



Compact Kernel for Real-Time Payload Processing Embedded Systems

Enea offers an array of Real-Time Operating Systems (RTOS) and development tools to fit your specific real-time application. The Enea OSE[®] real-time kernel for robust, fault tolerant and safety critical applications, and Enea OSE[®]ck, a full-featured, compact, real-time kernel optimized to suit the specific requirements of single and multicore high performance, memory constrained applications. Enea OSEck is a compact kernel and has an extremely small memory footprint, but still combines rich functionality with high performance and true real-time behavior.

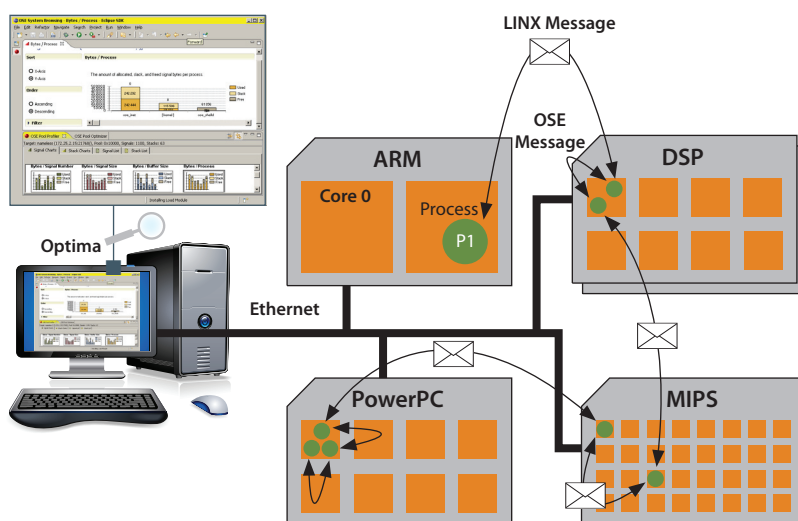
It is a fully pre-emptive kernel, optimized to provide low interrupt latency and high rates of data throughput, yet compact enough to meet the limited hard memory requirements of those applications such as found in digital signal processing (DSP) and packet processing.

Enea OSEck has been designed specifically for use in distributed multicore, and multiprocessor systems, much like OSE's full-featured real-time kernel. Interprocess communication is completely transparent, regardless of whether the communicating processes are located on the same or remote processor.

Enea OSEck also includes a comprehensive IP communication solution, as well as powerful application-level profiling and debugging features. For those reasons, Enea OSEck and the Enea OSE real-time kernel make an unbeatable tandem sharing the same APIs, same concepts and the same transparent IPC mechanism for mixed mode systems.

Enea OSE – a faster way of developing Embedded systems

Enea OSEck meets the needs of developers of real-time systems by combining richness of function with ease of use. Five powerful system calls are all that is needed to write the majority



FACTS

Product Features

- Enea OSE[®]ck is a compact kernel
- Extremely small memory requirements
- Fully event-driven
- Pre-emptive
- Automatic error detection and handling
- Transparent communication mechanism
- Enea LINX[®] interprocess communications services for distributed systems support
- Advanced secure IP networking with Enea DSPNet[®]
- Single- and Multicore processor support
- Support for shell services accessed through secure IP or LINX
- Enea OSE[®] Optima Eclipse tools support
- Support for Enea dSPEED embedded management, profiling and debug services
- Optional OSE simulation environment, OSE Soft Kernel, that allows OSE processes to run on a Windows or Linux host is available

ENEAA

ENEAE OSE[®]ck



of an application. Approximately 30 additional calls are provided for special purposes. This, combined with OSE's high-level modular approach, makes it easy to write compact, efficient and easily maintained code. Despite its high level of functionality, the kernel is remarkably small.

Enea OSEck automatically handles many things normally left to the developer by a conventional real-time operating system. It has sophisticated, built-in error handling and application level debugging. The kernel, regardless of which processor a process is located on, manages all interprocess communication transparently. Together these features significantly reduce the time and cost of developing and maintaining a real-time system, while greatly improving the quality and reliability of the final product.

