Measuring Execution Time and Real-Time Performance

Class ESC 341/361

Embedded Systems Conference Boston

September 2006

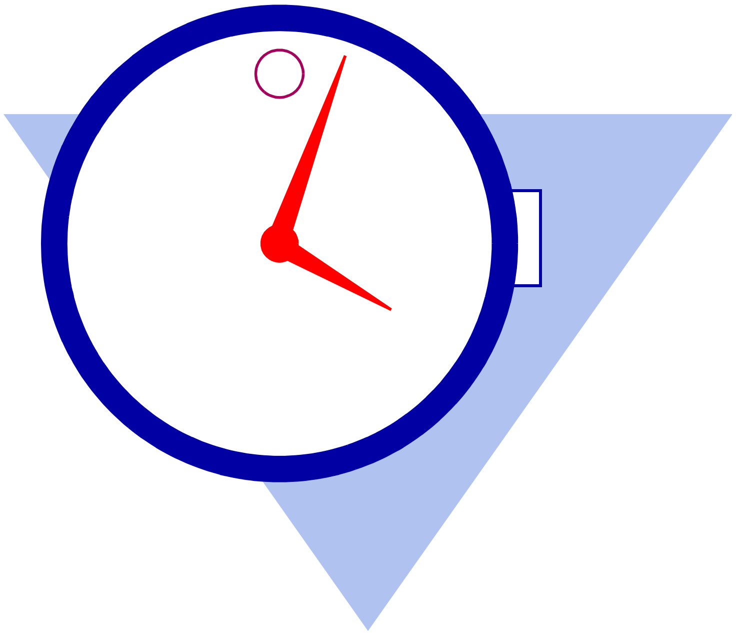
inhand electronics

Dave Stewart

Director of Software Engineering
InHand Electronics
Rockville, MD
dstewart@inhand.com
http://www.inhand.com

#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

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

Method	Typical Resolution	Typical Accuracy	Granularity	Difficulty of Use
stop-watch	0.01 sec	0.5 sec	program	easy
date	0.02 sec	0.2 sec	program	easy
time	0.02 sec	0.2 sec	program	easy
prof and gprof	10 msec	20 msec	function	moderate
clock()	15-30 msec	15-30 msec	line	moderate
software analyzers	10 μsec	20 μsec	function	moderate
timer/counter chips	0.5–4 μsec	1-8 μsec	line	very hard
logic analyzer / ICE	50 nsec	half μsec	line	hard
color code:		better	okay	worse

Resolution: measuring limitation of hardware

Accuracy: depends on method, $x \pm y \Rightarrow x$=measurement, y=accuracy

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

Difficulty: effort (and time) needed to make a measurement

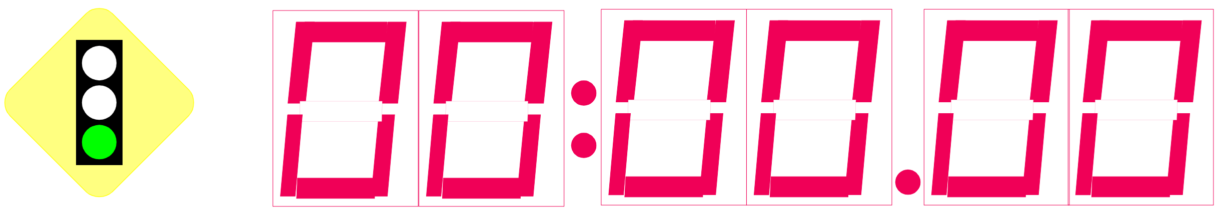
Measuring Execution Time
Stop-Watch

Method Synopsis

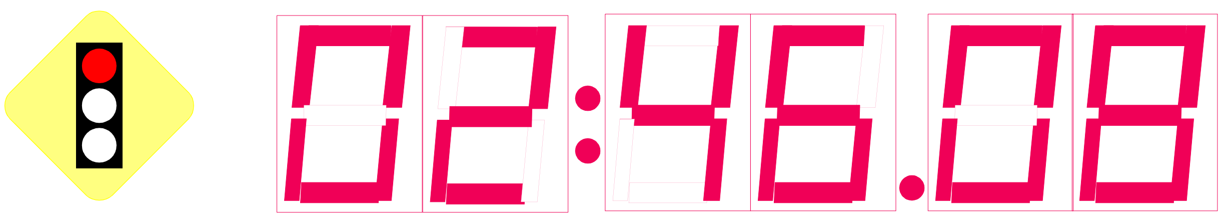
better ⟷ worse			
Affects system timing:		negligible	significant
Typical accuracy:	usec	msec	sec
Granularity:	line	function	process
Difficulty:	easy	medium	hard
Accounts for preemption:		yes	no
Interactive programs:		yes	no
Dependent on RTOS:		no	yes

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.**

Stop-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.

Measuring Execution Time
Date Command

Method Synopsis

better ⟷ worse

Affects system timing:		negligible	significant
Typical accuracy:	usec	msec	sec
Granularity:	line	function	process
Difficulty:	easy	medium	hard
Accounts for preemption:		yes	no
Interactive programs:		yes	no
Dependent on RTOS:		no	yes

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

Method Synopsis (better ⟷ worse)

Affects system timing:		**negligible**	significant
Typical accuracy:	usec	**msec**	sec
Granularity:	line	function	**process**
Difficulty:	**easy**	medium	hard
Accounts for preemption:		**yes**	no
Interactive programs:		**yes**	no
Dependent on RTOS:		no	**yes**

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

This is the important measurement

```
% time program
8.400u 0.040s 0:18.40 66.1%
```

- **8.400u** — execution time of *program* in seconds
- **0.040s** — execution time used by operating system on behalf of *program*
- **0:18.40** — Program executed for 18.40 seconds, including time for all preemptions
- **66.1%** — share of CPU used by this *program* whenever it was ready to execute

Measuring Execution Time
prof and *gprof* programs

Method Synopsis

better ⟷ worse

Affects system timing:		negligible	significant
Typical accuracy:	usec	msec	sec
Granularity:	line	function	process
Difficulty:	easy	medium	hard
Accounts for preemption:		yes	no
Interactive programs:		yes	no
Dependent on RTOS:		no	yes

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
```

prof and *gprof* programs
Example

```
// 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);
}
```

Outer loop executes 10000 times

Inner loop executes 400 times

This line executes 400,000 times

This line executes once

prof and *gprof* programs
Example

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

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

prof and *gprof* programs
Example

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

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

prof and *gprof* programs
Example

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

10000 calls to *sqrt()* used a grand total less than 0.01 seconds

(therefore each call to sqrt() less than 10 usec.)

Measuring Execution Time
clock()

Method Synopsis

better ↔ worse

Affects system timing:		negligible	significant
Typical accuracy:	usec	msec	sec
Granularity:	line	function	process
Difficulty:	easy	medium	hard
Accounts for preemption:		yes*	no*
Interactive programs:		yes	no
Dependent on RTOS:		no	yes

*Depends on RTOS

