EMERALDS: A Microkernel for Embedded Real-Time Systems

Khawar M. Zuberi and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122

Abstract—*EMERALDS (Extensible Microkernel for Embedded, ReAL-time, Distributed Systems) is a real-time microkernel designed for cost-conscious, small to medium size embedded systems. It not only offers standard OS services like multi-threaded processes, real-time scheduling, protected address spaces, message-passing, semaphores, and timers, but does so in an efficient manner while keeping the kernel size to just tens of kilobytes. For efficiency, EMERALDS uses the novel approach of mapping the kernel into each user-level address space, so even with full memory protection, system calls do not need context switches unless a user-level server is involved. EMERALDS also provides the flexibility for users to add communication protocol stacks and device drivers as user-level servers without modifying the kernel. We have completed a uniprocessor version of EMERALDS for the Motorola 68040 processor whose size is under 13 KBytes. Context switch takes under 12 μs and system calls have overhead just 1.8 μs more than that of simple subroutine calls.*

1 Introduction

Real-time computing systems [1] must behave predictably in possibly unpredictable environments. This predictability is usually ensured by a real-time operating system (RTOS). The variety of real-time applications — from soft multimedia applications to hard real-time automotive control systems — have resulted in dozens of RTOSs being designed for both commercial and research purposes. Some RTOSs are "general-purpose" in that they cater to a wide range of real-time applications. Examples include commercial RTOSs like pSOS [2], QNX [3], and VxWorks [4]; and research ones like Real-Time Mach [5]. Somewhat more specialized RTOSs include HARTOS [6] and the Spring Kernel [7], designed for parallel and distributed platforms. Even more specialized RTOSs include CHAOS [8] and Harmony [9], designed primarily for complex robotic systems.

In this paper we present the EMERALDS designed specifically for mass-produced, small- to medium-sized embedded systems. Such systems consist of a small number of microprocessors (about 10 or less) interconnected by a local area network. These systems are commonly used for automotive applications, robotics, industrial automation, etc. Keeping per unit costs down is vital in such systems. This necessitates that any RTOS to be used in these systems must not only be predictable but also *efficient* and *small* in size. The reason for requiring efficiency is obvious: an RTOS which incurs less overhead needs less powerful (and cheaper) processors to do the same job that a less efficient RTOS will do using more expensive hardware. Reason for requiring a small-sized OS also has to do with keeping costs down. Our target applications usually have their executable code (including the RTOS) stored in non-volatile memory like ROM. A smaller RTOS means that less ROM is needed to store it and less time to execute it. So, if the OS size is tens of KBytes instead of hundreds of KBytes (which is typical for modern RTOSs), it will result in considerable cost savings in ROM chips. Since the system is mass-produced, savings of even a few dollars per unit translate into millions of dollars overall savings.

The main goal in designing EMERALDS was to see which features of embedded systems can we use to reduce size and increase efficiency. Embedded systems provide many opportunities for simplification. They do not require the same level of security between processes as do more general RTOSs. Processes tend to exchange short, simple messages like sensor readings and actuator commands. A file system is usually not needed: all executable code is in ROM and all dynamic memory requirements are satisfied by RAM. These characteristics allowed us to reduce the system call overhead, simplify inter-process communication (IPC), and keep EMERALDS' size down to a minimum.

An important question related to reducing size was which OS services to include in EMERALDS and which to leave out. Many RTOSs leave out common OS features like memory protection and threads in an attempt to reduce size and increase speed. We did not take this approach. Instead, we provide all common OS services but transfer some addressing and naming responsibilities to the programmer, which we show not to be much of a burden at all for embedded systems. This enabled us to meet our goals of efficiency and small size without cutting back on OS services.

In the next section we give a brief overview of EMERALDS. Section 3 describes how EMERALDS was optimized by using the above-mentioned simplifying characteristics of embedded systems, and the features it provides to make it efficient and easy to use (including its novel system call mechanism and device driver support). Sections 4–7 give the details of EMERALDS, covering processes, threads, scheduling, memory protection, IPC,

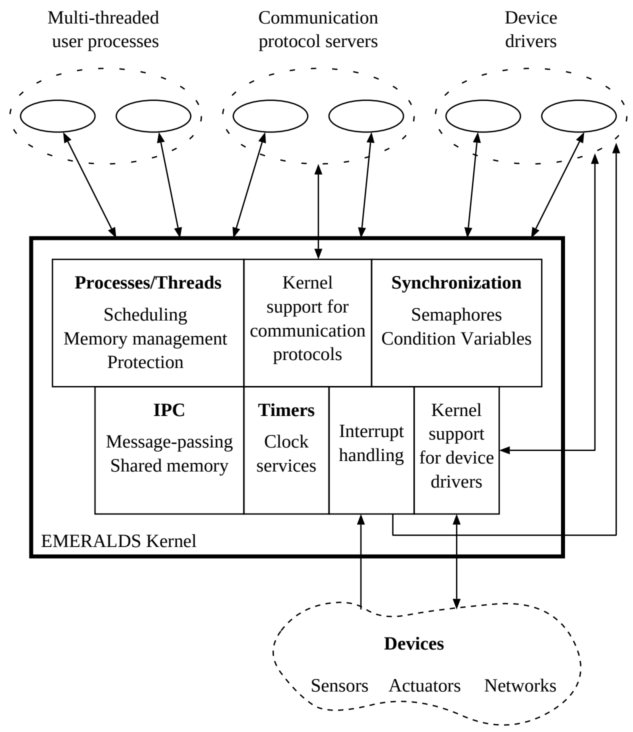

Figure 1: EMERALDS' architecture.

etc. Section 8 gives some timing results and Section 9 states conclusions and future work.

