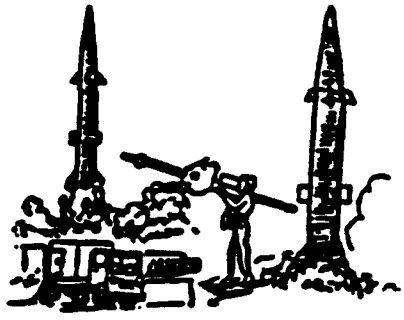
PROCEEDINGS OF

Technology Showcase

7-9 AUGUST 1990

Sponsored By

THE RESEARCH, DEVELOPMENT, AND
ENGINEERING CENTER



U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35898-5000

REAL TIME EXECUTIVE FOR MISSILE SYSTEMS (RTEMS)

Wanda M. Hughes and Phillip R. Acuff
Guidance & Control Directorate
Research, Development & Engineering Center
Redstone Arsenal, Alabama

ABSTRACT

A study was completed in 1988 which compared the various aspects of the Ada programming language as they relate to the application of Ada code in distributed and/or multiple processing systems. Several critical conclusions, which have a major impact on the way the Government develops software, were derived from the study. The two major conclusions were that the Ada programming language does not fully support multiprocessing and the run time executives being delivered with the Ada compilers were too slow and inefficient to be used in modern missile systems. Because of these shortfalls in the implementation of the Ada programming language for embedded systems, software developers in Army Research and Development and missile system prime contractors are purchasing and utilizing specialized third party kernel code to fill the void where Ada is lacking. The contractor, and eventually the Government, must pay a licensing fee for each copy of the kernel code used in an embedded system. The main drawback to this development environment is that the Government does not own, nor has the right to modify code contained within the kernel. Techniques for Verification and Validation (V&V) of software in this situation are more difficult than if the complete source code were readily available and could be modified.

INTRODUCTION

In 1974, the United States Department of Defense (DoD) realized that far too much money was being expended on software development and maintenance activities. As a result, a study was performed to determine how software costs were distributed over the various application areas. The study indicated that over half of all software expenditures were directly attributed to embedded systems [1]. In addition, the study concluded that two key factors were primarily responsible for these costs: an overabundance of programming languages and the "primitive" nature of those languages.

A proliferation of languages (more than 450) emerged as defense agencies and system project offices repeatedly spawned new languages from existing ones in an attempt to meet new project requirements. With numerous languages came numerous problems. For instance, languages were largely incompatible; They could not easily "talk" to each other. This computer "Tower of Babel" resulted in many costly mistakes. In one situation, an attempt was made to develop a tactical operations system that would use computers to assist battlefield commanders in making decisions. When this system was interfaced to other tactical systems (using different computer languages), translation was slow and error prone. The entire development program was halted after $100 million were expended. Another problem with numerous languages was that the software being developed was not portable. It could not be easily transported to different computers or projects. Similarly, software engineers could not transfer their skills across a broad spectrum of projects; rather, they had to become highly specialized. The use of so many languages also resulted in restricted competition in the maintenance and enhancement phases of projects as well as producing a minimal amount of software available in each language. Competition was restricted as competitors had to bear the initial investment associated with a new language, including the development and acquisition of both programmer training and support software. Obviously, the original developer of the software system would not incur these costs.

In addition to the overabundance of computer languages, the existing languages were inadequate because they were obsolete (they did not support modern software engineering principles). As a result, code generated in these languages was difficult to read and understand. In turn, these characteristics increased both design complexity and maintenance difficulty. The code lacked clear structure and contained low-level details that would have been hidden by a more modern language. These obsolete languages also resulted in software that was hard to reuse, because units of code were so interdependent. Modules often could not be extracted and used in different programs. This interdependency also made it difficult to modify code without introducing unwelcome and often "fatal" side effects.

All of these problems were especially severe in embedded systems. An embedded system is one in which a computer is part of a larger system, such as computerized radar used in aircraft. Embedded systems are typically complex real-time systems that contain many lines of code, are long lived, and are continually being modified. Because of the size, complexity, life-span, and volatility of these systems, they were responsible for 56% of DoD software costs in 1973.

The software issue, then, included such problems as software being excessively expensive, not portable, difficult to maintain, and not reusable. All of these problems were intensified in embedded systems. Clearly, something had to be done to help address this growing problem. The solution was the creation of a new standard software development language, Ada [2].

Ada was designed primarily for programming embedded computer systems. It is a real-time language and execution environment in that it not only contains a complete set of general purpose language features, but also provides facilities for multi-tasking, real-time synchronization, and direct programming of low level device hardware[3]. Ada offers many advantages over other programming languages. For instance, since no subsets or supersets are allowed, Ada source code may be ported between different Ada compiler systems, with minimal changes. In addition to code portability, people portability is enhanced. Programmers can move from project to project without having to learn new languages. Ada also helps manage the complexity and improve the maintainability of software by supporting modern software engineering principles [2]. This is accomplished by enforcing a strict programming discipline designed to make programs more readable, reliable, portable, modular, maintainable, and efficient – all qualities of good software [4].

It should be pointed out, however, that Ada is a large and complex language. The difficulties involved in learning how to use it effectively should not be underestimated [3]. In fact, the virtues of this language are still being tested within both the Government and in the private sector. In accordance with this, the study that was completed in 1988, comparing the various aspects of the Ada programming language as they related to the application of Ada code in distributed and/or multiple processing systems, brought to the forefront several critical conclusions which have a major impact on the way the Army develops application software for embedded systems. This impact applies to both in-house software development activities as well as contractor developed software. The MICOM/RDEC programs which have immediate impacts in this area include Non-Line Of Sight (NLOS/FOG-M), Army Unmanned Aerial Vehicle (AUAV), Multiple Launch Rocket System (MLRS), and Advanced Kinetic Energy Missile (AdKEM).

A conclusion of the analysis, which has been previously recognized by other agencies attempting to utilize Ada in a distributed or multiprocessing environment, is that the Ada programming language does not fully support multiprocessing (systems with more than 1 processor). Ada does provide a mechanism for multi-tasking, but this capability exists only for single processor systems. The language also does not have an inherent capability to access global named variables, flags or program code. These critical features are essential in order for data to be shared between processors. Although these drawbacks do have workarounds, they are sometimes awkward and defeat the real intent of a "self documenting" programming language, such as Ada.

However, the feature most essential for a distributed system is the capability to spawn tasks on another processor. This capability does not exist within the Ada language. Furthermore, there appears to be no suitable workaround within the language itself for this critical capability.

Another conclusion drawn from the analysis, was that the run time executives being delivered with the Ada compilers were too slow and inefficient to be used in modern missile systems. These run time executives are the core part of the Ada run time environment that provide operating systems functions such as task scheduling, input/output management and memory management.

REAL TIME SOFTWARE DEVELOPMENT

To properly evaluate the impact of these problems with the Ada language a thorough understanding of the "art" of developing effective real-time software is essential. Real-time embedded software development differs from other software development in that two computers, one for development and one for fielding, are typically used instead of one. Because embedded systems perform clearly defined, limited sets of functions, they often lack features needed for software development. They frequently have limited memory and disk storage, and often have a specialized interface to the external world, such as buttons, gauges, sensors, or actuators. Developing software for them usually requires a different computer with the necessary software development tools. This software development computer is called the host, while the embedded computer is called the target.

The implementation phase for embedded systems software consists of developing software on the host, and then downloading the software to the target for testing. When software errors are found, they are corrected on the host before downloading again to the target for more testing. This cycle continues until all the bugs are worked out. The software may then be burned into PROMS, if desired, and installed on the target system. Figure 1 illustrates this process.

Although Ada was designed with real-time embedded applications in mind, such implementations depend heavily on the interaction between the compiler, interface library, and the embedded system's kernel, or executive.

