Abstract – Mechanical functions and controls of industrial atomizing devices being replaced by Real time controls. The result is efficient, safe, convenient and reliable in industrial applications. Real time Embedded technology uses many embedded languages with real time operating systems. Software in real time embedded systems differs fundamentally from its desktop or internet counterparts. Embedded computing is not simple computation on small devices. In most control applications for example, embedded software engages the physical world. It reacts to physical and user-interaction events, performs computation on limited and competing resources, and produces results that further impact the environment. Of necessity, it acquires some properties of the physical world, most particularly, time.

I. INTRODUCTION

Control application is very much essential for real time based applications. This paper describes an embedded software design in a real time operating system (PSOS) to controlling a robotic arm movement. Real time system typically needs to perform timed multitasking. Timed multitasking modes are simple: since the activation of a task depends either on other tasks or on interrupts, by controlling the time at which outputs are produced and triggering tasks with new events, we can effectively control both the starting and stopping time of each task, thus obtaining deterministic timing properties.

pSOSystem - pSOSystem is a modular, high-performance real-time operating system designed specifically for microprocessors. It provides a complete multitasking environment based on open systems standards. pSOSystem is designed to meet three overriding objectives:
- Performance
- Reliability
- Ease-of-Use

The result is a fast, deterministic, yet accessible system software solution. Accessible in this case translates to a minimal learning curve. pSOSystem is designed for quick startup on both custom and commercial hardware. The pSOSystem software is supported by an integrated set of cross development tools that can reside on UNIX- or DOS-based computers. These tools can communicate with a target over a serial or TCP/IP network connection.

System Architecture

The pSOSystem software employs a modular architecture. It is built around the pSOS+ real-time multi-tasking kernel and a collection of companion software components. Software components are standard building blocks delivered as absolute position-independent code modules. They are standard parts in the sense that they are unchanged from one application to another. Unlike most system software, a software component is not wired down to a piece of hardware. It makes no assumptions about the execution/target environment. Each software component utilizes a user-supplied configuration table that contains application-and hardware-related parameters to configure itself at startup. Every component implements a logical collection of system calls. To the application developer, system calls appear as re-entrant C functions callable from an application. Any combination of components can be incorporated into a system to match your real-time design requirements. The pSOSystem components are listed below.
- pSOS+ Real-time Multitasking Kernel. A field-proven, multitasking kernel that provides a responsive, efficient mechanism for coordinating the activities of your real-time system.
- pSOS+m Multiprocessor Multitasking Kernel. Extends the pSOS+ feature set to operate seamlessly across multiple, tightly-coupled or distributed processors.
- pNA+ TCP/IP Network Manager. A complete TCP/IP implementation including gateway routing, UDP, ARP, and ICMP protocols; uses a standard
socket interface that includes stream, datagram, and raw sockets.
pRPC+ Remote Procedure Call Library. Offers SUN-compatible RPC and XDR services; allows you to build distributed applications using the familiar C procedure paradigm.
pHILE+ File System Manager. Gives efficient access to mass storage devices, both local and on a network. Includes support for CD-ROM devices, MS-DOS compatible floppy disks, and a high-speed proprietary file system. When used in conjunction with the pNA+ component and the pRPC+ subcomponent, offers client-side NFS services.
pREPC+ ANSI C Standard Library. Provides familiar ANSI C run-time functions such as printf(), scanf(), and so forth, in the target environment.

In addition to these core components, pSOSystem includes the following:
Networking protocols including SNMP, FTP, Telnet, TFTP, NFS, and STREAMS
Run-time loader
User application shell
Support for C++ applications
Boot ROMs

Automatic tracking of program execution through source code files.
Traces and breaks on high-level language statements.
Breaks on task state changes and operating system calls.
Monitoring of language variables and system-level objects such as tasks, queues and semaphores.
Profiling for performance tuning and analysis.
System and task debug modes.
The ability to debug optimized code.

The pROBE+ debugger, in addition to acting as a back end for a high-level debugger on the host, can function as a standalone target-resident debugger that can accompany the final product to provide a field maintenance capability.

pSOS+ Real-Time Kernel
This chapter focuses primarily on concepts relevant to a single-processor system. The pSOS+ kernel is a real-time, multitasking operating system kernel. As such, it acts as a nucleus of supervisory software that performs services on demand, schedules, manages, and allocates resources generally coordinates multiple, asynchronous activities. The pSOS+ kernel maintains a highly simplified view of application software, irrespective of the application’s inner complexities. To the pSOS+ kernel, applications consist of three classes of program elements:

Tasks
I/O Device Drivers
Interrupt Service Routines (ISRs)

Multitasking Implementation
A multitasked system is dynamic because task switching is driven by temporal events. In a multitasking system, while tasks are internally synchronous, different tasks can execute asynchronously. Figure 2-1 illustrates the multitasking kernel. A task can be stopped to allow execution to pass to another task at any time. In a very general way, Figure 2-1 illustrates multitasking and how it allows interrupt handlers to directly trigger tasks that can trigger other tasks.