Hard real-time performance

Enea OSEck provides fast, deterministic real-time performance. All time-critical parts of the kernel are highly optimized. In particular, the interrupt routines in the kernel are extremely efficient, resulting in very low interrupt latency, without affecting the high data throughput rates. All execution times in Enea OSEck are deterministic and are not affected by the size of the application.

Pre-emptive kernel

Enea OSEck is a fully pre-emptive real-time kernel. An interrupt can be served at any time, even during execution of a system call. The pre-emptive kernel ensures quick interrupt response times and general real-time behavior. Pre-emption also occurs while executing interrupt processes.

Compact size

While Enea OSEck delivers a high level of functionality, it is remarkably small. The minimum configuration of the kernel requires less than 4 kB of memory (Texas Instruments TMS320C5000 device). Enea OSEck has a modularity feature, which means that only system calls that are actually used are linked to the final target program image.

Fully event-driven system

In a hard real-time system, system behavior must remain strictly deterministic. Enea OSEck fulfills this requirement by being fully event-driven, resulting in systems where tasks are performed instantly upon request.

Multiple memory pools

With Enea OSEck, each process can access multiple memory pools for signal allocation through multiple algorithms by using specialized system calls. This makes it possible for applications to control the use of memory with special properties, for example internal/external or X/Y type memories, for system level optimization.

Power management

Enea OSE can provide information on what length of time (T) the device can be in sleep mode before the next scheduled OS event. A user-defined function puts desired devices in power down mode. When the system wakes up after the time T, or earlier if an external event occurs, the timer is adjusted with the corresponding time it has been inactive. Enea[®] dSPEED optionally further extends OSEck with host based power management services.

Building blocks of Enea OSE processes

The most fundamental building block in Enea OSE is a process. It is through the use of processes that a system shares CPU time. Enea OSE has different categories and types of processes, each with special functionality.

Process types

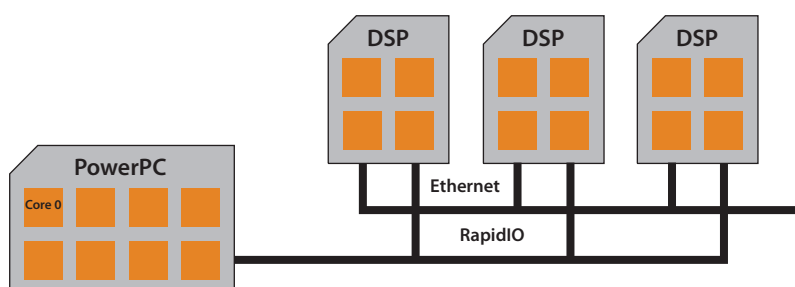
Interrupt processes are scheduled in response to a hardware interrupt, or, using a unique Enea OSE capability, in response to a software event. They run from beginning to end. Interrupt processes have context just as other processes and always have the highest priority among the process types. User interrupts offer faster response than interrupt processes and are advantageously used when low interrupt latency is critical. A user interrupt is faster because it is executed without communicating with the kernel, and it can activate an ordinary interrupt process by using a system call. A prioritized process is the most common process type and typically does the bulk of processing. It is written as an infinite loop that runs as long as no interrupt process or prioritized process with higher priority becomes ready to run.

Process categories

Processes can either be static or dynamic. Static processes are created at system start by the kernel. Static processes exist the entire time that a system exists. Dynamic processes can be created and killed at run-time enabling multiple instances of the same code and preserving system resources. By offering both types of processes Enea OSE enables applications to be optimized for the best performance and flexibility.

Interprocess communication

In Enea OSE, processes can communicate or synchronize in a number of ways including many mechanisms found in conventional operating sys-



A typical system solution includes a host processor, which controls a number of DSPs.



tems. The message-passing architecture and more specifically, the direct message-passing design of Enea OSE brings about the most powerful result of using Enea OSE. Messages are the most powerful and safe method of interprocess communication and can be used between two processes located on the same CPU or two processes in remote CPUs.