REAL-TIME EXECUTIVES

Real-time software development also differs from other software development by its very nature: in real-time, the right answer late is wrong. The system must respond to the unexpected events in the outside world rapidly enough to control ongoing processes. Special needs exist for an extensive set of support tools and technology to properly design and implement real-time software. Real-time design requires determinism, preemptive scheduling capabilities, real-time interrupt response with low interrupt latency, explicit tasking control, time-slicing, and prioritization of tasks to ensure reliability and predictability in a system's behavior [5].

Another key requirement is multitasking. Multitasking is the ability of the software to handle many tasks concurrently, because events in the real world usually overlap rather than occur in strict sequence.

The multitasking capability of a real-time operating system provides a framework that allows the design of very complex real-time software which has well-defined and controlled interactions among its various components. In addi-

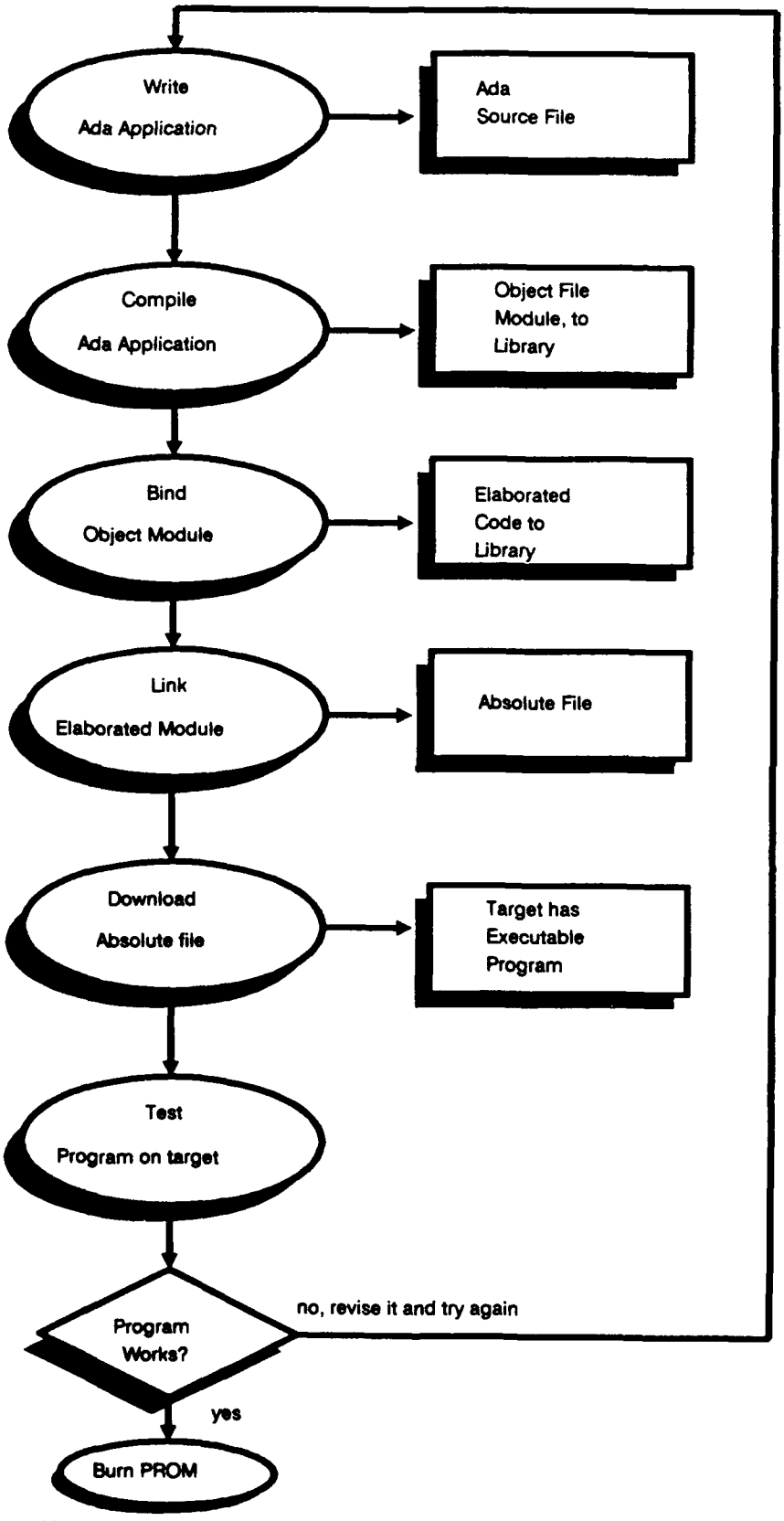


Figure 1: Real Time Software Development Procedure.

tion, the operating system supplies a number of prewritten and debugged software facilities such as interrupt handlers, data-transfer functions, real-time clocks, and I/O device drivers.

At the heart of a real-time executive is the concept of a task, or process. A task is an activity carried out by the computer. It consists of a program, data associated with the program, and computer resources such as memory space or I/O devices required to execute the program [6].

Programmers and designers of real-time systems frequently spend more time developing basic mechanisms such as intertask communications, synchronization, and memory management, than on the application program itself. In embedded applications, this set of mechanisms is called a real-time operating system or a real-time executive. Programmers build their applications using the real-time executive as the foundation.

It is important to note the difference between real-time multitasking operating systems and real-time executives. A real-time executive does not offer operating system commands and is not a replacement for an operating system such as UNIX. A real-time executive is used to create a single application on a target machine. Its sole purpose it to integrate a series of small programs into one real-time application [7]. A fullfledged real-time operating system adds a disk file system to the capabilities provided by a real-time executive.

A real-time operating system is distinguished from a normal multitasking operating system by its ability to schedule tasks on the basis of external events which are signaled to the computer by interrupts. Operating systems which are not intended for real-time applications usually do not give the user mechanisms to control how interrupt requests are handled and may actually disable interrupts for substantial periods while the operating system nucleus is executing. As a result, response to interrupts may be slow or interrupt requests may be missed entirely. Real-time operating systems are designed to provide fast response to interrupt requests. An interrupt latency time is often given in the specifications for a real-time operating system. This is the maximum amount of time it will take the operating system to recognize an interrupt request and begin servicing it.

Another distinguishing feature of real-time operating systems, or executives, is that their command interpreters are usually rather simple and they do not include many utility programs. The goal of a real-time executive is usually to serve the system it controls rather than human users.

The centralized multitasking capability offered by a real-time executive is not always required. In very simple real-time systems the application program can often be configured as a combination of a main program which executes sequentially and a set of interrupt service routines which respond to external events. However, as the system gets more complex and additional processors are added, it becomes more difficult to coordinate the interactions between the interrupt service routines and the main program. At some point, some well-organized means of coordination is required and multitasking becomes a necessity. This is the case for almost all DoD/military applications.

