Measuring Execution Time and Real-Time Performance

David B. Stewart

Director of Software Engineering
InHand Electronics, Inc.
Rockville, Maryland

Email: dstewart@inhand.com
Web: http://www.inhand.com

Abstract*: Many embedded systems require hard or soft real-time execution. To ensure the requirements are met, it is necessary to measure the execution time of individual tasks, as well as establish the overall real-time performance of the system. This paper presents a variety of techniques, at both coarse-grain and fine-grain levels, to measure execution time of both user code and operating system overhead. The measurements can then be used as the basis for accurate real-time scheduling analysis, for identifying timing problems, or to know what code needs to be optimized. The coarse-grain techniques are generally software-oriented and provide measurements with millisecond resolution. They are good for quick estimates of utilization. The fine-grain techniques are more elaborate and use specialized debugging hardware or logic analyzers, to provide microsecond resolution measurements. Creating measurable code, identifying timing errors, and filtering measurement data are also discussed.*

1. Introduction

Many embedded systems require hard or soft real-time execution that must meet rigid timing constraints. Further complicating the issue is that for a variety of reasons, most of these same embedded systems have very limited processing power; it is not uncommon for them to be using an 8-bit or 16-bit processor operating at 10 MHz or less.

Real-time systems theory advocates the use of an appropriate scheduling algorithm and performing a schedulability analysis prior to building the system. Adherence to this theory alone does not lead to working embedded systems, and thus use of this theory is often dismissed by practitioners.

Practitioners, on the other hand, spend days—if not weeks—of testing and debugging hard-to-find and difficult-to-replicate problems because their system is not performing to specifications. Often, these problems are related to the system's timing, because functional testing was done using good tools, and the system usually produces a correct response.

There exists a balance between theory and practice, where proper design of real-time code enables the real-time analysis of it. Systematic techniques for measuring execution time can then be used alongside the guidelines provided by real-time systems theory to help an engineer design, analyze, and if necessary quickly fix timing problems in real-time embedded systems.

This paper discusses techniques for measuring and optimizing real-time code, and analyzing performance by correlating the measurements with the real-time specifications through use of real-time systems theory. Since this paper is directed towards practitioners, simple rules of thumb that encapsulate the knowledge of complex theories and proofs are presented. References to the proofs are provided for readers seeking more in-depth knowledge, but this in-depth knowledge is not required to apply the theory.

Several other activities of the development process can benefit from estimating and measuring execution time using the methods described here. This includes debugging hard-to-find timing errors that result in *hiccups* in the system, estimating processing needs of software, and determining the hardware needs when enhancing functionality of an existing system or reusing code in subsequent generations of embedded systems.

The first part of this paper focusses on techniques for measuring execution time, by first providing a definition of key attributes and overview of methods (Section 2), then details for using each method (Section 3). The second part of the paper focusses on real-time analysis, first presenting techniques for analyzing real-time performance (Section 4) then providing suggestions for improving software so that it is more easily analyzed (Section 5).

2. Overview of Measurement Techniques

Many different methods exist to measure execution time, but there is no single best technique. Rather, each technique is a compromise between multiple attributes, such as resolution, accuracy, granularity, and difficulty. A summary of the key attributes follows:

Resolution is a representation of the limitations of the timing hardware. For example, a stop watch measures with a 0.01 sec resolution, while a logic analyzer might be able to measure with a resolution of 50 nsec?

Accuracy is the closeness of the measured value using a given method of measuring, as compared to the actual time if a perfect measurement was obtained. If a particular measurement is repeated several times, there is usually some amount of error in the measurements. Thus, measurements could yield answers of the form x +/- y. In this case, y is the accuracy of the measurement x.

Granularity is the part of the code that can be measured, and usually specified in a subjective manner. For example, coarse granularity (also called *coarse-grain*) methods would generally measure execution time on a per-process, per-procedure, or per-function basis. In contrast, a method that has fine granularity (also called *fine-grain*) can be used to measure execution time of a loop, small code seg-

Table 1: Summary of methods to measure execution time

Method	*Typical Resolutio n*	*Typical Accuracy*	*Granulariy*	*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	subroutines	moderate
clock()	15-30 msec	15-30 msec	statement	moderate
software analyzers	10 µsec	20 µsec	subroutine	moderate
timer/counter chips	0.5–4 µsec	1-8 µsec	statement	very hard
logic or bus analyzer	50 nsec	half µsec	statement	hard

Abbreviations used throughout this paper:
sec=second *msec*=milliseconds *µsec*=microseconds *nsec*=nanoseconds

ment, or even a single instruction. Important to note is that some fine-grain techniques can also be used to perform coarse-grain measurements, although the effort in doing so could be much greater than using a coarse-grain method.

Difficulty subjectively defines the effort to obtain measurements. A method that requires the user to simply run the code and it produces an instant answer or a table of results is considered easy. A method that requires usage of instrumentation such as a logic analyzer and filtering of data to obtain the answers is considered hard. Typically, software-only methods are easier, but yield only coarse-grain results. Hardware-assisted methods are hard, but they can provide fine-grain results with high accuracy.

Table 1 summarizes the methods and attributes of each method as presented in this paper Note that in many cases the attributes are approximated or subjective, not exact values; however comparing the attributes of different methods should provide sufficient information to help choose the best measurement for a particular need. Details of each method are given in Section 3.

The method of choice can also depend on the hardware features and instrumentation tools available. For example, some methods require special hardware features like a digital output port, while other techniques require a specific software application or measurement instrumentation to be available. In some cases, the hardware or tools needed can be quite expensive and the cost and lack of availability can prevent using a particular method. On the other hand, having access to the right tools can significantly decrease the amount of effort needed to obtain needed measurements, and thus obtaining the tools most suited to a project's needs could be a worthwhile investment.

The design of the software can also have a major impact on the ability to obtain measurements of execution time, but it is not classified as an attribute, as there is no way to quantify or qualify every possible variation. In particular, the execution time of software designed in an ad-hoc manner (also known as "spaghetti code") is very difficult to measure because the starting and stopping points of the code are not easy to identify, and if there are multiple and inconsistent entry or exit points to the same piece of code, then obtaining accurate measurements is near impossible. On the other hand, software designed so that it is "analyzable" clearly has a single entry and exit point for any part of it that needs to be measured, and those entry and exit points are defined consistently for all code segments that have similar functionality. The description of each method in Section 3 is independent of the design of the software. The discussion in Section 4 assumes an analyzable code, and Section 5 provides guidelines for creating analyzable code.

2.1 Selecting a Method

To select which measurement method to use, first consider the reason for measuring execution time. The most common reasons for measuring execution time are to refine estimates, optimize code, analyze real-time performance, and to debug timing errors.

Refining estimates is usually done during the design phase or early in the implementation phase. The estimates might be used to select which processor to use, or to obtain ballpark figures on how many iterations of a particular function can be executed per second. Coarse-grain measurements can provide some of these answers fairly quickly. Sometimes, the measurements can even be made on the host processor, with an approximated scale factor for the target processor (such as "embedded processor X is about 18 times slower than host processor Y".)

Optimizing code could use coarse-grain methods or fine-grain methods, depending on what is being optimized. If optimization is at a global scale, such as deciding whether it would be faster to use arrays or linked lists in a particular application, then a coarse-grain technique to measure execution time of complete functions is usually sufficient. On the other hand, for localized optimizations, such as those that are specific to a target processor and occur during the late stages of development or when trying to fine-tune an application, a fine-grain technique that can measure execution time of a single line of code is usually needed.

Analyzing real-time performance can use a coarse-grain technique, but often only fine-grain techniques can provide the necessary accuracy. The accuracy needs to be at least five to ten times faster than the period of the fastest task. Thus, if the fastest task in the system has a period of 10 msec, then a measurement technique that provides an accuracy of at least 1 to 2 msec for functions is needed to provide fairly good answers. More accuracy is better, especially if the Central Processing Unit (CPU) is either overloaded or operating at almost 100% utilization. In these cases, a technique with microsecond accuracy is needed.

Debugging timing errors usually needs a fine-grain method with maximum resolution. It is often necessary to measure not only user code, but also real-time operating system (RTOS) code, and to detect any anomalies that might be occurring, such as missed deadlines or tasks not executing at the desired rate.

Next, details of each method are given, with examples of how each method can be used.

3. Measurement Methods

The measurement techniques that were summarized in Table 1 are detailed in this section.

The first few methods are quite straightforward, but most are only applicable to UNIX-based systems, such as most embedded versions of Linux. The *software analyzer* method is an all-encompassing description of some features provided by commercial RTOS and tools. The techniques described towards the end of this section are the ones based on hardware, and can be used independent of the RTOS. These can provide the most accurate results, but also involve the most complexity. Thus more detailed explanations are given when necessary.

