

# Is Real-Time Ready for Prime-Time? A White Paper on What Is Missing from Real-Time Development Tools

*Real-Time system development has been around for as long as computers – more than 30 years. The modern mix of commercial RTOS and C compilers goes back at least 20 years. However these tools and their updated offspring still do not offer the most basic capabilities needed to turn real-time software and system development from an art to an engineering discipline. The lack of rigorous design, simulation, review, validation, test and update methodologies or tools is surprising, considering how long people have had to work on this problem. This white paper explores what is still missing in this field, and why it is needed. The paper concludes with definitions of 26 terms relevant to this topic.*

## INTRODUCTION

**R**eal-time systems are well defined. Most people agree that this definition includes a combination of low-latency, determinism, fast response, and the non-linear characteristic defined by a significant drop in system performance when one or more of these metrics falls below a threshold. Data- and tele-communications, as well as machine control, are classic fields that require real-time systems. In fact, most real-time applications are characterized by their primary interface to other machines (mechanical, computers, or data lines) rather than to humans.

Bugs in real-time systems take on a very different character than bugs in non real-time systems. Bugs in non real-time systems – typically a desktop single-user machine or a server with non-real time clients, or a larger computer running compute-only tasks – are primarily “behavioral”. That is, the software did not perform the function intended. Or it crashed.

Bugs in real-time systems are frequently time-oriented, rather than behavior. The right thing happened, or would have happened, but it took too long. But some real-time window closed first, causing a failure defined at the system level, or possibly lower down.

Just so we are all very clear on this problem, here are some examples of real-time failures:

The elevator slows down as it approaches a floor, but does so too slowly, causing it to miss the floor. Either the doors open when the elevator is misaligned with the floor, or the doors fail to open at all due to a mechanical safety interlock.

The control surfaces of a cruise missile are driven too slowly. Since this is a closed loop servo system, the missile goes into oscillation causing it miss its target or break up in flight due to structural stresses.

A disk controller cannot perform error-correction fast enough to read consecutive sectors. This causes a full revolution of the disk to occur between some sectors, which causes a loss of disk throughput by more than a factor of ten.

The time to process an interrupt and a context switch driven by the click of a mouse button takes longer than the time to the second click. This causes a double-click to be interpreted as a single click, which changes the meaning of the operator’s action.

Are these simple programming bugs? Would better coding, testing, review and management practices have uncovered them?

The answer is no.

The solution required is a set of tools and corresponding methodologies for real-time system development that do not exist today.

## IT’S NOT ENGINEERING

This paper is not about a need for “proof of correctness” programming. It is not a request for the industry to spend the huge amounts of money per line of code that is spent by NASA, or for medical products, or for flight-critical avionics products.

This paper is a request for tool makers and other knowledgeable people involved with real-time systems development to openly discuss these issues and needs and ideas. Then, hopefully, to recognize the profit opportunities presented. Then to create and offer suitable tools.

Real-time systems today are not engineering projects. They are medieval exercises in alchemy. Lets mix this together with that and see what happens. If it blows up, we will try something else. This ran last time, so we will do use it again. It seems to run fast enough, and hasn’t crashed for a while, so lets ship it.

In engineering, one designs, validates, simulates, builds, measures, tests, and reviews. The only one of these tasks that exists in real-time systems engineering is "build". Oh yes, there is some very limited measurement capability, too. You might think, "surely you can test the system?" But not engineering testing, where the inputs can be controlled and the outputs measured to see if they are in spec. Oh, a tiny fraction of real-time systems can be partially tested, true, by the use of extremely expensive line-protocol analyzers and even more staggeringly expensive custom built simulators (such as ISI offers).

## THE TRADITIONAL REAL-TIME SYSTEM ATTRIBUTES

All operating systems are about resource management. They manage critical resources such as CPU time (with a scheduler), memory (with a memory manager or by using hardware descriptors), the disk and screen with a file systems and GUI, etceteras.

