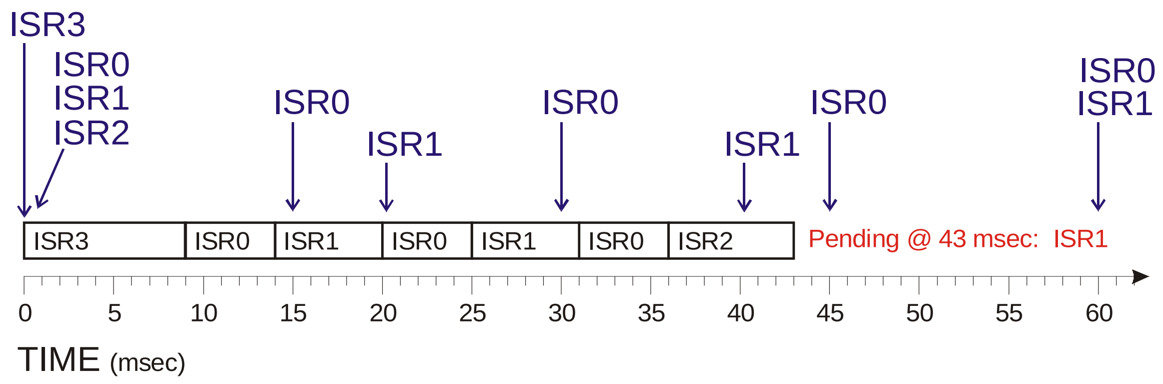


Figure 5. At 36 msec, ISR2 is the only task still pending, so it finally gets to execute.

4. Analytic Approach

Now that we have seen the types of complications that can arise when multiple tasks compete for CPU time, we can take a more rigorous, mathematical, approach to the analysis. In this section we'll create a set of equations that computes the worst case latency for any task in a set of tasks. These equations can be used to determine if each task in the set will meet its own particular deadline.

The following notation is used in the equations below:

- T_i : Task i
- R_i : Response time of Task i, which is the worst-case time between when T_i is ready to start executing and the time it actually starts execution.
- W_i : Completion time of Task i
- C_i : Computation time for one execution of T_i (worst case – largest possible C_i)

- P_i : Period for execution T_i (worst case – fastest possible P_i). If the task is aperiodic, then assume a P_i corresponding to shortest possible time between any two executions of T_i (i.e., reciprocal of worst case shortest inter-arrival time of task executions).
- D_i : Deadline for T_i
- B : Blocking time caused by background tasks that mask interrupts, or other dependencies.
- $\lfloor x \rfloor$: floor function; rounds x down to next lowest integer

The following assumptions are used in the equations below as a starting point:

- There are prioritized N tasks, numbered 0 through N-1, with Task i called T_i. In the previous example, each ISR handler was a task. Any other tasks running on the computer are referred to as background tasks.
- Background tasks interfere with Tasks 0 through N-1 only via disabling interrupts or task switching for some maximum blocking time B. (Blocking time was not shown in the preceding graphical example.)
- Tasks are statically prioritized, with Task 0 being the highest priority and Task N-1 being the lowest
- Each task T_i executes only when no other task with higher priority is ready to execute, then runs to completion without stopping (i.e., tasks are non-preemptable). If no task T_i is ready to execute, then background tasks are run until some task T is triggered to run.
- Any task T_i can and will preempt any background tasks, possibly with a delay caused by blocking time. (For example, ISRs preempt any non-ISR code.)
- Each task is triggered for execution no faster than once per stated period. Moreover, the period represents the worst-case minimum inter-arrival time between triggers for that task to execute. The period of each task may be different.
- The worst-case longest compute time for each task is known and used in the calculations.
- The deadline for each task is known, and is less than or equal to that task's period.
- The cost of changing tasks (e.g., processing an interrupt and corresponding RTI instruction) is accounted for in the worst-case compute time.

The values we are interested in finding are the completion times of all tasks. For a system to perform properly, all tasks must complete their work W_i at or before the applicable deadline D_i. So, the ultimate goal is to ensure that:

(1) $$\forall_i \left(W_i \le D_i\right)$$

Which states: for all values of *i*, the completion time of Task i is less than or equal to the deadline of Task i (i.e., all tasks complete before their deadline).

The completion time of a task has two components: the time spent waiting to start execution (the response time R_i) and the time spent actually doing the work of the task (the computation time C_i).

(2) $$W_i = R_i + C_i$$

In the systems we're looking at, tasks are non-preemptable, so once the computation of a task starts, that task runs to completion. Thus, C_i is a known constant value. But, R_i is trickier, because it must take into account the fact that Task i has to wait for all higher priority tasks to execute and also wait for any blocking time. For example, if Task 4 is an ISR, that ISR can't execute until any interrupt masking in the main program is completed (i.e., blocking time B) and all higher priority interrupts 0, 1, 2, and 3 execute at least one time (because in the worst case all four of those interrupts were triggered just after the beginning of the blocking time – such a small delay that we will just be conservative and consider it to be zero elapsed time in our equations).