Messages

Enea OSE messages are sent from one process to another. A message, also referred to as a signal, contains an ID and the addresses of the sender and the receiver – as well as data. Once a message is sent to another process, the sending process can no longer access it. Ownership of a message is never shared. This important feature eliminates the common error of memory access conflicts that can happen so easily in designs based on conventional operating systems. Furthermore, the unique design of this message “hand-off” in Enea OSE is tremendously more efficient and direct message passing is conceptually more straightforward than the indirect model of other systems.

The receiving process may specify which message types it wants to receive at any particular moment. The process can wait for or poll the message queue for a specific signal, one of a set of signals, or any signal. The

process can also specify a time limit for a signal before timing out.

Simple and safe

Because signals contain the sender PID, receiver PID and data, it is very easy to use and design with processes. Jobs that need to be handled in parallel are dedicated to different processes. When processes exchange information via messages, it is very much like sending a letter from one person to another. The kernel manages mutually exclusive ownership of messages, so a process will not accidentally receive information it should not have.

CPU transparency

Enea OSE message passing is fully transparent between different CPUs. A message is sent in the same manner regardless of whether the receiving process is running in the same or a remote CPU. OSE does not require a master CPU in a distributed system. Each CPU is the same and has the same capabilities. There is no difference in the way a globally visible and a local process is created. A local process can become globally visible to the whole system without re-creation. It is perfectly straightforward to design applications with Enea OSE where the processes across multiple nodes can be viewed as a single system image. This unique capability of Enea OSE enables complex

problems and hardware configurations to be vastly simplified in the software model created for the platform. Enea LINX makes this type of transparent interprocess communication possible.

Multicore

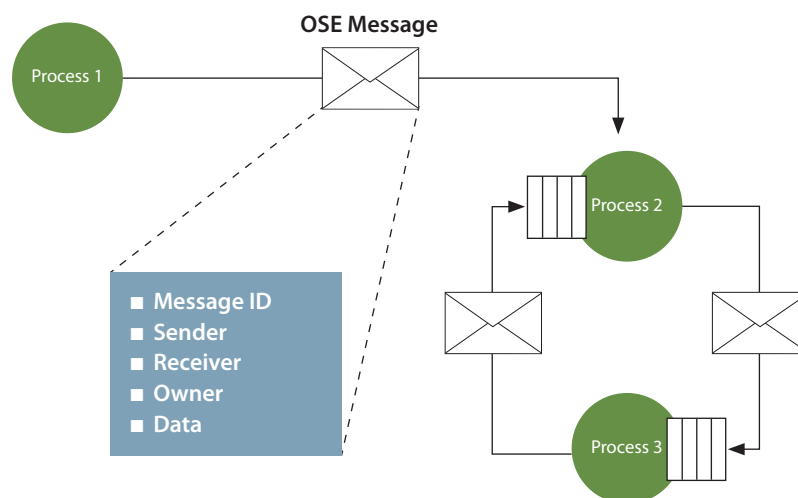
The Enea OSE message passing model is a perfect fit for multicore and multiprocessor environments. With its transparent message passing, support for shared communication peripherals, support for shared execution images, shared memory management, tools, and middleware platform, Enea OSEck abstracts and simplifies multicore development.

Fast semaphores

In addition to message-passing communication, Enea OSE uses a type of semaphore called a fast semaphore. Fast semaphores are intended for time-critical interfaces between interrupt processes and prioritized processes. They provide higher performance than messages but do not offer the same versatility or debugging capabilities. A fast semaphore's high throughput is ideal for embedded systems with fast interrupt handling requirements.

Error handling

Enea OSE supports an error handler which is automatically activated by error events during run-time. The error handler is either called from the application process or the kernel itself. The kernel does not simply return an error code to the user application if something goes wrong, which is the conventional solution. This is a very important safety issue. By automatically invoking the error handler from the kernel, the developer does not have to constantly check for any error codes after Enea OSE system calls. Actions in the error handler are provided by the developer, allowing maximum flexibility and minimal application footprint.