Figure 1-Psos Module architecture

Integrated Development Environment
The pSOSystem integrated cross-development environment can reside on a UNIX or DOS-based computer. It includes C and C++ optimizing compilers, a target CPU simulator, a pSOS+ OS simulator, and a cross-debug solution that supports source and system-level debugging. The pSOSystem debugging environment centers on the pROBE+ system-level debugger and optional high-level debugger. The high-level debugger executes on your host computer and works in conjunction with the pROBE+ system-level debugger, which runs on a target system. The combination of the pROBE+ debugger and optional host debugger provides aMultitasking debug solution that features:
A sophisticated mouse and window user interface.
Thus, a multitasked implementation closely parallels the real world, which is mainly asynchronous and/or cyclical as far as real-time systems apply. Application software for multitasking systems is likely to be far more structured, race-free, maintainable, and re-usable. Several pSOS+ kernel attributes help solve the problems inherent in real-time software development. They include pSOSSystem System Concepts pSOS+ Real-Time Kernel.

**Concept of a Task**

From the system’s perspective, a task is the smallest unit of execution that can compete on its own for system resources. A task lives in a virtual, insulated environment furnished by the pSOS+ kernel. Within this space, a task can use system resources or wait for them to become available, if necessary, without explicit concern for other tasks. Resources include the CPU, I/O devices, memory space, and so on. Conceptually, a task can execute concurrently with, and independent of, other tasks. The pSOS+ kernel simply switches between different tasks on cue. The cues come by way of system calls to the pSOS+ kernel. For example, a system call might cause the kernel to stop one task in mid-stream and continue another from the last stopping point. Although each task is a logically separate set of actions, it must coordinate and synchronize itself, with actions in other tasks or with ISRs, by calling pSOS+ system services.

**Overview of System Operations**

pSOS+ kernel services can be separated into the following categories:
- Task Management
- Storage Allocation
- Message Queue Services
- Event and Asynchronous Signal Services
- Semaphore Services
- Time Management and Timer Services
- Interrupt Completion Service
- Error Handling Service
- Multiprocessor Support Services

**Task States**

A task can be in one of several execution states. A task’s state can change only as result of a system call made to the pSOS+ kernel by the task itself, or by another task or ISR. From a macroscopic perspective, a multitasked application moves along by virtue of system calls into pSOS+, forcing the pSOS+ kernel to then change the states of affected tasks and, possibly as a result, switch from running one task to running another. Therefore, gaining a complete understanding of task states and state transitions is an important step towards using the pSOS+ kernel properly and fully in the design of multitasked applications. To the pSOS+ kernel, a task does not exist either before it is created or after it is deleted. A created task must be started before it can execute. A created-but-unstarted task is therefore in an innocuous, embryonic state. Once started, a task generally resides in one of three states:
- Ready
- Running
- Blocked

A ready task is runnable (not blocked), and waits only for higher priority tasks to release the CPU. Because a task can be started only by a call from a running task, and there can be only one running task at any given instant, a new task always starts in the ready state. A running task is a ready task that has been given use of the CPU. There is always one and only one running task. In general, the running task has the highest priority among all ready tasks; unless the task’s preemption has been turned off, as described. A task becomes blocked only as the result of some deliberate action on the part of the task itself, usually a system call that causes the calling task to wait. Thus, a task cannot go from the ready state to blocked, because only a running task can perform system calls.

**State Transitions**

Figure depicts the possible states and state transitions for a pSOS+ task. Each State transition is described in detail below. Note the following abbreviations:
- E for Running (Executing)
- R for Ready
- B for Blocked
Figure 3 - State Transition Diagrams

Task Scheduling
The pSOS+ kernel employs a priority-based, preemptive scheduling algorithm. In General, the pSOS+ kernel ensures that, at any point in time, the running task is the one with the highest priority among all ready-to-run tasks in the system. However, you can modify pSOS+ scheduling behavior by selectively enabling and disabling preemption or time-slicing for one or more tasks. Each task has a mode word with two settable bits that can affect scheduling. One bit controls the task’s preemptibility. If disabled, then once the task enters the running state, it will stay running even if other tasks of higher priority enter the ready state. A task switch will occur only if the running task blocks, or if it re-enables preemption. A second mode bit controls time slicing. If the running task’s time slice bit is enabled, the pSOS+ kernel automatically tracks how long the task has been running. When the task exceeds the predetermined time slice, and other tasks with the same priority are ready to run, the pSOS+ kernel switches to run one of those tasks.

