A Portable Real-Time Operating System For Embedded Platforms

By
Mehmet Onur OKYAY

A Dissertation Submitted to the Graduate School in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

Department: Computer Engineering
Major: Computer Software

İzmir Institute of Technology
İzmir, Turkey

July, 2004

We approve the thesis of **Mehmet Onur OKYAY**

Date of Signature

---------- **29.07.2004**
Prof. Dr. Sıtkı AYTAÇ
Supervisor
Department of Computer Engineering

---------- **29.07.2004**
Assoc. Prof. Dr. Ahmet H. KOLTUKSUZ
Department of Computer Engineering

---------- **29.07.2004**
Assist. Prof. Dr. Şevket GÜMÜŞTEKİN
Department of Electronics Engineering

---------- **29.07.2004**
Prof. Dr. Sıtkı AYTAÇ
Head of Department

ABSTRACT

In today's world, from TV sets to washing machines or cars, almost every electronic device is controlled by an embedded system. These systems are handling many tasks simultaneously. By using an operating system, handling of different tasks simultaneously is done in a more standardized fashion.

The purpose of this thesis is to design and write a portable real-time operating system for embedded systems, which can be compiled with any application by using an ANSI C compiler. The main target is to design it as small as possible to fit the smallest microcontrollers. Other targets are high flexibility, optimal modularity, high readability and maintainability of the source code.

ÖZ

Günümüzde televizyonlardan otomobillere kadar neredeyse bütün elektronik cihazlar gömülü sistemlerle kontrol edilmektedir. Bu sistemler bir çok işi eş zamanlı olarak yürütebilmek zorundadırlar. Bir işletim sistemi kullanarak, farklı işlerin bir arada yürütülmesi daha standart hale getirilebilir.

Bu tezin amacı herhangi bir ANSI C derleyicisi ile derlenebilecek, taşınabilir, gerçek zamanlı bir işletim sistemini tasarlamak ve gerçekleştirmektir. Ana hedef işletim sistemini, en küçük mikrodenetleyicilere bile sığabilmesi için olabildiğince küçük tasarlamaktır. Diğer hedefler; yüksek esneklik, optimum modülerlik ve; kaynak kodun yüksek derecede okunabilir, anlaşılabilir ve bakımı yapılabilir olmasıdır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

In today's world, from TV sets to washing machines or cars, almost every electronic device is controlled by an embedded real-time system. These systems are handling many tasks simultaneously. By using an operating system, handling of different tasks simultaneously is done in a more standardized fashion.

Running tasks in parallel and, synchronization and communication between them are common to all systems. There are many methods to achieve these purposes like using an infinite loop that checks the all the tasks continuously. But most of them are not reliable, not flexible and are not used very easily. Operating systems, however, provide with a common interface for the tasks that need to be run in parallel. This way the project source code is more understandable and reliable, and easier to maintain. Using operating systems embedded projects can be divided into independent blocks easier. And this decreases the implementation time and increases development team's efficiency. Because of these, using an operating system in embedded projects is very important.

Real-time systems are systems where correctness depends not only on the correctness of the logical result of the computation, but also on the result delivery time. A real-time system should respond in a timely, predictable way to unpredictable external stimuli arrivals.

Real-time and embedded operating systems are in most respects similar to general purpose operating systems: they provide the interface between application programs and the system hardware, and they rely on basically the same set of programming primitives and concepts. But general purpose operating systems make different trade-offs in *applying* these primitives, because they have different goals.

1.1 Reason and Purpose

There are many commercial and non-commercial operating systems for the embedded systems in the world today. Each has different features, capabilities, advantages and disadvantages. But in principal they all provide almost the same functionality to the user in terms of basic operations.

The main reason that pushes us to implement a new operating system is not to introduce some new features. It is the belief, that there is not enough research in Turkey, in the field of embedded systems and embedded software.

With the increase of production in consumer and industrial electronics products, the embedded software is becoming more important everyday. The Turkish companies like Vestel and Beko are continuously investing more on engineering power in research and development of embedded systems. This trend should also force the Turkish academic people to develop a national intellectual property in the field of embedded systems. This thesis is aimed to be a part of that intellectual property.

The purpose of this thesis is to design and implement a portable real-time operating system for embedded systems, which can be compiled with any application by using an ANSI C compiler. The main target is to design it as small as possible to fit the smallest microcontroller. Other targets are high flexibility, optimal modularity, high readability and maintainability of the source code.

1.2 Brief Information On Sections

Chapter 2 gives the necessary information about the operating systems background. It includes information on the basic concepts, other operating systems and OS standards.

Chapter 3 has all the details of the design and implementation of ARTIOS along with the API details.

Chapter 4 is the conclusion part where the results are explained.

1.3 Source Code Information

The source code of the operating system can be reached at the FTP server of the Computer Engineering Department of Izmir Institute of Technology.

For any questions about the operating system and updated source code, you can reach me from my e-mail address mehmetonurokyay@hotmail.com.

2. Background

This part introduces the concepts and primitives, with which general purpose and real-time and embedded operating systems are built. The text discusses the applicability and appropriateness of all these concepts in real-time and embedded operating systems.

2.1 Real-time and Embedded Operating Systems

This part discusses the basics of operating systems in general, and real-time and embedded operating systems in particular. (This text uses the abbreviations OS, RTOS and EOS, respectively.) This discussion makes clear why standard Linux does not qualify as a real-time OS, or as an embedded OS.

Real-time and embedded operating systems are in most respects similar to general purpose operating systems: they provide the interface between application programs and the system hardware, and they rely on basically the same set of programming primitives and concepts. But general purpose operating systems make different trade-offs in *applying* these primitives, because they have different goals.

2.2 OS Responsibilities

This part discusses the basic responsibilities of the operating system that are relevant for this text: (i) task management and scheduling, (ii) (deferred) interrupt servicing, (iii) inter-process communication and synchronization, and (iv) memory management. General-purpose operating systems also have other responsibilities, which are beyond the scope of a *real-time* operating system: file systems and file management, (graphical) user interaction, communication protocol stacks, disk IO, to name a few.

2.2.1 Task management and scheduling

Task (also "process", or "thread") management is a primary job of the operating system: Tasks must be created and deleted while the system is running; tasks can change their priority levels, their timing constraints, their memory needs; etc. Task

management for an RTOS is more critical than for a general purpose OS: If a real-time task is created, it *has* to get the memory it needs without delay,
and that memory *has* to be locked in main memory in order to avoid access latencies due to swapping; changing run-time priorities influences the run-time behavior of the whole system and hence also the predictability which is very important for an RTOS. So, dynamic process management is a potential headache for an RTOS.

In general, multiple tasks will be active at the same time, and the OS is responsible for sharing the available resources (CPU time, memory, etc.) over the tasks. The CPU is one of the important resources, and deciding how to share the CPU over the tasks is called "scheduling".

The general trade-off made in scheduling algorithms is the *simplicity* (and hence efficiency) of the algorithm, and its *optimality*. Algorithms that are designed to be globally optimal are usually quite complex, and/or require knowledge about a large number of task parameters, that are often not straightforward to find on line (e.g., the duration of the next run of a specific task; the time instants when sleeping tasks will become ready to run; etc.). Real-time and embedded operating systems favor simple scheduling algorithms, because these take a small and deterministic amount of computing time, and require little memory footprint for their code [Arcamano, 2002].

General purpose and real-time operating systems differ considerably in their scheduling algorithms. They use the same basic principles, but apply them differently because they have to satisfy different performance criterions. A general purpose OS aims at maximum *average* throughput, a real-time OS aims at *deterministic* behavior, and an embedded OS wants to keep memory footprint and power consumption low. A large variety of "real-time" scheduling algorithms exists, but some are standard in most real-time operating systems: *static priority scheduling*, *earliest deadline first (EDF)*, and *rate-monotonic scheduling*.

2.2.2 Interrupt servicing

An operating system must not only be able to schedule tasks according to a deterministic algorithm, but it also has to service peripheral hardware, such as timers, motors, sensors, communication devices, disks, etc. All of those can request the attention of the OS *asynchronously*, i.e., at the time that *they* want to use the OS

services, the OS has to make sure it is ready to service the requests. Such a request for attention is often signaled by means of an *interrupt*. There are two kinds of interrupts:

• *Hardware interrupts.* The peripheral device can put a bit on a particular hardware channel that triggers the processor(s) on which the OS runs, to signal that the device needs servicing. The result of this
trigger is that the processor saves its current state, and jumps to an address in its memory space, that has been connected to the hardware interrupt at initialization time.

• *Software interrupts.* Many processors have built-in software instructions with which the effect of a hardware interrupt can be generated in software. The result of a software interrupt is also a triggering of the processor, so that it jumps to a pre-specified address.

The operating system is, in principle, not involved in the execution of the code triggered by the hardware interrupt: this is taken care of by the CPU without software interference. The OS, however, does have influence on (i) connecting a memory address to every interrupt line, and (ii) what has to be done *immediately after* the interrupt has been serviced, i.e., how *"deferred interrupt servicing"* is to be handled. Obviously, real-time operating systems have a specific approach towards working with interrupts, because they are a primary means to guarantee that tasks get serviced deterministically.

2.2.3 Communication and synchronization

A third responsibility of an OS is commonly known under the name of *Inter-Process Communication* (IPC). ("Process" is, in this context, just another name for "task".) The general name IPC collects a large set of programming primitives that the operating system makes available to tasks that need to exchange information with other tasks, or synchronize their actions. Again, an RTOS has to make sure that this communication and synchronization take place in a deterministic way.

Besides communication and synchronization with other tasks that run on the same computer, some tasks also need to talk to other computers, or to peripheral

hardware (such as analog input or output cards). This involves some peripheral hardware, such as a serial line or a network, and special purpose device drivers.

2.2.4 Memory management

A fourth responsibility of the OS is *memory management*: the different tasks in the system all require part of the available memory, often to be placed on specified hardware addresses (for memory-mapped IO). The job of the OS then is (i) to give each task the memory it needs (*memory allocation*), (ii) to map the real memory onto the address ranges used in the different tasks (*memory mapping*), and (iii) to take the appropriate action when a task uses memory that it has not allocated. (Common causes are: unconnected pointers and array indexing beyond the bounds of the array.) This is the so-called *memory protection* feature of the OS. Of course, what exactly the "appropriate action" should depend on the application; often it boils down to the simplest solution: killing the task and notifying the user.

2.3 Trade-offs

This part discusses some of the trade-offs that (both, general purpose, and real-time and embedded) operating system designers commonly make.

• *Kernel space versus user spac*.

Most modern processors allow programs to run in two different hardware protection levels. Linux calls these two levels *kernel space* and *user space*. The latter have more protection against erroneous accesses to physical memory of I/O devices, but access most of the hardware with larger latencies than kernels space tasks. The real-time Linux variants add a third layer, the *real-time space*. This is in fact nothing else but a part of kernel space used, but used in a particular way.

• *Monolithic kernel versus micro-kernel*.

A monolithic kernel has all OS services (including device drivers, network stacks, file systems, etc.) running within the *privileged mode* of the processor. This does not mean that the whole kernel is one single C file. A micro-kernel, on the other hand, uses the privileged mode only for really core services (task management and scheduling, inter-process communication, interrupt handling, and memory

management), and has most of the device drivers and OS services running as “normal” tasks.

The trade-off between both is as follows: a monolithic kernel is easier to make more efficient (because OS services can run completely without switches from privileged to non-privileged mode), but a micro-kernel is more difficult to crash (an error in a device driver that does not run in privileged mode is less likely to cause a system halt than an error occurring in privileged mode). UNIX, Linux and Microsoft NT have monolithic kernels; QNX, FIASCO, VxWorks, and GNU/Hurd have micro-kernels. Linux, as well as some commercial UNIX systems, allow dynamically or statically changing the number of services in the kernel: extra functionality is added by loading a *module*. But the loaded functionality becomes part of the monolithic kernel. A minimal Linux kernel (which includes memory management, task switching and timer services) is some hundreds of kilobytes big. This approaches the footprint for embedded systems. However, more and more embedded systems have footprints of more than a megabyte, because they also require network stacks and various communication functionalities.

• *Preemptive kernel or not.*

Linux was originally a non-preemptive kernel: a kernel space task cannot be interrupted by other kernel space tasks, or by user space tasks. The kernel is “locked” as long as one kernel function is executing. This usage of locks makes the design of the kernel simpler, but introduces non deterministic latencies which are not tolerable in an RTOS.

• *Scalability*.

Finer-grained locking is good for *scalability*, but usually an overhead for single CPU systems. *Solaris* is an example of a very fine-grained and scalable operating system, which performs worse on “low-end” PCs. Scalability is *much less* of an issue in *real-time* applications, because the goals are so different: the desire behind scalable systems is to divide a large work load transparently over a number of available CPUs, while the desire behind real-time systems is have everything controlled in a strictly deterministic way.

• *Memory management versus shared memory*.

Virtual memory and dynamic allocation and de-allocation of memory pages are amongst the most commonly used memory management services of a general purpose operating system. However, this memory management induces overhead, *and* some simpler processors have no support for this memory management. On these processors (which power an enormous number of embedded systems!), all tasks share the same memory space, such that developers must take care of the proper use of that memory. Also some real-time kernels (such as RTLinux) have all their tasks share the same address space (even if the processor supports memory management), because this allows more efficient code.

• *Dedicated versus general*.

For many applications, it is worthwhile not to use a commercially or freely available operating system, but write one that is optimized for the task at hand. Examples are the operating systems for mobile phones, or Personal Digital Assistants. Standard operating systems would be too big, and they don't have the specific signal processing support (speech and handwriting recognition) that is typical for these applications. Some applications even don't need an operating system at all. (For example, a simple vending machine.) The trade-offs here are: cost of development and decreased portability, against cost of smaller and cheaper embedded systems.

• *Operating system versus language runtime*.

Application programs make use of "lower-level" primitives to build their functionality. This functionality can be offered by the operating system (via system calls), or by a programming language (via language primitives and libraries). Languages such as C++, Ada and Java offer lots of functionality this way: memory management, threading, task synchronization, exception handling, etc. This functionality is collected in a so-called *runtime*. The advantages of using a runtime are: its interface is portable over different operating systems, and it offers ready-to-use and/or safe solutions to common problems. The disadvantages are that a runtime is in general "heavy", not deterministic in execution time, and not very configurable. These disadvantages are important in real-time and embedded contexts.

2.4 Time

"time" plays an important role in the design and use of a real-time operating system. This part introduces some relevant terminology and definitions.

2.4.1 Real-Time

There are too many interpretations of the meaning of *real-time.*

One simple definition is: A real-time operating system is able to execute all of its tasks without violating *specified* timing constraints.

Another definition is: Times at which tasks will execute can be *predicted deterministically* on the basis of knowledge about the system's hardware and software. That means, if the hardware *can* do the job, the RTOS software *will* do the job deterministically.

One often makes distinction between "soft real-time" and "hard real-time". "Soft" indicates that not meeting the specified timing constraints is not a disaster, while it *is* a disaster for a hard real-time system.

For example: playing an audio or video file is soft real-time, because few people will notice when a sample comes a fraction of a second too late. Steering a space probe, on the other hand, requires hard real-time, because the rocket moves with a velocity of several kilometers per second such that small delays in the steering signals add up to significant disturbances in the orbit which can cause erroneous atmosphere entry situations. Precision mills and high-accuracy radiation or surgical robots are other examples that require hard real-time: moving the mill or the robot one tenth of a millimeter too far due to timing errors can cause the rejection of produced parts, or the death of patients.

Practically speaking, the distinction between soft and hard real-time is often (implicitly and mistakenly) related to the time scales involved in the system: in this reasoning, soft real-time tasks must typically be scheduled with (coarser than) *milliseconds* accuracy, and hard real-time tasks with *microseconds* accuracy. But this implicit assumption has many exceptions! For example, a one-dollar 4 bit processor controlling a traffic light can be more hard-real-time (in the sense of "deterministic") than a 5000 dollar Athlon-based e-commerce server.

