ADVANTAGES OF ENEA OSE®:
The Architectural Advantages of Enea OSE in Telecom Applications

Brian Gildon
Product Marketing Manager

The days are long gone when a real-time operating system (RTOS) was simply a small kernel providing basic services such as task scheduling and reliable inter-task communications. Today's real-time operating systems are expected to perform a wide variety of functions ranging from managing real-time communications to providing a reliable foundation for higher level applications.

In essence, today's realtime operating systems provide the foundation for complete software platforms that are increasingly purpose-built for specific applications.

What many embedded developers may not realize is that RTOS software architecture plays a pivotal role in meeting the specific needs of a particular application. Factors such as whether the RTOS is monolithic or based on a microkernel, whether it uses sockets, and whether it enforces a specific programming model can make a significant difference in the types of applications that best suit the RTOS. Enea OSE® has been designed from the ground up for the fault-tolerant distributed systems commonly found in telecommunications equipment, from mobile phones to radio base stations. This paper discusses the OSE architecture and design philosophy, and how that benefits telecommunicationsoriented applications.

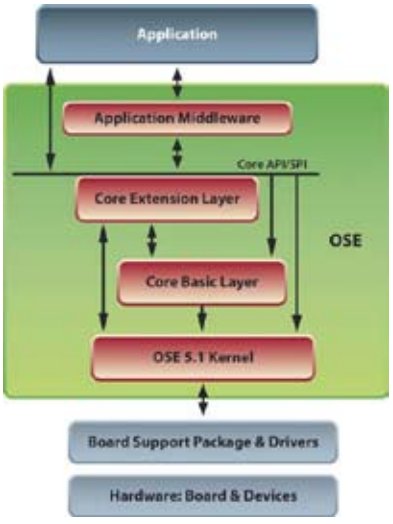

Figure 1. Layered Embedded Design Approach.

The Modular, Layered Concept

Most embedded systems are built on a modular, layered architecture with welldefined interfaces (Figure 1).

At the highest level of the system is the "application" layer, which contains the userwritten application software. At the lowest level is the embedded hardware, which can range from a single processor to a multicore/multiprocessor design spanning multiple blades. Just above the hardware is the firmware, which consists of the board support package, drivers, and related "glue" code.

The RTOS sits between the firmware and application. OSE specifically utilizes a layered architecture, including the "kernel", the "Core Basic Services layer" and the "Core Extensions layer," (Figure 2).

The OSE kernel provides basic services such as preemptive priority-based scheduling and direct, asynchronous message passing for inter-task communication and synchronization. It also provides a memory management package, which utilizes the processor's MMU hardware to provide memory protection, and manages special physical memories such as flash.

The Core Basic Services layer offers optional, configurable packages for services such as file system management, device driver management, heap management, and run mode / freeze mode debugging. In addition, this layer provides extensive C/C++ runtime support with a fully reentrant function library. Moving one level higher, the Core Extensions layer offers optional services such as communication protocol stacks (IPv4/v6, SNMP, DHCP, etc) and interprocess communications (IPC). Enea's LINX IPC software, for example, allows tasks running on different processors (or cores) to utilize the same message-

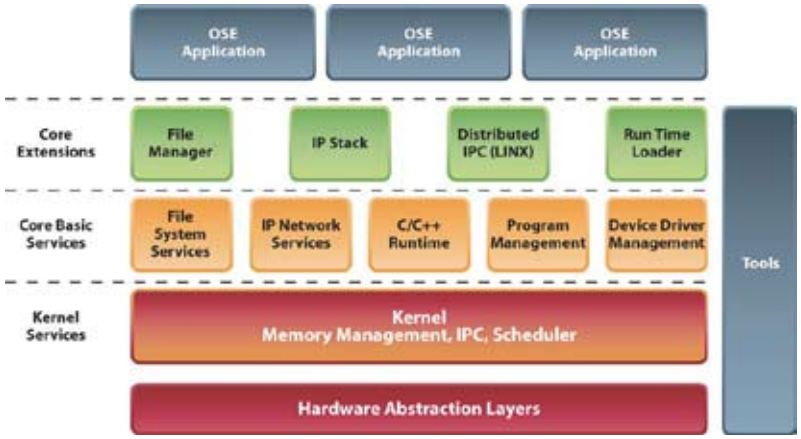

Figure 2. OSE's Modular, Layered Architecture.

based communications model that OSE uses for intertask communications on a single processor.