However real-time operating systems must manage far more critical resources and they must do it within numerical bounds. The timely management or at least allocation of some resources produces a few well-known real-time metrics of performance. Let's list a few of these:

- Interrupt latency
- Context switching time
- Throughput
- Prioritization
- Percent of CPU cycles

The algorithms and architecture the RTOS uses for resource allocation and resource implementation produce a range of system characteristics that may or may not be desirable. Let's list some of these:

- Priority inheritance
- Dynamic drivers
- Guaranteed buffers
- Not getting wait-listed due to buffer overflows and resource limitations
- Code/Data locked in cache
- RAM or Flash based file system
- Hardware memory protection of tasks
- Fault catching and isolation
- Precise (hardware) exceptions
- Interlock prevention (or detection)

Unfortunately these metrics and characteristics are "OS-centric". They do not describe any attribute that a task (or process or thread) needs to guarantee its own performance.

Let us instead look more carefully at "task-centric" critical attributes.

## TASK MODELING

A task may be modeled as having:

- An input stream or sequence
- An output stream or sequence
- A list of resource requirements

The input stream is typically characterized by a small number of parameters, such as data rate. The output stream, too is often characterized by some simple numbers, such as number of images (or packets) per second, or maybe even just a yes/no decision, or maybe a certain number of screen updates, or commands to a low-level motor controller.

The resources are not particularly complex, although you won't see this list well-defined out of today's development environments. Resources may include:

- Memory
- Buffers
- Queues
- Semaphores
- CPU cycles
- Co-processors cycles
- MMU descriptors
- Cache
- Interrupts
- File system reads
- File system writes
- External bus cycles
- Specific I/O
- Calls to other applications
- Send and Receive messages
- Other Services

The task model should include, for all of its input, output, and resource needs, both quantities and rates. The rates should be in units native to the task, which may be in bits-per-sec, or Mbytes-per-sec, but more likely will be a unit such as packets-per-second, or commands-per-second, or screen-updates-per-minute.

Resource needs will typically be a count or rate, such as file-writes-per-second or interrupts-per-second.

We will call these input, output and resource quantities "task model parameters". These are not fixed numbers but in fact are bounded. The critical bound is of course the maximum. However in many systems the typical and the maximum are very far apart. So both typical and minimum parameters are also required.

Why are all these model parameters needed?

They are required for simulation, and they are also certainly needed for verification, test and review. These parameters are nothing more than the numbers that are required for an engineering methodology.

Let's look at bandwidth in a bit more detail. This parameter is the most popular way of expressing input and output requirements for a real-time task.

Our example will be a I/O processor that is processing HDLC packets coming in on some T1 subchannels. There are 24 subchannels and packets are variable length. Normally, not all subchannels are busy at once, and there is also usually idle time between packets. Nonetheless, we can easily calculate a worst case load by taking the 64,000 bit/sec data rate, multiplying by 24 subchannels, then dividing by the minimum packet size of 56 bits to produce a maximum packet

rate of 27,429 packets per second. (Note that these numbers are intentionally not correct for actual T1 and real HDLC.) However in our application we expect the typical packets to be 1024 bits. Shorter packers are usually bad or special. So we calculate a new worst case packet rate of 1,500 "real" packets per second. Only our application usually has only one or two sub-channels that need decoding at any given time, and packets rarely come in back to back, so we set a design goal of 150 packets per second. We decide, based on current and expected equipment practice that typical packet rate will be 90 packets per second. We pick zero as our minimum rate, but for another application zero may be below the minimum we wish to specify.

We would repeat this procedure for the output data rate.

A similar procedure would be used for all resources. We analyze the interrupt sources, and look at buffer needs. By running actual code in a simulator, or by making an estimate we come up with a CPU cycle range, also.

We now have an accurate model for this task. We model the other tasks similarly.

We now turn to the Real-Time System simulator. What simulator? Ahh, we have finally come to the crux of the problem. These tools do not exist. A good fraction of the task model parameters should have come directly out of our tool chain. The compiler should tell us how many CPU cycles minimum and maximum, by analyzing program flow. It should estimate cache usage. Either the compiler or the simulator will count how many calls to the RTOS are made for buffers and messages and semaphores, so those basic resource needs are known.

