Portable Multitasking Real−Time Kernel Design and Implementation on DSP Systems

Table of Contents

Introduction....4
Part I Multitasking Real−Time Kernel....6
  Chapter One Task Management....8
    1.1 Definition of Task....8
    1.2 Types of Task Constraints....10
    1.3 Scheduling Policies....11
  Chapter Two Critical Section and Synchronization....15
    2.1 Race Condition....15
    2.2 Critical Section....16
    2.3 Busy Waiting....16
    2.4 Semaphores....18
  Chapter Three Interprocess Communication....21
    3.1 Shared Memory....21
    3.2 Message Pipe....22
    3.3 Mail Box....23
  Chapter Four Device Driver....24
    4.1 Goals and Benefits....24
    4.2 Device Driver Framework....25
    4.3 Blocking and Non−Blocking I/O....28
Part II Implementation on TI 320F24x DSP : Taunix....30
  Chapter Five Task Management....31
    5.1 Task Control Block....31
    5.2 Priority−based Task Scheduling....32
    5.3 Periodic Task Scheduling....34
    5.4 Data Structures and Member functions of Scheduling System....35
  Chapter Six Interrupt Handling....38
    6.1 TI 320F243 Interrupt Controller....38
    6.2 Taunix ISR Implementation....40
    6.3 Interaction Between ISR and Scheduler....43
  Chapter Seven Device Drivers....44
    7.1 Device Switch....44
    7.2 Non−Blocking I/O Support....45
    7.3 Supported Devices....46
  Chapter Eight Inter−Process Communication....47
    8.1 Semaphore....47
    8.2 Data Structure and Member Functions of Semaphore....49
    8.3 Message Pipe....50
    8.4 Message Queue....51
    8.5 Data Structures and Member Functions of Message Pipe and Queue....51
  Chapter Nine Communication....53
    9.1 SECS−I protocol....53
    8.2 SECSd implementation....54
    8.3 Data Structures and Member Functions of SECS daemon....57
  Chapter Ten Future Works....59

10.1 Taunix Virtual File System..59
10.2 System Call Library..59
10.3 Installable Scheduling System...60
10.4 Architecture Porting...60
10.5 Going to GPL...61
Chapter Eleven Benchmarking Index...62
Example: Motor Control...63
References...66
Appendix A : GNU General Public License...67
Appendix B : Taunix Source Code List...72

Figure List

Figure 1−1 Single−tasking and Multi−tasking..........8
Figure 1−2 Task states transitions..........9
Figure 2−1 Race condition..........15
Figure 3−1 Shared memory..........21
Figure 3−2 Data flow through a pipe..........22
Figure 3−3 Send message to mail box..........23
Figure 4−1 Role of the device drivers..........25
Figure 4−2 Blocking I/O sequence..........28
Figure 4−3 Non−blocking I/O sequence..........29
Figure II−1 Taunix Architecture..........30
Figure 5−1 Jumping Buffer..........31
Figure 5−2 Taunix Task Table..........33
Figure 5−3 Task State Transition..........34
Figure 6−1 TI 320F241, F242, F243 PIE..........39
Figure 6−2 Interrupt Handling Flow..........40
Figure 6−3 Interrupt Keyword Function..........41
Figure 7−1 Non−blocking I/O operation flow..........46
Figure 8−1 Message pipe operation..........50
Figure 8−2 Message queue operation..........51
Figure 9−1 SECS packet format..........54
Figure 9−2 SECS protocol block diagram..........54
Figure 9−3 SECS virtual device..........55
Figure 9−3 Secsd event flow chart..........56
Figure 9−4 RX buffer..........57
Figure 10−1 SimTaunix Architecture..........60
Figure ex−1 Motor control system block diagram..........64

Introduction

Every good work of software
starts by scratching a developer’s
personal itch.

Embedded systems are around us. From the watch on the wrist, in the desktop computer, even to the spacecraft that explores the solar system. As the semiconductor fabrication technique is getting more and more advanced, the computation power of the micro−processor is also tremendously increased and are capable of complex jobs. Software design cannot be single threaded any more. Take a bubble−jet printer as an example. It has several position control systems, communication systems, user interface panel, color adjusting system, and ink injection control. While printing a document, all these systems have to work concurrently. The firmware inside this printer can be divided into five basic tasks to handle each objective. By sequential programming method, programmer would design a polling loop to go through each task once a time. This is not efficient, especially cannot guarantee real−time constraint. Hard to upgrade and maintain are also the characteristics of sequential programming. By multitasking approach, each task can virtually runs concurrently on one micro−processor and get the CPU time as it needs it. Each task can be modularized. This system architecture is similar to the desktop computer. Now, modern operating systems support multitasking. Methodologies used in the modern operating systems are well applicable in embedded system software design.

On the other hand, Information appliance will become the next wave that will impact human’s life. Well constructed embedded operating system will help application developing. More and more information access devices will be manufactured, each will run an embedded operating system. In industry, there are many kinds of kernels used for wide range of applications, such as QNX, pSOS, VRTX, Linux, PalmOS, and WinCE. Some need powerful hardware, some are not. In this thesis, the real−time operating system theorems will be reviewed and a real−time embedded kernel: Taunix will be implemented on a simple DSP system. In the end, there will be an example of motor control system based on this DSP platform that runs Taunix.

Keywords: real−time, embedded system, operating system, kernel, DSP.

Part I Multitasking Real−Time Kernel

‘‘Perfection (in design) is achieved
not when there is nothing more to
add, but rather when there is nothing
more to take away.’’

Kernel, or in other terms operating system (OS), is a kind of resource allocator in the computer system. Usually there are many resources in computer system, like CPU time, memory and I/O devices and these resources can be requested by programs. But , from time to time, requests may conflict, resources may not be available. Kernel is like a manager that allocate these resources to the requested programs properly. In order to fully utilize CPU computation power, multitasking is required. Multitasking is mechanism that virtually runs multiple tasks concurrently on a single CPU. With multitasking, kernel needs to decide which task should be pick up to run now. Tasks are somehow not absolutely isolated, either. Sometimes they need to be synchronized or exchange data. Besides, many devices will be attached to the computer system. Kernel needs to drive them precisely, also.

How about real−time? In real−time system, the correctness of the computation is not just the result, but also when the computation is completed. By the definition in Oxford Dictionary of Computing:

> An system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to the same movement. The lag from input time to output time must be sufficient small for acceptable timeliness.

Levi and Agrawala has a more practical definition:

> A real−time system is a set of concurrently executed computations (in some models called process) which interact with each other and adhere to some timing constraints.

General speaking, the time constraints of real−time systems can be divided into two categories:

1. Hard real−time: each job must be completed before deadline. Otherwise system fails
2. Soft real−time: It’s no harm that sometimes job will miss deadline but only decreases system performance.

In the following chapters, each major component of real−time multitasking kernel, scheduling policy, task synchronization, interprocess communication and device driver, will be discussed before implementation.

Chapter One Task Management

Real−time kernel has to do multitasking. The basic entity in a kernel is task. Although, in many operating system textbooks use process[8] as the basic entity, both can be exchangeable and the term '*task*' is used in this thesis.

When a single processor has to execute a set of concurrent tasks, that is tasks that can overlap in time, the CPU can be assigned to the various tasks according to a predefined criterion, called *scheduling policy*. The set of rules that, at any time, determines the order in which tasks are executed is called *scheduling algorithm*. The specific operation of allocating the CPU to a task selected by the scheduling algorithm is referred as *dispatching*.

1.1 Definition of Task

A task (or process) is a running program, including the current values of the program counter, registers, and variables. Conceptually, each task has its own virtual CPU, but in fact, CPU switches very fast from one task to another task to create pseudo parallelism. This rapid switching back and forth is called *multi−tasking* and the switching itself is called *context switch*. As a result, if there are several tasks in system waiting for executing, the sequence will not be serial but overlapped. This is shown in figure 1−1.

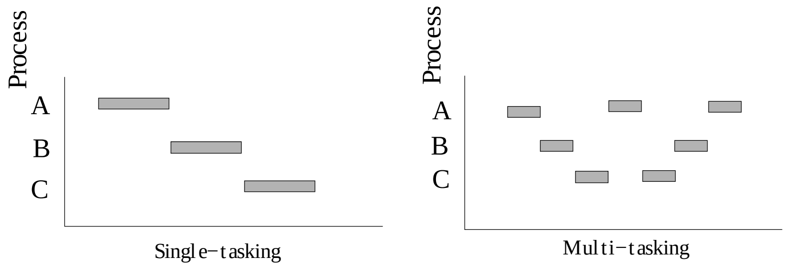


Figure 1−1 Single−tasking and Multi−tasking

As shown in figure 1−1, in multitasking system, task will not always running. Sometimes it will be interrupted and stopped by kernel, and another task will be chosen. This operation of suspending a running task is called *preemption*. In dynamic real−time system, preemption is important for three reasons[14]:

- Tasks performing exception handing may need to preempt existing tasks so that responses to exceptions may be issued in a timely fashion.
- When application tasks have different levels of criticalness expressing task importance, preemption permits to anticipate the execution of the most critical activities.
- More efficient schedules can be produced to improve system responsiveness.

Because of preemption, task will be experienced some states from creation to end of execution. General speaking, the states can be simplified to three[8]:

1. Running: actually using CPU at that instance.
2. Ready: runnable; temporarily stopped to let another task run.
3. Blocked: unable to run until external event happens.

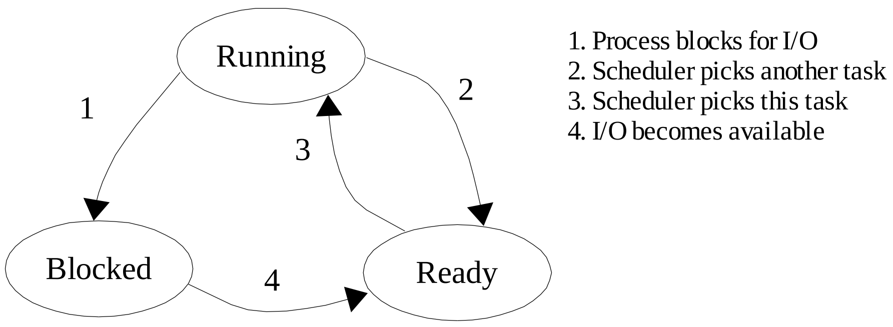

Running
Blocked
Ready
1
2
3
4
1. Process blocks for I/O
2. Scheduler picks another task
3. Scheduler picks this task
4. I/O becomes available

Figure 1−2 Task states transitions

As shown in the above figure, transition 1 tasks place when task can not continue, such as blocking on a slow peripheral or abandon CPU time by itself if there is an system call like `block()` or `suspend()` in operating system. Transition 2 and 3 are caused by *scheduler*. Scheduler is a set of program inside the operating system kernel that decides at when which task can be chosen to run by some predefined scheduling policy. It also can be thought as a CPU allocator. Transition 2 occurs when scheduler decides that the running task has run long enough or another higher priority task is in the ready state and waits for CPU. Transition 3 is happened when the task has higher enough priority or now is its turn to run. Transition 4 usually needs external event to trig, like data is ready from peripherals or resumed by another task. Task returned from blocked state will not directly been the running task. It returns to ready state and waits for scheduler to re−schedule.

1.2 Types of Task Constraints

Typical constraints that can be specified on real−time tasks are of three classes: timing constraints, precedence relations and mutual exclusion constraints on shared resources:

Timing constraints:

The typical timing constraint on task is the deadline, which represents the time before which a process should complete its execution without causing any damage to the system. Depending on the consequences of a missed deadline, real−time tasks can divided into two classes: hard real−time and soft real−time, as described before. Generally speaking, a real−time task has the following timing constraint parameters:

- Arrival time: the time at which a task becomes ready for execution.
- Computation time: the time necessary to the processor for executing the task without interruption.
- Deadline: the time before which task should be complete to avoid damage to the system.
- Start time: the time at which a task starts its execution.
- Finishing time: the time at which a task finishes its execution.
- Criticalness: parameter related to the consequences of missing the deadline. It could be hard or soft.
- Value: the relative importance of the task with respect to the other tasks in the system.

- Lateness: the delay of a task completion with respect to its deadline.
- Exceeding time: the time a task stays active after its deadline.
- Slack time: the maximum time a task can be delayed on its activation to complete within its deadline.

Another timing characteristic of real−time task is the regularity of its activation. Task can be defined as periodic or aperiodic.

Precedence constraints

In some applications, tasks cannot be executed in an arbitrary order. They have to respect some precedence relations defined at the design stage and be executed in a predefined order one by one.

Resource constraints

Resource is any software structure that can be used by task. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of peripheral device. A resource can be shared or dedicated to a particular task. To maintain data integrity, shared resources do not allow simultaneously access at the same time and require mutual exclusion among competing tasks. Operating systems must provide a synchronization mechanism that can be used by tasks to create critical sections of code. If two or more tasks have to access the same resource and need to be synchronized, it can say that these tasks has resource constraints.

1.3 Scheduling Policies

There are many scheduling policies proposed for different real−time problems, they can be identified as the following classes:

- **Preemptive**: With preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
- **Non−preemptive**: With non−preemptive algorithms, a task, once started, is

executed by the processor until completion. In this case, all scheduling decisions are taken as a task terminates its execution.

- **Static**: Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
- **Dynamics**: Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.
- **Off−line**: If the scheduler is executed on the entire task set before actual task activation, this is a off−line scheduler. The schedule generated in this way is stored in a table and later executed by a dispatcher.
- **On−line**: If scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates, then this scheduling policy is used on−line.
- **Optimal**: An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then an algorithm is said to be optimal if it may fail to meet a deadline only if no other algorithms of the same class can meet it.
- **Heuristic**: An algorithm is said to be heuristic if it tends toward but does not guarantee to find the optimal schedule.