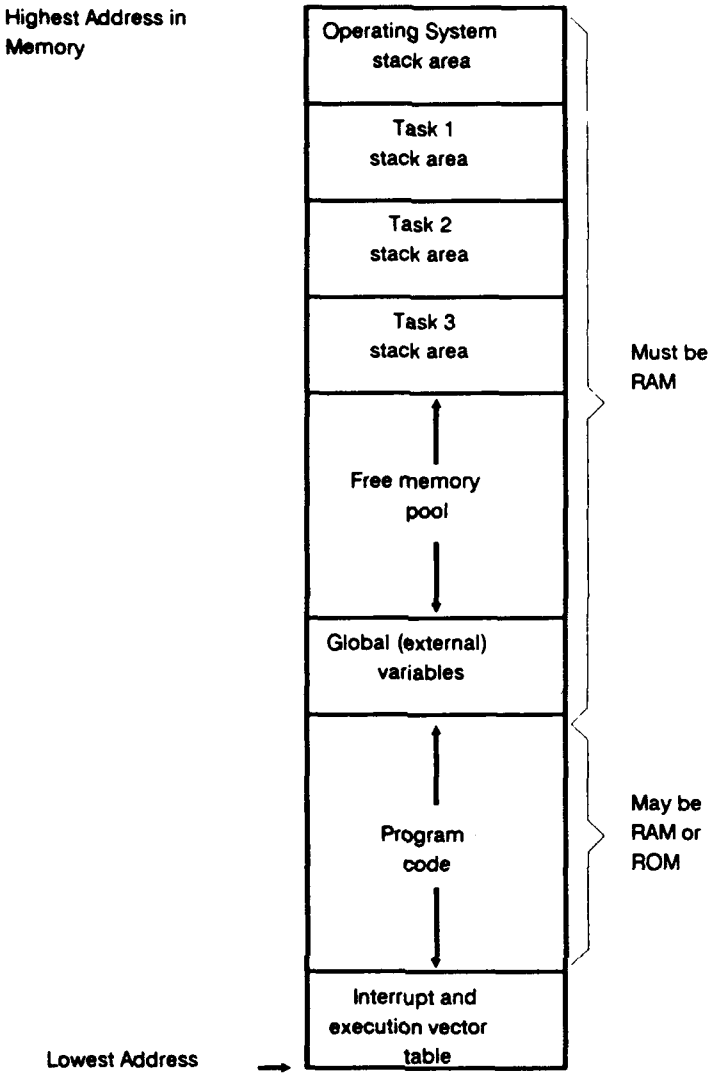


Figure 2: Memory Map for a Multitasking System.

A memory map for a multitasking system is shown in Figure 2. Each task and the operating system is assigned its own stack area in memory. This is used to store private data. A free memory pool is used by the operating system to create message channels or common data areas which allow tasks to exchange data. A certain number of global variables, accessible to all tasks and to the operating system, may also be required.

Two tasks can execute the same program but be distinct because they use different stack areas, message channels, and resources. For example, if a computer system contains three identical Analog/Digital (A/D) converters which provide input to the computer, three distinct tasks could be created to service the three devices. Each task will run the same code but will be assigned a different resource (A/D convertor), a different stack area, and a different message channel to transfer the incoming data to other tasks. The tasks will run independently (asynchronously) depending on when their A/D converter has data available. Since tasks can share program code, it is very important to use only reentrant programs in a real-time multitasking system.

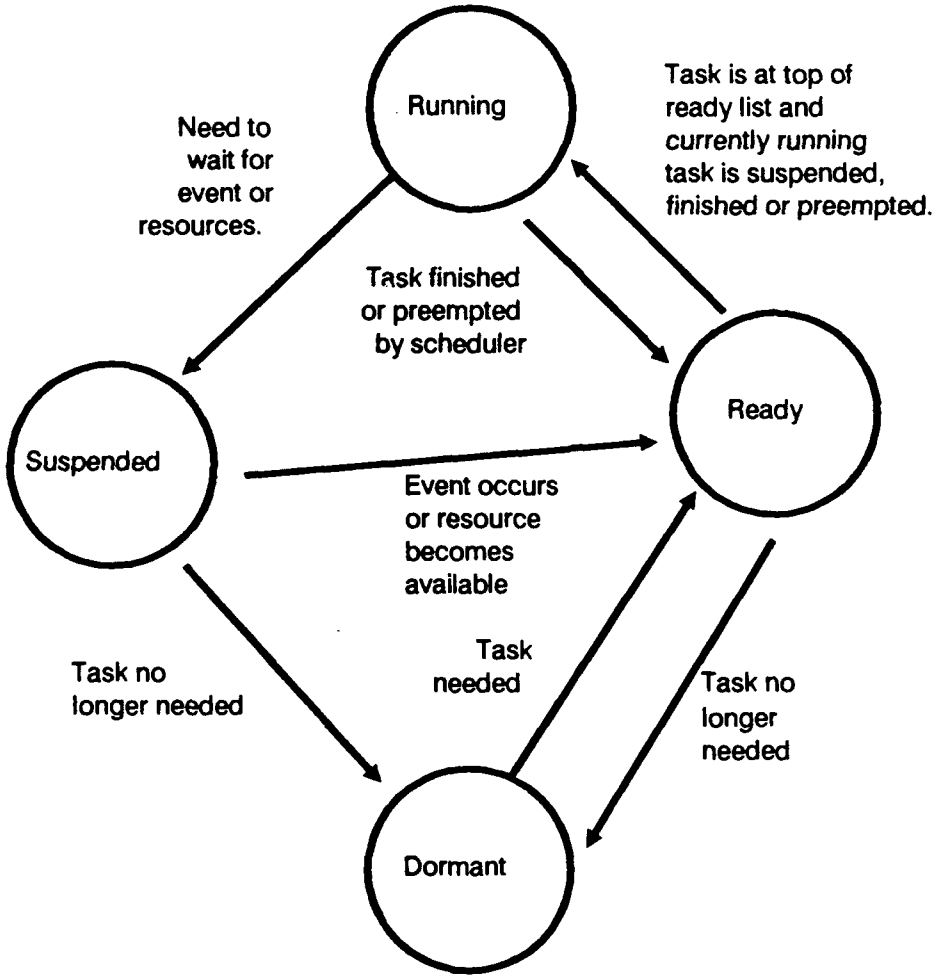


Figure 3: Task State Transistion Diagram.

Most real-time executives are designed so that a task can be in one of four states, as shown in the state transition diagram in Figure 3. If a task is in the running state, the computer is executing that task. If the task is ready to be executed but not actually executing, it is in the ready state. A task that is unable to execute because it is waiting for an event or a resource is in the suspended state or is said to be blocked. For instance, if a task is an interrupt service task, it must await the interrupt signal from the external device (an event) before it can execute. Similarly, a task which wishes to use a printer (a resource) may have to wait until another task has finished using the printer. Finally, if the task is not needed by the real-time system, it is in the dormant state.

The scheduler in a real-time operating system is responsible for controlling the transitions of the tasks among these states. The elements of a simple task scheduler are shown in Figure 4. The scheduler program receives interrupt requests from the computer's interrupt system. In addition, it receives messages from the running task in the form of system calls to the operating system. The scheduler maintains lists of ready, suspended, and dormant tasks. It also carries out the task switching function where the execution of the currently running task is stopped and a task on the ready list is started.