Accounting for blocking time starts the build-up of equations to obtain R_i:

(3) $R_i \geq B$

By this we mean that Ri is at least as long as the blocking time, but possibly longer.

Now let us consider Task 0. Is B the only factor that could delay the start of execution? Even though this is the highest priority task in the system, there is something else that can delay it. The other factor is some other task with a lower priority that has already begun execution, because tasks are non-preemptable (once started, they run to completion). The worst case is that the task with the longest possible computation time has just started execution, and must complete before Task 0 can run. In other words, a lower priority task can delay execution of a higher priority task because it is allowed to run to completion. This, in effect, is a different form of blocking. In general, for Task i, it is possible for some task with a higher task number (i.e., lower priority) to be executing, delaying the start of Task i. Thus,

(4) $$R_i \geq \max_{i<j<N}\left(C_j\right)$$
$$R_{N-1} \geq 0$$

This means that the response time for Task i must be at least as bad as the worst case wait caused by the longest computation time of any Task j with a lower priority than Task i. Task i itself is not considered, because we assume that the deadline for each task is longer than its period, so Task i must have completed execution before it attempts to execute again. For task N-1, which is the lowest priority task that isn't a background task, this delay is zero, since there is no lower priority task to get in the way (but, even this task is subject to blocking time from the background tasks).

Next, we combine the two starting factors of B and maximum C_j to get an initial lower bound on response time. But rather than adding them, we can simply take the maximum of the two, because both situations can't happen at the same time. Consider the two possible situations.

Situation (1): If a task has to wait for blocking time B, then that means a task T_i isn't already running (because blocking can only occur due to a task other than Tasks 1..N-1 executing). If that is the case, as soon as blocking has finished, the highest priority task will begin executing as soon as blocking is over. This makes it impossible for a task with lower priority than Task i to delay the start of task i after blocking. If Task i isn't ready to execute when blocking is completed, then the blocking time hasn't delayed its response time, since it wasn't ready to run.

Situation (2): If Task j, with lower priority than Task i, is already executing, then blocking can't occur, because when Task j completes, Task i (or some task with higher priority) will immediately start executing rather than the background tasks. The background tasks that can cause blocking won't resume execution until all prioritized tasks, including Task i, complete execution.

So, let's define the effective blocking time B' as:

(5) $$B'_i = \max\left[\max_{i<j<N}\left(C_j\right), B\right] \quad ; \quad i < N-1$$
$$B'_i = B \quad ; \quad otherwise$$

This means that the response time for Task i is bounded by an effective blocking time B', which is the longest lower priority task that might execute, or the blocking time B. Because there is no lower priority task than Task N-1, then blocking time B is the only issue for that particular task.

For Task 0, our response time calculation is done. Because there are no higher priority tasks, Task 0 will run to completion once the effective blocking effect has passed (either waiting for background task blocking B or the longest lower priority task to complete).

(6) $$R_0 = B'_0$$

The next factor in response time calculations is that higher priority tasks can execute before lower priority tasks. In the worst case, Task i will have to wait for every possible Task m with higher priority to execute at least once. From this point on, the response time will have to be computed iteratively to account for the fact that enough time may pass for high priority tasks to re-trigger.

We'll use the notation $R_{i,k}$ to represent the *k*th iteration of the computation for R_i, with the computation iterated by increasing k until the answer converges to a final value. To keep things simple, and conform to the graphic approached used previously, we start the iteration with the effective blocking value B':

(7) $$R_{i,0} = B'_i$$

Next, we have to account for the execution time of all tasks with higher priority than Task i, because it is possible all of them triggered just as Task i was triggering. The number of times a particular Task m executes in time T is one more than the rounded-down (integer floor function) number of times the response time $R_{i,k}$ can be divided by the period of Task m:

(8) $$executions_m = \left\lfloor \frac{T}{P_m} + 1 \right\rfloor$$

For example, with a period of 7 and an elapsed time of 22, a task could have been triggered not 22/7 = 3.14 times, but rather that number rounded down, which is 3, plus 1 to account for the fact the task must assumed to have been triggered at time zero, which gives a total of 4 times (i.e., at times 0, 7, 14, and 21 msec). (Note that a ceiling function might seem attractive instead of the floor function. But, the ceiling function doesn't quite work if a response time is an exact multiple of a period.)

Once the number of executions is known, the amount of delay that higher priority Task m causes to the waiting Task i by the time Task i is ready to run is the number of executions of Task m that have taken place by R_i times the computation time of Task m:

(9) $$delay_m = executions_m C_m = \left\lfloor \frac{R_i}{P_m} + 1 \right\rfloor C_m$$

The amount of time that is taken by each execution is the task's computation time C$_i$. Therefore, the total amount of waiting time for Task i caused by waiting for higher priority tasks is the sum across all those tasks:

(10) $$R_i \geq \sum_{m=0}^{m=i-1} \left\lfloor \frac{R_i}{P_m} + 1 \right\rfloor C_m \quad ; \quad i > 0$$

We still need to account for the initial effective blocking time before any of those high priority tasks can execute, so the complete equation is:

(11) $$R_i = B'_i + \sum_{m=0}^{m=i-1} \left(\left\lfloor \frac{R_i}{P_m} + 1 \right\rfloor C_m \right) \quad ; \quad i > 0$$

But, here's the tricky part. The amount of time during which other tasks can execute depends on the time spent waiting – it is a recursive equation with R$_i$ appearing on both sides. In this case, we can break the recursion by simply using an iterative evaluation, where we keep re-evaluating R$_i$ for longer and longer times until the result converges to a final value. (If the result doesn't converge, that means Task i will never execute in the worst case.)