In a proper development environment, the code for this task would now be tested. The simulator would create input streams and monitor output streams and watch resources. It would dynamically adjust the inputs over their ranges either randomly, or incrementally, or on a Poisson distribution, or based on some other testing model we specify. It is not necessarily true that the data be accurate to perform real-time testing of this task. The simulator watches program flow to be sure that there is 100% coverage of the code. It also measures output rate and resources usage to see that it is within the expected bounds.

A behavioral simulator would not even have to execute real code. It would simply model the task based entirely on the expected parameters. This is useful when doing full system architecture testing when only a fraction of the code actually exists. Behavioral simulation is a well known design step in IC design.

The designer may wish to examine the output of the simulator and modify some of the model task parameters. When she is satisfied that these are correct, they are locked down as part of the design. The code version tools attach these parameters to the code revision. In the final deployed system, the OS monitors many of these parameters. For example, if the interrupt rate or the number of buffers requested exceeds the design

parameter, the OS records a fault. It is critical that the OS do this on a real-time, continuous, and deployed basis, because it means that the system is now operating outside of its specified and tested bounds.

Many tasks take a bounded amount of CPU time to perform their function. For example, a task that decodes MPEG data has a minimum and maximum number of CPU cycles to decode a given size image. Although this real number depends on many factors, such a cache usage and image resolution, the bounds are still well known because they are part of the specification, validation and test process. Note that the upper and lower CPU time bounds may be well outside of "typical" execution time. If this is a critical system function (say, image reconstruction for a pattern recognition based moving weapon) then it is important that this task ALWAYS have sufficient CPU time to complete its task.

The traditional method to accomplish this is to make the task a "suitably" high priority. This is bogus. It might pass all lab and field testing, yet still fail on some image data. The correct answer is to model the task with its CPU utilization set to the maximum. The RTOS should also provide a testing mode where it will "idle-up" time allocated to this task until the actual CPU cycles (say, 40%) are reached in an executing system. This allows field testing for worst case CPU utilization of this task, while still using real (and less CPU intensive) image data.

You might think that this is unrealistic. You think: what is the worst case image data? MPEG is complicated. How is the coder to know what is the worst case image data. Worse yet, suppose the project is re-using code from somewhere else that is not fully characterized, although field proven. The answer is the tools. The simulator should be able to take a true worst case path through the code. Even if the project team cannot make up worst case data, the compiler/simulator knows what it is. Not so simple, you say. How about cache? Again, the tool knows this. The simple worst case is to start with nothing in cache. The true worst case is to assume that nothing is EVER in cache, perhaps due to an infinite number of concurrent context switches. The tools should provide rational worst case CPU cycle and timing numbers.

Note that this is not "proof of correctness". We are not going so far as to insist that this tool-generated worst case CPU time is an absolute worst case. The tool may not be able to guess certain pathological deviations. Common sense still applies. However the state of the art today is that you get nothing. You feed your code your favorite baboon and say, "this one takes as long as any I've tried". So what happens when the image sensor gets really noisy and the code takes six times longer than typical. I guess the weapon gets confused and blows up over friendly territory. (This is not far-fetched. A recent missile launch in France failed for exactly this reason.)

So let's say you have done all this, the modules are modeled and coded and tested. You put it all together and it runs slower than expected. You ask your tool to compare the earlier test results with the fully

integrated system. The tool immediately points out that the cache hit ratio is way down in the integrated system. Those rules of thumb you used for the working set and the “typical” number the chip vendor gave you from UNIX compilations turn out to be way off. Your boss, who is very smart, figures out that you can combine some modules and change the task priorities to get more temporal locality. Also, lowering the interrupt priority will cause less context switching. This is what he did on the last project to improve performance.

But you know that combining modules would require re-specifying and re-testing them. The functions the modules perform are logically distinct so combining them will produce a much wider range of performance and reduce system determinism. Also, you know from using the simulator that worst case is always the having empty cache so combining modules won't help the worst case performance, even though it will improve the typical performance. You also note that lowering the interrupt priority will change the worst case latency, which is a system specification and has already been validated, although not fully tested.