3.1 Stop-watch

A stop watch is only suitable for non-interactive programs, preferably running on single-tasking systems. It can be used to measure time of things like numerical code which may take minutes or hours to execute, and when measurements only need to be approximations (e.g. to nearest second).

The method simply involves using the chronograph feature of a digital wrist-watch (or other equivalent timing device). When the program starts, start the watch. When the program ends, stop the watch, and read the time.

3.2 *Date* command

The *date* command is useful when using a UNIX-based system or any other RTOS that has a command that displays the current date and time.

The *date* command is used like a stopwatch, except it uses the built-in clock of the computer instead of an external stopwatch. This method is more accurate than a stop-watch, but has the same granularity of only being able to accurately measure non-interactive processes.

A typical way to use the command is to wrap the program that is being measured in a shell script or alias with the following commands:

```
date > output
program >> output
date >> output
```

As with the stop-watch method, this will only provide an estimate of how long the full program takes to execute. It does not take into consideration preemption, interrupts, or I/O. Most accurate answers are obtained on non-preemptive systems.

This method is useful if the output serves as a log, so that the start and end time of each execution is logged into the file. A sample use is for long simulations that run in the background overnight, and it can provide information to know precisely when it ended.

3.3 *Time* command (UNIX)

The *time* command is useful when using a UNIX-based system. Other RTOS might provide a similar command.

Execution time measurement is activated by prefixing *time* to a command line. This command not only measures the time between beginning and end of the program, but it also computes the execution time used by the specific program, taking into consideration preemption, I/O, and other activities that cause the process to give-up the CPU.

The output depends on which version of the *time* command is being used. In some cases, the *time* command is part of the shell. In other cases, it can be found in */usr/bin/time*. In each case the output is the same information, just the format is different.

For example:

```
% time program
8.400u 0.040s 0:18.40 56.1%
```

Interpreting the output, the first item (with a *u* appended, *u*=CPU), is the execution time of program, shown here as 8.4 sec. This is the amount of time the CPU was actually executing the program. Any time spent preempted, blocked for I/O, or performing RTOS functions is excluded.

The second item (with *s* appended, *s*=system), is the execution time used by the RTOS while running the program. This includes execution time for items such as device drivers, interrupt handlers, or other system calls directly associated with the program. The example shows that 0.04 sec of execution time was for system functions.

The third item is the total time that the program was executing in the system, whether it be running or blocked or waiting on the ready queue. In this case, it was 18.4 sec. This time is the about the same time that would be reported using the *date* method above.

The fourth item is the average percentage of CPU time used when the task was ready or running. The value primarily depends on the load of the system, and has little meaning as far as measuring execution time.

3.4 *Prof* and *Gprof* (UNIX):

The previous methods can only be used to measure a complete program. Many times, it is necessary to measure execution time at a finer granularity.

One method to measure execution on a per function basis is to use the *prof* or *gprof* profiling mechanisms available in UNIX. Profiling means to obtain a set of timing measurements for all (or a large part) of the code. The granularity of a profile depends on the method. In this case, both *prof* and *gprof* measure execution time with the granularity of a function. The resolution is usually that of the system clock, meaning on the order of 10 msec.

Both *prof* and *gprof* do similar things, except that *gprof* gives much more detailed results than *prof*. The measured time properly takes into account preemption, such that if a process is preempted, the clock stops until the process starts to execute again. This profiling mechanism, however, does slow down execution of the program by a non-negligible amount. So the execution time measured when using *prof* or *gprof* will be greater than the real execution time of the program when it is not being profiled. Despite this inaccuracy, the method can be useful to identify which functions in the program are using the most execution time, to identify where optimizations might need to be made the most.

To use *prof*, compile with the *–p* option then run program as follows (other compiler options can be used too, this is just an example).

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

When the program terminates, the file *mon.out* is automatically created. It is a binary file that contains the timing data by function for the program. To view the timing data, type the following:

```
% prof program
```

A more detailed profile report can be obtained using *gprof*, by compiling with the *–pg* option as follows:

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

Running the program creates the file *gmon.out*, which can be viewed as follows:

```
% gprof program
```

For information that describes the format of the statistics and what each entry means, look at the online UNIX manuals for *prof* and *gprof* for the specific operating system version being used.

3.5 Clock()

Although the *prof/gprof* method provides more detailed information then the first few methods presented, it is often necessary to measure execution time with finer granularity than a function. Suppose *prof* was used and it shows that 90% of the time is spent in one subroutine. That subroutine becomes the primary target for optimization. But if the routine includes several loops, the next step is then to identify the most time-consuming parts within that subroutine.

A possible approach is to use the *clock()* function, as provided by many operating systems, including UNIX. In this case, however, the program must be *instrumented* such that the clock is read at the beginning and end of the code segment(s) being measured. Instrumenting the code means adding lines of code explicitly to perform the timing measurements. Such lines of code are temporary, and are removed once the desired data has been collected.

This method is useful for fine-grain measurements, such as a code segment or loop, but it is not as convenient as *prof/gprof* to obtain measurements of multiple functions or processes at once. Like prof and gprof, clock is

Here is an example of a program that uses *clock()*.

```
#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\n",total);
```

There are several issues that must be taken into account when using *clock()*. The issues stem from the fact that there is no standard implementation of this function, thus it can produce different results for different operating systems. For example, it can provide a value in microseconds, seconds, or clock ticks. The reference manual for the particular operating system should be referenced prior to using the *clock()* function.

Depending on the system, *clock()* might behave differently if the system is preemptive. In some cases, if the task is preempted, the value returned by *clock()* will include the time spent by the other task too. In other cases, it will only include time used by its own process. The clock() function is certainly more useful when the implementation properly deals with preemption. But even if it does not, see descriptions in the following sections on how to deal with preemption.

It is also important to note the resolution. Even though *clock()* might report time in microseconds, the resolution is usually the same as the system clock, which can be computed as 1/*sysconf(3)*. On many UNIX systems, this is 10 msec or longer. Calling the function *sysconf()* with the argument ‘3’ returns the value of the system clock.

If more resolution than 10 msec is needed, then one of two approaches can be used:

- Create a loop around what needs to be measured, that executes 10, 100, or 1000 times or more. Measure execution time to the nearest 10 msec. Then divide that time by the number of times the loop executed. If the loop executed 1000 times using a 10 msec clock, you obtain a resolution of 10 μsec for the loop.
- Use a hardware-based method, as described in following sections.

The advantage of the loop method is that it does not require any special hardware. The disadvantage is that it forces a change in the code; the change might affect the functionality, and could even cause the program to crash. At the very least, the code slows down by the number of iterations performed just to get a reading, and thus real-time performance is lost. If this is not acceptable, then one of the other methods must be used.

3.6 Software Analyzer

The term *software analyzer* is used as an all-encompassing phrase for software tools provided by a variety of RTOS and tool vendors designed specifically for measuring execution time. Examples include CodeTest [1], TimeTrace [7], and WindView [8].

It is beyond the scope of this paper to describe how to use any such tools, or to even recommend one tool over the other. Rather, this section provides a general discussion to aid in understanding capabilities of these tools.

The first step in using a software analyzer is to determine the resolution and granularity. The resolution should be one of the specifications of the product. It can also be determined experimentally by slowly increasing execution time of a code segment, then monitoring the measured value by the smallest time increment. That typically is the resolution.

If the software analyzer is based on the system clock, then the resolution will likely be on the order of a millisecond. If the analyzer is based on some other hardware-based method, such as using an onboard timer/counter chip, then the resolution might be in the microseconds range.

The granularity is another important item to identify. Some software analyzers will be like *prof/gprof*, and only be able to provide information on a per-function or per-process basis. As with *prof/gprof*, such analyzers are good if coarse-grain measurements are satisfactory, but not very useful when optimizing localized code segments or tracking down timing or synchronization errors.

A good software analyzer will not only provide information on per-function or per-process basis, but it will also contain a means for measuring execution time for smaller segments, such as a loop, block of code, or even a single statement. The ability to measure execution time of interrupt handlers and the RTOS overhead are also a bonus.

Some software analyzers provide a timing trace to show precisely what process is executing at what time. Such a timing trace could be helpful to an expert when debugging timing and synchronization errors, but they do not offer data in a convenient format to analyze real-time performance. Also, if the timing trace is not correlated to the source code, then it is not possible to identify what part of code is responsible for extended periods of execution when such an event is detected in the timing trace. Instead, a state mode that provides tabular data that can be analyzed or download is needed.

