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.

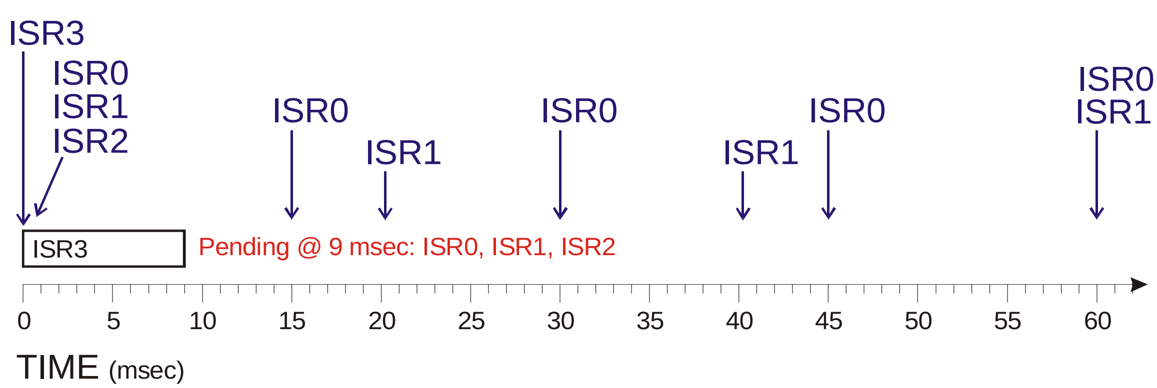

Figure 1. ISR3 Executes before ISR0, ISR1, and ISR2 are triggered.

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).

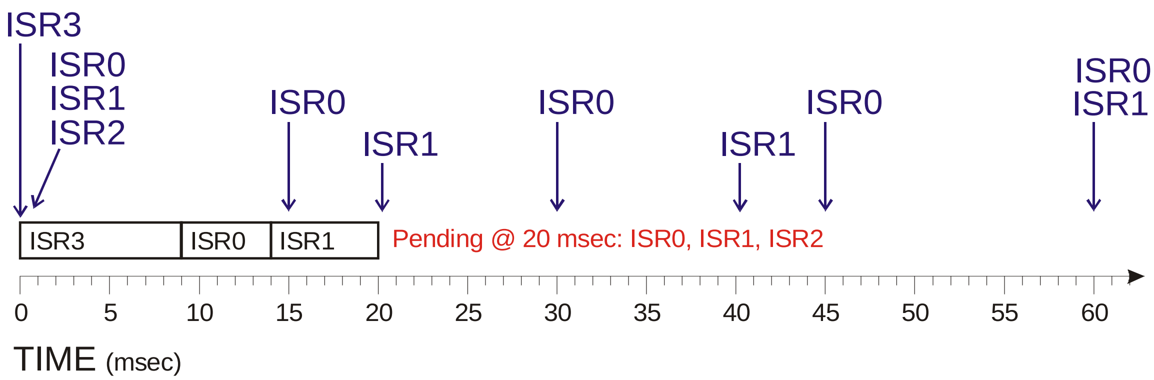

Figure 2. By the time ISR0 and ISR1 run once, they have been retriggered.

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).