Following are 3 commonly used real−time scheduling algorithm:

Rate monotonic priority scheduling algorithm

The Rate Monotonic(RM)[2] scheduling algorithm is a simple rule that assigns priorities to tasks according to their request frequencies(or request rates). Higher frequency task will have higher priority. RM is intrinsically preemptive and since the frequencies are fixed, RM is a fixed−priority assignment, priority of each task is assigned before activation and does not change over time.

In 1973, Liu and Layland showed that RM is optimal among all fixed−priority assignments in the sense that no other fixed−priority algorithms can schedule a task set that cannot be scheduled by RM. They also derived that the least upper bound of the processor utilization factor for a generic set of *n* periodic tasks will be:

$$u=\sum_{i=1}^{n} c_i \cdot f_i \leq n(2^{\frac{1}{n}}-1)$$

where:

n : the number of tasks

c_i: the computation time of task *i*

f_i: the frequency of task *i*

Earliest due date by Jackson’s algorithm

This algorithm considers not the request rate but the due date(or deadline). There are some constraints on tasks: all tasks consist of a single job, have synchronous arrival times, but can have different computation times and deadlines. No other constraints are considered, hence tasks must be independent. Jackson found an algorithm in 1955 and his rule tends to minimizing the maximum lateness among a set of *n* aperiodic tasks has to be scheduled on a single processor. It’s called *Earliest Due Date*(EDD)[14] and can be expressed by the following rule:

Jackson’s rule:

Given a set of n independent tasks, any algorithm that executes the tasks in order of non−decreasing deadlines is optimal with respect to minimizing the maximum lateness.

In other words, EDD will sort a set of *n* aperiodic tasks according to their deadlines in increasing order. To make sure all tasks can be feasibly scheduled by the EDD algorithm, the guarantee test can be performed by verifying the following *n* conditions:

$$\forall i=1,....,n \sum_{k=0}^{i} C_k \leq d_i$$

Earliest deadline first by Horn's algorithm

If tasks are not synchronous but can have arbitrary arrival times, Jackson's rule may not be suitable. In 1974, Horn found an elegant solution to the problem of scheduling a set of *n* independent tasks on a uniprocessor system, when tasks may have dynamic arrivals and preemption is allowed.

The algorithm, called *Earliest Deadline First*(EDF)[14], can be expressed by the following statement:

Horn's rule:

Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instance executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.

Unlike Jackson's rule, EDF will reorder task set once while inserting the newly arrival task. To make sure all tasks can be feasibly scheduled by the EDF algorithm, the guarantee test can be performed by verifying the following conditions:

$$\forall\, i=1,..,n \sum_{k=1}^{i} c_k(t) \leq d_i$$

where:

$c_k(t)$: the remaining worst−case execution time of task J_k

d_i: deadline of task J_i

Chapter Two Critical Section and Synchronization

In a multitasking system, tasks will read, write shared memory location, access resources. Without synchronization, the results will be unpredictable and system would fail. How to let two or more tasks access one resource without corrupting data is very important in multitasking and concurrent programming system.

2.1 Race Condition

Suppose two tasks, A and B, that can be executed concurrently and a shared variable, `index`, accessed by two tasks and initialize to zero. Task A has an infinite loop to increase `index` by 1 and task B is to decease `index` by 1. Then what is `index`'s value after 1−second's execution?

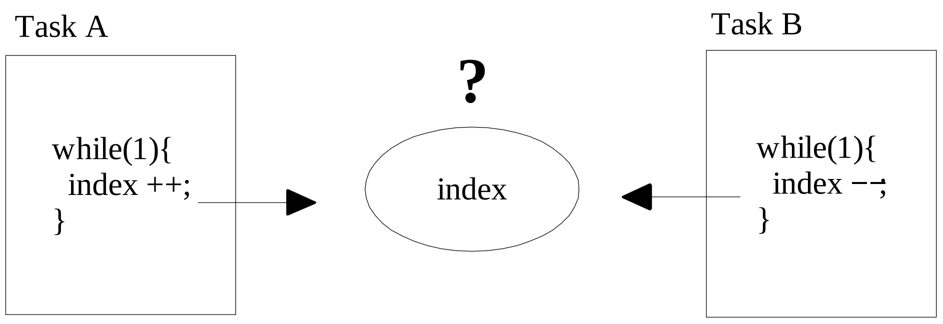


Figure 2−1 Race condition

In fact, there is no way to determine the result of `index`. At first, operating system may choose task A to run. It try to increase `index`. After a while, OS chooses task B. But the time slice for task A and B may not be the same. Meanwhile, there are other tasks in system, they need CPU's computation, too. To realize the value of index after 1−second's execution, one needs to accurately trace into OS's scheduling system to understand the execution order of each task and the time slices allocated. Conditions like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race condition**[8]. To avoid race condition, tasks need to access shared resources must be synchronized.

2.2 Critical Section

The idea to avoid race condition is very simple: find a mechanism to limit only one process to access shared resources at the same time. In other words, **mutual exclusion** is needed. For instance, OS chooses task A first and now task A is increasing `index`. For several time slices later, OS decides to choose task B to run. But now task A is accessing `index`, task B senses this situation, then it has to wait until task A stops to accessing `index` and vice versa.

The segment of program that accesses shared resource is called **critical section**[8]. If no two tasks can entering their critical sections at the same time, race condition will be avoided. To let the multitasking environment to run more correctly and efficiently using shared resources, there are four conditions need to hold:

1. No two tasks may be simultaneously inside their critical section.
2. No assumption may be made about speeds or the number of CPUs.
3. No task running outside its critical section may block other tasks.
4. No task should have to wait forever to enter its critical section.

With these constraints, now the mutual exclusion mechanism can be developed.

2.3 Busy Waiting

There are many algorithms to provide mutual exclusion mechanism. In this section two methods that are simplest and widely used in many systems will be discussed: disabling interrupts and lock variables with test−and−set−lock instruction.

Disabling Interrupts

The most straight−forward way to prohibit two or more processes into the same critical section is disabling all interrupts and re−enabling them before leaving. Since CPU uses interrupts to switch from task to task as the result of clock or other interrupts, with all interrupts disabled, CPU cannot notify operating system to do scheduling. This task can safely modify the shared resources as its wish without the fear that other tasks will interfere.

This solution is very simple, but destructive. What if one task disabled all interrupts and running forever or even crashed? Under this situation, interrupts cannot be restored and the whole system will stop. On the other hand, it's very convenient for kernel itself to disable interrupts for a very short time to update it's own data structures.

For a real−time kernel, even a short time of disabling interrupts sometimes will cause problems. If one interrupt occurs at that moment, kernel may miss it. In some implementations, it will insert so called *preemption point*[6] after re−enabling interrupts. At preemption point, kernel will check for any activities it has to respond during the critical section to make sure that no interrupt and no real−time request is missed. With hardware's support (test−and−set−lock instruction, see next section), a system that never disables interrupts is also achievable.

Lock Variables with Test−and−Set−Lock Instruction

The second way is a software solution, the lock variable. The basic idea is that: there is a single, shared lock variable, initially set to 0. When a task wants enter its critical section, it checks this lock first. If the lock is 0, this task has to set it to 1 and enters the critical section. But if the lock is 1, the task must wait until it becomes 0. As a result, 0 means no task is in critical section and 1 means some task is in its critical section.

The second simple way cannot protect the critical section, either because of its working flow. Lock variable needs two operations: test and set. When task A is try to enter its critical section, it checks the lock and get the value of 0. Therefore it considers that it can enter its critical section. But just before it sets it to 1, OS decides to switch to task B and this task is also trying to enter the same critical section. At this moment, task B will see the lock as 0, set it to 1 and enter the critical section. Before the task B completes its operation, task A is resumed and continuing to set the lock to 1 and entering the critical section. Now it has two tasks in one critical section, the lock variable method fails.

The key point is that the testing and setting lock operation has to become one. It’s no way to achieve this by software but hardware. Most modern CPUs has this test−and−set−lock (TSL) instruction build−in to support multitasking OS. This instruction can test a variable and set this variable to 1 if it is 0 with one instruction execution cycle. This is also called as **atomic operation**[8]. Now consider the situation again with TSL instruction. Task A wants to enter critical section, it checks the lock and sets to 1 by TSL instruction. Since CPU cannot be interrupted while an instruction is not compete, it can guarantee that no two tasks can modify the same lock at the same time.

2.4 Semaphores

The lock variable is also referred as busy−waiting: while task requires resource, it continuously tests the lock and waits until the lock is available. Lock variable is only suitable for very short critical section and concurrent scheduling system. If not, the mechanism would fail and dead lock occurs. Besides, it wastes CPU time. Consider this situation: two tasks t_H and t_L which the former has higher priority than later and the scheduling policy always chooses higher priority task to run if it’s ready, just as most real−time system would do. At first, t_L is running and entering the critical section. For a while, t_H is ready to run and the scheduling system decides to switch to t_H and stop t_L. Now t_H tries to enter the critical section hold by t_L. The trouble is t_H cannot get it and t_L will not be resumed to finish its critical section because of its priority. Two tasks are blocked and cannot continue. This situation is sometimes referred to as the **dead lock**[8,9].

In order to resolve the defect, E. W. Dijkstra (1965) suggested using an integer variable to count the number of pending tasks. In his proposal, a new variable type, called **semaphore**[8,9], was introduced. A semaphore could have the value 0 indicating that no pending task is allowed, or some positive value if one ore more pending tasks are acceptable.

There are two operations: P and V to operate semaphore. The P operation decreases semaphore value and checks if the value is greater than or equal to 0. If true, continues to get into critical section. If the value is less than 0, task should is put to sleep. The decrease and test operation must be done by atomic operation to ensure not causing race condition in semaphore operation. Atomic operation is essential to solving synchronization problem.

The V operation increases the value and check if the value is less than or equal to 0. If true, wakes up next pending task, otherwise, continues to leave critical section. As a result, tasks that are pending resource or trying to go into critical section, can be handled one by one, or by some particular sequence.

Now, re−considering the previous problem. Let the semaphore denoted as S. t_L issues P(S) first and goes into the critical section. For a while, t_H wants to go into critical section also, it issues P(S). Now since t_L owns S, t_H will be put into sleep until t_L finishes its critical section and issues V(S) from where t_H will be woke up and continue.

Semaphore with Basic Priority Inheritance Protocol[5]

Lets extend the problem more general. There is another task t_M which the priorities are $t_H > t_M > t_L$. After first t_L requests S and blocked, t_M is ready to run. Now t_L is suspended to wait the completion of t_M, so is t_H. Under this situation, t_M is nothing to do with t_H

but it blocks the operation of t_H. This can be referred as **priority inversion**.

The reason to cause this is that t_L is running at its own priority while t_H is waiting for the same critical section. The basic idea of priority inheritance protocols is that when a task blocks higher priority tasks, it executes its critical section at the highest priority level of all the blocked tasks. After exiting its critical section, this task has to return to its original priority. For example, while t_H pending for S which is owned by t_L, t_L will gain the priority from t_H. When t_M is ready, it will not preempt t_L since the priority is lower than t_H. Now, t_L can safely execute its critical section as fast as possible and when t_L finishes, it resumes t_H and goes back to its original priority. t_M cannot affect t_H anymore.

On top of PIP, Priority Ceiling Protocols(PCP) and Semaphore Control Protocols(SCP) can be developed. Further information can be found in reference[5].

Chapter Three Interprocess Communication

In multitasking system, task is not isolated from others, especially in real−time control system. On task may needs other tasks' information to decide what's the next step. This requirement forms Inter−Process Communication(IPC).

In fact, task synchronization is also part of IPC. But it has more relationship with task scheduling policy. In this chapter, the focus will be data exchange between tasks. Three models will be introduced: shared memory[8,9], message pipe[8,9] and mail box[12].

3.1 Shared Memory

A shared memory is a physical memory region that can be shared by multiple tasks. Task can attach this memory region as its own local memory and access this region without any other system calls. One task writes data into shared memory, the changes will be visible immediately to all tasks sharing this memory location. As the result, shared memory provide the fastest way to share data among tasks.

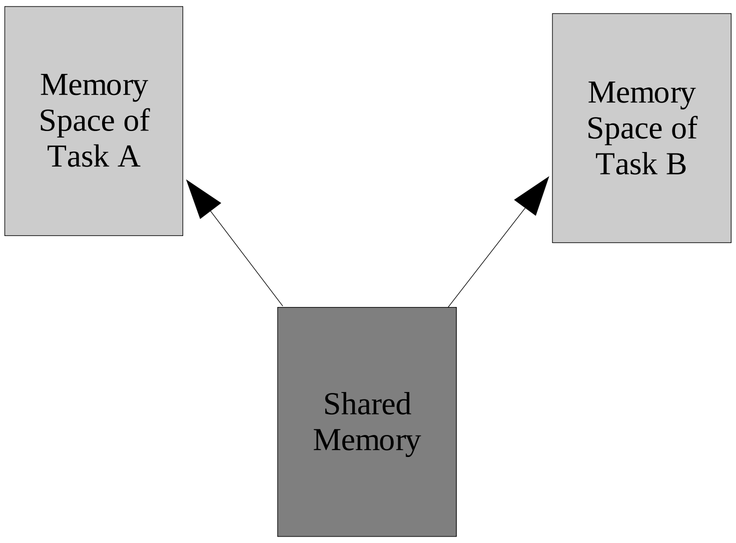


Figure 3−1 Shared memory

Shared memory provides a very fast and versatile mechanism that is suitable for large amount of data exchange without using any function call. But shared memory is lack of synchronization among tasks. When a task tries to read from a shared memory region, it has no idea that is it been changed or not. Or two tasks try to write data into the same location in the shared memory, kernel usually will not serialize the operations and result is unpredictable. Tasks sharing a shared memory need to provide their own synchronization facility, such as busy−wait or semaphore. Somehow, these facilities will involve one or more kernel calls and will introduce an overhead and slow down the performance of shared memory.

