Designing Real-Time Systems

By Nikhil Varma, Encercla Inc.

Disclaimer

This document is meant for free circulation; however no part of this document can be reproduced in any other documents. Every precaution has been taken to prepare this document however if any errors exists please contact the author at the following address

Nikhil Varma
nvarma@encercla.com
www.encercla.com

1. TASKING MODEL 5

1.1. Decomposition of system at task level. 5

1.1.1. Periodic tasks 5
1.1.2. ASYNCHRONOUS TASKS (aperiodic or event driven) 5
1.1.3. Control Objects**:** 6
1.1.4. User Interface 6
1.1.5. I/O Structuring 6
1.1.6. Task Cohesion 7

1.2. Assigning Priority /Scheduling Policy: 7

1.3. Task Scheduling: Static or Dynamic ? 7

1.4. Common Errors/Bad Programming Practices: 8

2. INTERRUPT SERVICE ROUTINES (ISR) 8

2.1. General Design Guidelines 8

2.2. Suggestion 9

3. INTER-PROCESS COMMUNICATION 9

3.1. Semaphore 9

3.2. Message Queue 10

3.3. Events 10

3.4. Deadlock 10

3.5. Priority Inversion 11

3.6. Starvation 11

4. DYNAMIC MEMORY MANAGEMENT 11

4.1. General Design guidelines 11

4.2. Suggestions 12

5. A CASE STUDY 12

5.1. The System 12

5.2. The Problem 12

5.3. The Solution 12

6. REFERENCES 13

Abstract

This document is intended to help in designing Real Time Systems. The document discusses the design considerations to be practiced so as to achieve a predictable and stable real time system.

This document addresses the design and implementation issues with tasking model, ISR, IPC mechanism, & memory management. It identifies the common errors & problems which designers and programmers frequently face and if addressed at earlier stage, it significantly reduces debugging time & efforts.

You will learn how to split an application into separate tasks, assign priorities to tasks, use kernel services such as semaphores, message mailboxes, message queues, time delays etc. You will also see how interrupts interact with tasks. This session assumes that you understand how a real-time kernel works.

Introduction

The history of Real time system dates back to 1971 when INTEL introduced the world's first microprocessor. This processor was developed on request by a Japanese company Busicom asked INTEL to design a set of custom integrated circuits for their new calculator models. Instead INTEL designed 4004 which could be used throughout their line of calculators.

Scope

- This document does not cover any RTOS internals. It is expected that reader understands RTOS concepts.
  - The intended Audience is a Project Team involved in developing Real-Time Applications.

Many embedded applications are designed as a Super Loop system but cannot benefit from the use of a real-time kernel. This document will describe how to use a real-time kernel in an embedded system.

1. Tasking Model

Real time systems typically have multiple tasks executing simultaneously. Each task consisting a set of statements that executes sequentially at some priority level.

System performance will be depend on :

- *Identifying the Concurrency* i.e. processes that run concurrently.
- Decomposition each process into task level.
- Assigning priority of each task.

One of the first problems in designing real time systems is to decompose the system work into tasks. An immediate and obvious question is whether to have large number of tasks or fewer tasks?
The following discussion will answer this question.

1.1. Decomposition of system at task level.

The decomposition of tasks at system level involves the following major considerations.

Identify the Concurrency
This helps in the following ways
1) Gives a brief idea of the system behavior.
2) Task decomposition and cohesion decisions are easy to make.

Internal task structuring
Tasks can be broken down into the following categories:

1.1.1. Periodic tasks

Periodic tasks are scheduled for execution at regular interval of time. i.e. at every T time units are typically characterized by hard deadlines.
These tasks have the following nature:

a) **Hard start time** : These tasks start at a specified time. The start time in these tasks cannot vary at any point of time.
b) **Hard completion time**: These tasks may start at any point of time but have to finish before the deadline.
c) **Hard cycle times**: The second cycle of the periodic task should start within a specified time.

Observation: One of the drawbacks observed is that a drift may occur over time. This time depends on how many times the task was preempted. The higher the priority the less the chances that preemption would affect the drift.

1.1.2. ASYNCHRONOUS TASKS (aperiodic or event driven)

The majority of tasks in most real time systems are asynchronous. These tasks run on a demand basis and are designed to handle internally generated events. The following sequence defines such a task

```
While TRUE
    Wait for event
    Process…
    Process…
End while
```

These tasks are driven entirely by asynchronous data. When designing such tasks the following have to be taken into consideration.

a) **Data structure**: The data type and format plays an important part in tasks as they can save valuable debug time later.

b) **Identify proper mechanism for IPC:** The inter-process communication mechanism is the most critical aspect in asynchronous tasks.

c) **Consider nature of event/trigger:** The following would help in this regard
-*Minimum inter-arrival time* with *average interval duration* and *standard deviation.*

d) **Bursty messages:** Events in bursts can be lost if proper mechanism is not implemented to track them. They can however be tracked by using the following

- *Counting Semaphores*
- *Message Queues*

Suggestion: Consider worst case scenario and then resolve the tasks

1.1.3. Control Objects**:**

Individual tasks performing some control functions are called control objects**.**
These are often used to implement state transition diagrams. A state machine is implemented as a series of asynchronous tasks by allocating each state its own task. This strategy however falls apart in situations where state transition is complex because there is a large volume of data that is to be transferred. A different strategy is to make the control task reentrant and for each instance of the state machine, a copy of the task can be created.

