99PC332

EMERALDS-OSEK: A Small Real-Time Operating System for Automotive Control and Monitoring

Khawar M. Zuberi, Padmanabhan Pillai, Kang G. Shin
University of Michigan

Takaaki Imai, Wataru Nagaura, Shoji Suzuki
Hitachi, Ltd.

ABSTRACT

Increasingly, microcontrollers are being used in automotive systems to handle sophisticated control and monitoring activities. As applications become more sophisticated, their design and development becomes complex, necessitating the use of an operating system to manage the complexity and provide an abstraction for improving portability of code. This paper presents EMERALDS-OSEK, an operating system we have designed and implemented based on OSEK/VDX, an open industry standard. We present some of the features and optimizations that make EMERALDS-OSEK appropriate for small, low-cost microcontrollers typically found in automotive applications. We also present measurements of operating system performance. We find EMERALDS-OSEK to be efficient, both in terms of processing overheads and memory usage. However, we also find some parts of the OSEK standard that may be improved, and present our ideas for such improvements.

INTRODUCTION

Contemporary automobiles use a vast array of modern technologies to improve performance and reliability, and reduce costs. In particular, the use of embedded microprocessors has allowed for the cost-effective replacement of many antiquated control circuits, use of sophisticated and adaptive control algorithms, and greatly improved monitoring and diagnostics.

The general-purpose microprocessors used in personal computers today are far too expensive for use in automotive applications. The typical microcontroller used in embedded automotive applications must cost just a few dollars and is limited to very small, low-speed CPUs (few megahertz) with minimal code memory (tens of kilobytes of ROM), and very little RAM (few hundred to few thousand bytes), all on a single chip. Increasing speed or memory will require upgrading to a more powerful processor, or increasing component count, resulting in cost increases. With the volumes dealt with in the automotive industry, an increase in component cost of a few dollars translates into millions of dollars added to the bottom line, so the controllers are cost-limited to the very low end of the processor spectrum.

Despite the relatively low computing power of the microcontrollers, the controllers are being called upon to perform increasingly complex tasks. The increasing sophistication and shear quantity of tasks greatly increases the complexity of the system and of system development. To manage this complexity, it is becoming more and more desirable to use operating systems in embedded automotive microcontrollers. An operating system will perform task management, abstracting away task switching and synchronization from the application code and simplifying application development and verification. However, such an operating system must be very efficient, both in terms of processing overheads and RAM usage, since upgrading to more powerful controllers is not a viable option.

Open industry standards for software allow multiple parties to develop applications and subsystems that can work together and easily be integrated. With this and portability as the primary goals, a consortium led by automotive and microcontroller corporations has recently developed the OSEK/VDX (Open systems and corresponding interfaces for automotive electronics / Vehicle Distributed eXecutive) standard, which includes specifications for an embedded operating system, communication subsystem, and an embedded network management system. Based upon our prior work on embedded operating systems [4] and the OSEK operating system specification [2], we have developed a small, efficient operating system called EMERALDS-OSEK, and implemented it on the Hitachi SH-2 microprocessor. We present an overview of EMERALDS-OSEK in the next section, followed by some of the optimizations we have incorporated. Based on our implementation experiences, we then describe how the OSEK standard can be improved. We present some measurements of operating system performance, and introduce a collision avoidance testbed application [3] being developed by Hitachi on EMERALDS-OSEK for further evaluation. We draw some conclusions and state future work.

OVERVIEW OF EMERALDS-OSEK

EMERALDS-OSEK is a small embedded operating system designed to meet the OSEK/VDX 2.0 specifications [2]. It has been developed at the University of Michigan (UM) Real-Time Computing Laboratory (RTCL) for the Hitachi SH-2 microprocessor. It incorporates the major features of OSs, namely task management, scheduling, resource management, and interrupt handling. As it is designed for embedded systems that are typically running on low processing power, low-capacity systems with well-defined applications, it does not have provisions for memory management or dynamic modification of the task set.