2 Architectural Overview

EMERALDS is a microkernel real-time operating system written in the C++ language. Following are EMERALDS' salient features as shown in Figure 1.

- Microkernel architecture:
  - Easy to install new communication protocols. Multiple protocol stacks can exist on one node.
  - Easy to install new device drivers.
- Multi-threaded processes:
  - Full memory protection between processes.
  - Threads are scheduled by the kernel.
- IPC based on message-passing and mailboxes. Shared memory is also provided.
- Semaphores and condition variables for synchronization; priority inheritance for semaphores.

The obvious question is then: why are these features included in EMERALDS and why are other OS features (like disk-related services) not part of EMERALDS? This question is answered in the next section.

3 Design for Embedded Use

An embedded systems designer cannot use an off-the-shelf RTOS because the RTOS is too large & inefficient, or it is not easy to use. To reduce the size of the kernel, we must omit some features, but this may make the OS difficult to use. This was the biggest challenge in the design of EMERALDS — deciding what to include and what to leave out. Fortunately, embedded systems have several characteristics which allow OS services to be simplified.[1] In the following, we first outline these characteristics, then summarize those features and design choices which make EMERALDS efficient and easy to use in small- to medium-sized embedded systems.

3.1 Simplifying Characteristics

Following are some characteristics of embedded systems that allowed us to simplify the implementation of EMERALDS to achieve our goal of a small, fast kernel.

Cooperative Processes: In time-shared systems, there is always the danger that some processes may act maliciously and may intentionally harm or confuse other processes. This makes protection an important concern in time-shared systems. But in embedded systems, all processes belong to one user — the application designer — so they do not act maliciously (at least not by intention). This means that an OS designed for embedded systems does not need all the protection features present in more general OSs. The way we use this characteristic in EMERALDS is as follows. EMERALDS provides several special system calls for writing communication protocol servers and device drivers. Some of these calls should never be used by a non-server thread, but instead of enforcing such rules, EMERALDS trusts the application designer to follow them. The (safe) assumption is that the application designer will want his design to work properly, so will never violate these simple guidelines. This assumption helps not only in reducing the size and increasing the speed of the kernel, but also in making device drivers and communication stacks easy to install and use in EMERALDS.

Location of Resources is Known: Most operating systems provide naming services to translate easy-to-remember names into numerical identifiers which may also contain location information. Examples include a file system directory (translation from filename to location on disk), internet domain name service (translation from machine name to IP address), etc. These services are necessary in large, rapidly-changing systems where resources may move about. However, in embedded systems, the designer is very much cognizant of the location of various resources. For example, he knows which threads run on which node (because he is the one who placed them there). These resources usually stay fixed unless there is a major design change. This is why EMERALDS does not provide any naming service. For example, to send a message to a remote thread, the local thread must know the node on which the remote thread runs and the identifier of that thread's mailbox. When creating a shared-memory segment, a process must supply a CPU-wide unique identifier. If some other process wants to use that segment it must also know that identifier. Same applies to semaphores and condition variables.

[1]This is why EMERALDS is not POSIX-compliant. POSIX requires features like signals, naming schemes, a file system, etc., to be included. These are not needed in embedded systems. So even though EMERALDS provides many POSIX features relevant to embedded systems (like threads, condition variables, and message-passing with queues), it cannot be fully POSIX-compliant and still be as small in size as it is.

In short, the programmer must now do the book-keeping otherwise done by a naming service. Then, the question is: how much of a burden is this for the programmer? We believe that when dealing with embedded systems, it is not much of a burden at all because the programmer knows where all the resources are and what their identifiers are. Macros placed in a common header file can be used to make this book-keeping easy and maintainable. For example, to send messages to some remote thread $T1$ on node number 3 and with a mailbox with identifier 5, the following macros may be defined:

```
#define     NODE1            3
#define     T1_MBOX          5
#define     T1               NODE1,T1_MBOX
```

Then, macro `T1` can be used as a "name" for $T1$'s mailbox. If, during the design cycle, location or identifier of any resource changes, only one header file has to be modified. That is, for embedded systems it is not difficult at all for the application programmer to do the address book-keeping himself, and if macros are used properly, this information is also easy to maintain.

Memory-Resident Applications: Our target applications, in general, do not use disks. ROM is used as non-volatile storage and on-board RAM satisfies all run-time memory requirements of the application. So EMERALDS provides neither a file system nor a backing store for virtual memory. This means no device drivers for disks, no disk buffer management, no page fault handlers, or any other such services.

Simple Messages: In embedded systems, the most common messages are sensor readings and actuator commands. Threads can exchange such simple messages by talking directly to the network device driver without using any protocol stack, so EMERALDS does not have a built-in communication protocol stack.