2.4.2 Latency

The *latency* of a task is the difference between the instant of time on which the task should have started (or finished) and the instant of time on which it actually did. (Or, in different contexts, the time between the *generation* of an event, and its *perception*.) Latencies are due to several factors: (i) the timing properties of processor, bus, memory (on-chip cache, off-chip RAM and ROM) and peripheral devices, (ii) the scheduling properties of the OS, (iii) the *preemptiveness* of its kernel, (iv) the load on the system (i.e., the number of tasks that want to be scheduled concurrently), and (v) the *context switch* time. This latter is the time the processor needs to save the data of the currently running task (e.g., registers, stack, and instruction pointer), and to replace it with the local data of the newly scheduled task.

Few of these factors are constant over time, and the statistical distribution of the latencies in the subsequent schedulings of tasks is called the *jitter*. This is a far from exhaustive list of kernel activities that introduce *indeterminism* into the timing behavior of a (general purpose) operating system:

• *Accessing the hard disk.* Because the alignment of sectors, and the distance between the tracks needed by a given task are variable, that task cannot be sure about how long it will take to access the data it needs. In addition, hard disks are mechanical devices, whose time scales are much longer than purely electronic devices (such as RAM memory); and accesses to the hard disk are *buffered* in order to reduce *average* disk access time.

• *Accessing a network.* Especially with the TCP/IP protocol, that re-sends packets in case of transmission errors.

• *Low-resolution timing.*

• Another delay related to time keeping is the fact that *programming the timer chip* often generates unpredictable delays. This delay is of the order of microseconds, so only important for really high-accuracy timing.

• *Non-real-time device drivers.* Device drivers are often sloppy about their time budget: they use busy waiting or roughly estimated sleeping periods, instead of timer interrupts, or lock resources longer than strictly necessary, or run in user space with the corresponding timing unpredictability.

• *Memory allocation and management.* After a task has asked for more memory (e.g., through a malloc function call), the time that the memory allocation task needs to fulfill the request is unpredictable. Especially when the allocated memory has become strongly fragmented and no contiguous block of memory can be allocated. Moreover, a general purpose operating system swaps code and data out of the physical memory when the total memory requirement of all tasks is larger than the available physical memory.

The exact *magnitude* of all the above-mentioned time delays changes very strongly between different hardware. Hence, it is not just the operating system software that makes the difference. For some applications, the context switch time is most important (e.g., for sampling audio signals at 44kHz), while other applications require high computational performance, at lower scheduling frequencies (e.g., robot motion control at 1kHz). But again, some tasks, such as speech processing, require both.

2.4.3 Timing constraints

Different applications have different timing constraints, which, ideally, the RTOS should be able to satisfy. However, there are still no general and guaranteed scheduler algorithms that are able to satisfy all the following classes of time constraints:

• *Deadline*: a task has to be completed before a given instant in time, but when exactly the task is performed during the time interval between now and the deadline is not important for the quality of the final result. For example: the processor must fill the buffer of a sound card before that buffer empties; the voltage on an output port must reach a given level before another peripheral device comes and reads that value.

• *Zero execution time*: the task must be performed in a time period that is zero in the ideal case. For example: digital control theory assumes that taking a measurement,

calculating the control action, and sending it out to a peripheral device all take place instantaneously.

• *Quality of Service* (QoS): the task must get a fixed amount of "service" per time unit. ("Service" often means "CPU time", but could also be "memory pages", "network bandwidth" or "disk access bandwidth".) This is important for applications such as multimedia (in order to read or write streaming audio or video data to the multimedia devices), or network servers (both in order to guarantee a minimum service as in order to avoid "denial of service" attacks).

The QoS is often specified by means of a small number of parameters: "s" seconds of service in each time frame of "t" seconds. A specification of 5 micro-seconds per 20 micro-seconds is a much more real-time QoS than a specification of 5 seconds per 20 seconds, although, on the average, both result in the same amount of time allotted to the task.

The major problem is that the scheduler needs complete knowledge about how long each task is going to take in the near future, and when it will become ready to run. This information is practically impossible to get, and even when it is available, calculation of the optimal scheduling plan is a search problem with high complexity, and hence high cost in time.

Different tasks compete for the same resources: processors, network, memory, disks . . . Much more than in the general purpose OS case, programmers of real-time systems have to take into account *worst-case* scenarios: if various tasks *could* be needing a service, then sooner or later they *will* want it at the same time.

2.5 Embedded OS

The concepts introduced in the previous sections apply of course also to embedded operating systems ("EOS"). Embedded operating systems, however, have some features that distinguish them from real-time and general purpose operating systems. But the definition of an "embedded operating system" is probably even more ambiguous than that of an RTOS, and they come in many different forms. But you'll

recognize one when you see one, although the boundary between general purpose operating systems and embedded operating systems is not sharp, and is even becoming more blurred all the time.

Embedded systems are being installed in tremendous quantities (an order of magnitude more than desktop PCs!): they control lots of functions in modern cars; they show up in household appliances and toys; they control vital medical instrumentation; they make remote controls and GPS (Global Position Systems) work; they make your portable phones work; etc.

The simplest classification between different kinds of embedded operating systems is as follows:

• *High-end embedded systems.* These systems are often down-sized derivatives of an existing general purpose OS, but with much of the "ballast" removed. Linux has given rise to a large set of such derivatives, because of its highly modular structure and the availability of source code. Examples are: routers, switches, personal digital assistants, set top boxes.

• *Deeply embedded OS.* These OSs must be really *very* small, and need only a handful of basic functions. Therefore, they are mostly designed from the ground up for a particular application. Two typical functions deeply embedded systems (used to) lack are high-performance graphical user interfacing or network communication. Examples are: automotive controls, digital cameras, portable phones. But also these systems get more graphics and networking capabilities.

The most important features that make an OS into an embedded OS are:

• Small *footprint*. Designers are continuously trying to put more computing power in smaller housings, using cheaper CPUs, with on-board digital and/or analog IO; and they want to integrate these CPUs in all kinds of small objects. A small embedded OS also often uses only a couple of kilobytes of RAM and ROM memory.

• The embedded system should run for years without manual intervention. This means that the hardware *and* the software should never fail. Hence, the system should preferably have no mechanical parts, such as floppy drives or hard disks. Not only

because mechanical parts are more sensitive to failures, but they also take up more space, need more energy, take longer to communicate with, and have more complex drivers (e.g., due to motion control of the mechanical parts).

• Many embedded systems have to control devices that can be dangerous if they don't work exactly as designed. Therefore, the status of these devices has to be checked regularly. The embedded computer system itself, however, is one of these critical devices, and has to be checked too! Hence, one often sees *hardware watchdogs* included in embedded systems. These watchdogs are usually retriggerable monostable timers attached to the processor's reset input. The operating system checks within specified intervals whether everything is working as desired, for example by examining the contents of status registers. It then resets the watchdog. So, if the OS does not succeed in resetting the timer, that means that the system is not functioning properly and the timer goes off, forcing the processor to reset. If something went wrong but the OS is still working (e.g., a memory protection error in one of the tasks) the OS can activate a *software watchdog*, which is nothing else but an interrupt that schedules a service routine to handle the error. One important job of the software watchdog could be to generate a *core dump*, to be used for analysis of what situations led to the crash.

• A long autonomy also implies using as little power as possible: embedded systems often have to live a long time on batteries (e.g., mobile phones), or are part of a larger system with very limited power resources (e.g., satellites).

• If the system does fail despite its designed robustness, there is usually no user around to take the appropriate actions. Hence, the system itself should reboot autonomously, in a "safe" state, and "instantly" if it is supposed to control other critical devices. Compare this to the booting of your desktop computer, which needs a minute or more before it can be used, and always comes up in the same default state. . .

• It should be as cheap as possible. Embedded systems are often produced in quantities of several thousands or even millions. Decreasing the unit price even a little bit, results in enormous savings.

• Some embedded systems are not physically reachable anymore after they have been started (e.g., launched satellites) in order to add software updates. However, more and more of them can still be accessed remotely. Therefore, they should support *dynamic linking*: object code that did not exist at the time of start is uploaded to the system, and linked in the running OS without stopping it.

Some applications require all features of embedded *and* real-time operating systems. The best known examples are mobile phones and (speech-operated) handheld computers ("PDA"s): they must be small, consume little power, and yet be able to execute advanced signal processing algorithms, while taking up as little space as possible.

The above-mentioned arguments led embedded OS developers to design systems with the absolute minimum of software and hardware. Hence, embedded systems often come without a memory management unit (MMU), multi-tasking, a networking "stack", or file systems. The extreme is one single monolithic program on the bare processor, thus completely eliminating the need for any operating system at all.

Taking out more and more features of a general purpose operating system makes its footprint smaller and its predictability higher. On the other hand, adding more features to an EOS makes it look like a general purpose OS. Most current RTOS and EOS operating systems are expanding their ranges of application, and cover more of the full "feature spectrum."

2.6 Operating system standards

Real-time and embedded systems are not user products by themselves, but serve as platforms on which applications are built. As for any other software platform, the availability of standards facilitates the job of programmers enormously, because it makes it easier, cheaper and faster to develop new applications, and to port an existing application to new hardware. In the world of real-time and embedded systems, standardization is not a big issue. Because many projects in this area have unique requirements, need unique extensions to already existing products, don't need frequent updates by different people, and are seldom visible to end-users. All these "features" do

not really help in forcing developers to use standards. . . (They do like standard *tools* though, which is one reason for the popularity of the Free Software GNU tools.)

This part lists some standardization efforts that exist in the real-time and embedded world.

2.6.1 POSIX

POSIX ("Portable Operating Systems Interface", a name that Richard Stallman came up with) is a standard for the function calls (the *Application Programming Interface*, API) of UNIX-like general purpose operating systems. POSIX has some specifications or real-time primitives too. Its definition of real-time is quite loose:

The ability of the operating system to provide a required level of service in a bounded response time. The standard is managed by the Portable Application Standards Committee (http://www.pasc.org/) (PASC) of the Institute for Electrical and Electronic Engineers (http://www.ieee.org) (IEEE), and is not freely available. There is an extensive *Rationale* document that explains the reasons behind the choices that the POSIX committees made, as well as lots of other interesting remarks. That document can be found here (http://www.opengroup.org/onlinepubs/007904975/xrat/contents.html).

The POSIX components relevant to real-time are: 1003.1b (real-time), 1003.1d (additional real-time extensions), 1003.1j (advanced real-time extensions). See this link (http://www.opengroup.org/onlinepubs/007904975/idx/realtime.html) or here (IEEE Std 1003.1-2001) (http://www.unix-systems.org/version3/ieee_std.html) for more details. These standards are often also denoted as ANSI/IEEE Std. 1003.1b, etcetera.

POSIX also defines four so-called *profiles* for real-time systems:

• *PSE51 (Minimal Real-time System Profile)*. This profile offers the basic set of functionality for a single process, deeply embedded system, such as for the unattended control of special I/O devices. Neither user interaction nor a file system (mass storage) is required. The system runs one single POSIX process that can run multiple POSIX threads. These threads can use POSIX message passing. The process itself can use this message passing to communicate with other PSE5X-conformant systems (e.g., multiple CPUs on a common backplane, each running an independent PSE51 system). The

hardware model for this profile assumes a single processor with its memory, but no memory management unit (MMU) or common I/O devices (serial line, Ethernet card, etc.) are required.

• *PSE52 (Real-time Controller System Profile)*. This profile is the PSE51 profile, plus support for a file system (possibly implemented as a RAM disk!) and *asynchronous* I/O.

• *PSE53 (Dedicated Real-time System Profile)*. This profile is the PSE51 profile, plus support for multiple processes, but minus the file system support of the PSE52 profile. The hardware can have a memory management unit.

• *PSE54 (Multi-Purpose Real-time System Profile)*. This is the superset of the other profiles and essentially consists of the entire POSIX.1, POSIX.1b, POSIX.1c and POSIX.5b standards. Not all processes or threads must be real-time. Interactive user processes are allowed on a PSE54 system, so all of POSIX.2 and POSIX.2a are also included. The hardware model for this profile assumes one or more processors with memory management units, high-speed storage devices, special interfaces, network support, and display devices. RTLinux claims to comply with the *PSE51* profile; RTAI claims nothing. Linux's goal is POSIX compliance, but not blindly and not at all costs. The /usr/include/unistd.h header file gives information about which parts of the standard have been implemented already [Lewine, 91].

2.6.2 Unix98

UNIX (UNIX98, *Single UNIX Specification, Version 2*) is the standardization of UNIX operating systems driven by the Open Group (http://www.unix-systems.org/unix98.html). It incorporates a lot of the POSIX standards.

2.6.3 EL/IX

EL/IX. The EL/IX (http://sources.redhat.com/elix/) API for embedded systems wants to be a standards-compliant subset of POSIX and ANSI C.

2.6.4 µITRON

µ*ITRON.* µITRON (http://www.itron.gr.jp/home-e.html) is a Japanese standard for embedded systems. "TRON" stands for *The Real-time Operating system Nucleus*; the letter "I" stands for *industrial*, and the "mu" for *micro*. (There are other TRONs too: BTRON for business, CTRON for communication . . .) [Sakamura, 98].

2.6.5 OSEK

OSEK. OSEK (http://www.osek-vdx.org/) is a German standard for an open architecture for distributed vehicle control units. The architecture is open, but no free software implementation is available.

3. Implemented Operating System

There are many operating systems for embedded systems on the market today. They all support almost the same set of features in terms of basic functions. Most of them are targeted to a specific CPU. And some of them are not open-source. Most of the open-source embedded operating systems are hard to understand in terms of code readability and they are not very modular to expand further. Therefore they cannot be maintained easily. The implemented OS is simple, understandable, readable and extensible. There are many Doxygen comments within the code, which can be used to create the code documentation easily.

The name of the kernel is ARTIOS, which we will call as **AOS** for the rest of the document. It is a fully static real-time kernel which has a small footprint in terms of RAM and ROM usage. In this chapter the structure of AOS will be explained in detail.

3.1 Scope & Environment

Scope of this thesis is to design and implement and embedded operating system. The implementation covers the basic kernel functions for a multitasking environment and one platform to test the code.

A PC will be used for simulation purposes only during development. As the purpose is implement a kernel for embedded systems, PC is not a target platform for this kernel. No PC-specific code like boot-loader, drivers etc. will be implemented.

The static portable real-time kernel code is aimed to have an optimal modularity. This means that modularity without too much overhead. C macros are used in many places to increase the readability, modularity and portability.

Dynamic structures like dynamic task creation or dynamic semaphore creation are not a part of this implementation.

An MS Windows based PC is used for development. The testing code is run in 16 bit X86 DOS simulation. The kernel is designed for real-mode. Protected mode is not supported in the scope of this thesis.

3.2 General Embedded System Structure Using AOS

The general structure of an embedded system consists of roughly three parts, which are application itself, kernel and the hardware abstraction layer (HAL), as shown in the Figure 1. Generally the application is supposed to be platform independent and the operating system should be as platform independent as possible to make the porting easier. AOS is designed to be platform independent.