TASK MANAGEMENT AND SCHEDULING

Task management is fairly limited in OSEK, since the task set is statically defined for the system at the time of system generation. The number of tasks in the system remains constant, removing the need for dynamic creation and deletion of tasks. A basic OSEK task will switch between active states (either READY to run, or currently RUNNING) and an inactive, or SUSPENDED state. There is also a notion of an extended task, which can also enter a WAITING state, and pause execution until some event occurs (see below). Multiple concurrent invocations of a task are not allowed, since this would require dynamically changing the number of tasks. Instead, if an activation call is made for a task that is already active, the activation request will be queued until the current invocation terminates (returns to the SUSPENDED state).

Tasks that are ready to run are given running time on the processor by the scheduler. The scheduling policy uses a static priority scheme to conform to the OSEK specification. Higher priority tasks, as specified at system generation, are processed before any lower priority tasks. Tasks of identical priority are served on a first-come-first-serve (FCFS) basis. Additionally, tasks may be preemptive or non-preemptive. In a non-preemptive task, once the task begins running, rescheduling occurs at only a few explicit points, and is therefore predictable to the application programmer. In a preemptively-scheduled task, however, rescheduling may take place at any time due to interrupts that activate higher priority tasks. Both types have their uses, so this flexibility can be exploited by the application designer.

SYNCHRONIZATION

Resources in EMERALDS-OSEK are semaphores used as mutual exclusion locks to protect critical data (resources) shared between tasks. Only one task can hold a resource at a given time. One potential problem that occurs when resources are used is the *priority inversion* problem. This may occur when a high priority task tries to obtain a mutually exclusive resource being held by a low priority task, so the high priority task must block until the low priority task releases the resource. If there are a large number of medium priority tasks, the low priority task will not get to run, and the high priority task will remain blocked. Priority inversion has occurred, since any medium priority task can run, while the high priority task cannot. To avoid this, OSEK specifies the use of the *priority ceiling* protocol. When a task obtains a resource, its priority is temporarily bumped up to that of the highest priority task that will ever use the resource, until it releases the resource. This results in two benefits. First, any task holding a resource will have priority at least as great as any other task that may request the resource, so these other tasks will not run until after the resource is released. So when a task tries to obtain a resource, no task can possibly be holding that resource, and hence no blocking can occur when obtaining resources. Secondly, the protocol is inherently deadlock-free. A *deadlock* occurs when two tasks that are holding resources block trying to access those held by the other, and thus end up blocking each other forever. Since no blocking can occur while getting a resource, deadlocks are inherently avoided.

OSEK additionally provides a mechanism of synchronization through *events*. An *extended* task can be signaled by setting an event for that task. It can enter a waiting state and then stay blocked until one of the specified events occur. Events can therefore be used to synchronize multiple tasks or as a very crude method of interprocess communication. Although only extended tasks may wait on an event, any task or interrupt may set them. In order to comply with the tenet of no blocking when obtaining a resource, tasks are prevented from entering the waiting state while holding resources.

ALARMS

Embedded systems typically require some time-based task activation. To facilitate this, OSEK incorporates alarms. An alarm is tied to the system clock (or to any counter), and is triggered when the count reaches the alarm value. When triggered an alarm will either activate a task, or signal that task with an event. The alarms are defined statically at system generation, but the time at which they are triggered is set dynamically to either relative or absolute values. Alarms may also be set to trigger cyclically, and may be used to activate periodic tasks.

INTERRUPTS

Interrupt handling is an important part of any operating system, but takes particular importance in many embedded systems because of their need to quickly react to real-world inputs that often drive interrupts. In EMERALDS-OSEK, interrupt handlers written by the application developer are linked into the system with wrappers that preserve task and system state. Interrupts are allowed to call a subset of the system calls, including those for task activation, event signaling, and setting alarms. Therefore, interrupts may activate a higher priority task, causing a reschedule to take place. There are system calls to enable and disable individual and groups of interrupts, and to inquire about which interrupts are enabled. Although not required by the OSEK standard, EMERALDS-OSEK is a reentrant kernel, so tasks that are executing kernel code, such as while executing a system call, may be interrupted and swapped off of the processor, if needed, without having to wait until the kernel code is exited. This may help improve the average latencies experienced in starting a high priority task activated by an interrupt.

