

# The Challenge of Developing an RTOS for DSP

**DSP applications are of the most demanding in terms of datarates and computational requirements. They also can have very diverse real-time requirements. It is important to understand the real design issues at hand to obtain maximum performance. In addition, the resulting complexity requires a careful design of the RTOS. All require a judicious analysis of the higher level as well as of the lowest level application requirements. We will give an overview of the different scheduling requirements and how they can be met.**

## THE TECHNOLOGY CONTEXT

In today's world, technology is fast changing our environment. Computers have grown from the size of a full room to a much more powerful item that we use as a portable desk. It contains all my files and using a credit card sized modem card, one can access the office files from wherever one is in the world, at least if one can find a suitable telephone plug. This is just a small illustration of what electronics has done to the way we work and the way we organize our lives.

The central shift behind all this, is the digitalization of data and the subsequent processing in its digital form. As a result, there are now processors anywhere. While most of them are microcontrollers, processing at fairly low rates, often measured in milliseconds, an increasing number of them are DSPs, processing audio and video signals at a much higher rate. While there is essentially no functional difference with what microcontrollers are used for, the main difference comes from the datarates and the complexity of the compute intensive algorithms.

As a result, DSPs are optimized to handle this kind of situation. They have large register contexts as the more data can be kept in the registers, the less wait cycles are incurred reading from and writing to the external memory. Special registers are added for executing zero overhead loops and even hardware stacks to save a cycle when making a function call. While processing, DSP applications receive data from the outside world in real-time, not seldom Mbytes/second. As a result, interrupt rates can be very high as in the range of 100 KHz. To cope with these requirements, most DSP processors have multiple interrupt priorities, shadow registers (at least one set) for low overhead context switching and a non maskable interrupt pin. It should not surprise you then that high end DSP applications (at least 50% of all that use floating point), use several DSPs in parallel. Just like scientific supercomputers, many DSP applications are in fact embedded supercomputers, offering GIGAFlops of performance. There is however an important difference: most DSP applications

also have important cost and power constraints. This means that they must run in a minimum amount of memory with a power consumption in the order of a couple of Watts. While many RISC architectures can deliver comparable performance, often by the use of heavy instruction pipelining techniques, they have no on-chip hardware for parallel processing and run at a much higher frequency, consuming about ten times more power.

Initially DSPs were often used as coprocessors, processing one stream of data at a time. This is still the

---

**The essence of an RTOS is to hide all the complexities from the application, while offering the functionality in an optimized and predictable way.**

case for low cost applications, where the (fixed point) DSP is often deeply embedded in a single chip ASIC solution. As applications became more complex and DSPs more powerful — a self fulfilling prophecy — things started to change. Applications now have multiple input datastreams, often coming in

at variable and different datarates, with the results also distributed to multiple channels. In parallel systems, interprocessor communication, with interrupt driven DMA operations, is also multiplexed with the incoming datastreams, making the programming of it sometimes very complex. To illustrate this, just remember that on today's high end DSPs like the TMS320C40 and the 21060, the interprocessor communication itself is associated with the simultaneous use of at least 12 interrupts and of 6 DMA engines. In addition there is the timer interrupt and the I/O device the application gets the data from. How to handle all this and still get the most out of the DSPs? Therefore the need for using Real Time Operating Systems (RTOS). The essence of an RTOS, besides providing a modular and scalable programming approach, is to hide all these complexities from the application, while offering the functionality in an optimized and predictable way.

The answer lies in a careful design of the RTOS, such as we tried to achieve with our own Virtuoso™ range of kernels. This resulted in a unique approach based on high level distributed semantics for ease of use and a multi-level, multi-tool approach for maximum performance with minimum overhead. This will become even more important for the next generation DSPs that are now coming to the market. Challenging

CETIA Ad

multimedia processors more and more, they deliver over a 1000 Mips, are superpipelined and can only be used to their potential in multi-channel, multi-function applications.