Another issue to consider when using software analyzers are the resources used. Some analyzers add overhead, and thus slow down code. Most analyzers require lots of memory to log data, making the tool ineffective when an embedded system's memory is already fully allocated. In such cases, the hardware-based methods described below can instead be used.

3.7 Timer/Counter Chip

Most embedded computers have timer/counter chips that are user programmable. If such a chip is available, then it can be used to obtain fine-grain measurements of code segments. The method presented here, however, is not very useful for coarse-grain measurements, such as total execution time used by a function or process.

This method is similar to using the *clock()* method described in Section 3.5, in that the starting and stopping points of the code being measured are instrumented directly into the code.

At the beginning of the code segment, the current countdown (or count-up) value of the timer/counter is read. At the end of the code, the value is read again. The difference between these two values represents how many *timer ticks* have elapsed.

It is then necessary to determine the value of a timer tick. The value of the timer tick is typically a multiple of the microprocessor clock speed. It could be fixed, or user-programmable. For example, an 8 MHz microcontroller has a cycle time of 125 nsec. A timer-chip on this microcontroller has the timer tick user programmable as 1x, 4x, 16x, or 256x, depending on the bit-pattern written to one of the timer's control registers. Suppose 16x is chosen. This means the timer-tick is 16 times 125 nsec, or 2 μsec. This yields a mechanism with a resolution of 2 μsec, and usually an accuracy of twice the resolution, meaning 4 μsec. With this accuracy, it is possible to measure execution time of rather small code segments.

If an RTOS is being used, there is a possibility that the RTOS has already configured the timer/counter chip. In such a case, either use a second timer/counter chip if one is available, or use the same chip as the RTOS, but only read it. Do not change the timer configuration in any way, as that can cause the RTOS to crash.

A question arises as to where do the answers go? In the *clock()* example of Section 3.5, a print statement displayed results. But on a system in which this timer/counter method is used, there is a good possibility that a video display is not available. If a small display is available (even a simple 4-digit 7-segment LCD display), then values can be shown on the display. An alternative is to send the data out on an output port, and collect it using a chart recorder or logic analyzer. A third possibility is to store the data in memory at a known location, then to peek into that memory using a debugging tool or a processor's built-in monitor.

One issue that needs to be considered is overflow. If the timer is 16 bits, and its resolution is programmed to be 2 μsec, then it will reset and start over every 130 msec. As a rule of thumb, the method should be restricted to measuring code segments that are at most 10% of this maximum range, meaning up to about 13 msec for a 16-bit timer with 2 μsec resolution. In such a case, if the measurement is continuing on a periodic basis, approximately 1 in 10 readings will be wrong, as it coincides with the timer overflowing. That reading needs to be spotted and discarded. This is quite easy to do as long as the code segment takes about the same amount of time every time, in which case the data reading that is discarded is the one that does not make sense.

Another issue occurs in a preemptive environment or when interrupts are present. If the code segment being measured can be preempted, then false data readings will be provided every time such a preemption occurs within the code segment. Several possibilities exist. One is to disable interrupts whenever a measurement begins, then re-enable interrupts when the measurement ends. This could affect real-time performance by causing priority inversion and cause the application to not meet the specifications. But often it is acceptable to do during the testing phase in order to get the measurements of various code segments. A second alternative is to discard readings that are much longer than the average reading, as they represent measurements that include preemption.

Anytime readings are discarded, care must be taken to not accidently keep an incorrect reading and discard a valid reading. As a general rule, any discarding of data must always be done with great care. Only discard a value if there is a reasonable explanation. If there is concern that a good value might accidentally be discarded, and such a mistake cannot be tolerated, then use a different method that is not subject to the overflow of the timer chip, or more suited to account for preemption.

3.8 Logic Analyzers

A logic analyzer is one of the best tools for accurately measuring execution time with microsecond resolution, especially when accurate timing is essential. The drawback is that the it requires specialized hardware and more effort than some of the previous techniques described above.

There are two approaches to using a logic analyzer. One approach is to hook up the probes to the CPU pins. Connecting the logic analyzer to a CPU emulator or using a bus analyzer has the same effect. While this method is least obtrusive on the real-time code, it is also the most difficult, as it requires reverse engineering the code to correlate logic analyzer measurements with the source code. Some logic analyzers provide processor disassembly support, but that only provides correlation to the assembly code, and not necessarily to the source code. This approach is not advocated, as it is very difficult and does not yield answers that are any better than the other approach described next. However, a variation of this approach is to monitor only a single memory location, in which case this becomes the same as the other approach.

The other approach is to send strategic signals to an output port, which are read by the logic analyzer as events. The code is instrumented to send signals at the start and end of each code segment. The instrumentation is encapsulated within a macro, so that redefining the macro to an empty statement disables the instrumentation without the need to change any part of the application code. This approach is compatible both with large applications that use commercial RTOS and smaller systems based on custom executives or even ad-hoc code.

3.8.1. Necessary Embedded Hardware Features

To use a logic analyzer to measure code, signals must be sent from the software to the analyzer. The easier way is to use a digital output port. It is highly recommended that any embedded application is designed with at least one such port dedicated to testing and debugging. A single 8-bit or 16-bit port can be used as a gateway to seeing inside the program to save tremendous development time.

Some embedded hardware features more sophisticated windows to the inside, such as JTAG and BDM. However, each of these require additional engines to drive the mechanism, and while very useful for debugging functional code, their use can greatly affect real-time performance, and thus not recommended when measuring execution time.

An 8-bit digital output port is usually sufficient for most applications. A 16-bit port might be desirable for larger applications as it enables the encoding more information to send to the logic analyzer. If at least an 8-bit port is not available, there are other alternatives. If there is access to the CPU's address and data lines (for example, if an emulator is attached to the system), then only a single memory location on the CPU needs to be reserved. The address of that memory location is used to trigger the logic analyzer, while the data lines contain the information that would otherwise be sent to the digital output port.

A similar method can be used if a bus-analyzer (such as a VMEbus or PCIbus analyzer) is present in the system. A bus analyzer only monitors accesses to external memory that go over the bus. Thus the single memory location that is selected must be an external memory location that can be captured by the bus analyzer. A bus analyzer is in fact a logic analyzer, with all the probes permanently affixed to each wire on the bus. Therefore the techniques for using a bus analyzer are the same as described in this section when using a logic analyzer.

Even if there is only a single bit of output available or even a single serial or digital-to-analog output port, it is still possible to measure execution time, although it is much more difficult. If the output is analog, then an oscilloscope is needed instead of a logic analyzer. More details of measuring execution time with only a single bit of information or with an analog output is given in Section 3.8.4.

3.8.2. Logic Analyzer Features

A logic analyzer must be setup to capture the data being sent to the digital output port or over the address lines. However, not every logic analyzer is the same. Some key features can greatly simplify collecting data for the purpose of measuring execution time and real-time performance.

First, the logic analyzer should support state mode. That is, it displays collected data as a list of hexadecimal numbers, one line per entry in the analyzer's buffer. All but the lowest-cost analyzers usually have this mode. It is still possible to measure execution time using timing graphs, but this is much more difficult, and forces each measurement to be performed manually.

The logic analyzer should support automatic detection of transitions (often called transitional mode). That is, it monitors the data lines, and collects one entry every time it detects the output on the data lines has changed. Even some high-end analyzers do not have this capability; while other low-end analyzers do have the capability. If the logic analyzer does not support this mode, then a more sophisticated external triggering combined with setting up the analyzer in sequence mode is needed. In this paper, it is assumed that transitional mode is available.

A deep buffer on the analyzer is highly desirable. The more data that can be collected during a single execution, the more different items that can be measured, and the more measurements of periodic or repeated code. This leads to higher confidence in measurements of average and worst-case execution times. A deep buffer also increases that ability to measure rare events, like an occasional interrupt. Some logic analyzers have buffers that are one or two million events. The general rule is the more the better.

To measure execution time, only 16 channels are needed, and if only an 8-bit output port is used, then only 8 channels are needed. Most logic analyzers—even the lowest-end ones—have this many channels. For purposes of measuring execution time, additional channels are not needed.

Measuring execution time can become tedious, thus automating parts of it is highly desirable. To automate some of the data filtering described later in this paper, there needs to be a method to dump the data from the logic analyzer to a host computer. Thus some form of output from the analyzer, either Ethernet, GPIB, or high-speed serial is very helpful. Alternately, one of the newer generations of logic analyzers with built-in host computer can also be used.

A search option that enables typing in a data pattern, and displaying only the data that matches the pattern, is also very helpful for quickly viewing some results. Lack of the search option, however, does not invalidate use of the analyzer, as the same effect can be achieved after uploading data from the analyzer to a host computer.

