Embedded Application Design Using a Real-Time OS

David Stepner
Integrated Systems, Inc
201 Moffett Park Drive
Sunnyvale, CA 94089.
+1-408-549-1800
dstepner@isi.com

Nagarajan Rajan
Integrated Systems, Inc
201 Moffett Park Drive
Sunnyvale, CA 94089.
+1-408-542-1724
rajan@isi.com

David Hui
Integrated Systems, Inc
201 Moffett Park Drive
Sunnyvale, CA 94089
+1-408-542-1735
dhui@isi.com

1. Embedded Systems Overview

You read about it everywhere: distributed computing is the next revolution, perhaps relegating our desktop computers to the museum. But in fact the age of distributed computing has been around for quite a while. Every time we withdraw money from an ATM, start our car, use our cell phone, or microwave our dinner, microprocessors are at work performing dedicated functions. These are examples of just a very few of the thousands of "embedded systems."

Until recently the vast majority of these embedded systems used 8- and 16-bit microprocessors, requiring little in the way of sophisticated software development tools, including an Operating System (OS). But the breaking of the $5 threshold for 32-bit processors is now driving an explosion in high-volume embedded applications. And a new trend towards integrating a full system-on-a-chip (SOC) promises a further dramatic expansion for 32-bit embedded applications as we head into the 21st century.

Aerospace companies were the first end-markets for embedded systems, driven by military and space applications. But this market has never developed the growth potential of the newer commercial applications. In the past five years, the major end-markets for embedded systems have been telecommunications, computer networking, and office equipment. But, now we see consumer and automotive electronics as major emerging markets. And looming on the horizon is the suspected wave of networked appliances, with Sun Microsystems, IBM, Microsoft and others targeting products and standards; envisioning billions of embedded network connections running embedded JAVA applications across a network.

Depending on whom you believe and how it is defined, the 32-bit end of the market is growing at 40% per year[1] to 100% per year[2]. This explosive growth has spawned a rapidly evolving, but up until now, very fragmented set of software tools and microprocessor suppliers. The market includes dozens of small companies competing in different market niches. Rapid market growth has recently attracted the serious attention of Microsoft, HP and Sun Microsystems. These new entrants will stimulate even further market growth.

Indeed, all the major semiconductor companies are targeting embedded applications with 32-bit processors. Motorola represents the market leader, and has continued to focus on this area by producing many variants of its processors with specific functionality added for communications, automobiles, electronics and consumer applications. ARM and MIPS represent the fastest growing segments, with the business model being the licensing of the core processors to a wide group of manufacturers. The Japanese manufacturers have also not been idle, with Hitachi, NEC, Mitsubishi, Fujitsu and others also targeting the embedded space. Hitachi's SH series has been the most successful with integration into digital cameras, cars, and many other consumer electronic applications.

An outstanding source of information on the latest developments in the embedded semiconductor market is MicroProcessor Report, which is published about every two to three weeks. Included in each issue is a *Chart Watch* section that graphs various performance metrics for various processors, such as Dhrystones vs. Power Consumption, or Dhrystones vs. Price. Each issue will focus on some segment of the market, such as low-power processors, high-end processors, or 32-bit processors.

2. Embedded Applications Development Characteristics

What characterizes an embedded system? Usually it means that there are a set of pre-defined, specific functions to be performed, and that the resources available (e.g., memory, power, processor speed, computational functionality) are constrained. Often, though not always, the application will run out of ROM. This, in comparison to a desktop computer, which is a general-purpose processor and support system designed for a wide range of applications. The range of embedded software is much broader than desktop software, where a handful of applications (word processors, spreadsheets, games, and so on) make up the vast majority of applications. The embedded market has an estimated 50,000 distinct designs every year.

Tools for embedded application development have traditionally lagged behind desktop tools. But embedded tools are now catching up with Windows GUI, object oriented programming and client/server architectures. In fact, embedded development tools are the fastest growing segment of EDA. These new tools are stressing ease-of-use and enabling team disciplines to work on the same project from different locations across the enterprise.