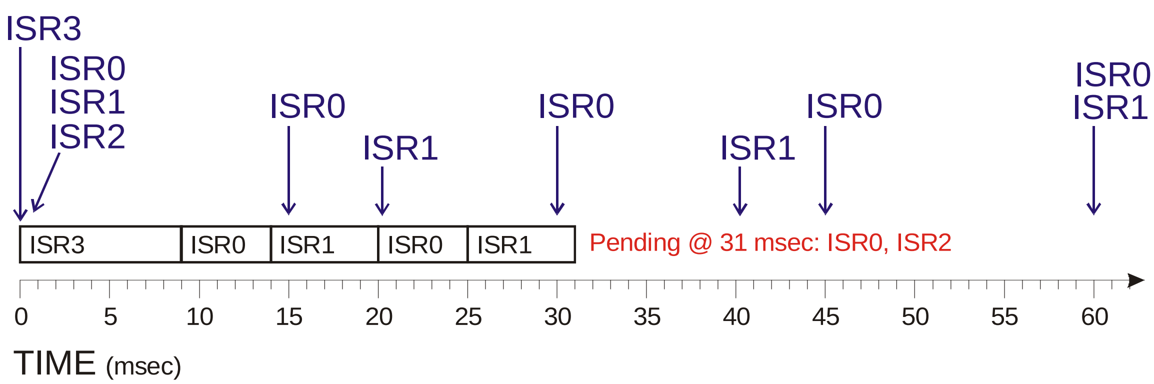

Figure 3. At 31 msec, ISR0 has retriggered and ISR2 is still pending, but it is not time for ISR1 yet.

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

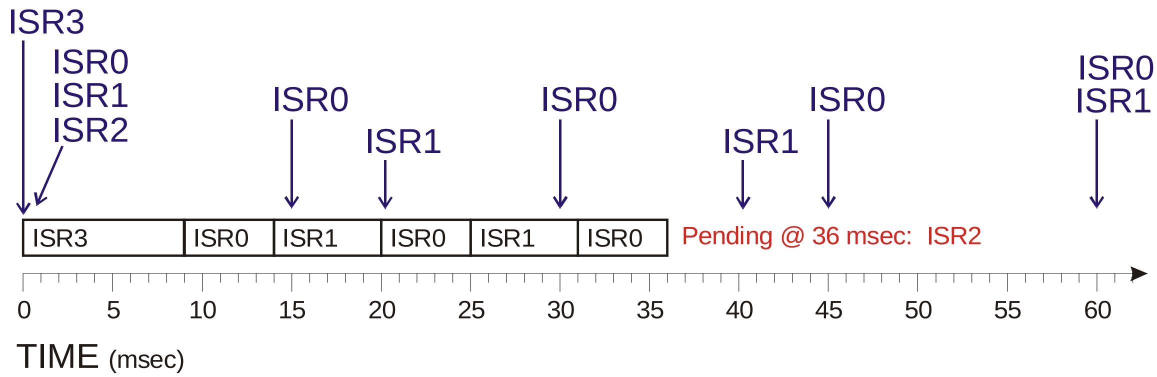

Figure 4. At 31 msec, ISR0 and ISR2 are pending, so ISR0 runs again.

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.

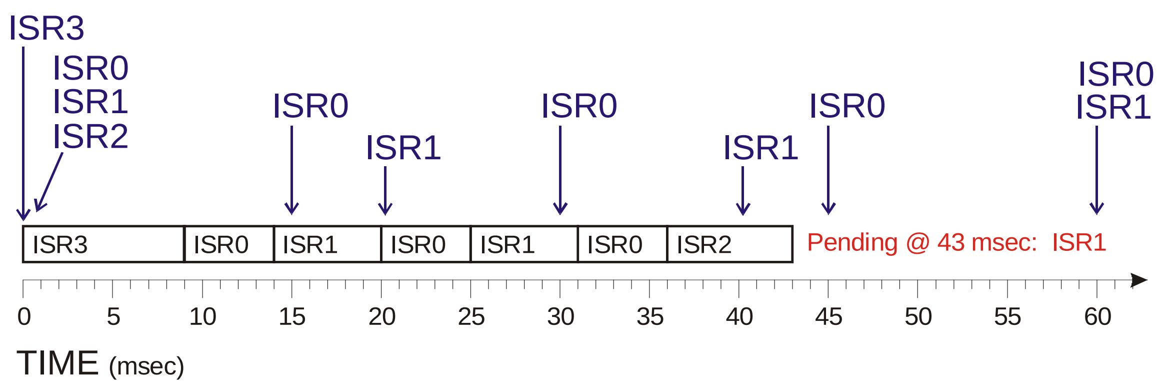

Figure 5. At 36 msec, ISR2 is the only task still pending, so it finally gets to execute.

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:

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

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:

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

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.)