Once an appropriate logic analyzer is selected, connecting it is straightforward. Simply connect the 16 bits of the digital output port to the corresponding first 16 channels of the logic analyzer. If an 8-bit output port is used, then only connect the first 8 channels. For simplicity, be sure that bit 0 of the output port is connected to channel 0 of the logic analyzer, bit 1 to channel 1, etc.

Next, measuring execution time for a single code segment is described. The method is then expanded in Section 4.1 for instrumenting complete tasks to measure code for an entire application at once.

3.8.3. Measuring Time for Code Segments

The following discussion assumes C or C++. If using any other language, including assembly language, it should be fairly obvious on how to adapt the method to the new language.

The first step is to setup macros for writing the output port. This step is recommended because different architectures and different output devices may require different methods of writing output. However, it is desirable to become accustom to a single set of commands.

Suppose the macros are called MEZ_START and MEZ_STOP, and a definition is created for an 8-bit output port. Following is a sample definition:

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

These definitions assume a multitasking system. *id* is an identification number that enables measuring execution time for multiple code segments at once. Each code segment is simply given a separate *id* number. This macro assumes a maximum of 16 id's (numbered 0 through 15).

The 0x50 and 0x60 codes are arbitrarily defined; they can be any number that use only the first four bits of the 8-bit value, as the bottom four bits are used for the *id*. As discussed in Section 4.1, a full profiling of an application might encompass a dozen or so codes. The encoding is quite flexible; although reserving the top four bits as the event code and bottom four bits as the *id* makes it easy to view the items in hexadecimal on the logic analyzer.

The code whose execution time is to be measured is then instrumented to include MEZ_START and MEZ_STOP macros. For example:

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

In this example, two code segments are being measured simultaneously. The first is to obtain the execution time of the function *funcA()*. The second is to obtain the execution time of the operation *y=a+b*c*.

The code is compiled. Prior to executing it, the logic analyzer is turned on, and setup in transitional mode to collect data from the output port. The code is then executed. Data collection on the logic analyzer is halted, and the output displayed. Depending on the analyzer, there could be many different columns. Two columns are most important for this task: the data column and the time column.

The data column will show the data codes that are output as a result of the MEZ_START and MEZ_STOP macros. For the above example, the data should be 0x51, 0x61, 0x52, and 0x62, in that order.

The logic analyzer automatically time-stamps every event. The timestamp can generally be displayed as relative or absolute. Relative means that the time column shows the amount of time that elapsed since the reading on the previous line. Absolute is a cumulative time. For example, assuming that *funcA()* took 358 μsec and the calculation to determine *y* took 14 μsec, the output would appear as follows (both relative and absolute time mode shown):

```
 (Relative Time Mode:)      (Absolute Time Mode)
Data     Time              Data     Time
0x51     --                0x51     --
0x61     358u              0x61     358u
0x52     3u                0x52     360u
0x62     14u               0x62     374u
```

The *u* represents microseconds; this is a typical convention used by most logic analyzers. Other common abbreviations are *n* for nanoseconds, *m* for milliseconds, and *s* for seconds.

From this output, the measured execution time is readily obtained. Relative mode is usually easiest to use if the start and stop operations are consecutive. Absolute mode is useful when nesting measurements.

Using this method, any code segment(s) in the application can be measured. Measuring individual code segments is especially helpful when optimizing code. The code can be measured prior to optimization then again after the optimization, and the amount of savings (if any) is readily known. When optimizing code, execution time should always be measured, to prevent making changes to the code that appear as optimizations, but in reality either do not affect execution time or worse, slow down execution time.

There are caveats to measuring execution time in this manner. In particular, it does not account for preemption or interrupts, and thus measured values could be misleading. Furthermore, it is incomplete, in that only a few code segments are measured. That is insufficient if trying to measure real-time performance for the entire application. Variations to using this technique follow in Section 4.1, some of which do take into account preemption and other related issues.

3.8.4. Collecting Data through a single bit

Some embedded systems are so restrictive that a spare 8-bit digital output port is an unavailable luxury. Measuring execution time can still occur with as little as a single output bit available.

The primary limitation with using only a single bit is that only one item can easily be measured at once. The MEZ_START macro is modified to set the bit to 1, while the MEZ_STOP macro resets the bit to 0.

If there is more than 1 bit, but less than 8-bits available, various encoding strategies can be used to measure more than a single item at a time. For example, with 3 bits, two of the bits can be used to identify the code segment, thus allowing four code segments to be measured at a time. The third bit is toggled as in the single-bit case, to provide measurements.

When using only one or two bits, an oscilloscope can replace the logic analyzer. Another alternative if an oscilloscope is available is to use a digital-to-analog output port. Several different clearly-distinguishable analog levels are pre-defined, with the occurrence of each one representing an event. The resolution of an analog output is generally a function of the conversion time. Values in the range of 10 to 50 μsec are not uncommon, while very accurate ones (like a 20-bit converter) could be in the millisecond range. This represents much lower resolution as compared to using digital outputs. Nevertheless, if this is the only means of sending signals from the embedded processor to a measurement instrument, then it is still better than not having such an ability.

4. Measuring Real-Time Performance

Knowing the execution time of various parts of the code is helpful for optimizing code, but it is not sufficient to analyze real-time performance. Rather, measuring execution time is a necessary step toward fully understanding the timing of an embedded system.

In this section, the logic analyzer method of measuring execution time is used as the basis for strategically measuring different parts of the code, to identify whether or not there are any timing problems in the system.

4.1 Utilization of Tasks

One of the more important values to quantify is the worst-case utilization of each task. Such utilization is used as the basis for most real-time scheduling analysis. While this section does contain some mathematical analysis, a simplified description is provided so that using the math is straightforward.

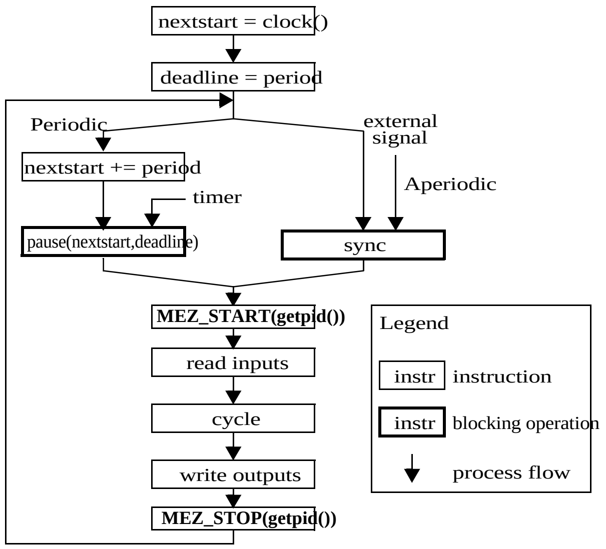
The worst-case utilization of a task (U_i) is computed as the ratio between the task's worst-case execution time (C_i), and its period (T_i). That is, $U_i = C_i/T_i$.

For periodic tasks, T_i is specified by the designer. Section 4.4.2 presents a technique for validating that the system is indeed executing a task at the proper period. If the system is specified as frequency or rate (e.g. "execute task at 40 Hz"), then $T_i = 1/frequency$ (40 Hz means T_i=25 msec).

For aperiodic tasks, which are tasks that execute in response to randomly occurring events or signals, T_i is the *minimum-interarrival time*, which means the smallest amount of time between two successive occurrences of the event. This is a feature of the application that needs to be specified if any type of real-time guarantees are to be provided.

C_i is a value that can be measured using one of the methods described in Section 3. For purposes of discussion, the remainder of this section assumes the logic analyzer method (Section 3.8) is used.

To measure execution time of a task, the task must have an analyzable design. This means that it has a definitive starting and stopping point each cycle. Preferably, there is also minimal blocking within the task. Any need to wait for input or delay providing output should be done at the beginning or end of the cycle. An example of a simple yet good model of a task is shown in Figure 1.



Equivalent Pseudocode (from [6]):

```
nextstart = clock();
deadline = period; // always relative to nextstart

while (task->state == TASK_ON) {
    if (tasktype == TASK_PERIODIC) {
        nextstart += task->period;
        pause(nextstart,deadline);
    } else {
        task->func->sync(task->local);
    }

    MEZ_START(task->id);
    ReadInputs()
    task->func->cycle(task->local);
    WriteOutputs;
    MEZ_STOP(task->id);
}
```

Figure 1: Basic structure of periodic tasks and aperiodic servers that our method assumes for purposes of automated profiling and analysis. For details on creating code that corresponds to this structure, see Section 5.3 and reference [6].