## SOME DEFINITIONS ON REAL-TIME

Below, we give some definitions that are essential to get around in the real-time domain. The term real-time in itself is not always consistently used. For some people it means just "fast" whereas for other people it means a reaction of the system within a well defined interval. We adopt the following definition: "A real-time system is one that not only produces logically correct results, but also produces them within the right time-interval." This means that real-time systems have safety critical characteristics and that it there will be serious performance degradation if the real-time constraints are not met. The results can range from the benign (e.g. lower audio quality) to the catastrophic (e.g. crashing airplane). Below, one finds some definitions (according to the Virtuoso concepts).

- **Hard real-time :**  
Real-time with strictly defined, absolute timing requirements.
- **Soft real-time :**  
Real-time with statistically defined requirements. (E.g. on-line transaction system).
- **Task :**  
A well defined function of the system.
- **Interrupt :**  
An internal or external event that can trigger a pre-defined action from the processor.
- **Interruptible :**  
The capability to interrupt executing code while returning to the interrupted code at the point it was interrupted.
- **Interrupt latency :**  
Time interval between the moment the hardware interrupt happens and the moment it can be acted upon (Virtuoso's definition is to take the moment the data is read).
- **Deadline :**  
A moment in time at which a certain processing action must be terminated.
- **Scheduling :**  
Arranging the order of execution of a set of processing tasks.
- **Priority :**  
Relative order of execution between executable processing tasks.
- **Round-robin scheduling :**  
Equivalent to scheduling all task at equal priority, in order of arrival.
- **Pre-emptive scheduling :**  
Capability to pre-empt a lower priority task in favour of a higher priority one.

## ALL REAL-TIME APPLICATIONS ARE NOT EQUAL, NEITHER SHOULD THE RTOS

Firstly, we should raise the question: do you need an RTOS (real-time operating system) at all? If we take a simple application with just one data input stream, one type of processing on it and a resultant output data stream, things are simple. One can even write the application with all interrupts disabled, it will work using simple polling until the data is available.

What happens if the application consists of multiple datastreams, each with their associated processing, and the data has to be produced at well defined time points? And to make things even more complicated, all datastreams having different frequencies and the processing times being stream dependent? In this case, things are no longer simple and unless one is falling back on brute force techniques, only a careful analysis and a resulting scheduling algorithm can address the application using a minimum amount of resources. To illustrate how one can find the optimal scheduling algorithm, we will start with a simple application, analyse its different components and deduct from that the essential elements of the scheduling algorithms.

### Data-acquisition

Real world data enters the DSP peripheral circuitry after A/D conversion and ends up in a memory location. Associated control logic raises a signal on which the DSP can poll or that triggers an interrupt service routine (ISR). Polling is only a solution in the simple cases where the processing can be finished before the next stream of data arrives. If this is not the case, an ISR is needed. Complications arise because the processor will automatically disable interrupts until the end of the ISR is reached. For this reason, the better DSPs (and other processors) allow interrupts to be prioritized (and nested) and have a set of shadow registers (unfortunately often just one) to minimize the context switch overhead. This allows higher interrupt rates for the most critical interrupts. If all interrupts arrive at well known frequencies, a careful assignment of the priorities will allow the timely processing of all data. If some of the interrupts arrive asynchronously or at time moments that cannot be known beforehand, this technique might fail especially if the associated processing is important. In more complex applications with parallel DSP's, some of the interrupts will be generated by the interprocessor communication and can delay the originating processor. Hence the need to disassociate the data-acquisition and the asynchronous interrupts from the processing. This introduces the need for a multi-level approach, whereby the application architecture is divided in multiple hierarchical levels, each especially designed to execute the functionality it is best adapted for.

Another way of seeing this, is to be aware that what happens at the ISR level is basically a loop. The ISR that gets activated first will be the one that gets processed first. Any subsequent interrupts can nest

themselves on top of the running ISR (if the processor or the system software supports it, if not, the interrupt will have to wait until the running ISR exits. However, in general, the processor will not be able to "accept" another interrupt of the same kind of an ISR that has not exited yet. In order to solve that, one must add specific external hardware to buffer these. This brings us to another issue. At the ISR level, one will often also buffer the data until enough data has been gathered to pass it on to the next higher level(s) in the system. The size of the buffers are mostly determined by the allowable latency between the reading of the input data and the generating of the output data.