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

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.

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

The worst case completion time W_i is then the worst case response time plus the execution time of Task i:

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

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:

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

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

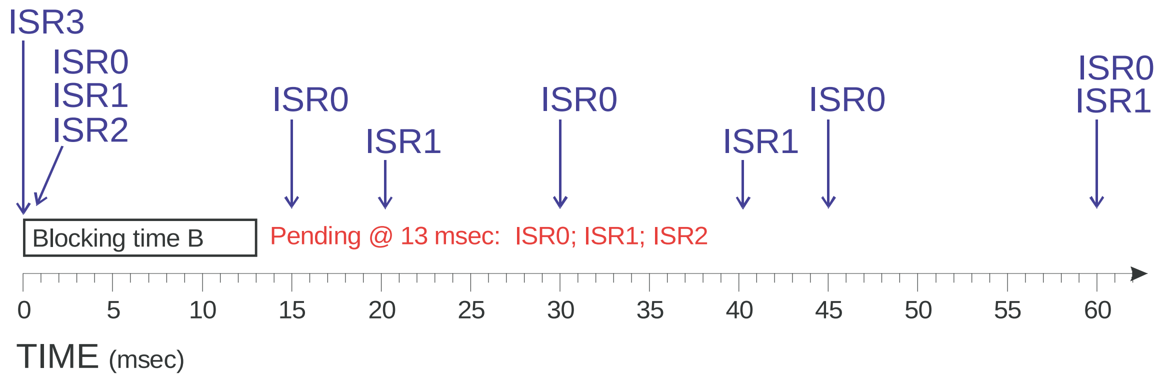

Figure 7. B=13 at time 13 msec.

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

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

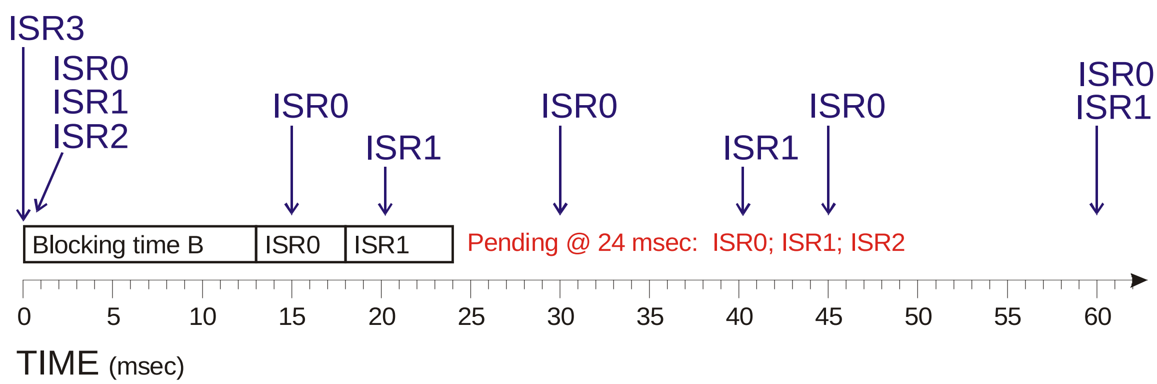

Figure 8. B=13 at time 24 msec.