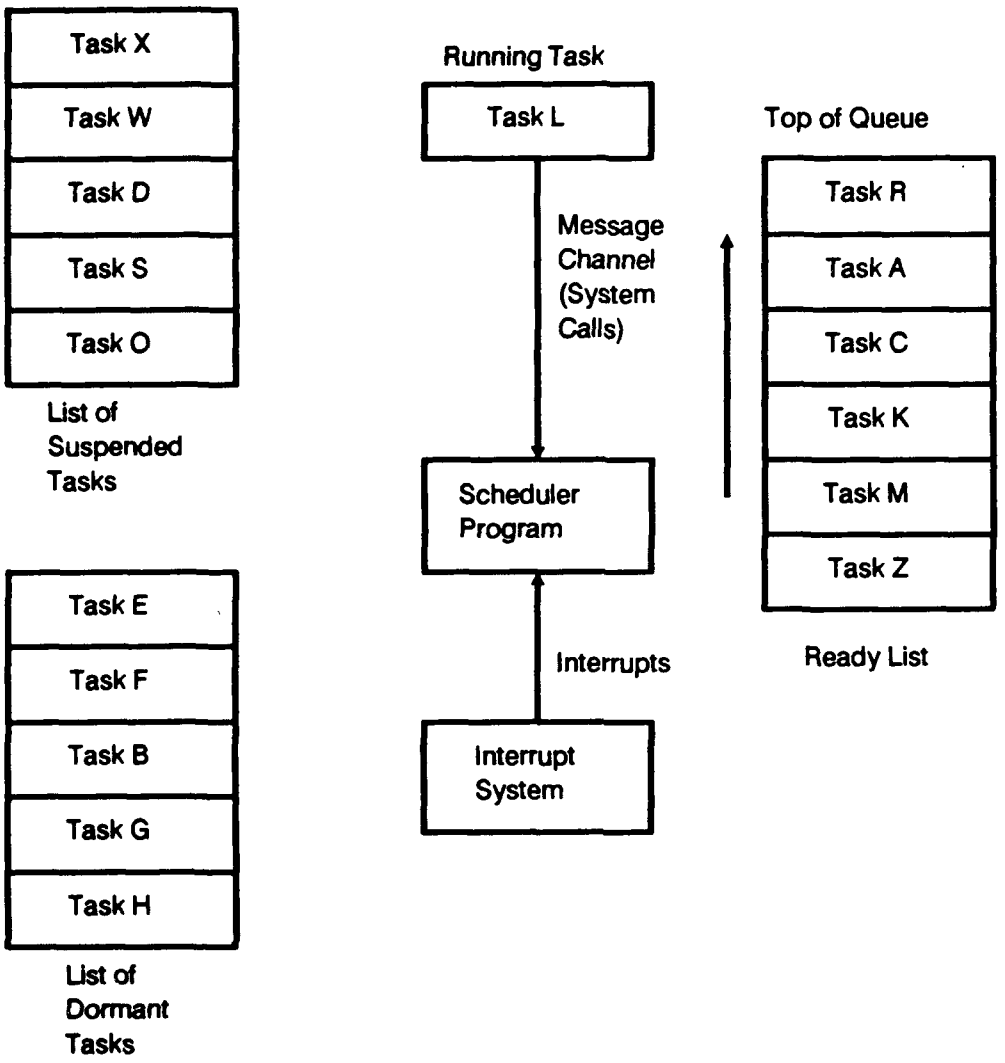


Figure 4: Elements of a Task Scheduler.

The scheduler's ability to control the execution of tasks is the key to the efficiency and speed of response of a real-time executive. A task which is waiting for an event to occur or a resource to become available does not execute and therefore does not take up any CPU time. When the event occurs or the resource becomes available (usually signaled by an interrupt or a message from the running task), the scheduler allows the task to continue execution. The amount of time that elapses between the occurence of an event and the execution of the task which was blocked on that event depends on the scheduling strategy used by the scheduler [6].

The actual scheduling method used to select the next task to run varies. In real-time operating systems, round-robin or priority-based preemptive scheduling techniques are typically implemented. In round-robin scheduling, all tasks have equal priority and the ready list is configured as a simple first-in-first-out (FIFO) queue. In priority-based preemptive scheduling, tasks are assigned different priorities. The highest priority task that is not in the suspended, or dormant, state is always running. In practice, a mixture of round-robin and priority-based scheduling is often used.

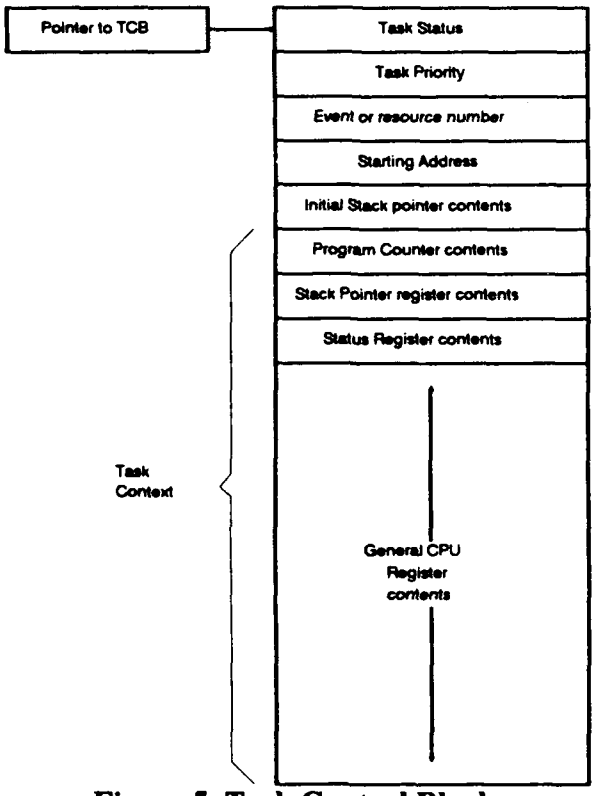


Figure 5: Task Control Block.

In order to carry out its scheduling functions, the scheduler makes use of task control blocks (TCB), or process descriptors. Each task is assigned one of these blocks, which is simply an array of data about the task. An example of a task control block is shown in Figure 5. The TCB contains data on the state of the task (i.e., dormant, suspended, ready, or running), the priority of the task, and events or resources for which the task is waiting. The starting address for the task's program code and the initial pointer value for the task's stack are also stored in the TCB. This information is used by the scheduler when a task is first activated and when a task is reactivated by moving it from the dormant to the ready state.

In addition, the task control block is used to store the task's context. The task context represents all the information that must be saved when the execution of a task is stopped and restored when task execution is resumed. In most real-time executives, this information is the contents of the computer's registers just prior to the moment when execution of the task was stopped. Note that this is the same information (the machine state) which must be saved and restored when the system responds to an interrupt.

In a simple system with more than one task, there is no inherent synchronization between the execution of tasks; each task executes independently. However, in most real-time systems, tasks must work closely together and may also have to perform their functions at defined times. Some synchronization mechanisms must be provided.

A common technique, used in real-time operating systems to synchronize two or more tasks, is to use the semaphore variable, s, and two system calls, Wait(s) and Signal(s), which act on the semaphore. A semaphore consists of a counter for signals that have not been received, and a queue for tasks that are waiting to receive the signals [6].

In addition, semaphores are used in the exchange of data between tasks. They prove quite useful in solving problems associated with the use of common data areas to transfer data between tasks. This process is also known as mutual-exclusion.

Most real-time operating systems provide a complete real-time clock facility. A task can make a system call to suspend itself for a time interval or until a certain time of day occurs. The scheduler suspends the task and places it back on the ready list only when the requested time interval has elapsed or the requested time of day has arrived. Thus, a task can ensure that it performs its functions at defined times [6].

To date, real-time executives have had their limitations. They must be custom designed for a given microprocessor. Therefore, the designer needs to choose a specific processor and hardware configuration before he/she can select an executive. Any executive chosen must meet certain performance criteria. It must be fast enough to allow the tasks under it to operate in real time. An executive's code should be small enough so that a large amount of program space does not have to be dedicated to it. Finally, the executive should not add a lot of unnecessary overhead to a task when that task needs to use executive utilities.

The ability to modify an executive can be important. Features can be modified or added to an executive as they are needed through the use of source code. Without the source code for an executive it is difficult, if not impossible, to alter it. Source code for some real-time executives cannot always be purchased. The cost of this code, when available, can easily cost up to $50,000 or more [7].

MICOM CASE HISTORY

Traditionally, whenever efficient executive code was required by the application, the user developed in-house custom code, sometimes written in assembler language. The FOG-M gunner station code is an example of an in-house custom executive.

In 1982 the Research and Development Center (then known as Army Missile Laboratory), MICOM, began a technology demonstration program to prove out the use of fiber optic cable control in missiles. This successful program is now known by all in the MICOM community as the Fiber Optic Guided Missile (FOG-M) program. Personnel in the Guidance and Control organization were directly responsible for the computer hardware and software designs for FOG-M. The FOG-M multiprocessor configuration uses several microprocessors, communicating with each other on a common MultiBus. These processors are tightly coupled under software control to distribute the processor load. Since the hardware configuration contained global memory, this configuration was not distributed processing in the strictest since, but was a multiprocessing configuration.