CONFORMANCE CLASSES

The OSEK specification also provides an organized mechanism for implementing a subset of the full standard. There are four partial standards, called conformance classes, specified: BCC1, BCC2, ECC1, and ECC2. BCC1 and BCC2 classes are limited to only basic tasks and do not have support for events, while ECC1 and ECC2 do support extended tasks and events. BCC1 and ECC1 are limited to having only one task per priority level, while BCC2 and ECC2 can support multiple tasks of the same priority. We have developed two versions of EMERALDS-OSEK, one that conforms to class ECC2 and therefore implements the full OSEK/VDX standard, and a more optimized version that conforms to ECC1 class.

SYSTEM GENERATION

In the OSEK specification, much of the system is statically defined at system generation. EMERALDS-OSEK uses a system description file that specifies all of the parameters of the application. Each task is described by its task name, whether it is a basic or an extended task, its scheduling priority, resources it uses, and whether it should be preemptively scheduled. Also, the amount of stack space needed by the task to store its local variables and the processor state when the task is swapped off is specified. All resources, counters, alarms, and interrupt handlers are also declared. This description file is processed by a system generation script that produces a few C language and assembly code files that are compiled with the application code and linked with kernel code (in the form of a library) to produce the runtime object file to be executed on the embedded processor. The generated files perform operating system initialization and also provide headers to map symbolic names to the internal operating system values, simplifying the job of the application programmer.

OPTIMIZATIONS

Although the OSEK/VDX standard specifies the operating system in great detail, there is still room for innovation and optimizations. In designing EMERALDS-OSEK, we have incorporated several optimizations that reduce the memory requirements of the OS, since available RAM is often the most restrictive constraint in a low-cost embedded microcontroller.

STACK OPTIMIZATION FOR BASIC TASKS

Basic tasks differ from extended tasks in that they cannot make blocking system calls, i.e., they cannot call `WaitEvent`. EMERALDS-OSEK uses this fact to optimize RAM usage for basic tasks in two ways:

1. If a basic task is also non-preemptive, then it is guaranteed that once this task begins execution, it will run to completion. No other task can preempt this task and the task will not block itself. Therefore, there is no need to retain the context of this task (especially the stack) between executions. Hence, EMERALDS-OSEK keeps **just one stack for all the non-preemptive basic tasks in the system**. The size of this stack is the maximum of the sizes of individual non-preemptive basic task stacks that the user specifies at system generation time. This can lead to significant savings in RAM.
2. For basic tasks sharing the same priority level, another stack optimization is possible. Since two basic tasks with the same priority cannot preempt each other, and since they will not block themselves waiting for events, basic tasks having the same priority share the same stack in EMERALDS-OSEK. The size of this stack is the maximum of the sizes of the stacks of basic tasks of the same priority as specified by the user at system generation time.

OPTIMIZATIONS FOR *x*CC1 CONFORMANCE CLASS TASKS

BCC1 and ECC1 conformance classes (together called *x*CC1) allow only one task per priority. This means that if there is only one task per priority level in the system, the OS can use a simple scheduler, leading to low run-time overhead.

The primary scheduler data structure used by the ECC1 version of EMERALDS-OSEK is just an array of tasks indexed by their priority. The scheduler simply selects the highest-priority (i.e., highest index) ready task to execute. EMERALDS-OSEK requires that tasks be assigned priorities from 1 to N, where N is the number of distinct priority levels. For the *x*CC1 conformance classes, *N* is the number of tasks in the system, thus allowing a simple array structure to be used for scheduling.

BCC2 and ECC2 conformance classes (together called *x*CC2) allow multiple tasks per priority. This means that if there is even a single pair of tasks sharing a priority level in the system, the OS must use a more complex (and higher overhead) data structure to keep track of tasks. The resulting run-time overhead is greater than the case when the system is limited to *x*CC1. EMERALDS-OSEK uses an array of lists for this situation. The index into the array is the priority and each item in the array is a linked list of tasks. The scheduler must first index into the array, and then locate the first task in the queue, requiring one more operation and slightly more RAM than the more restricted ECC1 conformance class version of EMERALDS-OSEK.