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

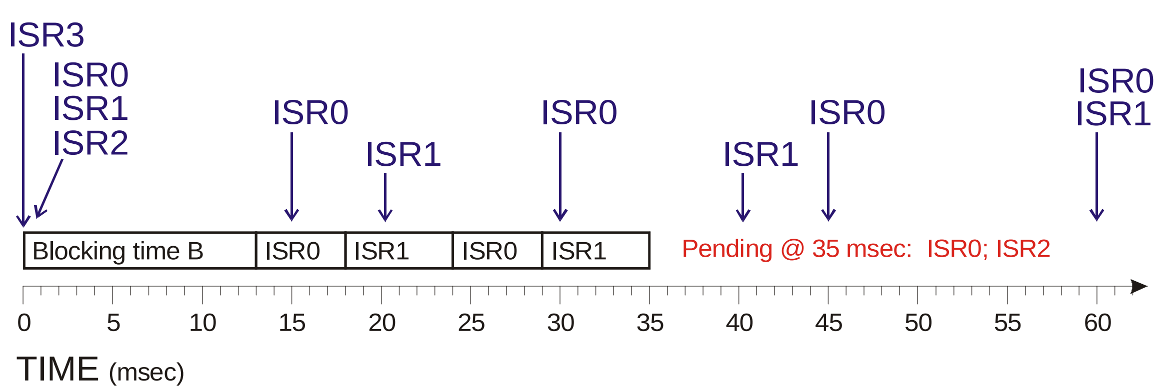

Figure 9. B=13 at time 35 msec.

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

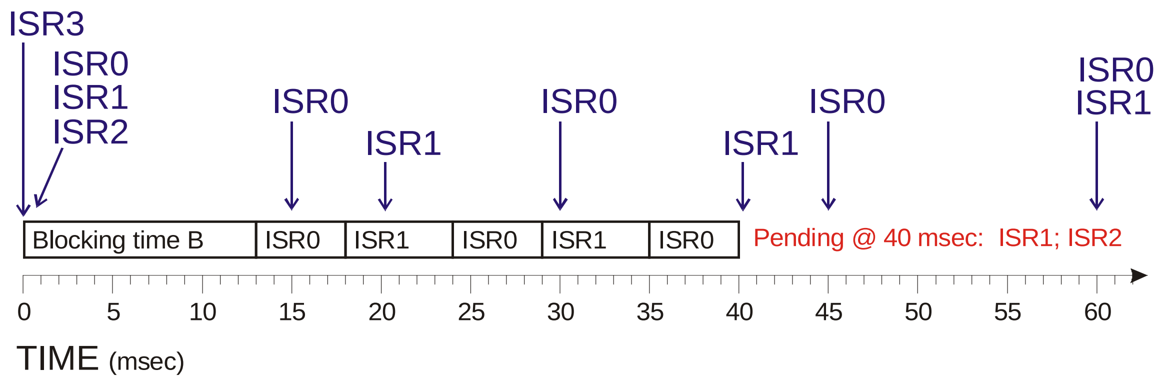

Figure 10. B=13 at time 40 msec.

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

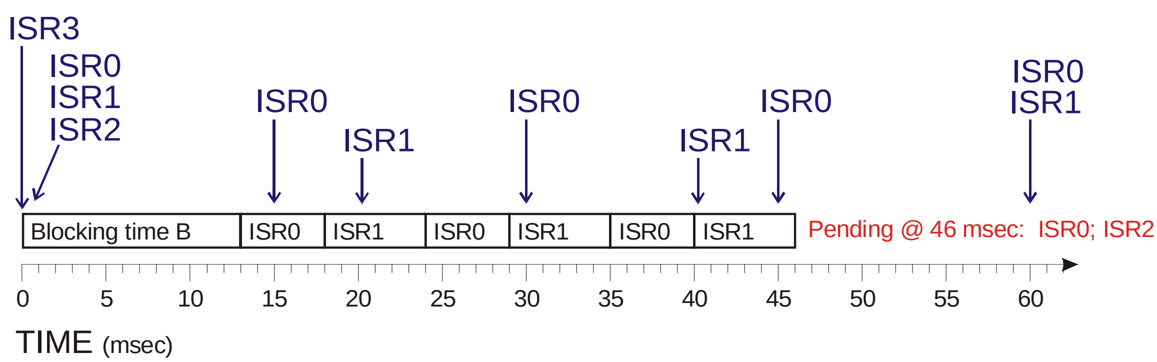

Figure 11. B=13 at time 46 msec.