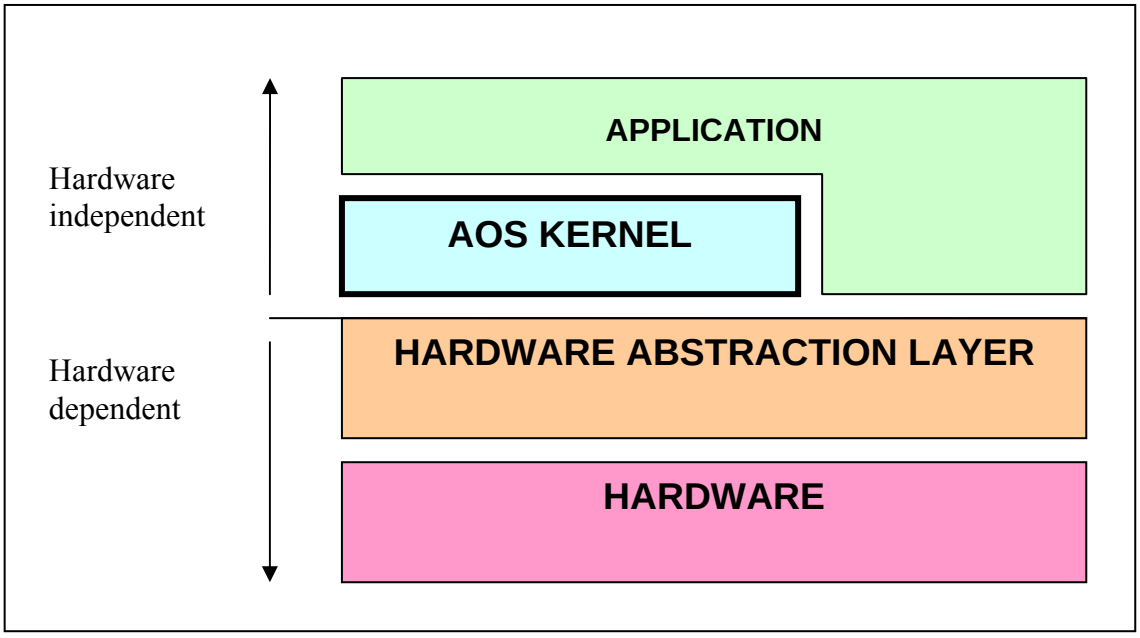

Figure 1. AOS system architecture

HAL is responsible for isolating the software from the hardware details. It may also consist of internal layers, but this is not in the scope of this document. The top layer of a HAL includes a platform independent application programming interface (API).

Kernel runs above the HAL and provides application with some functionality of pseudo parallel programming, i.e. multitasking; along with some other services like task synchronization, dynamic memory allocation, inter-task communication etc.

A clear, well defined kernel API, which provides a good abstraction, makes the programmer's life easier, as he/she should not deal with the details of internal implementation.

3.3 Internal Blocks of AOS

AOS consists of well defined, abstracted blocks as shown in Figure 2. The blocks are isolated from the application with an API layer, which defines all of the public AOS functions and structures that might be necessary to use within the application. The full AOS API is defined in the file AOSAPI.h, which is documented in the Appendix A1.

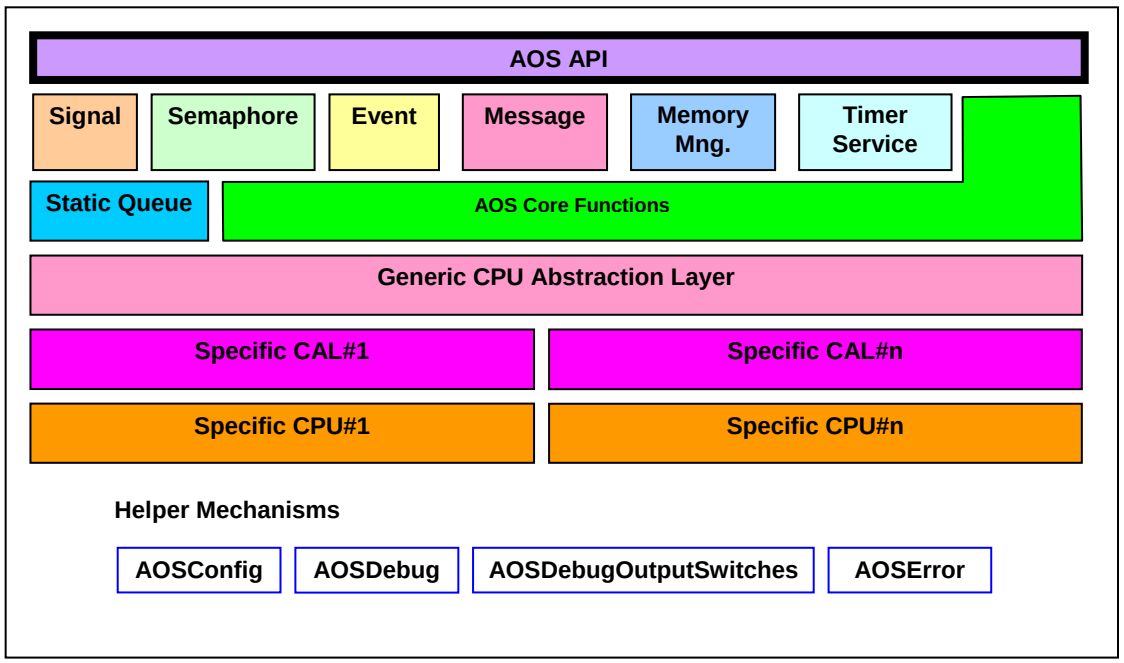

Figure 2. Internal blocks of AOS

3.3.1 The blocks of API

The API blocks can be separated into five logical parts, which are

- Scheduling and directly kernel related calls
- Task synchronization
- Task communication
- Memory management
- Timer services

Core functions include OS management, scheduling and task management. These are used to initialize and start the OS as well as the hardware; to create/ terminate tasks; and to control the scheduling mechanism by using the reschedule, sleep, suspend

and resume calls. Real-time scheduler checks the task states and starts the task which needs to be started immediately. The AOS kernel is a real-time kernel which means that the task that needs to be wakened is put in front of its ready-queue and dispatched as soon as the OS finishes its internal operations.

Task synchronization part consists of signaling subsystem, semaphores, mutexes and event flags. By using these mechanisms, critical region handling and resource management are done easily. In AOS every synchronization event is a subclass of signaling. The system handles the synchronization events as signals. This will be explained in more detail in AOS Mechanisms section.

AOS provides two ways of inter-task communication; which are task data and message queues or mailbox that each task has. Task data is a very simple way in terms of passing data between tasks. Message queue is more sophisticated as it includes blocking of tasks also. The details will be explained in more detail in AOS Mechanisms section.

Memory management of AOS is static like the other parts of the kernel. It provides fixed size memory blocks on allocation request and blocks the task if there is not enough memory. Because of its static and fixed size architecture it is fast and it does not cause external memory fragmentation. Fragmentation is kept only inside the fixed sized blocks. The details will be explained in more detail in AOS Mechanisms section.

The last service provided by the AOS is the asynchronous timer service. This service provides the tasks with asynchronous callbacks, as well as periodic callbacks, which can also be associated with signaling subsystem.

3.3.2 Static Queue

The static queue is the heart of the AOS. It is a double linked list which is simple and has a fast operation capability. The details will be explained in the AOS Mechanisms section.

3.3.3 Generic CPU Abstraction Layer

The CPU abstraction layer is the block that isolates the kernel implementation completely from the underlying hardware. Theoretically AOS can be ported to any CPU

by only changing this adaptation layer. The specific CALs must implement the interface that defined in the file GenericCAL.h.

3.3.4 Helper Mechanisms

AOS has four basic helper mechanisms which are used for debugging and configuring the OS itself and the application. These are AOSConfig, AOSDebug, AOSDebugOurSwitches and AOSError.

The debugging system can turned on and off totally or partially by using the debug switches. It provides a way of seeing the messages coming from the kernel to the terminal screen. When the debug switches are disabled all of the debug messages are removed out of the system and the performance is increased.

The error handling mechanism provides a way of seeing and analyzing the errors during system development. It can be disabled after the code has been fully debugged to increase the overall performance of the system. When disabled the functions that return an error return void.

AOSConfig provides scalability of system. Almost every parameter of the AOS is configurable to achieve a flexible kernel. The kernel can be configured as small as two tasks and all the options disabled; Or it can handle up to 255 tasks excluding the main with all the services enabled. The widths of the data structures like timer variables can be changed according to the need.

3.4 General Properties of AOS

Coding: The coding of the AOS is clear, modular and warning-free ANSI-C. In many parts of the kernel code macros were used to increase the readability of the code. And modularity that is achieved with the use of functions is kept at an optimum point in case not to slow down the system and overload the stack with many successive function calls.

Portable: The main implementation of the real-time kernel was written in highly portable ANSI C, with target microprocessor specific code written in C and assembly language, in completely separate module. This module is called CAL and isolates the

kernel implementation from the details of the underlying hardware. Assembly language is kept to a minimum to make it easy to port to other processors. The target kernel is planned to be able to run on most 8, 16 or 32 bit microprocessors or microcontrollers.

ROMable: The AOS kernel is designed for embedded applications. This means that it can be embedded by using the proper tool chain (compiler, assembler, linker and locator) as part of a product.

Scalable: AOS is designed so that one can use only the services needed for the specific application, which means that a product can use just a few services, while another product can benefit from the full set of features. Scalability allows us to reduce the amount of memory (both RAM and ROM). Scalability is accomplished with the use of conditional compilation.

Preemptive: AOS is a fully preemptive real-time kernel, which means that it always runs the highest priority task that is ready. Most commercial kernels are preemptive and it is a must for real-time operations. AOS also provides time slice support as a part of the kernel.

Multitasking: The kernel is able to manage up to 255 tasks (excluding main) simultaneously. Each task may have a unique priority assigned to it. The priorities can also be increased up to 255. Of course this means in an increase of RAM usage and slows down the system. AOS can be configured just as the system needs.

Deterministic: Execution times for all kernel functions and services are deterministic, which means that you can always know how much time it will take to execute a function or a service. The worst case and best case response performance of the system can be calculated easily depending on the CPU used.

Task stacks: Each task requires its own stack. There is a protection mechanism to avoid stack overflows, which can be enabled or disabled by a configuration switch in AOSConfig.h.

Services: The operating system provides a number of system services, such as semaphores, mutual exclusion, event flags, message queues, fixed size memory partitions, task management, timer management functions and etc.

Interrupt management: Interrupts can suspend the execution of a task. If a higher priority task is awakened as a result of the interrupt, the highest priority task runs as soon as all nested interrupts complete.

3.5 AOS Internal Mechanisms

This part explains the internal structure and mechanisms of AOS real-time kernel.

3.5.1 The Static Queue

The static queue is double linked circular list. The term *static* is used because the queue never allocates or clears nodes, as they are attached and detached. The objects that might be part of a queue already contain the queue node object, i.e. they are inherited from the queue node, as shown in Figure 3.

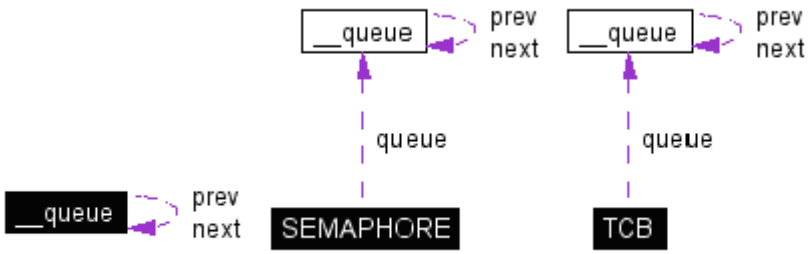

Figure 3. Objects that inherited from queue node.

The head of the queue is also a queue node instead of a pointer. This way there is no need to check null pointers and queue mechanism operates faster. But if has drawback that an object can only be a member of one queue at any time. The same object cannot exist in different queues simultaneously. An empty queue consists of the head node with previous and next pointers pointing to it. This is shown in Figure 4.

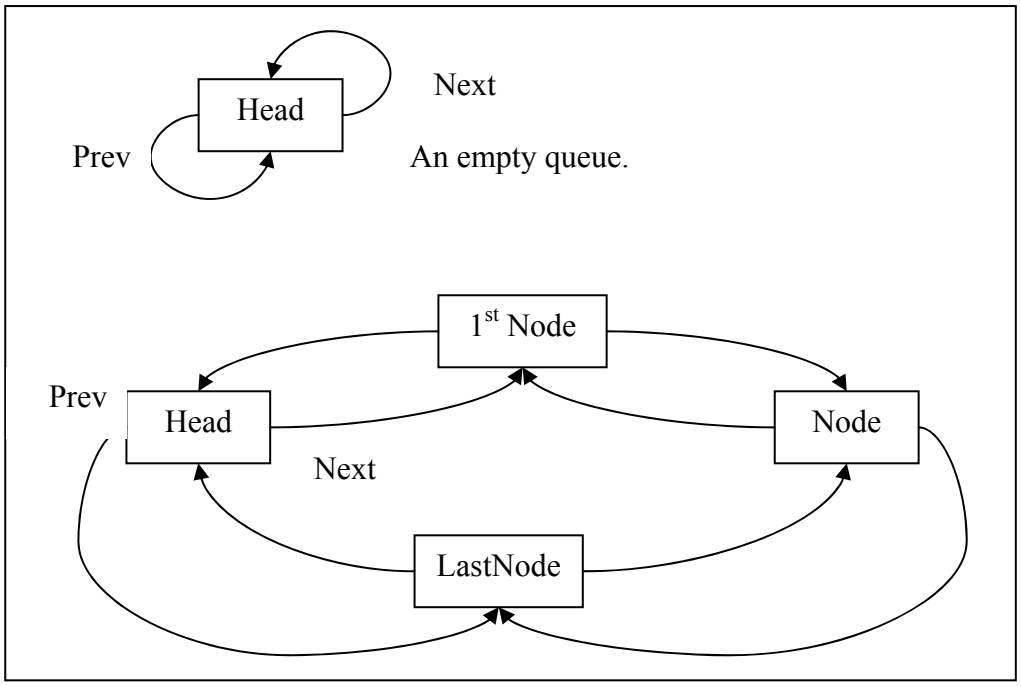

Figure 4. Queue nodes connection diagram.

The queue interface provides the following functionalities.

void SQ_ResetNode (QUEUE* node): Resets the pointers of the given queue node.

void SQ_Add (QUEUE * queue, QUEUE * node): Attaches the given node to the desired queue.

void SQ_AddFront (QUEUE * queue, QUEUE * node): Attaches the given node to the front of the queue. This function is used mostly when a signal is arrived.

void SQ_Remove (QUEUE * node): Detaches the node from any queue.

QUEUE *SQ_GetFirstNode (QUEUE *queue): Removes and returns the first node of a queue.

void SQ_Rotate (QUEUE *queue): Rotates the queue in way that the first node goes to the end and second node becomes the first one.

3.5.2 The Ready Queue

The ready queue is a list of queues, where each queue contains the ready tasks of a specific priority as shown in Figure 5. The scheduler chooses the first highest priority task inside the ready queue. The number of queues inside the ready queue is defined by the configuration variable AOS_PRIO_NUM in AOSConfig.h.

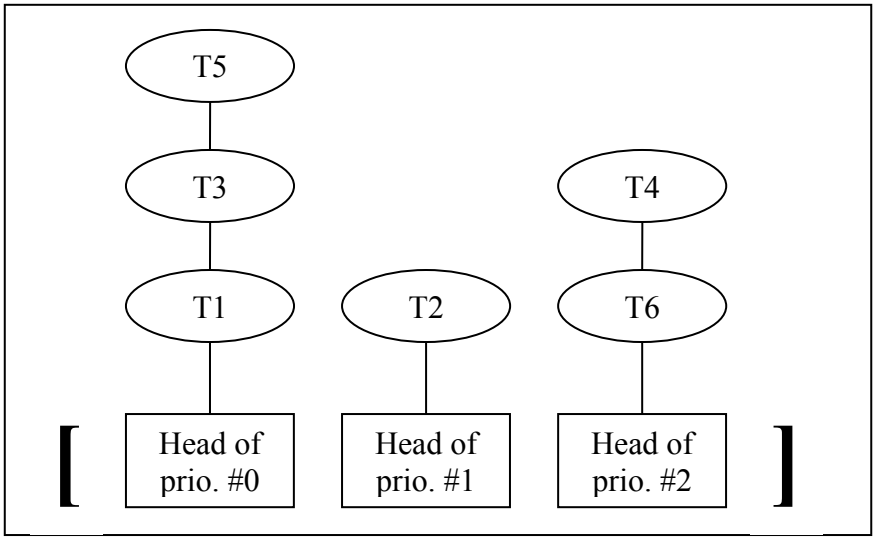

Figure 5. The ready queue

3.5.3 Priorities

AOS can supports up to 255 priority levels. But keeping this number at minimum decreases the memory usage and speeds up the execution, as for each priority level another ready queue node is introduced.