INTERRUPT HANDLING OPTIMIZATIONS

Some microprocessors include hardware support for an *interrupt stack*. Upon interrupt, the processor switches from the previously-active stack to this stack so it can be used by the interrupt service routines (ISRs). Without the interrupt stack, the ISRs would use the stack of whatever task was running at the time the interrupt occurred. This means that the stack for every task will have to be *b* bytes larger than the stack needs for the task, where *b* is the sum of the stack requirements for all ISRs which can preempt each other (to handle nested interrupts). So, if there are *N* tasks in the system, the total stack space reserved for ISRs will be *Nb* bytes which is *(N-1)b* bytes more than what's needed if an interrupt stack is available.

Unfortunately, the SH-2 microprocessor does not have hardware support for an interrupt stack. At the same time, reserving *(N-1)b* bytes extra for ISRs is simply unacceptable in small-memory embedded systems. An ISR that uses system calls (as is allowed in OSEK) needs to save on the stack many processor registers, including call return, status, and temporary registers, and requires at least 44 bytes on the SH-2. If there are 3 levels of interrupts (i.e., interrupts can be nested 3-deep) and the system contains 10 tasks, then (assuming no stack sharing among the tasks) 1188 bytes will be wasted, which is unacceptable in embedded systems with just a few kilobytes of RAM.

Our solution to this problem is to emulate an interrupt stack in software. When interrupts occur, EMERALDS-OSEK --- before executing the user-supplied ISR --- saves the stack pointer and then sets it to point to an interrupt stack. If another ISR preempts the first ISR, the stack is not switched, thus accurately emulating the behavior of a hardware interrupt stack. Finally, when the first ISR exits, the stack pointer is restored to its original value pointing to the stack for the preempted task.

POSSIBLE IMPROVEMENTS IN OSEK STANDARD

The OSEK/VDX standard was developed with automotive applications in mind, and as a result, it is well suited for small automotive microcontrollers. However, through our experiences in developing EMERALDS-OSEK, we have found a few places that improvements are possible in the standard.

EVENTS

In OSEK, events are a means for tasks to synchronize with each other by sending signals to each other. OSEK allows a task to block on multiple events. The task calls the `WaitEvent` system call and passes to it an *event mask* which is a bit mask indicating which events the task wishes to wait for. When any one of these events is signaled, the task is unblocked. However, the user has to make another system call `GetEvent` to find out which event was signaled. The OSEK specification can be improved by allowing the `WaitEvent` system call to return the event mask upon completion. This is trivial to implement as part of the `WaitEvent` call, will incur minimal extra overhead (equal to one memory copy operation), and will save the user from having to make an extra system call.

SYSTEM CLOCK

The notion of time is of central importance in real-time systems. Applications often need to take actions based on the current time. OSEK allows this in one way through the alarm mechanism. Applications can set an alarm to expire (and take certain actions) after a user-specified number of clock ticks. OSEK also has the notion of *counters* which are incremented every time certain special events occur (such as a hardware clock timer interrupt) and OSEK mandates that at least one counter (the system clock) must exist. However, OSEK deliberately does not provide a standardized API for counters. This makes sense because implementation of counters is hardware-dependent. But the downside is that the interface for accessing the system clock is not fixed. Considering the importance of time in real-time systems and the need to access the current time value, the OSEK specification can be improved by adding a standardized system call to read the system clock counter. Internally, this system call will use the hardware-specific counter interface provided by the OS, but externally, it will export a simple, standardized interface to help in portability.

TASK SCHEDULING

The OSEK specification requires that tasks of the same priority be executed in the order of their activation. Moreover, once a blocked task gets unblocked, it must be treated as the newest task among all the tasks of that priority.

In our opinion, an OS specification should not contain such detailed requirements of scheduler behavior because on one hand, it prevents any innovation or optimization, and on the other hand, it really does not help in improving portability. In defining the OS API for scheduling, it is sufficient to require fixed-priority scheduling with statically-defined task priorities to ensure portable applications. But OSEK goes beyond that and literally fixes the internal kernel data structures. Relaxing these scheduler requirements can improve the OSEK specification by allowing kernel designers to optimize internal kernel mechanisms in various ways without affecting the API as seen by the applications.

