#1 Problem with Real-Time Software Development

No Measurements of Execution Time!
For most embedded systems:
Execution Time == $$$$ 

$$$(Money) 
• Have a limited amount 
• Bills have deadlines 
• Can’t spend more than you have* 
• Count it to know how much you are spending 
• Don’t buy without knowing the price 
• Good budgeting leads to good cash flow

CPU Execution Time 
• Have a limited amount 
• Real-time deadlines 
• Can’t use more than you have* 
• Measure it to know how much you are using 
• Don’t execute code without knowing its CPU usage 
• Good real-time design leads to good CPU usage

* except if you use credit
Two-Part Class
Measuring Execution Time and Real-Time Performance

Part I (Class 341)

Measuring Execution Time
(It’s like Counting $$$$)

Part II (Class 361)

Analyzing Real-Time Performance
(It’s like Budgeting $$$$)
Overview

Part I: Measuring Execution Time
• Techniques for measuring execution
• Deciphering output of measurement tools
• Advantages/disadvantages/issues of each technique
• Optimizing code
• System overhead
• Execution times of tasks

Part II: Analyzing Real-Time Performance
• Obtaining accurate worst-case execution times (WCET)
• Analyzing CPU resource utilization
• Dealing with operating system overhead
• Using measured times to improve real-time scheduling
• Debugging timing errors
• Getting 110% effort from the CPU
## Overview of Measurement Techniques

<table>
<thead>
<tr>
<th>Method</th>
<th>Typical Resolution</th>
<th>Typical Accuracy</th>
<th>Granularity</th>
<th>Difficulty of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop-watch</td>
<td>0.01 sec</td>
<td>0.5 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>date</td>
<td>0.02 sec</td>
<td>0.2 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>time</td>
<td>0.02 sec</td>
<td>0.2 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>prof and gprof</td>
<td>10 msec</td>
<td>20 msec</td>
<td>function</td>
<td>moderate</td>
</tr>
<tr>
<td>clock()</td>
<td>15-30 msec</td>
<td>15-30 msec</td>
<td>line</td>
<td>moderate</td>
</tr>
<tr>
<td>software analyzers</td>
<td>10 µsec</td>
<td>20 µsec</td>
<td>function</td>
<td>moderate</td>
</tr>
<tr>
<td>timer/counter chips</td>
<td>0.5–4 µsec</td>
<td>1-8 µsec</td>
<td>line</td>
<td>very hard</td>
</tr>
<tr>
<td>logic analyzer / ICE</td>
<td>50 nsec</td>
<td>half µsec</td>
<td>line</td>
<td>hard</td>
</tr>
</tbody>
</table>

**color code:** better | okay | worse

**Resolution:** measuring limitation of hardware

**Accuracy:** depends on method, \( x \pm y \Rightarrow x=\text{measurement}, y=\text{accuracy} \)

**Granularity:** what can be measured: program, function, or line of code

**Difficulty:** effort (and time) needed to make a measurement
Measuring Execution Time

Stop-Watch

Use the chronograph feature of a digital watch

• Reset the watch to zero.

• When the activity begins, start the watch

• When the activity ends, stop the watch

• Elapsed time is shown on the watch.
A Stop-Watch measures “Real” Time but not effective for “execution” time

Examples of “Real” Time:
- Every 10 seconds, switch the image on the billboard.
- 2 seconds after the button is pushed, close the door.
- Flash the light at a rate of twice per second.
Use like a stopwatch, except it uses the built-in clock of the computer

Run program as follows:
% date > output
% program >> output
% date >> output

To produce output similar to following:
Thu Mar 1 14:46:09 EST 2001
Program Output Shows up here
Thu Mar 1 14:46:14 EST 2001

Execution time is difference between times shown

14:46:14 - 14:46:09 = 5 seconds
Measuring Execution Time

**Time Command**

Useful when using a UNIX-based system, such as most versions of Embedded Linux

This is the important measurement

\[
\% \text{ time program} \\
8.400u \ 0.040s \ 0:18.40 \ 66.1\%
\]

- **Program executed** for 18.40 seconds, including time for all preemptions

- **Execution time used by operating system on behalf of program**

- **Execution time of program in seconds**

- **Share of CPU used by this program whenever it was ready to execute**

### Table: Method Synopsis

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Typical Accuracy</th>
<th>Granularity</th>
<th>Difficulty</th>
<th>Accounts for Preemption</th>
<th>Interactive Programs</th>
<th>Dependent on RTOS</th>
<th>Affects System Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>usec</td>
<td>line</td>
<td>easy</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>negligible</td>
</tr>
<tr>
<td>Typical Accuracy</td>
<td>msec</td>
<td>function</td>
<td>medium</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>significant</td>
</tr>
<tr>
<td>Granularity</td>
<td></td>
<td>process</td>
<td>hard</td>
<td></td>
<td></td>
<td></td>
<td>sec</td>
</tr>
<tr>
<td>Accuracy</td>
<td>usec</td>
<td>line</td>
<td>easy</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>negligible</td>
</tr>
<tr>
<td>Granularity</td>
<td>msec</td>
<td>function</td>
<td>medium</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>significant</td>
</tr>
<tr>
<td>Difficulty</td>
<td></td>
<td>process</td>
<td>hard</td>
<td></td>
<td></td>
<td></td>
<td>sec</td>
</tr>
<tr>
<td>Accounts for Preemption</td>
<td>usec</td>
<td>line</td>
<td>easy</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>negligible</td>
</tr>
<tr>
<td>Interactive Programs</td>
<td>msec</td>
<td>function</td>
<td>medium</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>significant</td>
</tr>
<tr>
<td>Dependent on RTOS</td>
<td></td>
<td>process</td>
<td>hard</td>
<td></td>
<td></td>
<td></td>
<td>sec</td>
</tr>
</tbody>
</table>
Measuring Execution Time

**prof and gprof programs**

prof (not supported on newer systems)

```
% gcc -p -o program program.c  
% program

% prof program
```

**gprof**

```
% gcc -pg -o program program.c  
% program

% gprof program
```
// useless function calls to show gprof
#include <stdio.h>
#include <math.h>

#define LOOP 10000

double f(double x, double y) {
    double z, sum = 0.0;
    int i;
    for (i = 0; i < LOOP; ++i) {
        for (z = x; z < y; z += 0.01) {
            sum += fabs(sin(z) / cos(z));
        }
    }
    sum = sqrt(sum);
}

int main(void) {
    double start = 1.01, end = 5.0;
    double sum;
    sum = f(start, end);
    printf("sum=%f\n", sum);
}
**prof and gprof programs**

### Example

<table>
<thead>
<tr>
<th>% time</th>
<th>cumulative self</th>
<th>self seconds</th>
<th>total seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>36.4</td>
<td>2.28</td>
<td>2.28</td>
<td>internal_mcount</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.2</td>
<td>3.42</td>
<td>1.14</td>
<td>__libm__rem_pio2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.4</td>
<td>4.32</td>
<td>0.90</td>
<td>__libm__k_sin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.6</td>
<td>4.92</td>
<td>0.60</td>
<td>__libm__k_cos</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>5.34</td>
<td>0.42</td>
<td>__sin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>5.76</td>
<td>0.42</td>
<td>1 420.00 3830.00 f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>6.11</td>
<td>0.35</td>
<td>__cos</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>6.26</td>
<td>0.15</td>
<td>_mcount</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>6.26</td>
<td>0.00</td>
<td>10000 0.00 0.00 __sqrt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>6.26</td>
<td>0.00</td>
<td>24 0.00 0.00 _return_zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 call to \( f() \).