3.2 Extensibility, Efficiency, and Usability

Here we describe those features of EMERALDS which make it fast and user-friendly.

Microkernel Architecture: The microkernel architecture of EMERALDS was necessary to allow users the flexibility to install their own communication protocols and device drivers.

As already mentioned, nodes in most embedded systems exchange simple messages for which no protocol stack is needed. Such systems prefer to access the network directly to get maximum bandwidth. However, some applications require more complicated communication protocols to handle duplicate detection, ordering, message fragmentation, etc. In EMERALDS, communication protocols run as user-level servers, so users are free to use protocols which suit their particular applications. The server can even be bypassed completely to directly access the communication network if needed. Moreover, EMERALDS provides the flexibility to have multiple protocol stacks on the same node. For example, one set of processes may require that messages be causally ordered, then they can use one protocol stack. Another set of processes may not have this requirement, so they can use a simpler (and faster) protocol stack.

Similar flexibility is needed regarding device drivers. Since there are so many devices (e.g., sensors, actuators, network adapters) used in embedded systems, it is virtually impossible for the OS designer to supply device drivers for all of them. The next best thing is to make it as easy as possible for users to write their own device drivers. EMERALDS does just that. A device driver is just a user process (instead of being part of the kernel), and special system calls are available for device drivers to access devices and deal with interrupts. We use the cooperative processes assumption (as already discussed) to rely on the application designer to ensure that these special system calls are never used with conflicting parameters. For example, processes are expected not to try to attach their own separate interrupt service routines to the same interrupt level. This and other such simple restrictions can be easily ensured at design time.

Need for Memory Protection: Providing memory protection requires maintaining page tables and programming the memory management unit. This not only increases the size of the kernel, but also adds overhead to several kernel services, thus being contrary to our primary goal of building a small and fast kernel. Here we justify providing memory protection in EMERALDS.

The need for memory protection in time-shared systems is indisputable. One user's processes must be protected from all other — possibly malicious — users. But in embedded systems, all processes are cooperative and will never try to intentionally harm another process, so providing memory protection seems extraneous. However, bugs in application code can manifest themselves as malicious faults. For example, suppose some pointer in a C program is left uninitialized. If this pointer is used for writing, one process can easily corrupt another process or even the kernel. With memory protection, such an access will cause a TRAP to the kernel and recovery action may be taken, providing a form of software fault-tolerance. Without memory protection, such a fault may not even be detected until the CPU crashes with possibly catastrophic consequences.

Another benefit of memory protection is easier debugging of application code. During application development, if there is no memory protection, each time one process crashes, the entire CPU may crash. This makes tracking down bugs extremely frustrating and time-consuming. With memory protection, software failures are contained within an address space and can be easily tracked down.

Efficient System Calls: Above we mentioned the advantages of memory protection. Its disadvantage is the context switch overhead incurred when making system calls (because the user and kernel usually exist in separate address spaces). This is why some RTOSs omit memory protection so they only have to make subroutine calls to access kernel services; not so with memory protection.

We resolved this problem by mapping the kernel into each user-level address space (unlike other OSs in which the kernel runs in its own address space). This way, a system call reduces to a TRAP, then a jump to the appropriate kernel address, without the need to switch address spaces. Details are given in Section 5. □

The flexibility EMERALDS offers in writing protocol stacks and device drivers, its small size and speed set it apart from other modern RTOSs. EMERALDS also provides many standard features found in today's RTOSs such as multi-threaded processes, message-based IPC, real-time priority-based scheduling, synchronization variables, timers, etc. These features are described in detail in the following sections.

4 Processes and Threads

EMERALDS provides multi-threaded processes. A process in EMERALDS is a passive entity, representing a protected address space in which threads execute. Each thread has a user-specified priority and is preemptively scheduled by the kernel based on this priority. Table 1 lists the EMERALDS system calls related to processes and threads.

System call	*Imp. Params*	*Function*
`create_proc()`	Thread priority	Create process with 1 thread
`create_thread()`	Thread priority	Create thread
`join_thread()`	Thread ID	Wait for child thread to finish
`detach_thread()`	Thread ID	Tell kernel: will not wait for thread

Table 1: Process and thread system calls.

The two most important features of EMERALDS processes and threads are memory protection and real-time scheduling.

4.1 Memory Protection Implementation

The benefits of memory protection mentioned earlier will not be of much practical use if the implementation of memory protection was not efficient and small-sized. To meet these goals, we made full use of the fact that our target applications are in-memory. This enabled us to reduce the total size of a page table to a few KBytes compared to several megabytes for virtual memory systems with disk backing stores. In the latter, the entire page table must exist, even if most of the address space is unused. This is needed to distinguish unmapped pages from those which have been swapped out to disk. But for in-memory systems, this distinction is not needed. This allows the page table to be trimmed down using the hierarchical nature of most page tables. For example, the Motorola 68040 has three-level page tables. Each third level page table represents 256 KBytes of address space. So, if a process has three segments — code, data, and stack — and each is less than 256 KBytes, then its page table will be as shown in Figure 2.

