

Support for High Availability Applications in New-Generation RTOS's

High availability applications are those that must be robust with respect to expected and unexpected failure events. They are based on aggressive error detection and error handling, as well as redundant hardware and software. Operating systems for these kinds of applications must provide unique infrastructure to support these specialized system designs. This article will survey some prominent special features of real-time operating systems to support high availability applications.

INTERTASK COMMUNICATION

A synchronous message passing is the preferred technique for communication between tasks on different processors or on different hardware boards. This approach avoids the synchronization issues inherent in mechanisms such as a semaphore, which can create long delays in a multi-processor environment as tasks wait for a distant semaphore to be locked or unlocked. It also avoids traditional semaphore pitfalls such as unbounded priority inversions, ease of deadlocking, and difficulty in debugging. Mutexes, which are often viewed as a solution to problems inherent in semaphores, work by changing priorities of tasks - and so they too are inappropriate in a multi-processor environment.

Asynchronous message passing is a loosely-coupled intertask communication mechanism, where a task sending a message does not wait for any information from the receiver task -- and thus can not fail even if the receiver task has failed or becomes inaccessible.

DYNAMIC MEMORY ALLOCATION

The "traditional" mechanism for dynamic memory allocation is the heap, famous throughout the world of programming under the names "malloc" and "free". Heaps suffer from fragmentation, a situation in which their memory is gradually broken into smaller and smaller

free areas interspersed between areas still being used. Some RTOSs will force tasks to stop and wait when they request some memory larger than the largest available fragment. Other RTOSs will do a "garbage collection" or "defragmentation" procedure to rearrange

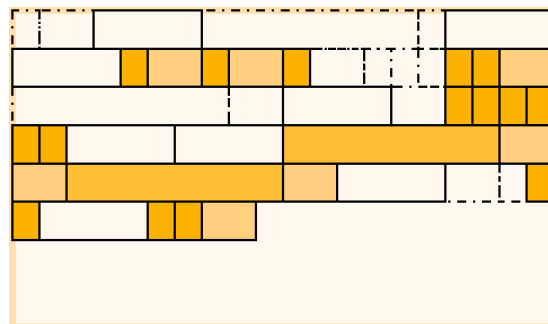


Figure 1. Memory Pool for Buffers Allocation.

the heap memory so as to collect the fragments and coalesce them together into large allocatable chunks. At first glance, this sounds like a good solution to memory fragmentation. But this solution actually entails stopping tasks at what appear to be random times, for random-appearing lengths of time while the defragmentation is being done. For applications in which high-availability may be required in the range of

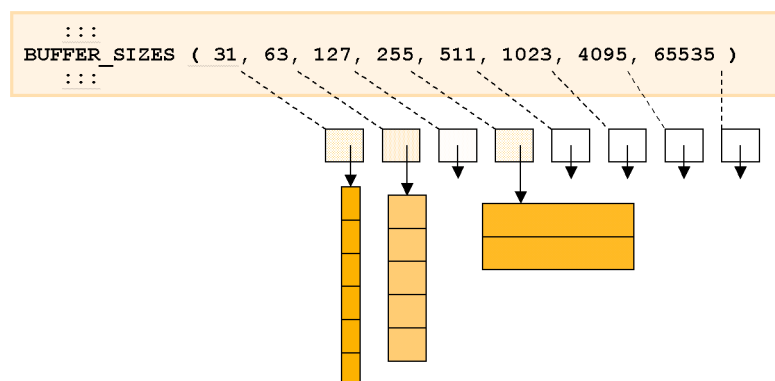


Figure 2. Pool Buffer Re-Use via Free Lists.

HIGH AVAILABILITY

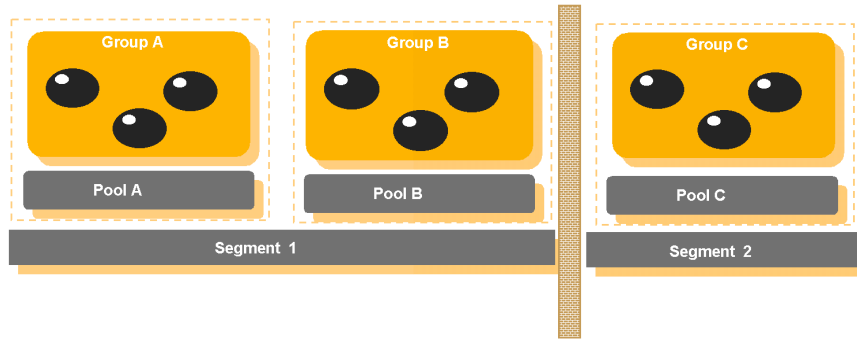


Figure 3. Exploiting Hardware Memory Protection.

99.999%, less than 5 seconds of stopping per year is tolerable for all reasons. Hence, defragmentation itself may cause availability problems.