The core extensions layer also provides a dynamic run time loader for upgrading and hot swapping application software. The run time loader is used to load ELF files, typically through OSE load modules. Load modules are relocatable program units that can be loaded into a running system and dynamically bound to that system. A loadable module can be uploaded, rebuilt, and quickly downloaded while the remainder of the system continues to run. This capability is especially valuable for applications that need to be field upgradeable.

The Core Extensions layer is separated from user applications by the "Core API", a programming interface that allows OSE users to take advantage of its services without having to master internal RTOS complexities. However, users who possess a more detailed knowledge of the RTOS infrastructure may also utilize the "Core SPI", or System Programming Interface, to tackle tasks such as developing device drivers or industry-specific application platforms.

The Enea OSE Message Passing Philosophy

OSE is optimized for distributed, fault-tolerant systems, and built on an event-driven, communicating state machine model. OSE's natural programming model is based on passing direct, asynchronous messages between tasks (or "processes" in OSE terminology). This model tends to promote, though not strictly enforce, consistent behavior when coding applications. This makes training new programmers, designing new programs, and maintaining existing programs far more straightforward than with traditional RTOSes, which have no explicit programming model whatsoever.

Direct, asynchronous message passing is a simple and intuitive, loosely-coupled approach that provides transparency in data transfers from process to process. Processes that send messages do not have to wait for information from the process that receives the message. Thus, the sending process cannot fail, even if the receiving process fails or becomes inaccessible. This loosely coupled approach naturally lends itself to distributed, fault-tolerant, multiprocessor (and multicore) systems. OSE's direct, asynchronous message passing architecture offers many unique advantages for distributed systems:

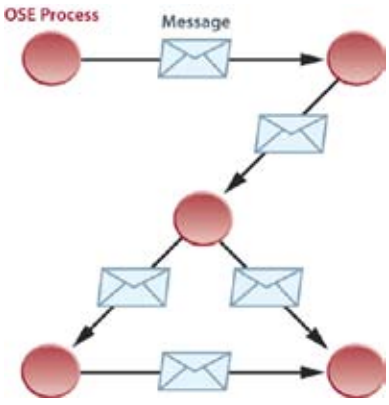

Figure 3. OSE Message Passing Design.

- In-herently modular, distributed architecture
- Simple, intuitive model that is easy to learn
- Consistent application design simplifies long term maintenance
- No shared memory among applications
- Task (process) ownership is never shared
- Messages may be traced and monitored
- Messages may be used for synchronization

These advantages are more easily understood when the OSE kernel services are examined in more detail.

The Enea OSE Kernel Design

Today's modern RTOSes are complex pieces of software that provide a wide array of services, including network communications, file system management, and dynamic application loading. They are typically architected in a modular, scalable fashion, which allows services to be added or removed as necessary. The kernel is the most significant part of the RTOS, as it is responsible for managing hardware and software resources.

The most important services provided by the kernel are:

- Process management
  - Process scheduling
  - Interprocess communications
  - Interprocess synchronization
- Memory management
  - Dynamic memory allocation
  - Memory protection
  - Demand paging
- Error handling

Process Management – Process Scheduling

Process management involves the coordination, prioritization, execution, and synchronization of processes that comprise and support applications

ENEA

running on the hardware. OSE manages application process execution through priority-based preemptive scheduling. The governing principle is that the highest priority process ready to run should always be the process that is running.

OSE manages processes in a different manner than many other RTOSes. For example, the OSE process scheduler manages hardware interrupts via an interrupt process. In this way, all hardware interrupts are managed in the same fundamental way as other software processes, thereby maintaining programming model consistency and simplifying system level debugging and troubleshooting. By contrast, many RTOSes cannot respond to hardware interrupts using their internal task/process schedulers. Instead, they require programmers to use an external interrupt service routine, which can sometimes complicate system-level debugging.