Driving this need for tools is a basic fact of the embedded market. With increased competition, companies supplying embedded products cannot afford schedule slippage that result in missed market opportunities. Increasing the productivity of engineers and

---

[1] "Changes in Embedded Design Methodology," Jack Ganssle, *Embedded Systems Programming*, Sept 1998

[2] Ainsley Capital Management, April 22, 1998

programmers has become the critical factor in bringing embedded products to market quickly and at reasonable cost.

The most developed segment of the embedded tools market are the off-the-shelf real-time operating systems (RTOSs), including their support programming tools: source level debuggers, integrated development environment, and compilers. While in-house developed RTOSs and tools still have an estimated 50% of the market, the complexity of 32-bit design has fostered an increasing demand to move to these third-party, off-the-shelf, standards-based RTOSs.

The commercial RTOS market is highly fragmented with offerings from dozens of vendors, supplying products for many different microprocessors. This fragmentation is caused by three factors. First, there are dozens of different microprocessors optimized for different embedded applications. A different version of the RTOS, with a corresponding set of development tools, must be written for each microprocessor. Second, different applications offer widely variable sets of available programming resources. Some RTOSs are optimized for more resource constrained environ-ments, and others are aimed at less constrained environments. And third, different end market applications have different needs and levels of complexity. Some RTOSs offer a wide range of available services, while others are simpler. A single RTOS cannot provide the optimal solution for every application.

3. Real Time Operating Systems

What does "real-time" mean when used in the context of an operating system? Simply put, this means that the embedded application can and will respond to external events in real time, as opposed to waiting for some other task or process to complete its operation. This is made even more confusing by the use of the terms "hard real-time" and "soft real-time."

Hard real-time means an activity must be completed **always** by a specified deadline (which may be a particular time or time interval, or at the arrival of some event), usually in tens of microseconds to few milliseconds. Some examples include the processing of a video stream, the firing of spark plugs in an automobile engine, or the processing of echoes in a Doppler radar.

Soft real-time applies to those systems that are not hard real-time, but some sort of timeliness is implied. That is, missing the deadline will not compromise the system's integrity, but will have a deleterious effect. Examples of this type of system are point of sale (POS) systems in retail stores, ATMs and other credit card machines, and PDAs. When a POS system can not read the bar code because the item was scanned too quickly, the system simply indicates an error, and the item will be scanned again for identification.

Further confusing the notion of "hard" and "soft" real-time is increased processor speeds. When the processor speed increases, interrupts are processed more quickly. More importantly, the interrupt window in which interrupts are disabled keeps shrinking and this will improve the timeliness of response. So "soft" real-time performance may improve just as a function of processor speed. But countering this trend is the increasing complexity of the applications, requiring more processing to be done at interrupts, and the blurring of the hardware-software interface.

But be they hard or soft, real-time (or perhaps a general term should be "embedded") OSs have four characteristics in common that differentiate them from desktop or mainframe OSs.

Bounded Interrupt Servicing. There is a maximum allowable time that the system can be diverted to process an interrupt. The interrupt service routine must do the absolute minimum processing and terminate. Similarly, an RTOS must minimize the window during which interrupts are disabled.

Priority Based Scheduling. In a real-time system, all tasks are assigned a level of priority, viz a viz each other. This priority may be based on any number of criteria (including run time). This implies that tasks do not execute just because they are "ready," but rather because they are the highest priority task that is ready.

Pre-Emptive Tasks. All tasks and routines must be constructed in such a way that they can be pre-empted by some higher priority task or routine becoming ready.

Scalability. The OS services provided are not monolithic. Rather, they are provided as a set of modules or libraries. The services needed for an application are included in the build by simply setting flags at the time of the application build; or, in the case of libraries, by having the linker pull in the services used by the application; or by using conditional compilation to scale the OS.

But, besides these four, there are other differences between real-time and desktop OSs that have more to do with the needs of the end application, the needs of the embedded developer, and the restrictions placed on the application by the resources available. The most obvious is the RAM requirement. Considering the volumes and tight end user pricing of most embedded systems, RAM is a very precious commodity. The OS must use this memory efficiently while preventing fragmentation, recovering RAM when tasks are terminated, requiring the minimum amount of RAM when tasks are created, and providing for efficient stack and heap structures.