All but three entries in the first-level page table are null, so only three second-level page tables exist. An attempt to access an address covered by an invalid entry will result in a TRAP to the kernel indicating a bug in the software. Similarly, in each second-level page table, only one entry is valid and all other third-level page tables do not exist. This way, total size of the page table is

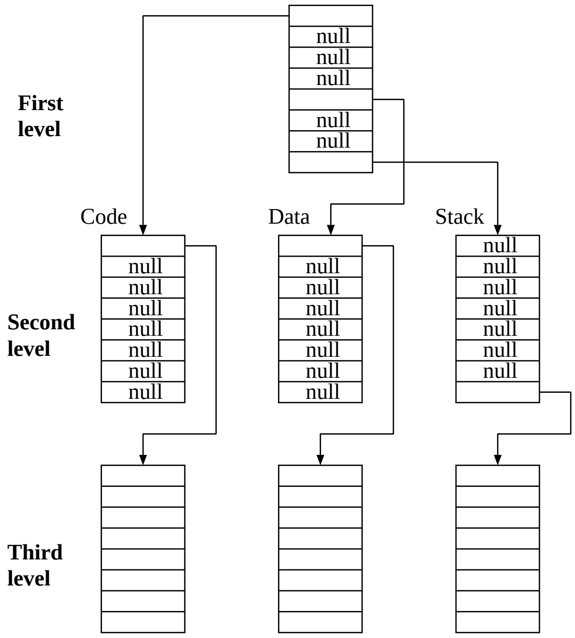

Figure 2: A typical page table in EMERALDS. The hierarchical structure is used to reduce the size of the page table.

just 2432 bytes for a page size of 8 KBytes. (More third-level page tables are needed if any segment exceeds 256 KBytes). While this example is specific to the MC 68040, most other modern CPUs also provide three-level page tables with similar parameters.

The small size of page tables not only saves memory, but also enables other optimizations like mapping the kernel into every address space (see Section 5). This greatly reduces the overhead associated with system calls, making our implementation of memory protection feasible for embedded systems.

4.2 Preemptive Real-Time Scheduling

Currently, EMERALDS provides fully-preemptive, fixed-priority scheduling and partial support for dynamic scheduling. The user specifies a thread's priority at creation time. Users may choose priorities based on rate-monotonic [10], deadline-monotonic [11], or any other fixed-priority scheme suitable for the application at hand. Also, a system call is provided to change a thread's priority at run-time to respond to changing operating conditions. This can be used to emulate dynamic earliest-deadline first (EDF) [10] scheduling at the user level (even though currently, EDF is not explicitly supported by the kernel). EMERALDS allows for 32-bit non-unique thread priorities, so by setting a thread's priority to its deadline (and re-adjusting every time the task is invoked), EDF scheduling can be realized.

5 Efficient System Call Mechanism

The biggest advantage microkernels have over monolithic operating systems is modularity. By moving functionality to user-level server processes, services can be added, removed, or modified as needed. The disadvantage is the extra overhead. A system call which relies on

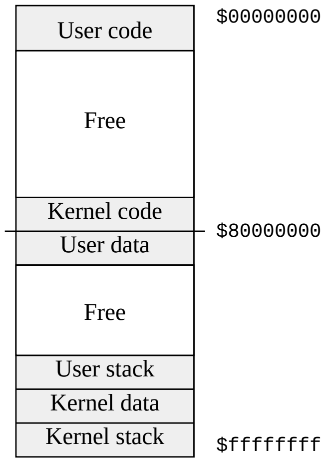

Figure 3: A typical address space in EMERALDS. Area labeled *kernel stack* is used for interrupts and area labeled *user stack* is used by both the user and the kernel.

a user-level server involves at least four context switches: from user to kernel to server back to kernel and returning back to the user. Moreover, the server process will usually make some system calls, each requiring at least two context switches. This extra overhead is particularly unattractive in embedded systems such as in automotive applications. This overhead can force designers to use more powerful processors than they would otherwise, costing millions of dollars more when the system is mass-produced. Hence, for a microkernel to be viable for use in such an embedded system, it must somehow reduce the system call overhead. EMERALDS achieves this by mapping the kernel into each process's address space. A typical 32-bit EMERALDS address space is shown in Figure 3.

With this type of mapping, a switch from user to kernel involves just a TRAP (which switches the CPU from user to kernel/supervisor mode) and a jump to the appropriate address; there is no need to switch address spaces. Also, system call code in EMERALDS is designed to take parameters straight off the user's stack (possible since both kernel and user are in the same address space). This scheme has the following advantages.

- No need to copy parameters from user space to kernel space. All that Locore.S (assembly code used for making system calls) does is point the kernel stack pointer to the user stack and some other minor stack adjustments. As a result, system calls in EMERALDS (except those involving servers) have an overhead comparable to that of a subroutine call (see Section 8).
- No need to translate pointers. If the user and kernel are in separate address spaces, the data pointed to by a pointer must be copied to the kernel's address space, and a pointer to this copy passed to the system call routine. But in EMERALDS, all user pointers are valid inside the kernel, so no need to do any data copying.