(12) $$R_{i,k+1} = B'_i + \sum_{m=0}^{m=i-1} \left(\left\lfloor \frac{R_{i,k}}{P_m} + 1 \right\rfloor C_m \right) \quad ; \quad i > 0$$

The response time is the result of iterating the above until it converges, which is obtained by taking the limit of R$_{i,k}$ as k approaches infinity. As a practical matter the process only needs to be repeated until the same answer is obtained on two successive iterations.

(13) $$R_i = \lim_{k \to \infty} \left(R_{i,k} \right)$$

The worst case completion time W$_i$ is then the worst case response time plus the execution time of Task i:

(14) $$W_i = R_i + C_i = \lim_{k \to \infty} \left(R_{i,k} \right) + C_i$$

As a reminder, we are assume the tasks are non-preemptable, so it is not possible for another task to interrupt the execution of Task i once it has started.

This completes all the pieces we need. To recap, below is the final set of working equations:

$$B'_i = \max\left[\max_{i<j<N}(C_j), B\right] \quad ; \quad i < N-1$$

$$R_{i,0} = B'_i \quad ; \quad i > 0$$

$$R_{i,k+1} = B'_i + \sum_{m=0}^{m=i-1}\left(\left\lfloor \frac{R_{i,k}}{P_m} + 1 \right\rfloor C_m\right) \quad ; \quad i > 0$$

$$D_i \geq W_i = \lim_{k\to\infty}(R_{i,k}) + C_i$$

With the following equations applying instead of the above for some special cases:

$$B'_{N-1} = B$$

$$R_0 = B'_0$$

Figure 6. Summary of equations.

5. Examples

After all this, we can get the answer to whether tasks will meet their deadlines by computing W_i for all tasks. Let's do this using the example from the previous graphical analysis and see how the equations work.

Let us revisit the previous example and see if the analytic approach yields the same result as the graphical approach. The example we used was:

N=5
B=0

ISR0 takes 5 msec and occurs at most once every 15 msec;	$C_0 = 5$; $P_0 = 15$
ISR1 takes 6 msec and occurs at most once every 20 msec;	$C_1 = 6$; $P_1 = 20$
ISR2 takes 7 msec and occurs at most once every 100 msec;	$C_2 = 7$; $P_2 = 100$
ISR3 takes 9 msec and occurs at most once every 250 msec;	$C_3 = 9$; $P_3 = 250$
ISR4 takes 3 msec and occurs at most once every 600 msec;	$C_4 = 3$; $P_4 = 600$

5.1. B=0 example

For B=0, let's find the worst case completion time of ISR2, which is W_2.

$$B'_2 = \max\left[\max_{2<j<5}(C_j), B\right] = \max\left[\max(C_3, C_4), B\right] = \max[9,3,0] = 9$$

$$R_{2,0} = B'_2 = 9$$

Given this starting point (which corresponds to Figure 1), we us $R_{2,0}$=9 to iterate $R_{i,k}$:

$$R_{2,1} = B'_2 + \sum_{m=0}^{m=1}\left(\left\lfloor \frac{R_{i,k}}{P_m} + 1 \right\rfloor C_m\right) = B'_2 + \left\lfloor \frac{R_{2,0}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,0}}{P_1} + 1 \right\rfloor C_1 =$$

$$9 + \left\lfloor \frac{9}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{9}{20} + 1 \right\rfloor 6 = 9 + 1 \cdot 5 + 1 \cdot 6 = 9 + 5 + 6 = 20$$

Note that $P_{2,1}$ is 20 msec, which is the same result as Figure 2. From this, it becomes evident that the iterative equation is doing the same thing mathematically that we did graphically in the previous approach.

$$R_{2,2} = B'_2 + \left\lfloor \frac{R_{2,1}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,1}}{P_1} + 1 \right\rfloor C_1 = 9 + \left\lfloor \frac{20}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{20}{20} + 1 \right\rfloor 6 = 9 + 10 + 12 = 31$$

This iteration brings us to 31 msec, corresponding to the situation shown in Figure 3.

$$R_{2,2} = B'_2 + \left\lfloor \frac{R_{2,1}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,1}}{P_1} + 1 \right\rfloor C_1 = 9 + \left\lfloor \frac{31}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{31}{20} + 1 \right\rfloor 6 = 9 + 15 + 12 = 36$$

This iteration brings us to 36 msec, corresponding to the situation shown in Figure 4 in which all ISRs with higher priority than ISR2 have just finished execution.