It used 420 msec **excluding** time for other functions it calls

It used 3.830 seconds **including** time for other functions
### Example

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>self seconds</th>
<th>total calls</th>
<th>ms/call</th>
<th>ms/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>36.4</td>
<td>2.28</td>
<td>2.28</td>
<td>internal_mcount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.2</td>
<td>3.42</td>
<td>1.14</td>
<td>8000000</td>
<td>0.00</td>
<td>__libm__rem_pio2</td>
</tr>
<tr>
<td>14.4</td>
<td>4.32</td>
<td>0.90</td>
<td>4000000</td>
<td>0.00</td>
<td>__libm__k_sin</td>
</tr>
<tr>
<td>9.6</td>
<td>4.92</td>
<td>0.60</td>
<td>4000000</td>
<td>0.00</td>
<td>__libm__k_cos</td>
</tr>
<tr>
<td>6.7</td>
<td>5.34</td>
<td>0.42</td>
<td>4000000</td>
<td>0.00</td>
<td>__sin</td>
</tr>
<tr>
<td>6.7</td>
<td>5.76</td>
<td>0.42</td>
<td>1</td>
<td>420.00</td>
<td>3830.00 f</td>
</tr>
<tr>
<td>5.6</td>
<td>6.11</td>
<td>0.35</td>
<td>4000000</td>
<td>0.00</td>
<td>cos</td>
</tr>
<tr>
<td>2.4</td>
<td>6.26</td>
<td>0.15</td>
<td>_mcount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>6.26</td>
<td>0.00</td>
<td>10000</td>
<td>0.00</td>
<td>__sqrt</td>
</tr>
<tr>
<td>0.0</td>
<td>6.26</td>
<td>0.00</td>
<td>24</td>
<td>0.00</td>
<td>_return_zero</td>
</tr>
</tbody>
</table>

400*10000 = 4000000 calls to cos()
each call less than 10.00 msec (resolution is 10 msec)
grand total is 0.35 seconds for 4000000 calls = 87.50 nsec/call
### Example

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self time</th>
<th>seconds</th>
<th>Calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>internal_mcount</td>
</tr>
<tr>
<td>36.4</td>
<td>2.28</td>
<td>2.28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>__libm__rem_pio2</td>
</tr>
<tr>
<td>18.2</td>
<td>3.42</td>
<td>1.14</td>
<td>8000000</td>
<td>0.00</td>
<td></td>
<td>0.00</td>
<td>__libm__k_sin</td>
<td></td>
</tr>
<tr>
<td>14.4</td>
<td>4.32</td>
<td>0.90</td>
<td>4000000</td>
<td>0.00</td>
<td></td>
<td>0.00</td>
<td>__libm__k_cos</td>
<td></td>
</tr>
<tr>
<td>9.6</td>
<td>4.92</td>
<td>0.60</td>
<td>4000000</td>
<td>0.00</td>
<td></td>
<td>0.00</td>
<td>__libm__k_cos</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>5.34</td>
<td>0.42</td>
<td>4000000</td>
<td>0.00</td>
<td></td>
<td>0.00</td>
<td>__sin</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>5.76</td>
<td>0.42</td>
<td>1</td>
<td>420.00</td>
<td>3830.00</td>
<td>sqrt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>6.11</td>
<td>0.35</td>
<td>4000000</td>
<td>0.00</td>
<td></td>
<td>0.00</td>
<td>__cos</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>6.26</td>
<td>0.15</td>
<td>mcount</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>6.26</td>
<td>0.00</td>
<td>10000</td>
<td>0.00</td>
<td></td>
<td>0.00</td>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>6.26</td>
<td>0.00</td>
<td>24</td>
<td>0.00</td>
<td></td>
<td>0.00</td>
<td>_return_zero</td>
<td></td>
</tr>
</tbody>
</table>

10000 calls to `sqrt()` used a grand total less than 0.01 seconds (therefore each call to `sqrt()` less than 10 usec.)
#include <time.h>

clock_t start, finish;
double total;

start = clock();
do stuff;
finish = clock();

total = ((double)(finish - start)) / ((double)CLK_TCK)

printf("Total = %f seconds\n", total);
Software Analyzer is a general term for Commercial Tools that perform some kind of time measurement. E.g.:

- CodeTest (Applied Microsystems)
- TimeTrace (TimeSys)
- WindView (WindRiver)
- Various Processor Simulators

Each tool focusses on one or more ways to collect and represent data.

- Two examples in following slides

Tools are usually specific to an RTOS or architecture or development environment
Software Analyzer
Questions to Ponder when Selecting a Tool

What is the resolution and accuracy?
• If based on the system clock, the resolution will likely be on the order of a millisecond
• If based on timer hardware support, resolution will be on the order of microseconds

What is the granularity?
• Some only provide information on a per-function or per-process basis (like gprof), not on a per-statement basis
• To be effective, it must have a means for measuring execution time for small code segments
• When measuring code that is used by multiple tasks, it must be able to measure the call separately for each task

Can the tool pinpoint when execution time is used?
• The tool should be able to produce detailed timing data, including system overhead, to help identify problems such as race conditions.
• A graphical timing diagram enables easy viewing of the data
• A text data log enables automatic processing of the data
Software Analyzer

More Questions to Ponder when Selecting a Tool

Does the tool take into account system overhead?
- interrupt handling
- interprocess communication,
- context switch and scheduling
- exception handling

What special hardware is needed?
- hardware can include pods, emulators, or custom device
- tools might only be able to deliver on promised accuracy if special hardware is used
- tools that don’t use special hardware might not yield sufficient detail

Is the tool intrusive?
- most tools involve instrumenting code (either manually or automatically) which changes execution time.
- non-intrusive tools don’t affect timing, but might require specialized hardware specific to the processor
- non-intrusive tools with no special hardware tend to not yield good accuracy
Software Analyzer Example
Timing Diagram

10 20 30 40 50 60 70 80 90 100 (msec)
Software Analyzer Example
Timing Diagram

list of tasks and interrupts

Interrupt appears periodic

Over 10 msec for Task A

very little idle time

Two cycles? Or one cycle w/ preemption or blocking?

Intr
TaskA
TaskB
TaskC
TaskD
TaskE
idle

10 20 30 40 50 60 70 80 90 100 (msec)
Software Analyzer Example
Timing Diagram

How it helps
• view interactions between tasks
• identify patterns and obvious anomalies
• approximate load average for each task

What it doesn’t tell you
• correlation of data to source code
• average and worst-case execution times
• timing errors** and RTOS overhead

** (although an expert might deduce errors from viewing the diagram, the diagram itself does not state what is normal and what isn’t)
### Software Analyzer Example

**Functional Profile**

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Max Time</th>
<th>Avg Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioWrite</td>
<td>5</td>
<td>0.033</td>
<td>0.023</td>
</tr>
<tr>
<td>ioRead</td>
<td>3</td>
<td>0.095</td>
<td>0.054</td>
</tr>
<tr>
<td>calc</td>
<td>48</td>
<td>0.431</td>
<td>0.121</td>
</tr>
<tr>
<td>taskSensor</td>
<td>1</td>
<td>48.055</td>
<td>48.055</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### How it helps
- quickly identify where most time is spent
- useful for optimizing code
- counts how often each function is called

#### What it doesn’t tell you
- timing errors
  (eg clock skew due to sleep())
- preemption and synchronization
- task execution time
- function—task relationships

```c
taskSensor() {
    while (1) {
        x = ioRead();
        y = calc(x);
        ioWrite(y);
        sleep()
    }
}
```
Software Analyzer Example
Other Capabilities

Vendors are constantly improving the timing information provided by their tools.

Consult each tool vendor regularly to check for their latest offerings.
Measuring Execution Time
Timer/Counter Chip

Similar to using clock()
• except that a hardware timer or counter chip is used instead of an operating system’s clock() function.

Read count-down value of timer/counter chip
• Read the value at the start of a measurement
• Read again at the end.
• The difference is the execution time, measured in timer ticks.

Biggest problem is count-down value rollover.
• Logic Analyzer (described next) is often a better method.

Adaptive scheduling
• This method mainly useful if the program’s scheduling is adaptive based on execution time of tasks
A good hardware tool can provide the most accurate timing view of the system.