At the initial development stage, the FOG-M computer software and hardware was the most complex multiprocessing architecture using the Intel 8086 family of processors in existence. This complexity was mandated by the complex functions required of the system. Due to this complexity the software had to be very efficient, thus the choice of a higher order language to develop the code, was a critical design decision. Since the requirements of the Gunner Station software dictated the software operate under a multitasking software architecture, engineers began looking at commercial off the shelf software to fulfill the needs of a real time executive.

After several trade studies were completed, an initial design choice was made. After the initial code development, debug and testing were completed, it was determined that this commercial real time executive was not suitable for the speeds required of the software (e.g.; trigonometric functions were derived by table lookup versus a math coprocessor because of the timing constraints.) Engineers working on the FOG-M software program began studying the feasibility of developing a custom in-house executive which met the needs of the FOG-M program, and only the FOG-M program.

As a result, a minimal executive was developed which contained specific functions required for the executive (i.e.; task scheduler, interrupt handler, global memory handler, exception handler, etc.). This executive was not contained in a single section of program code, but was distributed around the software as the need arose. Since the code was developed for the specific application, it is highly unlikely much of the code could be used in applications other than follow on work to FOG-M using the same processor family.

Although the FOG-M program was a success on most every front, the above example illustrates the way real time executives have been developed and used at MICOM in the past. Individual software developers "re-invent the wheel" each time an executive is needed for an application. With the primary push for software reusability in the Ada programming language, the software developers within the G&C Directorate realized that this programming practice must end. The development of RTEMS is an attempt to alleviate this redevelopment cycle.

STANDARDS

The advent of Ada has created the need for efficient Ada-based development tools to augment the standard Ada runtime environment. This support must provide for efficient and flexible concurrent program execution that meets demanding real-time constraints. As we move from single to tightly-coupled to loosely-coupled multiple-CPU architectures, this support must be standardized to the extent practical. (Tightly-coupled system are generally characterized by the ability to communicate over a backplane. Loosely-coupled systems generally use some external communications media such as RS232 or network connection.) Existing Ada compiler runtime systems do not meet these needs.

Many real-time embedded computer systems require efficient, deterministic, and adaptable software concurrency, communication, and synchronization support. This support is needed for single processor systems and for distributed computer systems. The tasking model in Ada provides for concurrency, com-

munication, and synchronization and applies in principle to all computer architectures and applications. However, existing Ada compilers and their runtime support do not meet industry demands for efficiency, determinism, and adaptability in real-time embedded applications, and Ada implementations for distributed targets are only now becoming available.

The Ada language addresses a large application domain and offers many design features needed to promote reliable software; however, its intrinsic capability has not as yet been implemented with the efficiency and adaptability required to support many real-time computer system requirements. The need is the same for other languages used in real-time applications, but the power and formality of Ada dictate special emphasis on well-engineered execution environments that are integrated with Ada compilers.

Runtime support must be provided that allows embedded computer systems to be implemented with inherent runtime efficiency and with designed-in determinism. And, to increase reliability and reduce development cost and schedules, implementors must be able to tailor proven, available runtime support capabilities to their specific program needs. The Ada standard does not preclude any of those needs, nor does it overtly support them. Available implementations of the Ada tasking model do not meet either efficiency or determinism needs, and compiler vendor products and third-party products do not meet the system tailoring requirements of many DoD systems.

We have established that the run time executives being delivered with the Ada compilers are too slow and inefficient to be used in the increasingly complex modern missile systems of today. This code is purchased from compiler vendors who are not in the business of writing and properly testing and debugging real-time executive code. Indeed, their primary concern is in providing the customer (the Government) with a good compiler. The real-time executive is included as part of the run time environment in order to satisfy the underlying requirements of a run time support system and to attempt to overcome some of the shortfalls of the Ada programming language.

To date, software developers are purchasing and utilizing specialized third party kernel code. Problems imposed from purchasing third party kernel (executive) code is that the contractor, and eventually the Government, must pay a licensing (royality) fee for every copy of the kernel code used in an embedded system. This concept is similar to purchasing multiple copies of word processing software. In both cases, the original manufacturer of the software is the sole owner, while the end user pays a licensing fee for the right to use the code in a system design. This obligates the government on a per copy basis, every time the code is being utilized.

Another drawback to this development scenario is that the Government does not own, nor has the right to modify any code contained within the kernel. Techniques for Validation and Verification (V&V) of software in this situation are more difficult than if the complete source code were available. Most commercially available real-time executive manufacturers do not offer source code to the licensee. If they do, it is usually very expensive. Responsibility for system failures due to faulty software is yet another area to be resolved under this en-

vironment. The vendor will not accept any responsibility for a failure of any system containing its' code.

Typically, real-time software has been ignored by standardization and software engineering groups. Each system is almost an original, free-form expression of the development team. Unfortunately, real-time design is still in its infancy. This results in schedule overruns and maintenance problems. As a result, the cost of software continues to accelerate, even as the cost of hardware continues to decline [8].

To meet these challenges, the software industry must develop new and better tools for the design, analysis, development, and verification of real-time systems. Although research is constantly advancing the techniques of software development, the benefits of this research often take time to become available to the real-time software development community. As the results of real-time operating system research become accessible, the number of real-time operating systems and executives available to software developers will continue to grow. This fact makes it imperative that system developers recognize the common features and capabilities required to provide the needed support environment and incorporate them into an industry standard interface environment. This standard interface should allow the software developer to concentrate on the hardware dependencies and unique requirements of the application system being developed instead of learning to use yet another real-time executive.

RTEID

One such standard interface, the Real Time Executive Interface Definition (RTEID) has been developed by Motorola, Inc. with technical input from Software Components Group. It has been submitted to the VMEbus International Trade Association (VITA) for adoption as a standard multiprocessor, real-time executive interface. RTEID defines a standard interface for the development of real-time software to facilitate the writing of real-time applications programs that are directly portable across multiple real-time executive implementations. This interface includes both the source code user interface and the run-time behavior as seen by a real-time application. It does not include the details of how a kernel implements these functions. Simply stated, the RTEID goal is to serve as a complete definition of external interfaces so that application code which conforms to these interfaces will execute properly in all real-time executive environments. With the use of an RTEID compliant executive, routines that acquire memory blocks, create and manage message queues, establish and use semaphores, and send and receive signals need not be redeveloped for a different real-time environment as long as the new environment is also RTEID compliant. Programmers need only concentrate on the hardware dependencies of the real-time system. Furthermore, most hardware dependencies for real-time applications can be localized to the device drivers [9,10].

An RTEID compliant executive provides simple and flexible real-time embedded multiprocessing. It easily lends itself to both tightly-coupled and loosely-coupled configurations (depending on the system hardware configuration). Both forms of multiprocessing, tightly-coupled and loosely-coupled, have unique advantages, disadvantages, and suitability for a specific application. RTEID does not favor one form over the other, but leaves this decision to the developer of an

RTEID compliant executive. Objects such as tasks, queues, events, signals, semaphores, and memory blocks can be designated as global objects and accessed by any task regardless on which processors the object and the accessing task reside. Each object may exist on a single processor configuration; or in a multiprocessor system. The system is defined as the collection of interconnected processors; including processors connected by network or other communications media [9].

