Lecture #15

Interrupt & Cyclic Task Response Timing

18-348 Embedded System Engineering
Philip Koopman
Monday, 14-March-2016
First Boeing “fly by wire” aircraft

- Only computer networks between pilot sticks and control surfaces
777 Triplex Redundancy – 3 PFCs; 3 Networks

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

[Image of a diagram showing the interconnections of various systems and components related to the 777 Triplex Redundancy 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
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
Bad Code on an RTOS

```c
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?**

```c
// 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();
    poll_uart();
    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

```c
// 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:
    \[
    \text{execution time of do_task1()} + \text{execution time of do_task2()} + N \times \text{execution time of handle_uart()} \]
  • 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:

\[
\text{execution time of do_task1()} + \text{execution time of do_task2()} + M \times \text{execution time of ISR1()} + N \times \text{execution time of ISR2()} + P \times \text{execution time of ISR3()}
\]

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 \textit{incorrect} – the correct version is \textit{more complex}
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)
Given:
- Main loop with no ISRs executes in MainLoopOnly
- ISR_m takes ISRtime_m to execute and runs at most every ISRperiod_m

\[ \text{MainTime}_0 = \text{MainLoopOnly} \]

\[ \text{MainTime}_{i+1} = \text{MainTime}_0 + \sum_{\forall \text{ISRs}_j} \left( \frac{\text{MainTime}_i}{\text{ISRperiod}_j} + 1 \right) \text{ISRtime}_j \]

- 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
<table>
<thead>
<tr>
<th>Vector Address</th>
<th>Interrupt Source</th>
<th>CCR Mask</th>
<th>Local Enable</th>
<th>HPRIO Value to Elevate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFE, 0xFFFFF</td>
<td>External reset, power on reset, or low voltage reset (see CRG flags register to determine reset source)</td>
<td>None</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFFFC, 0xFFFFD</td>
<td>Clock monitor fail reset</td>
<td>None</td>
<td>COPCTL (CME, FCME)</td>
<td>—</td>
</tr>
<tr>
<td>0xFFFFA, 0xFFFFB</td>
<td>COP failure reset</td>
<td>None</td>
<td>COP rate select</td>
<td>—</td>
</tr>
<tr>
<td>0xFFFF8, 0xFFFF9</td>
<td>Unimplemented instruction trap</td>
<td>None</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFFF6, 0xFFFF7</td>
<td>SWI</td>
<td>None</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFFF4, 0xFFFF5</td>
<td>XIRQ</td>
<td>X-Bit</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFFF2, 0xFFFF3</td>
<td>IRQ</td>
<td>I bit</td>
<td>INTCR (IRQEN)</td>
<td>0x00F2</td>
</tr>
<tr>
<td>0xFFFF0, 0xFFFF1</td>
<td>Real time Interrupt</td>
<td>I bit</td>
<td>CRGINT (RTIE)</td>
<td>0x00F0</td>
</tr>
<tr>
<td>0xFFFFE, 0xFFFFF</td>
<td>Standard timer channel 0</td>
<td>I bit</td>
<td>TIE (C0I)</td>
<td>0x00EE</td>
</tr>
<tr>
<td>0xFFEC, 0xFFED</td>
<td>Standard timer channel 1</td>
<td>I bit</td>
<td>TIE (C1I)</td>
<td>0x00EC</td>
</tr>
<tr>
<td>0xFFEA, 0xFFEB</td>
<td>Standard timer channel 2</td>
<td>I bit</td>
<td>TIE (C2I)</td>
<td>0x00EA</td>
</tr>
<tr>
<td>0xFFE8, 0xFFE9</td>
<td>Standard timer channel 3</td>
<td>I bit</td>
<td>TIE (C3I)</td>
<td>0x00E8</td>
</tr>
<tr>
<td>0xFFE6, 0xFFE7</td>
<td>Standard timer channel 4</td>
<td>I bit</td>
<td>TIE (C4I)</td>
<td>0x00E6</td>
</tr>
<tr>
<td>0xFFE4, 0xFFE5</td>
<td>Standard timer channel 5</td>
<td>I bit</td>
<td>TIE (C5I)</td>
<td>0x00E4</td>
</tr>
<tr>
<td>0xFFE2, 0xFFE3</td>
<td>Standard timer channel 6</td>
<td>I bit</td>
<td>TIE (C6I)</td>
<td>0x00E2</td>
</tr>
<tr>
<td>0xFFE0, 0xFFE1</td>
<td>Standard timer channel 7</td>
<td>I bit</td>
<td>TIE (C7I)</td>
<td>0x00E0</td>
</tr>
<tr>
<td>0xFFDE, 0xFFDF</td>
<td>Standard timer overflow</td>
<td>I bit</td>
<td>TMSK2 (TOI)</td>
<td>0x00DE</td>
</tr>
<tr>
<td>0xEFFC, 0xEFFD</td>
<td>Pulse accumulator A overflow</td>
<td>I bit</td>
<td>PACTL (PAOVI)</td>
<td>0x00DC</td>
</tr>
</tbody>
</table>
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

`TIME (msec)`

0 5 10 15 20 25 30 35 40 45 50 55 60

Pending @ 9 msec: ISR0, ISR1, ISR2
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*

![Chart showing execution times and priorities]

- ISR3
- ISR0
- ISR1
- ISR2

**TIME (msec)**

- ISR3
- ISR0
- ISR1

Pending @ 20 msec: ISR0, ISR1, ISR2
ISR0 & ISR1 Retrigger, then ISR2 goes

Pending @ 31 msec: ISR0, ISR2

Pending @ 43 msec: ISR1
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 $m$

\[
    ilatency_0 = 0
\]

\[
    ilatency_{i+1} = \max_{j > m}(ISRtime_j) + \sum_{\forall ISRs j < m} \left[ \frac{ilatency_i}{ISRperiod_j} + 1 \right] ISRtime_j
\]

- 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 \textit{one worst-case} 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
    - *You don’t really want to do this***!!! (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) { ...... }
...

Diagram:

```
Tasks

Task 0
Priority 0

Task 1
Priority 1

...Scheduler...

Task N-1
Priority N-1

Ready To Run
```
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)

\[
R_{i,0} = \max_{i < j < N} (C_j) \quad ; i < N - 1
\]

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

- \( R_i \) 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

<table>
<thead>
<tr>
<th>Task# i</th>
<th>Period (P_i)</th>
<th>Execution Time (C_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>6</td>
</tr>
</tbody>
</table>

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

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

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

\[ R_{2,\infty} = 10 \]
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

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

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

\[
R_{N,0} = C_N
\]

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

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

```c
// 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 ISR_0)
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)

```c
// 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 integral
  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!
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?

```c
void interrupt 16 timer_handler(void) // TOI
{
    TFLG2 = 0x80;
    timer_val += 0x10C6; // 16 bits fraction; 48 bits intgr
#ifasm
    CLI ; re-enable interrupts   ** BAD IDEA! **
#endifasm

    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?**

  - 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*

http://betterembsw.blogspot.com/2014/01/do-not-re-enable-interrupts-in-isr.html
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:

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

\[ R_{i,k+1} = R_{i,0} + \sum_{m=0}^{m=i-1} \left( \left[ \frac{R_{i,k}}{P_m} + 1 \right] C_m \right) \quad ; i > 0 \]

\[ R_{N,0} = C_N \]

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