Probably just as important are the scheduling algorithms, since these are at the heart of system performance. There are wide varieties of algorithms that have been developed and, depending on the end application, the developer would want to choose the one satisfying the response requirements while being the stingiest on resources. Some of the algorithms developed include:

- *Heuristic.* At any time, the task with the earliest deadline will be executed. This algorithm is efficient, but it may not find a feasible schedule even if one exists.
- *Cyclic.* A cyclic executive is typically based on one or several major cycles that describe the order in which minor cycles are executed. One segment of the program is executed per minor cycle, and typically all minor cycles are the same length. The major cycle is executed repeatedly. If a segment is shorter than the minor cycle, the processor will idle.
- *Round Robin*: Each task is assigned a fixed amount of processor time, and when that time is up, the next task executes.
- *Simple Priority:* The user assigns a priority at the time of thread creation. The next thread to execute is based on the priority of the "ready" thread. In a large system, it can be difficult for the user to decide the priorities of each thread.
- *Rate Monotonic:* The tasks of the program are assigned priorities in descending order according to the length of the period. The task with the shortest period has the highest priority, and the task with the longest period has the lowest priority. In its simplest form, it does not provide support for sporadic events. Modifications have been proposed, such as polling, priority exchange algorithm, and deferrable server algorithm.

- *Deadline Monotonic Scheduling:* This is close to the Rate Monotonic but accommodates sporadic tasks.
- *Priority Inheritance Protocol:* Neither of the rate monotonic scheduling algorithms assure protection from priority inversion block. The priority inheritance protocol ensures that priority inversion is controlled and the blocking time for a task is bounded.

Not as obvious, but just as important, are mechanisms that have been created to synchronize and communicate between tasks. While also found in non-RTOSs, these mechanisms take on a critical role in embedded systems due to the requirements on response and the scarcity of resources. The well known synchronization mechanisms are semaphores, mutexes, and condition variables, with message queues and mailboxes being among the more common task communication devices. But just having these mechanisms is not sufficient. These mechanisms must be designed in such a way as to take a bounded amount of time for worst case situations. For example, if a set of tasks have to wait for a semaphore in a wait queue, the wait queue should not be singularly linked, since removing a task from the list will require traversing the entire list of waiting tasks. Some more efficient algorithm, with bounded worst case performance, must be used.

Also the intimacy of the RTOS with the hardware imposes unique requirements. Embedded developers need flexibility and ease of use in managing IO devices along with minimal overhead from the Operating System. For example, a serial device driver may need to provide a synchronous or asynchronous driver interface and process ASCII or binary characters in buffered or non-buffered mode. Also the OS must provide timer services that allow a thread to wait for a message or semaphore for a fixed timeout value or the ability to sleep for a specified time. They should also allow for multiple timers to be set by the user threads to make state transitions or notify the application of any system failure.

An example of a commercial RTOS architecture is given in Figure 1 for pSOSystem from Integrated Systems, Inc.

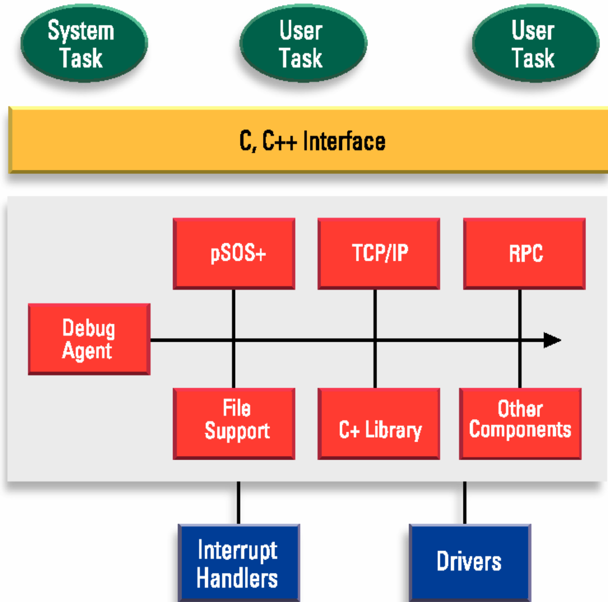

Figure 1

pSOSystem uses a modular architecture, containing the pSOS+ real-time, multi-tasking kernel and a collection of companion software components and libraries. These components are delivered as "black boxes," remaining unchanged from application to application. This assures high reliability to the end user.