The instrumentation points shown in the figure indicate the best place for placing the MEZ_START and MEZ_STOP macros. Every time the task is executed, data points will be logged on the logic analyzer. Of course, doing this for every task and looking at the results of the logic analyzer can be very tedious. Thus it is recommended to automate as much of the process as possible.

Instead of instrumenting specific code segments, the MEZ_START and MEZ_STOP macros are used to instrument a framework, rather than any of the application code. Figure 1 illustrates the model of a real-time task that enables instrumenting a framework. The task is defined by a collection of functions, including the *cycle()* and *sync()* functions [6]. The corresponding framework code in C is also shown. MEZ_START is placed at the beginning of each iteration of the task, before a task's input is read and its cycle routine is called. MEZ_STOP is placed after the cycle routine is called and after outputs have been written. The task's ID is passed as the *id* parameter to the macros. See see Section 5.3 and [6] for more details on implementing a framework that corresponds with this model.

The code is then executed. Assuming a preemptive environment with three tasks, Figure 2 shows a sample event log, as collected by the logic analyzer. On its own, the information is quite cryptic. In this section, the discussion focusses first on interpreting this data, then on how the data can be used as input to more complex and accurate analysis.

The markings on the right-hand side show how to interpret the data. A MEZ_START operation is indicated by a code 5*x*, where *x* is the *id* of the task, and the MEZ_STOP operation is indicated by 6*x*. If a preemption occurs, the start and stop markers will be nested, as shown starting at line 8 where Task C gets preempted.

For tasks that are not preempted, it is straightforward to read the execution time, using the relative time. For tasks that are preempted, for example task C starting a line 8, the execution time is the difference between the absolute stop marker and the start marker, minus the execution time used by other tasks during that time period. In the example, the task C stop marker is at line 17. The execution time of task C is thus computed as (55.081–28.677)–(3.0669+5.0944+3.3741+3.0464) = 11.8222 msec.

It is important to note that the measured execution times include most RTOS overhead, but not all of it. As a result, the measured times could contain some inaccuracy. In fact, the lower the priority of the task, the less accuracy it has, because the overhead will affect the execution time of the lowest priority tasks more. Further explanation is given in Section 4.2.

If most or all the tasks are periodic and the system is executed for several seconds or minutes, it is possible to capture many instances of execution time for each task. This enables obtaining more measurements of each task, just in case the execution time of any task is not constant.

The number of measurements, however, quickly becomes too cumbersome to view manually. The following equation provides an estimate of how many lines will be in the logged by the logic analyzer, assuming sufficient memory in the analyzer:

$$E = \sum_{i=1}^{N_i} \frac{2}{T_i} + (1 + A_b) \sum_{i=1}^{N_t} \frac{2}{T_i} \qquad (1)$$

Task Set:

```
Task     ThrID       Type     Period(msec)     Priority
TaskA    01          T        0.010            High
TaskB    02          T        0.025            Medium
TaskC    03          T        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
```

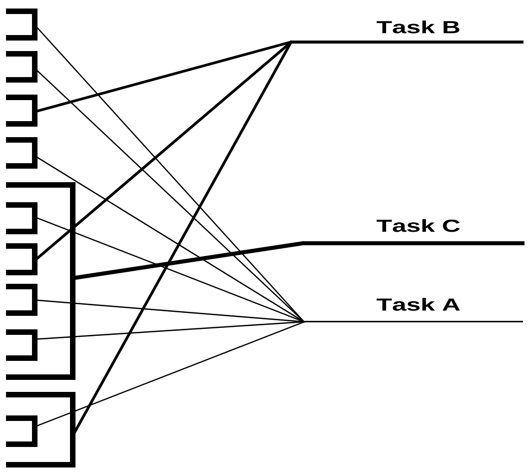


Figure 2: Sample event log collected by a logic analyzer. The real-time behavior (Note that unused fields from the logic analyzer output are not included)

where E is the number of events to log per second, N_i is the number of interrupt handlers, and N_t is the number of tasks. A_b is the average number of times that each task calls a function that provides event log information. If only logging beginning and end of the task, then A_b=2. Other events can be logged, such as beginning and end of interprocess communication, in which case A_b is increased accordingly.

Since A_b is an approximation, E is at best an approximation. For example, an application with a 1 msec system clock interrupt handler, four tasks with periods of 5 msec, 10 msec, 30 msec, and 100 msec, and on average two calls to functions that generate event logs would require approximately 4K of trace buffer on the logic analyzer for each second of profiling that is desired. With a 1 MByte trace buffer, 256 sec (or about 4.5 minutes) worth of profiling can be performed. With more tasks or higher frequency tasks, the length of the profile would decrease accordingly.

Extracting data from an event log to produce accurate results is not meant to be done manually. Rather, the event log is uploaded from the logic analyzer to a host workstation, and a program written to extract the data automatically, then compute the execution time of each task. Since the program to convert from the logic-analyzer system to a list of execution times for each task is specific to several factors, including the data format produced by the logic analyzer, the format of the application code, and the definitions of the instrumentation macros, it is not possible to provide a general program to perform the functions. Writing the program to automatically extract data is also a good exercise for truly understanding the behavior of a real-time system.

After automatically processing an entire buffer, output similar to that shown in Figure 3 is desirable.

The output shows the results of extracting data for eight tasks. The C_{ref} column is the estimated C_i that was used in analysis. C_{max} is C_i as measured with the logic analyzer. C_{avg} is the average case execution time. C_{avg} is useful when considering soft real-time systems, when it is okay to occasionally miss deadlines. For such a system to not be overloaded, the average case must be schedulable, even if in the worst-case the CPU is temporarily overloaded.

It is important to note that C_{max} is not guaranteed to be the absolute worst-case execution that a task will ever encounter. Rather, it is simply the worst-case execution time for the time period collected on the logic analyzer, which in this case is 10.18 sec of data. Thus, when measuring code, it is important

```
Total sum of execution times = 10.183
ID   Cref      Tref      Cavg      Cmax     R-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
```

Figure 3: Desired measured execution time information for performing a schedulability analysis. This information is extracted from a logic analyzer event log after collecting data as described in this Section 4.1.

to try to execute the code in a state that produces the worst-case execution time. At the same time, it is important to keep in mind that this method is not foolproof, and errors are possible. Real guarantees of worst-case timings would only occur by using formal verification techniques, but those types of techniques are not yet available for practice, as research has not yet yielded good solutions. Alternately, a timing error detection mechanism, as described in [5], could be used as a basis for identifying over long periods of time (e.g. days or weeks) whether or not C_{ref} is ever exceeded.

The column *R-Deadline* is a count of how many real-time deadlines were missed. In this example, tasks 3 and tasks 5 through 8 all missed deadlines. This implies the CPU was overloaded. Details on extracting missed deadline information from an event log is given in Section 4.4.

4.2 The Effect of Operating System Overhead

Extracting measured execution time from a logic analyzer log as shown in Figure 2 is fairly straightforward. However, in preemptive environments, the effect of RTOS overhead for context switching and scheduling must be considered if very accurate measurements are needed. This section provides a detailed analysis of the overhead and its effect on measured execution times.

When reading measured execution time from the log in Figure 2, it is easy to assume that the overhead is included in the measurements, and evenly distributed across each task. The problem is it is not possible to evenly distribute the overhead.

Specifically, the execution time of task i, C_i, is calculated as t_{end}–t_{start}–$t_{preempt}$, where t_{end} is the time the MEZ_STOP macro executed, t_{start} is the time the MEZ_START macro executed, and $t_{preempt}$ is computed as the amount of time that another task with higher priority executed (i.e. for the example in the previous section, this would be the value (51.975–29.114).