3.2 Message Pipe

On the conceptual view, a pipe is a unidirectional, first−in first−out, unstructured data stream of fixed maximum size. Writers add data to the end of a pipe and readers remove data from the head of the pipe. If there is no data can be read in the pipe, readers will be blocked until data is available and writers will be blocked, too if the pipe is fulfilled.

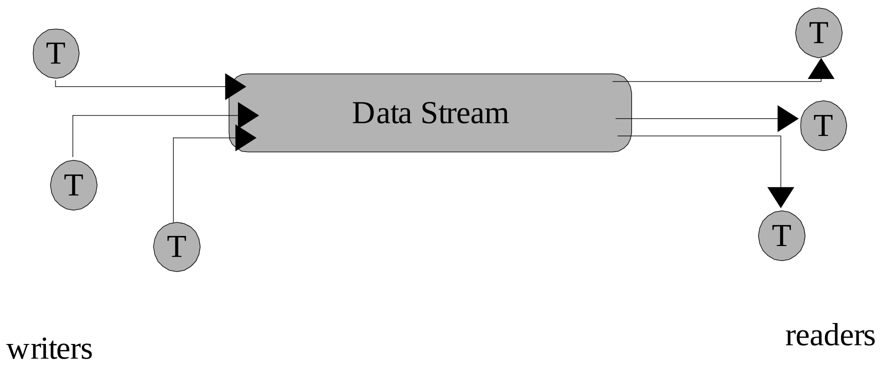


Figure 3−2 Data flow through a pipe

From the IPC perspective, pipes can provide an efficient way of transferring data from task to another. They still have some limitations:

- Since reading data will remove it from the pipe, therefore, a pipe cannot be used to broadcast data to multiple tasks.
- Data in pipe is treated as a byte−stream and has no knowledge of message of boundaries. If a writer writes several items of data and each has different

size, the reader may not read and re−construct all data correctly.

- If there are many readers on a pipe, the writer cannot direct data to a specific one. Likewise, readers cannot determine which one wrote the data into pipe if there are multiple writers.

In implementation, pipes have two types: anonymous pipes and named pipes. Anonymous pipes are used in parent task and child task and will have no record on the file system. Therefore, anonymous pipes can be used to transfer secure data. Named pipes will have a file name on the file system. As it’s a file, it can be access by unrelated tasks and are less secure than anonymous pipes.

3.3 Mail Box

Since pipes has no idea that who are the readers and who are the writers, sometimes it will be a problem if the received messages need acknowledgment. Mail box is attached to task. When a task wants send message to mail box, it needs to know who is receiver and when it reads from its own mail box, it also can know who sends the message.

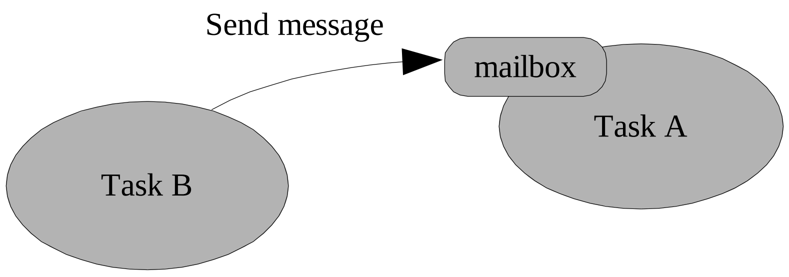


Figure 3−3 Send message to mail box

Chapter Four Device Driver

The I/O subsystem handles the movement of data between memory and peripheral devices. Tasks and kernel access device through *device drivers*. Device driver is an interface between all other parts of system and hide the hardware from them. For a real−time control system, device driver is also an important part just as real−time scheduling policy.

4.1 Goals and Benefits

There are 4 goals to achieve in designing device driver software:

1. Device independence:
   This is the key concept of device driver software. It should be possible to write programs that can be used with different types of devices without having to modify programs for each device.
2. Error handling:
   Devices may fail, RS−232 connection may be broken. Device driver has to detect these abnormal situations and try to resolve it as possible. It’s also the device driver’s responsibility to notify user task when serious problem happens and cannot be overcame by itself.
3. Synchronous (blocking) versus asynchronous (interrupt−driven) I/O:
   Most I/O devices are asynchronous. CPU starts the transfer and go off to do other computation until device interrupts. But, on the other hand, it’s easier to write program if the I/O operations are synchronous. Device driver has to take care of both kinds of operations.
4. Sharable and dedicated devices:
   Each device can be owned and operated by one task at the very time. Sharable devices have to deal with multiple requests at the same time without ruining user’s data. Dedicated devices should create its own critical section and let the tasks to access it one by one.

Device driver contains a set of data structures and well−defined interfaces. By these well−defined interfaces, a device driver can be separated from kernel and make the goals mentioned above possible. The benefit of this device interfaces are:

- Isolate device−specific code in a separate module.
- It’s easy to add a new device.
- Other programmers can add devices without kernel code.
- The kernel and tasks can has a clear view of all devices and accesses devices through the same interface.

Device driver accepts command from kernel or tasks, it also receives messages from hardware device which includes operation complete, status, and error notification. Device can generate an interrupt and kernel switches the execution into corresponding interrupt handler which is also part of device driver. The role of device driver is shown in figure 4−1.

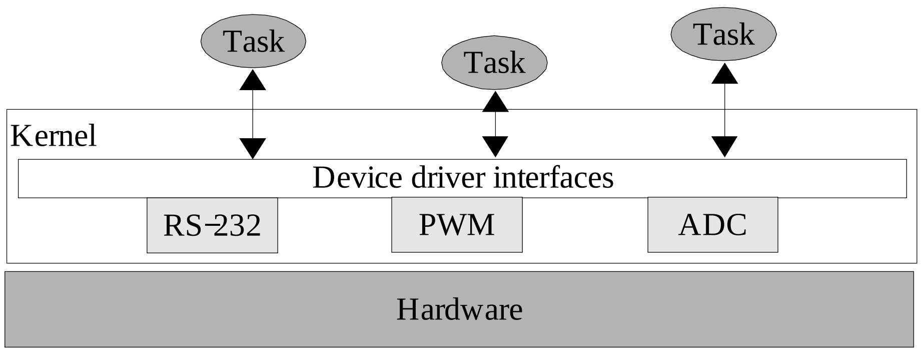


Figure 4−1 Role of the device drivers

Basically, devices can be roughly divided into block devices and character devices. In this thesis and implementation in next part, only character devices will be discussed since most micro−processors and embedded systems handle character type of devices.

4.2 Device Driver Framework

In order to hide device−dependent part of all I/O operations, it needs a high−level, procedural view of devices. In this situation, a device is a black box that supports a standard set of operations. Each device implements these operation differently, but I/O subsystem is not concerned with that. In object−oriented terms, the device interface forms an abstract class, each device driver inherits it and has different implementation.

The kernel invokes the device driver in several ways:

- Configuration — The kernel calls the driver at boot time to check for and initialize the device.
- I/O — The I/O subsystem calls the driver to read or write data.
- Control — The user may make control requests such as opening or closing the device, or rewinding a tape drive.
- Interrupts — The device generates interrupts upon I/O completion, or other change in its status.

To handles these requests, kernel has to define a driver interface. Referred from UNIX® system, a *device switch*[13] is used to achieve this goal. The device switch is a data structure that defines the entry points each device must support. There are two types of switches in UNIX® — `struct bdevsw` for block devices and `struct cdevsw` for character devices. Take character device switch as an example, typically the data structure is defined as follow:

```
struct cdevsw {
      int (*d_open)();
      int (*d_close)();
      int (*d_read)();
      int (*d_write)();
      int (*d_ioctl)();
      int (*d_mmap)();
      int (*d_seqmap)();
      int (*d_xpoll)();
      int (*d_xhalt)();
      ...
}cdevsw[];
```

The switch defines the abstract interface that each driver provides specific implementations of these functions. Whenever the kernel wants to perform an action, it locates the driver in the switch table and invokes the appropriate function of the driver. For instance, to read data from device, the kernel will invoke the `d_read()` function of the device. Here is the detail description of each function:

`d_open()`	Called each time the device is opened, and may bring device on−line or initialize data structures. Devices that require exclusive access may set a flag when opened and clear it when closed. If the flag is already set, `d_open()` may block or fail.
`d_close()`	Called when the last reference to this device is released, that is, when no process has this open. May shutdown device or take it off−line.
`d_read()`	Reads data from a character device.
`d_write()`	Writes data to a character device.
`d_ioctl()`	Generic entry point for control operations to a character device. Each driver may define a set of commands invoked through its *ioctl* interface. The arguments to this function include `cmd`, an integer that specifies which command to execute, and `arg`, a pointer to command−specific set of arguments. This is a highly versatile entry point that supports arbitrary operations on the device.
`d_segmap()`	Maps the device memory into the process address space. Used by memory−mapped character devices to set up the mapping in response to the *mmap* system call.
`d_mmap()`	Not used if the `d_segmap()` routine is supplied. If `d_segmap()` is `NULL`, the *mmap* system call on a character device calls `spec_segmap()`, which in turn calls `d_mmap()`. Checks if specified offset in device is valid and returns the corresponding virtual address.
`d_xpoll()`	Polls the device to check if an event of interest has occurred. Can be used to check if a device is ready for reading or writing without blocking, if an error condition has occurred, and so on.

`d_xhalt()` Shuts down the devices controlled by this driver. Called during system shutdown or when unloading a driver from the kernel.

The driver entry point doesn't contains interrupt handling routine and initialization. they will be specified by another configuration file which is used to build the kernel.

4.3 Blocking and Non−Blocking I/O

While accessing a device, the ordinary sequence is:

1. Task calls `d_read()/d_write()`.
2. Device driver transfers data between device and task's data space.
3. Data transfer is done, return to task.
4. Back from device driver, task continues.

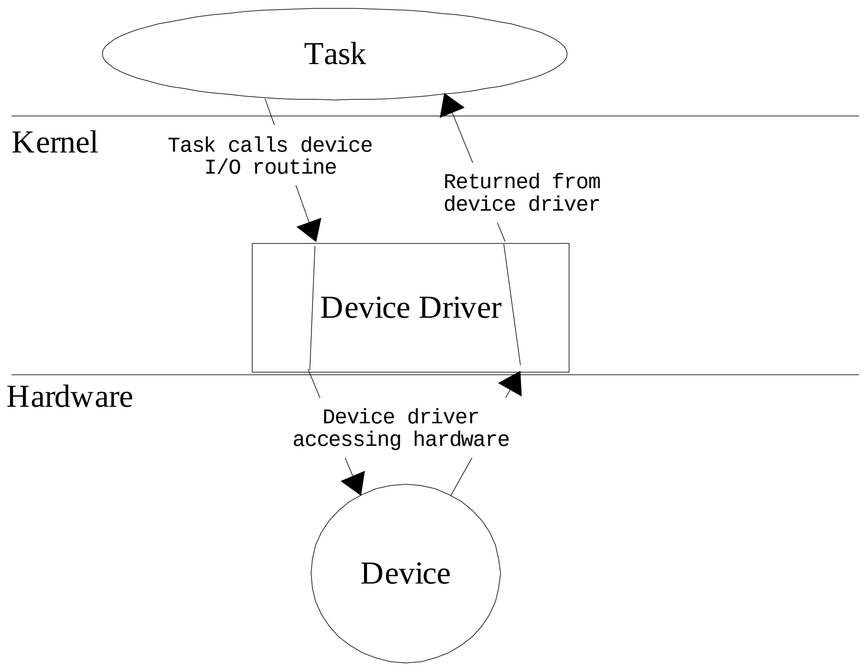

Task
Kernel
Task calls device I/O routine
Returned from device driver
Device Driver
Hardware
Device driver accessing hardware
Device

Figure 4−2 Blocking I/O sequence

In the duration of transferring data from device to task's data space, task is stopped to wait for the completion of device driver. This is the most strait forward procedure and suitable for fast peripherals. Since most CPU execution speed is much higher than devices, this access procedure is not efficient and wasting CPU computation power.

Non−blocking I/O, as its name means, will not block task while I/O operation is proceeding. Each time a task calls an I/O routine, this routine will returns immediately and leaves the I/O operation in uncompleted state. As a result, task can continue to do computation and will not be blocked by the I/O operation. While I/O operation completes, device has to generate an interrupt into device driver's ISR and confirms to the task that the operation has been done. This is shown in Figure 4−3.