So what is needed in RTOSs that support high-availability applications, is a flexible memory allocation scheme that avoids memory fragmentation rather than repairing it. An example of such a scheme is a pool from which memory buffers can be allocated in up to 8 predefined sizes, as shown in Figure 1.

In this scheme, buffers once allocated do not merge or split. Instead, once they are freed they are recycled at their original sizes, using a set of 8 "free lists", one per buffer size. This is shown in Figure 2.

If no buffers of a given size are available in the free list, the RTOS can create a new buffer in the unused portion of the pool.

This pool scheme provides a variety of memory buffer sizes while completely avoiding memory fragmentation, and so it is appropriate for high-availability applications.

EXPLOITING HARDWARE MEMORY PROTECTION

Many high-end microprocessors and communication processors such as PowerPC chips, offer built-in hardware memory management (MMU). This is particularly important for applications programmed in pointer-oriented languages such as C or C++ where "stray pointers" are a common software bug. Memory management facilities can intercept stray pointers before they cause damage such as writing data into the wrong addresses. But ironically, most traditional RTOSs

ignore MMUs and treat processor memory as a flat, globally accessible expanse.

An RTOS for high-availability applications must take advantage of MMU capabilities of processors, when present. An example of this is shown in Figure 3 below.

In this example we see three groups of tasks, each group having its own memory pool. Group C is separated from Groups A and B by a memory protection barrier enforced by processor hardware. If a task in Group C tries to write anywhere into Segment 1 (to the left of the memory protection barrier), hardware will report this to the RTOS which will handle it as an error. Similarly if a task in Groups A or B tries to write into Segment 2, hardware will work with the RTOS to handle the error.

In addition, if a task in Group A would like to communicate with a task in Group B or C, the RTOS must be aware of the memory protection barrier between them and perhaps handle this message communication in a special way. A good RTOS will make this special handling transparent to software developers.

TASK PRESENCE MONITORING

In an application environment where we must be extra-sensitive to software and hardware failures, we would like an RTOS to be able to notify an application if a critical task has "died" or disappeared. This service is needed for both "local" tasks and tasks running on other hardware boards. In addition, a service is needed to search for tasks containing critical functions. This too is needed for both "local" tasks and tasks running on other boards. An example of such services is shown in Figure 4.

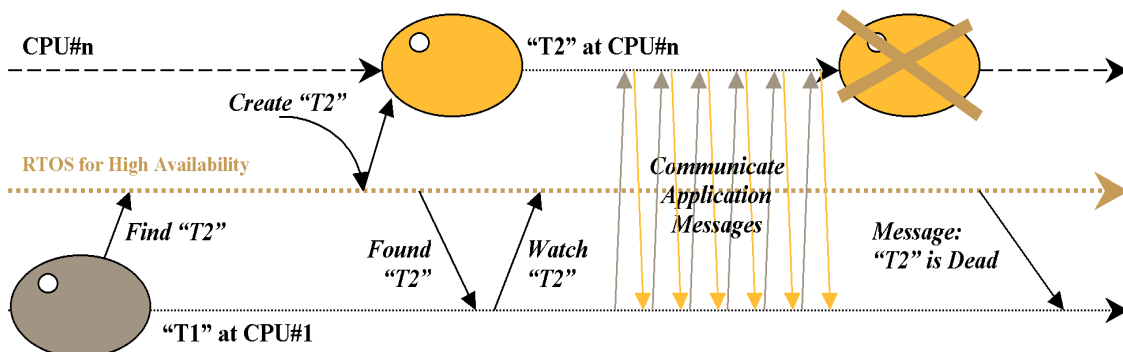


Figure 4. Task presence monitoring time-line.

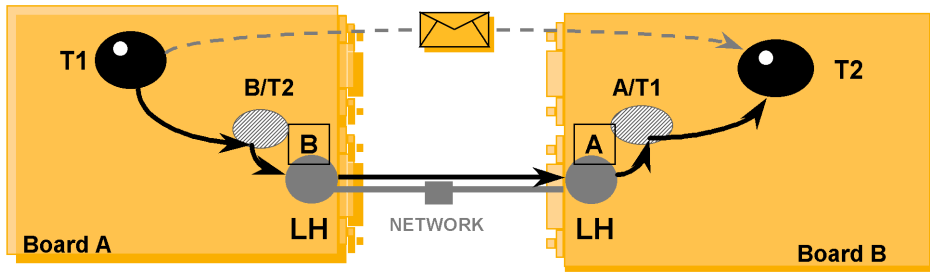


Figure 5. Conceptual path vs. actual path through link handler.

This is a timeline graph, with time progressing from left to right. On processor CPU#n, shown in the top timeline, a task "T2" may appear at any time, then disappear at any time.