Using OSE's message-passing architecture, OSE messages are sent directly from one process to another. A message contains the identity of both sender and receiver processes.



Debugging and Profiling

Enea OSE builds in support for source code debugging with kernel awareness: processes can be examined for process type, status, signaling queues, and more. Enea[®] Optima goes further with powerful application-level debugging by supporting the visualization of the events based on message passing or errors within the system. Message-level debugging is a powerful way of analyzing multiple boards in a distributed system simultaneously. As events in the system occur, they can be monitored and traced. Application-level debugging complements simple source level instruction stepping for debugging the distributed system design as a whole.

OSEck Soft Kernel Simulation Environment

For development prior to hardware availability, the OSEck Soft Kernel enables engineers to start developing and testing OSE code within a standard Windows or Linux PC environment

Conclusion

Enea's vision has always been to provide telecom, datacom, and safety critical customers, who need to get their product to market quickly, with a complete development environment. Enea OSEck enables customers to differentiate themselves from their competition by using the unique features and benefits found only in Enea OSE products.

SEND
RECEIVE
HUNT
ALLOC
FREE_BUF

OSE's message-based concept, where five powerful system calls are enough to write most of the application code, significantly simplifies the coding and reduces errors of an application.



Systems calls

Basic System Calls

alloc	Allocates a buffer of the requested size.
send	Sends a signal to a destination process.
receive	Receives selected signals.
free_buf	Returns a signal buffer to the pool.
hunt	Search for a process byname and returns the process ID.

Advanced System Calls

addressee	Finds the addressee of a signal.
add_ticks	Adds number of ticks to the system clock.
alloc_nil	Allocates a buffer of the requested size from a system pool. Returns NIL on failure.
attach	Register supervision of another process
cond_halt	Conditionally calls a power off handler.
create_process	Creates a process and returns the process ID.
current_process	Returns the identifier of the calling process.
delay	Puts a process to sleep for a specified number of system ticks.
detach	Remove supervision of another process
error	Reports an error to the error handler.
error2	Reports an error to the error handler, with an additional information parameter.

get_pri	Returns the priority of a process.
get_ticks	Returns the number of ticks since system start.
JUMP_INT	Triggers an OS interrupt process from a user interrupt handler.
NIL	A manifested constant defined by the OSE Kernel.
receive_w_tmo	Receives selected signals with selectable timeout.
reset_pool	Clears all allocation information from a pool.
restore	Makes the caller owner of a signal & clears the redirection information.
s_alloc	Allocates a signal buffer of the requested size from a specified pool.
s_alloc_nil	Allocates a signal buffer of the requested size from a specified pool. Returns NIL if the specified pool is full.
s_create_pool	Creates a new pool and returns the pool ID.
s_dispatch_lock	Locks the dispatcher.
s_dispatch_unlock	Unlocks the dispatcher
send_w_s	Sends a signal with a stated sender.
sender	Returns the identifier of the process that last sent a specified signal.
set_sigsize	Changes the size of a signal buffer reported by a sigsize call.
sigsize	Returns requested size of a signal buffer.
start	Starts a previously stopped process.

stop	Suspends the execution of a single process.
tick	Increments the system timer.
wake_up	Informs an interrupt process of how it was invoked.

Fast Semaphore Calls

get_fsem	Reads the current value of a fast semaphore.
set_fsem	Initializes a fast semaphore with the specified value.
signal_fsem	Increments a fast semaphore value.
wait_fsem	Waits for the fast semaphore to become non-negative.

Semaphore Calls

create_sem	Creates and initializes a semaphore.
get_sem	Reads the current value of a semaphore.
kill_sem	Returns a semaphore to the memory pool.
signal_sem	Increments the semaphore value.
wait_sem	Waits for the semaphore to become non-negative.