```
#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);
```

Measuring Execution Time
Software Analyzer

Method Synopsis (better ⟷ worse)

Affects system timing:		negligible*	significant*
Typical accuracy:	usec*	msec*	sec
Granularity:	line*	function*	process
Difficulty:	easy	medium	hard
Accounts for preemption:		yes*	no*
Interactive programs:		yes	no
Dependent on RTOS:		no	yes

*depends on tool

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

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

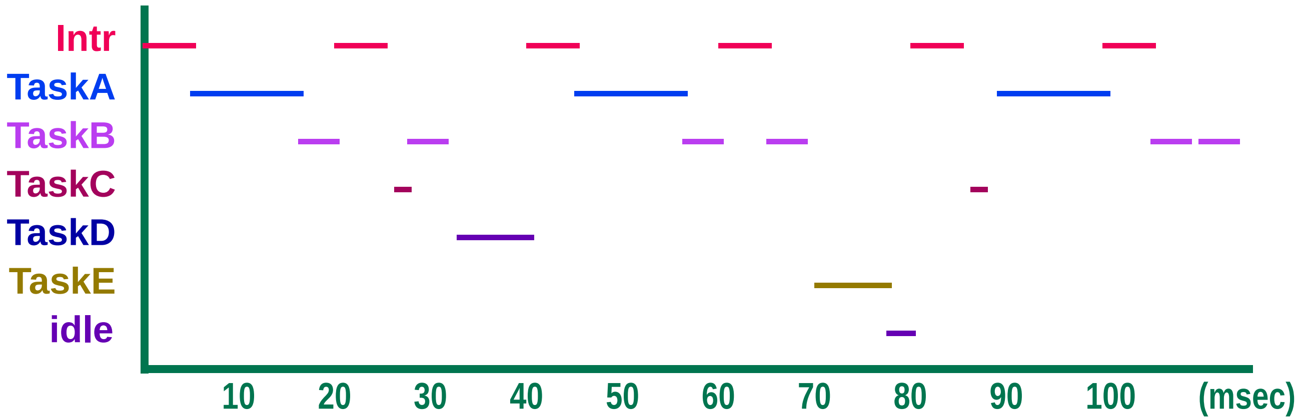
What special hardware is needed?

- 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

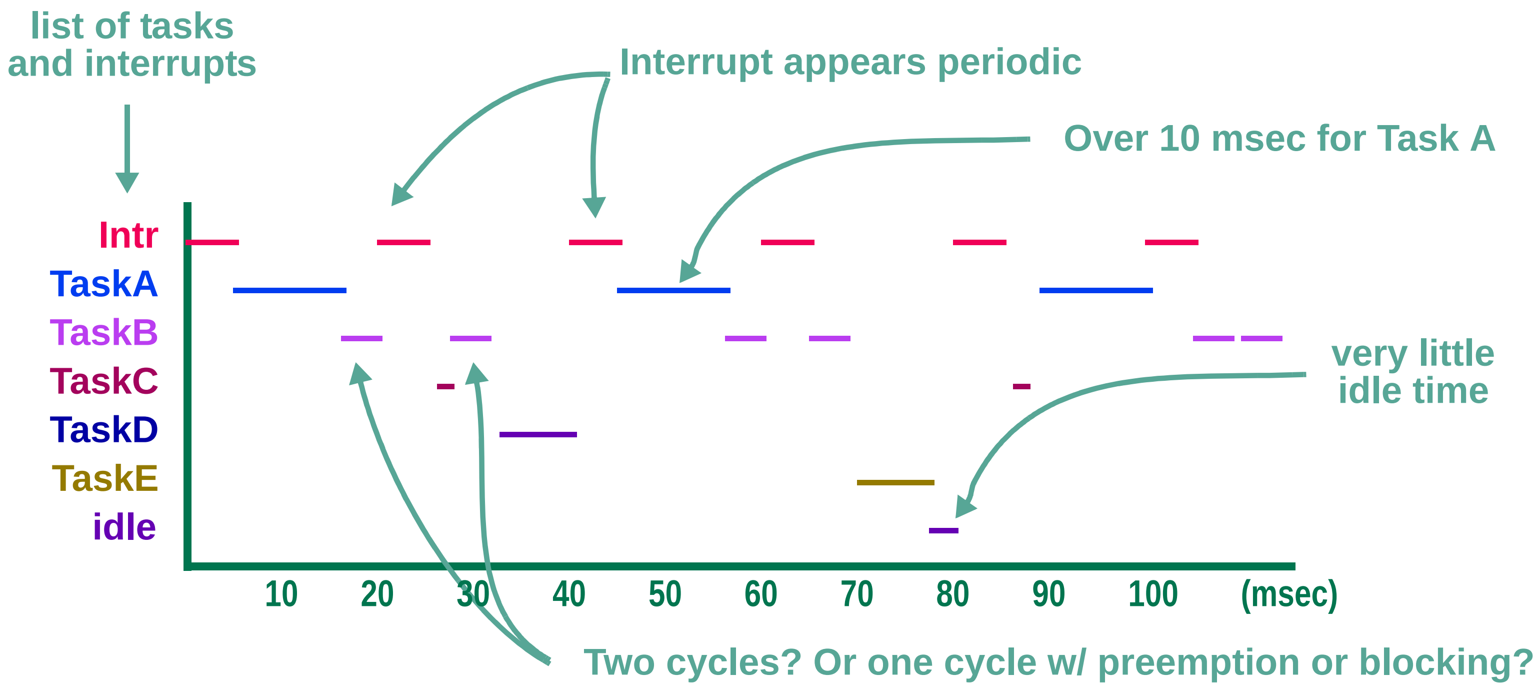
Is the tool intrusive?

- most tools involve instrumenting code (either manually or automatically) which changes execution time.
- non-intrusive tools don't affect timing, but might require specilized hardware specific to the processor
- non-intrusive tools with no special hardware tend to not yield good accuracy

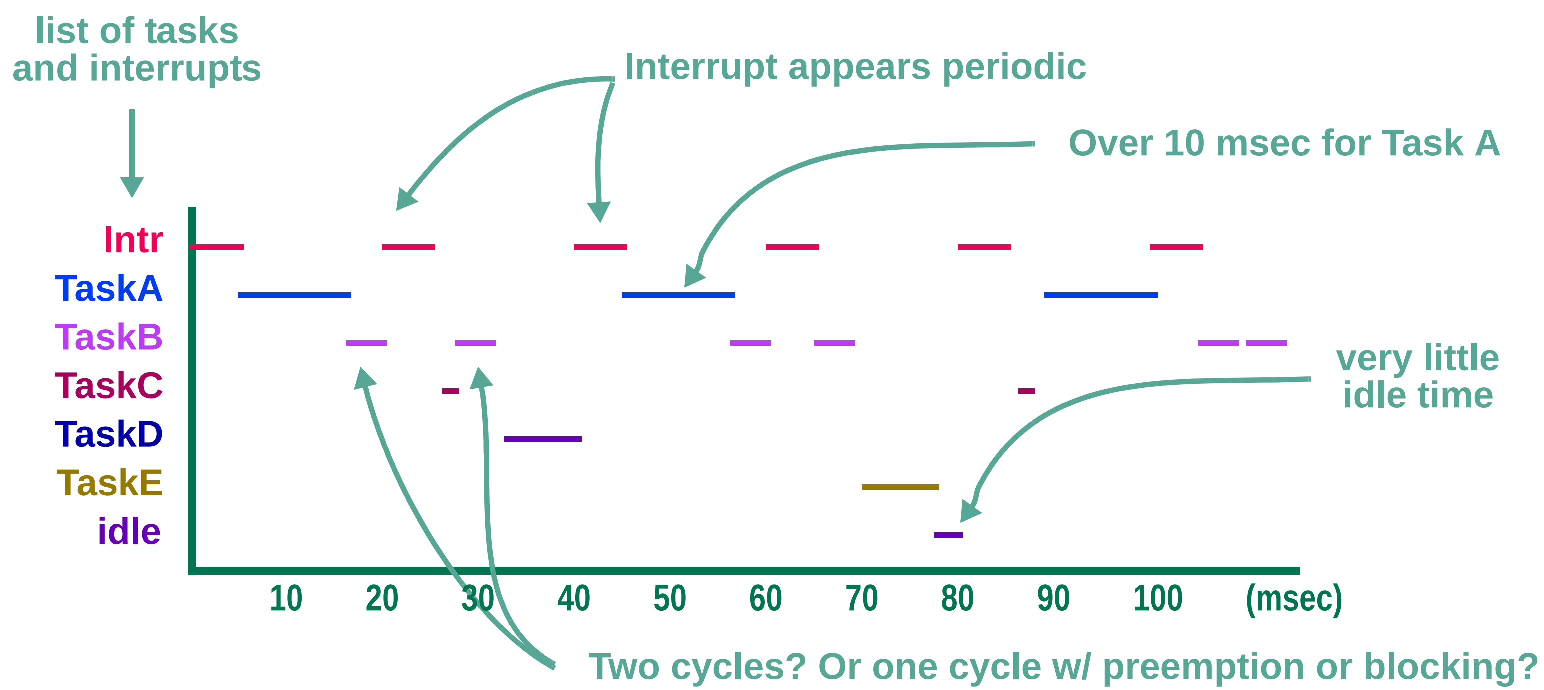
Software Analyzer Example
Timing Diagram



Software Analyzer Example
Timing Diagram



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

Function	Count	Max Time	Avg Time
ioWrite	5	0.033	0.023
ioRead	3	0.095	0.054
calc	48	0.431	0.121
taskSensor	1	48.055	48.055
etc.			