INTERRUPT CONTROL

Often it is necessary for an application to disable some or all interrupts around certain critical portions of code. To do this, OSEK specifies the `DisableInterrupt` and `EnableInterrupt` system calls. Both take a single parameter indicating a single or multiple interrupts to be affected. In order to be completely flexible, and allow an arbitrary subset of interrupts to be specified, we use a bit-mask as the parameter to select from the Hitachi SH-2's 32 independently maskable interrupt sets. Although this is a very general mechanism that can functionally satisfy any needs, scanning a bit field is relatively slow and can cause significant performance degradation in applications that frequently disable and enable interrupts.

It is our opinion that a less flexible, very fast mechanism is suitable for many common uses of interrupt control. A very common use of interrupt control is to disable all interrupts, perform some critical function, and then restore the interrupts to their prior settings. This currently requires three system calls: `GetInterruptDescriptor` to obtain the current interrupt settings, followed by `DisableInterrupt` and `EnableInterrupt` around the critical code. What is really needed is a pair of system calls: `EnterCritical` and `LeaveCritical`. Since many microprocessors (including the SH-2) provide a mechanism for disabling all interrupt processing through a control register without changing individual interrupts, these system calls can be implemented very efficiently.

Because we feel this situation will occur very frequently, the addition of the less flexible, but more efficient interrupt control mechanism is justified.

EVALUATION AND MEASUREMENTS

We have implemented EMERALDS-OSEK on the Hitachi SH-2 microprocessor, and measured some of the primitive operating system performance characteristics. In addition, Hitachi Research Lab has been developing a collision avoidance application that incorporates communication over a Controller Area Network, and are evaluating the use of EMERALDS-OSEK as the operating system for such a system [3].

TIMING MEASUREMENTS

One of the significant overheads introduced by an OS is the task switching overhead. We measure the task switching times for both the full version (ECC2 conformance class) and ECC1 version of EMERALDS-OSEK running on an 8 MHz Hitachi SH-2 processor. We look at several scenarios that result in task switching:

1. Task Activation: When a running, preemptive task calls `ActivateTask` on a task with higher priority, a task switch occurs and the higher priority task is allowed to run. From the start of the system call to the beginning of the activated task on average requires 27.4 µs for ECC2, and 23.8 µs for ECC1.
2. Task Termination: When a running task terminates (returns to SUSPENDED state), the scheduler is invoked to determine the next task to run. The amount of time required depends on the number of priority levels there are, and how far down the list is the next task to activate. The worst-case switching times during task termination are summarized in Figure 1. The ECC1 version is more efficient, but is limited to only one task per priority level.
3. Wake on Event: A high priority extended task that is waiting for an event will be resumed immediately when a lower priority task sets the event. The task switch here requires 25 µs for ECC2 and 21.1 µs for the ECC1 version of EMERALDS-OSEK.

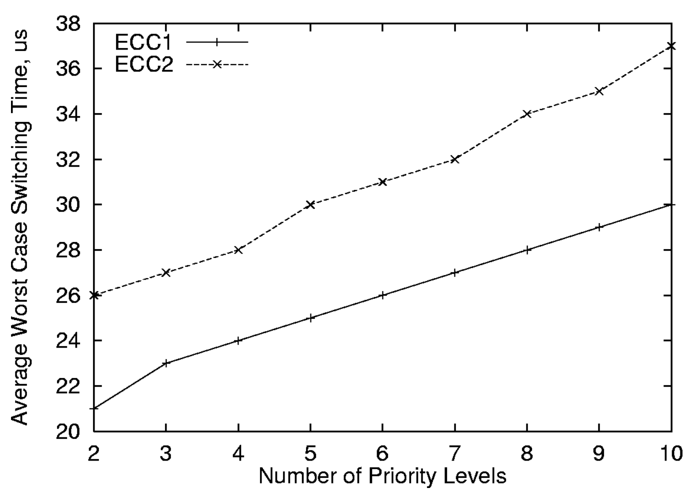

Figure 1: Average worst-case task switch timing during task termination.