OSE manages processes that must run on a periodic, repeating basis using the same philosophy. The process scheduler reserves a separate range of high process priorities for these period processes, also known as timer interrupt processes. Each timer-interrupt process has its own OSE control block (Figure 4), which makes it easy to support system and process level debugging.

Most commercial RTOSes require an external timer to indirectly force application software to run on a periodic repeating basis. The timer-interrupt

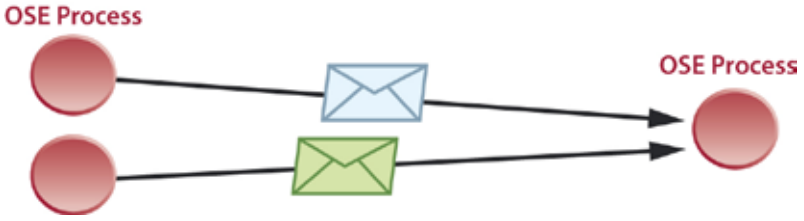

Figure 5. Enea OSE's Direct, Asynchronous Message Passing Model.

priority is set above the priority of other tasks/processes running. The net result is that most RTOSes require the application software to be organized and executed in strikingly different ways, depending on whether the processes are periodic, triggered by the RTOS scheduler, or triggered by hardware interrupts. Under, all application software is managed in a consistent fashion using the same programming model.

Process Management – Interprocess Communications

Preemptive multitasking requires interprocess communication and synchronization in order to prevent processes from conveying corrupted information or otherwise interfering with each another. To minimize the impact on hardware resources OSE uses direct message passing for interprocess communications and synchronization (Figure 5). Its processes send messages "directly" to other processes, but without actually copying the message. Instead, OSE transfers a pointer (to the message buffer) from the sender process to the recipient process, while obliterating the pointer value stored by the sender process (for mutual exclusion).

By contrast, many RTOSes employ an indirect message model that uses intermediate mechanisms (such as message queues) to buffer messages between processes (Figure 6). The problem with this approach is that it often requires the application to create the message queue through which messages are sent and received. In other cases, a message may be copied many times – once when it is sent from the sending process to the message queue, and a second time when the receiving process fetches the message.

For short messages, the impact on system resources is minimal. However, for longer messages, which are common in telecom applications, the performance burden is often unacceptable.

In extreme cases, RTOSes may be constructed with "queues of queues" (Figure 7), further exacerbating the performance burden. In the example in Figure 7, there are three queues, one for processes waiting to send messages, one for processes waiting to receive messages, and one central message queue. Managing this hierarchy adds complexity and can significantly degrade performance in communications-intensive applications.

Process Management – Interprocess Synchronization

RTOSes use a variety of interprocess communication and synchronization mechanisms, including message

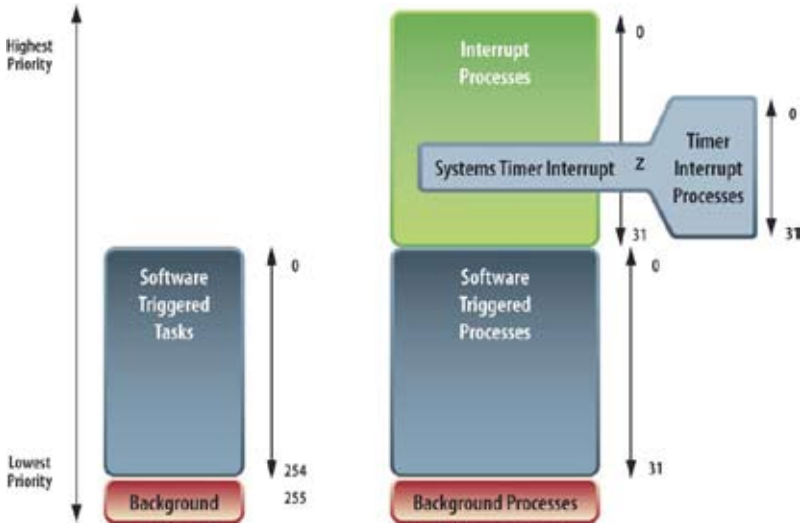

Figure 4. Typical RTOS Priority Schema (left) vs. Enea OSE Priority Process (right).