```
taskSensor() {
    while (1) {
        x=ioRead();
        y=calc(x);
        ioWrite(y);
        sleep()
    }
}
```

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

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

Method Synopsis

	better ⟷		worse
Affects system timing:		negligible	significant
Typical accuracy:	usec	msec	sec
Granularity:	line	function	process
Difficulty:	easy	medium	hard
Accounts for preemption:		yes	no
Interactive programs:		yes	no
Dependent on RTOS:		no	yes

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

Measuring Execution Time
Logic Analyzer

Method Synopsis (better ⟷ worse)

Affects system timing:		negligible	significant
Typical accuracy:	usec	msec	sec
Granularity:	line	function	process
Difficulty:	easy	medium	hard
Accounts for preemption:		yes	no*
Interactive programs:		yes	no
Dependent on RTOS:		no	yes

*not directly, but technique shown later can account for it

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 (eg 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

Logic Analyzer
Measuring Code Segments with Parallel Output Port

Step 1: Setup macros for writing the output port:

```
#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:

```
    :
MEZ_START(1);
funcA();
MEZ_STOP(1);
MEZ_START(2);
y = a + b * c;
MEZ_STOP(2);
    :
```

Measuring Code Segments with Parallel Output Port

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:)

Data	Time
0x51	--
0x61	358u
0x52	3u
0x62	14u

y=a+b*c took 14 usec

funcA took 358 usec

(Absolute Time Mode)

Data	Time
0x51	--
0x61	358u
0x52	360u
0x62	374u

y=a+b*c took 374–360 = 4 usec

funcA took 358–0= 358 usec

Logic Analyzer
Using a Single Output Bit

Modify macro to write only the single bit

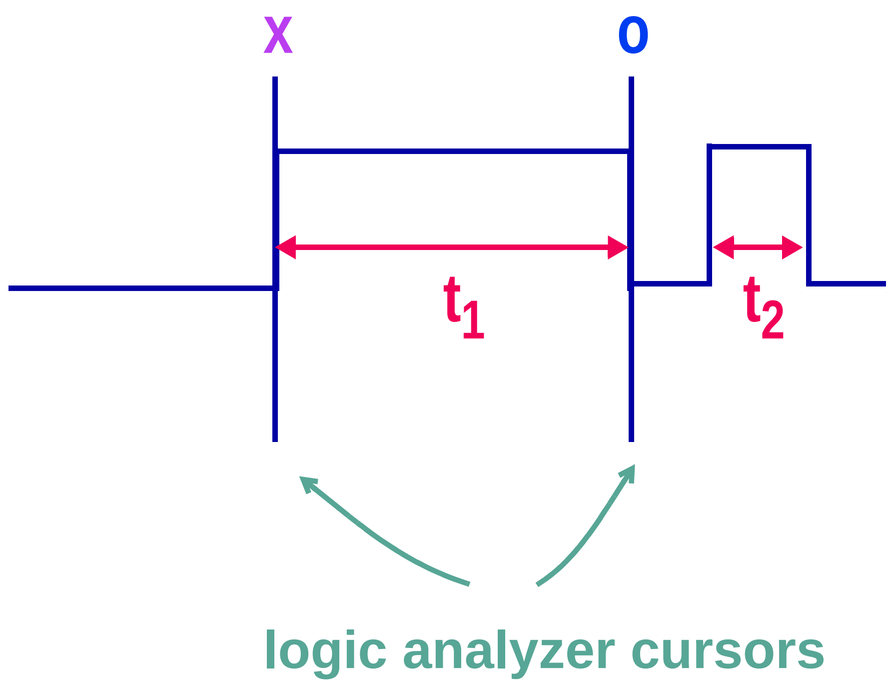
```
   // Example 1: register is shared; only bit 3 is available;
   // port is memory mapped, dioport is a ptr to it
   // 'id' is ignored, but kept for compatibility
#define MEZ_START(id) *dioport |= 0x80
#define MEZ_STOP(id) *dioport &= ~0x80

   // 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);
       :
```