The main conclusion for writing an ISR is simple : keep it as short as possible. This will disable interrupts for a minimum time and will allow higher interrupt rates, without any modifications to the hardware.

So what is now this "next higher system level"? We already know this : it must be interruptible.

### **Static scheduling**

If the application must not be interruptible, we are in the domain of synchronous dataflow applications. In this kind of application, all timings are known (up to the cycle) at compile time and given either hard work or a good tool, it is possible to calculate when the data arrives, when the functions must execute and when the results will be available. This domain of applications is very often very static and characterized by short input-to-output delays (e.g. active noise control). So no runtime scheduling is needed and the code size will be minimum. The price to pay is that one must find the static schedule and the fact that (asynchronous) interrupts cannot be tolerated. Note that a variation on static scheduling is timer based scheduling. If the data can be processed in larger buffers and the timings are all known at compile time, it is possible to have a central timer (often implemented as a lower priority ISR) to call the processing functions on the basis of a timer list. This allows asynchronous interrupts, provided one is sure that the worst case runtime conditions will never violate any deadline.

### **Superloop scheduling**

If one only uses the compiler/assembler, the interruptible level is the main() loop. Essentially, once the application is initialised, one starts a loop that tests for the availability of the data (signaled by the ISRs). If the test is positive, an associated processing function is called that upon completion returns to the main loop. In the main loop, one tests for the next data, calls the associated function, etc. Hence the name polling loop or also called "superloop". This works well if all the data arrives at the same frequency or a nice multiple of some base frequency and all the processing can be done fast enough to keep up with it. As soon as the frequencies are no longer nice multiples, the need for "buffer time" arises, the CPU effi-

ciency will drop and things can get fairly complicated. One can also increase the buffersize, but application latencies and scarcity of the memory will become the limiting factors. Another problem is that many polling cycles can get wasted on infrequently occurring events, like events that are associated with a state transition of the application (e.g. alarm signals, error conditions, changing environmental parameters, etc.).

### **Round-robin scheduling**

In the superloop solution, the main problem is that there is a sequentialisation that starts at the data-acquisition, followed by the buffering and ends when the processing loop returns to allow the start of processing on the next databuffer. As long as the subloop has not returned, the passing of the data to the next higher level is blocked. Hence the need to

disassociate the two actions, at the one hand passing the data and on the other hand processing it. This requires that one can "prepare" the processing to start as soon as the data arrives, without the need to actively poll on it. Polling is out of the question as there is only one CPU and this means we would be back at the superloop solution.

The solution is a runtime scheduler that will activate the processing functions dynamically. This requires that the processing functions can be descheduled when they reached the point where they are ready to start working on the data. Descheduling means that the status of the "waiting" processing function must be saved. The result is commonly called multi-tasking or process scheduling (sometimes also called multi-threading, but here we enter a domain of semantic discussions). The most simple and natural type of scheduling is round-robin (also called FIFO based scheduling). In itself not much different from a superloop in simple applications, but a lot easier to use when the data arrives at not so simple multiples of a base frequency or when the data arrives at irregular intervals. Therefore round-robin scheduling is ideal for all applications that have to deal with asynchronous interrupts. Are there no problems? Sure, once a task or process has been activated, it can monopolise the CPU and delay the execution of the already rescheduled tasks (called the ready queue). This can result in long worst case conditions. Therefore, round-robin scheduling is sometimes combined with time-slicing to limit the time a task can monopolise the CPU. Note however that besides the reduction of the worst case time that a task can be kept waiting in the queue, timeslicing in itself is often an unnecessary overhead for a real-time application (entailing the saving and restoring of the task's context). Timeslicing has more to do with "fairness" policy.