In general, we have found that task switches that result in running a task of higher priority will need 20 to 25 µs with ECC1 and 24 to 30 μs with ECC2. Switching to a lower priority task follows a pattern very similar to that of task termination, with worst-case switching times increasing linearly with the number of priority levels. Because of the internal data structure optimizations incorporated into the restricted ECC1 conformance class version, it consistently provides lower switching times than the full, ECC2 implementation. However, since the *x*CC1 conformance classes are restricted to only one task per priority level, switching time performance will quickly degrade as the number of tasks increases. In contrast, the full ECC2 implementation of EMERALDS-OSEK will scale switching time much better with the number of tasks since multiple tasks may share a priority level.

Because obtaining a resource can never result in blocking (see above), the `GetResource` call never results in a task switch, and executes in just 5 μs, including bumping up the task priority according to the priority ceiling protocol. `ReleaseResource`, however, may result in a task switch, so in the worst case requires 24.5 μs and 29 μs respectively for the ECC1 and ECC2 versions of EMERALDS-OSEK.

As mentioned earlier, the OSEK interrupt control system calls have been implemented with the flexibility to affect an arbitrary subset of interrupts in a single call. We have hand-optimized the code to operate on 4 bits of the bit-field parameter at a time, greatly improving performance. Even though they may need to update 32 independent interrupt masks, calls to `DisableInterrupt` and `EnableInterrupt` require 11.6 μs and 14.3 μs, respectively, in the worst case. As we suggest in Section 4.4, if we were to implement the restricted, non-OSEK-compliant functions, `EnterCritical` and `ExitCritical`, they would require 1 μs and less than 1 μs respectively. On a related topic, the built-in clock interrupt handler, which increments the system clock and checks if any alarms have expired, incurs an approximately 20 μs overhead per clock tick.

Projecting from these measured times, we can expect a system that requires 1000 task switches and 100 clock ticks per second will have a 2.5 to 3% operating system overhead, primarily consumed by task switching.

MEMORY REQUIREMENTS

EMERALDS-OSEK is designed to be memory-efficient. The ROM image of the OS requires less than 5.5 KB, including the SH-2 processor's 1024 byte exception vector table. This small size is partly due to the carefully limited feature set of the OSEK standard and partly due to the processor instruction set architecture. Although it is a 32-bit reduced instruction set machine, the Hitachi SH-2 uses 16-bit long instructions that allow for good code density.

The RAM requirements of EMERALDS-OSEK are application dependent. The basic internal variables require only 96 bytes, but each component of the application will require additional OS data structures. Requirements for various components are summarized in Table 1. Most memory is consumed by the stacks needed for the tasks. The minimum size stack must be able to hold the processor context (all processor registers and some OS state) in addition to interrupt requirements and the task's own requirements. In EMERALDS-OSEK, 88 bytes are needed for context, consisting of 21 processor registers and 1 control word. By making use of the emulated interrupt stack, the per-task stack overhead of interrupts can be reduced. However, since the interrupt stack feature is emulated in software, rather than integrated into the processor, the overhead cannot be completely eliminated and in the worst case, 16 bytes are needed per interrupt nesting level used.

As discussed above, the optimizations incorporated into EMERALDS-OSEK can conserve memory usage by sharing stacks among tasks. To maximize the sharing of tasks, we suggest the following guidelines in application design:

1. Avoid using extended tasks: Since extended task can block waiting on an event, during which the task context must be preserved while other tasks execute, they cannot share stacks with other tasks.
2. Use non-preemptive basic tasks as much as possible: All non-preemptive basic tasks can share a single stack. In addition, the rescheduling points are all explicit, and system behavior is under better control of the application programmer.
3. Allocate as many tasks to as few priority levels as possible: This will improve stack sharing among preemptive basic tasks. Additionally, fewer priority levels results in better worst case task switching times as we have seen earlier.