These measurements, however, already include the RTOS overhead, but not in a consistent manner. Whenever a high priority task preempts a low-priority task, the preemption overhead (Δ_{thr}) is added to the execution time of the lower priority task, assuming that the code was instrumented to output to the logic analyzer immediately when it begins executing a task's period, and again immediately when the task's period ends. This is contrary to what is desired: the overhead should be associated with the higher priority tasks. If the overhead is associated with the higher priority task, then it can be modelled as $2\Delta_{thr}$ for each task. [2]

This is best demonstrated by example. Consider the schedule shown in Figure 4. The goal is to accurately measure C_1, C_2 and C_3, knowing that there is overhead during each context switch. The event log includes the start and stop timestamps at the instants indicated by the triangles. From this diagram, the execution times should be computed as following (where $C_{x,y}$ means task x, cycle y):

$C_{1,1} = t_g$ $C_{2,1} = t_c$; $C_{3,1} = t_a$
$C_{2,2} = t_f + t_h$; $C_{3,2} = t_b + t_d$
$C_{idle} = t_y + t_z$ $C_{3,3} = t_e + t_i + t_k$

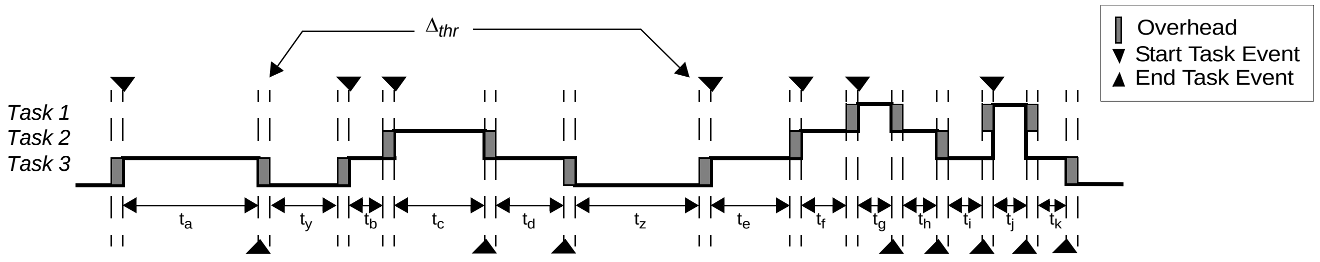


Figure 4: Schedule showing the effect of overhead on measured execution times.

However, these are not the results that would be extracted from the event log. Rather, the results would include the overhead Δ_{thr}, and result in the following measurements:

$C_{1,1} = t_g$ $C_{2,1} = t_c$; $C_{3,1} = t_a$

$C_{2,2} = t_f+t_h+2\Delta_{thr}$ $C_{3,2} = t_b+t_d+2\Delta_{thr}$

$C_{idle} = t_y+t_z+4\Delta_{thr}$ $C_{3,3}= t_e+t_i+t_k+4\Delta_{thr}$

Note the inconsistency of the overhead becoming part of the measurements. If Δ_{thr} is zero or very small, then it is negligible, and the correct values are read. However, consider the case of a typical RTOS on a 16-bit processor, where Δ_{thr} is about 50 μsec. Already the measurement for $C_{3,3}$ is wrong by 200 μsec. Suppose the system consisted of 7 or 8 tasks. The overhead can easily contribute to errors on the order of milliseconds: suppose there was a task with low priority that was preempted 40 times. That would mean its measured execution time is $C_k+80\Delta_{thr}$, which is much higher than its true execution time.

On the other hand, suppose that the start task event was logged *before* the overhead, and the end task event was logged *after* the overhead. Then the following measurements are obtained:

$C_{1,1} = t_g+2\Delta_{thr}$ $C_{2,1} = t_c+2\Delta_{thr}$ $C_{3,1} = t_a+2\Delta_{thr}$

$C_{2,2} = t_f+t_h+2\Delta_{thr}$ $C_{3,2} = t_b+t_d+2\Delta_{thr}$

$C_{idle} = t_y+t_z$ $C_{3,3} = t_e+t_i+t_k+2\Delta_{thr}$

The overhead for each task is now constant, such that the measurements conform to the overhead model presented in [2]. Furthermore, the overhead of the idle task, C_{idle}, is now properly reflected, and not inflated by the overhead. The time measured for the idle task represents the amount of available execution time. Since overhead is CPU time used by the RTOS, it is not available CPU time.

Unfortunately, profiling data with these event log points is not practical to implement. Specifically, to achieve this type of event log output, which task is being swapped in needs to be known before it is even selected, as part of the overhead is selecting the task. On the ending side, when a period ends, it is feasible to postpone outputting the event log until after the overhead. This adds a overhead on each event log, as a stack needs to be maintained to keep track of the ID of the previously running task to output to the logic analyzer.

The alternative is to recognize the overhead exists, and adjust the measured execution times are extracted from the data log. Assuming that the overhead is constant, when extracting the data, adjust the execution time of each task by keeping track of the depth of preemption. The extractor adjusts the execution time such that $2\Delta_{thr}$ is spread uniformly across every cycle for every task, rather than non-uniformly. This translates to adding $2\Delta_{thr}$ to the measured execution time for any task that is never preempted, and subtracting $2(n_p-1)\Delta_{thr}$ for any task that is preempted, where n_p is the number of preemptions that occur between the start and stop event tags. With this adjustment, measured data for each task will be more accurate.

There still remains an approximation, as there is no guarantee that the RTOS overhead is constant. For example, many real-time schedulers maintain linked lists of ready tasks sorted by priority. The more tasks in the queue, the longer it takes to insert a new task into the ready queue, and thus the more overhead. Nevertheless, assuming a constant overhead Δ_{thr} produces results that are quite accurate as compared to not taking into account the effect of overhead.

4.3 Measuring Operating System Overhead

To perform the manual adjustments to measured execution to account for overhead, as described in the previous section, it is necessary to know the overhead. This section presents a technique for measuring Δ_{thr} and Δ_{intr}. Δ_{thr} is the overhead of context switching from one task to another (including scheduling), and Δ_{intr} is the overhead incurred due to being interrupted by an interrupt handler. Δ_{thr} and Δ_{intr} are used in the calculations of Section 4.5.

An easy way to measure overhead is to build an application of a few periodic tasks that do nothing more than toggle bits on an 8-bit digital output port. This is the same port used for collecting data using the logic analyzer. In this case, however, set the logic analyzer to collect all data and to display timing diagrams instead of state values.

Suppose there are two tasks each at a different priority. The slowest task (in this case connected to channel 1 of the logic analyzer) would get preempted by the higher task (connected to channel 2), yielding a timing diagram as shown in Figure 5.

The RTOS overhead can be approximated as

$$2\Delta_{thr} = t_1 - t_2 - t_3. \qquad (2)$$

The reason t_3 is also subtracted from t_1-t_2 is that during the long pulse of t_1, some code of t_1 was executed: the code to

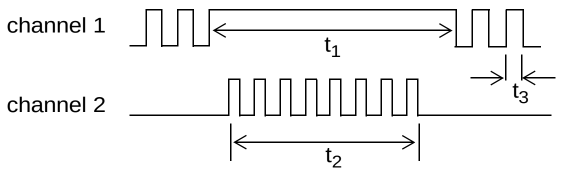


Figure 5: Measuring the context switch overhead.

produce one pulse, hence the time for t_3. It is possible that the pulse on channel 1 is during the low cycle, that is when channel 1 is 0. In this case, t_3 should be measured as the length of a 0 pulse instead of the length of a 1 pulse.

The context switch overhead includes time for the RTOS to perform scheduling. Quite often, the time for scheduling is a function of the number of tasks on the ready queue. To observe overhead with multiple tasks, create eight tasks, each toggling a different bit on the digital output port, then show the timing of all eight tasks on the logic analyzer. Anytime one task preempts another, the same calculation as shown in (2) can be performed. A better approximation of overhead with larger number of tasks can be obtained.

As with measuring execution time, there is no guarantee that exact answers will be obtained. But in general, a close approximation is sufficient to either identify timing problems and to use in the real-time schedulability tests discussed in Section 4.5.

Measuring the overhead of an interrupt handler can be done in the same way. The interrupt handler, however, only needs to toggle a bit once, to show that it has executed. The start of the pulse is then the start time of the interrupt handler, and the end of the pulse is the end time. If interrupt latency needs to be measured, then the actual hardware IRQ line should be connected to one of the logic analyzer channels. The difference in time between when the IRQ line becomes active and the pulse of the interrupt handler is seen is the interrupt latency.

4.4 Detecting Timing Errors

An important piece of information that can be extracted from an event log is the presence of timing errors. The most common timing error is a missed deadline. Each missed deadline corresponds to a problem with meeting timing constraints in the system, and thus identifying them enables the designer to rapidly pinpoint problems.

4.4.1. Missed Deadlines

The event log captures the start and end times of each cycle of a periodic task. However, the event log has no knowledge of the deadlines for each task, hence it cannot identify missed deadlines. The best solution in this case is to have the RTOS detect the timing errors. My article published in January 1997 in Communications of the ACM [5] provides details for efficiently building such a mechanism into an RTOS. With such a mechanism, it is easy to log an event on the logic analyzer whenever a missed deadline is detected.

Unfortunately, most commercial RTOS do not provide missed deadline detection. In this case, it is possible to deduce the deadlines for most (and sometimes all) the tasks from the event log, assuming that the deadline is always the start of the next period.

To find all the deadlines for Task *k*, search the event log for an instance where the start of a cycle for Task *k* preempts a lower priority task. This indicates a real start of a period, rather than a delayed start due to a high priority task using the CPU. Using the time of this event as a baseline, continually add (or subtract) the task's period to this time, to obtain every deadline for the task. This method fails only if a task never preempts a lower priority task, which only occurs if the CPU is overloaded during its steady state.