With networking a key element in most of today's embedded applications, pSOSystem also offers a broad range of protocol support and networking features to embed TCP/IP, SNMP, Internet utilities, or multi-protocol environment. These features are also supplied as components, as shown in Figure 2, to be included in the application if required.

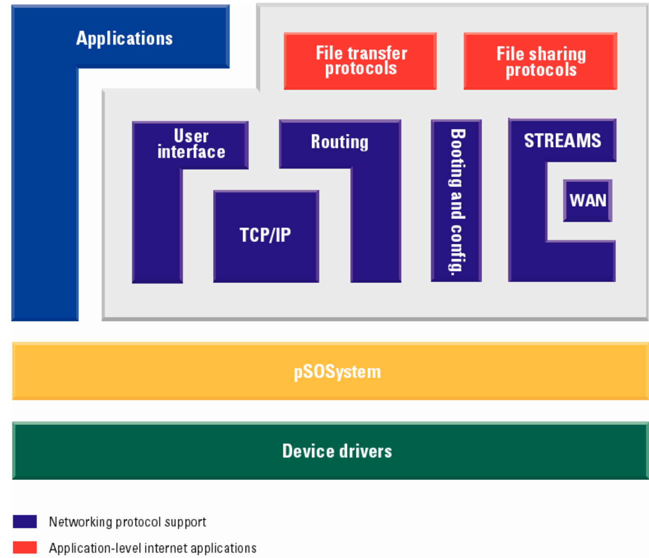

Figure 2

4. Popular Commercial RTOSs

A review of the popular RTOSs is beyond the scope of this paper, but there have been several compendiums provided to do just that. For example, the RTOS Buyer's Guide[3] characterizes more than 70 products to assist you in choosing an RTOS that best suits your needs. They have been classified into three large groups:

1. Non-commercial RTOSs. They are mainly targeted towards small embedded systems. They are most likely free, but the support may be poor, or non-existent, and the list of supported devices may be short.
2. Real-time extensions to NT. These have a different philosophy, with two OSs running on the same platform.
3. Commercial RTOSs. These are divided into two: small embedded applications and large complex real-time applications.

A different analysis is found in a special report, "Selecting a Real Time Operating System[4]." This article proposes categorizing the environment of the embedded system (for example, questions include: Are interrupt routines more important than memory management or queuing? or Will there be a lot of time-based functions?) and bases the RTOS decision on the characterizations.

---

[3] *Real Time Magazine*, 97-3

[4] *Embedded Systems Programming*, April 1996

Yet another report[5] deals with the issues in analyzing a make/buy decision on an RTOS, and if the decision is to buy, then how to choose a commercial RTOS vendor. The article discusses this choice based on criteria such as footprint, performance, software components and device driver support, debugging tools, standards compatibility, technical support, availability of source code, licensing and vendor reputation. While there are advantages to designing one's own RTOS, the article clearly makes the point that choosing a commercial product can save a lot of effort, and provide a product that will be considerably more bug free.

In addition to these overview articles, Real Time Magazine has run a series of more detailed reports, covering one RTOS with each article. The series began with the 98-3 issue, with an article on how the evaluation framework was set up, how the test plan was laid out, and a description of the measurement procedure. The Evaluation Project was launched with reports of Windows NT and its real-time extensions. Real Time Consulting also maintains the RTOS Buyers Guide.

One observation that the 98-4 issue of Real Time Magazine makes is the proliferation of RTOS products available. In determining the products that the reader audience base wanted to see evaluated, 13 products garnered 65%, with all others getting a healthy 35%. Only three products stood out, pSOSystem, VxWorks and QNX Neutrino, with about equal 11-12% of the responses.

5. Design Issues for Embedded Systems

Embedded Systems are, if nothing else, characterized by constraints such as response, size, performance, costs, and so on. And it is optimizing for these constraints (or rather perhaps working within them) that makes designing an embedded system a difficult task. Numerous questions have to be answered before the design even begins:

- What are the worst case performance requirements for each activity?
- What are the number and complexity of activities to be performed?
- How should these activities be distributed amongst the software tasks so that the processor load is balanced (and thereby get the best cost/performance out of the processor selection)?
- What is the degree of coupling of these tasks (critical deadlines, type of data flow among tasks, event interdependencies)?
- How much RAM and ROM does the hardware design provide?
- How much RAM and ROM will be consumed for the specific set of tasks, ISRs, queues, and so on?
- When should queues and messages be used to allocate resources, and how should these be used to avoid priority inversion deadlock?
- How much buffer space should be allocated for stack usage?