Structure	RAM required (bytes)
task	36
alarm	18
counter	8
resource	2

Table 1: OS memory requirements for some application components

HITACHI COLLISION AVOIDANCE TESTBED

Hitachi Research Labs is currently developing a prototype for an adaptive cruise control system [3], in which EMERALDS-OSEK will be used and evaluated. The system is distributed on multiple controllers connected through a Controller Area Network (CAN). The adaptive cruise control application will reside on a controller that will receive periodic inputs from software on a collision avoidance radar unit. It will compute driving or breaking force needed and communicate this with the powertrain control software running on a networked controller. The complexity of this system with multiple interacting components over a network makes it desirable to use an operating system such as EMERALDS-OSEK to manage task complexity and ease the development process. In addition, an open standard communication middleware has been developed at Hitachi based on the OSEK Communication specification [1], and will run on top of EMERALDS-OSEK.

At present, only very preliminary data is available regarding the performance of the communication primitives. A 20 MHz SH-2 drives a relatively fast network (1 Mbit/sec CAN) with a small transmission unit (8-byte payload per packet) in the Hitachi testbed [3]. With such small packets on a network that is fast relative to the processor, we expect the data throughput to be primarily restricted by processing latencies. Indeed, the controller was capable of sustaining 394 Kbits/sec with only the communications primitives, as compared to a theoretical maximum of 575 Kbits/sec. Furthermore, throughput drops drastically with any additional processing overhead, so with the addition of the operating system, the peak throughput came down to 285 Kbits/sec. Of course, these peak rate measures do not take into account application code processing overheads. In a real application, the rate of communication will be limited by the processing requirements of the application, hence the transmission bandwidth drop mentioned above will not be a problem.

In the future, once the actual application code is written, we hope to gain useful measurements of operating system overheads in a real system, as well as profiling measures for the types OS services most used and how improvements can be made.

CONCLUSION

We have developed EMERALDS-OSEK, an operating system based on the OSEK/VDX standard on the Hitachi SH-2 microprocessor. We have found the OSEK standard to be well suited for the low computing, power, small memory embedded controllers used in automotive applications. However, we have also found some parts that can be improved in the standard and have provided some suggestions. EMERALDS-OSEK, with the various optimizations we have incorporated, is a small and efficient operating system, in regard to CPU usage and especially RAM requirements, and is well suited to the highly-constrained world of embedded automotive controllers. The SH-2 is also a good choice in this regard, since it is a relatively powerful 32-bit processor, yet uses 16-bit long instructions to produce good code density. The availability of standard (GNU) compilers and good emulators for this processor also helped development greatly.

In the future, we would like to better tune the operating system for task sets typically found on automotive controllers today. We would also look into creating better system generation tools that can adapt the applications to better use the optimizations within the OS.

ACKNOWLEDGMENTS

We would like to thank Hasdi Hashim, Jonathan Arnold, and Ann Lockwood for their contributions to the coding, debugging, and performance evaluation of EMERALDS-OSEK.

CONTACT

Khawar M. Zuberi, Padmanabhan Pillai, Kang G. Shin
Real-Time Computing Laboratory
Dept. Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
{zuberi, pillai, kgshin}@eecs.umich.edu
734-936-2495 (voice); 734-763-4617 (fax)

Takaaki Imai, Wataru Nagaura, Shoji Suzuki
Hitachi Research Laboratory
1-1 Omika-cho, 7-chome, Hitachi-shi
Ibaraki-ken, 319-12 JAPAN
{timai, nagaura, suzukish}@hrl.hitachi.co.jp

REFERENCES

1. OSEK/VDX Communication Version 2.1 revision 1, OSEK Group, June, 1998.
2. OSEK/VDX Operating System Specification 2.0, OSEK Group, June, 1997.
3. S. Suzuki, W. Nagaura, T. Imai, S. Kuragaki, and T. Yokoyama, "A Distributed Control System Framework for Automotive Powertrain Control with OSEK Standard and CAN network," in Proc. SAE International Congress & Exhibition, March, 1999.
4. K. M. Zuberi and K. G. Shin, "EMERALDS: A Microkernel for Embedded Real-Time Systems," in Proc. RTAS pp. 241-249, June, 1996.