Now, let's go back to the overhead of system calls involving servers. Instead of having four address space switches, there are only two (from user's address space to server's and back). Switching between user and kernel levels doesn't cause an address space switch and incurs far less overhead. Even if the server has to make its own system calls, no more address space switches will occur, unless the calls involve other servers (which typically occurs only when the server accesses a device driver). In this way, the EMERALDS microkernel is feasible for use in cost-conscious embedded systems while retaining the full flexibility and modularity typical of microkernels.

Implementation: Mapping the kernel into each user address space is feasible in EMERALDS because both the kernel and its data segment are so small. In other operating systems with standard virtual memory, the size of the kernel's data segment is so large (due to large page tables) that mapping it into each address space is not feasible. The mapping is achieved by having appropriate second-level page table entries point to common third-level page tables which map the kernel. Thus, size of a process's page table is not affected. Also, the kernel areas are protected from corruption by faulty user code by using page table entries to mark them as read-only for user mode. This way, user processes are protected from each other and the kernel is protected from user processes.

6 Inter-Process Communication (IPC)

The primary IPC mechanism in EMERALDS—for both inter- and intra-processor communication—is message-passing using mailboxes.[2] For intra-processor communication, EMERALDS also provides shared memory.

6.1 Message-Passing

EMERALDS provides the system calls listed in Table 2 for exchanging messages between threads. These calls are used to create & delete mailboxes and send & receive messages. EMERALDS also allows a 32-bit priority to be assigned to each message which is used to sort messages in a mailbox so that the receiver thread retrieves the highest-priority message first.

Message-passing in EMERALDS has been designed with efficiency and flexibility in mind. Most communication networks designed for embedded, real-time systems such as CAN [12], TTP [13], SERCOS [14], SP50, etc., provide the bottom two layers of the ISO OSI reference stack (the physical and data-link layers) which is sufficient for exchanging simple messages (all that the sender has to do is talk directly to the network device driver). For more complex IPC, the remaining stack layers must be implemented in software. So EMERALDS allows both direct network access as well as use of protocol stacks — providing the former as a special case of the latter. EMERALDS also provides optimizations for local message-passing between threads on the same node. In the following, we first describe the EMERALDS mechanisms for simple local message-passing, then give details of how protocol stacks and network device drivers can be used for more complicated IPC.

[2] In the current uniprocessor version of EMERALDS, only the intra-processor part has been implemented.

System call	*Important Parameters*	*Function*
`mbox_create()`	CPU-wide unique identifier	Create mailbox
`mbox_delete()`	Mailbox identifier	Delete mailbox
`msg_send()`	Destination node and mailbox, local server mailbox	Send message to mailbox
`msg_receive()`	Mailbox identifier	Retrieve message from mailbox
`try_msg_receive()`	Mailbox identifier	Non-blocking version of `msg_receive()`
`msg_send_direct()`	Destination mailbox	Send message, but bypass redirection
`msg_receive_full()`	Mailbox identifier	Retrieve message with meta-information

Table 2: Message-passing system calls. The last two calls are for use by protocol servers.

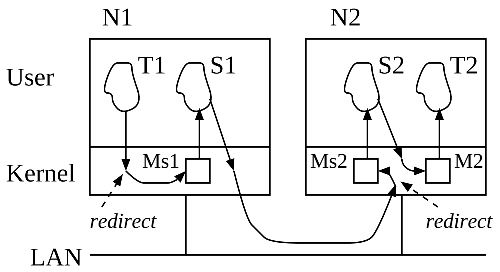

Figure 4: Message passing with protocol stacks (device drivers not shown for simplicity).

Local Message-Passing: Suppose thread $T1$ wants to send a message to another thread $T2$ on the same node. The latter has a mailbox with identifier $M2$. Thread $T1$ will use the `msg_send()` system call to send this message, specifying the destination mailbox ($M2$ in this case) and message priority. The kernel deposits the message directly into $M2$, then unblocks $T1$. $T2$ can then retrieve this message from $M2$ in two ways:

1. $T2$ can execute a `msg_receive()` system call.
2. When $T2$ creates $M2$, it can specify an interrupt service routine (ISR) to be executed whenever a message arrives in $M2$. This ISR may execute `msg_receive()` to retrieve the message.

The first mechanism is suitable for periodic messages while the second one is for infrequent sporadic and aperiodic messages.

Note that `msg_receive()` is a blocking system call which may not always be suitable for real-time systems. Thus, EMERALDS provides its non-blocking counterpart `try_msg_receive()` which returns an error if a mailbox has no messages.

Using Communication Protocol Stacks: There are many reasons why an embedded system may need a (partial or complete) communication stack. One simple reason is to add count information to detect duplicate transmissions. Some applications may also require that messages be properly ordered in which case timestamps [15] or logical clocks [16] may have to be attached to messages. To cater to the needs of such applications, the EMERALDS microkernel allows communication stacks to be used with message-passing in the form of user-level servers. Figure 4 shows this form of message-passing. The figure shows message-passing across the network, but the same applies to local message-passing as well.