The number of priorities can be configured by AOS_PRIO_NUM. The highest priority is zero and has a constant definition assigned to it, which is AOS_HIGHEST_PRIORITY. The lowest priority is one less than the number of priorities and it is designated with AOS_LOWEST_PRIORITY in the file AOSConfig.h.

3.5.4 The Task Control Block – TCB

The Task Control Block (TCB) keeps the initial and runtime variables related to a specific task. The size of the TCB changes according to the selected features within the configuration file. The definition of TCB structure is in the file artios.h. The basic

members of the TCB are current stack pointer, identification number, priority and status information of the task.

3.5.5 The Task List

The taskList is the array of the created tasks. It is a static array like the rest of the system. The number of the TCBs (number of tasks) is defined by the configuration parameter AOS_TASK_MAX. This configuration parameter should be defined just as the number tasks that will be used. Defining this more only increases the memory usage and decreases the performance. The maximum allowed task number is 255. AOS_TASK_NONE is invalid task id and its value is 0xff.

3.5.6 Active Task

The activeTask is the id of the current running task. The general structure of AOS is based on task ids instead of TCB pointers. The reason of this to keep the memory and stack requirement at minimum. Because passing the pointer means that passing between 2 or 4 bytes data at each function call, where id passing needs only one byte to be passed. Of course this introduces a little slowdown when the performance is considered, because the real address should be calculated from the index each time. But when the CPU has only 256 bytes of RAM this trade-off is not that important.

3.5.7 Task States

A task may be in one of the following states. The state transitions as results of the API and internal calls are shown in Figure 6.

TS_READY: The task is ready to run and is waiting in its ready queue.

TS_RUN: Task is currently active and running.

TS_TERMINATED: Task is terminated. It is not a part of a queue anymore. It can be started by an AOS_RestartTask () call, if enabled in configuration.

TS_SUSPENDED: Task is suspended and removed from the queues. It can be resumed by an AOS_Resume () call.

TS_WAIT_TIMEOUT: Task is waiting for a timeout event. The timeout wait is the only state that can be used in combination with one of other wait states. The purpose of this is to achieve the timed wait mechanism. For example; A task might wait for an event only for a certain amount of time. In this case the wait state is a combination of event wait and timeout wait. Whichever comes first, the task is signaled and run again.

TS_WAIT_SIGNAL: Task is waiting for an external signal.

TS_WAIT_SEMA: Task is waiting for a semaphore or mutex.

TS_WAIT_MSG: Task's message queue is empty and a ReceiveMessage call has been issued.

TS_WAIT_EVENT: Task is waiting for an external event or events to occur.

SET_TASK_STATE() and RESET_TASK_STATE() macros are responsible for changing the task state. IS_TASK_STATE() macro can be used to query the current state of the task.

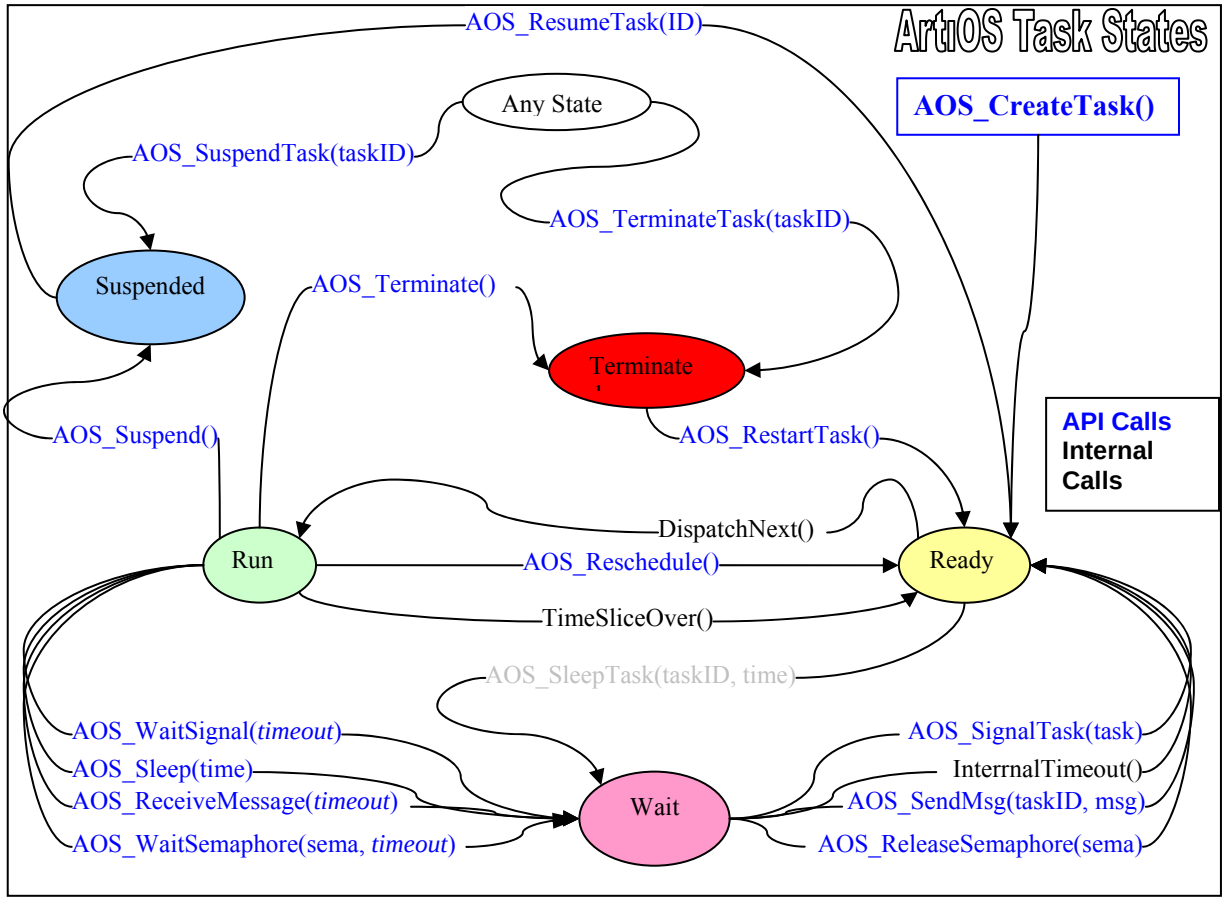

Figure 6. Task state transition diagram

3.5.8 AOS Internal States

The kernel has internal states that determine behavior of the system. These are as follows and the transition diagram is shown in Figure 7.

OSS_NORMAL: OS functions called from normal operation. The dispatch is enabled.

OSS_INIT: OS is not started. It is in the initialization phase. Dispatch is disabled.

OSS_WAIT: Ready queue empty. Interrupts are enabled and waiting for a task to become ready. In this state no real task is running except just an infinite wait loop. Therefore when AOS is in this state the value of the activeTask variable is invalid and no operation on this variable can be performed.

OSS_INT: OS functions called when the system is inside an interrupt. Dispatch is disabled. This state is exited when all the nested interrupts are finished. AOS_EnterInterrupt() and AOS_ExitInterrupt() calls are responsible for entering and exiting this state.

OSS_LOCKED: Disable OS Dispatch. This state is entered after an AOS_Lock() call and is exited after an AOS_Unlock() call is issued.

The tasks can only be changed when the internalState is OSS_NORMAL. The internalState can be a combination of OSS_INT and OSS_LOCKED.

SetInternalState(), ResetInternalState() macros are responsible for changing the OS state. IsInternalState() macro is used to query the internal state.

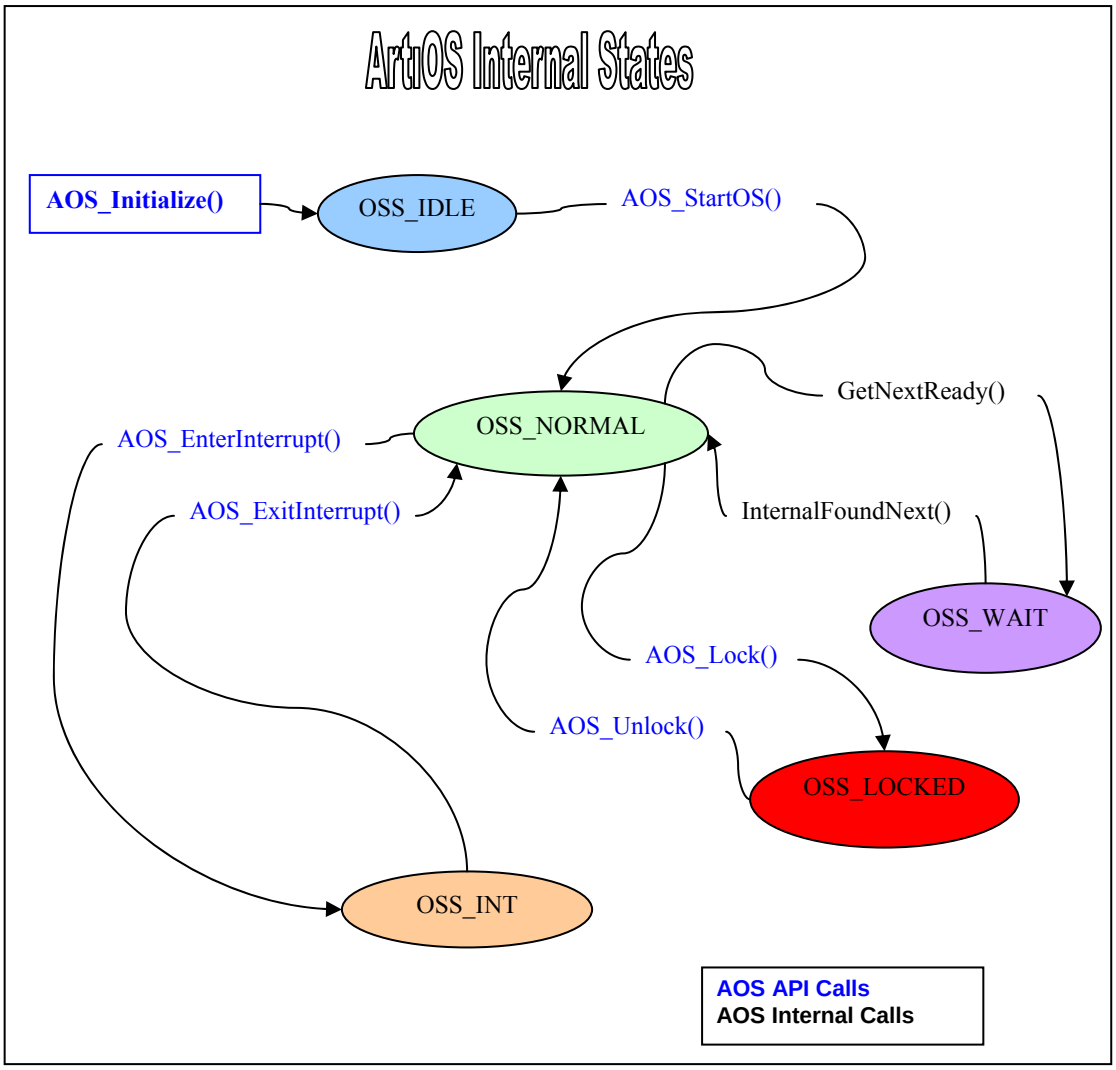

Figure 7. AOS Internal state transition diagram

3.5.9 Task Dispatching

Task dispatching is done with internal call DispatchNext(). For task dispatching the internal state of the kernel should be OSS_NORMAL. If not, this function cancels the dispatch process and sets the dispatchRequested flag. The requested dispatch is accomplished later on as soon as the OS state becomes OSS_NORMAL. This behavior is called delayed dispatch.

DispatchNext() has nothing to do with current running task. It only gets the next highest priority ready task from ready queue and starts the execution of it from where it left off. It calls GetNextReady(), which returns the id of the next ready task and if there is no ready task at the moment, GetNextReady enables interrupts, sets the internal state

to OSS_WAIT and enters an infinite loop, where it waits for a task to become ready. The search for the next ready process is done by the function SearchNextReady().

3.5.10 Interrupt Control

Interrupt control is very important for real-time operating systems. Tasks cannot be changed within an interrupt, therefore the kernel must be aware of the arriving interrupts. This is done through AOS_EnterInterrupt() and AOS_ExitInterrupt() calls.

These functions should be used at beginning and at the end of all interrupt service routines (ISR). AOS_EnterInterrupt() sets the internal state to OSS_INT and increases the nested interrupt counter. This operation disables the dispatcher. On exiting the ISR AOS_ExitInterrupt() call decreases the nested interrupt counter. If the nested interrupt counter is zero it sets the internal state again to OSS_NORMAL and starts a dispatch process. The usage of the AOS_EnterInterrupt() and AOS_ExitInterrupt calls area as follows.

```
INTERRUPT OSTimerISR()
{
        AOS_EnterInterrupt();

        AOS_TimerInterruptHandler();
        /// Acknowledge PC interrupt controller
        outportb(INT_8259A_PORT, 0x20);

        AOS_ExitInterrupt();
}
```

3.5.11 Critical Region Management

Some operations of the kernel should be atomic. This means that one operation should not be interrupted until it is finished. This can be done by disabling the interrupts on entry to critical region and by enabling them on exit. However this should be achieved without platform dependency to keep the kernel code portable. Therefore there

are two macros provided by the CPU abstraction layer (CAL); CAL_EnterCriticalRegion() and CAL_ExitCriticalRegion().

CAL_EnterCriticalRegion() saves the last CPU flag values and disables the interrupts. The CAL_ExitCriticalRegion() restores back the CPU flags. It does not directly enable the interrupts as they might already be disabled before entering critical region.

One important thing that should be taken into account when using these functions is to keep the critical region as short as possible. Because; as the interrupts are disabled, the kernel is also inactive which means that long stay in critical regions may result in missing OS ticks, which means that some tasks miss their deadlines. This is very dangerous for the systems, where human life is important, like ABS brake system controllers.

Example to the usage is as follows.

```
void InternalWaitSignal(TASK_STATUS waitState)
{
        CAL_EnterCriticalRegion();

        RESET_TASK_STATE(activeTask, TS_RUN);
        SET_TASK_STATE(activeTask, waitState);
        DispatchNext();

        CAL_ExitCriticalRegion();
}
```

3.5.12 Locking The Dispatcher

If long operations like a communication burst are needed, where the active task should not be swapped out, then the application programmer may lock the dispatcher instead of disabling interrupts, which results in system irresponsiveness. Locking the dispatcher does not kill the system. It only disables the dispatch mechanism so that no other task can interrupt the current operation. This functionality is achieved by

AOS_Lock() and AOS_Unlock() kernel calls, which will be explained in detail in API Details section.

3.5.13 CPU Abstraction Layer – CAL

CAL is used to isolate the kernel implementation from the details of the underlying hardware. CAL has a very clear well defined generic API, which makes the rest of the kernel portable. When porting the AOS theoretically only the CAL should be rewritten. CAL has following calls

CAL_InitializeHardware(): Initializes the hardware, which means initializing the IO, interrupts and timers.

CAL_InitializeTaskStack(): This initializes the task's stack according to the parameters that are passed in. And it returns the real stack pointer after initializing the stack. This is explained in Initializing Task Stacks section.

CAL_ChangeStack: This is the function, where the context switch happens. This is explained in Context Switching section.

CAL_EnterCriticalRegion(): Enter the critical region by disabling the interrupts.

CAL_ExitCriticalRegion(): Exit the critical region by restoring the CPU flags.

3.5.14 Context Switching

In a strict sense multitasking cannot occur on a single CPU, and therefore, multitasking must be simulated by sharing CPU time between different tasks.

Context switching is the name of the operation which changes the current running task on the CPU. This is done by changing the current stack pointer with the next task's stack pointer, after saving the registers of the current running task. CAL_ChangeStack() function is responsible for this.

Tasks cannot share a common stack like subroutines. Because using common stack means that tasks corrupt each other's data. The solution is for each task to have its own stack.