Logic Analyzer
Using a Single Output Bit

Use timing mode of analyzer to measure width of pulses



x= 12 usec

o= 370 usec

t_1 =Δ= 358 usec

Logic Analyzer
Using a Bus Analyzer or Emulator

Similar to parallel I/O port, except macros setup differently.

```
    // 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 neeD0–D7.**

Setup logic analyzer to trigger on address, and store data.

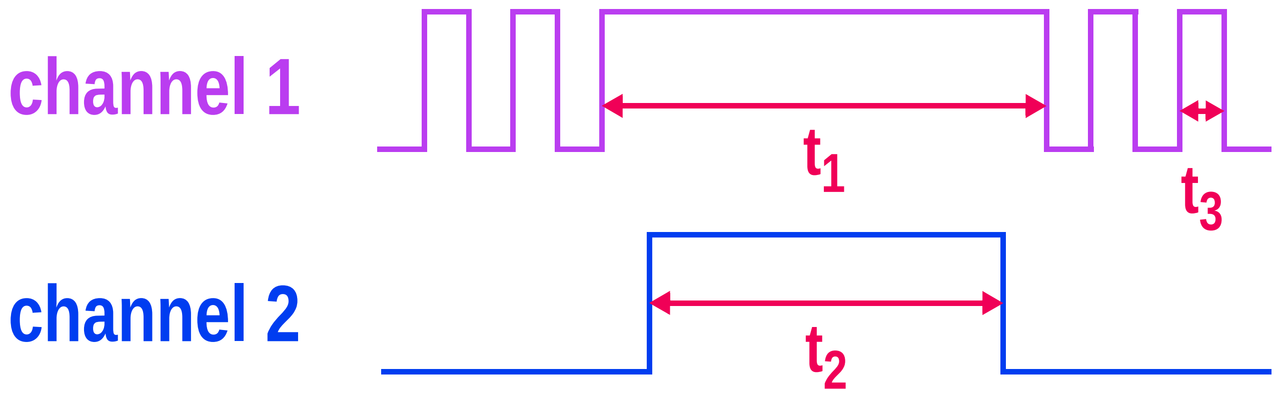
- **This provides the same data log as for the parallel output method.**

Measuring Interrupt Handler Overhead

Main code continually sends pulses to channel 1.

Interrupt handler provides a single pulse to channel 2

This results in the following timing diagram:

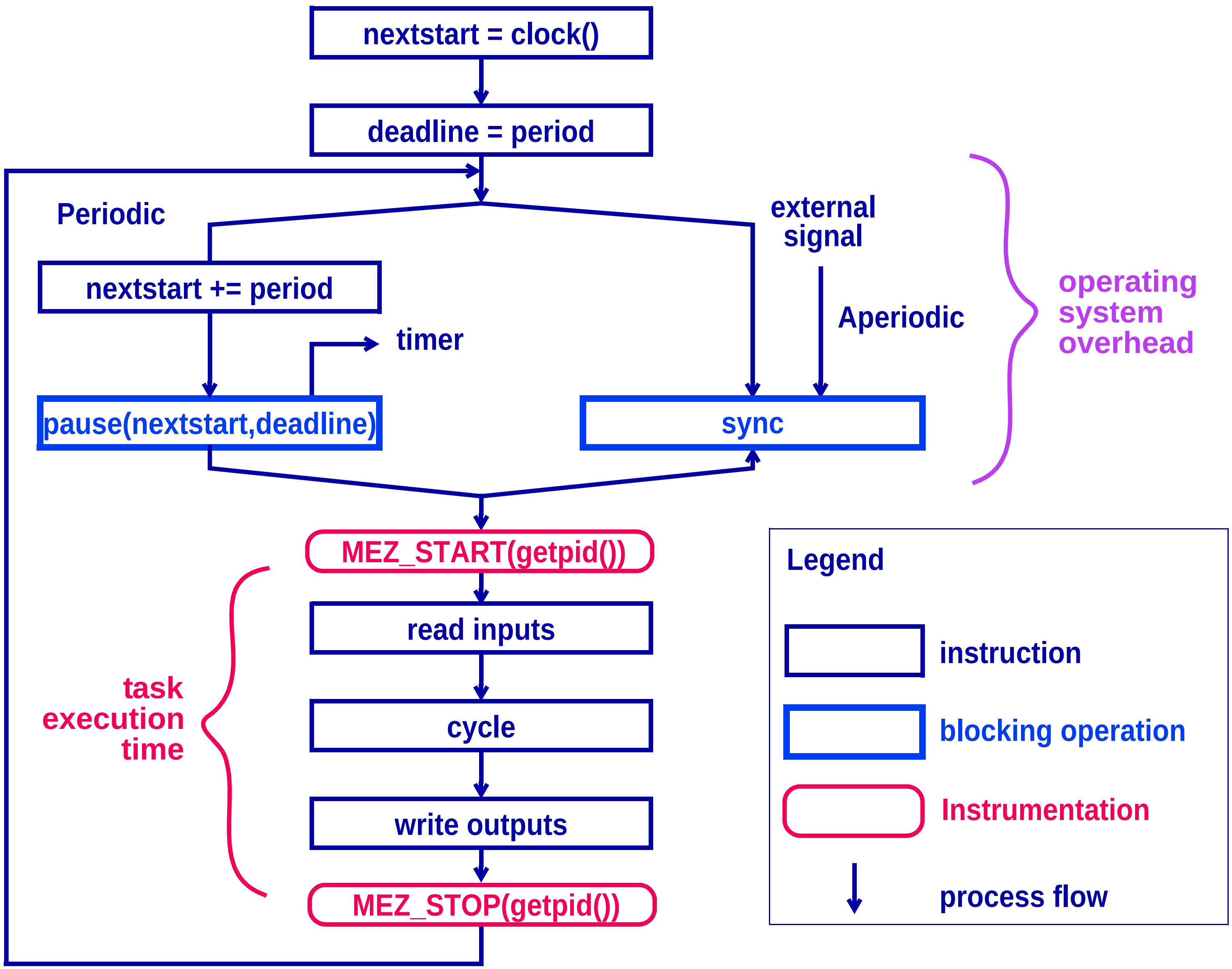


Interrupt handler execution time = t_2

Overhead to call handler = $(t_1 - t_2 - t_3) / 2$

The same method can be used to measure operating system's context switch overhead

Measuring Execution Time of Tasks
Model of a Task



Measuring Execution Time of Tasks
Sample Logic Analyzer Event Log

Task Set:

Task	Thread ID	Period(msec)	Priority
TaskA	01	0.010	High
TaskB	02	0.025	Medium
TaskC	03	0.040	Low

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

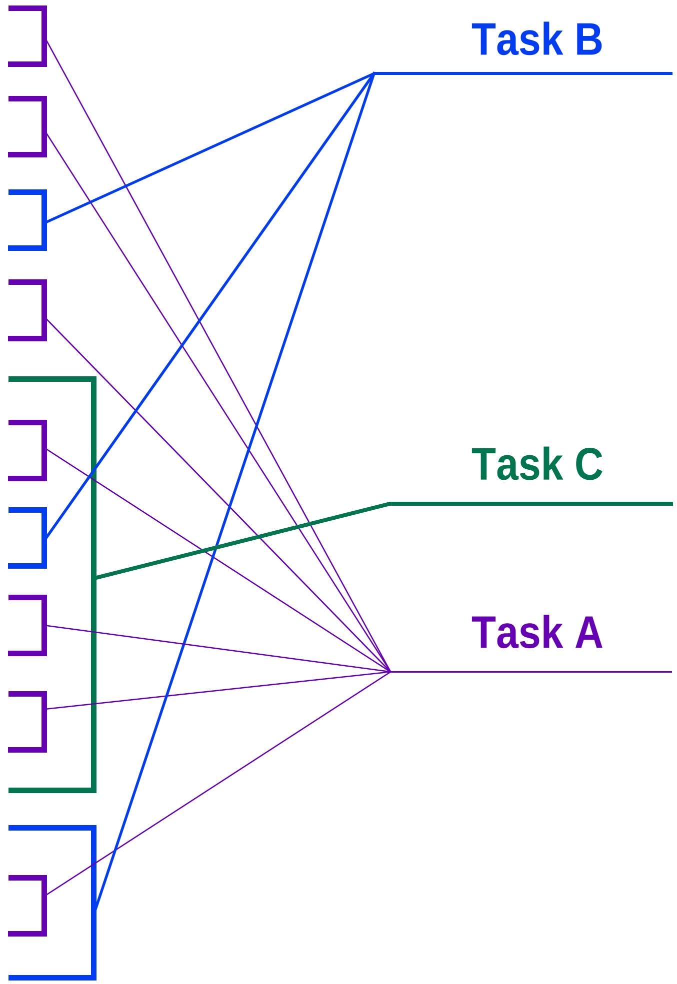
Measuring Execution Time of Tasks

Sample Logic Analyzer Event Log

Task Set:

Task	Thread ID	Period(msec)	Priority
TaskA	01	0.010	High
TaskB	02	0.025	Medium
TaskC	03	0.040	Low

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



Measuring Execution Time of Tasks
Interpreting the Logic Analyzer Event Log

Task Set:

Task	Thread ID	Period(msec)	Priority
TaskA	01	0.010	High
TaskB	02	0.025	Medium
TaskC	03	0.040	Low

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

Task B

execution time for this iteration of Task B is
5.0739 msec

Measuring Execution Time of Tasks
Interpreting the Logic Analyzer Event Log

Task Set:

Task	Thread ID	Period(msec)	Priority
TaskA	01	0.010	High
TaskB	02	0.025	Medium
TaskC	03	0.040	Low

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

Task C

execution time for this iteration of Task C is

(55.081–28.677)–(3.0669+5.0944+3.3741+3.0464)=11.8222 msec

Measuring Execution Time of Tasks
Interpreting the Logic Analyzer Event Log

Task Set:

Task	Thread ID	Period(msec)	Priority
TaskA	01	0.010	High
TaskB	02	0.025	Medium
TaskC	03	0.040	Low

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

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

An Aside: Real-Time Debugging using the Logic Analyzer

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

An Aside: Real-Time Debugging using 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:

Trace	Data	Rel Time
0	32	0ns
1	34	3.0669ms
2	37	5.9341ms

Line Number (in hex)

Another Example of a Debug Macro

Macro to output variable name and its value