RTEMS

The Guidance and Control Directorate began a software development effort in 1989 to alleviate many of the problems discussed in this paper. A project to develop an experimental run time kernel was begun that will eliminate the two major drawbacks of the Ada programming language mentioned previously: that the Ada programming language does not fully support multiprocessing and that the run time executives being delivered with the Ada compilers are too slow and inefficient to be used in modern missile systems. The Real Time Executive for Missile Systems (RTEMS) is an implementation based on Draft 2.1 of the RTEID specification. RTEMS provides full capabilities for task management, interrupt management, time management, multiprocessing, and other managers typical of generic operating systems. The code will be Government owned, so no licensing fees need be paid. The executive was designed as a linkable, ROMable library with the Ada programming language. Initially the library code is being developed on the Motorola 68000 family of processors using the 'C' programming language as the development language. The 'C' programming language was chosen because of its portability and efficiency. However, other language and processor family interfaces are planned in the future.

The final RTEMS product will be capable of handling either homogeneous (processors of the same family type) or heterogeneous systems. The kernel will automatically compensate for architectural differences (byte swapping, etc.) between processors. This will allow a much easier transition from one processor family to another without a major system redesign.

RTEMS was designed to fulfill three fundamental design objectives: performance, reliability, and ease of integration. It provides a multitasking environment for single or multiprocessor real-time application systems. The RTEMS executive was developed by contractors for MICOM to perform research for multiprocessor based weapons systems. RTEMS is currently implemented on the Motorola MC68020 microprocessor [10].

The RTEMS executive provides a high performance real-time environment which include the following features: [10,11]

- *multitasking capabilities*
- *event-driven, priority-based, preemptive scheduling*
- *intertask communication*
- *semaphore, signal, and event synchronization mechanisms*
- *dynamic memory allocation*

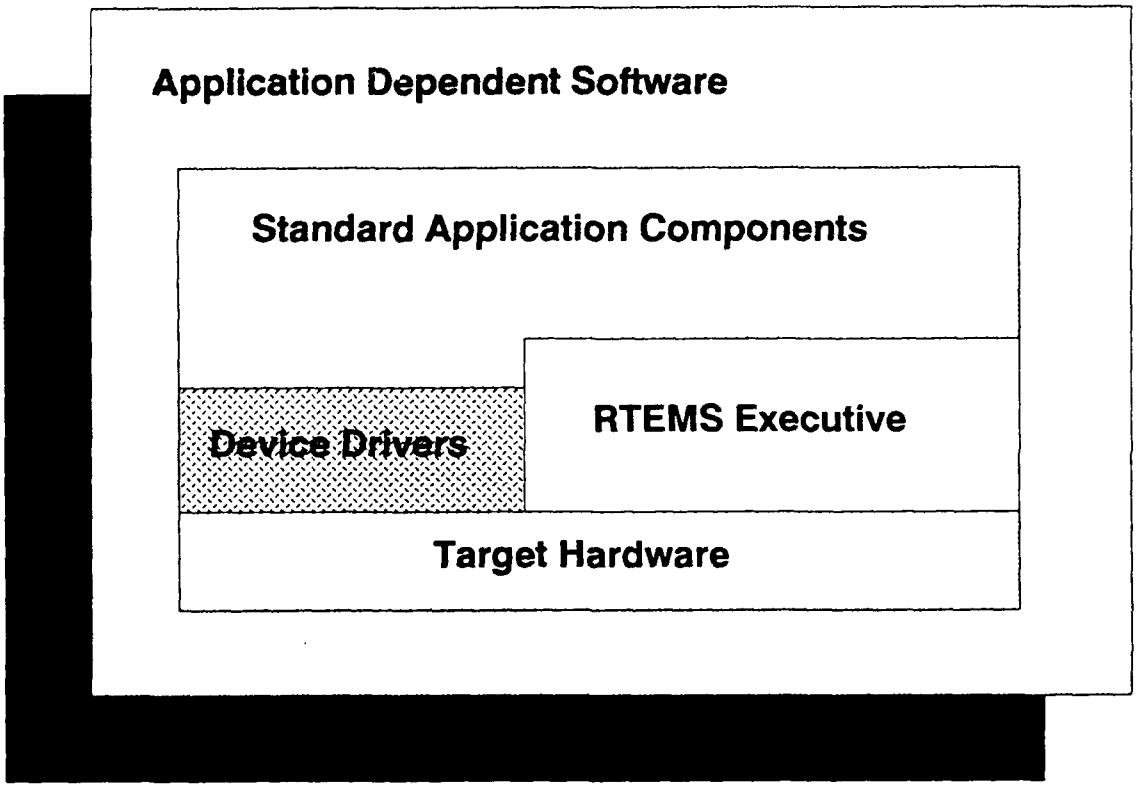


Figure 6: Real-Time System Architecture.

- *real-time clock management*
- *user-specified configurations*
- *user-extendable directives*

These features provide a robust set of cabability that allow system designers the flexibility to efficiently and cost effectively solve the complex problems associated wtih real-time systems.

Another important design goal of the RTEMS executive was to provide a bridge between two critical layers of typical real-time systems. It serves as a buffer between the project dependent application code and the target hardware. Standard software routines that acquire memory blocks, create and manage message queues, establish and use semaphores, and send and receive signals need not be redeveloped for a different real-time environment as long as the new environment is RTEMS/RTEID compliant. RTEMS provides efficient tools for incorporating these hardware dependencies into the system while simultaneously providing a general mechanism to the application code required to access them. A well designed real-time system, such as RTEMS should maximize these two concepts to build a rich library of standard application components which can be used repeatedly in other real-time projects [9,10,11].

The executive can be viewed as a set of components that work in harmony to provide a collection of services to a real-time application system. These com-

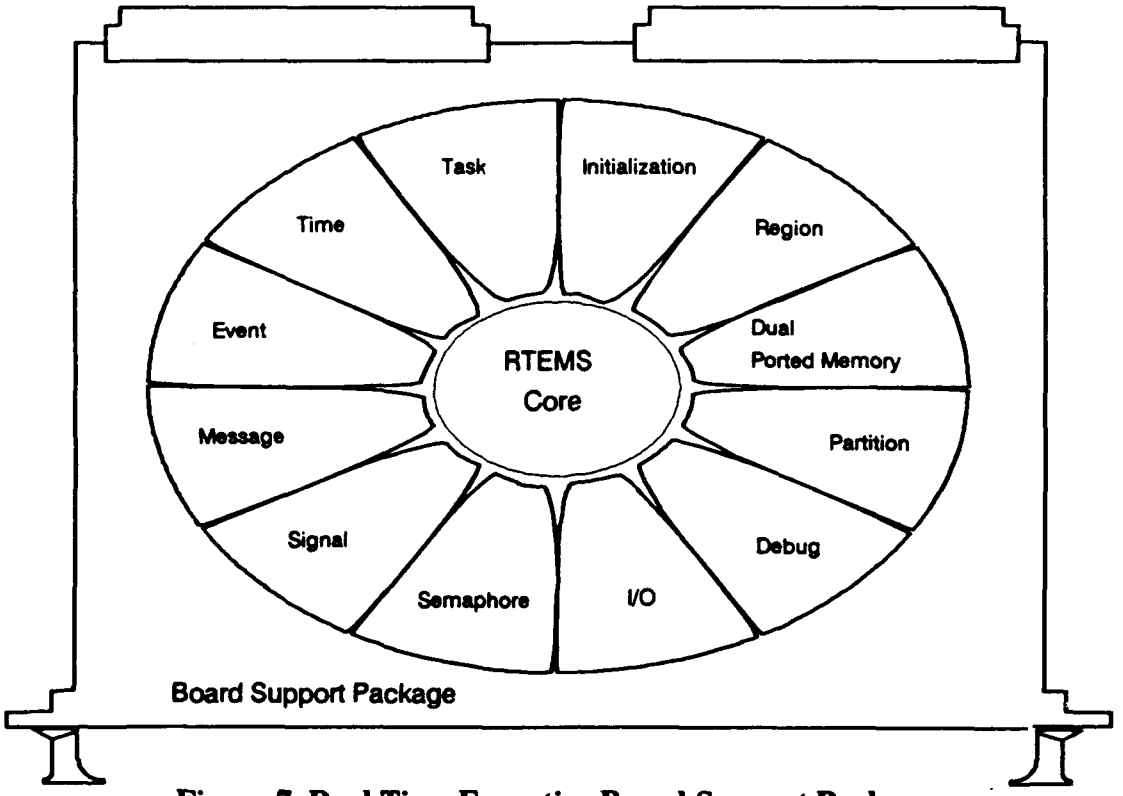


Figure 7: Real Time Executive Board Support Package.

ponents consist of a board support package, an executive core and a set of resource managers. Figure 7 shows this concept.