$$R_{2,3} = B'_2 + \left\lfloor \frac{R_{2,2}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,2}}{P_1} + 1 \right\rfloor C_1 = 9 + \left\lfloor \frac{36}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{36}{20} + 1 \right\rfloor 6 = 9 + 15 + 12 = 36$$

Because iteration $R_{2,3}$ hasn't changed compared to $R_{2,2}$, we can terminate the computation and know that additional iterations won't change the answer from 36. This gives us a time to completion of:

$$W_2 = \lim_{k \to \infty}(R_{2,k}) + C_2 = 36 + 7 = 43$$

which is 43 msec, the same answer shown in Figure 5 and is the worst case completion time. If the deadline were 50 msec, this task would always be able to meet its deadline under the given assumptions.

5.2. B=13 Example

As an example of what happens when the effective blocking time is dominated by background task blocking time rather than lower priority tasks, consider what happens when B=13 msec instead of 0 msec:

$$B'_2 = \max\left[\max_{2<j<5}(C_j), B\right] = \max\left[\max(C_3, C_4), B\right] = \max[9,3,13] = 13$$

$$R_{2,0} = B'_2 = 13$$

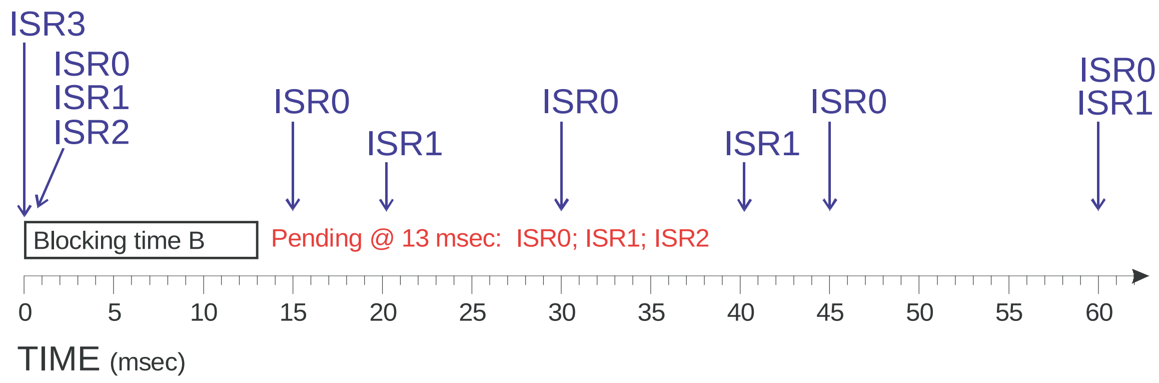


Figure 7. B=13 at time 13 msec.

$$R_{2,1} = B'_2 + \sum_{m=0}^{m=1}\left(\left\lfloor \frac{R_{i,k}}{P_m} + 1 \right\rfloor C_m\right) = B'_2 + \left\lfloor \frac{R_{2,0}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,0}}{P_1} + 1 \right\rfloor C_1 =$$

$$13 + \left\lfloor \frac{5}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{5}{20} + 1 \right\rfloor 6 = 13 + 5 + 6 = 24$$

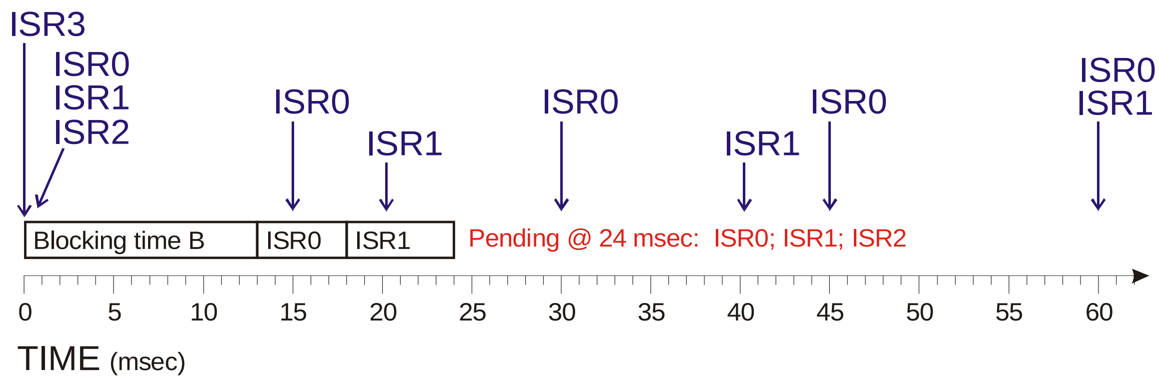


Figure 8. B=13 at time 24 msec.

$$R_{2,2} = B'_2 + \left\lfloor \frac{R_{2,1}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,1}}{P_1} + 1 \right\rfloor C_1 = 13 + \left\lfloor \frac{24}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{24}{20} + 1 \right\rfloor 6 = 13 + 10 + 12 = 35$$