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

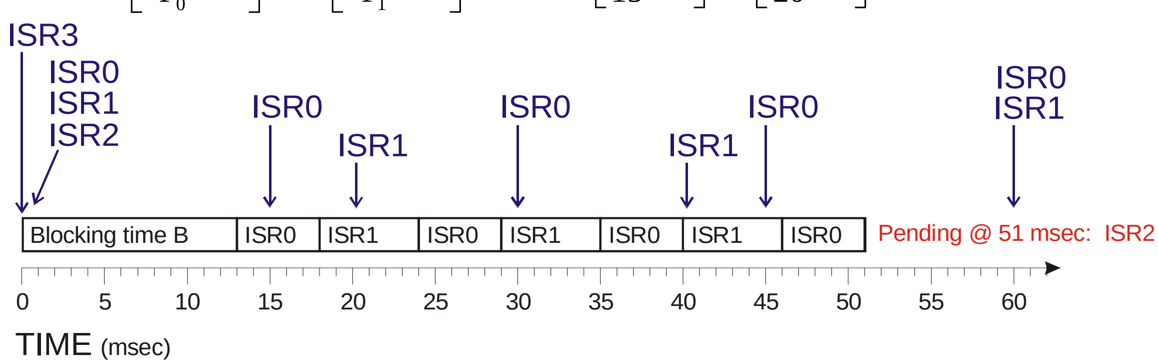

Figure 12. B=13 at time 51 msec.

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

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

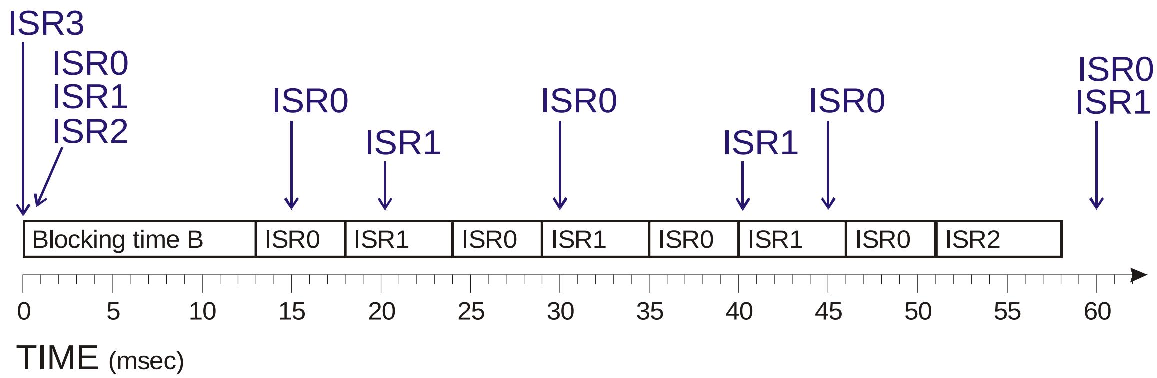

Figure 13. ISR2 executes starting at 51 msec and ending at 58 msec for B=13.

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

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:

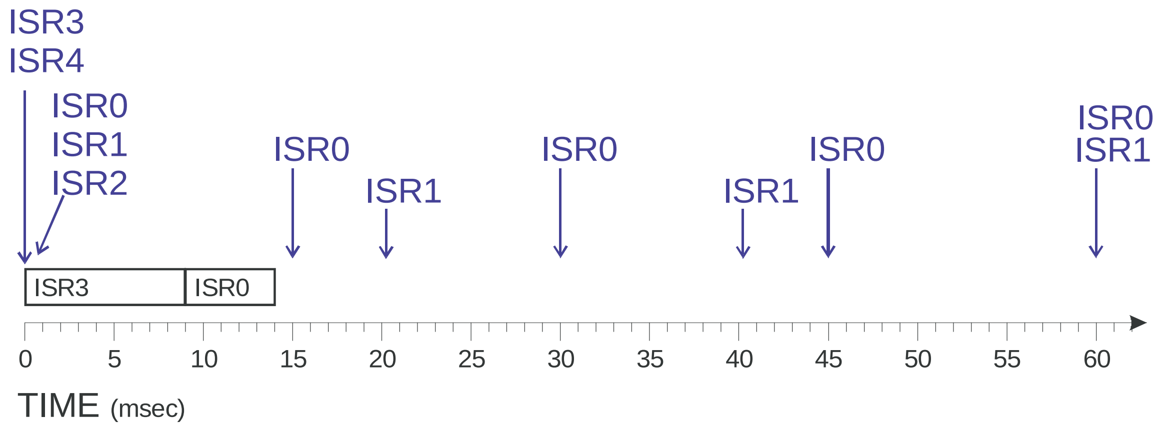

Figure 13. ISR0 worst case latency for B=2. ISR3 causes the longest effective blocking time.