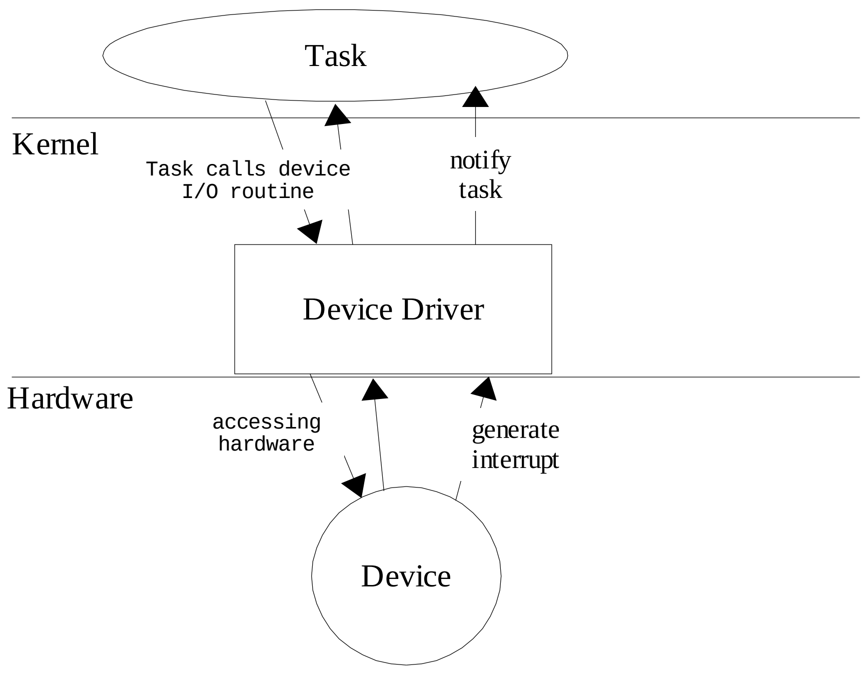


Figure 4−3 Non−blocking I/O sequence

The most benefit of non−blocking I/O is the improvement of CPU utilization. For instance, a task wants to send the computation result to the remote device through low speed RS−232, it issues a write command on RS−232 device driver, then continues its own computation. Both jobs can be done simultaneously. But on the other side, task has to make sure not to change the data that is currently transferring until the device complete the operation. Otherwise, data will be corrupted.

Part II Implementation on TI 320F24x DSP : Taunix

Good programmers know
what to write. Great ones know
what to rewrite (and reuse).

Taunix is a small real−time multitasking kernel implemented on TI 320F24x DSP. It uses modern operating system's concept to modularize each component and form an embedded system development platform. It hides hardware information from user tasks and let the programmer focus on his/her control procedures and algorithms. Taunix is a portable kernel, too. Most codes of Taunix are written in C language to ensure that it can be easily ported to another platform. The footprint of basic multitasking kernel is 714 words text with 233 words data on TI's 320F243 DSP. This can satisfy most applications of embedded systems.

The architecture of Taunix is divided into 3 parts: task management, device driver and inter−process communication. Each part will be discussed in the following sections.

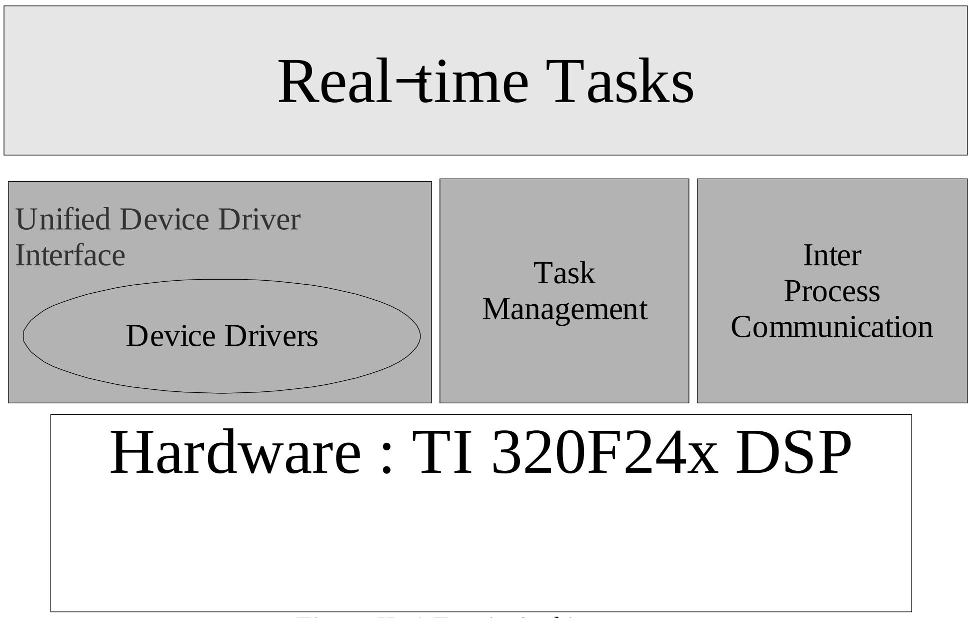

Real−time Tasks
Unified Device Driver Interface
Device Drivers
Task Management
Inter Process Communication
Hardware : TI 320F24x DSP

Figure II−1 Taunix Architecture

Chapter Five Task Management

The basic scheduling unit in Taunix is task. Each task is represented by a data structure called TCB, Task Control Block. TCB keeps track task's starting address, stack address, task state and necessary context. Scheduler will use this information to manage all ready tasks.

5.1 Task Control Block

The structure of TCB has a strong relation with task switching or context switching mechanism. Because of portability, Taunix chooses setjmp/longjmp as the basis of task switching and Taunix TCB has to be compatible with jumping buffer. It is declared in setjmp.h as:

```
typedef int jmp_buf[5];
```

This declaration itself doesn't provide enough information. By tracking into TI's C library, the five integers represent the following structure:

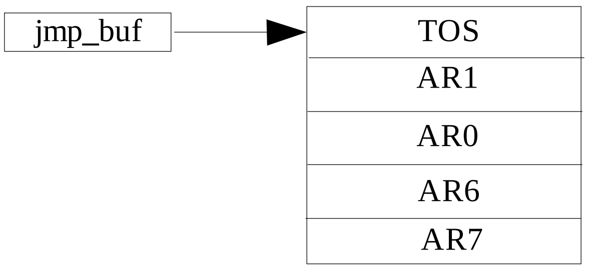


Figure 5−1 Jumping Buffer

In TI's C language definition, TOS is used as function return address, AR1 as stack pointer, AR0 as stack frame and AR6, AR7 are for register variables. Take these as the basis, Taunix TCB is defined as follows:

```
typedef struct TCB{
     void *(ret_addr)();
     void *stack_pointer;
     void *stack_frame;
     int AR6;
     int AR7;
     char state;
     char *caption; /*reserved for expansion*/
};
```

There is an extra field, state, to store task's running state. The task switching procedure will be very simple:

```
void switch_task(TCB *from, TCB *to)
{
     if(setjmp((void*)from) == 0)
          longjmp((void *)to, 1);
}
```

By observing the jumping buffer's structure, it only saves five registers' context. Is it enough? The answer is positive for C language environment but negative for assembly. Since all variables in C represent memory locations. C compiler will store all variables before any function call to prevent data loss. Therefore, before calling `switch_task(from, to)`, all data will safely saved by compiler. Assembly program has to deal with all registers other than AR6 and AR7, which will handled by C compiler.

Should be noticed that there is no priority field in task control block. Taunix will group all tasks in an array and the array index will form the priority level. This will be explained in the next section.

5.2 Priority−based Task Scheduling

The objective of scheduler is to choose one proper task and let it be executed by CPU.

Taunix uses fixed priority scheduling policy. At any time, only the task with highest priority will be chosen. Taunix supports 32 tasks, each one has its own priority. Priority 0 is the highest and 31 is the lowest. All 32 tasks form a task table, every 8 tasks are grouped as one catalog. By using an index mapping table[10], the highest task can be computed by one rule with deterministic computation time.

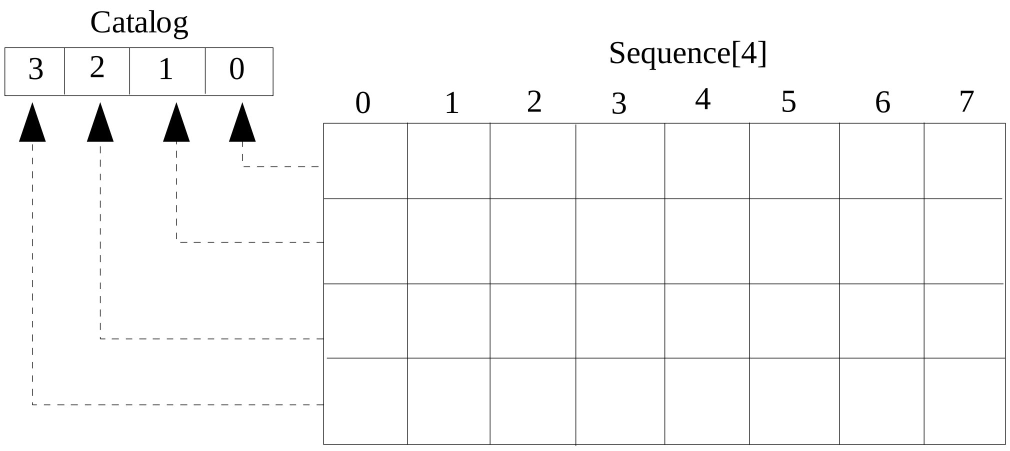


Figure 5−2 Taunix Task Table

Index mapping table:

```
const unsigned char rdy_index[16]={0,3,2,2,1,1,1,1,0,0,0,0,0,0,0,0};
```

Task choosing rule:

```
unsigned char catalog;
unsigned char sequence[4];

catalog_temp = rdy_index[catalog];
seq_temp1 = sequence[catalog_temp];
seq_temp2 = seq_temp >> 4;
highest task = (seq_temp2 ? rdy_index[seq_temp2] :
                       rdy_index[seq_temp1]+4) + catalog_temp*8;
```

The task table records each ready task by putting a mark in `catalog` and `sequence[]`. For example, task 6(with priority level 6) belongs to catalog 0 with sequence 6. Therefore, `catalog` will have a mark in bit 0 and `sequence[0]` has a mark in bit 1. In the task choosing rule, it finds out the highest catalog first, then separates the sequence into the upper part and lower part. The upper part of sequence represents the higher tasks. If it is greater than zero, use it to determine the highest task by index mapping. Otherwise, use lower part. By using this method, only 16(2^4) elements are needed in the index mapping array instead of 256(2^8) that consumes too much memory space.

Tasks in Taunix may experience four states by issuing kernel calls. They are `TS_FREE`, `TS_READY`, `TS_CURRENT` and `TS_SUSPEND`. Initially, all tasks in task table are at `TS_FREE` state. After a `task_create()` call, that task will be created and transferred into `TS_READY` state. Each time the scheduler chooses the highest task, it will change its state to `TS_CURRENT` to indicate that this task now owns CPU. By calling `suspend()`, a task will be transferred into `TS_SUSPEND` state and can be back to `TS_READY` by calling `resume()`. The task state transition diagram is shown in figure 5−3.

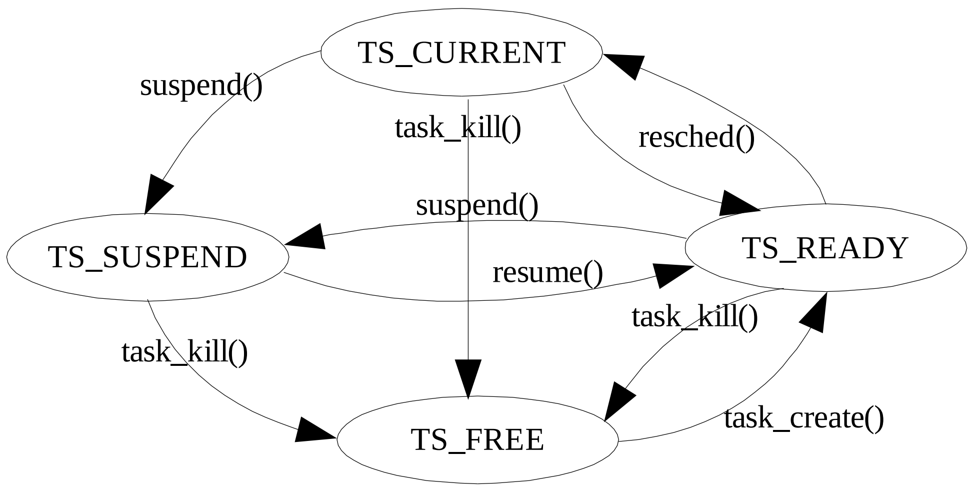


Figure 5−3 Task State Transition

Should be noticed is that this scheduling policy knows nothing about external time event but priority. Since some micro−processors are lack of real−time clock support (like TI’s 320F243), Taunix can use another add−on module to link with real−time clock or general purpose timer to provide time event service or periodic task scheduling and the scheduling policy will be priority based with time service extension. This is the main goal of Taunix: modularization.

5.3 Periodic Task Scheduling

In order to support time event, Taunix uses an add−on module, atd which means AT daemon to link with real−time clock signal. Atd uses a job table to store information of periodic jobs and a simple round−robin scheduling policy.

AT job entry:

```
typedef struct _atentry{
      unsigned scheduled;
      unsigned period;
      unsigned period_count;
      unsigned (*job)();
}ATENTRY;
extern ATENTRY    attab[MAX_PERIODICJOB];
```

While CPU receives a real−time clock signal and generates an interrupt, at daemon will scan each entry in job table and decreases the `period_count`. If `period_count` reaches 0, the corresponding job will be launched. The algorithm could be summarized as follow:

```
For each entry in job table
      If has job assigned
            Decrease period count
            If time out
                  Launch this job and reset period count
            Endif
      Endif
Done
```

At daemon itself has been designed as the form of call−back function(see Device Driver section). Therefore it can be normally called or assigned to general purpose timer device driver's call−back functionality to fit different system arrangement.