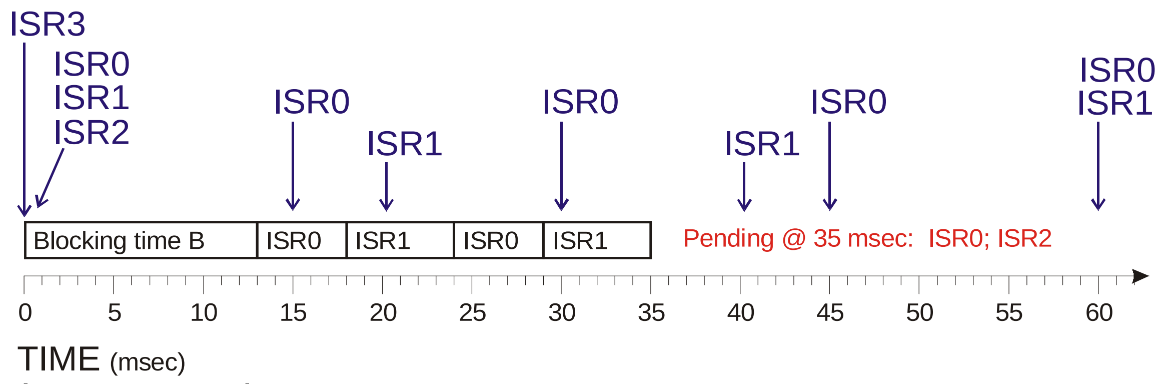


Figure 9. B=13 at time 35 msec.

$$R_{2,3} = B'_2 + \left\lfloor \frac{R_{2,2}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,2}}{P_1} + 1 \right\rfloor C_1 = 13 + \left\lfloor \frac{35}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{35}{20} + 1 \right\rfloor 6 = 13 + 15 + 12 = 40$$

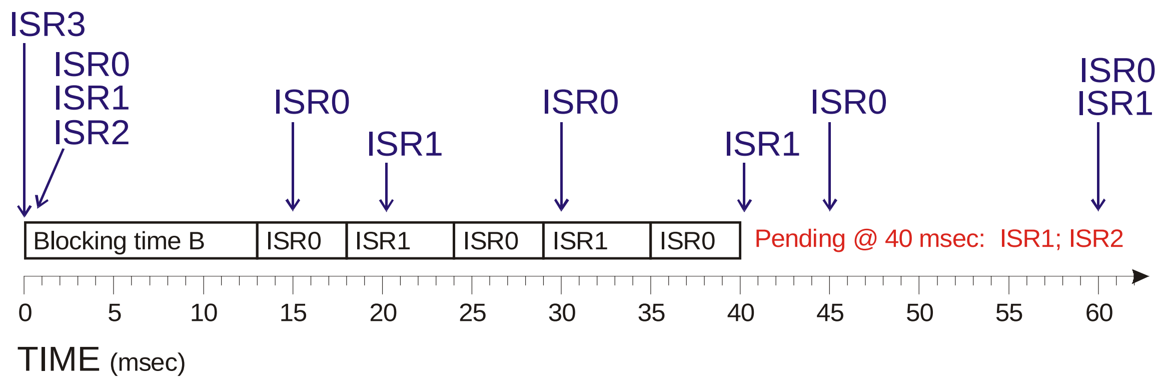


Figure 10. B=13 at time 40 msec.

$$R_{2,4} = B'_2 + \left\lfloor \frac{R_{2,3}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,3}}{P_1} + 1 \right\rfloor C_1 = 13 + \left\lfloor \frac{40}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{40}{20} + 1 \right\rfloor 6 = 13 + 15 + 18 = 46$$

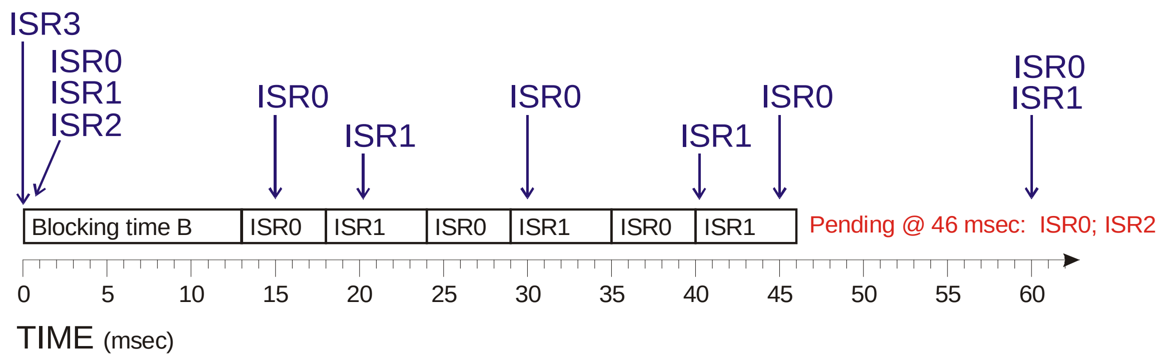


Figure 11. B=13 at time 46 msec.