Over the years, the articles in *Embedded Systems Programming* magazine have dealt with many, if not most, of the issues facing embedded systems designers. A number of the more commonly faced issues are summarized here.

5.1 Time Constraints

Real-time operating systems bow to a combination of time specific constraints[6]. Some routines must execute at precisely fixed intervals, while other routines are not bound to a critical time alignment. The most critical task of an embedded programmer is to characterize each of the actions to be performed so he will know how to assign priority and resources to that action in order that the overall system performance objectives are met.

To aid in this task, it is helpful to break the actions in an embedded system down into the following four task groups:

1. Time critical task routines are those that must occur at a fixed rate with a minimum startup latency (e.g., servicing an A/D converter).
2. Time sensitive task routines are different from time critical tasks in that they can tolerate a large latency before being serviced. Like time critical task routines, they may also occur at fixed rates or they may be initiated at random intervals, but are guaranteed to execute no more frequently than some fixed rate by the task handler itself (e.g., debouncing front panel switches).
3. Idle task routines are important background operations, and they execute as frequently as possible–at more or less random intervals–when it is convenient (e.g., using the task class to poll a UART transmit buffer status to determine when to transfer a character from a software queue to the transmit buffer).
4. Mainline tasks routines interpret the user commands, perform non-real-time functions, and make calls to the time sensitive and idle task service routines.

5.2 Safety

While the reliability of hardware has improved dramatically, when the mission of the embedded system is critical[7], the embedded designer must build tests of the processor and memory into the application. There are a variety of ways that this can be accomplished.

Probably the first and simplest safety technique learned by many embedded programmers consists of filling unused program memory with either halt instructions or illegal instructions. This technique guards against illegal jumps outside of the program space and provides cheap insurance.

Another common protection is to use buffers that guard against stack underflow/overflow or the corruption of a task's stack. Many of the commercial RTOSs now contain facilities and functions that support stack checking.

To verify the integrity of a program or data stored in ROM, a simple ROM test should be included, as well a watchdog timer to prevent the software from getting caught in a loop.

It is also well known that a rogue pointer, for example, can lead to wholesale corruption of memory. So how does one protect against

[5] *Embedded Systems Programming*, March 1999

[6] "Designing RTOSs for Embedded Microcontrollers," Robert Richards, *Embedded Systems Programming*, May 1997

[7] "Solving the Software Safety Paradox," Doug Brown, *Embedded Systems Programming*, Dec 1998

the corruption of program data? One technique is the redundant storage of critical variables, and comparison prior to being used. Another is the grouping of critical variables together and keeping a CRC over each group.

5.3 Device Drivers

It is well known that writing efficient device drivers requires knowledge of both hardware and software[8]. The resulting device drivers become the keys to embedded system performance since they are called repeatedly, and therefore dictate real-time performance in terms of response time, and utilization of memory and other system resources. While this article promotes the use of a commercial tool for configuring the devices integrated within the modern generation of microcontrollers, it effectively makes the point that although perhaps 10% of the code in an embedded system is device drivers, writing device drivers for more complex systems may require over 30% of the development time. It furthermore makes the obvious point that to achieve the optimal, device drivers should take maximum advantage of a microcontroller's capabilities.

5.4 Interrupt Service Routines

Using interrupt processing is a powerful technique that is often more appropriate than using software loops to continuously poll peripheral devices[9]. However, interrupt processing strategy is not dictated by the compiler, and most RISC processors do very little in response to interrupts. These constraints place a burden on the embedded developer in that he must decide which interrupt architecture is best. Some approaches are to save the interrupted context on a memory stack. Another is to preserve the context in a cache, be it on-chip registers (if there are a lot of them to use) or off-chip memory. To simplify debugging, it is best to keep ISRs short.