5.4 Data Structures and Member functions of Scheduling System

Task Control Block:

```
typedef struct _tcb{
      void (*ret_addr)();           /* TOS */
      void *stack_pointer;          /* AR1 */
      void *stack_frame;            /* AR0 */
```

```
        int AR6;                      /* reserved  */
        int AR7;                      /* reserved  */
        char *caption;
        char state;
  }TCB;

  #define MAX_TASK  32
  extern TCB  task_table[MAX_TASK];
```

Kernel Scheduling Flags and Member Functions:

```
  extern int  current_task;

  extern unsigned int sched_flag;
  extern unsigned int int_flag;

  void task_table_init();
  int  task_create(void (*start_add)(),void *stack,int prio,
                                                  char *caption);
  int  task_kill(int prio);
  void texit();
  int  suspend(int prio);
  int  resume(int prio);
  void resched();
  void task_start();
```

Periodic Scheduling Support

```
  typedef struct _atentry{
        unsigned scheduled;
        unsigned period;
        unsigned period_count;
        unsigned (*job)();
  }ATENTRY;

  #define MAX_PERIODICJOB 10
  extern ATENTRY    attab[MAX_PERIODICJOB];

  void  atd_init();
  int   at(unsigned (*job)(),unsigned peiord);

  #define DEFINE_ATTAB_BEGIN    void attab_init(){
  #define ATJOB(job_no,job_func,job_period)
        attab[job_no].scheduled=1;\a
        ttab[job_no].period=\
              attab[job_no].period_count=job_period;\
              attab[job_no].job=job_func;
  #define DEFINE_ATTAB_END      }
  #define atkill(job_no)        attab[jobno].sheduled=0;
```

Chapter Six Interrupt Handling

Interrupt handling is quite hardware dependent subject. To decrease interrupt latency, Taunix does not use complex interrupt handling architecture but depends on hardware’s definition. First, let’s see how TI 320F243’s interrupt works.

6.1 TI 320F243 Interrupt Controller

TI 320F243 supports one non−maskable interrupt(NMI) and six maskable prioritized interrupts requests. Since 243 has many peripherals and each can generate more than one interrupt requests, 6 interrupt level are not enough. Therefore, all interrupts are grouped and formed a hierarchical architecture by Peripheral Interrupt Expansion(PIE) controller.

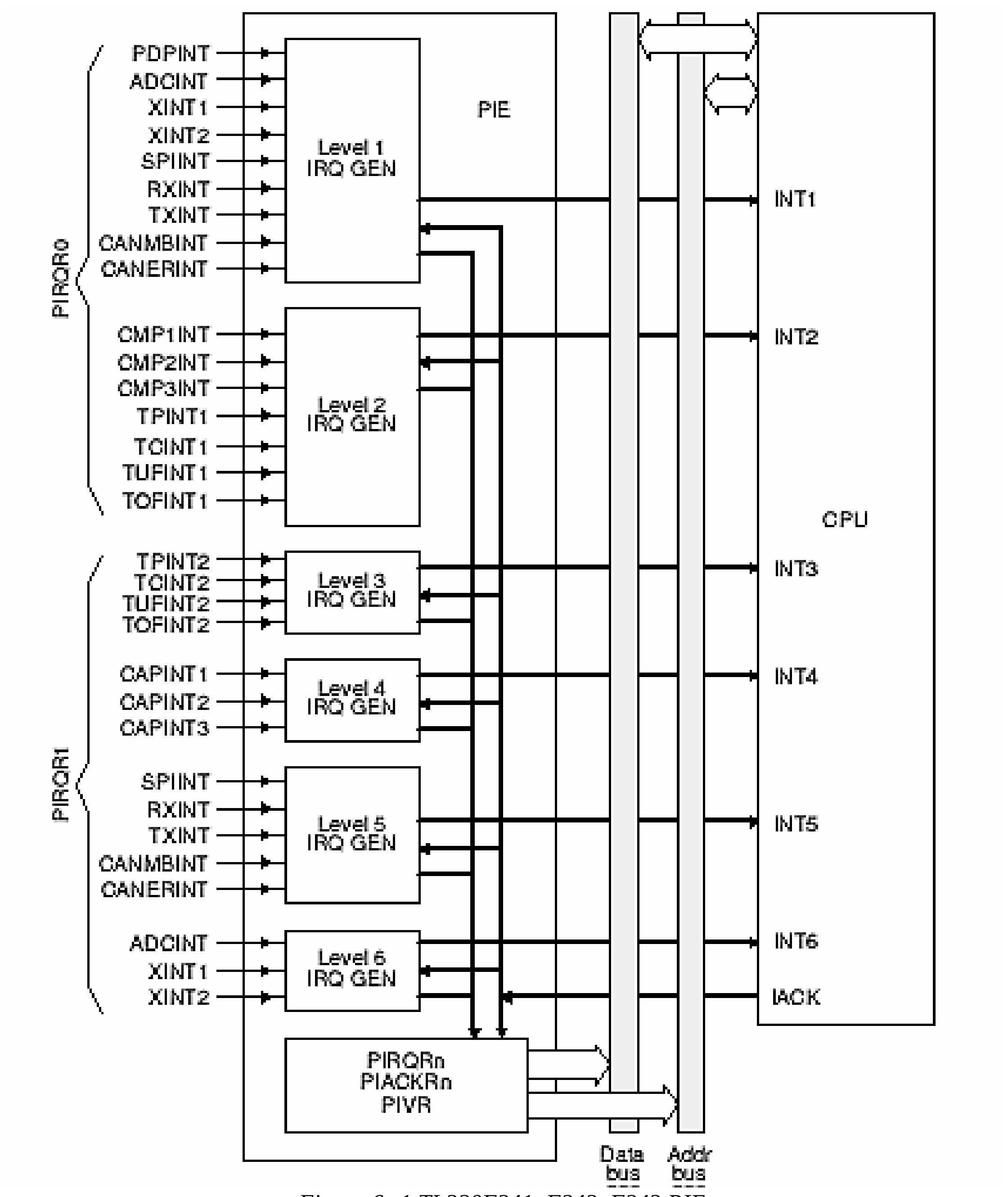
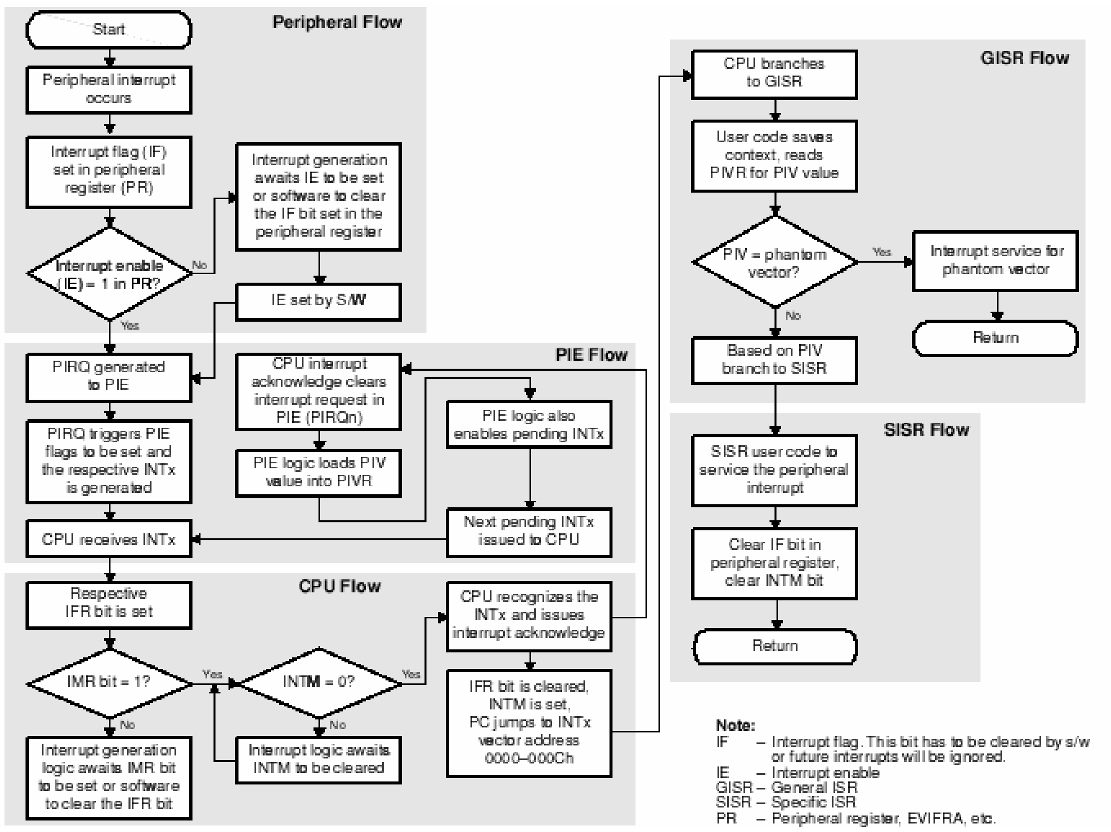


Figure 6−1 TI 320F241, F242, F243 PIE

In the lower level, peripheral interrupt request(PIRQ) will be ORed together by PIE controller and send interrupt request(INTn) to the CPU core. At this time, CPU has no idea that which peripheral requests interrupt. PIE controller will load interrupt vector into its peripheral interrupt vector register(PIVR). It can be read by CPU and used to generate a vector to branch to the interrupt service routine(ISR) which corresponds to the event being acknowledged.



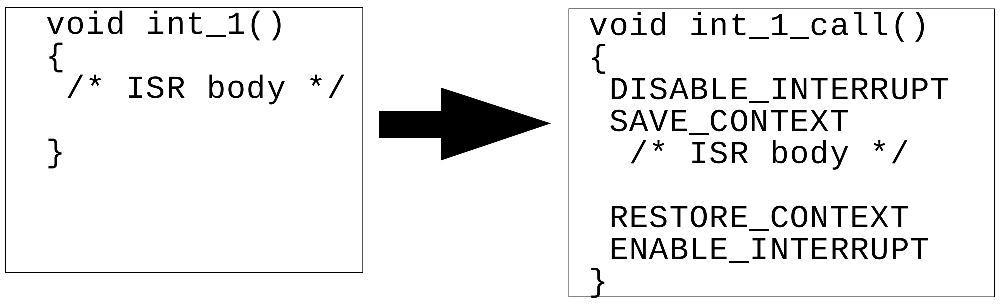
Peripheral Flow
Start
Peripheral interrupt occurs
Interrupt flag (IF) set in peripheral register (PR)
Interrupt generation awaits IE to be set or software to clear the IF bit set in the peripheral register
Interrupt enable (IE) = 1 in PR?
No
Yes
IE set by S/W
PIE Flow
PIRQ generated to PIE
CPU interrupt acknowledge clears interrupt request in PIE (PIRQn)
PIE logic also enables pending INTx
PIRQ triggers PIE flags to be set and the respective INTx is generated
PIE logic loads PIV value into PIVR
Next pending INTx issued to CPU
CPU receives INTx
CPU Flow
Respective IFR bit is set
CPU recognizes the INTx and issues interrupt acknowledge
IMR bit = 1?
Yes
INTM = 0?
Yes
No
No
IFR bit is cleared, INTM is set, PC jumps to INTx vector address 0000–000Ch
Interrupt generation logic awaits IMR bit to be set or software to clear the IFR bit
Interrupt logic awaits INTM to be cleared
GISR Flow
CPU branches to GISR
User code saves context, reads PIVR for PIV value
PIV = phantom vector?
Yes
Interrupt service for phantom vector
Return
No
Based on PIV branch to SISR
SISR Flow
SISR user code to service the peripheral interrupt
Clear IF bit in peripheral register, clear INTM bit
Return
Note:
IF – Interrupt flag. This bit has to be cleared by s/w or future interrupts will be ignored.
IE – Interrupt enable
GISR – General ISR
SISR – Specific ISR
PR – Peripheral register, EVIFRA, etc.

Figure 6−2 Interrupt Handling Flow

In effect, there are two vector tables: The CPU's vector table which represents INT1 to INT6 and named as general interrupt service routine(GISR); and the peripheral vector table that is used to get event specific interrupt service routine(SISR) to responses to the event that generates PIRQ.

6.2 Taunix ISR Implementation

For a multitasking operating system, interrupts are like an asynchronous event to the scheduling system and always interrupt current job. Interrupts have higher priorities than tasks. As interrupts will preempt current running tasks, the task switch mechanism in Taunix cannot handle it because the preempted task is not giving up CPU voluntarily. ISR has to save the whole CPU context to prevent data corruption. Fortunately, TI's C compiler supports 10 extension function keywords, c_int0 to c_int9, for ISR. The differences between common function call and ISR extension is that ISR extension function will save CPU context before entering function body and restore it before leaving.



```
void int_1()
{
 /* ISR body */

}
```

```
void int_1_call()
{
 DISABLE_INTERRUPT
 SAVE_CONTEXT
  /* ISR body */

 RESTORE_CONTEXT
 ENABLE_INTERRUPT
}
```

Figure 6−3 Interrupt Keyword Function

By using ISR keyword, the GISR vector table can be easily established. As mentioned in 243's programmer's guide, the interrupt vector table starts at address 0000h by inserting a sequence of branch instructions. For each branch instruction, the destination address should be the starting address of GISR. The whole table will look like:

```
      .global _c_int0
      .global _c_int1
      .global _c_int2
      .global _c_int3
      .global _c_int4
      .global _c_int5
      .global _c_int6
      .sect ".vectors"
INT_VETTAB:
      B     _c_int0
      B     _c_int1
      B     _c_int2
      B     _c_int3
      B     _c_int4
      B     _c_int5
      B     _c_int6
      .end
```

For each GISR, it will read peripheral interrupt vector register(PIVR) and decide where to branch to. The structure of GISR is shown in the following program segment.

```
c_int6()
{
      ISR_ENTRY
      switch(MMREGS[PIVR]){
       case 0x0004:
            /* ADC SIVR */
            break;
       case 0x0001:
            /* XINT1 SIVR */
            break;
       case 0x0011:
            /* XINT2 SIVR */
            break;
       case 0x001F:
            /* XINT3 SIVR */
            break;
      }
      ISR_EXIT
      if(sched_flag==SCHED_DELAYED){
            enable_sched;
            disable_int;
      }
}
```

In this example, `ISR_ENTRY` is used to notify the Taunix scheduler that CPU is in ISR. Then `MMREGS[PIVR]` will read PIVR register to get the peripheral’s vector number. According to PIVR, GISR will branch to corresponding SIVR codes. Before leaving GISR, notify the scheduler again that CPU is leaving ISR and re−schedule once to make sure the highest priority task can hold the CPU.