$$R_{2,5} = B'_2 + \left\lfloor \frac{R_{2,4}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,4}}{P_1} + 1 \right\rfloor C_1 = 13 + \left\lfloor \frac{46}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{46}{20} + 1 \right\rfloor 6 = 13 + 20 + 18 = 51$$

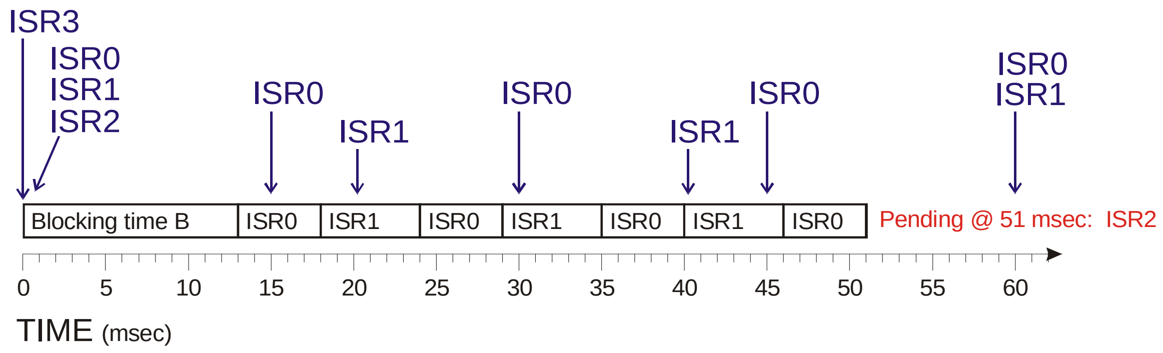


Figure 12. B=13 at time 51 msec.

$$R_{2,5} = B'_2 + \left\lfloor \frac{R_{2,5}}{P_0} + 1 \right\rfloor C_0 + \left\lfloor \frac{R_{2,5}}{P_1} + 1 \right\rfloor C_1 = 13 + \left\lfloor \frac{51}{15} + 1 \right\rfloor 5 + \left\lfloor \frac{51}{20} + 1 \right\rfloor 6 = 13 + 20 + 18 = 51$$

At this point the computation has converged, so we know that ISR2 will start execution at time 51 msec.

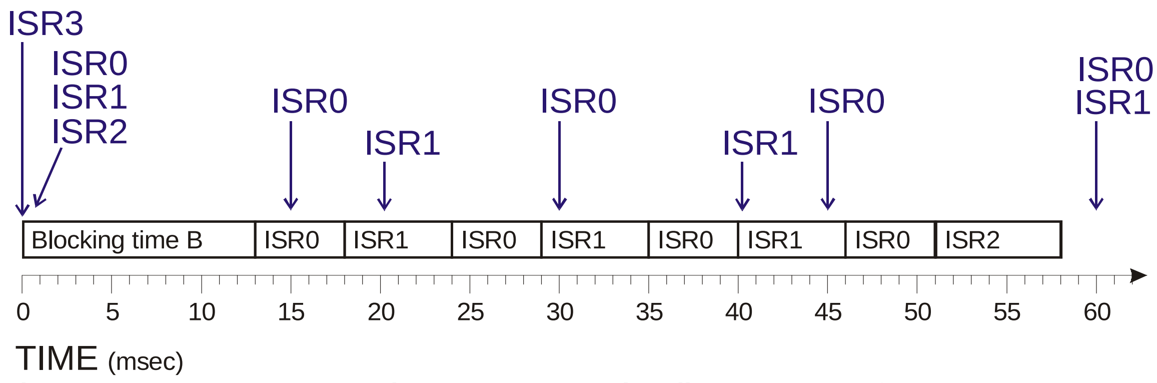


Figure 13. ISR2 executes starting at 51 msec and ending at 58 msec for B=13.

$$W_2 = \lim_{k \to \infty} (R_{2,k}) + C_2 = 51 + 7 = 58$$

Thus the graphical and analytic techniques both arrive at the same answer in the same way, and ISR2 has a worst-case execution time of 58 msec for this particular case.

5.3. Other Examples

As further exercises, the reader should confirm the following results both graphically and analytically for this example task set with various values of B:

B=0	B=2	B=4	B=12	B=13
W_0 = 14	W_0 = 14	W_0 = 14	W_0 = 17	W_0 = 18
W_1 = 20	W_1 = 20	W_1 = 20	W_1 = 28	W_1 = 29
W_2 = 43	W_2 = 43	W_2 = 43	W_2 = 46	W_2 = 58
W_3 = 46	W_3 = 46	W_3 = 47	W_3 = 66	W_3 = 67
W_4 = 57	W_4 = 59	W_4 = 61	W_4 = 91	W_4 = 92

As an additional example, the graphical results showing the timing for the B=2 case are below:

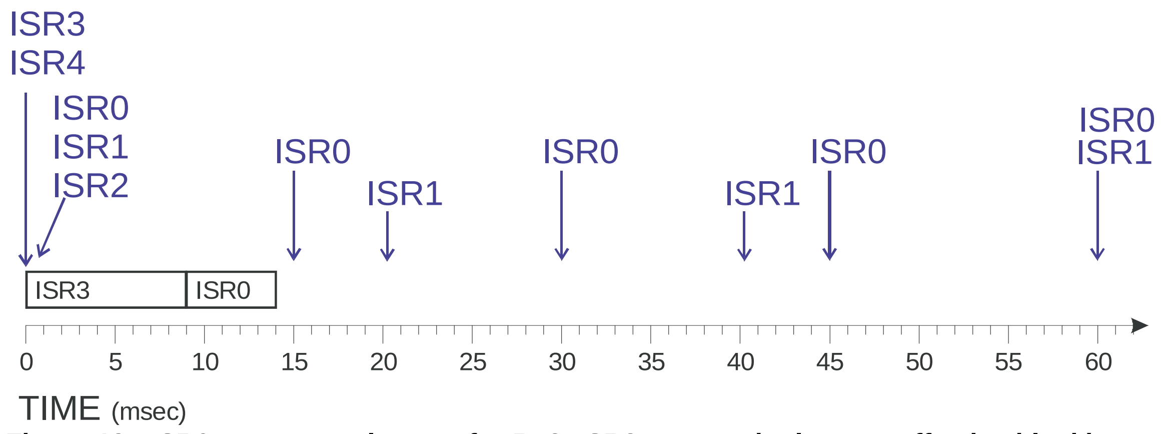


Figure 13. ISR0 worst case latency for B=2. ISR3 causes the longest effective blocking time.

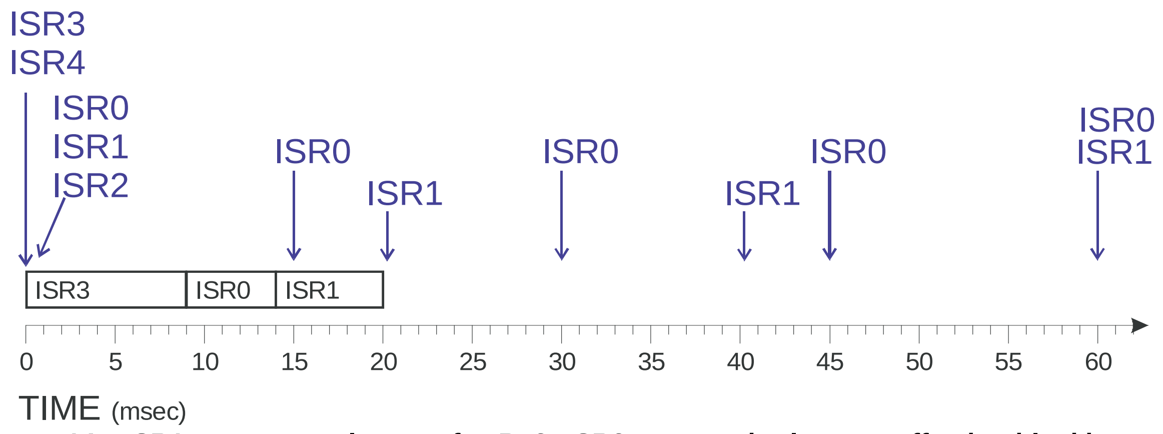


Figure 14. ISR1 worst case latency for B=2. ISR3 causes the longest effective blocking time.

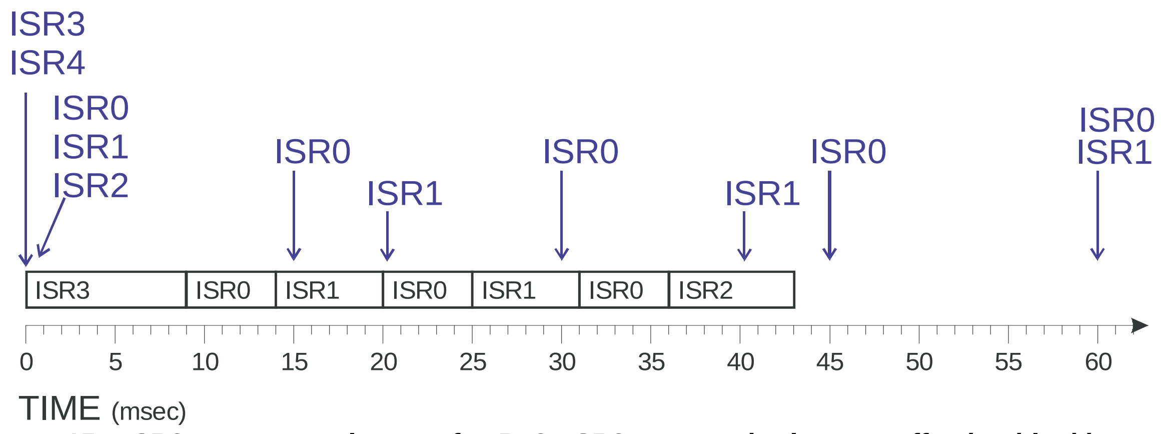


Figure 15. ISR2 worst case latency for B=2. ISR3 causes the longest effective blocking time.