### **Pre-emptive scheduling**

As indicated, round-robin scheduling uses a FIFO order of execution and this has the drawback that for all processing tasks, the worst case time it can spend

---

**The main conclusion for writing an ISR is simple : keep it as short as possible. This will disable interrupts for a minimum time and will allow higher interrupt rates, without any modifications to the hardware.**

in the pipeline is dependent on what tasks were inserted before it in the queue. If a processing task is one that must occur very frequently or if there is a strict deadline associated with it, there is potential problem. The functionality needed to solve it, is called pre-emption. This means that the processing tasks are assigned a priority level and that higher priority tasks can pre-empt the lower priority tasks that are running. In this case, the runtime scheduler will be a bit more complicated than with the round-robin scheduler. First, as the descheduling points are not known beforehand, full context information must be saved and restored. Second, as the concept of priority is an integral part of the scheduling semantics, the runtime system must at all time take the priority into account, e.g. when the task is added to a waiting list to get access to a resource or when it generates inter-processor communication. This allows one to measure and define the worst case latency to start up a task. The highest priority task will have a minimum latency for which an upper limit can be defined.

How does one determine the priorities? A well known technique is called Rate Monotonic Analysis. It has been proved that given some boundary conditions, that for a given set of periodic tasks, one can always meet the deadline by assigning the priorities in order of the frequencies that the tasks must run. Real applications are not always that simple, e.g. with asynchronous interactions with the outside world and with synchronisation points between the tasks. In these case real-time monitoring tools and what-if type analysis tools are needed as well.

Pre-emptive scheduling is typically needed for those applications that are control bound. In the control world, reaction times are primordial, an extreme case being fault-tolerant systems. In DSP applications, often round-robin scheduling will be sufficient for the pure data processing, depending on the mix of frequencies and number of datastreams. More complex DSP applications with asynchronous elements (e.g. file I/O) still often need pre-emption because prioritization is needed to make sure that the higher priority data processing tasks will not be delayed by the asynchronous processing. Often exceptional state transitions (e.g. runtime failure detection) can then still pre-empt all tasks by being assigned the highest priority even if they never execute if all goes well.

## RELATED ISSUES

In real applications, some other issues need to be addressed to handle the real-time problem. Without covering them exhaustively, I outline the most important ones :

### **Buffer size versus latency**

There is a tendency to define the interrupt latency as a single figure. In practice, when measured, it will not be a single figure but a histogram. The interrupt latency on a processor depends on many factors : the level at which it is measured, the probability that inter-

rupts are disabled for a certain interval, the probability that several interrupts happen simultaneously, the number of waitstates on the bus (e.g. influenced by cycle stealing DMA engines), etc. In a stress test, the measured worst case figures will depend on what else the processor can be executing simultaneously. For hard real-time applications, the designer must take the worst case figures. As we take as definition the time it takes to read the data at a certain level, the worst case latency will also determine the minimal buffer length that one needs to use to process the data at a given data rate. For example, even if the typical interrupt latency is 1 microsecond, if the worst case figure is 4 microseconds, one can at most guarantee to process 250 000 interrupts/s. In practice, the figure will be lower because we neglect the time it

takes to pass the data and the time it takes to process them. If the worst case latency to a higher level is 40 microseconds, one can at most process 25000 interrupts/s. (We assume here that the hardware has no buffering mechanism). So we will need to buffer the data at a lower level (with a minimum buffer length of 10), to reach 250 000

interrupts/s. Of course, this again neglects the handling overhead. But if we go to data buffers of 1000, we only need to pass to the higher level every millisecond. The latter means that the overhead for using the higher level is just 40 nanoseconds per sample.

### **System level functions (e.g. drivers)**

In the above set-up, we neglected the impact of the system level functions, like link port drivers, external and often asynchronous I/O drivers, etc. They interfere with the normal local processing and their impact must be kept minimal. As an example we take the typical functionality of a C40 or 21060 link port. In the general case, a DMA engine will have been programmed to receive data on a port. When the data arrives, the DMA starts (often using cycle stealing on the bus), copies the data to a buffer or to its final destination (e.g. the workspace of a receiving task). If the received data has to be through routed, the receiving function will need to decode the header, use a look-up table to determine the outgoing linkport, start up the transmitting function (hence a DMA) and reprogram the DMA on the receiving linkport. We neglect here the more complicated situations whereby a protocol is started between tasks and/or the runtime system on different processors, but the essential functions remain the same. The problems are that often the data communication will trigger other actions like starting up a waiting task and/or updating of protected datastructures. At first sight, the fastest way is to use the ISR level, as the overhead is minimal. One of the problems is that interrupts are disabled inside an ISR (even more so if several ISR can update the same system level datastructures) jeopardising application level interrupt latencies. Another problem is that ISR can't wait (unless polling). A more natural choice is then to use runtime processing tasks, but these