Before switching from task A to task B, all information necessary to resume task A must be saved. The information that must be saved is:

1. The address of the instruction at which execution should resume.
2. The CPU flags.
3. All registers.
4. The stack pointer.

All of this information must be saved because it will change when control is switched to task B.

The simplest way to save this information is to use the TCB. Before switching out the task A all of its information should saved to its TCB. Then, the context of task B could be restored from task B's TCB. This approach, however, has shortcomings. The preferred way is to save only the task's stack pointer in the TCB, and save everything else on the task's private stack. Task switching from task A to task B happens as follows.

1. All registers, including IP and the flags are pushed onto task A's stack.
2. Task A's stack pointer is saved in its TCB
3. Task B's stack pointer is restored from task B's TCB.
4. All register including the flags are popped from the stack. Task B's stack is now the system stack.
5. Control is returned to the place where task B left off by jumping to the value of IP that was saved on task B's stack.

The interrupts must be disabled during task switching, to disable another task switch in the middle of task switch.

3.5.15 Initializing Task Stacks

Task stacks are initialized by the CAL_InitializeTaskStack() call. This function gets the stack buffer and the size as parameters. Fills the required data, including the task entry and return pointers, to the bottom of the stack according to the current

configuration of the system and returns the resulting stack pointer value. The initial stack looks like in Figure 8.

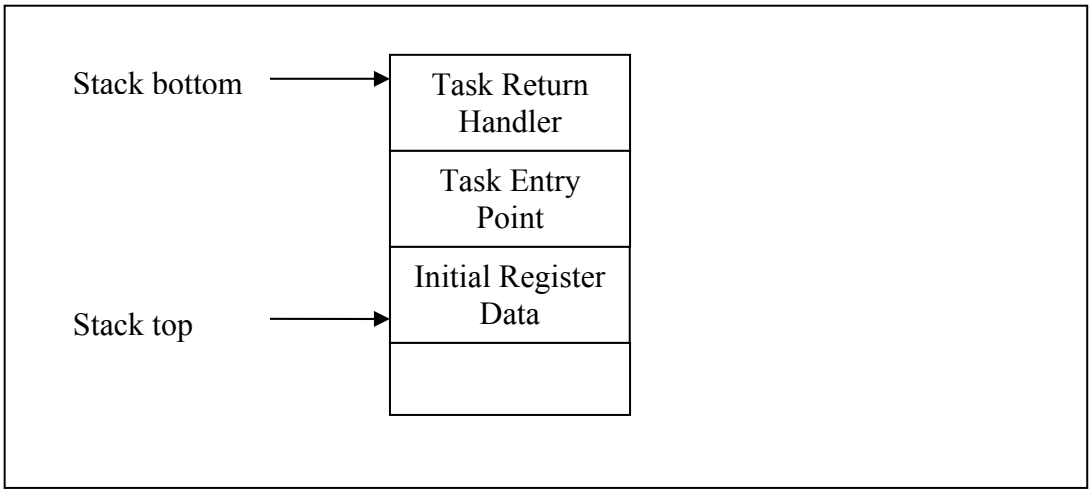

Figure 8. Initial stack of a task

3.5.16 Task Scheduling

There are various ways in which a task can be scheduled. AOS deals only with techniques for implementing priority based preemptive scheduling.

The ready queue of AOS is a doubly linked list of tasks waiting to run, ordered by task priority, the highest priority waiting task being at the front of the queue, (referred to as the *ready task*), and the lowest at the rear of the queue. The kernel schedules a task to run by inserting it into the queue according to its priority. Each element of the list contains a reference to a TCB. Implicit is the fact that the active task is of a higher priority than the highest priority waiting task.

The kernel begins multi-tasking by removing the task at the front of the ready queue and executing it via a context switch. The task will run until it voluntarily relinquishes control or is preempted, at which time a context switch to the ready task is performed.

A task can voluntarily relinquish control explicitly with a AOSReschedule or AOS_Suspend, or; implicitly with AOS_ReceiveMessage, AOS_WaitSemaphore etc. When the active task relinquishes control, the task referenced by the node at the front of the ready queue becomes the new active task.

Tasks can involuntarily relinquish control when a message is passed as described below, or via time-slicing.

When a message is sent, the destination task is scheduled to run, which has some interesting implications. One implication is that when a lower priority task sends a message to a higher priority task, the lower priority task should suspend so that the higher priority task can run. This can be used as a mechanism for interrupt handling as described below.

Sometimes an ISR should not return to the interrupted task, but rather to a higher priority task that must run as a result of the interrupt. This can be effected using the message passing rule alluded to above. So, if lower priority task A is interrupted to run ISR A and ISR A sends higher priority task B a message, then on executing the IRET instruction, task A should remain suspended, and task B resumed.

3.5.17 Signaling Subsystem

In AOS each wait operation is translated to an internal wait signal. For example a call to AOS_WaitSemaphore() is translated to an InternalWaitSignal(TS_WAIT_SEMA). Each task can wait only for one signal at any time, except for the case of timeout signal.

A timeout signal can determine the wait state alone or it can be combined with one of the other wait signals like message signal. This is used to achieve the purpose of timed wait mechanism. It means that a task should not wait for an event to occur infinitely.

The function InternalSendSignal resets the state of the task according to the event that is sent and reschedules the task.

3.5.18 System Startup and The Main Task

main function of the application is the main task for the kernel. During kernel initialization a priority is given to the main task as well. The only difference of the main task is that it does not need an explicit stack definition. Because the stack for the main task is allocated by the compiler at compile time. And main task's id is always zero.

The main task is responsible for creating other tasks and initializing and starting the kernel. After it may suspend or terminate itself. But main task is not allowed to return. Normally the approach is to use the main task like any other system task and run it at a desired priority. Following is a simple system startup code.

```
main()
{
   AOS_Initialize(AOS_LOWEST_PRIORITY);
    AOS_CreateTask( TASK1_ID,
                        Task1Func,
                        (TASK_STACK)stack1, 0x1000,
                        AOS_HIGHEST_PRIORITY, AOS_FALSE);
   AOS_StartOS();
// Now main task starts to wait messages from others.
   while(1)
   {
        MESSAGE msg;
        msg = AOS_ReceiveMessage(AOS_TRUE);

        printf("Message %d from %s\n", msg.id, (char*)msg.param);
   }
}
```

3.5.19 Time Slicing

AOS supports time slicing with the configuration parameter AOS_USE_TIMESLICE defined in the AOSConfig.h. AOS_TIME_SLICE_PERIOD defines the period at which the task change will happen. When time slicing is enabled the AOS_TimerInterruptHandler counts the time and when the counter reaches to zero it issues an AOS_Reschedule call to resume the next task. The scheduling happens according to the priorities of the tasks waiting in the ready queue.

3.5.20 Task Sleep

AOS supports task sleep with the configuration parameter AOS_USE_TASK_SLEEP defined in the AOSConfig.h file. The sleeping tasks are not kept in a queue. At each timer tick all of the non-zero TCB counters are decremented and the tasks that reach to zero count will be scheduled to run.

A delay queue is not used because when there are only few tasks the update procedure does not take much time. Therefore there is no need to call queue handling functions. This decreases the system complexity and increases the performance for small number of tasks.

3.5.21 Task Synchronization and Communication

To allow tasks to cooperate and compete for resources, it is necessary to provide mechanisms for synchronization and communication. The classic synchronization mechanisms are semaphores and mutexes. There are provided in the AOS kernel, together with other synchronization/communication mechanisms that are common to real-time systems, such as event flags and message queues.

The API calls that AOS provide for synchronization and communication are as follows and explained in detail in API Details section.

- AOS_WaitSignal and AOS_SignalTask
- AOSWaitSemaphore and AOS_ReleaseSemaphore
- AOS_EnterMutex and AOS_ExitMutex
- AOS_WaitEvent and AOS_SendEvent
- AOS_WaitMessage and AOS_ReceiveMessage

Timed wait versions of the above API functions are also available.

3.5.22 Support for Priority Inversion

One of the problems that must be dealt with in any real-time systems is priority inversion. This is where a high priority task is (wrongly) prevented from continuing by one in lower priority. The normal example is of a high priority task waiting at a mutex already held by a low priority task. If the low priority task is preempted by a medium

priority task then priority inversion has occurred since the high priority task is prevented from continuing by and unrelated task of lower priority.

This problem got much attention recently when Mars Pathfinder mission had to reset the computers on the ground exploration robot repeatedly because a priority inversion problem would cause it to hang.

The solution is to use the priority inheritance protocol. Here the priority of the task that owns the mutex is boosted to equal that of the highest priority task that is waiting for it. The priority of the owning task is only boosted when a higher priority is waiting [Yodaiken, 2002].

When this support is enabled by setting the configuration parameter AOS_AVOID_PRIORITY_INVERSION the initial priority of each task is kept in the TCB and the active priority can be changed when priority inversion occurs. The task's priority is reset back to the initial value once it releases the resource. Enabling this support increases the memory usage by one byte for each TCB and adds a small overhead when handling the semaphores.

3.5.23 Task Management

AOS supports the standard task management functions that all the other real-time kernels do. These are AOS_CreateTask, AOS_Terminate, AOS_RestartTask, AOS_Suspend, AOS_ResumeTask and AOS_Sleep. These are explained in detail in API Details section.

3.5.24 Memory Management

AOS provides fixed size memory allocation from a given memory pool. The fixed size allocation is fast and does not cause external memory fragmentation. There are two basic functions that AOS provides. These are AOS_RequestMemoryBlock and AOS_ReleaseMemoryBlock. Requesting a memory block can also be a blocking call depending on the parameters. The details are explained in API Details section.

3.5.25 Timer Service

AOS has support for asynchronous timers, which can be configured as once or periodic. The timers are in a timer pool and the tasks requesting the timer get the handle to the timer object. When the requested timeout passes the given callback function is automatically executed by the kernel.

3.6 Helper Mechanisms

The mechanisms explained in this part provide flexibility, configurability and debug ability to the kernel.

3.6.1 AOS Types

AOSTypes.h is one of the mechanisms that make AOS portable and flexible. All of the types that are used inside the kernel are defined in this file. This brings portability to the system. Also the width of kernel variables like time period, which is 16bit by default, can be changed whenever needed. This means that AOS can be configured just as needed. It can have a very small footprint in one configuration and in another one; it can be changed to a kernel that handles time delays of hours in resolution of milliseconds.

3.6.2 AOS Configuration

All of the features of AOS are configurable and the configuration switches reside in the file AOSConfig.h.

The configuration is at function level for most of the features. Also the number of priorities, number of tasks, sizes of arrays, timers, etc. can be configured here. The main debug switch and error handling enabling and disabling are configured within this file.

3.6.3 AOS Debug Mechanism

AOS has debug messages inserted in necessary and vital parts of the kernel that can be enabled and disabled partially or totally. Total disabling of debug messages is

done by the configuration switch AOS_DEBUG inside the file AOSConfig.h. And each module's messages can be configured by module trace control switches inside the AOSDebugOutSwitces.h file. This way only the necessary part of the messages can be enabled during a debugging process. By disabling the main switch AOS_DEBUG all of the debug messages are removed completely from the code. Debugging mechanism should only be used during development process as it slows down the system and increases the size of the final binary code.

3.6.4 AOS Error Handling

Each kernel function that has the possibility to fail to complete its operation returns an error. Errors are printed to the terminal screen if the switch DBG_OUT_ERROR is enabled.

Error codes are enumerated single byte values and their base is configurable.

The error checking can be disabled by disabling the configuration switch AOS_HANDLE_ERRORS. In this case error checking is disabled and the functions that normally return an AOS_ERR return void.

Disabling the error handling mechanism increases the overall system speed around 10% and decreases the code size. Therefore it is better to use this mechanism only during development phase of a project.

3.7 API Details

3.7.1 Kernel Control

3.7.1.1 AOS_Lock( ) / AOS_Unlock( )

Brief:

Suspends all real-time kernel activity while keeping interrupts (including the kernel tick) enabled.

After calling AOS_Lock() the calling task will continue to execute without risk of being swapped out until a call to AOS_Unlock () has been made.

All the calls that try to resume another task will be delayed until AOS_Unlock().

Details:

At some point the task wants to perform a long operation during which it does not want to get swapped out. It cannot use CAL_EnterCriticalRegion() and CAL_ExitCriticalRegion() as the length of the operation may cause interrupts to be missed - including the ticks.

Prevent the real-time kernel swapping out the task by issuing an **AOS_Lock()** call. And then perform the operation. There is no need to use critical sections as we have all the microcontroller's processing time.

During this time interrupts will still operate and the kernel tick count will be maintained. After the operation is complete enable the context switching by issuing an **AOS_Unlock()**.

Example Usage:

```
void AOSTask1( )
 {
    while(1)
    {
      // Task code goes here.

          AOS_Lock();

// Perform the operation
ReadTemperatureSensors(tempData *, size);

       AOS_Unlock();
    }
 }
```

3.7.1.2 AOS_Initialize(TASK_PRI priority)

Brief:

Initializes the real-time kernel and assigns the priority to the main task.

Details:

Resets all internal variables, queues and the main task variables.

Calls the CAL_InitializeHardware to initialize the hardware and sets the internal state to OSS_INIT.

3.7.1.3 AOS_StartOS( )

Brief:

Starts the kernel.

Details:

Puts the internal state the OSS_NORMAL. And starts the hardware.

3.7.1.4 AOS_EnterInterrupt( )

Brief:

This function should be used on entry of each interrupt service routine to notify the kernel that an isr is about to start.

Details:

Puts the OS to interrupt handling state OSS_INT. Also increases nested interrupt counter.

3.7.1.5 AOS_ExitInterrupt( )

Brief:

This function should be used on exit of each interrupt service routine to notify the kernel that an isr has just finished executing.

Details:

Decreases nested interrupt counter. If the counter reaches zero changes the internal state back to OSS_NORMAL and dispatches next ready task if dispatch is requested.

Example Usage:

```
INTERRUPT OSTimerISR()
{
        AOS_EnterInterrupt();

        AOS_TimerInterruptHandler();
        /// Acknowledge PC interrupt controller
        outportb(INT_8259A_PORT, 0x20);

        AOS_ExitInterrupt();
}
```

3.7.1.6 AOS_TimerInterruptHandler( )

Brief:

This function is called by the OS timer tick isr to let the kernel update its internal structure and state.

3.7.1.7 AOS_Reschedule( )

Brief:

Reschedules the current running task.

Details:

The task is sent to end of its ready queue and the next ready task is dispatched.

3.7.2 Task Management

3.7.2.1 AOS_CreateTask(TASK_ID task,

PFV entryPoint,

TASK_STACK stack,

AWORD stackSize,

TASK_PRI priority,

ABOOL createSuspended);

Brief:

Initilalizes the specific task

Details:

Initializes the task stack buffer and sets the start state.

Example Usage:

```
#define TASK1_ID    1

char stack1[0x1000];
void Task1Func()
{
        while(1)
        {
                // Do something
        }
}

//inside main
  AOS_CreateTask(TASK1_ID,  Task1Func,  (TASK_STACK)stack1,  0x1000,  0,
AOS_FALSE);
```

3.7.2.2 AOS_ChangeTaskPriority(TASK_ID task, TASK_PRI newPri)

Brief:

Changes the priority of the given task.

Details:

If the requested priority is higher than the priority of the active task a context switch occurs.

3.7.2.3 AOS_ERR AOS_TerminateTask(TASK_ID task) & AOS_Terminate()

Brief:

Used to terminate the current or the requested task.

Details:

The task is removed from any queue and its state is set to TS_TERMINATED.

3.7.2.4 AOS_Sleep(SLEEP_TIME time)

Brief:

Puts the task in to sleep for a desired period of time.

Details:

The resolution is 1 ms. The next ready task is dispatched.

3.7.2.5 AOS_ERR AOS_SuspendTask(TASK_ID task) & AOS_Suspend()

Brief:

Used to suspend the current or the requested task.

Details:

The task is removed from any queue and its state is set to TS_SUSPENDED.

3.7.2.6 AOS_RestartTask(TASK_ID task)

Brief:

Used to restart a terminated task.