Task Priority
A priority must be assigned to each task when it is created. There are 256 priority levels — 255 is the highest, 0 the lowest. Certain priority levels are reserved for use by special pSOSystem tasks. Level 0 is reserved for the IDLE daemon task furnished by the pSOS+ kernel. Levels 240 - 255 are reserved for a variety of high priority tasks, including the pSOS+ ROOT. A task’s priority, including that of system tasks, can be changed at runtime by calling the t_setpri system call. When a task enters the ready state, the pSOS+ kernel puts it into an indexed ready queue behind tasks of higher or equal priority. All ready queue operations, including insertions and removals, are achieved in fast, constant time. No search loop is needed. During dispatch, when it is about to exit and return to the application code, the pSOS+ kernel will normally run the task with the highest priority in the ready queue. If this is the same task that was last running, then the pSOS+ kernel simply returns to it. Otherwise, the last running task must have either blocked, or one or more ready tasks now have higher priority. In the first (blocked) case, the pSOS+ kernel will always switch to run the task currently at the top of the indexed ready queue. In the second case, technically known as preemption, the pSOS+ kernel will also perform a task switch, unless the last running task has its preemption mode disabled, in which case the dispatcher has no choice but to return to it. Note that a running task can only be preempted by a task of higher or equal (if timeslicing enabled) priority. Therefore, the assignment of priority levels is crucial in any application. A particular ready task cannot run unless all tasks with higher priority are blocked. By the same token, a running task can be preempted at any time, if an interrupt occurs and the attendant ISR unblocks a higher priority task.

Task Control Block
A task control block (TCB) is a system data structure allocated and maintained by the pSOS+ kernel for each task after it has been created. A TCB contains everything the kernel needs to know about a task, including its name, priority, remainder of timeslice, and of course its context. Generally, context refers to the state of machine registers. When a task is running, its context is highly dynamic and is the actual contents of these registers. When the task is not running, its context is frozen and kept in the TCB, to be restored the next time it runs. There are certain overhead structures within a TCB that are used by the pSOS+ kernel to maintain it in various system-wide queues and structures. For example, a TCB might be in one of several queues — the ready queue, a message wait queue, a semaphore wait queue, or a memory region wait queue. It might additionally be in a timeout queue. At pSOS+ kernel startup, a fixed number of TCBs is allocated reflecting the maximum number of concurrently active tasks specified in the pSOS+ Configuration Table entry kc_ntask. A TCB is allocated to each task when it is created, and is reclaimed for reuse when the task is deleted. Memory considerations for TCBs are given in the “Memory Usage” chapter of the pSOSystem Programmer’s Reference. A task’s Tid contains, among other things, the encoded address of the task’s TCB. Thus, for system calls that supply Tid as input, the pSOS+ kernel can quickly locate the target task’s TCB. By convention, a Tid value of 0 is an alias for the running task. Thus, if 0 is used as the Tid in a system call, the target will be the calling task’s TCB.

The Idle Task
At startup, the pSOS+ kernel automatically creates and starts an idle task, named IDLE, whose sole purpose in life is to soak up CPU time when no other task can run. IDLE runs at priority 0 with a stack

ISSN: 2231-2803 http://www.ijcttjournal.org
allocated from Region 0 whose size is equal to kc_rootsst. On most processors, IDLE executes only an infinite loop. On some processors, pSOS+ can be configured to call a user-defined routine when IDLE is executed. This user-defined routine can be used for purposes such as power conservation. See “pSOS+ and pSOS+m Configuration Table Parameters” in pSOS System Programmer’s Reference for more details. Though simple, IDLE is an important task. It must not be tampered with via t_delete, t_suspend, t_setpri, or t_mode, unless you have provided an equivalent task to fulfill this necessary idling function.

II. TM programming concepts

The building blocks in the TM model are actors with further annotations. Actors in a TM model not only declare their computing functionality, but also specify their execution requirements in terms of trigger conditions, execution time, and deadlines.

Dead lines

The deadline of a reaction is expressed as a real-time value, indicating that the computational results are produced if and only if the deadline is reached. Explicitly expressing deadlines has two benefits: By knowing the deadlines of all actors and their triggering dependencies, designers know exactly what time delay their programs will introduce at run time, so that they can be more confident in choosing algorithm parameters. Explicit deadlines are also useful in resource-aware algorithms, such as anytime algorithms, which can provide results with different fidelities, depending on the computational time available.

If a deadline is not specified, the results will be produced as soon as the computation has finished. This is useful to handle soft real-time actors or intermediate computational steps to provide prompt triggering for downstream actors.

Execution time

An actor also declares its execution time, which is the amount of time it needs to finish its reaction, provided that it is not preempted by any other actors. Unlike the deadline, which is given in terms of real time no matter how many times the actor is preempted, the execution time spent on the actor’s execution.

Software synthesis

Building real-time embedded software using actors and the timed-multitasking model allows certain levels of analysis of timing properties and generation of run-time software and scheduling. This formal step preserves the TH execution semantics and reduces the burden of writing error-prone code for event queues and task synchronization.

III. Conclusion

This paper reviewed the challenges of developing embedded software for real-time control systems. We further described the timed multitasking model that uses an event-triggering mechanism and deadlines to provide deterministic timing behavior for embedded software. The actor-oriented programming model and a highly structured communication style allowed interface and scheduling code to be generated from the high-level specification. An industrial tool control system was designed as an example.

IV. Reference