ENEA

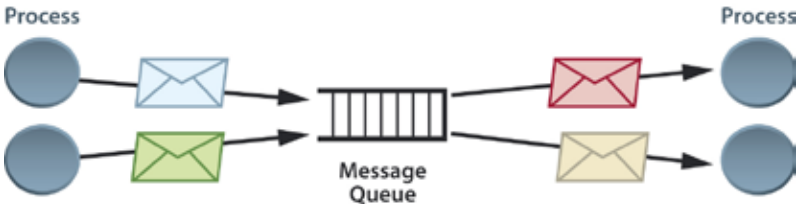

Figure 6. Typical RTOS Indirect Message Queue Architecture.

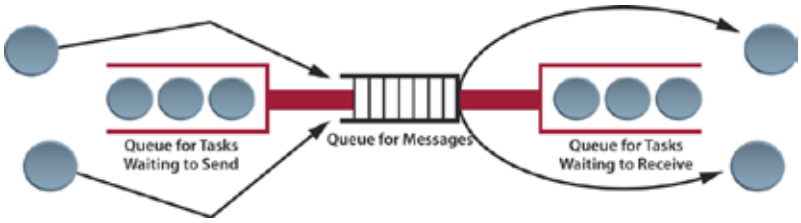

Figure 7. "Queues of Queues" Required in Some RTOSes.

queues, pipes, semaphores, mailboxes, event groups and asynchronous signals. Semaphores are often the mechanism of choice for basic synchronization. However, semaphores are subject to unbounded priority inversions and deadlocks, and can be difficult to debug. Mutexes solve the problem of unbounded priority inversions, but are still subject to deadlocks and debugging difficulty. Moreover, because mutexes work by changing thread priorities, they are not optimal for highly distributed, heterogeneous multicore/multiprocessor environments such as those common to telecommunications applications. OSE supports semaphores and mutexes (for applications with ultra-tight time constraints), but the preferred method for interprocess synchronization is direct, asynchronous message passing. To illustrate this concept, take the simple case of two processes attempting to share a printer

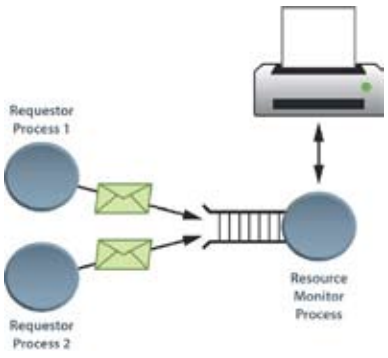

Figure 8.

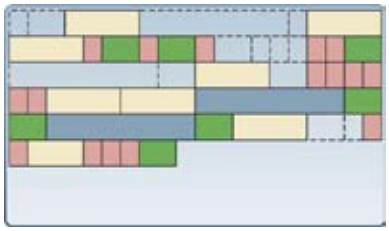

Figure 9. OSE Memory Pool for Buffers Allocation.

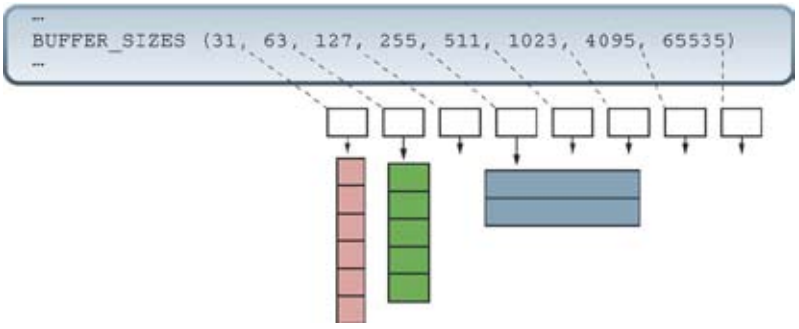

Figure 10. Enea OSE Memory Pool Buffer Reuse via Free Lists.

resource, as shown in Figure 8.