This is the general form of an ISR. Sometimes, in order to reduce SISR’s interrupt latency, it should avoid to use conditional branch instructions, like if−else, switch−case, and etc. But the drawback is this ISR will be dedicated to one peripheral and hard to share with others.

6.3 Interaction Between ISR and Scheduler

Since the occurrence of interrupt can not be controlled by scheduler, ISR needs extra code to prevent corrupting scheduling data structure. First, ISR has to push kernel into critical section and disables all interrupts before it leaves. This has been done by interrupt keyword function. By tracking into the assembly code generated by TI's C compiler, it shows that it will mask all interrupts and save all CPU context while entering ISR. It is the very definition of critical section for Taunix.

Second, ISR should notify the scheduler that CPU is in ISR, not normal tasks. In the example of `c_int6()`, `ISR_ENTRY/ISR_EXIT` will set/clear an interrupt flag and notify the scheduler that CPU is in ISR now or leaving. At this moment, if any scheduling kernel function is called, like `suspend()` or `resume()`, the scheduler will be disabled and set the scheduling flag as `SCHED_DELAYED`. Before leaving ISR, it has to re−enable the scheduling system and re−schedule once to make sure the highest priority task can get the computation time and this is done by the last `if` statement block.

Chapter Seven Device Drivers

Taunix driver refers UNIX® device switch to form its own driver model. By using device switch, all drivers' interfaces can be unified. Besides, blocking and non−blocking access are also supported by call−back function to let the program more responsive.

7.1 Device Switch

Taunix device switch is a structure that records all access functions of each driver. The declaration of Taunix device switch is:

```
typedef struct _cdevsw{
        char *caption;
        int   owner_task;
        int   (*d_open)(int flags,int mode);
        int   (*d_close)();
        int   (*d_read)(void *buf,unsigned int size);
        int   (*d_write)(const void *buf,unsigned int size);
        int   (*d_ioctl)(unsigned request,void *argp);
        void  (*read_callback)(void *param);
        void  (*write_callback)(void *param);
}CDEVSW;
```

- Caption

  The name of this driver. Currently, reserved for future releases.
- Owner_task

  The task id that is currently using this driver.

  Normally, a driver will control one peripheral or resource. If the access cannot accept more than one task's requests, device driver can use this field to do basic restriction instead of using semaphore.
- D_open()

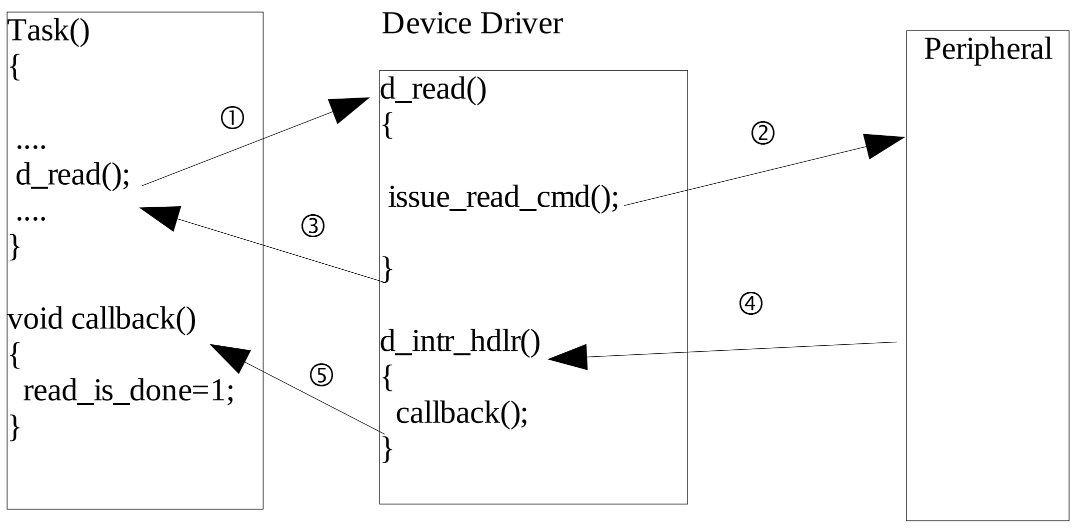
  Open this driver and prepare to be used.

  By definition of POSIX, flag is the parameter that will be passed to driver(or file) and mode is the attribute mode(readable, writable or executable). This is general for all file−styled UNIX file system object including ordinary files and device nodes. For Taunix, there is no file actually. These two parameters are for passing device specific arguments. The format is free for implementer.

- D_close()
  Close a device.
- D_read()
  Read data from device.
- D_write()
  Write data into device.
- D_ioctl()
  I/O control.
  This interface is for further control of device that cannot be performed in d_open(). D_open() is suitable for initialization and d_ioctl() is for runtime operation. D_ioctl() needs two parameters: request and relative argument pointer.
- Read_callback() / write_callback()
  The callback functions for non−blocking read and write operation. See next section.

7.2 Non−Blocking I/O Support

Non−Blocking I/O is supported by callback mechanism. On 320F24x, this feature has to combined with interrupt handling. For each read or write operation, driver issues the command and returns to the calling function immediately. The driver interface has to return an error code to indicate that the required operation is not complete. While the operation is done, peripheral will generate an interrupt into CPU core. ISR must recognize that this operation is under pending by some task. Then call the callback function to do the post processing. The whole operation flow can be shown as Figure 10−1.



① *Tasks call d_read() of device driver.*
② *Device driver issues the read command to the peripheral*
③ *Device driver returns to task immediately*
④ *Peripheral completes the read operation and generates an interrupt*
⑤ *ISR calls pre-defined task call-back function*

Figure 7−1 Non−blocking I/O operation flow

7.3 Supported Devices

Currently Taunix supports the following devices:

- Analog to digital convertor
- Capture unit
- General purpose timer
- Compare unit(generate space vector PWM signal)
- Serial communication interface
- Real−time timer(need external interrupt source on F243)

Chapter Eight Inter−Process Communication

There are three kinds of inter−process communication(IPC) mechanisms in Taunix: semaphore, message queue and message pipe to do synchronization and message passing.

8.1 Semaphore

Semaphore is an important method to do access control of shared resources. Since there are many types of semaphores, Taunix chooses to implement counting semaphore because of its simplicity. The semaphore object is declared as follow:

```
typedef struct _sem{
        int sem_state;
        int sem_value;
        int owner;
        priList pending_list;
        priNode pending_table[MAX_PENDING];
}SEM;

extern SEM      sem_table[];
```

In order to prevent or resolve dead lock and to improve efficiency , two important features are added into Taunix semaphore:

- Limited priority−based pending list

  The pending list of each semaphore is a priority queue. All pending tasks will be queued according to its priority except the one that currently owns this protected resource. This can guarantee the higher priority task can hold the resource it needs as soon as possible.

  The quantity of pending tasks is also limited to 5 by default. This can prevent too many tasks are starving for one resource.

- Non−blocking pending

By default semaphore algorithm, pending task will be suspended to wait for its turn if the required resource is not available. It’s not a good idea to let a task be blocked too long especially the one holding the resource has lower priority which is called priority inversion. Taunix semaphore can allow non−blocking pending. That is task pending a resource owned by another one can only register itself into the pending list and then do further processing. Semaphore will return an error code to indicate that this resource is currently not available and have to wait. The pending task can poll the owner of resource to see if it holds this semaphore and has the right to access, or sleeps for several millisecond(wait for time−out) and gives up pending if the resource is still not available.

By adding above two features to counting semaphore, the operation algorithm is modified as follows:

```
semaphore count initialized to 1

Pending:
      If too many tasks are pending, return error
      Decrease count by 1
      If count is smaller than 0, then
            Insert current task into pending list
      else
            Current task can hold semaphore and
            return success
      If mode is non−blocking, then
            Return non−blocking pending code
      else
            Suspend current task
      After resumed, check current owner
      If current task owns semaphore, then
            return success
      else
            return error

Post:
      If current task owns semaphore, then
            Increase count by 1
            If none is pending, then
                  return success
            else
                  Resume next one and remove it from the
                  pending list
                  return success
      If current task doesn’t own semaphore, then
            Remove current task from pending list and
            give up pending
            return success
```

Although the add−on features can help, it still cannot prevent the situation that a task holds a resource too long or the problem caused by priority inversion. By this case, programmer should implement a server task to manage the access of resource. The server task has to accept requests from other tasks through message queue or pipe(will be discussed in the next section) and complete the operation. In the next chapter, it will show that how secsd manages the SCI device and accepts reading/writing commands from other tasks without blocking them.

8.2 Data Structure and Member Functions of Semaphore

```
#define MAX_SEM     10 /* Define maximum of semaphores */
#define MAX_PENDING 5  /* Define maximum of pending tasks */

typedef struct _sem{
      int sem_state;
      int sem_value;
      int owner;
      priList     pending_list;
      priNode     pending_table[MAX_PENDING];
}SEM;

extern SEM  sem_table[];

void sem_table_init();
int  semget();
int  sem_release(int sem_handle);
int  pend(int sem_handle,int sem_mode);
int  post(int sem_handle);
int  sem_pendings(int sem_handle);
int  sem_owner(int sem_handle);
```

8.3 Message Pipe

Message pipe can provide stream data exchange. Unlike pure global data variables, message pipe will help to synchronize both end and ensure the receiver can fully and correctly receive data. Internally, there is a circular buffer inside each message pipe. Producer writes data at the tail of the buffer and consumer reads from the head. This is shown in figure 8−1.

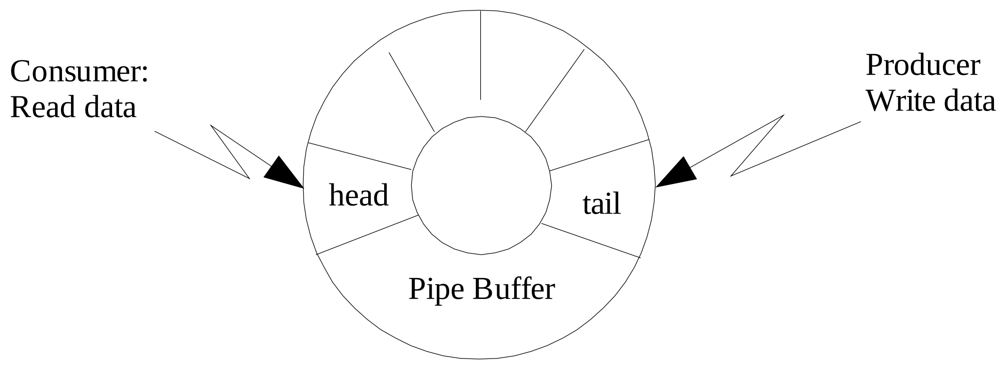


Figure 8−1 Message pipe operation

In Taunix implementation, message pipe’s buffer is an unsigned integer array with fixed size. For efficiency consideration, Taunix pipe will not block producer or consumer task. Instead, Taunix pipe will return an error code when read or write operation fails. Overwriting and Non−overwriting mode are supported in Taunix pipe, too. The pipe object is:

```
typedef struct _pipe{
     unsigned int *buffer;
     int  size;
     int  head;
     int  tail;
     int  count;
     int  ow_mode;
}PIPE;
extern PIPE    pipe_tab[];
```

8.4 Message Queue

Message queue is task based and implemented on top of message pipe. Each task can only have one message queue(but multiple pipes) and the task ID is also the ID of its message queue. Others are the same as pipe in Taunix implementation.