```
    // first, output first two letters of a 16-bit variable name,
    // then output the value of that variable.
#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:

Trace	Data	Abs Time
0	7661	0us
1	0003	1.00us
2	6964	48.9us
3	000F	49.9us
4	6964	102.us
5	001E	103.ms

0x7661 = "va", val = 0x03

0x6964 = "id", idx = 0x0F = 15

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 outputing 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

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

Measuring Execution Time and Real-Time Performance

Class ESC 341/361

Embedded Systems Conference Boston

September 2006

Dave Stewart

Director of Software Engineering
InHand Electronics
Rockville, MD
dstewart@inhand.com
http://www.inhand.com

Recall Part I (Class 341)

Measuring Execution Time

(It’s like Counting $$$$)

Part II (Class 361)

Analyzing Real-Time Performance

(It’s like Budgeting $$$$)

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

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

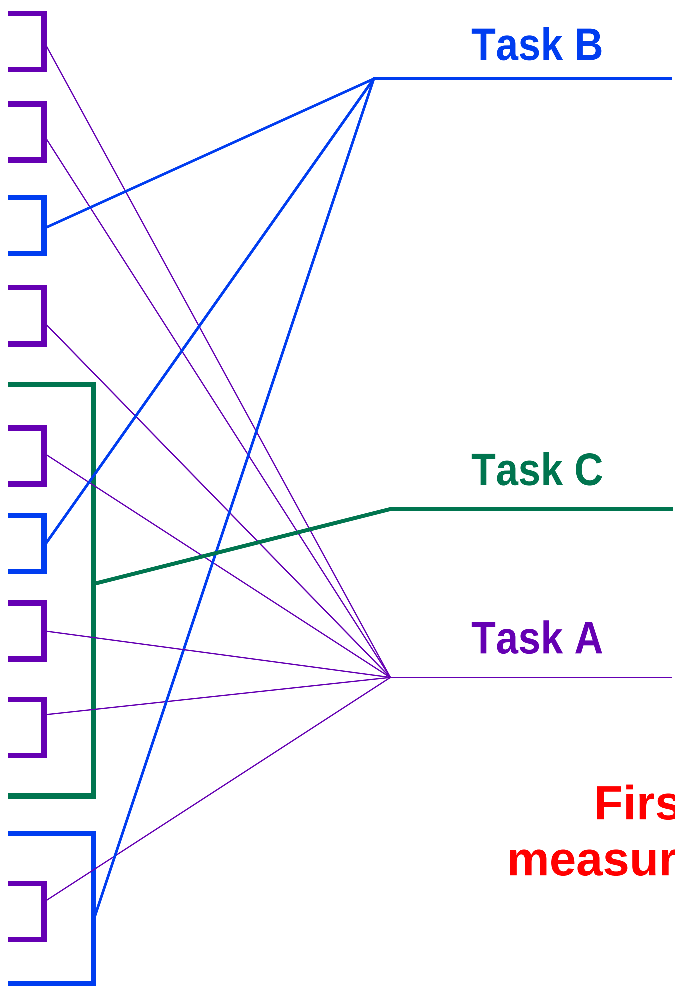
Recall Measuring Execution Time of Tasks

Example of Logic Analyzer Event Log

Task Set:

Task	Thread ID	Period(msec)	Priority
TaskA	01	0.010	High
TaskB	02	0.025	Medium
TaskC	03	0.040	Low

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



First, must obtain measured execution time by task

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:

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

Total sum of execution times = 10.183

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

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:

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

Task ID 0 is actually an interrupt handler

Total sum of execution times = 10.183

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

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 *rate=1/period*)