The logic analyzer can be connected to the target in one of several ways:

- to parallel output port (the more bits, the better)
- to pins on an in-circuit emulator (ICE)
- to pins on a shared bus (e.g., VMEbus, PCI, etc.)
- to serial output port
- to analog output port

The serial or analog output ports can be used, but they are less accurate and could affect the real-time performance as the data requires more encoding.
Logic Analyzer
Desirable Features

- Support state mode
- Support automatic detection of transitions
- A deep buffer on the analyzer is highly desirable
- To measure execution time, only 8 channels are needed
- Some form of high-speed output from the analyzer, like Ethernet, GPIB, or USB is very helpful
- Alternately, built-in workstation, so that data can be captured into a database, spreadsheet, or custom analysis program.
- A search option is also very helpful
- Logic analyzer pod that plugs into a laptop is also acceptable
Step 1: Setup macros for writing the output port:

```c
#define MEZ_START(id) output(dioport,0x50|id&0xF)
#define MEZ_STOP(id) output(dioport,0x60|id&0xF)
```

Step 2: Instrument the code whose execution time is to be measured:

```c
MEZ_START(1);
funcA();
MEZ_STOP(1);
MEZ_START(2);
y = a + b * c;
MEZ_STOP(2);
```
Step 3: Execute code, and capture data on logic analyzer

Step 4: Analyze data

- Timestamp can usually be displayed as relative or **absolute**:
  
  (Relative Time Mode:)  (Absolute Time Mode)
  
<table>
<thead>
<tr>
<th>Data</th>
<th>Time</th>
<th>Data</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x51</td>
<td>--</td>
<td>0x51</td>
<td>--</td>
</tr>
<tr>
<td>0x61</td>
<td>358u</td>
<td>0x61</td>
<td>358u</td>
</tr>
<tr>
<td>0x52</td>
<td>3u</td>
<td>0x52</td>
<td>360u</td>
</tr>
<tr>
<td>0x62</td>
<td>14u</td>
<td>0x62</td>
<td>374u</td>
</tr>
</tbody>
</table>

  y=a+b*c took 14 usec
  funcA took 358 usec

  y=a+b*c took 374–360 = 4 usec
  funcA took 358–0 = 358 usec
Modify macro to write only the single bit

// Example 1: register is shared; only bit 3 is available;
// port is memory mapped, dioport is a ptr to it
// ‘id’ is ignored, but kept for compatibility
#define MEZ_START(id) *dioport |= 0x80
#define MEZ_STOP(id) *dioport &~ 0x80

// Example 2: register is not shared, I/O Mapped Device
#define MEZ_START(id) output(dioport,1)
#define MEZ_STOP(id)  output(dioport,0)

Instrumented code is same as before. ‘id’ is not needed, but kept so that macros
are consistent with multiple-bit version shown previously:

MEZ_START(1);
funcA();
MEZ_STOP(1);
MEZ_START(2);
y = a + b * c;
MEZ_STOP(2);
Use timing mode of analyzer to measure width of pulses

\[ t_1 = 358 \text{ usec} \]
\[ x = 12 \text{ usec} \]
\[ o = 370 \text{ usec} \]
\[ t_1 = \Delta = 358 \text{ usec} \]
Similar to parallel I/O port, except macros setup differently.

```c
// Select an unused memory address
// it must be in external memory so that the
// address appears on the address lines
uint8_t *traceAddr = 0x40009000;
#define MEZ_START(id) *traceAddr=(0x50|id&0xF)
#define MEZ_STOP(id) *traceAddr=(0x60|id&0xF)
```

Connect logic analyzer to address and data lines
- Don’t need to connect all address lines; only need enough for triggering the logic analyzer
- Don’t need all data lines; if pointer is unsigned char, only need D0–D7.

Setup logic analyzer to trigger on address, and store data.
- This provides the same data log as for the parallel output method.
Main code continually sends pulses to channel 1.
Interrupt handler provides a single pulse to channel 2

This results in the following timing diagram:

```
          channel 1
        /\         \t1
      /       \   \t3
     /           \--
     channel 2     \
         \t2
```

Interrupt handler execution time = \( t_2 \)
Overhead to call handler = \( \frac{t_1 - t_2 - t_3}{2} \)

The same method can be used to measure operating system’s context switch overhead
Measuring Execution Time of Tasks
Model of a Task

nextstart = clock()
deadline = period

Periodic
nextstart += period
timer
pause(nextstart, deadline)
MEZ_START(getpid())
read inputs
cycle
write outputs
MEZ_STOP(getpid())

Aperiodic
external signal
operating system overhead
sync

Legend

- instruction
- blocking operation
- Instrumentation
- process flow

Task execution time
### Measuring Execution Time of Tasks

#### Sample Logic Analyzer Event Log

<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Rel Time</th>
<th>Abs Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>1</td>
<td>61</td>
<td>3.0669ms</td>
<td>3.0669ms</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>5.9341ms</td>
<td>9.0010ms</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>3.0669ms</td>
<td>12.067ms</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>240.69us</td>
<td>12.308ms</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>5.0739ms</td>
<td>17.382ms</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>1.5974ms</td>
<td>18.980ms</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>3.0669ms</td>
<td>22.047ms</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>6.6304ms</td>
<td>28.677ms</td>
</tr>
<tr>
<td>9</td>
<td>51</td>
<td>436.53us</td>
<td>29.114ms</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
<td>3.0669ms</td>
<td>32.180ms</td>
</tr>
<tr>
<td>11</td>
<td>52</td>
<td>773.17us</td>
<td>32.954ms</td>
</tr>
<tr>
<td>12</td>
<td>62</td>
<td>5.0944ms</td>
<td>38.048ms</td>
</tr>
<tr>
<td>13</td>
<td>51</td>
<td>896.05us</td>
<td>38.944ms</td>
</tr>
<tr>
<td>14</td>
<td>61</td>
<td>3.3741ms</td>
<td>42.318ms</td>
</tr>
<tr>
<td>15</td>
<td>51</td>
<td>6.6099ms</td>
<td>48.928ms</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
<td>3.0464ms</td>
<td>51.975ms</td>
</tr>
<tr>
<td>17</td>
<td>63</td>
<td>3.1078ms</td>
<td>55.083ms</td>
</tr>
<tr>
<td>18</td>
<td>52</td>
<td>2.8006ms</td>
<td>57.883ms</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>1.0086ms</td>
<td>58.892ms</td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>3.0464ms</td>
<td>61.938ms</td>
</tr>
<tr>
<td>21</td>
<td>62</td>
<td>4.3162ms</td>
<td>66.255ms</td>
</tr>
</tbody>
</table>

#### Task Set:

<table>
<thead>
<tr>
<th>Task</th>
<th>Thread ID</th>
<th>Period (msec)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>01</td>
<td>0.010</td>
<td>High</td>
</tr>
<tr>
<td>TaskB</td>
<td>02</td>
<td>0.025</td>
<td>Medium</td>
</tr>
<tr>
<td>TaskC</td>
<td>03</td>
<td>0.040</td>
<td>Low</td>
</tr>
</tbody>
</table>
## Measuring Execution Time of Tasks

### Sample Logic Analyzer Event Log

<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Rel Time</th>
<th>Abs Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>1</td>
<td>61</td>
<td>3.0669ms</td>
<td>3.0669ms</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>5.9341ms</td>
<td>9.0010ms</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>3.0669ms</td>
<td>12.067ms</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>240.69us</td>
<td>12.308ms</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>5.0739ms</td>
<td>17.382ms</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>1.5974ms</td>
<td>18.980ms</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>3.0669ms</td>
<td>22.047ms</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>6.6304ms</td>
<td>28.677ms</td>
</tr>
<tr>
<td>9</td>
<td>51</td>
<td>436.53us</td>
<td>29.114ms</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
<td>3.0669ms</td>
<td>32.180ms</td>
</tr>
<tr>
<td>11</td>
<td>52</td>
<td>773.17us</td>
<td>32.954ms</td>
</tr>
<tr>
<td>12</td>
<td>62</td>
<td>5.0944ms</td>
<td>38.048ms</td>
</tr>
<tr>
<td>13</td>
<td>51</td>
<td>896.05us</td>
<td>38.944ms</td>
</tr>
<tr>
<td>14</td>
<td>61</td>
<td>3.3741ms</td>
<td>42.318ms</td>
</tr>
<tr>
<td>15</td>
<td>51</td>
<td>6.6099ms</td>
<td>48.928ms</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
<td>3.0464ms</td>
<td>51.975ms</td>
</tr>
<tr>
<td>17</td>
<td>63</td>
<td>3.1078ms</td>
<td>55.083ms</td>
</tr>
<tr>
<td>18</td>
<td>52</td>
<td>2.8006ms</td>
<td>57.883ms</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>1.0086ms</td>
<td>58.892ms</td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>3.0464ms</td>
<td>61.938ms</td>
</tr>
<tr>
<td>21</td>
<td>62</td>
<td>4.3162ms</td>
<td>66.255ms</td>
</tr>
</tbody>
</table>