Instead you insist on a larger cache, which turns out to cost only a little more. Worst case performance is unchanged, but typical performance improves significantly.

As part of the final field test you enable some global flags in the RTOS. This forces all processes to use their maximum allocated CPU time and allocate their maximum amount of memory. The RTOS also records the usage rate of all identified resources to an OS-owned log file. A number of non-critical features never get any CPU time but all critical functions work in this mode.

After the field test the customer reviews the log file, using an automated tool, against the specified typical and maximum resource needs of the tasks. All tasks are always within their limits, but some of the typical rates are off from what you originally specified. You agree to review the numbers to understand why this test may have produced some atypical resources utilization rates.

## PERSPECTIVE

The RTOS industry has had 25 years to implement the above scenario. Yet I would say that only about 5% of that exists today. VentureCom's RTX does a small amount of real-time performance measurement, for example. There are some high-end network simulators that can model system performance based on task-level parameters. But these tools are not targetted at RTOS development, nor are supplied by the RTOS vendors. Besides, they don't work as well as an old SPSS simulator I ran in 1975.

## A NOTE ON BANDWIDTH

Bandwidth is the number one metric for any real-time task. Bandwidth is a number of somethings (bits, bytes, packets, commands) per unit of time, usually seconds. Some people like to talk about latency, but latency is an internal number, bandwidth is external. Also, latency is usually a calculated (or measured) number in order to support a required bandwidth.

For example, an RTOS vendor may specify interrupt latency. But this isolated number is nearly worthless. What is the minimum? Maximum? What is it sensitive to? What do I get after this latency? My interrupt service routine (ISR) may have to make half a dozen service calls before it starts to do any real work. First, I need a pointer to my private memory, then I need to know more about the interrupt, then I have to allocate some stack space, then I have to check of some things, then I can start to process the interrupt. Operating systems that have high interrupt latencies invariably provide more built-in services to the ISR. (There are exceptions, of course.)

The only meaningful performance parameters are those that can be obtained from a simulator or real system operating in a functioning target environment.

Bandwidth is not a single number. It is actually a chart. Bandwidth needs to be specified over a time interval; particularly so for worst case bandwidth. For example, someone might say that a system has to process 240 disk writes per minute. This is not the same as 4 disk writes per second. A large fraction of bandwidth needs are non-uniform, or “bursty”. The system disk write requirements should be specified over time. For example, how many per 100 milliseconds, per second, per ten seconds, per 100 seconds, and per 1000 seconds. This allow buffer size to be properly specified and simulated.

Different applications will have very different “burstiness” yet a simple bandwidth number may completely mask this. File systems tend to not be real time.

Here is an example. An image processing module has a fixed input rate so the system specification is 24 pictures per minute. The maximum picture size is 30 sectors, so you compute a bandwidth of 12 sectors per second. The disk itself can handle 60 sectors per second so this looks like a great margin. During testing, there are no problems. Then, in the field, fatal errors occur. During examination it turns out that the image processing module was sometimes producing giant all-white images. These looked like 30 sector writes spaced very close together. The RTOS has only allocated 4 sectors worth of buffering. When its buffers go full, it wait-listed the generating task. Unfortunately this task also provided buffering for the constant rate input stream, so this stream overran the system.

The fault was in not properly modeling the disk write “bandwidth”. It turns out the “peak” bandwidth was 30 sectors in nearly zero time. This model had to be changed to include image data buffers as a managed resource. Even assigning a fixed time to those 30 sectors would be wrong. If the system spec included a “peak disk data rate” of “30 sectors per millisecond” who could deal with that?

Typical data rate does not work either. “Typical” is what you get with a Microsoft desktop OS. “Typical” means “it works most of the time for most users”. No one can build a real-time system based on “typical”.