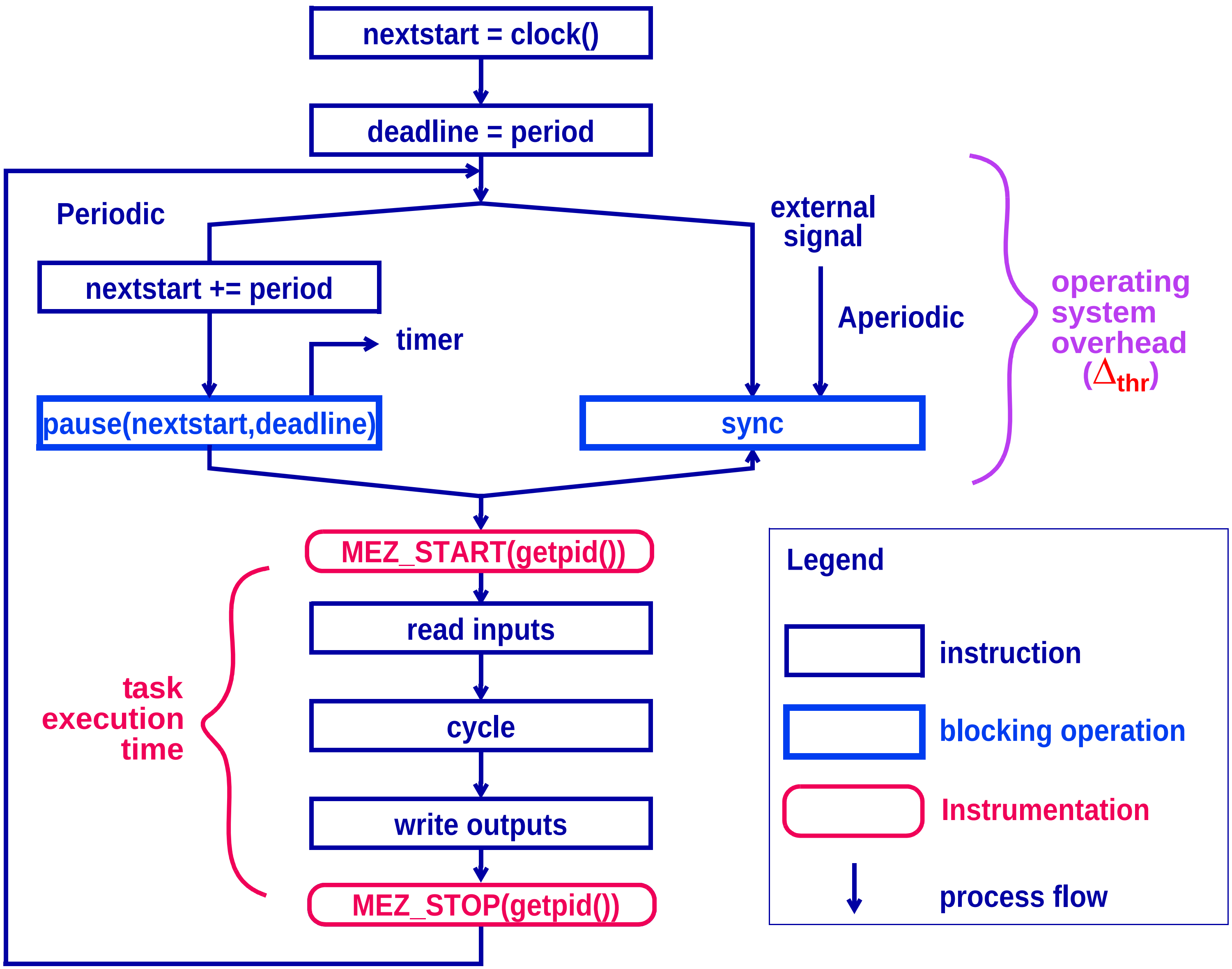
Measured Average-Case Execution Time

Measured Worst-Case Execution Time

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

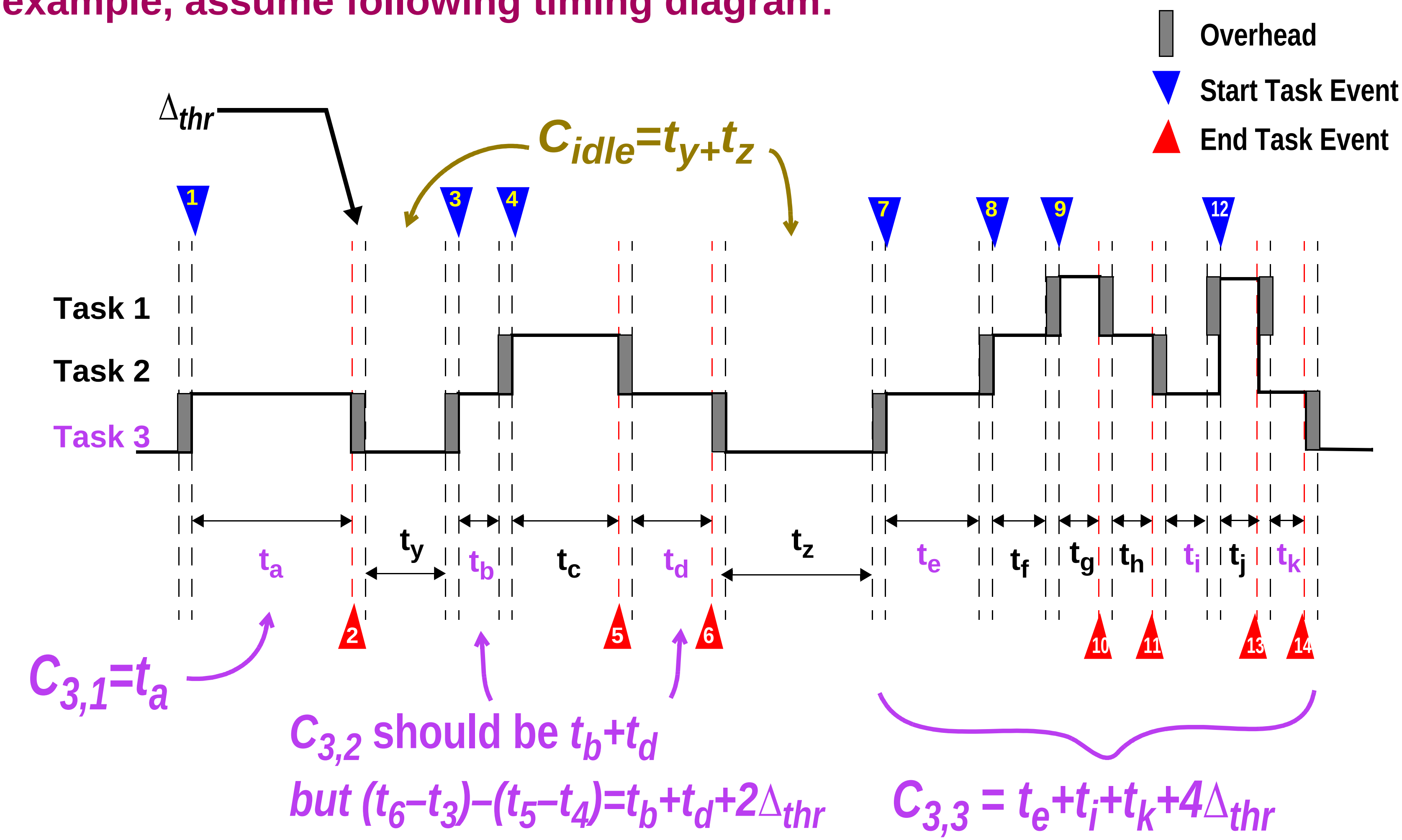
Recall Model of a Task



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:



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:

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

$C_{1,1} = t_g$

$C_{idle} = t_y + t_z + 4\Delta_{thr}$

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

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

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:

$C_{1,1} = t_g + 2\Delta_{thr}$

$C_{idle} = t_y + t_z$

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

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

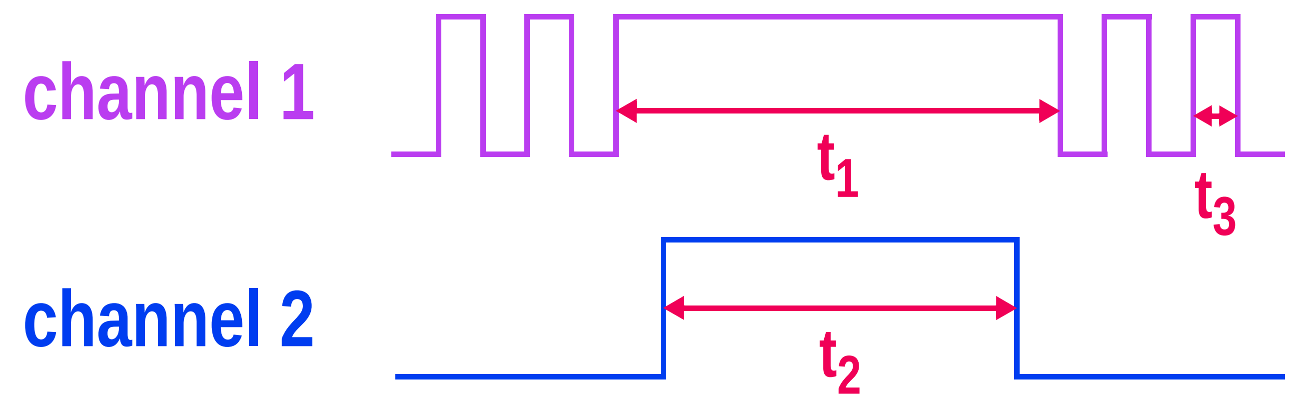
If accurate answers are needed, measurements must be adjusted to account for number of preemptions

Measuring Operating System Overhead Δ_{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:



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?