When a task misses its deadline, it continues to use CPU time that belongs to itself during the next cycle, or to another task. A properly built real-time system has a backup plan, so that when a missed deadline is detected, some kind of action is taken to ensure that it does not begin a sequence of every deadline being detected. One advocated method is to complete execution for the current cycle, but intentionally skip the next cycle. This prevents the task from trying to catch up thus overloading the CPU even more than it already is overloaded. If this approach is taken in the real-time system, then parsing the event log must take into account that following a missed deadline, the next deadline is skipped. A more in depth discussion on dealing with missed deadlines in the real-time system is given in [5].

4.4.2. Measured Period

Depending on the RTOS and hardware, it is possible that code is not executing at the rate or period as specified by the user, due to limitations in the software or hardware.

As an example of an error caused by limitations of the RTOS, if a task is supposed to execute every 30 msec, but it is detected to be executing every 40 msec or every 20 msec, then obviously there is a problem in the system. This type of error is very difficult to observe through any method other than a timing analysis. If the task is running to fast, it might use critical CPU resources, and cause other tasks to fail to meet timing constraints. If the period is slower than expected, the performance of the system might be degrade; in a control system this would show up as reduced accuracy of the application. In a signal processing application, this can show up as missed packets or dropped frames. There could me many reasons for the period changing. For example, suppose a task needs to execute 400 times per second, which is a period of 2.5 msec. The RTOS uses a system clock set to 1 msec. The RTOS might round up the request to 3.0 msec, or truncate to 2.0 msec, without any indication to the system designer.

As an example of hardware limitations creating timing errors. Sometimes it is not possible to provide the exact timing base requested by the designer. For example, the definition of "one second" might be approximated. Since most timers are based on the CPU's clock frequency, and prescalar factors are generally a multiple of 2^n, it is not always possible to setup a system clock to exactly the desired rate. For example, rather than 1 msec, an RTOS might be forced to program the timer with a value that gives the closest value to 1 msec, but not exactly 1 msec. Suppose it is 998 µsec. This means that when the designer believes one second has passed, it is really 998 msec. Although this may not seem like much, it amounts to 172 sec(almost 3 minutes) per day! Suppose the software relies on the system clock to show the time-of-day, customers will get upset because their clock is gaining 3 minutes per day. Using the logic analyzer to timestamp events will expose such problems, and allow the designers to adjust their design as necessary if accurate real-time is needed. If the RTOS's timer is used, then it is possible that the timestamps in the event log are also skewed by this amount. Section 4 provides additional discussion on analyzing these types of real-time issues.

The event log does not directly store information about a task's period. Rather, the period of a task needs to be deduced, in a similar manner as deducing deadline times. Two real starting points for the task must be found in the event log, using the same procedure as finding a real starting point for detecting missed deadlines. The difference between these two points, divided by the number of iterations of the task between those two points, will yield an accurate result for the period. From this information, every starting point of the task can then be identified, and the amount of jitter of the starting times can then also be computed.

The goal of a program that automatically extracts timing information from the logic analyzer is to provide an output as shown in Figure 3. This output has all the information needed to perform accurate real-time scheduling analysis, as described next.

4.5 Real-Time Scheduling Analysis

Real-time scheduling analysis involves a mathematical verification of whether or not there are potential timing problems in the system. The equations appear very complex, but they are really quite simple to implement when all the data is available. In this section, a simplified view of real-time scheduling analysis when using a fixed priority scheduler is presented.

First, lets present the seemingly complex equation to analyze this, assuming a fixed priority, highest-priority first scheduling algorithm. The theory states that a specific task set is schedulable if and only if:

$$\forall i, 1 \le i \le (n_{intr} + n_{thr})$$
$$\min_{0 < t \le D_i}\left(\sum_{j=1}^{\min(i,\,n_{intr})} \frac{C_j + 2\Delta_{intr}}{t}\left\lceil \frac{t}{T_j}\right\rceil + \sum_{j=n_{intr}+1}^{i} \frac{C_j + 2\Delta_{thr}}{t}\left\lceil \frac{t}{T_j}\right\rceil\right) \le 1 \qquad (3)$$

This model is a result of integrating the results of several different research efforts. The derivation of the equation given in [3], but it is not necessary to understand the derivation to make use of the above equation. The following description greatly simplifies the equation.

First, lets assume no interrupts. That is, there is only periodic tasks are in the system. This means n_{intr}=0, and the equation reduces to the following:

$$\forall i, 1 \le i \le n_{thr}$$
$$\min_{0 < t \le D_i}\left(\sum_{j=1}^{i} \frac{C_j + 2\Delta_{thr}}{t}\left\lceil \frac{t}{T_j}\right\rceil\right) \le 1 \qquad (4)$$

If there are interrupts, then they are simply considered as high-priority periodic tasks for analysis purposes. The net effect is a slightly more conservative analysis, because the overhead of a task switch (Δ_{thr}) is used instead of the overhead of an interrupt switch (Δ_{intr}). Almost always, Δ_{thr}>Δ_{intr}.

The minimum function is that the test is supposed to be performed at many different check-points. However, a simplification is to simply use the end-point of the logged data. Assume that the output that was shown in Figure 3 is to be analyzed. The total time of the data log is 10.183 sec. So lets simply set the check-point to 10 sec. When the check-point is much greater than the period of the slowest task, the analytical difference is negligible: if the system is *very* close to the threshold of being schedulable and not schedulable, the analysis might mistakenly say it is not schedulable, when in reality it is schedulable. Most of the time, however, a correct answer will be given.

Setting the check-point to a fixed value of 10 sec means t=10 in (4), and the minimum function is eliminated. The equation is now

$$\forall i, 1 \le i \le n_{thr}$$
$$\sum_{j=1}^{i} \frac{C_j + 2\Delta_{thr}}{10}\left\lceil \frac{10}{T_j}\right\rceil \le 1 \qquad (5)$$

The first line states that the equation should first be tried with one task, then two tasks, and so on. Always start with the highest priority task, then when trying the equation with the next task, select the next highest priority task. In the output shown in Figure 3, it is assumed that the tasks are ordered from highest to lowest priority.

Using the data shown in Figure 3, the analysis can be performed either with the estimated worst-case execution time C_{ref} or the actual measured time C_{max}. The value T_j is the period T_{ref}. The value $2\Delta_{thr}$ is a measurement of the overhead, which is obtained ahead of time using the techniques shown in Section 4.3. Lets assume that $2\Delta_{thr}$ was measured to be 100 µsec.

The summation means that the computation is performed for each task, up to the number of tasks being tested.

So to start, assume only the first task. Lets use the estimated times as given by the column C_{max}. Substituting real values into (5) gives the following:

$$\frac{0.000056 + 0.0001}{10}\left\lceil \frac{10}{0.001}\right\rceil \le 1 \qquad (6)$$

The symbol $\lceil x/y \rceil$ is the ceiling function, which means compute x/y, and always round up. Thus 6/2 is 3, but 6.1/2 is 4. Therefore $\lceil 10/0.001 \rceil$ is 10,000. The left side of the equation computes to 0.156, which is certainly less than the right hand side value of 1. That means the system is always schedulable if there was only the single task in the system.

Repeating with the first two tasks, the equation becomes:

$$0.156 + \left(\frac{0.001096 + 0.0001}{10}\left\lceil \frac{10}{0.004}\right\rceil\right) \le 1 \qquad (7)$$

The left side now adds up to 0.455. Therefore if only the first two tasks in the system are present, all deadlines will be met.

This can be continued for 3 tasks (0.156+0.299+0.322= 0.777), 4 tasks (0.156+0.299+0.322+0.302=1.079), and so on. At some point (in this case when the first four tasks are considered), the left hand side adds up to more than 1.0 meaning that missed deadlines are expected. This is confirmed by the presence of missed deadlines as shown in Figure 3.

So how does this help? Look at the C_{ref} column instead. Repeat the calculations for just the first four tasks. It comes to

0.958. So according to the worst-case estimated execution times (i.e. C_{ref}), the task set is schedulable. But when it was measured, there were missed deadlines. Comparing C_{ref} to C_{max} shows clearly that the measured execution time was greater than anticipated, since $C_{max} > C_{ref}$ for three of the tasks.

Once this information is known, improving the system can be done systematically, rather than through many iterations of trial-and-error.

Lets suppose the goal is to fix the system so that the first four tasks (PID 0, 1, 2, and 3) are guaranteed to meet their deadlines. The first step is to revise the estimates C_{ref} using the value C_{max}. This provides much more accurate estimates of the worst-case execution time.