Protocol stacks can be used in the following way:

- On the receiver side, a server is attached to mailbox $M2$ by specifying the server's mailbox $Ms2$ as a parameter to `mbox_create()`. Then any message which arrives for $M2$ is redirected by the kernel and deposited in $Ms2$. The server will strip message headers, take appropriate actions, then send the message directly to $M2$.
- On the sender side, the server's mailbox $Ms1$ is given as a parameter to `msg_send()`. The kernel then simply deposits the message in $Ms1$. The server will append headers, then use the network device driver to send the message to its destination (as described in the Section 6.1).

EMERALDS provides two special system calls for writing protocol servers. The call `msg_receive_full()` not only retrieves the message from the mailbox but also returns meta-information like the final message destination (mailbox $M2$ on node $N2$ in our example) and whether this message is incoming or outgoing. The `msg_send_direct()` call is used by receiver-side servers to send a message (after stripping headers) directly to the destination mailbox ($M2$ in our example). It is different from `msg_send()` in that it bypasses the redirection scheme; otherwise, the message will be caught in an infinite loop.

Priority of Server Processes: User-level servers which implement protocol stacks cannot run at a fixed priority all the time because of a form of the priority inversion [17] problem. Suppose the server runs at a low priority and some message arrives for a high-priority thread $T1$. As long as there are runnable threads of medium priority, the server will not run and $T1$ will be forced to wait for the message and may miss its deadline. Making the server run at a high priority is also infeasible for similar reasons.

Our solution to this priority inversion problem is to use priority inheritance [17] which, in the context of message-passing in EMERALDS, works as follows. Define an *inherited thread priority* (ITP) associated with each message. For an outgoing message, it is the priority of the sender thread, while for incoming messages, it is the priority of the receiver thread. We want the server thread's priority to be the highest ITP of all messages in its mailbox. To do this, the kernel must know two things: which threads are server threads and when does a server thread finish processing a message (at which time its priority should be re-adjusted). We use semantic information for this purpose. A server runs an endless loop of the form

```
loop:  msg_receive_full();
            ⋮
```

```
handle message
   ⋮
goto loop;
```

To distinguish a server thread from ordinary threads, we use the fact that only servers use the `msg_receive_full()` system call (that is, we rely on the application designer not to use this system call in non-server threads). Also, once a server finishes handling a message (and it is time for the server's priority to be re-adjusted), the next statement it executes is `msg_receive_full()`. So we adjust the server's priority as part of this system call. When this call is made, first the calling thread's (i.e., the server's) priority is adjusted based on ITP of queued messages, and then a message is popped from the queue and returned to the server. Also, if a new message arrives in the server's mailbox with an ITP higher than the server's current priority, then the server's priority is immediately increased to that ITP.

Network Device Drivers: EMERALDS device drivers in general are discussed in Section 7.3, but here we mention some details specific to network device drivers and their interaction with other threads.

Network device drivers in EMERALDS must be cognizant of the concept of mailboxes, i.e., they must implement part of the network layer of the ISO OSI stack. Specifically, they have to add the mailbox identifier to a message before transmitting it, and strip it off when receiving a message.

On the sender side, accessing a network device driver is no different from accessing a protocol stack because both run as user-level servers. All that has to be done is to specify the device driver's mailbox in the `msg_send()` call. The device driver can use `msg_receive_full()` to retrieve the message, append the destination mailbox to the message, then send it out over the network to the destination node.

On the receiver side, when the device driver receives this message, it will look at the destination mailbox, then use local message passing to send the message to that mailbox. If a protocol stack has been tied to that mailbox, the message will be automatically redirected to the protocol server.

This scheme allows communication protocol stacks to be completely bypassed, if needed. User threads can send messages directly to device drivers, and receive messages directly from device drivers. This type of message passing is useful for exchanging short, simple messages like sensor readings and actuator commands, and is more efficient than message-passing using protocol stacks.

6.2 Shared Memory

EMERALDS allows page-based sharing of memory between processes running on the same CPU. Two system calls are provided for this purpose: `shm_attach()` and `shm_detach()`.

The system call `shm_attach()` is called with an identifier. If no shared-memory segment exists with this identifier, then physical memory is allocated and a new segment is created. This segment is mapped into the calling process's address space and a pointer to the start of the segment is returned. When `shm_attach()` is called again with the same identifier by any process on the same CPU, the kernel finds the segment with that identifier and maps it into the calling process's address space (no new memory is allocated).

The system call `shm_detach()` does the opposite of `shm_attach()`. It unmaps the named segment from the calling process's address space. Moreover, if no other process has this segment mapped into its address space, then the physical memory associated with the segment is also freed up. This provides a simple programming model. When processes need to use a shared memory segment, they call `shm_attach()` with that segment's identifier. The first such call allocates physical memory and all the later ones just map in the segment. When processes no longer need a segment, they call `shm_detach()`. These calls unmap the segment from their address space, except the last call which also frees up the physical memory. These semantics are easier to use than, for example, UNIX semantics where shared memory must be explicitly created before mapping it into an address space, and must be explicitly deleted after unmapping it from each address space.