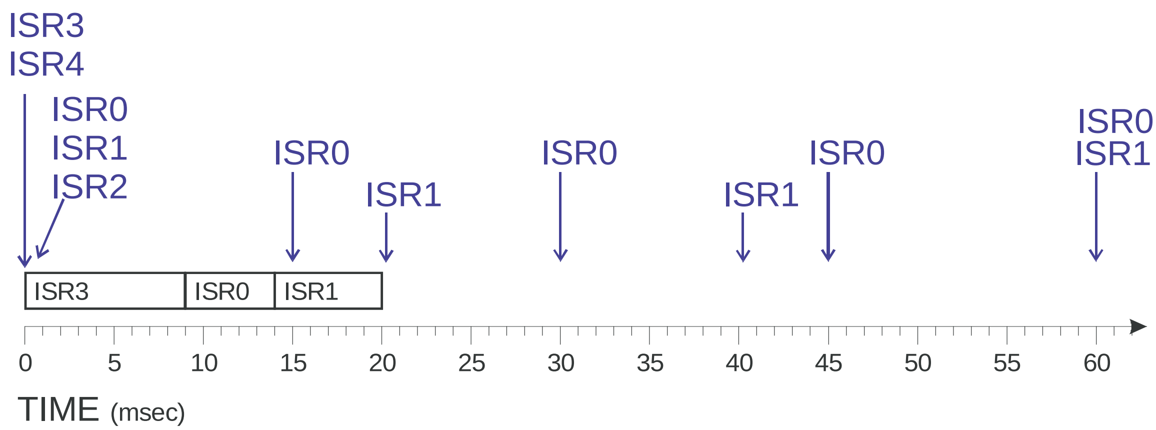

Figure 14. ISR1 worst case latency for B=2. ISR3 causes the longest effective blocking time.

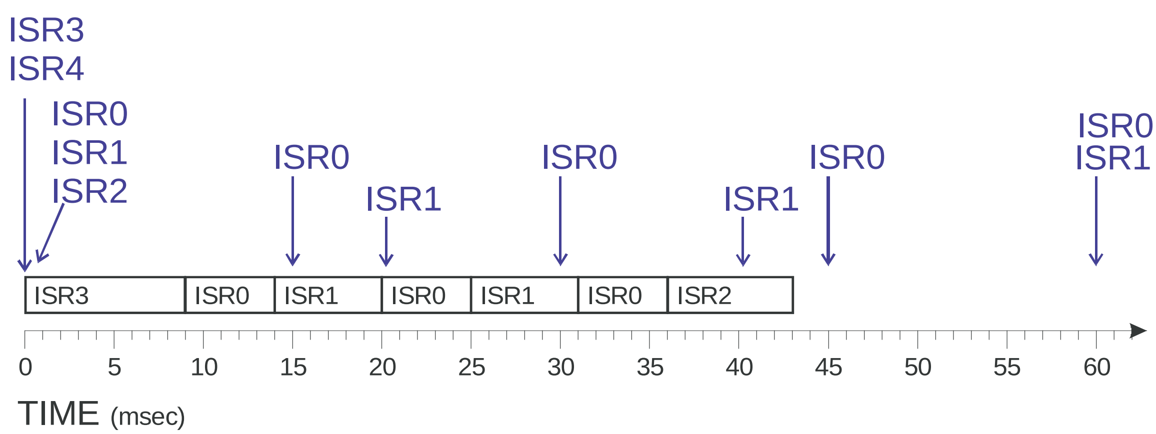

Figure 15. ISR2 worst case latency for B=2. ISR3 causes the longest effective blocking time.