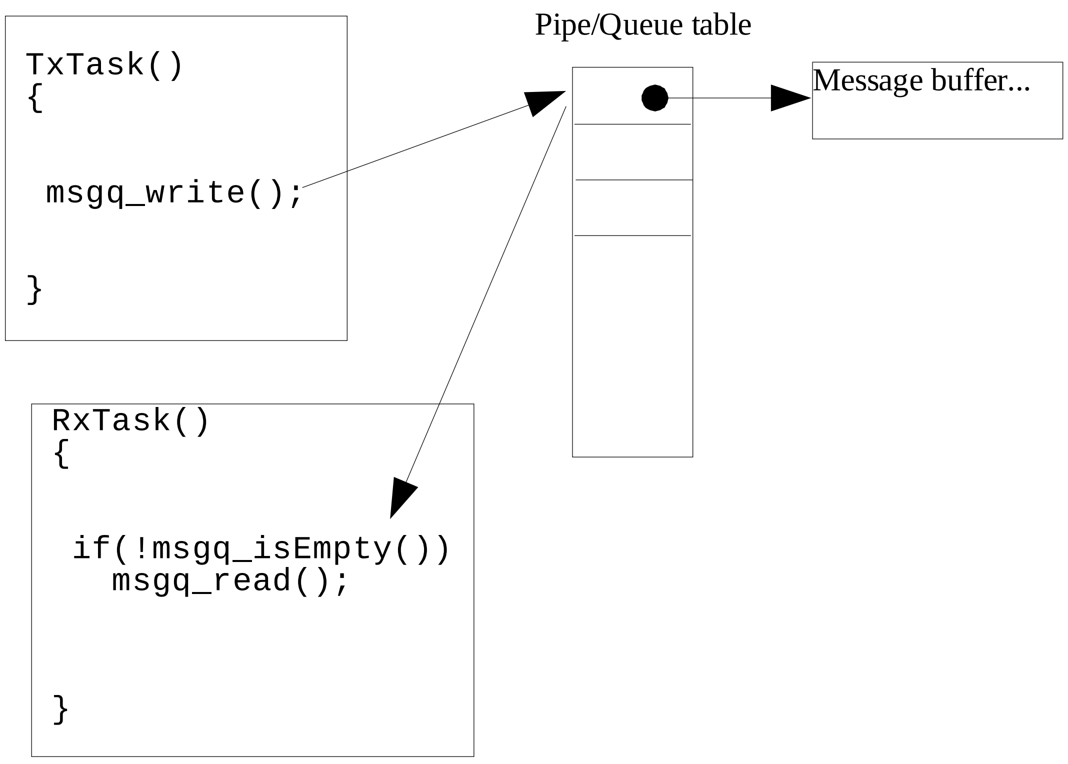


Figure 8−2 Message queue operation

8.5 Data Structures and Member Functions of Message Pipe and Queue

```
typedef struct _pipe{
     unsigned int *buffer;
     int  size;
     int  head;
     int  tail;
     int  count;
     int  ow_mode;
}PIPE;

extern PIPE    pipe_tab[MAX_TASK+MAX_PIPE];
void pipe_init();
int pipe(unsigned int *pbuf,int size,int mode);
int pipe_close(int phandle);
int pipe_read(int phandle,unsigned int *buf,unsigned int
                                                      size);
int pipe_write(int phandle,unsigned int *buf,unsigned int
                                                      size);
int pipe_isEmpty(int phandle);
int pipe_isFull(int phandle);
int msgq(unsigned int *pbuf,int size);
int msgq_close();
int msgq_read(unsigned int *buf);
int msgq_write(int task,unsigned int msg);
int msgq_isEmpty();
int msgq_isFull();
```

Chapter Nine Communication

In Taunix, it contains a simple master−slave communication protocol module: SECS−I, which is a point−to−point serial communication protocol. This module is a server task in Taunix. Besides, it’s also a great demonstration that how a task and an interrupt cooperate together to perform an time−consuming operation outside of device interrupt and forms a virtual device serves for all other tasks.

9.1 SECS−I protocol

The whole SECS−I protocol has four states in operation: idle, line control, send and receive:

- Idle:

  Initially, SECS will wait for send or receive command.
- Line control:

  While a command is received, SECS will do the line control to make sure the transmission can be performed. If the command is "send", SECS will send an ENQ and wait for EOT to be returned. But if SECS receives ENQ also at this time, that means line contention occurs. SECS should move to receive state in slave side or continue to send in the master side.

  While the command is "receive" comes from the other side, the operation is simple. SECS returns an EOT and moves to receive state. "Receive" command has higher priority than "send".
- Send

  In send state, SECS will calculate the checksum and send the whole packet. SECS packet contains 3 parts(figure 9−1): packet length, data and checksum. After all bytes are send, SECS will expect an ACK returned. If the counterpart doesn’t return ACK, that means something wrong and SECS should re−send this packet again.
- Receive

  In receive state, SECS will get whole packet and re−calculate the checksum to make sure all data is received correctly. If everything seems good, returns ACK.

Otherwise, returns NAK and waits for a second try.

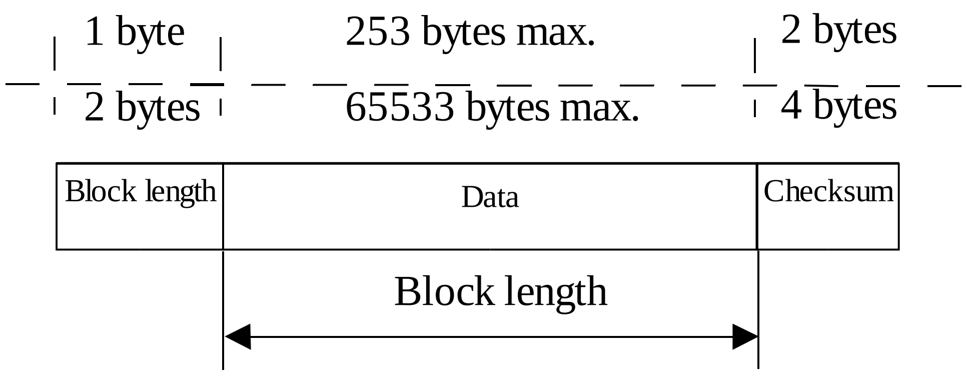


Figure 9−1 SECS packet format

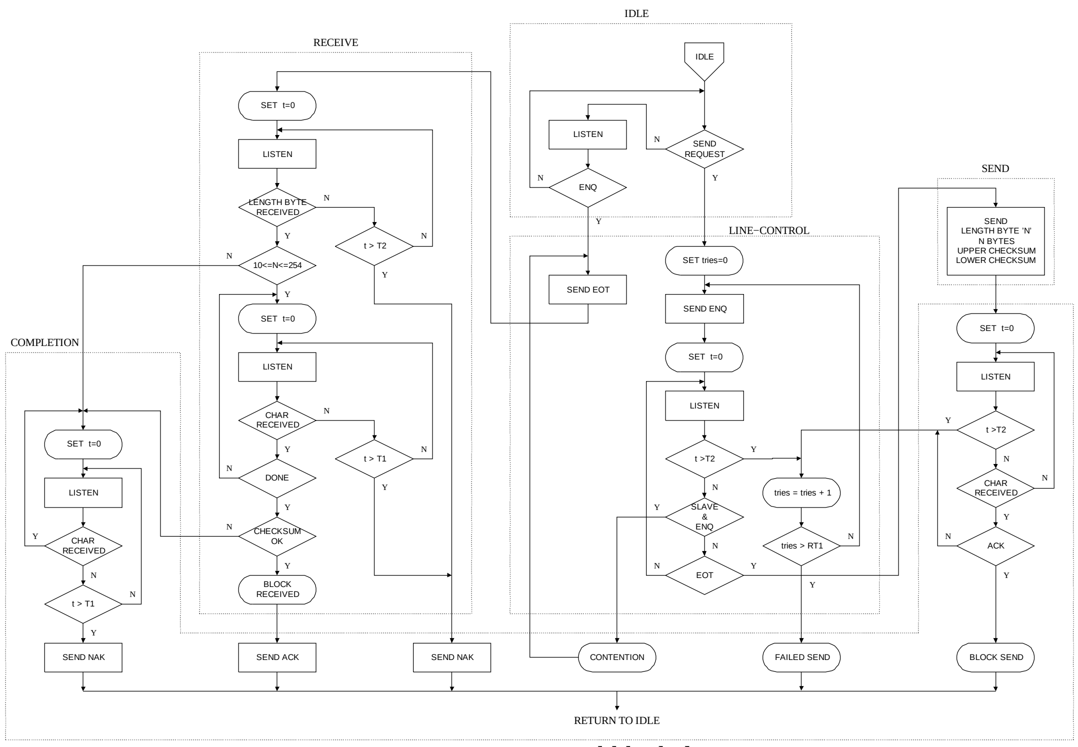


Figure 9−2 SECS protocol block diagram

8.2 SECSd implementation

The implementation can be divided into two parts: SECS server task and the virtual device. The server task: secsd handles the protocol flow, transmits and receives SECS packets. The virtual device is the bridge between server task and other tasks that require transmission. Like all other devices, SECS virtual device has a device switch to represent this device. In this device, it uses two pipe, tx_pipe and rx_pipe to communicate between tasks. The block diagram is shown in figure 9−3.

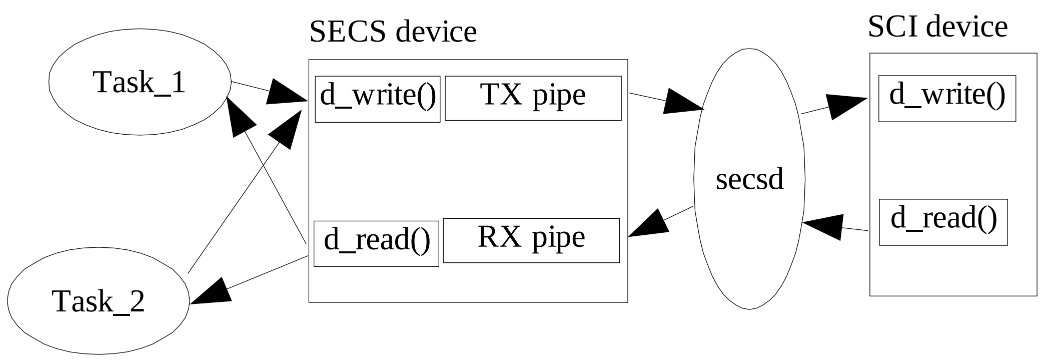


Figure 9−3 SECS virtual device

Tasks send and receive raw data to SECS virtual device through d_read()/d_write() interface. In SECS virtual device, the raw data will be packed into the SECS packet. As described in previous section, the SECS packet in Taunix is declared as:

```
typedef struct _secsblock{
        unsigned int    uiDataLength;
        unsigned int *  uipData;
        unsigned long   ulDataCHKSUM;
        int             owner;
}SECSBLOCK;
```

In d_write() interface, this packet will be send into tx_pipe, and d_read() will read from rx_pipe. In the other words, SECS virtual device doesn’t really handle the transmission activity. It is like a stub that helps other tasks to communicate with secsd. For the task side, by using uniformed device interface, it’s the same as using other devices to read or write data through SCES protocol.

The process of secsd has three stage: idle, arbitration and transmission. Initially, secsd waits for commands. It checks the tx_pipe first. If no packet needs to be send, secsd will listens to the SCI and waits for incoming packet. The second stage is arbitration. While secsd received an command, no matter it is send or receive, secsd arbitrates to check if it can perform that command at the very time. Then, secsd will enter transmission stage to send or receive SECS packet. The event flow chart is shown in figure 9−4.

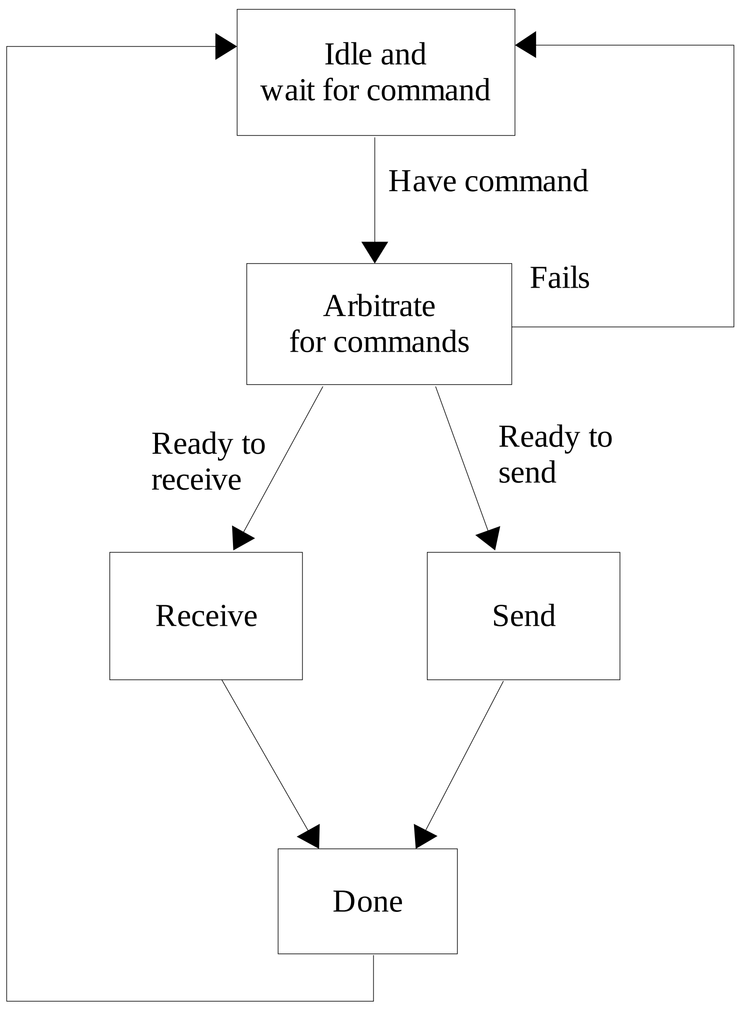


Figure 9−3 Secsd event flow chart

There are two implementation techniques should be discussed here. One is the data sending synchronization with tasks and memory management. In sending process, tasks write data into SECS virtual device. But this virtual device cannot guarantee when secsd can really send those data due to the line status might be busy. It’s unwise to let tasks to wait until transmission is complete. Basically, SECS virtual device always uses asynchronous I/O mode. In SECSBLOCK object, there is a field named owner that will record the owner task of this packet. After sending, secsd will send a message: SECS_SEND_COMPLETE to the owner task’s message queue. Packet owner task needs to open its own message queue and checks the sending complete message. If owner task doesn’t open its message queue, the write operation of message queue will fail and this notification message will not be send.