7 Miscellaneous OS Services

7.1 Semaphores

Threads often need to ensure mutual exclusion when accessing critical regions of code dealing with shared resources. EMERALDS provides semaphores (sometimes also known as *mutexes* as in POSIX terminology) for this purpose. The system calls in Table 3 are used to create, delete, lock, and unlock semaphores. If a thread tries to acquire a semaphore which is already locked, that thread will block and will be added to a queue of threads waiting for that semaphore. When the lock holder releases the semaphore, the highest-priority thread in the queue will be unblocked. An alternative to the blocking `sem_lock()` call is the `sem_trylock()` call which returns an error if the semaphore is already locked. If the semaphore is free, it will be locked.

System call	*Important Params*	*Function*
`sem_create()`	CPU-wide unique ID	Create sem.
`sem_delete()`	Semaphore identifier	Delete sem.
`sem_lock()`	Semaphore identifier	Acquire sem.
`sem_trylock()`	Semaphore identifier	Non-blocking
`sem_unlock()`	Semaphore identifier	Release sem.
`cv_create()`	CPU-wide unique ID	Create CV
`cv_delete()`	CV identifier	Delete CV
`cv_lock()`	CV identifier	Acquire CV
`cv_unlock()`	CV identifier	Release CV

Table 3: System calls for semaphores and condition variables.

Priority Inheritance: The priority inversion problem with locks is well understood. The standard solution is priority inheritance which is built into EMERALDS.

If a high-priority thread T_h calls `sem_lock()` on a semaphore already locked by a low-priority thread T_l, the latter's priority is temporarily increased to that of the former. Without priority inheritance, a medium priority thread T_m can get control of the CPU by preempting T_l while T_h remains blocked on the semaphore. With priority inheritance, T_l will keep on running until it unlocks the semaphore. At that point, its priority will go back to its original value, but now T_h will be unblocked and it can continue execution.

7.2 Condition Variables

Condition variables differ from semaphores in the effect of signaling the variable. When a semaphore is signaled (using `sem_unlock()`), its effect lasts. This means that even if no thread is currently blocked waiting for the semaphore, the signal will not be lost. If later on a thread tries to acquire the semaphore, it will succeed. On the other hand, signaling a condition variable has no effect if no threads are waiting on that condition variable at the time of the signal.

The system calls for condition variables are similar to those for semaphores and are listed in Table 3. Note that a system call such as `cv_trywait()` does not make sense with condition variable semantics, so it is not provided by EMERALDS.

7.3 Device Drivers

EMERALDS provides two special system calls to write device drivers. The first, `map_device()`, allows a device driver to map a memory-mapped device into its address space.[3] From then on, the device driver can use standard memory operations to access the device. The `set_isr()` system call allows device drivers to handle interrupts. Device drivers use this call to tell the kernel which ISR subroutine to execute when an interrupt occurs. A separate ISR can be attached to each interrupt level. As far as communication between user threads and device drivers is concerned, standard EMERALDS IPC mechanisms (message-passing and shared memory) can be used since EMERALDS device drivers run as user-level threads.

EMERALDS allows even non-device driver threads to use the above system calls. When used responsibly, this can be a great asset in embedded real-time systems. Usually, just one (possibly multi-threaded) process is responsible for directly communicating with a certain device like a sensor or an actuator. In this situation, it becomes very efficient to integrate the device driver with that process. This way, the device can be accessed using subroutine calls — completely avoiding context switch overheads. Such optimizations are not possible in other OSs which require device drivers to reside inside the kernel.

7.4 Memory Management

The system call `mem_alloc()` can be used by a process to get the desired number of pages of physical memory mapped into its address space. This call returns the starting address of the allocated space, and can be used to build library-based memory allocators to provide C calls like malloc() and free(). Memory obtained through `mem_alloc()` is retained by a process until it terminates, at which time all its memory is reclaimed by the system.

[3] Currently, EMERALDS does not support I/O-mapped devices because the MC 68040 — on which EMERALDS is presently implemented — does not have a separate I/O space.

7.5 Timers

The call `start_timer()` can be used to create and start a timer. This call has two variants. The first is a blocking version, in which the calling thread blocks for the specified duration of time. The second non-blocking version is used to execute a timer ISR. The calling thread specifies a routine to be executed as the ISR and a time delay. When the timer expires, the ISR is executed, and it can reset the timer for the next interrupt. This way, the ISR can execute periodically.

8 Performance

We have completed a uniprocessor version of EMERALDS for the MC 68040 processor which is one of the most popular and commonly used processor in embedded systems with a wide installed base. The size of this version of EMERALDS is about 13 KBytes. Comparing this to other major RTOSs for embedded applications (Table 4), we see that our goal of a small-sized RTOS has been achieved.

RTOS	*Size (KBytes)*
QNX	101
VxWorks 5.1	286
EMERALDS	13

Table 4: Sizes of various RTOSs (uniprocessor versions). Size of QNX is from [3] and includes the "kernel," `Proc`, and `Dev` modules which is the minimal configuration with device driver support. VxWorks' size is from a compiled stand-alone version.

Table 5 shows the latencies of some system calls and other operations in the current version of EMERALDS on a 25 MHz 68040 processor with two independent 4 KByte instruction and data caches. Latencies are measured using a 5 MHz clock (the fastest clock available on the Ironics IV-3207 boards we use). The operations labeled with * involve a context switch to another thread. All other operations return to the calling thread. System calls which change a thread's priority such as process/thread calls and calls with priority inheritance (`sem_lock()` and `sem_unlock()`) also include the overhead of sorting the list of threads according to their new priorities.[4] This overhead depends on the number of threads in existence. The timing results are for 20 threads unless stated otherwise.