### Task Set:

<table>
<thead>
<tr>
<th>Task</th>
<th>Thread ID</th>
<th>Period(msec)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>01</td>
<td>0.010</td>
<td>High</td>
</tr>
<tr>
<td>TaskB</td>
<td>02</td>
<td>0.025</td>
<td>Medium</td>
</tr>
<tr>
<td>TaskC</td>
<td>03</td>
<td>0.040</td>
<td>Low</td>
</tr>
</tbody>
</table>
# Measuring Execution Time of Tasks
## Interpreting the Logic Analyzer Event Log

<table>
<thead>
<tr>
<th>Task Set:</th>
<th>Task</th>
<th>Thread ID</th>
<th>Period(msec)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TaskA</td>
<td>01</td>
<td>0.010</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>TaskB</td>
<td>02</td>
<td>0.025</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td>TaskC</td>
<td>03</td>
<td>0.040</td>
<td>Low</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Rel Time</th>
<th>Abs Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>1</td>
<td>61</td>
<td>3.0669ms</td>
<td>3.0669ms</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>5.9341ms</td>
<td>9.0010ms</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>3.0669ms</td>
<td>12.067ms</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>240.69us</td>
<td>12.308ms</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>5.0739ms</td>
<td>17.382ms</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>1.5974ms</td>
<td>18.980ms</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>3.0669ms</td>
<td>22.047ms</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>6.6304ms</td>
<td>28.677ms</td>
</tr>
<tr>
<td>9</td>
<td>51</td>
<td>436.53us</td>
<td>29.114ms</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
<td>3.0669ms</td>
<td>32.180ms</td>
</tr>
<tr>
<td>11</td>
<td>52</td>
<td>773.17us</td>
<td>32.954ms</td>
</tr>
<tr>
<td>12</td>
<td>62</td>
<td>5.0944ms</td>
<td>38.048ms</td>
</tr>
<tr>
<td>13</td>
<td>51</td>
<td>896.05us</td>
<td>38.944ms</td>
</tr>
<tr>
<td>14</td>
<td>61</td>
<td>3.3741ms</td>
<td>42.318ms</td>
</tr>
<tr>
<td>15</td>
<td>51</td>
<td>6.6099ms</td>
<td>48.928ms</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
<td>3.0464ms</td>
<td>51.975ms</td>
</tr>
<tr>
<td>17</td>
<td>63</td>
<td>3.1078ms</td>
<td>55.083ms</td>
</tr>
<tr>
<td>18</td>
<td>52</td>
<td>2.8006ms</td>
<td>57.883ms</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>1.0086ms</td>
<td>58.892ms</td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>3.0464ms</td>
<td>61.938ms</td>
</tr>
<tr>
<td>21</td>
<td>62</td>
<td>4.3162ms</td>
<td>66.255ms</td>
</tr>
</tbody>
</table>

The execution time for this iteration of Task B is 5.0739 msec.
### Measuring Execution Time of Tasks

#### Interpreting the Logic Analyzer Event Log

<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Rel Time</th>
<th>Abs Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>1</td>
<td>61</td>
<td>3.0669ms</td>
<td>3.0669ms</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>5.9341ms</td>
<td>9.0010ms</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>3.0669ms</td>
<td>12.067ms</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>240.69us</td>
<td>12.308ms</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>5.0739ms</td>
<td>17.382ms</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>1.5974ms</td>
<td>18.980ms</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>3.0669ms</td>
<td>22.047ms</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>6.6304ms</td>
<td>28.677ms</td>
</tr>
<tr>
<td>9</td>
<td>51</td>
<td>436.53us</td>
<td>29.114ms</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
<td>3.0669ms</td>
<td>32.180ms</td>
</tr>
<tr>
<td>11</td>
<td>52</td>
<td>773.17us</td>
<td>32.954ms</td>
</tr>
<tr>
<td>12</td>
<td>62</td>
<td>5.0944ms</td>
<td>38.048ms</td>
</tr>
<tr>
<td>13</td>
<td>51</td>
<td>896.05us</td>
<td>38.944ms</td>
</tr>
<tr>
<td>14</td>
<td>61</td>
<td>3.3741ms</td>
<td>42.318ms</td>
</tr>
<tr>
<td>15</td>
<td>51</td>
<td>6.6099ms</td>
<td>48.928ms</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
<td>3.0464ms</td>
<td>51.975ms</td>
</tr>
<tr>
<td>17</td>
<td>63</td>
<td>3.1078ms</td>
<td>55.083ms</td>
</tr>
<tr>
<td>18</td>
<td>52</td>
<td>2.8006ms</td>
<td>57.883ms</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>1.0086ms</td>
<td>58.892ms</td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>3.0464ms</td>
<td>61.938ms</td>
</tr>
<tr>
<td>21</td>
<td>62</td>
<td>4.3162ms</td>
<td>66.255ms</td>
</tr>
</tbody>
</table>

### Task Set:

<table>
<thead>
<tr>
<th>Task</th>
<th>Thread ID</th>
<th>Period(msec)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>01</td>
<td>0.010</td>
<td>High</td>
</tr>
<tr>
<td>TaskB</td>
<td>02</td>
<td>0.025</td>
<td>Medium</td>
</tr>
<tr>
<td>TaskC</td>
<td>03</td>
<td>0.040</td>
<td>Low</td>
</tr>
</tbody>
</table>

The execution time for this iteration of Task C is:

\[
(55.081\text{ms} - 28.677\text{ms}) - (3.0669 + 5.0944 + 3.3741 + 3.0464) = 11.8222\text{ms}
\]
Measuring Execution Time and Real-Time Performance

Dave Stewart

Task Set:

<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Rel Time</th>
<th>Abs Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>1</td>
<td>61</td>
<td>3.0669ms</td>
<td>3.0669ms</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>5.9341ms</td>
<td>9.0010ms</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>3.0669ms</td>
<td>12.067ms</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>240.69us</td>
<td>12.308ms</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>5.0739ms</td>
<td>17.382ms</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>1.5974ms</td>
<td>18.980ms</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>3.0669ms</td>
<td>22.047ms</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>6.6304ms</td>
<td>28.677ms</td>
</tr>
<tr>
<td>9</td>
<td>51</td>
<td>436.53us</td>
<td>29.114ms</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
<td>3.0669ms</td>
<td>32.180ms</td>
</tr>
<tr>
<td>11</td>
<td>52</td>
<td>773.17us</td>
<td>32.954ms</td>
</tr>
<tr>
<td>12</td>
<td>62</td>
<td>5.0944ms</td>
<td>38.048ms</td>
</tr>
<tr>
<td>13</td>
<td>51</td>
<td>896.05us</td>
<td>38.944ms</td>
</tr>
<tr>
<td>14</td>
<td>61</td>
<td>3.3741ms</td>
<td>42.318ms</td>
</tr>
<tr>
<td>15</td>
<td>51</td>
<td>6.6099ms</td>
<td>48.928ms</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
<td>3.0464ms</td>
<td>51.975ms</td>
</tr>
<tr>
<td>17</td>
<td>63</td>
<td>3.1078ms</td>
<td>55.083ms</td>
</tr>
<tr>
<td>18</td>
<td>52</td>
<td>2.8006ms</td>
<td>57.883ms</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>1.0086ms</td>
<td>58.892ms</td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>3.0464ms</td>
<td>61.938ms</td>
</tr>
<tr>
<td>21</td>
<td>62</td>
<td>4.3162ms</td>
<td>66.255ms</td>
</tr>
</tbody>
</table>

The conversion from event log to table should be automated, using either a custom program, spreadsheet, or a software analyzer tool that has the necessary features to give you execution time for each cycle of every task.
Debugging embedded code becomes especially difficult when ...