5.5 Storage Allocation

One important feature to be considered in the selection of an RTOS or embedded system design is storage allocation[10]. Ill-designed dynamic storage allocation can be wasteful for two reasons. First, allocating memory from the heap can be both slow and non-deterministic. The time it takes for the memory manager to search the free-list for a block of the right size may not be bounded. Second, one may create the possibility of a memory allocation fault caused by a fragmented heap. One typical solution is to statically declare all objects up front and get rid of dynamic allocation. However, this can waste storage since the objects always exist and take space. Whilst difficult, the apparently conflicting goals of a dynamic storage allocator can be achieved.

5.6 Optimizing Performance

Writing embedded code that runs efficiently brings about a whole new set of rules[11]. Often optimizing for speed and size are opposing design goals–an improvement on one often degrades the other. In trying to achieve this balance, the article promotes the use of three techniques: (i) the judicious use of the optimization options found with most embedded cross-platform compilers (for example, eliminating redundant code, or replacing operations with equivalent but faster operations, or "unrolling loops," optimizing the use of registers, or removing code segments that the compiler knows cannot be reached); (ii) the mix of fixed and floating-point operations; and (iii) the employment of user optimizations, making the most out of available resources.

5.7 Debugging Memory Problems

Since many RTOSs and/or embedded microprocessors do not support memory protection, tracking down software memory bugs can become a serious debugging problem[12]. In attacking this problem, it is best to categorize the problem by the type of memory affected. In general, they fall into three categories:

- Global memory bugs: those bugs that result in corruption of global memory data areas.
- Stack memory bugs: these often cause a complete failure of the program execution; they are the hardest to track down as they are often a function of external events and the current state of the stack.
- Dynamically allocated memory bugs: examples are, heap memory allocated by a malloc service; or problems caused by writing past the boundaries of an allocated memory block or using one that is no longer allocated.

The article proceeds to discuss various approaches to debugging and protection, depending on the category of the memory problem.

6. Future Trends in Embedded System RTOS

There is a flood of trends rushing through the embedded market today, many influencing the RTOS requirements in conflicting ways. It is hard to envision that five years from now RTOS products will bear much resemblance to what is supplied today. Some of these trends are application driven while others are device driven, and it is important to understand the influences these trends will have.

6.1 Application Specific

In several markets, the end users have banded together to issue specific requirements for RTOSs to be used in their future products. They have purposely chosen to drop their proprietary behaviors of the past in order to get the benefits of multiple suppliers and interoperability of software.

Probably the best example of this today is OSEK, a specification issued by OSEK/VDX, an automotive consortium aimed at all embedded controllers for future cars. The specification deals with kernel API, communications protocol support and network management. The specification is independent of word-length, being equally applicable to 8-, 16-, and 32-bit microprocessors. By being very specific about what services OSEK is to support, and which it is not, the resulting implementation has a very low RAM and ROM requirement. In addition, an OSEK application is configured statically, at compile time, with the OSEK Implementation Language (OIL) used to define the configuration.

---

[8] "The Hazards of Device Driver Design," Shaul Gal-Oz, *Embedded Systems Programming*, May 1997

[9] "An Uncommon ISR Technique," Daniel Mann, *Embedded Systems Programming*, Jan 1995

[10] "An Efficient Dynamic Storage Allocator," David Lafreniere, *Embedded Systems Programming*, Sept 1998

[11] "Optimizing C Code in a Real Time Environment," Mark Kraeling, *Embedded Systems Programming*, Jan 1996

[12] "Chasing Down Memory Bugs," Bill Lamie, *Embedded Systems Programming*, April 1996

In this manner, only the needed software is linked into the application, preventing additional overhead and allowing for an extremely efficient kernel implementation.

In many ways, the resulting OSEK kernel is a throw back to embedded system RTOSs of 10 years ago. A review of what is supported and what is not provides a view of its focused design:

- Two types of tasks exist, basic and extended, where basic will not be able to wait on objects.
- Priority based scheduling is used, with priority statically assigned to task, which allows for non-preemptive task and priority ceiling protocol for priority inversion protection.
- No memory management is supported.
- Central error handler for system calls are used, with "standard" and "extended" status.

Several of the major RTOS vendors have announced OSEK designs for the automotive industry, and interest is being shown in the possible use of the OSEK kernel for consumer applications as well.