Bandwidth should always be specified as a curve with the “somethings per second” on the vertical axis and time on the horizontal axis. The time axis ideally goes from zero to infinity, but in practice it may start and stop

where the bandwidth number becomes a constant. For example, the bandwidth over one minute may well be the same as the bandwidth over a week. As another example, a data-comm line may have a maximum data rate of 56K bits/sec. Since this is its true peak data rate, the peak data rate over 1, 10 and 100 milliseconds will be the same. The system may only be

connected for a few minutes per day, however, so the data rate per day would be a far lower rate.

## ISSUES AND DEFINITIONS

Below is an alphabetical list of issue that I have touched upon in this paper, each with a definition or comment relevant to this topic.

<b>Bandwidth</b>	A curve of rate v. time that describes the input, output, or resource usage. Each point on the curve is normally the maximum rate over the corresponding period of time on the horizontal axis. The curve should be extended towards zero and towards infinite time until it is flat.
<b>Behavioral simulation</b>	The process of simulating one or more tasks in a system looking at only tasks' model parameters. That is, the computation or data processing functions of the task are not executed, only modeled as input, output and resource rates.
<b>Bounds</b>	Bounds are part of a task's model parameters that specify minimum, typical and maximum rates for all inputs, outputs, and resources usage. The bounds are used in modeling, verification and testing. The option should exist in a deployed real-time system to do bounds checking on all rates.
<b>Buffers</b>	Buffers are a resource. They are often associated with queuing. See Queuing.
<b>Cache</b>	A well-known part of a memory architecture hierarchy that has a major impact on CPU execution performance. Cache has the characteristic that its performance for any one task is always dependent on both system history and on other tasks. Cache in a real-time system nearly guarantees that task performance is never independent of other tasks. Thus it is critical that cache usage be modeled and must be a task and system parameter in order to assure worst case system performance.
<b>Complexity</b>	The goal of system design is to manage complexity. Tools should direct assist in this.
<b>Context Switch</b>	The operation within an operating system where one task is suspended and another starts or resumes execution. Understanding context switching and accurate modeling is critical if a real-time system is to be simulated or validated for worst-case performance. Context switch time is popular RTOS metric, although it is nearly useless taken in isolation of other factors.
<b>Emulation</b>	Emulation is a real-time or near-real-time "simulation" of a task or a system or part of a system, using real data and computation. Emulation may be used in testing or validation to mimic a real-time event or external system. Emulation differs from simulation in that the operating part of the system "looking at" the emulated portion should not be able to distinguish the emulated sub-systems from the real sub-system (within some specified constraints).
<b>Hardware Descriptors</b>	Some processors have internal hardware to help manage, isolate, and protect tasks, threads, or blocks of memory. These are a finite resource, and so must be modeled.
<b>Inheritance</b>	Task Priority Inheritance is a feature of many real-time operating systems that requires (or permits) a service task to acquire during its execution the priority (and typically other rights) of the client task. Since the service task is operating on behalf of the client task, this is generally deemed to be the "correct" architecture for real-time systems and can preserve the intended execution priorities. Inheritance can be difficult to model and tends to reduce the isolation of tasks. It may also produce a (false) reliance on task priority as a method to insure worst case performance.
<b>Interrupts</b>	A change in the flow of a program, usually in response to an external event. Interrupts are historically fundamental to real-time systems. Interrupt latency is a popular (if not very meaningful) metric of real-time operating systems. Interrupts are by their definition non-deterministic and therefore somewhat difficult to model properly. Interrupts are nearly impossible to test 100%. For this reason some highly deterministic or ultra-reliable systems have done away with interrupts. Interrupts themselves and interrupt rate needs to be a managed resource.
<b>Latency</b>	Latency is the time between two events. The term is often used to describe the time between a need and the service of that need. Latency is a number "internal" to an OS or system. As a requirement it is often derived from a bandwidth number, which is an "external" parameter. As a specification it is likely to be both measured and "typical". Latency is a popular metric, but unless very well characterized it cannot be used in a model.
<b>Logging</b>	The hoped-for feature of some future real-time OS that would record the resource usage rates of the executing tasks. The log file could be used for validation and debugging. It would also be extremely helpful in determining the cause of field real-time software failures. Logging would be part of real-time performance monitoring.