RTEMS makes minimal assumptions about its hardware environment. It does not depend on any particular implementation of interrupts, timers, buses, or I/O devices. This concept is necessary to allow the executive to be truly portable. To achieve this portability, the user must supply a small amount of code to interface RTEMS to those aspects of the surrounding hardware that it needs to know about. The **board support package** is a single piece of code that sets up the environment and controls the initialization process for the remainder of the software.

In every executive there exists a set of basic functions that must be performed, but do not naturally align themselves with any of the logical sets or groups of directives. Such things as scheduling, error processing, and data structure manipulation are very critical to insure the proper functioning of a real-time executive. The RTEMS design groups these functions together forming a component called the **executive core**.

The RTEMS core consists of the following components:

- *scheduler*
- *dispatcher*
- *chain handler*

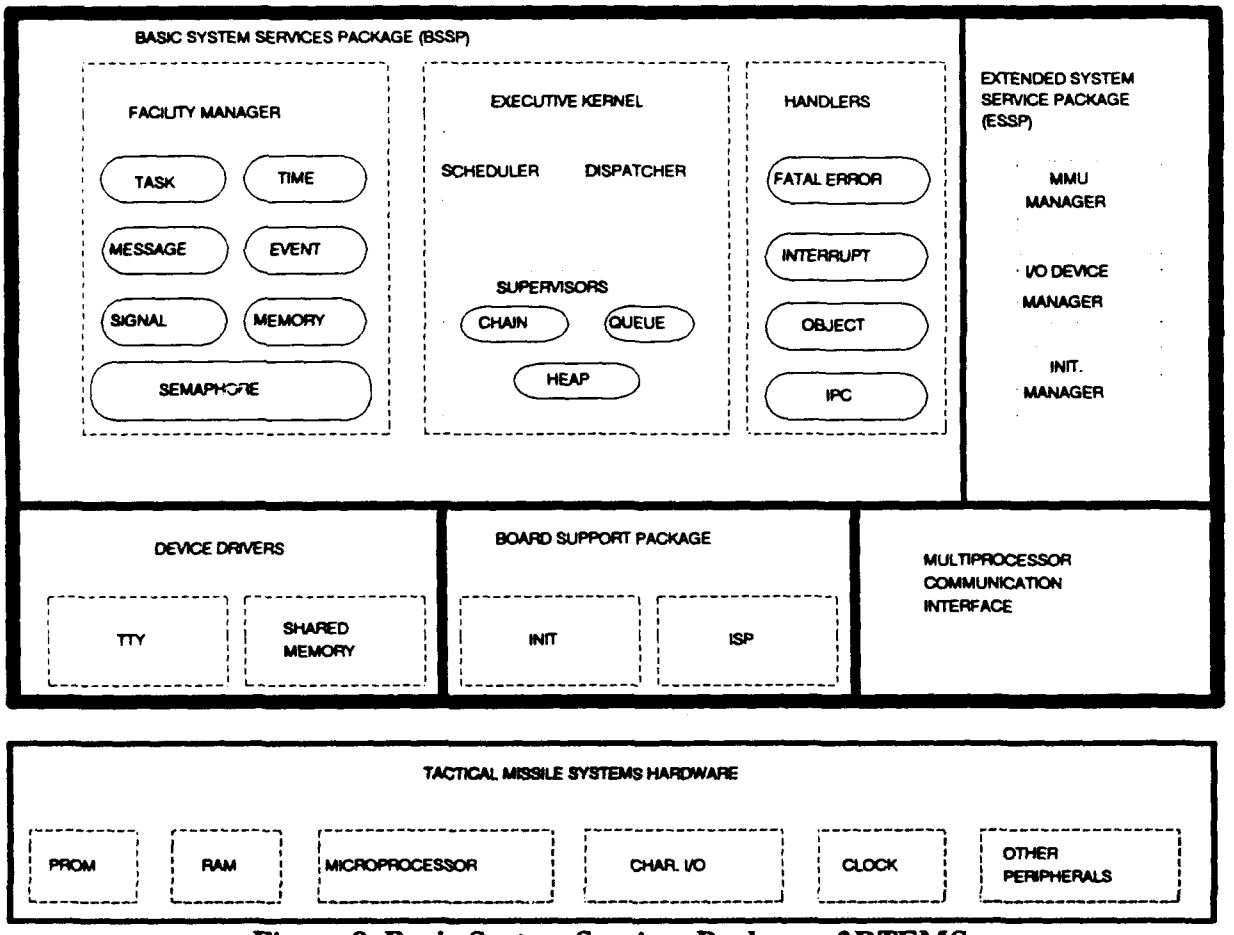


Figure 8: Basic System Services Package of RTEMS.

- *queue handler*
- *heap handler*
- *utilities*

Although no requirement exists for the RTEMS user to understand the details of the executive's implementation, it is important to be familiar with the basic concepts and algorithms used to control the real-time environment.

The RTEMS interface presented to the application is formed by grouping the RTEID specified directives into logical sets called **resource managers**. Together these components provide a powerful run time environment that promotes the development of efficient real-time application systems.

The following managers are included in an RTEMS compliant executive:

- *task manager*
- *message manager*
- *event manager*

- *signal manager*
- *semaphore manager*
- *time manager*
- *interrupt handler*
- *fatal error handler*
- *region manager*
- *partition manager*
- *initialization manager*
- *dual ported RAM manager*
- *I/O manager*
- *debug manager*

The **task manager** provides control features that act upon a task or a set of tasks as defined by the eleven task manager directives specified by the RTEID specification. A task is simply a sequence of closely related computations. A task may execute concurrently with or independent of other tasks.

In real-time multitasking applications, the ability for cooperating tasks/Interrupt Service Routines (ISRs) to communicate and synchronize with each other is imperative. A real-time executive should provide an application with the following capabilities:

- *Data transfer between cooperating tasks*
- *Data transfer between tasks and ISRs*
- *Synchronization of cooperating tasks*
- *Synchronization or tasks and ISRs*

RTEMS provides for communication and synchronization between tasks and between tasks and ISRs with the message, event, signal, and semaphore managers. The **message manager** supports one type of inter-task communication and synchronization as defined by the seven message manager RTEID directives. The message manager supports communication and synchronization between multiple tasks as well as tasks and ISRs using a basic support mechanism called a message queue. Messages are defined to be fixed length (16 bytes) blocks of information. The **event manager** provides a second, higher performance method of inter-task communication and synchronization as defined by the two event manager directives. The event manager will support communication and synchronization using a basic support mechanism called an event set. Event sets may only be directed at other tasks instead of queues. Events are defined to be bits encoded into an event mask. The **signal manager** supports a third type of inter-task communication and synchronization as defined by the three signal manager directives. The signal manager provides directives that allow asynchronous communication between tasks. The **semaphore manager**

supports a type of inter-task synchronization as defined by the five semaphore manager directives. These directives provide the ability to arbitrate access to a shared resource.

The **time manager** provides timing features based on both calendar and elapsed time as defined by the eight time manager directives. This manager requires a periodic timer interrupt to perform its required functions. The board support package will inform this manager that a clock tick has occured.