OSE uses a Resource Monitor Process to manage priorities, completely hiding all information about the shared resource. The requestor processes asynchronously send "print" messages directly to the Resource Monitor Process, avoiding intermediate mechanisms such as message queues. The Resource Monitor processes any message it receives as soon as it is ready. If it is already processing another message, OSE automatically builds a dedicated message queue as a linked list of message buffers, without any involvement from the programmer. In this manner, all messages are handled one at a time, even if they arrive in asynchronous bursts. Unlike semaphores and mutexes, which are difficult (or impossible) to distribute, this mutual exclusion approach works in single processor, multi-processor, and multicore environments.

Memory Management – Dynamic Memory Allocation

Like many traditional RTOSes, OSE can borrow buffers of RAM for temporary or permanent use. OSE's primary memory allocation mechanism is called a "memory pool." Most RTOSes manage this memory like a traditional heap. The problem with this approach is that heaps tend to fragment when the required buffer sizes are larger than the underlying memory page size. In contrast, OSE's memory pools are non-fragmenting. Available buffer sizes are limited to no more than eight standard buffer sizes per pool, where the buffer sizes are user-defined. Figure 9 shows an OSE memory pool from which memory buffers can be allocated in one of eight different buffer sizes, as required.

OSE uses a learning algorithm to determine how many buffers of each buffer size are needed by the application software. Initially, all of the pool's memory is available for allocation. When memory buffers are requested, they are taken from the pool's memory, starting at one end of the pool. Each

buffer taken from the pool is permanently assigned its initially allocated size (Figure 9). Once allocated, buffers do not merge or split. Instead, when they are freed, they are recycled with their original size using a set of "free buffer lists", one per buffer size (Figure 10).

When a buffer is freed, it is inserted into the free buffer list corresponding to its buffer size. Whenever an OSE "alloc" request is made, it first checks for the availability of a buffer in the free buffer list of the appropriate size. If no buffers of the required size are available in the free list, OSE creates a new buffer in the unused portion of the pool. In this way, OSE "learns" the worst-case buffer needs of an application for each buffer size and keeps sufficient buffers available to meet those needs in its free buffer lists.

Memory Management – Memory Protection

The OSE kernel allows its processes to be collected into "blocks", where each block of processes can be assigned its own separate (non-fragmenting) RAM memory pool (Figure 11). This compartmentalization prevents problems in one memory pool (such as a memory leak) occurs from affecting other blocks.

If the target CPU has a memory management unit ("MMU"), OSE can take advantage of the MMU to establish hardware-enforced, RTOS-aware memory protection between OSE blocks (or groups of blocks) within a single processor. This allows separate blocks (or groups of blocks) to have their own separate memory address spaces. OSE can also intercept stray pointers before they cause damage, such as writing data to far-flung addresses.

In Figure 11, there are three blocks of OSE processes. Each block has its own memory pool. Block C is separated from Blocks A and B by a memory protection barrier enforced by processor hardware, depicted as a vertical "brick wall".

If a process in Block C tries to write anywhere into Domain 1, hardware will report this to OSE, which will handle the error. Similarly, if a process in Blocks A or B tries to write into Domain 2, hardware will cooperate with OSE to handle the error.

If a process in Block A would like to communicate with a process in Block B or C, OSE (which is aware of the memory protection barrier between the blocks) will handle the message-based communication in a manner that is transparent to the programmer.

Within each separate memory address space, OSE's memory manager can make sections of memory non-cacheable or read-only accessible, and assign other MMUsupported attributes. But its main job is to give separate blocks (or groups of blocks) of processes their own separate, protected memory address space. During the development and test phase, where bugs are more prevalent, and may throw stray pointers, programmers may elect to have the memory manager provide full process separation and address space protection. However, this protection can have significant overhead, so developers may elect to disable memory protection between some blocks in the final product. For example, in Figure 11, there is no memory protection between Blocks A and B.

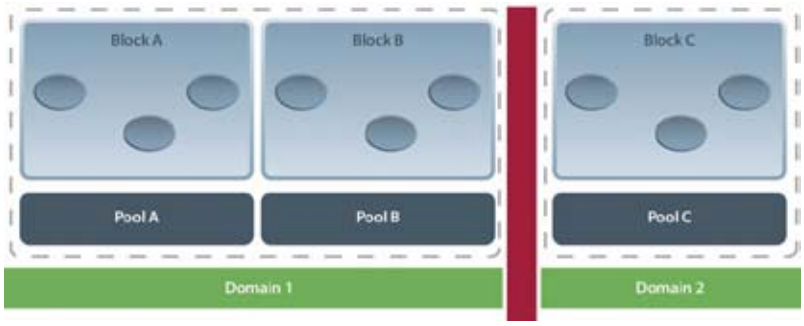

Figure 11. Enea OSE Blocks and Memory Protection.