Two other examples are indicative of the application specific trend as well. A set-top box consortium has been working with CableLabs for specifying a variety of common application APIs for set-top boxes. So far, they have yet to specify RTOS characteristics, but there is a strong feeling that they will. A second trend, though not perhaps in the same category, is the growing requirement for "high-availability" features within the datacom industry. As in the other cases, these end users have historically built their own proprietary high availability features on top of standard commercial or in-house RTOSs. But with the availability of CompactPCI "hot swap" standards and the pressure to meet time-to-market demands, these end users are seeking commercial RTOSs with full high availability feature sets. Again, several major RTOS vendors have announced conforming products.

What is clear from this, is that no one RTOS product will span the range from the minimalist OSEK specification to the maximal high availability RTOS. The best that can hoped for is that RTOS designs evolve that have much higher degrees of scalability than exist today. But will this complexity come at the price of the "to die for" reliability that commercial RTOSs have had to this point? And how will common RTOS APIs (desired by the RTOS suppliers) be maintained when the end user API specifications are different?

6.2 System On A Chip (SOC)

As mentioned earlier, SOCs are beginning to appear throughout the embedded market, in at least three different ways. First, the semiconductor suppliers are providing developers the ability to pick and choose from a combination of industry standard functions integrated around a 32-bit core processor. These functions may include memory, IO drivers, a bus interface, network protocol support, or algorithms for special functions, such as an MPEG decoder. Second, end product manufacturers are integrating custom ASICs with common 32-bit core processors to provide complete solutions. Some recent examples include cable modems and ATM switches. And third, startups are emerging that will provide custom design services, complete with optimized RTOS, compiler, and debuggers.

SOC will be particularly well suited for a whole range of consumer electronics and wireless communications devices where size, cost, and low power requirements are crucial. It will also drive cost reductions in networking and telecom equipment, where more functionality can be added at lower costs.

A subset of this SOC trend is the emergence of multi-core devices on single silicon. The most common to date has been the combination of standard microprocessors and Digital Signal Processors (DSPs). Commercial versions already exist from TI and Hitachi, and more will definitely follow. In some cases, the DSPs are dedicated function processors, but emerging trends have the DSP as a full programmable device.

The challenge for the commercial RTOS suppliers will be to keep pace with the proliferation of processor configurations that will result. For one, this will involve close partnerships between silicon and software suppliers, and the integration of the software with the silicon, breaking the existing model of separately supplied software. While this integration seems to run counter to the semiconductor industry norm of supporting a variety of RTOSs with new microprocessors, it is hard to see how this can continue. And second, there must be closer coordination (or consolidation) amongst the RTOS and other tools suppliers, since these devices have radical implications with regard to compilers and debuggers.

6.3 Automatic Code Generation

Probably the most radical notion is the idea that application code can be generated automatically from graphical diagrams depicting the logic flow for the end product. To a limited extent, this has already been accomplished, with products like MATRIX$_x$®, BetterState™, Statemate™, and MATLAB® being used for application modeling and code generation. In the case of MATRIX$_x$, flight ready code for the international space station has been used for some time now, and the technology is being extended into the more restrictive automotive market.

If these tools were to become reality, the whole notion of commercial RTOS and development tools will be upset, as the developer will only interact with the graphical tool, and will be totally isolated from the resulting software implementation.

7. Summary

While perhaps not up to the multiples being shown by the ".com" companies, commercial suppliers of products for the embedded space have received a lot of investor and press attention. The reason is simple: embedded devices, including those that will interact with and support the internet, will continue to grow in number and function as the unstoppable trend to higher integration and lower prices makes the use of embedded processors routine. Supporting this trend is a large, fragmented collection of tool suppliers for embedded developers. Among this collection are those supplying Real-Time Operating Systems. While the basic characteristics of RTOSs are common, there are a lot of differences as well, and these are tracked and catalogued by a number of trade magazines and services. An embedded developer faces the task of not only deciding what commercial RTOS and other tools to use, but of learning a programming environment that might be completely foreign. He must deal with resource constraints, hardware device drivers, interrupt routines, memory allocation and a host of other issues that as a systems developer, he probably never had to face.