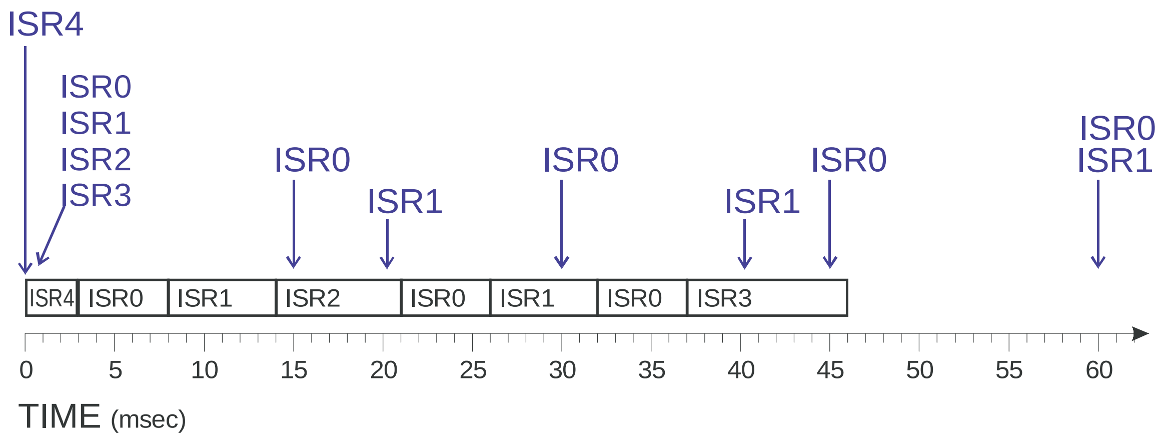


Figure 16. ISR3 worst case latency for B=2. ISR4 causes the longest effective blocking time.

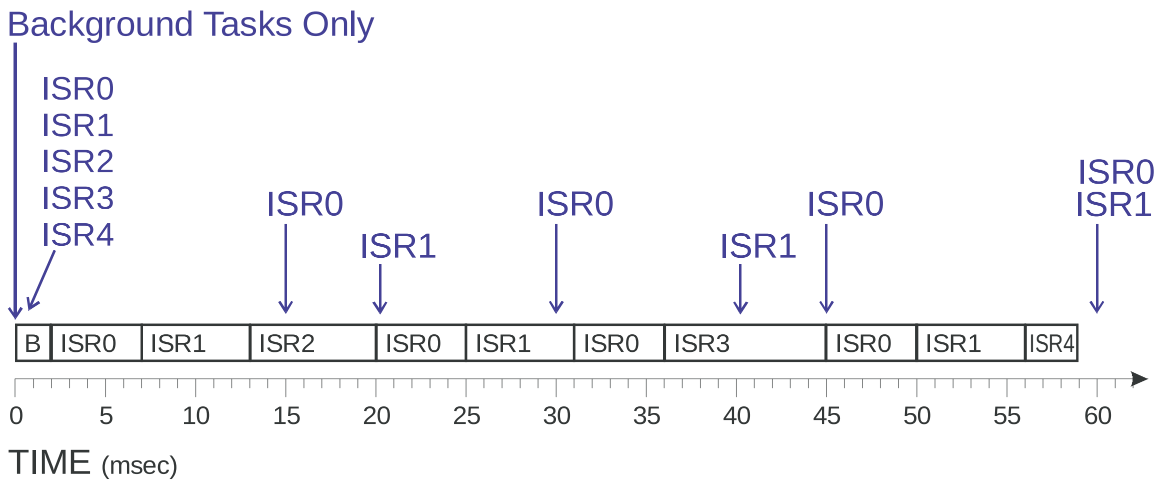


Figure 17. ISR4 worst case latency for B=2. B is the longest effective blocking time because there are no lower priority interrupts to process.

6. More Information

The more generalized problem includes computing execution times for both preemptive tasks running under an operating system and non-preemptive ISRs. A description of the math for that more general case can be found in: Y. Wang, M. Saksena, Scheduling fixed-priority tasks with preemption threshhold, *IEEE International Conference on Real-Time Computing Systems and Applications*, December 1999.

Thanks to Jen Morris Black for her research work in this area.

18-348 Spring 2016
Mid-Semester Informal Course Feedback

The official feedback systems don't give us information in time to make mid-course corrections. So we've developed this simple feedback form to make sure you get a chance to influence how the remainder of the course is run. You can put your name on the sheet if you want a personal reply, but we'll read all the feedback carefully even if it is anonymous.

1. What is the one most important thing you'd like to see changed about this course between now and the end of the semester? (Some things we can change and some we can't, but most years someone has an idea we can implement right away. So we will seriously consider what you say, and change things that we can do without breaking other aspects of the course.)

2. What is the thing you like most about this course? Do you want to see us do more of it, or is the extent about right already?

3. Any other comments on the course?