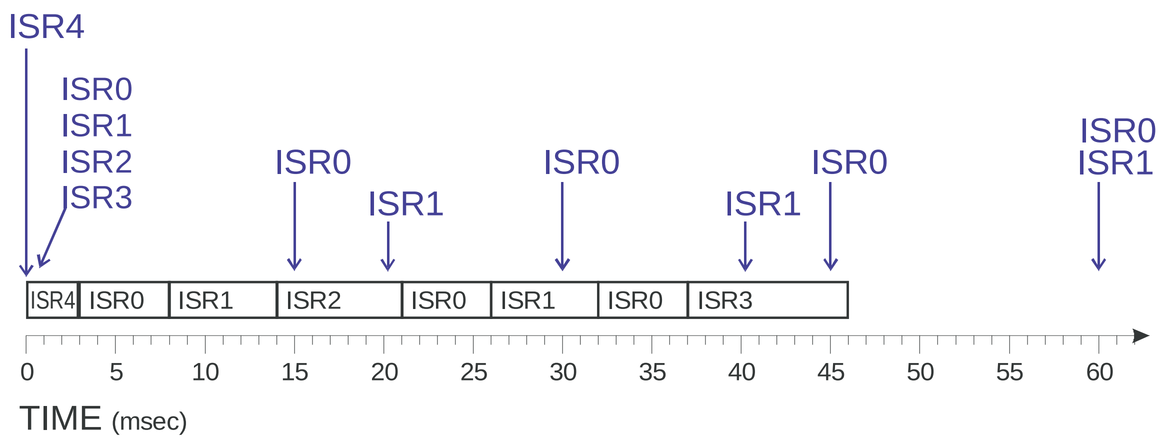

Figure 16. ISR3 worst case latency for B=2. ISR4 causes the longest effective blocking time.

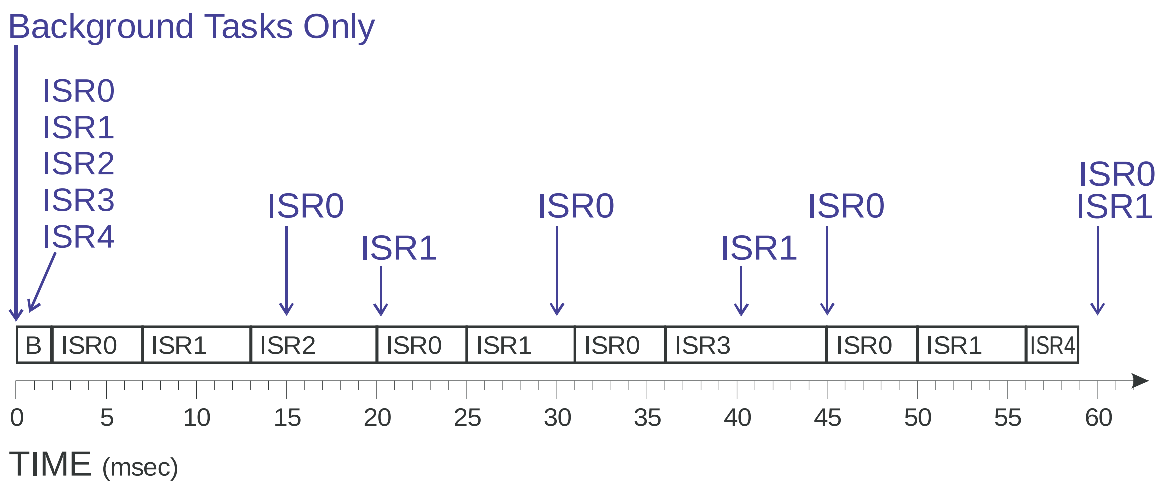

Figure 17. ISR4 worst case latency for B=2. B is the longest effective blocking time because there are no lower priority interrupts to process.

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.