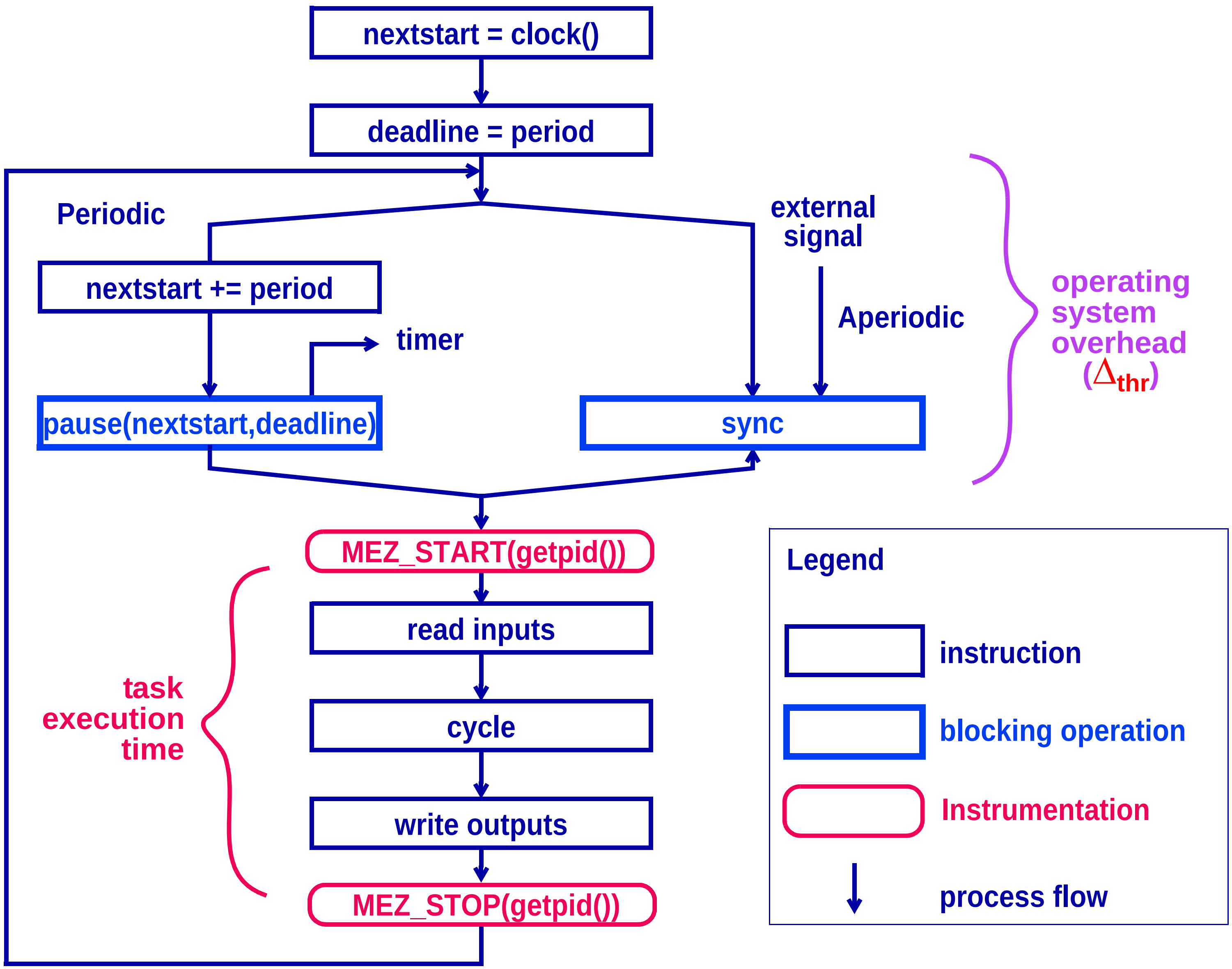
Quite often, they are not, and could be the cause of errors!

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

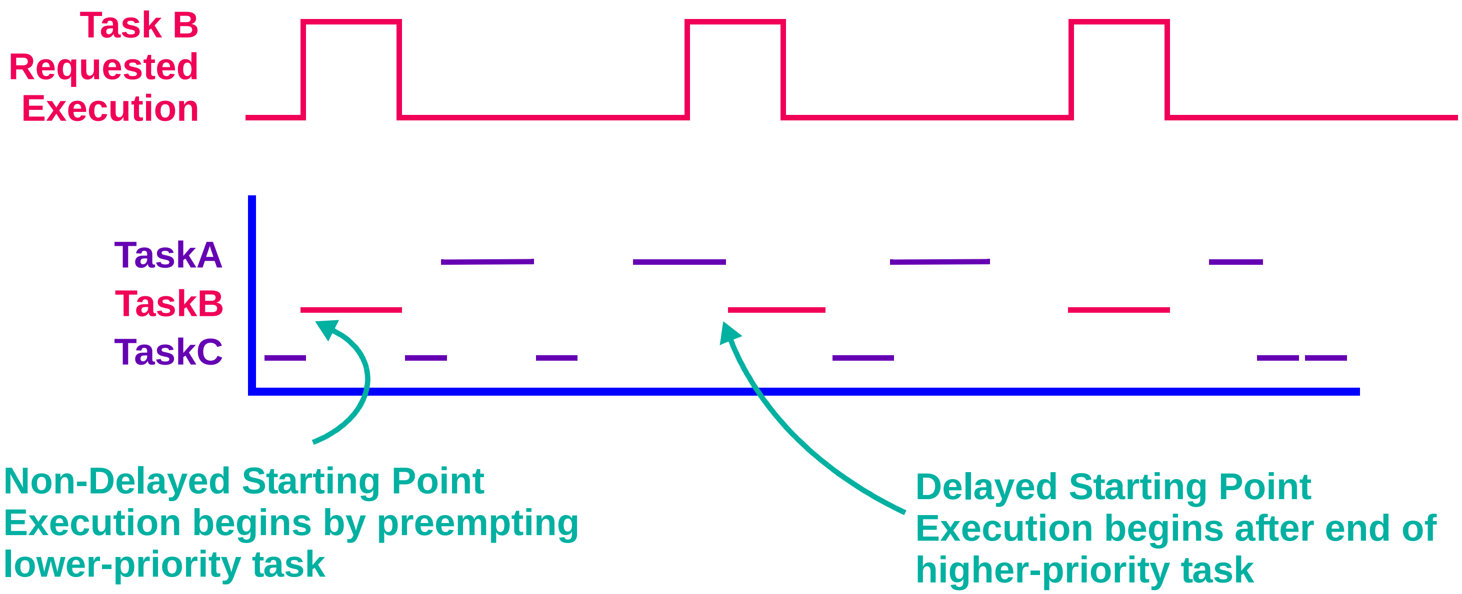


Validating Periods for Periodic Tasks
Find "Non-Delayed" Starting Points in Event Log