- There is real I/O, so the simulator doesn’t work
- An emulator is not available
- Stepping through the code (eg using BDM) makes the program behave differently
- There is no console, so printf() (or equivalent) doesn’t work
- Writing debug output to a serial port changes the timing too much
- There might be a race condition or other synchronization problem
- The code that needs to be debugged is executing 10,000 times per second
- The embedded code crashes, but there is no feedback as to where
- Software appears to be fine, it might be a hardware error ...
- Memory is getting corrupted, but the source cannot be identified
- etc. etc. etc.

A logic analyzer can be a very effective tool for debugging even the toughest embedded code problems.

- The technique is very similar to measuring execution time with the logic analyzer
Create several debug macros that output codes to a digital I/O port or to a specific memory or bus address

    // 16 bits of output makes it a lot easier to debug code,  
    // but it still works with 8 bits with some additional effort.  
    uint8_t  *dio8port  = 0x9999;  // if 8 bits  
    uint16_t *dio16port = 0x9999; // if 16 bits  

    // Output current line number  
    #define DEBUG_LINENO() {*dio8port = __LINE__}  
    // if file has more than 256 lines and 8-bit port used:  
    #define DEBUG_LINENO() {*dio8port=(__LINE__>>8);*dioport=__LINE__}  

Use this macro as necessary to trace code:

    some code here
    DEBUG_LINENO();
    f();  
    DEBUG_LINENO();
    if (condition)
        g();
    DEBUG_LINENO();
    more code here

---

Logic Analyzer Trace:

<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Rel Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>0ns</td>
</tr>
<tr>
<td>1</td>
<td>34</td>
<td>3.0669ms</td>
</tr>
<tr>
<td>2</td>
<td>37</td>
<td>5.9341ms</td>
</tr>
</tbody>
</table>
Macro to output variable name and its value

```c
#define DEBUG_INT16(var) {*dio16port = *(int16_t *)#var; 
                          *dio16port = var }
```

Sample Usage
```
DEBUG_INT16(val);
idx = val*f();
DEBUG_INT16(idx);
idx *= g();
DEBUG_INT16(idx);
```

Example Trace:
```
<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Abs Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7661</td>
<td>0us</td>
</tr>
<tr>
<td>1</td>
<td>0003</td>
<td>1.00us</td>
</tr>
<tr>
<td>2</td>
<td>6964</td>
<td>48.9us</td>
</tr>
<tr>
<td>3</td>
<td>000F</td>
<td>49.9us</td>
</tr>
<tr>
<td>4</td>
<td>6964</td>
<td>102.us</td>
</tr>
<tr>
<td>5</td>
<td>001E</td>
<td>103.ms</td>
</tr>
</tbody>
</table>
```

The output is fairly cryptic, but the data can be extremely valuable when all other methods fail.
Real-Time Debugging using the Logic Analyzer

Other Examples of Debug Macros

It is possible to create debug macros for just about anything.

- When outputting line number, also output first two letters of filename using __FILE__
  
  ```
  #define DEBUG_FILELINE() {*dio16port = *(int16_t *)__FILE__; 
  *dio16port = __LINE__}
  ```

- Output only on condition.
  
  ```
  #define DEBUG_COND(_cond_) {if (_cond_) *dio16port = __LINE__}
  // use it as follows
  Some code here
  x = f(y);
  DEBUG_COND(x>3);
  more code here
  ```

- Output a string.
  
  ```
  #define DEBUG_STRING(_str_) {char *p=_str_; 
  while (*p != '\0') *dio8port=*p++;
  // use it as follows
  char name[N];
  some code here
  DEBUG_STRING(name);
  more code here
  ```

The logic analyzer can be the final line of defense for debugging embedded code.
## Summary of Part I: Measuring Execution Time

<table>
<thead>
<tr>
<th>Method</th>
<th>Typical Resolution</th>
<th>Typical Accuracy</th>
<th>Granularity</th>
<th>Difficulty of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop-watch</td>
<td>0.01 sec</td>
<td>0.5 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>date</td>
<td>0.02 sec</td>
<td>0.2 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>time</td>
<td>0.02 sec</td>
<td>0.2 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>prof and gprof</td>
<td>10 msec</td>
<td>20 msec</td>
<td>function</td>
<td>moderate</td>
</tr>
<tr>
<td>clock()</td>
<td>15-30 msec</td>
<td>15-30 msec</td>
<td>line</td>
<td>moderate</td>
</tr>
<tr>
<td>software analyzers</td>
<td>10 µsec</td>
<td>20 µsec</td>
<td>function</td>
<td>moderate</td>
</tr>
<tr>
<td>timer/counter chips</td>
<td>0.5–4 µsec</td>
<td>1-8 µsec</td>
<td>line</td>
<td>very hard</td>
</tr>
<tr>
<td>logic analyzer / ICE</td>
<td>50 nsec</td>
<td>half µsec</td>
<td>line</td>
<td>hard</td>
</tr>
</tbody>
</table>

**Color code:** better, okay, worse
Part I (Class 341)

Measuring Execution Time
(It’s like Counting $$$$)

Part II (Class 361)

Analyzing Real-Time Performance
(It’s like Budgeting $$$$)
Recall Part I (Class 341)

Measuring Execution Time
(It’s like Counting $$$$)

Part II (Class 361)

Analyzing Real-Time Performance
(It’s like Budgeting $$$$)
Analyzing Real-Time Performance

Outline

• Execution Time by Task
• Analyzing Effect of Operating System Overhead
• Validating Periods of Periodic Tasks
• Detecting Missed Deadlines
• Practical Fixed Priority Scheduling Analysis
• Making Unschedulable Task Sets Schedulable
• Discussion: Getting 110% Effort from the CPU
### Recall Part I

#### Measuring Execution Time

<table>
<thead>
<tr>
<th>Method</th>
<th>Typical Resolution</th>
<th>Typical Accuracy</th>
<th>Granularity</th>
<th>Difficulty of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop-watch</td>
<td>0.01 sec</td>
<td>0.5 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>date</td>
<td>0.02 sec</td>
<td>0.2 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>time</td>
<td>0.02 sec</td>
<td>0.2 sec</td>
<td>program</td>
<td>easy</td>
</tr>
<tr>
<td>prof and gprof</td>
<td>10 msec</td>
<td>20 msec</td>
<td>function</td>
<td>moderate</td>
</tr>
<tr>
<td>clock()</td>
<td>15-30 msec</td>
<td>15-30 msec</td>
<td>line</td>
<td>moderate</td>
</tr>
<tr>
<td>software analyzers</td>
<td>10 μsec</td>
<td>20 μsec</td>
<td>function</td>
<td>moderate</td>
</tr>
<tr>
<td>timer/counter chips</td>
<td>0.5–4 μsec</td>
<td>1-8 μsec</td>
<td>line</td>
<td>very hard</td>
</tr>
<tr>
<td>logic analyzer / ICE</td>
<td>50 nsec</td>
<td>half μsec</td>
<td>line</td>
<td>hard</td>
</tr>
</tbody>
</table>
Recall Measuring Execution Time of Tasks
Example of Logic Analyzer Event Log