---

**Not all real-time systems will meet all of their deadlines at all times. This kind of system doesn't fail in overload conditions but goes into graceful degradation.**

exhibit much higher interrupt latencies. Also they must then be scheduled at highest priority jeopardising the application level tasks. The solution is an intermediate level that uses multi-tasking, but simplified for maximum efficiency, e.g. round-robin scheduling, a reduced context and simpler services. In the next part we will illustrate such an architecture.

### **Overload conditions**

Strange as it may sound, not all real-time systems will meet all of their deadlines at all times. This kind of system doesn't fail in overload conditions but goes into graceful degradation. Examples include telephone switches (where speech packets are dropped if not processed on time) and autonomous robots (where environmental maps are outdated if not updated on time). Such a system can often not be economically designed to handle the worst case loads. Such a real-time system must have an efficient means for killing a task in midstream or dropping data based on a timestamp.

### **Time-outs**

Time-outs are often used to detect failure modes. Normal runtime systems provide 3 different instances of services. Non-blocking services always return immediately with a valid return value (success or fail), but can force a task in an endless polling loop if the service never becomes available. Blocking services avoid the polling by descheduling the calling task until the service becomes available. If this never happens, the task can remain blocked for ever. Therefore, a time-out mechanism allows to return anyway if the service is not provided within the specified time-out interval.

### **Resource protection**

Most real-time systems contain logical resources that must be protected from simultaneous access. Therefore there is a need for a mechanism that allows one to wait on the resource until it becomes free again. When pre-emptively scheduled, one must provide for priority ordering of the waiting tasks and for priority inversion. The latter can occur when a higher priority task is blocked by a lower priority task that owns the resource but is prohibited from running by a task of intermediate priority.

### **Memory allocation**

While most C runtime systems provide the malloc() function for this, in real-time systems, the system can fail after some time if multiple tasks share the same memory heap. This is a result of the memory fragmentation that can occur after a number of malloc / free cycles. The solution here is to use compile time defined memory "maps".

### **Memory protection**

Another issue is the protection of data in memory. In the simplest case, one can use a resource or a map. But often data is overwritten unintentionally. This is

even more true for multi-tasking systems where global variables are used but tasks can be pre-empted. For this reason, memory protection is at the heart of reliable real-time programming by applying a sound programming methodology. The runtime system can be of great help here. First of all, one should avoid global variables, but preferably use the task's local stack. Secondly, the services provided by the runtime system should be side-effect free and semantically prohibit memory from being overwritten. E.g. when exchanging data between tasks, the services should operate identically whether the tasks are on the same processor or not, while the data producer should only return when the data has been copied or set free by the data consumer. Passing pointers and using shared memory can increase the performance but must be used with great care, mainly as a second step optimization technique. Passing a pointer is fast but how does one pass a pointer across processor boundaries? How is one sure that the data is not prematurely overwritten? The essence is to make good use of the runtime services.

### **Runtime services**

Runtime services provide for a consistent interaction between the tasks. If the set of services is orthogonally organized, one can easily map the application requirements. Events are used for single event signaling, counting semaphores provide for events with buffering, queues for buffered, fixed size communication and mailboxes for flexible synchronous and asynchronous message passing. Synchronizing services have the advantage of safety as they avoid runaway situations (e.g. when more data is produced than can be consumed) and inherently allow to avoid memory overwriting. If however, more decoupling is needed, asynchronous services are helpful. As memory protection is vital, the runtime system itself will then safely buffer the data.