The second problem is memory management. In communication protocol, dynamic memory allocation somehow is always needed. So is SECS. But dynamic memory allocation is not preferred in real−time programming. Here a static packet pool is used. In initialization process, an RX buffer should be assigned to SECS module. The whole buffer space is divided into pages. Each page contains two fields: data size and data buffer and each one has the same size. While SECS receives a packet, it searches for a free page in RX buffer instead of dynamic memory allocation.

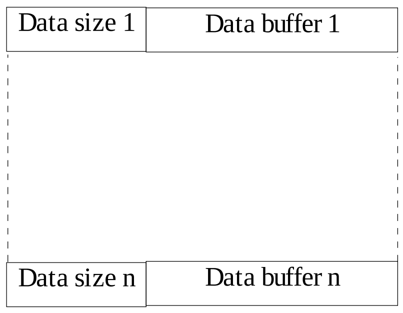


Figure 9−4 RX buffer

While receiving data, raw data size will be assigned to data size field to indicate how many words data in the data buffer field. If the received packet size is larger the data buffer size, secsd will reject this transmission to prevent data overrun. After receiving all data in packet, data page pointer will be written into rx_pipe and prepared to be read from other tasks.

8.3 Data Structures and Member Functions of SECS daemon

SECS block:

```
      typedef struct _secsblock{
            unsigned int      uiDataLength;
            unsigned int *    uipData;
            unsigned long     ulDataCHKSUM;
            int               owner;
```

```
}SECSBLOCK;
```

SECS daemon:

```
extern CDEVSW     secs;

void secs_init(int sci_flag,int sci_brr,int secs_mode,
            unsigned *data_buf,unsigned data_size,
            unsigned data_page);

int secs_read(void *buf,unsigned int size);

int secs_write(const void *buf,unsigned int size);
```

Chapter Ten Future Works

Taunix system is still at early stage from real world operation. There are many features need to be implemented or enhanced. Here comes a brief list.

10.1 Taunix Virtual File System

Currently, all resources in Taunix are accessed by calling corresponding maintenance functions directly. In future release, all resources, like peripherals, semaphores, message pipes, will be assigned an identification number and form an hierarchical structure. It is called Taunix Virtual File System(TVFS). A layer of file system calls would be implemented to provide the interfaces to the kernel resources. Drivers can register or de−register itself into kernel and export device switch functions to the upper tasks. On the other hand, tasks can access them through file system calls.

TVFS should also provide a trap interface that can allow separation of kernel code and tasks. Tasks can issue a trap instruction to access all kinds of kernel APIs. This will be useful while kernel code can be programmed into DSP’s flash memory and tasks can be loaded into RAM on the fly.

10.2 System Call Library

Based on TVFS, it will not very convenient that issues trap instruction for every kernel service. A system call library is a set of helper functions to access kernel APIs.

This library will tend to be release under GNU Library General Public License (LGPL). This will allow Taunix to be used in proprietary applications.

10.3 Installable Scheduling System

There is only one scheduling policy supported in Taunix currently and cannot be changed online. For some applications, scheduling policy may need to be adaptive. Future release of Taunix will support on−line scheduler changing facility. This feature should also be helpful while the kernel code is programmed into DSP's flash and cannot be changed by re−compiling and re−linking with task codes.

10.4 Architecture Porting

As mentioned before, the architecture of Taunix is hardware independent. Currently, Taunix is running on TI's 24x DSPs. It's also suitable for other micro−controllers like 8051, 68000 series or ARM core.

Besides, It's possible to establish a simulation environment of Taunix system. In fact there is a working prototype running on Linux kernel. It uses PThread to simulate Taunix's task, UNIX signal architecture as interrupt source, and host CPU to be the DSP's CPU core. The whole simulating system can be divided into two parts: one is hardware part which simulates the peripherals, interrupts and memory space. The other one is software part that simulates Taunix kernel and drivers. Two parts are linked together by memory sharing and UNIX singling mechanism. This will have great help in education and product planning.

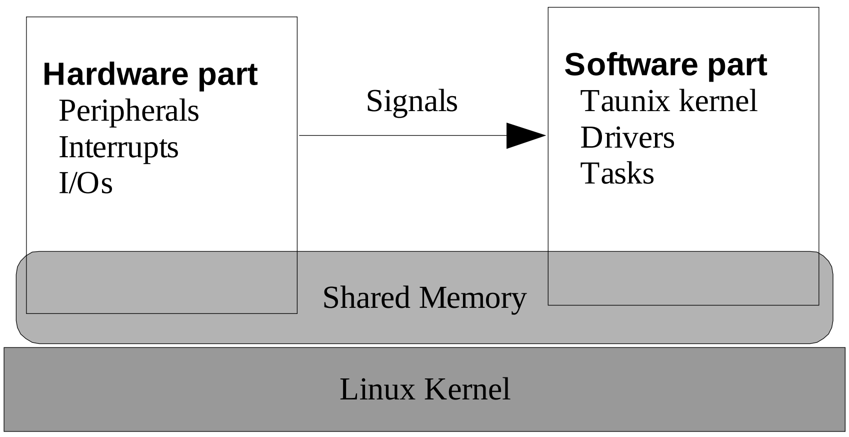


Figure 10−1 SimTaunix Architecture

Currently, this simulation environment, named as SimTaunix, is running Linux native code. That means SimTaunix is source code compatible with DSP version. By migrating with TI’s DSP simulator, if they could do the porting , the software part can be replaced by this simulator and can execute native DSP code in this simulation environment.

10.5 Going to GPL

While developing Taunix, I saw the revolution of open source software(OSS). Taunix is also trying to use part of POSIX system, which is an open system specification, as its basis. I believe the strength of OSS can make software more reliable and better. That’s why Taunix is released under GPL. With LGPLed system call library, Taunix can be used in proprietary system, too. I’m very proud to contribute Taunix to Open Source Community and hope that it will be useful.

Chapter Eleven Benchmarking Index

Test environment: TI 320F243 running at 20 MHz with program loaded in the external SRAM.

- Interrupt latency: 6.3 µs
- Task switch latency: 8.1 µs
- Scheduling latency (from low priority to high priority): 23.4 µs

Code and data size:

Unit: word

Module	***Code size***	***Data size***
TASK	714	233
AT daemon	207	41
Sleep/delay	166	135
Semaphore	730	290
Pipe	844	252
SECS	1151	1144
Priority List	704	0
ADC driver	444	10
Capture unit driver	631	18
GPT driver	414	16
PWM driver	459	10
SCI driver	867	16
WDRTI driver	106	3
Interrupts 1,2,3	229	7

Example: Motor Control

If you treat your beta−testers
as if they’re your most valuable
resource, they will respond
by becoming your most
valuable resource.

Here will provide an example. This is developed for electrical motorcycle in the thesis of Mr. Shu T.Y. The goal of this system is to receive the throttle commands from PC and drive a space vector PWM DC motor with phase advance algorithm. The design can be divided into three modules:

- GetPcCommand:

  As its name states, the main objective is to receive commands from PC and transfer them to proper modules.
- MotorDrive:

  This is the kernel module of this system. This is a periodic task. It accepts reference commands from PC and compares with present motor states, computes the next control outputs. By using space vector PWM, drive the motor to the desired speed.
- GetStatus:

  This is the monitoring module of the system. It monitors the motor current and throttle and also provides these information to MotorDrive module to help to decide the best control commands.

The block diagram is as follow:

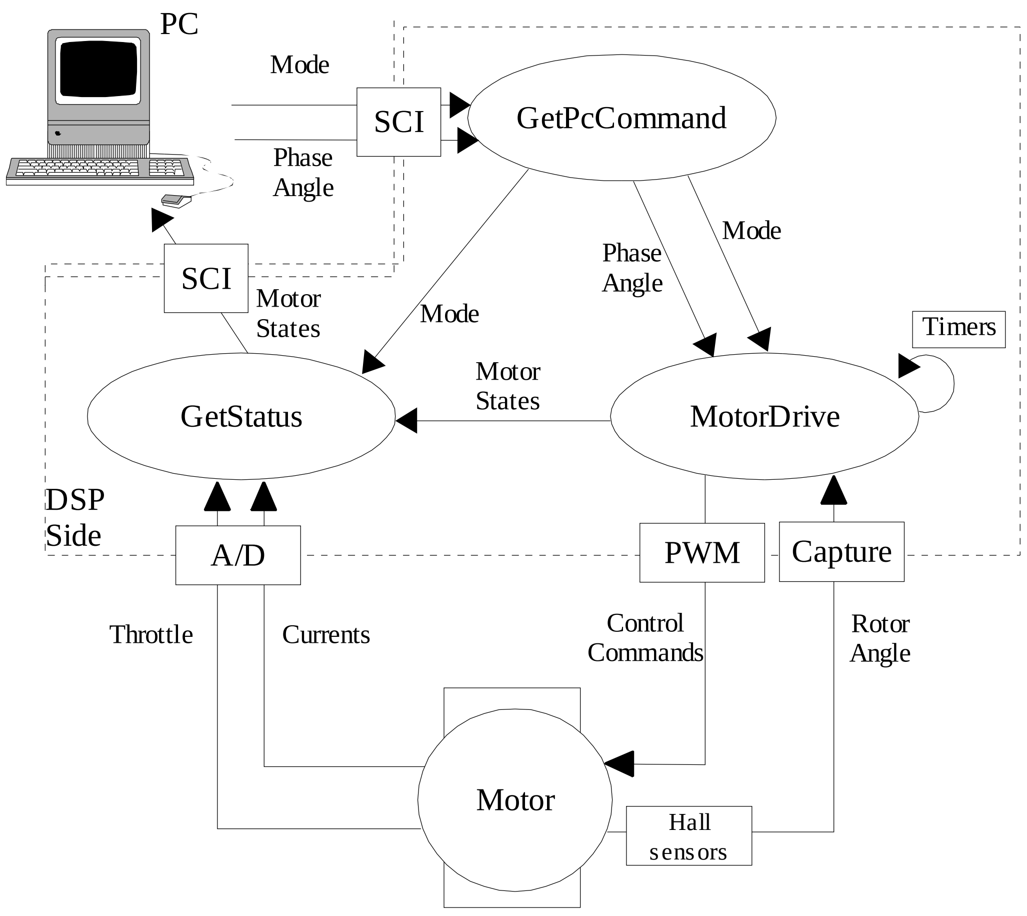


Figure ex−1 Motor control system block diagram

As shown in figure ex−1, the main control algorithm and functionality are concentrated in GetPcCommand, MotorDrive and GetStatus modules. The interface between software modules, PC and motor are SCI, PWM, capture units and analog−to−digital convertors which are device drivers in Taunix. In fact this control system only knows how to control motor but not the peripherals of DSP chip. Device drivers hide the details and form an abstract layer between control algorithm and plant. Because of Taunix multitasking kernel, the whole system can be easily divided into several parts at design stage and each part can be scheduled properly by the underlying kernel.

To sum up, real−time multitasking kernel can greatly help control system programmers modularize their systems and focus the man power to the control law design without concerning the detail underlying hardware specifications. Hardware can be abstract by kernel. As a result, control software can also become intellectual property and can be ported to different system as easy as possible.

References

1. Douglas Comer, Timothy Fossum, *Operating System Design: Volume I The XINU Approach*, 1988.
2. Shem−Tov Levi, Ashok K. Agrawala, *Real−Time System Design*, 1990.
3. Chuang, T.N, *AT&T UNIX® Operating System*, 1991.
4. Halang, Wolfgang A., Stoyenko, Alexander D., *Constructing Predictable Real−Time Systems*, 1991.
5. Van Tilborg, Andre M. Koob, Gary M., *Foundations of Real−Time Computing:Scheduling and resource management*, 1991.
6. Borko Furht, et al, *Real−Time UNIX® Systems, Design and Application Guide,* 1991.
7. Rajkumar, Ragunathan, *Synchronization in Real−Time System: Priority Inheritance Approach*, 1991.
8. Andrew S. Tanenbaum, *Modern Operating Systems*, 1992.
9. Abraham Silberschatz, James L. Peterson, Peter B. Galvin, *Operating System Concepts, Third Edition*, 1992.
10. Jean J. Labrosse, µ*C−OS, The Real−Time Kernel* , 1992.
11. Custer, Helen, *Inside WindowsNT*™, 1993.
12. Hu, J.S., Yin, Yen−tau, *Real−Time Multitasking Kernel Design*, 1995.
13. Uresh Vahalia, *UNIX® Internals, The New Frontier*, 1996.
14. Giorgio C. Buttazzo, *Hard Real−Time Computing Systems, Predictable Scheduling Algorithms and Applications*, 1997.
15. Eric Raymond, *The Cathedral and The Bazaar*, 1997.
16. Bruce Powel Douglass, *Real−Time UML, Developing Efficient Objects for Embedded Systems*, 1998.
17. TMS320F243/F241/C242 DSP Controllers Reference Guide, System and Peripheral.
18. TMS320F243/F241/C242 DSP Controllers Reference Guide, CPU and instructions.
19. TMS320C2x/C5x Optimized C Compiler, User’s Guide.

Appendix A : GNU General Public License

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a)Accompany it with the complete corresponding machine−readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b)Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine−readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c)Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such
parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the

Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

  7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

  10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix B : Taunix Source Code List

Taunix is released under GPL. Source code listed here is version 0.7.4 and for your reference. This version is considered as alpha version of codes and no warranty is provided. Use it under your own risk. Zipped file can be obtain from me by email: maxyin@StarTrekMail.com or maxyin@alumni.nctu.edu.tw. Please feel free to send me your opinions. I'll appreciate for any professional advises, bug fixes and, well, your contributions.