First, must obtain measured execution time by task

<table>
<thead>
<tr>
<th>Trace</th>
<th>Data</th>
<th>Rel Time</th>
<th>Abs Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>1</td>
<td>61</td>
<td>3.0669ms</td>
<td>3.0669ms</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>5.9341ms</td>
<td>9.0010ms</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>3.0669ms</td>
<td>12.067ms</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>240.69us</td>
<td>12.308ms</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>5.0739ms</td>
<td>17.382ms</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>1.5974ms</td>
<td>18.980ms</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>3.0669ms</td>
<td>22.047ms</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>6.6304ms</td>
<td>28.677ms</td>
</tr>
<tr>
<td>9</td>
<td>51</td>
<td>436.53us</td>
<td>29.114ms</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
<td>3.0669ms</td>
<td>32.180ms</td>
</tr>
<tr>
<td>11</td>
<td>52</td>
<td>773.17us</td>
<td>32.954ms</td>
</tr>
<tr>
<td>12</td>
<td>62</td>
<td>5.0944ms</td>
<td>38.048ms</td>
</tr>
<tr>
<td>13</td>
<td>51</td>
<td>896.05us</td>
<td>38.944ms</td>
</tr>
<tr>
<td>14</td>
<td>61</td>
<td>3.3741ms</td>
<td>42.318ms</td>
</tr>
<tr>
<td>15</td>
<td>51</td>
<td>6.6099ms</td>
<td>48.928ms</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
<td>3.0464ms</td>
<td>51.975ms</td>
</tr>
<tr>
<td>17</td>
<td>63</td>
<td>3.1078ms</td>
<td>55.083ms</td>
</tr>
<tr>
<td>18</td>
<td>52</td>
<td>2.8006ms</td>
<td>57.883ms</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>1.0086ms</td>
<td>58.892ms</td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>3.0464ms</td>
<td>61.938ms</td>
</tr>
<tr>
<td>21</td>
<td>62</td>
<td>4.3162ms</td>
<td>66.255ms</td>
</tr>
</tbody>
</table>

Task Set:
- Task A
- Task B
- Task C

<table>
<thead>
<tr>
<th>Task</th>
<th>Thread ID</th>
<th>Period(msec)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskA</td>
<td>01</td>
<td>0.010</td>
<td>High</td>
</tr>
<tr>
<td>TaskB</td>
<td>02</td>
<td>0.025</td>
<td>Medium</td>
</tr>
<tr>
<td>TaskC</td>
<td>03</td>
<td>0.040</td>
<td>Low</td>
</tr>
</tbody>
</table>
Use one of the measurement methods to create the following table:

<table>
<thead>
<tr>
<th>ID</th>
<th>Cref</th>
<th>Tref</th>
<th>Cavg</th>
<th>Cmax</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000100</td>
<td>0.001000</td>
<td>0.00025</td>
<td>0.00056</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.001000</td>
<td>0.002000</td>
<td>0.00288</td>
<td>0.002472</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.002000</td>
<td>0.008000</td>
<td>0.00253</td>
<td>0.002922</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0.002000</td>
<td>0.010000</td>
<td>0.00253</td>
<td>0.002922</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0.001000</td>
<td>0.040000</td>
<td>0.00149</td>
<td>0.000587</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.003000</td>
<td>0.050000</td>
<td>0.00570</td>
<td>0.006311</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0.002000</td>
<td>0.100000</td>
<td>0.005229</td>
<td>0.006910</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>0.004000</td>
<td>0.200000</td>
<td>0.009570</td>
<td>0.011306</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.005000</td>
<td>0.400000</td>
<td>0.017208</td>
<td>0.017208</td>
<td>1</td>
</tr>
</tbody>
</table>

Total sum of execution times = 10.183

T = Period
C = Execution Time
ref = reference (ie from specs)
avg = measured average
max = measured worst-case
### Measuring Execution Time and Real-Time Performance

**Tabulate Execution Time by Task**

Example for set of 8 tasks and 1 interrupt handler

Use one of the measurement methods to create the following table:

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Cref</th>
<th>Tref</th>
<th>Cavg</th>
<th>Cmax</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000100</td>
<td>0.001000</td>
<td>0.000025</td>
<td>0.000056</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.001000</td>
<td>0.004000</td>
<td>0.001094</td>
<td>0.001096</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.002000</td>
<td>0.008000</td>
<td>0.002288</td>
<td>0.002472</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.002000</td>
<td>0.010000</td>
<td>0.002530</td>
<td>0.002922</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0.001000</td>
<td>0.040000</td>
<td>0.00149</td>
<td>0.000587</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.003000</td>
<td>0.050000</td>
<td>0.005756</td>
<td>0.006311</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0.002000</td>
<td>0.100000</td>
<td>0.005229</td>
<td>0.006910</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>0.004000</td>
<td>0.200000</td>
<td>0.009570</td>
<td>0.011306</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.005000</td>
<td>0.400000</td>
<td>0.017208</td>
<td>0.017208</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total sum of execution times = 10.183**

**Task ID 0 is actually an interrupt handler**

- \( T = \text{Period} \)
- \( C = \text{Execution Time} \)
- \( \text{ref} = \text{reference (ie from specs)} \)
- \( \text{avg} = \text{measured average} \)
- \( \text{max} = \text{measured worst-case} \)

**Number of RTOS-Detected Missed Deadlines**

(requires an instrumentation point within the timing error handler)

- Estimated Worst-Case Execution Time
- Desired Period (or rate) (note \( \text{rate} = 1/\text{period} \))
- Measured Average-Case Execution Time
- Measured Worst-Case Execution Time
Analyzing Real-Time Performance
Outline

• Execution Time by Task
• Analyzing Effect of Operating System Overhead
• Validating Periods of Periodic Tasks
• Detecting Missed Deadlines
• Practical Fixed Priority Scheduling Analysis
• Making Unschedulable Task Sets Schedulable
• Discussion: Getting 110% Effort from the CPU
Recall Model of a Task

Periodic

nextstart = clock()

deadline = period

nextstart += period

timer

pause(nextstart,deadline)

Aperiodic

external signal

operating system overhead (Δthr)

Legend

- instruction
- blocking operation
- Instrumentation
- process flow

Legend

- instruction
- blocking operation
- Instrumentation
- process flow

InHand Electronics
http://www.inhand.com

Measuring Execution Time and Real-Time Performance
Dave Stewart

Embedded Systems Conference
Boston, September 2006, Class 341/361
Analyzing Effect of Operating System Overhead

Overhead is already part of the Measurements

Using the logic analyzer technique, overhead becomes part of the measurements, but it is not uniformly distributed across tasks. This is important to consider when precise measurements are needed.

For example, assume following timing diagram:

\[
\begin{align*}
\Delta_{thr} &= t_y + t_z \\
C_{idle} &= t_y + t_z \\
C_{3,1} &= t_a \\
C_{3,2} &= t_b + t_d \\
C_{3,3} &= t_e + t_i + t_k + 4\Delta_{thr}
\end{align*}
\]
Analyzing Effect of Operating System Overhead
Distribution of Overhead in Measurements

The real execution times, without overhead:

\[
C_{1,1} = t_g \\
C_{idle} = t_y + t_z
\]

\[
C_{2,1} = t_c; \\
C_{2,2} = t_f + t_h;
\]

\[
C_{3,1} = t_a \\
C_{3,2} = t_b + t_d \\
C_{3,3} = t_e + t_i + t_k
\]

The data we measure:

\[
C_{1,1} = t_g \\
C_{idle} = t_y + t_z + 4\Delta_{thr}
\]

\[
C_{2,1} = t_c; \\
C_{2,2} = t_f + t_h + 2\Delta_{thr};
\]

\[
C_{3,1} = t_a \\
C_{3,2} = t_b + t_d + 2\Delta_{thr} \\
C_{3,3} = t_e + t_i + t_k + 4\Delta_{thr}
\]

This is misleading! Idle CPU time means it is available for other functions. OS overhead, however, is not available.

If approximate answers are acceptable, then this issue can be considered negligible.

For consistency in analysis, this is the data we want with overhead considered:

\[
C_{1,1} = t_g + 2\Delta_{thr} \\
C_{idle} = t_y + t_z
\]

\[
C_{2,1} = t_c + 2\Delta_{thr} \\
C_{2,2} = t_f + t_h + 2\Delta_{thr}
\]

\[
C_{3,1} = t_a + 2\Delta_{thr} \\
C_{3,2} = t_b + t_d + 2\Delta_{thr} \\
C_{3,3} = t_e + t_i + t_k + 2\Delta_{thr}
\]

If accurate answers are needed, measurements must be adjusted to account for number of preemptions
Measuring Operating System Overhead $\Delta_{thr}$

Similar technique as Measuring Interrupt Handler Overhead

Low priority task continually sends pulses to channel 1.
High priority task provides a single pulse to channel 2

This results in the following timing diagram:

channel 1

channel 2

High priority task execution time = $t_2$
Operating system overhead to switch tasks $\Delta_{thr} = (t_1 - t_2 - t_3) / 2$

Note that overhead is not necessarily constant. The overhead is often a function of the number of tasks in the system. For most analysis, however, assuming that the overhead is constant is sufficient.
Analyzing Real-Time Performance
Outline

- Execution Time by Task
- Analyzing Effect of Operating System Overhead
- Validating Periods of Periodic Tasks
- Detecting Missed Deadlines
- Practical Fixed Priority Scheduling Analysis
- Making Unschedulable Task Sets Schedulable
- Discussion: Getting 110% Effort from the CPU
Validating Periods for Periodic Tasks

Are tasks in the application executing at the rates requested?