The **interrupt handler** provides the ability to preempt from an interrupt service routine while still maintaining a fast interrupt response and satisfies the basic design goals to provide a zero latency time to enter an interrupt and to insure that the highest priority ready task always executes. The **fatal error handler** also provides the ability to preempt from an interrupt service routine while still maintaining a fast interrupt response. Fatal errors can be detected from three sources: the executive (RTEMS), user system code, or user application code.

RTEMS provides two types of memory management: region management and partition management. The **region manager** deals with the allocation and deallocation of variable size segments in a specified region. Whereas, the **partition manager** deals with the allocation and deallocation of fixed (equal) size buffers in a specified partition.

The RTEID specification allows executive developers to define their own initialization mechanism. The RTEMS **initialization manager** provides for the initialization of RTEMS and the initiation of multitasking.

The **dual ported memory manager** provides a mechanism for converting addresses from internal to external representations. Dual ported memory can be accessed at two different address ranges. Typically, one of these ranges, the internal addresses, is used exclusively by the node which owns the memory. All other nodes in the system must use the external addresses to access the memory.

The **I/O device manager** provides a standard interface for accessing device drivers. This standard interface encourages the development of well-structured RTEID compliant device drivers.

The **debug manager** provides an interface between the executive and a debugger. This allows the development of debuggers that work efficiently and correctly with any implementation of an RTEID compliant executive. The debug manager provides three groups of features for: debugging tasks, debugging entire systems, and monitoring a running system.

To realize the goal of hardware independence, RTEMS makes no assumptions about the physical media connecting the nodes, or the topology of the connection. To perform interprocessor communication, RTEMS calls a user provided communication layer known as the Multiprocessor Communications Interface (MPCI). This MPCI routines enable the nodes in a multiprocessor system to communicate with one another.

Remote procedure calls (RPC) are used in RTEMS to transcend the physical boundaries of the set of processors included in the system. Conceptually an RPC can be viewed as a simple call to a procedure. The called procedure just happens to reside on another processor.

The software developer uses the set of directives provided by RTEMS to be free from the problems of controlling and synchronizing multiple tasks and processors. This freedom allows the programmer to concentrate all creative efforts on the application system.

The system calls, including optional managers provided by the executive are shown in the following figure [10,11].

Task Manager	
t_create	Create a task
t_ident	Get id of task
t_start	Start a task
t_restart	Restart a task
t_delete	Delete a task
t_suspend	Suspend a task
t_resume	Resume a task
t_setpri	Set task priority
t_mode	Change task mode
t_getreg	Get value in tasks's register
t_setreg	Set task's register to value
Message Manager	
q_create	Create a message queue
q_ident	Get id of message queue
q_delete	Delete a message queue
q_send	Send message to message queue
q_urgent	Put message at front of message queue
q_broadcast	Broadcast N messages to queue
q_receive	Receive message from message queue
Event Manager	
ev_send	Send an event to a task
ev_receive	Receive an event
Signal Manager	
as_catch	Establish ASR
as_send	Send a signal to a task
as_return	Return from ASR
Semaphore Manager	
sm_create	Create a semaphore
sm_ident	Get id of semaphore
sm_delete	Delete a semaphore
sm_p	Get a semaphore
sm_v	Release a semaphore
Time Manager	
tm_set	Set system date and time
tm_get	Get system date and time
tm_wkafter	Wake up after specified interval
tm_wkwhen	Wake up at specified date and time
tm_evafter	Send event after specified interval
tm_evwhen	Send event at specified date and time
tm_cancel	Cancel timer event
tm_tick	Announce clock tick
Interrupt Processing	
i_return	Return (exit) from interrupt
Fatal Error Processing	
k_fatal	Invoke the fatal error handler
Region Manager	
rn_create	Create a region
rn_ident	Get id of region
rn_delete	Delete region
rn_getseg	Get segment from region
rn_retseg	Return segment from region
Partition Manager	
pt_create	Create a partition
pt_ident	Get id of partition
pt_delete	Delete a partition
pt_getbuf	Get buffer from partition
pt_retbuf	Retrun buffer to partition
Initialization Manager	
ex_init	Initialize RTEMS
ex_start	Initiate multasking
Dual Ported RAM Manager	
m_ext2int	Convert external address to internal address
m_int2ext	*Convert internal address to external address*
I/O Manager	
de_init	Initialize device driver
de_open	Open device for I/O
de_close	Close device
de_read	Read from device
de_wrtie	Write to device
de_cntrl	Special device services
Debug Manager	
db_control	Control a task
db_remote	Perform directive on remote cpu
db_block	Prevent a task from running
db_unblock	Run a task under control
db_getmem	Get a task's memory
db_setmem	Set a task's memory
db_getreg	Get a task's register
db_setreg	Set a task's register
db_system	Control a system
db_level	Set minimum proecssor mask level
db_get_id	Get identifier for an item
db_get_item	Get information about an item
MPCI	
mc_init	Initialize the MPCI
mc_getpkt	Obtain a packet buffer
mc_retpkt	Return a packet buffer
mc_send	Send a packet to another node
mc_broadcast	Send a packet to all other nodes
mc_receive	Called to get an arrived packet

Figure 9: RTEMS Directives.

CONCLUSION

The primary purpose of this paper is to summarize the impacts of Ada software development on current MICOM software development philosophies. The limitations of Ada in a distributed processing architecture and what the software development community was doing to complement the shortfalls of Ada was discussed. A brief discussion was given outlining how real-time software development is accomplished on Army missile systems. A detailed explanation of real-time executives in general was provided to give the reader an idea of the complexity of the task involved with developing an environment where Ada could be used in real-time systems. An example was then described showing how embedded computer software has been developed in the past at MICOM.

Many of the same events that lead to the development of the Ada programming language are also the same events that are mandating the development of a standardized real-time executive such as RTEMS. The need for developing a standard executive for use in embedded missile systems was then described. The Real Time Executive Interface Definition (RTEID) was introduced as the first attempt to develop a standard executive. And finally, the Real Time Executive for Missile Systems (RTEMS) was described as the MICOM solution for all the Ada shortfalls and standardization thrusts described in the previous sections.

It is hoped that this paper will spawn interests in real-time executives and standardization within the Army community.

REFERENCES

[1] J.G.P. Barnes. Programming in Ada. Addison-Wesley Publishing Company. London, England. 1984.

[2] David Naiditch. Rendezvous With Ada: A Programmers Introduction. John Wiley & Sons. New York, NY. 1989.

[3] Stephen J. Young. An Introduction to Ada. Ellis Horwood Limited. Halsted Press. New York, NY. 1983.

[4] Narain Gehani. Ada: An Advanced Introduction. Prentice-Hall, Inc. Englewood Cliffs, NJ. 1983.

[5] Ready Systems, RTAda Real-Time Ada User's Guide for VAX/VMS-to-68020. Ready Systems. Sunnyvale, CA. 1989.

[6] Peter D. Lawrence and Konrad Mauch. Real-Time Microcomputer System Design: An Introduction. pages 482-489. McGraw-Hill Book Company. 1987.

[7] Gary Elfring. A Guide to Real-Time Executives. Computer Language. pages 65-70. June 1986.

[8] Transforming Software Design, From Art into Science. Software Components Group. Santa Clara, CA.

[9] R. Vanderlin, P. Raynoha, B. Hansche, and L. Dion, "RTIED: The Quest for Real Time Standards." Motorola Microcomputer Division. Tempe, AZ.

[10] RTEMS-68020/C User's Manual. G&C Internal Report. Redstone Arsenal, AL.

[11] Real Time Executive Interface Definitiaon, Motorola Microcomputer Division. Tempe, AZ. 22 January 1988.