Details:

When this feature is enabled the initialization data for each task is kept in their TCB. On restarting the task this initial data is loaded again.

This function only restarts a terminated task.

3.7.2.7 AOS_ResumeTask(TASK_ID task)

Brief:

Used to resume a suspended task.

Details:

The state of the task is set again to TS_READY.

This function only resumes a suspended task.

3.7.3 Task Synchronization & Communication

3.7.3.1 AOS_WaitSignal() / AOS_SignalTask(TASK_ID task)

Brief:

This function pair is the simplest synchronization mechanism. A task that issues a WaitSignal is scheduled again with a signal from another task.

Details:

To signal a task the task need not to be in wait signal state. In any case signaling reschedules the task, if it is not waiting for other signals.

3.7.3.2 AOS_InitSemaphore(SEM_ID id, SEM_CNT cnt, SEM_CNT maxCnt)

Brief:

Initializes the desired counting semaphore in the semaphore table.

Details:

The maximum number of semaphores can be changed by using the configuration parameter AOS_SEMA_MAX.

3.7.3.3 AOS_WaitSemaphore(SEM_ID id) / AOS_ReleaseSemaphore(SEM_ID id)

Brief:

Wait semaphore blocks the calling task if the semaphore is not available. Releasing a semaphore increases the semaphore count by one and schedules any task waiting in the semaphore's queue.

Details:

Releasing semaphore cannot increase the count of the semaphore to more than its maximum defined at the initialization.

3.7.3.4AOS_GetSemaphoreCount(SEM_ID id)

Brief:

Returns the current count of the semaphore.

3.7.3.5 AOS_SendMessage() / AOS_ReceiveMessage

Brief:

This function pair is responsible for sending and receiving messages. Sending message does not block the sender if the receiver is unable to receive it. But if the receiver's message queue is empty an attempt to receive message blocks the receiver.

Details:

When the receiver is already blocked and the sender has a lower priority than the receiver, receiver is scheduled to run on message arrival.

3.7.4 Task Information

3.7.4.1 AOS_GetActiveTaskID( )

Brief:

Returns the id of the current running task.

3.7.4.2 AOS_GetTaskStatus(TASK_ID task)

Brief:

Returns the status information of the desired task.

3.7.4.2 AOS_GetTaskPriority(TASK_ID task)

Brief:

Returns the priority information of the desired task.

4. Results

The kernel has been successfully implemented with all the targeted features and the trial applications run without any problem under 16bit DOS emulation. Table 1 shows the feature comparison of ARTIOS with VxWorks , a commercial RTOS.

	VxWorks	ARTIOS
Development Methodology	Host ≠ Target	Host ≠ Target
POSIX Compatibility	Most 1003.1b	Native C API
File Systems	MS-DOS, RT-11, raw disk, SCSI, CD-ROM	Not supported
Shared Memory Objects	VxMP (semaphores, message queues, memory regions between tasks on different processors)	semaphores, message queues, data pointer passing between tasks
Virtual Memory	VxVMI – support for boards with MMU	Not supported
Debugging	Target Agent – remote debugging	Remote debugging,
Multitasking	Process-task based Interrupt-driven, priority-based task scheduling, round-robin 256 priority levels (0-255); 0 highest, 255 lowest May be set dynamically	Process-task based Interrupt-driven, priority-based task scheduling, round-robin 256 priority levels (0-255); 0 highest, 255 lowest May be set dynamically
Semaphores	Counting, binary, mutual-exclusion (recursive), and POSIX, timeouts	Counting semaphores. Max count limitation if enabled. Timeout support can be added easily.
Intertask Communications	Message queues (priority or FIFO), pipes, sockets, signals, RPCs	Message queue, task data, signals
Priority inversion	Yes – tasks that owns a resource executes at the priority of the highest priority task blocked on that resource	Yes – receiving processes will float to higher level priority of sending processes
States	Ready – Not waiting for any resource except CPU	Wait state consists of substates, that can be used

	Pend – Blocked, waiting on resource Delay – Asleep for some duration Suspend – Unavailable for execution Delay + S – Delayed & suspended Pend + S – Pended & suspended Pend + T – Pended with timeout value Pend + S + T – Pended, suspended with timeout value State + I – state plus an inherited priority	single of in combination. See Figure 6 for details.
Interrupts	Run in special context outside of task (no task context switch) Share single stack if architecture allows (must be large enough for all possible nested interrupt combinations) Must not invoke routines that may cause the caller to block (taking semaphores, I/O calls, malloc, and free) Must not call routines that use FP-coprocessor (floating point registers are not saved and restored); otherwise must explicitly save and restore FP registers	The same

Table 1. VxWorks vs. ARTIOS comparison

The static structure of ARTIOS makes it deterministic. This means that all the API functions complete their task within a certain amount of time.

The portability goal is achieved by using no CPU dependent line in the source code.

ROM requirement of the kernel, when all of the features are disabled, is less than 4 Kbytes. And the required RAM space for a two task system with all the features disabled is around 40 bytes.

When all the options are enabled, the ROM requirement is still less than 10K. These values are valid when compiled with Borland C++ 5.01 compiler for DOS mode and may change according to the cpu and the compiler used. Figure 9 shows a sample compiler output screenshot with all the options enabled.

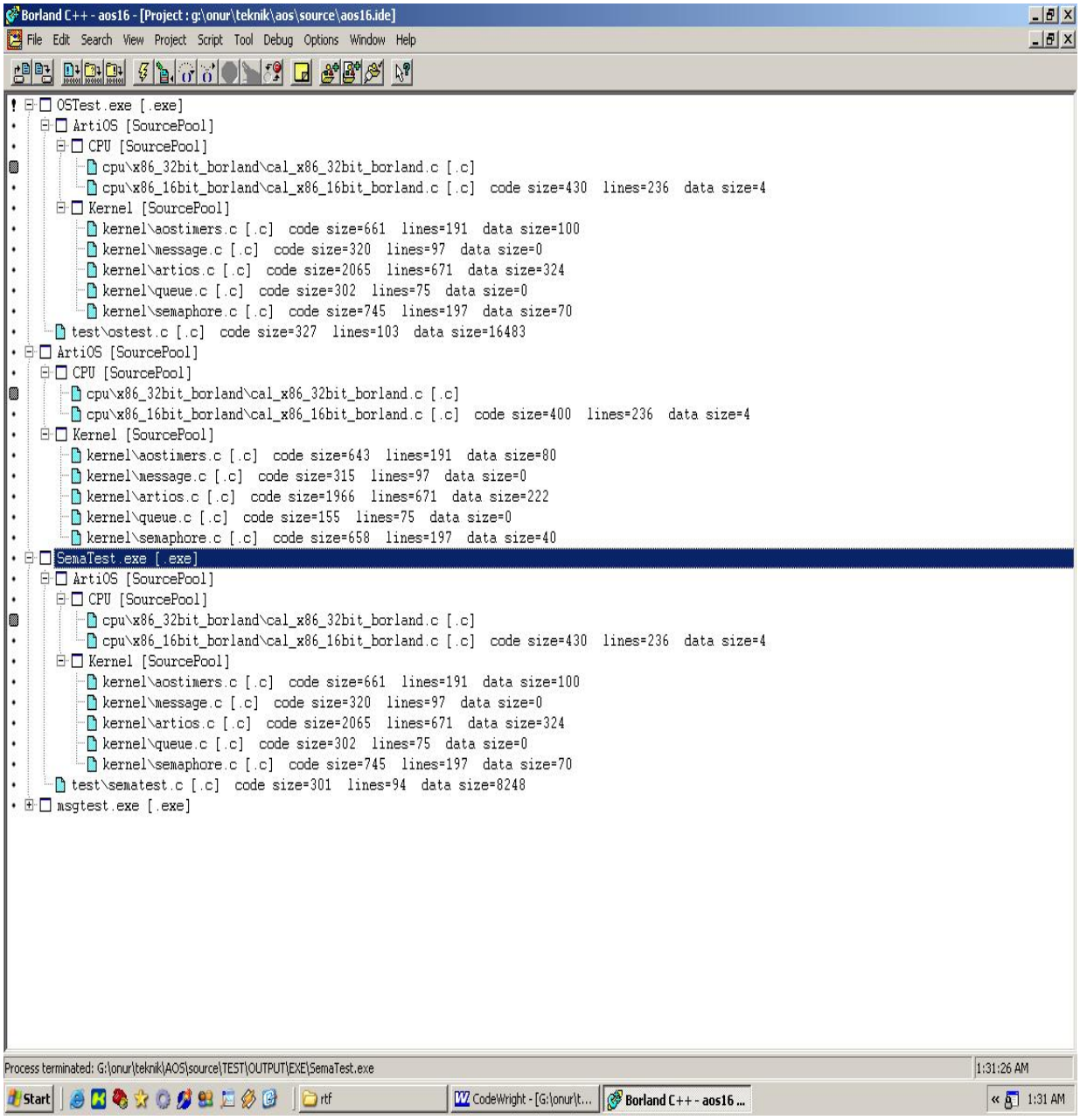

Figure 9. Sample compiler output

5. Conclusion

We developed a portable real-time embedded operating system in the scope of this thesis. The main targets were portability, scalability and extensibility. And a reasonably readable source code implementation was achieved with optimal modularity and with the use of C macros. One can easily add new features and change the existing ones. The system performance and memory foot prints can be increased by analyzing the code deeply.

Some of the future works might be listed as follows for the ones, who want to develop this system further.

- The AOS state machine was designed to support the timed wait mechanism for semaphores and other signals. This feature can be implemented.

- The code can be optimized more in terms of code size and memory usage.

- Real-time analysis and interrupt performance tests can be performed on AOS and the response of the AOS can be improved.

- New API calls can be defined and added, like mutexes and binary semaphores, which are subsets of existing semaphores.

- Currently AOS does not support any generic operating system interface standard. An adaptation layer could be written to achieve this. For example one can write an adaptation layer for POSIX compliance.

- New CPU adaptation layers could be implemented to port AOS to different platforms.

REFERENCES

URLs

Embedded Linux Howto
(http://linux-embedded.org/howto/Embedded-Linux-Howto.html)

Linux, Real-Time Linux, & IPC
(http://www.ddj.com/documents/s=897/ddj9911b/9911b.htm)

POSIX thread API concepts
(http://as400bks.rochester.ibm.com/pubs/html/as400/v5r1/ic2924/index.htm?info/apis/rzah4mst.htm)
.
Product Standard: Multi-Purpose Realtime Operating System
(http://www.opengroup.org/branding/prodstds/x98rt.htm)

Real-time FAQ (http://www.realtime-info.be/encyc/techno/publi/faq/rtfaq.htm)

Real-time Linux FAQ (http://www.rtlinux.org/rtlinux.new/documents/faq.html)

The IEEE Computer Society Real-time Research Repository
(http://cs-www.bu.edu/pub/ieee-rts/Home.html)

Arcamano, Roberto (2002):*Kernel Analysis-HOWTO*
(http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html)

Hyde, Randall (1997): *Interrupts on the Intel 80x86*
(http://www.ladysharrow.ndirect.co.uk/library/Progamming/The%20Art%20of%20Assembly%20Language%20Programming/)

Locke, Doug(2002): *Priority Inheritance: The Real Story,*
(http://www.linuxdevices.com/articles/AT5698775833.html)

Yodaiken, Victor (2002): *Against priority inheritance,*
(http://www.linuxdevices.com/files/misc/yodaiken-july02.pdf)

Articles and books

Barr, Michael (1999): *Programming embedded systems in C and C++*, O'Reilly.

Burns, Alan (2001): *Real-time systems and Programming Languages*, Addison-Wesley.

Buschmann, Frank (1996): Pattern-oriented software architecture: a system of patterns, Wiley Chicester.

Dijkstra, Edsger Wybe (1965): "Cooperating sequential processes", 43–112, 1968, *Programming Languages*, Academic Press.

Gamma Erich (1994): Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley.

Hansen, Per Brinch (1973): "Concurrent Programming Concepts", 223–245, 5, 4, 1973, *ACM Computing Surveys*

Herlihy, M. (1991): "Wait free Synchronization", 124–149, 13, 1, 1991, *ACM Transactions on Programming Languages and Systems*.

Lewine, Donald (1991): POSIX Programmer's Guide: Writing Portable UNIX Programs, O'Reilly.

McDonnell, Michael D. (2004): *An Introduction to Real-time Programming Using the Tics RTOS*, Tics Real-time

Sakamura, Ken (1998): “µITRON 3.0: An Open and Portable Real-Time Operating System for Embedded Systems”, IEEE Computer Society.

Simon, David E. (1999): *An Embedded Software Primer*, Addison-Wesley.

Walmsley, Mark (2000): *Multi-threaded programming in C++*, Springer.

APPENDIX

Real-time and Embedded Definitions

active task

Refers to the task that is currently running. Multi-tasking kernels need to know the identity of the currently running task so that it can know where to save the task's context when the kernel performs a task switch. For example, a global variable called ActiveTcb might point to the tcb of the active task. The kernel would save the active task's context on the active task's stack (i.e. the current stack)

application

Another name for program.

assert, asserted

Designates the active state of an electrical signal. For example, raising a normally 0 volt CPU input control pin to 5 volts *asserts* the line. Similarly, pulling down a normally 5 volt CPU input control pin to 0 volts *asserts* the line. In either of the above examples, the line is said to be *asserted*. When an input put is asserted, it performs its function. For example, asserting the interrupt pin on the 8086 generates an interrupt.

block, blocked task

A task is blocked when it attempts to acquire a kernel entity that is not available. The task will suspend until the desired entity is available. A task is blocked when: it attempts to acquire a semaphore that is unavailable, or when it waits for a message that is not in its queue, or while it waits for a *pause* to expire, or when, in a time-slicing environment, its time-slice has expired, to give a few examples.

break, breakpoint

A program breaks when a breakpoint is reached. A breakpoint is an instruction address at which a debugger has been commanded to stop program execution, and execute the debugger so that the user can enter debugger commands. At some point after breaking the user typically executes a debugger command like "continue" which continues program execution from the point at which the program was stopped.

busy waiting

The process by which code repetitively checks for a condition. The following code exhibits busy waiting.

```
while (TRUE)  {
        if (switchDown) {
                lampOn();
                break;
        }
```

```
}
```

The code stays in the loop until the condition is met. Contrast with event driven code.

context, context switching

A CPU's context typically refers to all its registers (including IP and SP) and status register(s). When a task switch occurs, its context is saved, and the context of the next task to run is restored. The saving of the current task's context and the restoring of the context of the next task to run is called a context switch. (Special Note: In a more complete sense context also includes the task's local variables and its subroutine nesting information (for example, the task may be 7 subroutine calls deep when the context switch is made, and when the task is eventually resumed the task's code must be able to return from all 7 subroutines). The local variables and subroutine nesting level are preserved by giving each task its own private stack when the task is created, since subroutine return information and local variables are stored on the stack.).

cooperative scheduling

A group of tasks run *cooperatively* when each must voluntarily relinquish control so that other tasks can run. In other words, a task will run forever, thus starving other tasks, unless it voluntarily gives up control. Control can be relinquished explicitly by a *yield*, *pause*, or *suspend*; or implicitly by waiting for an event. Contrast with priority based preemptive scheduling and time-slicing.

crash

A system is crashed when the CPU has halted to due a catastrophic error, however, the word can be used in the same sense has hung.

cyclical executive, cyclical kernel, cyclical scheduling

See also the article A Cyclical Executive for Small Systems. Cyclical (also referred to as round robin) scheduling is done without a kernel per se by simply calling one "task" (typically a C function) after another in an infinite loop as shown below.

```
void main(void)
{
  for (;;) {
            task0();
            task1();
            task2();
            /* and so on... */
  }
}
```

In this scenario, each task must do its work quickly, save its state if necessary, and return back to the main loop. The advantages with this approach are small size, no kernel required, requires only 1 stack, and easy to understand and control. The disadvantages are no priorities, tasks must retain their states, and more responsibility is placed on the programmer and the possibility of excessive loop latency when there are many tasks.