Comparing the `null()` system call to the `null()` subroutine call, we see that EMERALDS' technique of mapping the kernel into each address space results in efficient system calls, incurring only a 1.8 μs more overhead than subroutine calls. Even when a context switch to a different address space is required, it incurs less than 12 μs overhead.

9 Conclusion

Small to medium sized embedded real-time systems are becoming increasingly common in applications like

[4] In many OSs which support dynamic scheduling, this sorting/searching is done as part of the context switch. In EMERALDS, we do the sort only when necessitated by changed thread priorities, instead of doing it on every context switch.

Operation	*Latency (μs)*
Context switch (kernel to thread)	11.8
`null()` subroutine call	0.2
`null()` system call	2.0
`create_proc()`*	194.4
`create_thread()`*	50.4
`join_thread()` (thread already exited)	17.2
`detach_thread()` (thread already exited)	16.4
`detach_thread()` (thread has not exited)	2.2
`shm_attach()` (one page mem. allocated)	10.6
`shm_attach()` (attach existing segment)	8.6
`shm_detach()` (one page mem. deallocated)	10.0
`shm_detach()` (just unmap segment)	8.0
`sem_create()`	4.8
`sem_delete()`	3.2
`sem_lock()` (semaphore free)	5.8
`sem_lock()`* (sem locked; 20 threads)	39.8
`sem_lock()`* (sem locked; 10 threads)	33.6
`sem_unlock()` (no thread waiting)	7.0
`sem_unlock()`* (thread waiting; 20 threads)	42.0
`sem_unlock()`* (thread waiting; 10 threads)	35.6
`cv_create()`	6.2
`cv_delete()`	5.4
`cv_wait()`*	25.4
`cv_signal()` (no thread waiting)	3.4
`cv_signal()`* (thread waiting)	30.4
`mbox_create()`	6.4
`mbox_delete()`	3.6
`msg_send()` (10 bytes, no server)	12.6
`msg_receive()` (10 bytes, no server)	11.0

Table 5: Timing of various operations in EMERALDS.

automotive control, robotics, and industrial automation. To be competitive in the market, these systems must reduce cost to a minimum. Any RTOS to be used in these systems must therefore not only support predictability (essential in any real-time system and provided in EMERALDS in the form of predictable scheduling of threads) but also be efficient and small in size. Efficiency allows cheaper processors to be used and small size decreases the cost of ROM needed to store the executable code. Most other modern RTOSs are either too large in size (hundreds of KBytes or more) or they do not offer several popular OS features like memory protection and threads in an attempt to reduce size and increase speed. Our goal in designing EMERALDS was to develop an RTOS which was not only predictable but also small and efficient, without cutting back on standard OS services relevant to embedded systems. To achieve this goal, we made use of several features of embedded systems which allowed us to increase the efficiency of system calls and keep the size of EMERALDS to just 13 KBytes (uniprocessor version). Another objective in the design of EMERALDS was to make it flexible and easy to use. This is why we chose a microkernel architecture and provided features such as user-level servers for communication protocols and device drivers.

The next step in the development of EMERALDS is networking. We plan to use CAN [12] for this purpose, which is a popular network for real-time control applications. We also plan to add features like kernel-supported deadline-driven scheduling and clock synchronization to EMERALDS.

References

[1] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," *Procedings of the IEEE*, vol. 82, no. 1, pp. 6–24, January 1994.

[2] L. M. Thompson, "Using pSOS+ for embedded real-time computing," in *COMPCON*, pp. 282–288, 1990.

[3] D. Hildebrand, "An architectural overview of QNX," in *Proc. Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[4] *VxWorks Programmer's Guide, 5.1*, Wind River Systems, 1993.

[5] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a predictable real-time system," in *Proc. USENIX Mach Workshop*, October 1990.

[6] K. G. Shin, D. D. Kandlur, D. Kiskis, P. Dodd, H. Rosenberg, and A. Indiresan, "A distributed real-time operating system," *IEEE Software*, pp. 58–68, September 1992.

[7] J. Stankovic and K. Ramamritham, "The Spring Kernel: a new paradigm for real-time operating systems," *ACM Operating Systems Review*, vol. 23, no. 3, pp. 54–71, July 1989.

[8] P. Gopinath and K. Schwan, "CHAOS: Why one cannot have only an operating system for real-time applications," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 3, pp. 106–125, February 1989.

[9] W. M. Gentleman, "Realtime applications: Multiprocessors in Harmony," in *Proc. BUSCON/88 East*, pp. 269–278, October 1988.

[10] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[11] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, December 1982.

[12] *Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication. ISO 11898*, 1st edition, 1993.

[13] H. Kopetz and G. Grunsteidl, "TTP – a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[14] *Electrical Equipment of Industrial Machines — Serial Data Link for Real-Time Communication between Controls and Drives*, International Electrotechnical Commission, 1994. Revision 8.

[15] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 460–467, June 1992.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[17] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 3, pp. 1175–1198, 1990.