## **DISTRIBUTED SEMANTICS**

It is worth explaining the distributed semantics more in detail. This essentially means that when developing the application, the programmer only sees one virtual single processor, independently of the processor type used or of the number of processors and the way they are interconnected. Concretely, develop a program on a single 8086 and port it to a parallel DSP system with no changes to the source code. In a second step, add a couple of DSPs and still, your original source code remains intact. Of course, hardware dependent parts will have to be adapted. How can this be achieved? Porting between processors is done by using the same kernel on all processors. The major part, essentially the part that interfaces with the application tasks, is written in portable ANSI C. For the kernel developer, the thing left to do, is to port the processor specific low level task swapper and interface routines.

---

**When developing the application, the programmer only sees one virtual single processor, independently of the processor type used or of the number of processors and the way they are interconnected.**

As for the transparent parallel processing, things are a bit trickier. The first thing is to make sure that no side-effects can occur when the application developer moves around kernel objects (tasks, semaphores, queues, mailboxes, etc.) in his parallel processing target. This requires a very careful analysis of the semantics. As an example, we take a semaphore. Most kernels use binary semaphores. This means that they can only hold one event at a time (the semaphore has been signaled or has not yet been signaled). How then to avoid that two tasks will signal the semaphore simultaneously? Even on a single processor, this requires

some not trivial coding. one micro-controller kernel simply called a "crash" routine (an endless loop) upon simultaneous signalling! Messages are even more trickier. Most kernels pass a pointer. How to assure that the data is never overwritten prematurely?

What is the result of passing a pointer in a distributed memory system? If this kernel is controlling your car or an airplane would you trust it? You can just hope that the application developer was careful enough to avoid these circumstances. When designing the semantics of an RTOS, one should start from a distributed memory point of view and made sure it also works on a single processor. The result is that the application developer is relieved from these potential problems and can concentrate on his application. He can still pass pointers if he wants to optimize, but it will be visible in the source code.

The second thing is to assure that the implementation guarantees the same behavior, logically and real-time wise, whether the object is moved in the parallel processing network from one processor to another or kept on the same processor. The trick here is to use a message based protocol, using prioritized packet switching at the lower levels. The protocol is needed to provide the same logical behavior independently of the kernel object location in the parallel DSP network. Essentially the programmer must be assured that when the task returns from the kernel service, that the logical behavior is always the same. This is not always trivial to implement. In the case of a distributed "select with time-out" service, where the task is waiting on the first of a list of semaphores being signaled, including a time-out, this even entails generating anti-messages to subsequent semaphores that have been signaled, during the inherent communication delay. The prioritized packet switching is needed to avoid that a communication medium is monopolized for too long, while the prioritization assures that any higher priority activity is handled first, notwithstanding unavoidable communication delays.

The minimization of the communication delays, simultaneously with an adequate handling of I/O device interrupts, can be obtained by the so-called multi-level approach. Essentially it amounts to a careful handling of the registers as a resource, minimizing context switching. At the highest level, the developer can write in a high level language like C or C++. Everything he writes at this level (the micro-kernel

task level) is portable and scalable. As the context is high, the overhead is not acceptable for hardware related functions, e.g. like interprocessor communication. Note that on some processors like the 21060, tasks can even use a "super-context", allowing to call assembler optimized functions that use more registers than the C compiler can handle. A natural reflex is then to use interrupt service routines (ISR). There you only save the registers you need. But ISRs by default disable generally all other interrupts as soon as you enter them. In addition, incoming communication will result in system level

datastructures being modified and will result in synchronisation with other events in the system. This means that in system with many interrupt sources, like the C40 and the 21060, you are in trouble. E.g. the first principle is to separate the processing of the interrupts in two

levels, one with hardware interrupts disabled, and one where interrupts are globally re-enabled, hence minimizing interrupt disabling times. As a result, the Virtuoso RTOS allows up to 1 Million interrupts per second (even on DSPs like the C40 that has only one hardware interrupt priority level).