debugger

A debugger is a software program used to break program execution at various locations in an application program after which the user is presented with a debugger command prompt that will allow him to enter debugger commands that will allow for setting breakpoints, displaying or changing memory, single stepping, and so forth.

deadlock

The term can have various meanings, but typically, a deadlock is a condition in which a task will remain forever suspended waiting for a resource that it can never acquire. Consider a system comprised of a keyboard and a display, with a separate semaphore for each. In order for a task to interact with the user it must acquire the "console" (keyboard and display) and therefore must acquire both the keyboard and the display semaphores. If more than one task decides to interact with the user at the same time a condition can arise where taskA acquires the keyboard semaphore and taskB acquires the display semaphore. Now taskA will wait forever for the display semaphore and taskB will wait forever for the keyboard semaphore. The solution in this example is to treat the keyboard and display as a single resource (console). Deadlocks can occur for a variety of reasons, but the end result is typically the same: the task will wait forever for a resource that it can never acquire.

differential timer management

A method of managing timers such that only 1 timer count is decremented regardless of the number of pending timers. Consider 3 pending timers with durations of 50 ms, 20 ms, and 5 ms respectively. The timers are sorted and stored in a list with the lowest timer first: 5, 20, and 50. Then the timer durations are replaced with the difference of the timer duration and the preceding duration, i.e., 5, 20, 50, is replaced with 5, (20-5), (50-20), which equals 5, 15, 30. This allows for only the first duration (5 ms in this example) to be decremented. Thus, when the first duration of 5 decrements down to 0, there are only 15 ms required to meet the 20 ms duration, and when the 50 ms duration is to be started, already 5 + 15 = 20 ms have expired, so only a count of 30 ms is required. This technique is attractive because the timer counts are typically decremented inside the timer isr and time inside any isr should be kept to a minimum.

download

Refers to the transfer of executable code from a host to a target, typically using an RS-232 serial line. The target must have resident software (e.g., EPROM) that can read the incoming data, translate it if necessary (the file format may be ASCII hex for example) and load and run the code. If the target board has no resident software, then an ICE is required. The ICE connects to the host via RS-232 typically and accepts the download and loads the code into memory.

dynamic priorities

Priorities that can be changed at run-time. Contrast with fixed priorities.

event, event driven code

Code that remains dormant until an event occurs. The following is an example.

```
while (TRUE) {
        msg = waitMsg(MOTOR_ON);
```

```
        freeMsg(msg);
        turnMotorOn();
    }
```

Until the message MOTOR_ON is received, the function *waitMsg* will suspend the current thread of execution until the message is received. The receipt of the message is the *event* that causes the task to be scheduled and eventually run. Contrast with busy waiting.

exception

A software interrupt generated by the CPU when an error or fault is detected.

fairness

A concept which dictates that if a task runs too long, its priority is lowered so that other tasks can get their fair share of the CPU. This concept is repugnant to real-time principles and is used chiefly in multi-user systems with the intent of disallowing one user to monopolize the CPU for long compiles and the like.

fault

A fault is an exception that is generated when the current instruction needs help before it can be executed. For example, supposed in a virtual memory system, a memory access is made to a page that is not in memory. In this case, a fault is generated which vectors to an isr that will read in the page. The fault exception is different in the way it returns from interrupt in that control is returned to the instruction that caused the fault, not the following instruction as with a normal interrupt. This allows the instruction to access the memory again, and this time succeeds.

fixed block memory allocation

A memory allocation/deallocation scheme in which a number of memory pools of fixed sized blocks is created. For example, 3 pools are created as follows: pool 1 contains 100 64 byte blocks, pool 2 contains 50 256 byte blocks, and pool 3 contains 10 1K byte blocks. When a user wants memory he takes and returns blocks from/to a pool. For example, if the user needed a 256 byte block of memory he would take a block from pool 2. Similarly, if he needed a 100 byte block he would still have to take a block from pool 2 even though bytes are wasted. The advantage is no fragmentation and high speed.

fixed priorities

Priorities are set once, typically at compile time, and unalterable at run-time. Contrast with dynamic priorities.

fragmentation

In a kernel or language which allows for memory allocation and deallocation, the available free memory can eventually become fragmented, i.e., non-contiguous, if memory is allocated from one large pool. For example after allocating and deallocating many different size blocks of memory, there may be 12K available, but it is not contiguous, and therefore, if software needs 12K it cannot use it, even though it is available. Contrast with fixed block memory allocation.

hang, hung

A system is hung when it does not respond to external input (e.g. keyboard) but has not completely crashed. A hung system is typically spinning in an endless loop.

hard real-time

Hard real-time refers to the strict definition of real-time. See real-time.

hardware interrupt latency

The time it takes for the isr to be entered, once the processor interrupt pin is asserted.

hook, chain

Intercept an interrupt by saving the current interrupt vector and writing a new interrupt vector. When the new interrupt service routine is finished it calls the old interrupt service routine before returning.

host

See target.

in-circuit emulator, ICE

An electronic tool that allows for debugging beyond the capabilities of a standard software debugger. An ICE is essentially a hardware box with a 2 to 3 foot cable attached to it. At the end of the cable is a multi-pin connector connected to a CPU processor chip which is identical to the processor on the target board. The target processor is removed from your target board, and the connector plugged in. The ICE allows for trapping the following types of activities: read/write to an address range, read/write a specific value from an address range, setting a breakpoint in EPROM, mapping emulator memory to target board memory, and other similar features. It also turns the host (a PC for example) into a debug station with supplied software. This allows for debugging of the target board even though the target does not have a keyboard, screen, or disk. For more information, see articles 1, 2, and 3.

idleTask

A kernel owned task that is created and scheduled during system initialization. It runs at the lowest possible priority. The idle task has the lowest possible priority and runs only when no other tasks are scheduled. When any other task is scheduled, the idle task is preempted. The idle task is typically a "do nothing" tight loop. Its only purpose is to run when no other tasks run. It is a convenience mechanism in that no special kernel code is required to handle the case of all tasks being idle; the kernel sees the idle task like any other task. The key is that the idle task has a lower priority than *any* other task, and even though it is scheduled to run and occupies a place in the ready queue, it will not run until all other tasks are idle. And, conversely, when any other task is scheduled, the idle task is preempted.

instantiate, instantiation

To start multiple instances of a task.

intercept

Identical to hook except that the old isr is not called before returning.

interrupt, interrupt vector

Most CPU's have a pin designated as the "interrupt" pin. When this pin is asserted the CPU (1) halts, (2) saves the current instruction pointer and the CPU flags on the stack, and (3) jumps to the location of an interrupt service routine (isr). how the address of the isr is determined is CPU dependent. Some CPU's have multiple interrupt pins, and each pin refers to a reserved memory location where the isr address can be found. For example, consider an imaginary 16 bit CPU with 8 interrupt pins I0 through I7, which refer to interrupt numbers 0 through 7, and an interrupt vector table starting at address 0. The addresses of the isr's for interrupt pins I0 through I7 would be located at addresses 0, 2, 4, 6, 8, 10, and 12. The dedicated interrupt pin approach works fine until more than 8 interrupts are desired. The 8086 handles this problem by relegating the handling of the interrupt signals with a special interrupt controller chip like the 8259A. The 8086 has only 1 interrupt pin. When the interrupt pin is asserted, the 8086 determines which device generated the interrupt by reading an interrupt number from the interrupt controller. The interrupt number is converted to an interrupt vector table index by multiplying the interrupt number by 4 because each entry in the 8086 interrupt vector table requires 4 bytes (2 bytes for the CS register and 2 bytes for the IP register). Although the 8259A interrupt controller handles only 8 incoming interrupt lines, the 8259A's can be cascaded to handle more interrupts.

interrupt controller

(The following explanation uses the 8259A as an example interrupt controller). Sometimes many interrupts are required. Most CPU's with dedicated interrupt pins have no more than 8 interrupt pins, so that no more than 8 incoming interrupts can be handled. A way around this problem is to handle interrupts with an interrupt controller chip. Typically interrupt controller chips can be cascaded so that even though a single interrupt controller chip handles only 8 incoming interrupts, a cascade of 4 can handle 32 incoming interrupts. The typical arrangement is for the hardware device to signal an interrupt by asserting one of its output pins which is directly connected to one of the interrupt controller's interrupt pins. The interrupt controller then signals the CPU by asserting the one and only CPU interrupt pin. The CPU then begins a special interrupt bus cycle with the interrupt controller and obtains the interrupt number which is converted into an index into an interrupt vector table, which enables the CPU to jump to the proper isr for the interrupt. If other interrupts are asserted while the CPU is handling an interrupt, the interrupt controller waits until it receives a signal from the CPU called and EOI (end of interrupt) before interrupting the processor again. The EOI signal is generally asserted with a software IO output instruction. Note however, that if more than one interrupt is generated on a single interrupt controller pin while the CPU is processing an interrupt, then all but the first waiting interrupt will be lost. For more information see interrupts.

interrupt service routine, isr

A routine that is invoked when an interrupt occurs.

interrupts and CPU flags register, IRET instruction

It is necessary to save the flags on entering an isr and restore them on exit; otherwise, the isr may change the flags, which can cause problems. The 8086

automatically saves flags on the stack prior to calling the isr, and restores the flags before returning using the IRET instruction. The reason for saving the flags can be understood by considering the following code fragment.

```
; Assume that AX = 0.

CMP AX,0
JE LABEL7
```

The CMP instruction executes, and because AX = 0, the Z bit in the flags register is set by the CPU. Let us say that after the compare instruction, an interrupt occurs and the isr is jumped to which executes the following instruction.

```
; Assume BX = 1.

CMP BX,0
```

The above instruction will clear the Z bit because BX is not 0. The isr will eventually return to the instruction above JE LABEL7. If a standard RET instruction is used instead of IRET, the flags will not be restored, and the instruction JE LABEL7 will not perform the jump as it should because the Z bit was cleared in the isr. For this reason flags must be saved before entering an isr and restored prior to returning from an isr.

interrupt vector table, interrupt service routine (isr), software interrupt

Some CPU's handle interrupts with an interrupt vector table. On the 8086, the interrupt vector table is a 1K long table starting at address 0, which will support 256 interrupts numbered 0 through 255. The numbers 0 through 255 are referred to as "vector numbers" or "interrupt numbers". The table contains addresses of interrupt service routines, which are simply subroutines that are called when the interrupt occurs. The only difference between an interrupt service routine and a subroutine is that on entry to the isr the CPU flags are stored on the stack in addition to the return address; for an explanation of why the flags must be saved, see the definition. This means that an interrupt service routine must return with a special return instruction (IRET on the 8086) that pops the CPU flags off the stack and restores them before returning. On the 8086, an address is specified by a segment and an offset, and therefore each interrupt vector table entry uses 4 bytes; two bytes for the CS register and 2 bytes for the IP register (IP is the lower word in the vector table and CS is the higher word). An interrupt service routine can be invoked by software using the 8086 INT instruction, which in addition to the return address pushes the CPU flags onto the stack. An interrupt service routine can also be invoked by software using a standard CALL instruction as long as the CPU flags are first pushed onto the stack. For details of how an interrupt is generated and serviced, see interrupts.

kernel, multi-tasking kernel, real-time kernel

Software that provides interfaces between application software and hardware and in general controls and programs all system hardware. A multi-tasking kernel provides for multi-tasking in the form of threads, processes, or both. A real-time kernel provides is a multi-tasking kernel that has predictable worst case tasks switch times and priorities so that high priority events can be serviced successfully.

monitor

Refers to a kernel, or a very minimal kernel used for debugging, typically written by the programmer for a particular project, or supplied on ROM from the board manufacturer.

message

A data structure that is passed to a task by placing a pointer to the structure into a queue. The queue is a doubly linked list that is known to the task. Kernel functions are supplied to remove messages from the queue.

message aliasing

A technique by which the sender of the message changes the "sender" field in the message structure before sending it so that the receiver of the message will reply to a task other than the actual sender.

messages vs. mail

Messages are queued, mail is not. New mail data may optionally over-write old, which is advantageous for certain applications. Consider taskA that updates a display and receives the latest data on a periodic basis from taskB. Assume further that the system is busy to the point that taskB sends taskA two data updates before taskA can resume execution. At this point the old data is useless, since taskA only displays the latest value. In this situation, queuing (using messages) is undesirable, and mail with over-write enabled is preferred.

multi-processing

A term that describes an environment in which 2 or more CPU's are used to distribute the load of a single application.

multi-tasking

A term that describes an environment in which a single computer is used to share time with 2 or more tasks. Contrast with multi-user. For more information and implementation details see articles 1, and 2.

multi-user

A term that describes a single computer that is connected to multiple users, each with his own keyboard and display. Also referred to as a time-sharing system. Contrast with multi-tasking.

non-reentrant, reentrant code

Non-reentrant code cannot be interrupted and then reentered. Reentrancy can occur for example, when a function is running and an interrupt occurs, which transfers control to an interrupt service routine, and the interrupt service routine calls the function that was interrupted. If the function is not reentrant, the preceding scenario will cause problems, typically a crash. The problem typically occurs when the function makes use of global data. To illustrate, consider a function that performs disk accesses and uses a global variable called "NextSectorToRead", which is set according to a parameter passed to the function. Consider the following scenario. The function is called with a parameter of 5, and "NextSectorToRead" is set to 5. The function is then interrupted and control is transferred to an isr which calls the function with a parameter of 7, which results in "NextSectorToRead" being set to 7. The sector

is read, control is returned to the isr, the isr performs a return from interrupt, and the function resumes with an erroneous value of 7 instead of 5 in "NextSectorToRead". This problem can be avoided by only using stack variables.

page

A unit of memory, typically around 4K, which is used as the smallest granule of memory in virtual memory systems.

preemption - causes of

Preemption occurs when a dormant higher priority task becomes ready to run. A dormant higher priority task can become "ready to run" in a variety of ways. If time-slicing is enabled, (which is rarely the case in properly designed real-time systems), preemption occurs when the task's time-slice has expired. Preemption can also occur when a message is sent to a task of a higher priority than the active task. Consider the following. The active task, taskB, invokes *waitMsg* to wait for a particular message named START. Since the message START is not in taskB's message queue (i.e., no task has sent taskB a message named START), taskB will be suspended until such time as the message START is received. Once taskB is suspended, the highest priority waiting task, (the task at the front of the ready queue), is removed from the ready queue, (call it taskA which is of a lower priority than taskB) and it becomes the new active task. Now if taskA sends taskB a message named START, the kernel will suspend taskA, and resume taskB since taskB has a higher priority *and* it now has a reason to run (it received the message that it was waiting for). Task preemption can also occur via interrupt. If a message is sent from within an isr to a task with a higher priority than the active task, (which was interrupted to run the isr), then on exiting the isr, control should not return to the interrupted task, but rather to the higher priority task to which the message was sent.

preemptive, preemption

Preemption occurs when the current thread of execution is stopped and execution is resumed at a different location. Preemption can occur under software (kernel) or hardware (interrupt) control. See priority based preemptive scheduling.

priority

A number attached to a task that determines when the task will run. See also scheduling.

priority based preemptive scheduling

A multi-tasking scheduling policy by which a higher priority task will preempt a lower priority task when the higher priority task has work to do. Contrast with cooperative scheduling and time-sliced scheduling.

priority inversion

An inappropriately named condition created as follows. Consider three tasks: taskA, taskB, and taskC, taskA having a high priority, taskB having a middle priority, and taskC having a low priority. taskA and taskB are suspended waiting for a timer to expire, and therefore taskC runs. taskC then acquires semaphoreA, and is subsequently preempted by taskA before it releases semaphoreA. taskA

then attempts to acquire semaphoreA, and blocks since the semaphore is in use by taskC. taskB now runs, because it is of a higher priority than taskC and its timer has expired. Now, taskB can run until it decides to give up control, creating a condition where a high priority task (taskA) cannot run because a lower priority task has a resource that it needs. taskA will remain blocked until the lower priority task runs and releases the semaphore. This situation can be avoided by proper coding, or by using a server task instead of a semaphore.