"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:



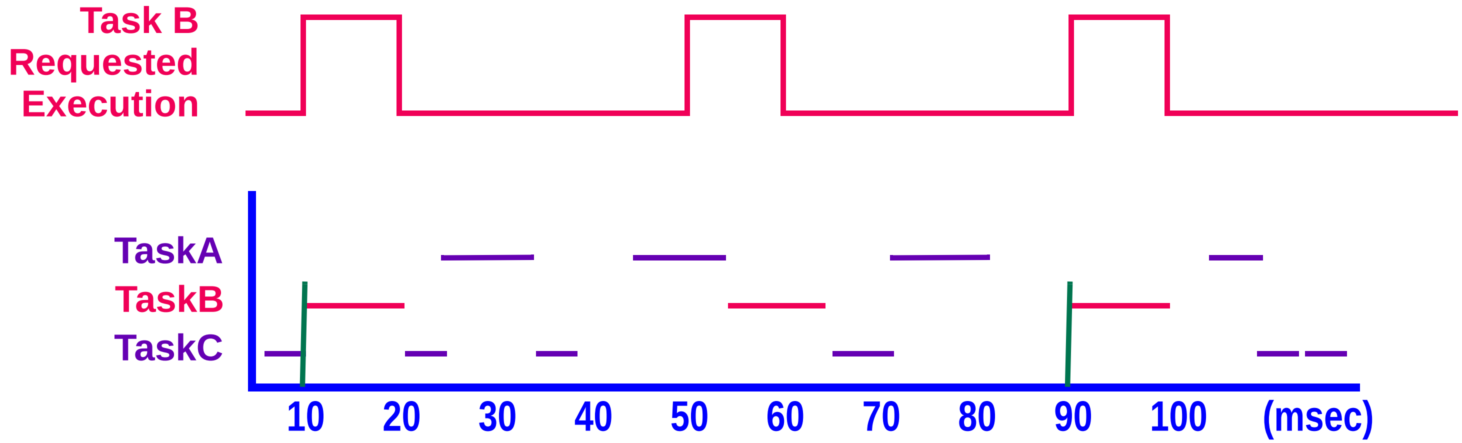
Validating Periods for Periodic Tasks
Compute Time Differences

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

Divide by (number_of_delayed_starting_points+1) between those starting point.

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

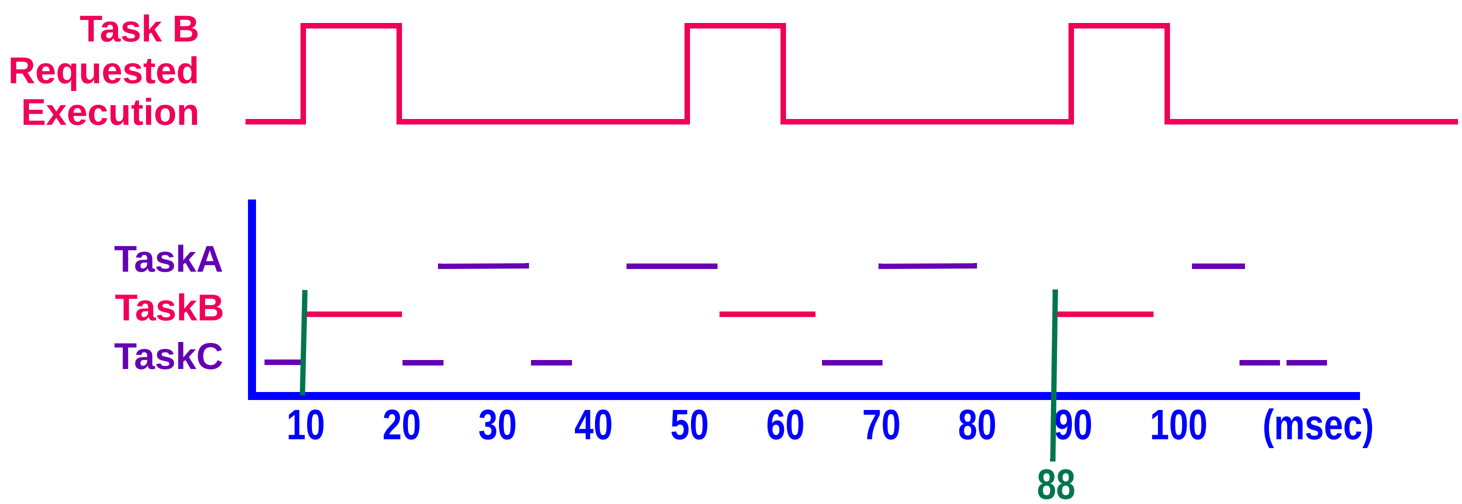


period = (90 – 10)(1+1) = 80/2 = 40 msec

Repeat for every pair of non-delayed starting points.

Validating Periods for Periodic Tasks
Example of a Problem

Suppose the following timing diagram was obtained:



period = (88 – 10)(1+1) = 78/2 = 39 msec ← **Not 40 msec as requested!**

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.

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

Practical Real-Time Scheduling Analysis

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.

Real-Time Scheduling Theory
Fixed-Priority Scheduling Analysis

A task set consisting of n periodic threads is schedulable using the Rate Monotonic Algorithm if the following equation holds:

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

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

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

Total sum of execution times = 10.183

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

Fixed-Priority Scheduling Analysis
Simplifying the Equations

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

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 $0 < t <= D_i$, use longest length of trace that is a multiple of the slowest task**

This reduces the equation:

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

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

$$\forall i, 1 \le i \le n_{thr}$$

to:

$$\left. \left(\sum_{j=1}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil \right) \right|_{t = 10.0} \le 1$$

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

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

Plug into equation

$$i = 1$$

$$\left.\left(\sum_{j=1}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil\right)\right|_{t = 10.0} \le 1$$

$$\text{LHS} = \frac{0.000056 + 0.0001}{10} \left\lceil \frac{10}{0.001} \right\rceil = 0.156$$

This means task 0 is always schedulable

Applying the Equations using Real Data
With 2 Tasks

With 2 tasks:

Total sum of execution times = 10.183

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

Plug into equation

$i = 2$

$$\left.\left(\sum_{j=1}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil\right)\right|_{t = 10.0} \leq 1$$

LHS = $0.156 + \dfrac{0.001096 + 0.0001}{10} = 0.455$

This means tasks 0 & 1 are always schedulable

Applying the Equations using Real Data
With 3 Tasks

With 3 tasks:

Total sum of execution times = 10.183

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

Plug into equation

$i = 3$

$$\left.\left(\sum_{j=1}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil\right)\right|_{t = 10.0} \leq 1$$

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:

Total sum of execution times = 10.183

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

Plug into equation

$i = 4$

$$\left.\left(\sum_{j=1}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil\right)\right|_{t = 10.0} \le 1$$

LHS = $0.156 + 0.299 + 0.322 + 0.301 = 1.079$

This means tasks 0 & 1 & 2 & 4 are **NOT** always schedulable

So What?

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?

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

Fixed Priority Scheduling Analysis
Making an Unschedulable Task Set Schedulable

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.

Part I (Class 341)

Measuring Execution Time

(It's like Counting $$$$)

Part II (Class 361)

Analyzing Real-Time Performance

(It's like Budgeting $$$$)