To make these tasks schedulable, it is necessary for the left-side of the equation (computed as 1.079 above) to be reduced down to a value below 1.0. This can be accomplished by either decreasing execution time C or increasing period T for one or more of the tasks:

- Decreasing execution time means that the task will need to be optimized so that it uses less resources. As a general rule, the more CPU time a task uses, the easier it will be to optimize it. It is also easier in many cases to reduce the execution time of two or three tasks by just a small amount, rather than one task by a larger amount.
- Increasing the period means a task will execute less often. Whether or not this is acceptable is dependent on both the application and the hardware. Review the reason that a particular period was selected for a task (remembering that period = 1/rate). If it is okay to increase the period, then that is a much easier solution than optimizing the code to reduce execution time.

By simply repeating the math, precisely which tasks need to be optimized, and by how much, can be determined. Furthermore, a fine-tuning of the periods of each task can also be performed. The computed C_{ref} to make the system works provides a goal to the person optimizing code, so that when the goal is achieved, the system will work and the optimization effort can end.

Note that the analysis is performed using worst-case execution time. Quite often, the tasks use less than the worst-case, as indicated by C_{avg}. So it is possible that on average everything can execute, but in the worst-case it will not. Such a system is called a soft real-time system. Using the above analysis techniques, it is possible to determine which tasks will always execute (i.e. which are hard real-time), and which tasks might miss occasional deadlines.

If interrupts are present, then use (1), which takes into consideration interrupt handling as well. Each interrupt is simply treated as a high-priority task. The period T_{ref} is set to the minimum interarrival time, and C_{ref} is the worst-case execution time of a single invocation of the interrupt handler. If a task set is scheduled using a different scheduling algorithm, then there likely exists an analytical solution that can be used in practice. [3]

5. Designing Real-Time Software for Analyzability

While in theory the execution time of any part of embedded software can be measured, a more important question is whether or not those measurements are meaningful in the global sense of analyzing real-time performance. Simply measuring execution time of each function or code segment does not necessarily provide the answers needed to adequately determine real-time performance.

In this section, guidelines are given that enable the design of software that is more easily analyzed. The guidelines are based on the premise that the more accurately code reflects the ideals of theory, then the more likely existing proven theory can be effectively used to analyze the code.

Many situations occur in embedded systems that are not adequately reflected in most real-time systems theory. In these cases, practical solutions that produce functionality are kept, but the theory is expanded with approximations that take into account the typical practices. Using these approximations, analysis of measured code can still yield accurate results, even if the results are not perfect. An example is the consideration for RTOS overhead described in Section 4.5.

5.1 Ideal Systems

According to most real-time scheduling theory, an ideal system consists of a number of independent periodic tasks. The reason is that such a collection of tasks results in the most predictable, easy-to-analyze systems. An independent task set means each task can operate fully on its own, without any interaction with any other task set. This means no synchronization, no communication, and no sharing of resources. Periodic tasks repeat the same code at regular intervals, thus the exact time and length of execution of each task can be determined analytically, to foresee whether or not problems in meeting timing constraints would occur.

The problem with an ideal system of independent periodic tasks is that it is not practical. Real-time task sets are decomposed into multiple tasks as a good software engineering practice. To integrate these different pieces, interprocess communication is necessary. Furthermore, due to the severe limitations of the processing hardware, and the often strict requirements of the application, the sharing of resources, synchronization between multiple tasks, and the injection of aperiodic events, such as interrupts, are unavoidable. Yet each one of these significantly increases the complexity of the analysis, and often yields theoretical answers that say the task set is unschedulable, meaning in theory it will not work properly. But in practice, engineers are faced with the reality that they must make it work!

The runtime system that enables this integration and sharing of resources, often in the form of an RTOS or executive, further complicates analysis. The runtime system adds overhead that is not accounted for in most scheduling theory, and not accurately accounted for in scheduling theory that does consider overhead.

Given the huge disparity between the ideal of theory and the reality of practice, it is understandable why many practicing embedded system engineers never apply real-time scheduling analysis. Unfortunately, ignoring real-time systems theory is a practice that leads to less reliable systems and much greater effort in producing a system that meets real-time specifications.

The alternative is to consider the ability to perform real-time analysis as one of the necessary requirements when designing the software. With this criteria, some common designs and certainly most ad-hoc code would fail, which leads to the inability to perform scheduling analysis.

Software designs that are easy to analyze are not only possible, but usually they are preferable to the alternatives. If software can be analyzed, then the amount of time to test, debug, and maintain software can be greatly reduced, because a few hours of analysis can replace days of agony in trying to replicate then fixing hard-to-find problems.

5.2 Keys to Designing Analyzable Code

There are two important keys to designing analyzable software.

The first key is to design the software so that it is as ideal as reasonably possible. This means minimizing the amount of interprocess communication, synchronization, and resource sharing. It means to minimize the use of aperiodic events, such as interrupts and signals. It also means creating software to be as regular as possible, so that as much of the system's behavior can be identified analytically.

When software is designed to be "close" to ideal, then it is much easier to capture the non-ideal aspects either as precise mathematical equations or as reasonable approximations. Real-time performance is then determined by measuring the idealized parameters, such as each task's execution time, then the small number of non-ideal parameters, such as the overhead incurred by the small number of interrupt handlers and interprocess communication invocations.

The second key is to ensure that any item that needs to be measured has a single entry point and a single exit point. Furthermore, if several tasks, objects, or functions are similar, then the entry and exit point for each of those items should be the same. For example, every periodic task has several different components, such as the initialization code, the cyclic part, and the termination or cleanup code. Designing every task so that it has an explicit initialization section or method, and explicit body of code that executes every cycle, greatly helps in making code easier to measure, and thus easier to analyze.

5.3 Creating Analyzable Code

Execution time for almost any code, whether well-designed objects or ad-hoc spaghetti code, can be measured. However, only code designed especially for analyzability can have the code for the entire application collected quickly, and real-time performance determined automatically by executing some analysis code.

While there are certainly a variety of software designs that can be analyzed, one such design methodology that we use regularly is based on the port-automaton theory of a concurrent process [4]. It has led to reliable real-time implementations that are easy to analyze. The approach is suitable both for small systems that use a cyclic or multi-rate executive, and larger systems that use a preemptive multitasking kernel. The software design is also independent of the actual scheduling algorithms used, and thus enables selection of the best scheduling algorithm as one of the design choices.

It is not necessary to understand the details or proof of the port-automaton theory. Rather, a simplified but complete explanation is presented in the paper entitled, "Designing Software Components for Real-Time Applications," [6] that is also contained in these *Embedded Systems Conference* proceedings. The paper describes designing software that has a model equivalent to the one shown in Figure 1.

6. Summary

The focus of this paper is on measuring execution time of code, and using the measurements as a basis for optimizing the application, identifying possible timing problems, and analyzing the real-time performance.

A variety of techniques are presented. One of the more effective and accurate techniques is to output data to a digital output port, then use a deep-buffer logic analyzer to collect the data. The data can then be parsed, to extract information such as execution time of each task, missed deadlines, and improper scheduling rates. Knowing this information is an important step towards fixing problems and ensuring that an embedded real-time system conforms to the application's timing specifications.

7. References

[1] CodeTest Embedded Software Test and Analysis Tools, Applied Microsystems Corp., http://www.amc.com/products/embedded_sw_test.html

[2] D. Katcher, H. Arakawa and J. Strosnider, "Engineering and Analysis of Fixed Priority Schedulers", *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, Sep. 1993.

[3] A. Secka, "Automatic Debugging of a Real-Time System Using Analysis and Prediction of Various Scheduling Algorithm Implementations," M.S. Thesis, Dept. of Electrical and Computer Engineering, University of Maryland, Supervisor D. Stewart, Nov. 2000, http://www.ece.umd.edu/serts/bib/thesis/asecka.shtml

[4] M. Steenstrup, M.A. Arbib, and E.G. Manes, "Port Automata and the Algebra of Concurrent Processes," *J. Computer and System Sciences*, Vol. 27, No. 1, pp. 29-50, August 1983.

[5] D.B. Stewart and P.K. Khosla, "Mechanisms for Detecting and Handling Timing Errors," *Comm. the ACM*, Vol. 40, No. 1, pp. 87–94, January 1997, http://www.embedded-zone.com/bib/mags/cacm97.shtml

[6] D.B. Stewart, "Designing Software Components for Real-Time Applications," in *Proc. of Embedded Systems Conference*, San Francisco, CA, Class 507/527, Apr. 2001.

[7] TimeTrace, TimeSys Corp., http://www.timesys.com/products/timetrace.html

[8] WindView, Wind River Systems, http://www.wrs.com/pdf/wvGuide.pdf.