A major problem however remains. How to synchronize in such a system? As ISRs have no context, they cannot deschedule and inactively wait. Very annoying when one develops asynchronous communication drivers. Programming this at the task level is more flexible but results in very high overheads (context switch latencies) and excessive use of memory. In the Virtuoso RTOS this problem was solved by creating an intermediate level, the so-called nanokernel level. This level is truly multitasking but uses a well defined subset of registers and a minimal semantic behavior to minimize the overhead. The result is impressive. On most processors this offers channel based communication between nanokernel processors (or call the light-weight tasks) in less than a microsecond while the codesize is in the order of 200 instructions. As such, this is ideal for system level functions. With a minimal overhead, the nanokernel handles the interprocessor communication while offering true multitasking albeit in a round-robin fashion. The nanokernel is in fact the key to the performance of the distributed Virtuoso kernels. It is also ideal for the upcoming multi-channel, multi-function DSP applications. The modularity being a side-effect of the multitasking model, the low overhead makes it practical.

## CASE STUDY : ARCHITECTURE OF VIRTUOSO CLASSICO /VSP

The reality of the real-time applications mean that a single type of kernel is not adequate for all real-time applications, especially if one is targeting DSP. In the Virtuoso approach this is achieved by supplying a multiple set of tools, each addressing a complementary type of scheduling. We take this as a case study and as it is the only RTOS family with this multi-tool approach. Why use a general purpose tool if your

**The Virtuoso RTOS allows up to 1 Million interrupts per second even on DSPs like the C40 that has only one hardware interrupt priority level.**

Lynx Ad

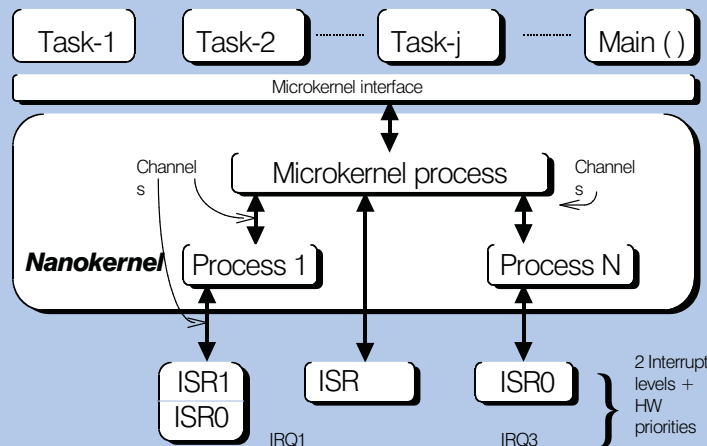


Figure 1. Virtuoso (Classico)'s generic architecture

needs are less? The essence is to look at the real-time scheduling requirements. Each tool is functionally a subset of Virtuoso Classico /VSP. The benefit of this approach is the portability and scalability of the code, allowing to use the same base technology as well as on larger parallel DSP targets and ASIC DSP cores. Below a schematic diagram illustrating the four levels in Virtuoso Classico /VSP (as seen on a single node). We leave the exercise to the reader to compare it with other RTOS solutions on the market.

The four major levels are as follows :

- **ISRO :**  
Interrupt Service Routines with interrupts globally disabled.
- **ISR1 :**  
Interrupt Services Routines, with a status of globally enabled interrupts.

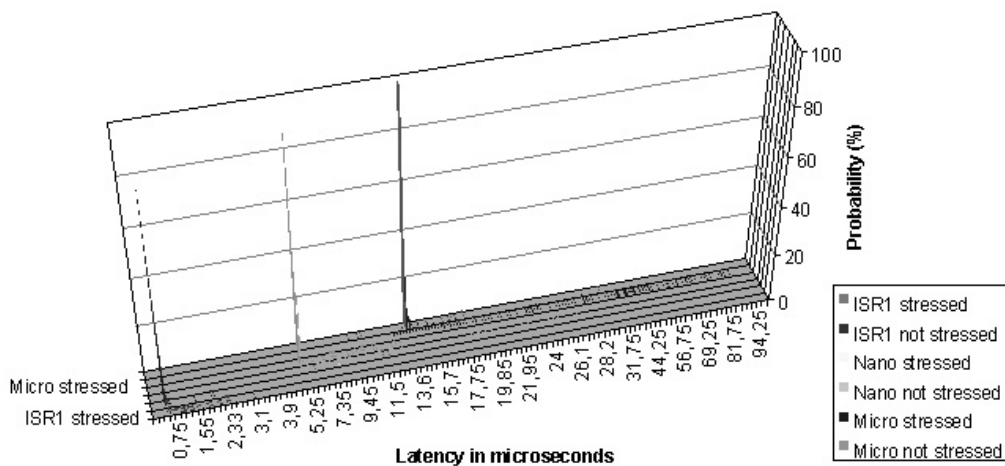
- **Nanokernel :**  
Round robin scheduled processes and communicating through so-called "channels"
- **Microkernel :**  
Pre-emptive, priority based scheduled tasks (full C context).