process

A process is a single executable module that runs concurrently with other executable modules. For example, in a multi-tasking environment that supports processes, like OS/2, a word processor, an internet browser, and a data base, there are separate processes and can run concurrently. Processes are separate executable, *loadable* modules as opposed to threads which are not loadable. Multiple threads of execution may occur within a process. For example, from within a data base application, a user may start both a spell check and a time consuming sort. In order to continue to accept further input from the user, the active thread could start two other concurrent threads of execution, one for the spell check and one for the sort. Contrast this with multiple .EXE files (processes) like a word processor, a data base, and internet browser, multi-tasking under OS/2 for example.

program space

The range of memory that a program can address.

protection violation

Through a scheme involving special hardware and kernel software, tasks can be restricted to access only certain portions of memory. If a program attempts to jump to a code location outside its restricted area or if it attempts access data memory outside its restricted area, a *protection violation* occurs. The protection violation generates an interrupt to an interrupt service routine. The interrupt service routine will typically terminate the task that caused the violation and schedule the next ready task. When the interrupt service routine returns, it returns to the next ready to run task. Special hardware is needed because each code or data access must be checked. This is essentially done through special hardware tables that kernel software can write to. For example, when a task is created, the kernel assigns a hardware table to the task (each task is assigned a different hardware table). The kernel then writes into this table the lower limit and the upper limit of memory that the task is allowed to access. When a task starts, a special hardware register is set to point to the hardware table for the task so that the special memory management hardware (which may be on the CPU chip or external to it) will know which table to use when making access checks. Each time the task accesses memory, the memory management hardware checks the requested memory access against the allowable limits, and generates a protection violation if necessary.

real-time

In a strict sense, real-time refers to applications that have a time critical nature. Consider a data acquisition and control program for an automobile engine. Assume that the data must be collected and processed once each revolution of the engine shaft. This means that data must be read and processed before the

shaft rotates another revolution, otherwise the sampling rate will be compromised and inaccurate calculations may result. Contrast this with a program that prints payroll checks. The speed at which computations are made has no bearing on the accuracy of the results. Payroll checks will be generated with perfect results regardless of how long it takes to compute net pay and deductions. See also hard real-time and softRealtime.

real-time kernel

A real-time kernel is a set of software calls that provide for the creation of independent tasks, timer management, inter-task communication, memory management, and resource management.

real-time Operating System

A real-time kernel plus command line interpreter, file system, and other typical OS utilities.

relinquish, voluntarily relinquish control

The process by which the active task voluntarily gives up control of the CPU by notifying the kernel of its wish to do so. A task can voluntarily relinquish control explicitly with a yield, pause, or suspend; or implicitly with waitMsg, waitMail, etc. and the actual system calls will vary from kernel to kernel. When the active task relinquishes control, the task referenced by the node at the front of the ready queue becomes the new active task.

ready queue

The ready queue is a doubly linked list of pointers to tcb's of tasks that are waiting to run. When the currently active task relinquishes control, either voluntarily, or involuntarily, (for example by an interrupt service routine which schedules a higher priority task), the kernel chooses the task at the front of the ready queue as the next task to run.

real-time responsive

Describes a task switching policy that is consistent with real-time requirements. Specifically, when ever it is determined that a higher priority task needs to run, it runs immediately. Contrast this with a policy in which the higher priority task is *scheduled* when it is determined that it must run, but the active lower priority task continues to run until a system scheduler task interrupts it and then switches to the higher priority task.

resource

A resource is a general term used to describe a physical device or software data structure that can only be accessed by one task at a time. Examples of physical devices include printer, screen, disk, keyboard, tape, etc. If, for example, access to the printer is not managed, various tasks can print to the printer and inter-leave their printout. This problem is typically handled by a server task whose job is to accept messages from various tasks to print files. In this way access to the printer is serialized and files are printed in an orderly fashion. Consider a software data structure that contains data and the date and time at which the data was written. If tasks are allowed to read and write to this structure at random, then one task may read the structure during the time that another task is updating the structure, with the result that the data may not be time stamped correctly.

This type of problem may also be solved by creating a server task to manage access to the structure.

resume

To run again a suspended task at the point where it was suspended. Formally, the process by which the active task is suspended and a context switch is performed to the previously suspended task.

round robin, round robin scheduling

A multi-tasking scheduling policy in which all tasks are run in their turn, one after the other. When all tasks have run, the cycle is repeated. Note that various scheduling policies are run in round robin fashion, e.g., cooperative scheduling, time-sliced scheduling, and cyclical scheduling. Cyclical and round robin scheduling are used inter-changeably.

dynamic scheduling

A scheduling mechanism that allows for creating and scheduling new tasks and changing task priorities during execution. See also scheduling and static scheduling.

static scheduling

A scheduling mechanism in which all tasks and task priorities are described and bound at compile-time; they cannot be changed during execution. See also scheduling and dynamic scheduling.

schedule, scheduling

A task can be in any of the following states: (1) dormant (inactive, usually waiting for an event - a message for example), (2) waiting to run (a task that was non-existent or dormant was *scheduled* to run), (3) running (on a single CPU system, only one task at a time can be in the *running* state. A dormant task is scheduled by notifying the kernel/OS that the task is now ready to run. There are various scheduling policies: see cooperative scheduling, time-sliced scheduling, cyclical scheduling, and priority based preemptive scheduling. For software implementation details see the article How Tasks Work - Part 2.

semaphore

This term has several meanings. In its simplest form, it is a global flag - a byte of global memory in a multi-tasking system that has the value of 1 or 0. In multi-tasking systems, however, a simple global flag used in the normal manner can cause time dependent errors unless the flag is handled by a special test and set instruction like the 80x86 XCHG instruction. A semaphore is typically used to manage a resource in a multi-tasking environment - a printer for example. See XCHG for a code example. In another form, the semaphore may have a count associated with it, instead of the simple flag values of 1 or 0. When the semaphore is acquired, its count is made available to the caller, and can be used for various purposes. For example, consider a system that has three printers numbered 1, 2, and 3. A semaphore could be created with a count of 3. Each time the semaphore is acquired its count is decremented. When the count is 0, the semaphore is not available. When the semaphore is released, its count is incremented. In this way a pool of 3 printers can be shared among multiple tasks. The caller uses the semaphore count value returned to him to determine

which numbered resource (e.g., printer 1, 2 or 3) he is allowed to use. Numerous variations on this theme can be created. Compare also with server task. The advantage of a server task is that it does not block the calling task if the resource is not available, thus avoiding priority inversion. In general semaphores should be avoided; furthermore they are unnecessary in a message based OS. The server task approach to resource management is preferred over semaphores as it is cleaner and less error prone.

serial line analyzer, datascope
A piece of test equipment that is connected in series with a serial line for the purpose of displaying and monitoring two way serial communications. The term usually refers only to RS-232 serial lines.

server task
A task dedicated to the management of a specific resource. A server task accepts requests in the form of messages from other tasks. For example, a server task could be created to manage access to a single printer in a multi-tasking environment. Tasks that require printing send the server a message that contains the name of the file to print. Note that the server task approach does not block the calling task like a semaphore will when the resource is unavailable; instead the message is queued to the server task, and the requesting task continues executing. This is an important consideration when is a concern.

single step
A debugger command that allows an application program to execute one line of the program, which can either be a single assembly language instruction, or a single high level language instruction. There are typically two distinct single step commands - one that will single step "into" subroutine calls, and one that will step "across" them (i.e., enter the routine, but do not show its instructions executed on the screen). The latter command is useful, otherwise many unwanted levels of subroutines would be entered while single stepping.

soft real-time
Refers to applications that are not of a time critical nature. Contrast with hard real-time. See also real-time.

stress testing
The process by which a software system is put under heavy load and demanding conditions in an attempt to make it fail.

subroutine nesting, subroutine nesting level
Refers to the scenario in which subroutineA calls subroutineB which calls subroutineC... which calls subroutineN. This is relevant for real-time systems because if a task is 7 subroutines deep, and the task is preempted, the return addresses must be preserved so that when the task is later resumed, it can return back up the chain. The most practical way of preserving subroutine nesting is by assigning each task its own stack. This way when a task is resumed, the SP is first set to the task's stack.

suspend, suspension
The immediate cessation of a thread, preceded by the saving of its context.

synchronization

Sometimes one task must wait for another task to finish before it can proceed. Consider a data acquisition application with 2 tasks: taskA that acquires data and taskB that displays data. taskB cannot display new data until taskA fills in a global data structure with all the new data values. taskA and taskB are typically *synchronized* as follows. (1) taskB waits for a message from taskA, and since no message is available, taskB suspends. When taskA runs and updates the data structure, it sends a message to taskB which schedules taskB to run, at which time it displays the new data, then again waits for a message, and the scenario is repeated.

target, target board, host

Refers to the situation where two computers are involved in the software development process - one computer to develop software (edit, compile, link, etc.), referred to as the host and one computer to run the software referred to as the target. The target is the actual product on which the software is to run. In most common situations, the host and target are the same. For example, a word processor is developed on the PC and runs as a product on the PC. However, for various real-time and embedded systems, this is not the case. Consider a small single board computer that controls a robot arm. The software cannot be developed on this board because it has no keyboard, display, or disk, and therefore, a host computer, like a PC, is used to develop the software. At some point, when it is believed that the software is ready for testing, the software is compiled and linked to form an executable file, and the file is downloaded to the target, typically over an RS-232 serial connection. Debugging on the target is typically done with an ICE.

task

A thread of execution that can be suspended and later resumed.

task collisions, time dependent error

Data corruption due to preemption. Consider a preemptive multi-tasking environment in which tasks use a global flag to gain access to a printer. The following code will not have the desired effect.

```
tryAgain:
  CMP flag,1        ; Printer available?
  JE tryAgain       ; No, then try again.
  MOV flag,1        ; Printer was available. Set it as busy.
  ...use printer...
  MOV flag,0        ; Set printer as available.
```

The problem is that after the CMP instruction, the current task can be preempted to run another task which also checks the flag, sees that it is available, sets the flag, does some printing, and is then preempted to run the original task, which resumes at the JE instruction. Since the 80x86 *flags* register (not to be confused with the *flag* variable), which contains the status of the last CMP instruction, is restored with the original task's context, the JE will fail, since the flag was 0. The original task will now acquire the flag and interleave its printing with the other task's printing. This problem can be overcome by using a read-modify-write instruction like XCHG or by managing the printer with a server task. Time dependent errors can also occur when preemption is used with non-reentrant

code. Consider a graphics library that is non-reentrant. If taskA calls a non-reentrant graphics function and is then preempted to run taskB which calls the same function, errors will result. Sometimes the non-reentrancy problem is not so obvious. Consider a non-reentrant floating software library that is called invisibly by the compiler. For example the statement y = 5.72 * x; would make a call to the floating point software multiplication function. If the current task were preempted while executing this instruction, and the next task to run also performed a multiply, then errors can again result. Note that floating point time-dependent errors could still occur even if a floating point chip was used instead of a floating point software library. An error could occur if the floating point CPU's context is not saved by the kernel.

task control block, tcb

A task control block is a data structure that contains information about the task. For example, task name, start address, a pointer to the task's instance data structure, stack pointer, stack top, stack bottom, task number, message queue, etc.

task instance, generic task, instance data

A single generic task can be created to handle multiple instances of a device. Consider 5 analog channels that must be read and processed every nth millisecond, the number "n" being different for each channel. A generic task can be created that is channel independent through the use of instance data. Instance data is a C structure that in this case contains information specific to each channel, i.e., the IO port address of the channel, the sample interval in milliseconds, etc. The task code is written so that it references all channel specific data through the instance data structure. When the task is created a pointer to the task's task control block is returned. One of the fields of the task control block is a user available void pointer named "ptr". After starting the task, the "ptr" field is set to an instance data structure that contains the sample rate, IO port address, etc. for that task instance.

```
for (i = 0; i < 5; i++) {
  tcb = makeTask(genericAnalogTask, i);
  tcb->ptr = &InstanceDataArray[i];
  startTask(tcb);
}
```

The code above shows how 5 instances of the same task are created, and each given a unique identity via tcb->ptr which points to the instance data.

```
void genericAnalogTask(void) {
  typeAnalogData * d;

  d = (typeAnalogData *) ActiveTcb->ptr;

  while (TRUE) {
        pause(d->sampleIntervalInMs);
        readAndProcessChannel(d->channelIOPortAddress);
  }
}
```

The code above shows what the generic task might look like. For further information see the example.

thread

A thread is a task that runs concurrently with other tasks within a single executable file (e.g., within a single MS-DOS EXE file). Unlike processes, threads have access to common data through global variables.

thread of execution

A sequence of CPU instructions that can be or have been executed.

time-slice, time-slicing, time-sliced scheduling, time-sharing

Each task is allotted a certain number of time units, typically milliseconds, during which it has exclusive control of the processor. After the time-slice has expired the task is preempted and the next task at the same priority, for which time-slicing is enabled, runs. This continues in a round-robin fashion. This means that a group of time-sliced high priority tasks will starve other lower priority tasks. For example, in a 10 task system, there are 3 high priority tasks and 7 normal priority tasks. The 3 high priority tasks have time-slicing enabled. As each high priority task's time-slice expires, the next high priority task is run for its time slice, and so on. The high priority tasks are run forever, (each in its turn until its time-slice expires), and the low priority tasks will never run. If however, all high priority task waits for an event, (for example each pauses for 100 ms), then lower priority tasks can run. The behavior of tasks in a time-slicing environment is kernel dependent; the behavior outlined above is only one of many possibilities, but is typically the way time-sliced tasks should behave in a real-time system. Some kernels may implement the concept of fairness to handle this situation, however, fairness is repugnant to real-time principles as it compromises the concept of priority.

trace

An ICE command that will save the most recent "n" instructions executed. The trace can also be conditional, e.g., trace only those instructions that access memory between 0 and 1023.

trace buffer

A buffer in ICE memory that stores the last "n" instructions executed. It is useful while debugging as it shows a history of what has occurred.

virtual memory

A technique in which a large memory space is simulated with a small amount of RAM, a disk, and special paging hardware. For example, a virtual 1 megabyte address space can be simulated with 64K of RAM, a 2 megabyte disk, and paging hardware. The paging hardware translates all memory accesses through a page table. If the page that contains the memory access is not loaded into memory a fault is generated which results in the least used page being written to disk and the desired page being read from disk and loaded into memory over the least used page.

XCHG (80x86 XCHG instruction), protected flag, read-modify-write

The 80x86 XCHG instruction can be used in a *single processor*, preemptive multi-tasking environment to create a flag that is shared between different tasks. A simple global byte cannot be used as a flag in this type of environment since tasks may collide when attempting to acquire the flag. The XCHG instruction is typically used as follows:

```
tryAgain:
  MOV AL,1
  XCHG AL,flag
  CMP AL,0
  JNE tryAgain
```

flag is a byte in RAM. If the flag is 1, then the XCHG instruction will place a 1 into AL. If AL is 1 after the XCHG then the loop must continue until finally AL is 0. When AL is 0 the following is known: (1) The flag was 0 and therefore available, and (2) the flag is now 1 due to the exchange, meaning that the flag has been acquired. The effect is that with a single instruction, a task can check, and possibly acquire the flag, thus eliminating the possibility of a collision with another task.

When using multiple processors with the flag in shared RAM, the above solution will not work, since although the instruction is indivisible for the processor that executes it, it is not indivisible with respect to the shared RAM data bus, and therefore, the shared RAM bus arbitration logic must provide a method by which one processor can lock out the others while the XCHG executes. This is typically done with the 80x86 LOCK instruction as shown below.

```
tryAgain:
  MOV AL,1
  LOCK
  XCHG AL,flag
  CMP AL,0
  JNE tryAgain
```

Note that the LOCK instruction provides information only. It is up the shared RAM bus arbitration logic to incorporate the LOCK* pin into its logic. Newer processors like the 80386 incorporate the LOCK implicitly into the XCHG instruction so the LOCK instruction is unnecessary.

yield, yielding

The process by which a task voluntarily relinquishes control (via a kernel call) so that other tasks can run; the task will run again in its turn (according to its priority).