On processor CPU#1, shown in the bottom timeline, is task "T1" that would like to communicate with "T2". The middle timeline represents the RTOS for high availability.

The sequence of activities shown in the timeline graph is a typical "life cycle" of communication between "T1" and "T2": First, "T1" asks the RTOS to find "T2". The RTOS may need to search a number of CPUs until "T2" is found. When "T2" is found, the RTOS notifies "T1" of this. At that time, "T1" could begin communicating with "T2" if it wished to do so; but instead, it first asks the RTOS to begin monitoring the continued presence of "T2". This is shown as "Watch T2" in the figure. Once the RTOS has agreed to monitor "T2", task "T1" can initiate communication with "T2" with confidence. If, some time later, "T2" ceases to exist or ceases to be accessible, the RTOS will notify "T1" that "T2" is dead.

At that point "T1" must decide what to do next. Often, "T2" will be on a board that will soon be replaced. Or CPU#n and CPU#1 were connected by a network that failed temporarily. So after "T1" is notified that "T2" is dead, it will typically just go back to the left side of the timeline and ask the RTOS again to find "T2". In this way, fault-tolerant systems can be designed using software tasks that repeatedly cycle through this timeline sequence.

TASK PRESENCE MONITORING PEER-TO-PEER IMPLEMENTATION

This kind of task presence monitoring can be done in a multi-processor and multi-board environment using a special RTOS component often called a "Link

Handler". Different Link Handlers can support different interconnection media between processors, such as network connections, serial lines, buses or backplanes. There must be a Link Handler as part of the RTOS running on every processor in the multi-processor environment. A well-implemented Link Handler will set up a peer-to-peer working relationship with Link Handlers on other processors, and will support task identification ("find"), task presence monitoring ("watch"), and routine message communication between tasks on different processors. It can detect down communications and down processors as part of its task presence monitoring duties.

Figure 5 shows that application software developers program their message communication between tasks on different boards in the same way as if the tasks were on the same board and running on the same CPU. The actual path of the message is quite different, going through proxy "phantom" tasks, link handlers ("LH") and a physical interconnection medium ("Network").

HANDLING PROBLEMS OF SCALE

Many systems with high-availability requirements are quite large, containing perhaps tens or hundreds of processors and boards. Duplicate boards and software may be in place for redundancy purposes, while additional boards and software may be inserted into the system at any time and others yanked out. The tracking of these component availability issues is in itself a complex activity. A "Name Server" component of a high-availability RTOS is a distributed database designed to handle these kinds of issues. It must itself be fault-tolerant, as application tasks rely upon it to provide information about available application services, redundant providers of these services, and system

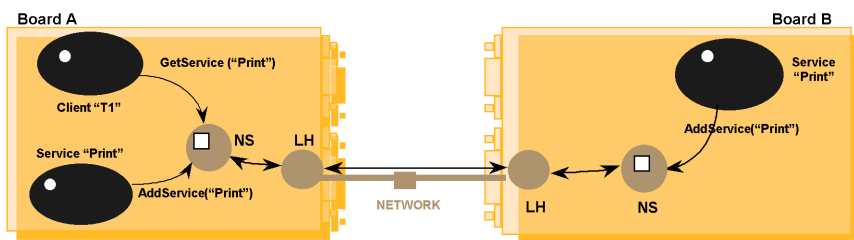


Figure 6. Name server example.

HIGH AVAILABILITY

topology for accessing these services.

In Figure 6, a task on Board A and a task on Board B are both printer servers. They each register as printing service providers with their local Name Servers ("NS"). The two Name Servers on the two boards exchange information via Link Handlers ("LH"), so that both Name Servers know about both printing service providers.

Whenever client task "T1" needs to find a new printing service provider, it can simply ask its local Name Server. Or it can "subscribe" to printing name service, and thus receive continually updated information about current printing service providers.

CONCLUSION

High availability applications are based on aggressive error detection and error handling, as well as redundant hardware and software. A new generation of real-time operating systems provide unique infrastructure to support such specialized application designs. Prominent special features of real-time operating systems to support high availability applications include: high-performance message-based intertask and interprocessor communication, non-fragmenting dynamic memory allocation, extensive exploitation of hardware memory management units, task presence monitoring, peer-to-peer Link Handlers and distributed Name Servers.

These new features will make it as easy to build complex high-availability embedded systems, as it was to build single-processor multi-tasking devices in the 1990's ■

David Kalinsky is director of customer education at Enea OSE Systems. He is a popular lecturer and seminar leader on technologies for embedded software in North America, Europe and Israel. In recent years, David has built high-tech training programs for a number of Silicon Valley companies, on aspects of software engineering for the development of real-time and embedded systems. Before that, he was involved in the design of many embedded medical and aerospace systems. David holds a Ph.D. in nuclear physics from Yale and can be reached by e-mail at david@enea.com.