<b>Maximum</b>	The critical method of specifying any parameter or rate for a task or module if one wishes to assure performance under worst-case conditions. "Maximums" are common in engineering, but rare in computer systems today.
<b>Memory Architecture</b>	The collection of hardware elements, their interconnection, and algorithms that determine the average and limits of the memory sub-system performance. Memory architecture includes such elements as first and second level cache, error detection and correction, paging, cache line size, cache replacement algorithms, pre-fetching, write merging, snooping, write posting, contention and sharing, purging, initialization, refresh, and many other factors. Modern memory systems are very hard to accurately model, but frequently a "good enough" behavioral model is not difficult. Most memory architectures today are designed specifically to produce good "typical" behavior at the expense of "worst case" behavior.
<b>MMU</b>	Memory Management Unit. Part of Hardware Descriptors.
<b>Model</b>	For a task, the specification of the input, output and resource parameters. The parameters are counts and rates, typical, minimum and maximum. For a system the model consists of the task models plus the OS, CPU, memory, other computer sub-systems, and the external processes. The model is used to simulate the system.
<b>Percent of CPU time</b>	A key resource. Real-time operating systems allocate the CPU to tasks based on priorities, interrupts, internal needs, and other factors. Ideal RTOS's would instead manage CPU time as a percent of real-time over a period of time in order to assure all time-critical tasks of guaranteed time to perform their function.
<b>Proof of correctness</b>	An esoteric specialty of computer science not used in any known commercial systems. This approach tries to use mathematics, rather than modeling, to assure correct operation.
<b>Queuing</b>	Queuing is a feature - usually in the OS - of isolating a resource from the tasks requesting them. Some queues may be thought of as a waiting line between the service and the client. All queues add delay and are finite in size. Therefore all queues in a system must be modeled and their input rate, outputs rate, and size must be parameterized and bounded.
<b>Resources</b>	The fundamental things that any operating system manages and allocates. Traditional (non real-time) operating systems never treat a rate as a resource, but real-time operating systems should consider ALL rates to be managed, allocated and bounded resources.
<b>Scalability</b>	The characteristic of an architecture that allows part of a system to grow over a large range without producing non-linear or undesirable effects on key operating parameters.
<b>Simulation</b>	Simulation is the modeling of a task, system or sub-system to observe some aspect of its performance. Simulation is frequently non-real time, and often models only a specific subset of the performance of the real task, however these are not constraints on the use of simulation. Simulation is critical for modeling and understanding the operation of a task outside of the bounds of the real world. For example, testing for correct operation in the presence of controlled errors frequently can be done only via simulation. Simulation is the only method to assure that a complex real-time system operates correctly under all of its specified bounds.
<b>Tool Chain</b>	The Tool Chain is the set of software tools such as compilers, simulators, debuggers, source control, emulators, performance monitors that are used by the software and system development team to design, code, review, debug, validate, test, deploy, monitor, maintain, update and log the software and firmware (and sometimes the hardware) in a real-time system.
<b>Tools</b>	Fancy name for software used by programmers.
<b>Typical</b>	Important for modeling, but useless for determining worst-case performance. It is important that the engineers always maintain a very clear distinction between typical rates and maximum rates.

"The opinions expressed in this paper are attributable solely to its author, and do not necessary represent the opinions of SBS or its management." ■

*Kim Rubin is currently Chief Technical Officer of the Computer Group of Companies within SBS Technologies. Previously, he was Executive Vice President and CTO of GreenSpring Computers,*

*which was purchased by SBS. Mr. Rubin is regarded as the father of IndustryPacks, the VITA 4-1995 Standard, and PC-MIP modules currently balloting as the VITA-29 Standard. He is credited with an unusual breadth of inventions, including American Express' billing system, an automated urban gunshot detection system, a gigabit reflective memory system, and a fundamental patent on ESD protection. Kim relaxes by ice skating and gymnastics.*