Quite often, they are not, and could be the cause of errors!

For Example

• Resolution of system clock
  400 Hz task cannot be implemented if system clock is 1 msec

• Resolution of hardware timer affects accuracy of system clock
  A 1 msec clock might be approximated to 998 usec
  This affects time by over 3 minutes per day!

• Skew due to implementation error
  Occurs when using sleep(X) — block task for X msec
  Instead, must use pause(X) — block task until time=X

• Real-Time Scheduling Problems cause actual periods to vary
  Overloaded processor
  Priority Inversion
  Missed Deadlines
  Interrupts executing for too long

Note: Many software analyzers are unable to detect these problems.
Validating Periods for Periodic Tasks
Recall the MEZ\_START() and MEZ\_STOP Macros

nextstart = clock()

deadline = period

Periodic

nextstart += period
timer

pause(nextstart,deadline)

sync

e external signal

Aperiodic

operating system overhead ($\Delta_{thr}$)

MEZ\_START(getpid())

read inputs

cycle

write outputs

MEZ\_STOP(getpid())

Legend

instruction

blocking operation

Instrumentation

process flow
“Non-delayed” starting points starting points occur when MEZ_START() is detected to occur either:
• after MEZ_START() of a lower-priority task’s cycle, but before the lower-priority task’s MEZ_STOP()
• when the idle task is executing

Example:

Task B
Requested
Execution

TaskA
TaskB
TaskC

Non-Delayed Starting Point
Execution begins by preempting lower-priority task

Delayed Starting Point
Execution begins after end of higher-priority task
Validating Periods for Periodic Tasks

Compute Time Differences

Compute the time difference between each two non-delayed starting points.

Divide by \((\text{number_of_delayed_starting_points}+1)\) between those starting points.

This number should be exactly the period of the task; if it isn’t, there could be timing errors.

Example of what should be expected

\[
\begin{align*}
\text{period} &= (90 - 10)(1+1) = 80/2 = 40 \text{ msec} \\
\text{Repeat for every pair of non-delayed starting points.}
\end{align*}
\]
Suppose the following timing diagram was obtained:

```
period = (88 – 10)(1+1) = 78/2 = 39 msec
```

In this case, 40 msec period is requested, but because the system clock is set to 3 msec, exactly 40 msec cannot be achieved. The RTOS instead rounded to a 39 msec period, without any notification of error to the designer.
Analyzing Real-Time Performance
Outline

• Execution Time by Task
• Analyzing Effect of Operating System Overhead
• Validating Periods of Periodic Tasks
• Detecting Missed Deadlines
• Practical Fixed Priority Scheduling Analysis
• Making Unschedulable Task Sets Schedulable
• Discussion: Getting 110% Effort from the CPU
Checking for Missed Deadlines

To check for missed deadlines, do the following for each task:

- Find non-delayed starting points for each task in the event log (use same method as for validating periods)
- For the delayed starting points, compute the requested starting point by using the non-delayed starting points and adding to it the measured period
- Compute the deadline time relative to each starting point
- The MEZ_STOP() event that indicates end of a task should execute before this computed deadline time every cycle.

This procedure can be automated by using data downloaded from the logic analyzer as input. Theoretical details in “A Tool for Analyzing and Fine-Tuning the Real-Time Properties of an Embedded System,” on the web.
Analyzing Real-Time Performance
Outline

- Execution Time by Task
- Analyzing Effect of Operating System Overhead
- Validating Periods of Periodic Tasks
- Detecting Missed Deadlines
- Practical Fixed Priority Scheduling Analysis
- Making Unschedulable Task Sets Schedulable
- Discussion: Getting 110% Effort from the CPU
Real-Time Scheduling Analysis is very theoretical and mathematically oriented.

Here, a practical approximation is presented, using real measured data, that can lead to quickly converging on resolving all the real-time issues in a system.

We’ll start with the theory, then simplify it to make it more practical for use in system design and debugging.
A task set consisting of n periodic threads is schedulable using the Rate Monotonic Algorithm if the following equation holds:

\[ \forall i, 1 \leq i \leq (n_{intr} + n_{thr}) \]

\[
\min_{0 < t \leq D_i} \left( \sum_{j=1}^{\min(i, n_{intr})} \frac{C_j + 2\Delta_{intr}}{t} \left\lceil \frac{t}{T_j} \right\rceil + \sum_{j=n_{intr}+1}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil \right) \leq 1
\]

Example, lets apply equation to the data collected and shown in the following table:

<table>
<thead>
<tr>
<th>ID</th>
<th>Cref</th>
<th>Tref</th>
<th>Cavg</th>
<th>Cmax</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000100</td>
<td>0.001000</td>
<td>0.00025</td>
<td>0.00056</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.001000</td>
<td>0.004000</td>
<td>0.00194</td>
<td>0.00196</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.002000</td>
<td>0.008000</td>
<td>0.00288</td>
<td>0.00292</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0.002000</td>
<td>0.010000</td>
<td>0.00253</td>
<td>0.00292</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0.001000</td>
<td>0.040000</td>
<td>0.00149</td>
<td>0.00158</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.003000</td>
<td>0.050000</td>
<td>0.00575</td>
<td>0.00631</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0.002000</td>
<td>0.100000</td>
<td>0.00522</td>
<td>0.00691</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>0.004000</td>
<td>0.200000</td>
<td>0.00957</td>
<td>0.01130</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.005000</td>
<td>0.400000</td>
<td>0.01720</td>
<td>0.01720</td>
<td>1</td>
</tr>
</tbody>
</table>

Total sum of execution times = 10.183
Fixed-Priority Scheduling Analysis

Simplifying the Equations

First, some approximations with negligible effect except when during the most critical fine-tuning.

- Assume ID 0 is highest priority task instead of an interrupt
- Instead of $0 < t \leq D_i$, use longest length of trace that is a multiple of the slowest task

This reduces the equation:

$$\forall i, 1 \leq i \leq (n_{intr} + n_{thr})$$

$$\min_{0 < t \leq D_i} \left( \min(i, n_{intr}) \sum_{j = 1}^{i} \frac{C_j + 2\Delta_{intr}}{t} \left\lceil \frac{t}{T_j} \right\rceil + \sum_{j = n_{intr} + 1}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil \right) \leq 1$$

This simply leads to conservative estimates, since it results in computing more overhead than there really is.