Memory Management – Demand Paging

OSE gives programmers the option of tightly controlling RAM usage in applications where RAM is in short supply. In feature-rich mobile phones, for example, RAM is one of the most significant contributors to overall cost.

OSE's demand paging reduces RAM requirements by allowing programmers to store their programs and data in NAND flash, and then copy only the needed pages into RAM for execution. This allows designers to replace a large portion of their RAM with much cheaper NAND flash.

Error Handling

Most traditional RTOSes return a e of "0" when a kernel call is successful. When the call fails, the return is a non-zero error code that is referenced in a document or a header file. Often, the software that manages the error codes must be put into the same task or process that is called by the RTOS service. This can make the combined code difficult to read and maintain, as depicted in Figure 12.

OSE uses an error handling schema similar to that employed in the C++ language. Error information is not delivered to the calling OSE process. Instead, when OSE detects an error while answering a service request, it simply stops execution of the requesting process and switches to a separate piece of code associated with that process, known as a "Process Error Handler".

A separate handler is written for each OSE application software process, and contains the code needed to identify and remedy the error. If the handler reports to OSE that it has fixed the error, it will allow the stopped process to continue running.

OSE error handling also incorporates an error escalation sequence as part of the core kernel architecture, as seen in Figure 13.

If the Process Error Handler reports that it has not been successful in deal-

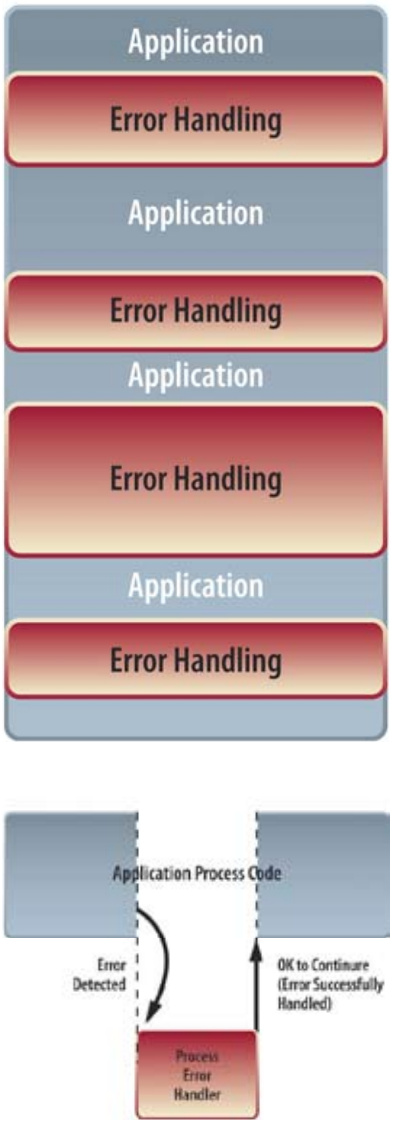

Figure 12. Typical RTOS Error Handling Interspersed Code (first picture above) vs. OSE Error Detection Schema (second picture above).

ing with the error, OSE will escalate the error to a higher-level error handler, and if it too is not able to handle the error, OSE will escalate the error to a System-Level Error Handler. There is one System-Level Error Handler for each processor running OSE, and it is the central error handler for "last ditch" recovery attempts such as board reset or alerting another node to take over as system master.

From a software architectural perspective, the OSE approach to error handling creates a clean separation between the normal activity of processes and the error handling activity processes may need from time to time. Normal, desirable process activities are coded in the main body of the process code, while error-handling and recovery activities are coded separately in the Process Error Handler.