### Scheduling overview in the Virtuoso product range

The following tables summarizes how the different levels and types of scheduling are found back in the different products. In the last column, we provide some typical applications for which the products are being used.

Product	Scheduling	Levels	Typical applications
Virtuoso Micro (pre-emptive RTOS)	Pre-emptive	3	Low to medium speed control
Virtuoso Classico (distributed pre-emptive RTOS)	Pre-emptive + round-robin	4	High-end DSP and control
Virtuoso Nano (small multi-tasking RTOS)	Round-robin	3	DSP asynchronous dataflow
Virtuoso Synchro (R&D version) (static schedule generator)	Static	2	DSP synchronous dataflow
Virtuoso Modulo (DSP Libraries) <ul style="list-style-type: none"> <li>• with compiler/asm only</li> <li>• with Virtuoso RTOS</li> </ul>	Superloop See above	2 see above	Single stream applications See above

Table 1.



Latency in microseconds	Best	Typical (peak) (at probability)	Worst	99% value
ISR1 stress load	0.38	0.40 (55.6%)	2.88	1.90
ISR1 low load	0.40	0.40 (86.8%)	1.93	0.45
Nanokernel stress load	2.70	4.40 (31.3%)	28.95	18.80
Nanokernel low load	4.4	4.40 (92.7%)	13.05	5.51
Microkernel stress load	8.25	13.0 (24.0%)	97.50	52.75
Microkernel low load	8.25	13.0 (98.0%)	23.00	16.75

Table 2. Interrupt Latency Test on 21062 - 40 MHz

**Interrupt latency for Virtuoso Classico /VSP on a 21062 at 40 MHz**

As was explained above, an important parameter that determines the real-time capabilities of an RTOS is the interrupt latency. At Eonic Systems we tested a dual 21062 system running at 40 MHz. The time was measured from the hardware interrupt till the instruction where the data could be read at each level. Note that even at the ISR level, this is not the first instruction as the ISR has to save first some registers on the stack of the interrupted activity. For the nanokernel level we took the first line in a nanokernel process and for the microkernel level, we took the first line in a high priority microkernel task. Two load conditions were applied. In low load conditions (about 25%), there is some slow interaction with the host computer and a maximum of three asynchronous interrupts can happen. Under stress load conditions, the processor on which the latency is measured is saturated with a set of 8 continuously interacting tasks, DMA operations in the background (between the two processors) and a background task that consumes all remaining CPU power (load conditions of 100%). In both cases, a 1 millisecond timer interrupt is used. In the following table, we have summarized the main extreme values. For each test a total of about 10000 interrupts were used. Note that the results are typical and specific for the test program and the test run. Any other test will provide similar but different results, mostly depend-

ing on how interactive the program is and on the probability to have multiple simultaneous interrupts.

**CONCLUSION**

In this paper we have explained some fundamental properties of real-time, in particular DSP applications. We have shown that only the simple ones do not benefit much from the use of an RTOS. In general the RTOS will provide for a much more reliable programming methodology, a higher real-time predictability and shorter development time, especially when targeting parallel DSP applications. ■

*Eric Verhulst is the Managing Director of Eonic Systems Inc., a company specialising in applications for parallel computers, which he founded in 1986. This is backed by a broad range of activities ranging from conceptual research, applications research as well as acquiring the necessary know-how and managerial skills. Current activities concentrate on distributing transputer systems and inhouse development of a real-time fault-tolerant operating system. Mr. Verhulst holds a Civil Engineering Degree in Tele-communications and Ballistics at The Royal Military School Brussels, and a Licence in Computer Science at the Free University Brussels.*