This means try equation first with $i=1$, then $i=2$, then $i=3$, until LHS > 1, or $i == n_{thr}$.
Fixed-Priority Scheduling Analysis

Applying the Equations using Real Data

With 1 task:

Total sum of execution times = 10.183

<table>
<thead>
<tr>
<th>ID</th>
<th>Cref</th>
<th>Tref</th>
<th>Cavg</th>
<th>Cmax</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000100</td>
<td>0.001000</td>
<td>0.000025</td>
<td>0.000056</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.001000</td>
<td>0.004000</td>
<td>0.001094</td>
<td>0.001096</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.002000</td>
<td>0.008000</td>
<td>0.002288</td>
<td>0.002472</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.002000</td>
<td>0.010000</td>
<td>0.002530</td>
<td>0.002922</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0.001000</td>
<td>0.040000</td>
<td>0.000149</td>
<td>0.000587</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.003000</td>
<td>0.050000</td>
<td>0.005756</td>
<td>0.006311</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0.002000</td>
<td>0.100000</td>
<td>0.005229</td>
<td>0.006910</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>0.004000</td>
<td>0.200000</td>
<td>0.009570</td>
<td>0.011306</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.005000</td>
<td>0.400000</td>
<td>0.017208</td>
<td>0.017208</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
LHS = \frac{0.000056 + 0.0001}{10} \cdot \frac{10}{0.001} = 0.156
\]

This means task 0 is always schedulable.
With 2 tasks:

Total sum of execution times = 10.183

<table>
<thead>
<tr>
<th>ID</th>
<th>Cref</th>
<th>Tref</th>
<th>Cavg</th>
<th>Cmax</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000100</td>
<td>0.001000</td>
<td>0.000025</td>
<td>0.000056</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.001000</td>
<td>0.004000</td>
<td>0.001094</td>
<td>0.001096</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.002000</td>
<td>0.008000</td>
<td>0.002288</td>
<td>0.002472</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.002000</td>
<td>0.010000</td>
<td>0.002530</td>
<td>0.002922</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0.001000</td>
<td>0.040000</td>
<td>0.000149</td>
<td>0.000587</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.003000</td>
<td>0.050000</td>
<td>0.005756</td>
<td>0.006311</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0.002000</td>
<td>0.100000</td>
<td>0.005229</td>
<td>0.006910</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>0.004000</td>
<td>0.200000</td>
<td>0.009570</td>
<td>0.011306</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.005000</td>
<td>0.400000</td>
<td>0.017208</td>
<td>0.017208</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \sum_{j=1}^{i} \left( \frac{C_j + 2\Delta_{thr}t}{t} \right) \left\lfloor \frac{t}{T_j} \right\rfloor \leq 1 \]

\[ \text{LHS} = 0.156 + \frac{0.001096 + 0.0001}{10} = 0.455 \]

This means tasks 0 & 1 are always schedulable.
Applying the Equations using Real Data
With 3 Tasks

With 3 tasks:

<table>
<thead>
<tr>
<th>ID</th>
<th>Cref</th>
<th>Tref</th>
<th>Cavg</th>
<th>Cmax</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000100</td>
<td>0.001000</td>
<td>0.000025</td>
<td>0.000056</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.001000</td>
<td>0.004000</td>
<td>0.001094</td>
<td>0.001096</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.002000</td>
<td>0.008000</td>
<td>0.002288</td>
<td>0.002472</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.002000</td>
<td>0.010000</td>
<td>0.002530</td>
<td>0.002922</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0.001000</td>
<td>0.040000</td>
<td>0.000149</td>
<td>0.000587</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.003000</td>
<td>0.050000</td>
<td>0.005756</td>
<td>0.006311</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0.002000</td>
<td>0.100000</td>
<td>0.005229</td>
<td>0.006910</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>0.004000</td>
<td>0.200000</td>
<td>0.009570</td>
<td>0.011306</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.005000</td>
<td>0.400000</td>
<td>0.017208</td>
<td>0.017208</td>
<td>1</td>
</tr>
</tbody>
</table>

Total sum of execution times $= 10.183$

\[
\sum_{j=1}^{i} \left( \frac{C_j + 2\Delta_{th}}{t} \right) \left( \frac{t}{T_j} \right) \leq 1
\]

\[
LHS = 0.156 + 0.299 + 0.322 = 0.777
\]

This means tasks 0 & 1 & 2 are always schedulable.
Applying the Equations using Real Data
With 2 Tasks

With 4 tasks:

<table>
<thead>
<tr>
<th>ID</th>
<th>Cref</th>
<th>Tref</th>
<th>Cavg</th>
<th>Cmax</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000100</td>
<td>0.001000</td>
<td>0.000025</td>
<td>0.000056</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.001000</td>
<td>0.004000</td>
<td>0.001094</td>
<td>0.001096</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.002000</td>
<td>0.008000</td>
<td>0.002288</td>
<td>0.002472</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.002000</td>
<td>0.010000</td>
<td>0.002530</td>
<td>0.002922</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0.001000</td>
<td>0.040000</td>
<td>0.000149</td>
<td>0.000587</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.003000</td>
<td>0.050000</td>
<td>0.005756</td>
<td>0.006311</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0.002000</td>
<td>0.100000</td>
<td>0.005229</td>
<td>0.006910</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>0.004000</td>
<td>0.200000</td>
<td>0.009570</td>
<td>0.011306</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.005000</td>
<td>0.400000</td>
<td>0.017208</td>
<td>0.017208</td>
<td>1</td>
</tr>
</tbody>
</table>

Total sum of execution times = 10.183

\[
\sum_{j=1}^{i} \left( \frac{C_j + 2\Delta_{thr}}{t} \left( \frac{t}{T_j} \right) \right) \leq 1
\]

\[
LHS = 0.156 + 0.299 + 0.322 + 0.301 = 1.079
\]

This means tasks 0 & 1 & 2 & 4 are NOT always schedulable.
What does the analysis mean?

It says a lot:

- Unless changes are made only Tasks 0, 1, and 2 are guaranteed to not miss deadlines.
- In order for Task 3 to meet deadlines, either
  - Decrease collective execution time \((C_i)\) of Tasks 0, 1, 2, and 3
  - Increase period \((T_i)\) of Tasks 0, 1, 2, and/or 3
- None of Tasks 4 through 8 are guaranteed to meet deadlines as long as Task 3 is not guaranteed.

**Question:** To make Task 3 schedulable, by how much do we decrease \(C_i\) or increase \(T_i\)?
Analyzing Real-Time Performance
Outline

• Execution Time by Task
• Analyzing Effect of Operating System Overhead
• Validating Periods of Periodic Tasks
• Detecting Missed Deadlines
• Practical Fixed Priority Scheduling Analysis
• Making Unschedulable Task Sets Schedulable
• Discussion: Getting 110% Effort from the CPU
Suppose goal is to fix the system so that the interrupt handler (PID 0) and first three tasks (PID 1 through 3) are guaranteed to meet their deadlines.

First, revise $C_{ref}$ using the values of $C_{max}$. This refines the estimated worst-case execution times to ensure they are more accurate.

To make tasks schedulable, must reduce left side of equation to below 1.0 (it was 1.079 in example). One of the following must be done:

- Decrease CPU usage by optimizing code
  I.e. reduce $C_{ref}$ for one or more tasks

- Increase period if application and hardware allow it
  I.e. increase $T_{ref}$ for one or more tasks

Repeat the math only, without actually performing the above optimization or period changes.

- Only when a valid answer is obtained should the actual code be modified.
- Increasing period $T_{ref}$ is a lot easier than reducing execution time $C_{ref}$.
- To reduce $C_{ref}$, it is necessary to optimize or delete some code.
Analyzing Real-Time Performance
Outline

- Execution Time by Task
- Analyzing Effect of Operating System Overhead
- Validating Periods of Periodic Tasks
- Detecting Missed Deadlines
- Practical Fixed Priority Scheduling Analysis
- Making Unschedulable Task Sets Schedulable
- Discussion: Getting 110% Effort from the CPU
Real-Time Scheduling
Discussion: Getting 110% Effort from the CPU

Real-Time Scheduling is often idealized. In reality, the following situations are quite common, and appear to negate the use of proper real-time analysis:

- A task’s worst-case execution time is much greater than its typical case.
- The system is mostly event-driven.
- This is not a hard real-time system: deadlines are soft.
- A task’s period changes depending on what it is doing.
- Tasks waste time busy-waiting.
- It’s a fast CPU, so it doesn’t matter how long the code takes.
- Can’t use more than 80% utilization to ensure a buffer zone for timing problems
- “I need 110% utilization!”
  - Equivalent to “I need 110% $$$$” => Use Credit!

No matter what the problem, however, knowing the execution time of each task, and using it as the basis for analyzing the system, is essential to ensure a predictable real-time system that meets application requirements.

Regardless of the application, systematically engineering the timing of the system will lead to a system with minimal timing errors and efficient budgeting of the CPU while lowering the amount of time needed to implement and debug the code.
Summary

Measuring Execution Time and Real-Time Performance

Part I (Class 341)

Measuring Execution Time
(It’s like Counting $$$$)

Part II (Class 361)

Analyzing Real-Time Performance
(It’s like Budgeting $$$$)