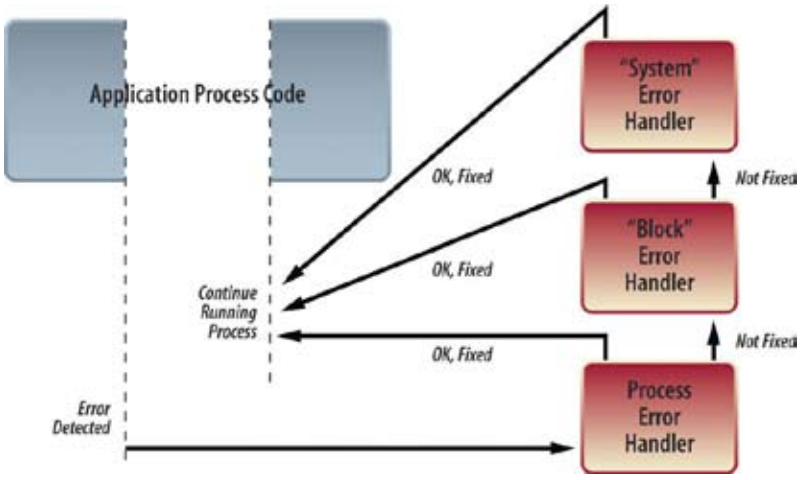

Figure 13. Enea OSE Error Escalation Sequence.

Separating process code from error handling code greatly simplifies development and maintenance. It also facilitates error escalation logic that allows the system to correct errors at different levels of abstraction before triggering the "last ditch" System Error Handler – all without effecting the structure of the normal process code.

Distributed Systems Support

Enea® LINX is a suite of interprocess communications (IPC) services that builds upon OSE's message passing architecture. Enea LINX effectively extends OSE's transparent IPC services from multiple processes running on a single CPU to multiple processes running on multiple CPUs and even multiple operating systems. These services make complex distributed systems easier to conceptualize, model, partition, and scale.

Enea LINX operates as an OSE service in the Core Extensions layer. It permits direct, asynchronous message passing among OSE (and other operating systems) instances in both homogeneous and heterogeneous processing environments, and across a wide variety of interconnects.

Since OSE applications are only concerned with the passing of messages from process to process, it is not necessary for applications to understand the structure of a distributed system. In fact, the location of a given process's communication partner(s) is transparent to the application. Enea LINX takes care of the message delivery, regardless of where the receiving process physically resides (i.e., a different instance of the operating system, another processor). From a distributed systems point-of-view, this approach is merely a transparent extension of OSE's simple, straightforward message passing architecture.

In addition to simplifying distributed design, LINX enhances availability by providing a means of "watching over" processes spread across multiple processors or cores. LINX can send and receive "heartbeat" messages to periodically check the integrity of remote hardware and communication links. When critical processes "die" or become inaccessible, LINX can notify interested processes on any processor, taking the burden off of the application software. This service greatly simplifies the design of distributed, fault tolerant systems.

High-Availability

High-availability is a baseline requirement for many telecommunications systems and is a necessity in NGN, IP-

based communications deployments. In many applications, systems must achieve 99.999% (or "five nines") uptime in order to deliver the continuous, high-quality service that customers have come to expect from PSTN-based systems of the past.

OSE is designed from the ground up with high availability in mind. OSE's memoryprotected, message passing architecture facilitates the design of modular, compartmentalized applications that prevent errant or malicious processes from corrupting the kernel and other application processes. OSE's run time loader enables applications to be field upgraded and hot swapped without shutting down running systems. OSE's advanced multi-level error handling capabilities enhance availability for every load module, whether at the process, block, or system level. And the Enea LINX IPC services extend these high availability (HA) capabilities to multiple instances of OSE running on different processors, cores, and blades. All of these characteristics contribute to OSE's strength as a "five nines", highly available, fault tolerant, real time operating system.

Conclusion

RTOSes have evolved over time to perform more application-specific functions. Unlike most traditional RTOSes, OSE was designed specifically with distributed, fault-tolerant telecommunications systems in mind. OSE's message-passing architecture and general approach to process and memory management, process scheduling, error handling, and distributed communications have made it the choice for millions of telecommunications applications worldwide.

When stability, high availability, development simplicity, maintainability, and performance are the primary criteria for RTOS selection, OSE stands alone, and its track record speaks for itself.

ENEA