1.1.4. User Interface

It is advisable to isolate user interface functions into their own tasks, as the response time is easier to control. The systems, which are user driven, should have the user tasks at a high priority.

1.1.5. I/O Structuring

I/O structuring partitions events under three categories:

- **Interrupt Driven Events**: It should be made clear which events are to be processed at the interrupt level and which events at the task level so as to reduce the interrupt latency.
- **Polled I/O Events:** These are implemented as an I/O event when the operating system schedules on a fixed time basis.
- **Output Only Events:** They are often designed as reentrant utilities and not as tasks. The advantage being that high priority tasks do not wait for the output event and low priority tasks are not preempted each time they use the output event.

Suggestion: Assign separate task for each I/O device e.g. Scanner, Printer, Serial I/O, Display etc.

1.1.6. Task Cohesion

Task cohesion simplifies the tasking model and minimizes the tasking overhead. The identical features like priority are taken into consideration to minimize complexity. Cohesion can be done according to the following criteria:

- **Functional Cohesion**: This type of cohesion can be done in closely related functions

- **Temporal Cohesion**: This is a cohesion method where dissimilar functions, which are activated by one event, are grouped into one.

- **Control Cohesion:** All functions driven by a specific control structure are grouped together. This reduces the overhead of transferring large volume of data.

1.2. Assigning Priority /Scheduling Policy:

Grouping of functions should be based on their priority this reduces overhead such as task switching. If there are several functions with high priority and cannot be preempted then the best policy is to group them in one task so that one function can preempt the other.

Observation: *System less than 70% loaded may fail due to poorly designed scheduling policy.*

Priority of a task is assigned based on the ***urgency*** of the task and its ***importance.***
Importance is the value to the correct system performance of the completion of the specific action
Urgency refers to the nearness of its deadline.

- Tasks with a hard deadline or which are *time critical* should be assigned a *high priority*
- Tasks that require *heavy computation* should be assigned *low priority.*
- Tasks that depend on external on external events should have a higher priority than the tasks triggered by internal events. E.g. In a reception/transmission process , reception should have higher priority than transmission process

1.3. Task Scheduling: Static or Dynamic ?

<u>Static and Dynamic Tasks</u>

It is important to decide whether a task has to be made static or dynamic. Normally all tasks are made static. Dynamic tasks come into picture when we deal in mutually exclusive tasks triggered by some event , e.g. Reception and transmission tasks cannot co-exist together if we have a single exit point (modem) , hence these tasks can be made dynamic.
An ideal situation could be all tasks be made static due to memory constraints (specially in real time systems) this is not possible always so some tasks have to be made dynamic.
There is however, a risk involved in dynamic tasks. If all the dynamic tasks are triggered at one go there is chance, that memory will not be available at one point of time. This risk has to be kept in mind during analysis to minimize the chances of a memory overflow.

Most kernels provide services to create tasks. A task can typically either be created statically (before you give control to the kernel) or dynamically (as your application executes). Creating a task simply means that you tell the kernel that it needs to manage the task. Most task are generally created statically in embedded systems.
Kernels will also allow you to delete tasks dynamically. You can use this feature to 'abort' a task that is no longer needed. For example, your system may need to perform an emergency stop. In this case you may want to abort control loops (tasks) and have a 'shutdown task' to position control outputs to their safe states.

1.4. Common Errors/Bad Programming Practices:

1. A task has a return statement
   A task must never have a return statement because it makes the system behavior unpredictable. The task should be in a loop.

2. Use of 'Suspend' - 'Resume' in pairs indicates bad design

3. Either get a resource and immediately check/wait for another combined tasks, or split them apart incorrectly.

4. Change place within a task when an event occurs: (e.g. use of ASR in pSOS) - might combine multiple tasks into one task.

5. Avoid delete of another task. Deleting task may not get a chance to 'clean-up'.
The same applies for 'restart' also.

6 Never base task operation on passing of time. A 'faster' task will only get done fast, if it begins work.

Suggestions:
Consider worst case stack size for each task. Optimize it, if required with common debugging techniques.

2. Interrupt Service Routines (ISR)

ISR constitute the most complex part in any real time systems. Fortunately, there are only a few ways that interrupts are commonly handled. By far the most prevalent way is the Vectored Scheme. A hardware device either external or internal to the I/O port asserts the CPU's interrupt input.

2.1. General Design Guidelines

- The ISR are special purpose codes invoked for a special task. They should be kept as short as possible. For example, if we invoke an interrupt for taking in any data from the external world, all processing related to the data gathered by an interrupt should be handed over to another task. This is done to prevent any chances of data loss, which might come during the processing of data, by the ISR. Ideally an ISR should have 10-15 lines and be not more than a page including the comments.

- The ISR should communicate with a task by a proper mechanism. E.g. message queue, semaphore or events.
- The ISR code should be written in a readable and understandable format. Crummy code is hard to debug. Crummy ISRs are virtually un-debuggable.
- Code for an embedded system should never be written until an interrupt map is laid out.
- It is also important to estimate the interrupt frequency,

- Loops should be avoided, long complex statements should not exist in interrupt service routines.
- The code should be reentrant. It means that during execution it can be re-invoked by itself or any other routine. Also check that the function calls made from the ISR are also reentrant.

2.2. Suggestion

Whenever possible write the ISR's in C and use the local variable scoping. Reentrant C code is orders of magnitude easier to write than reentrant assembly code.

- All interrupts should be maskable. NMI (non-maskable interrupt) should not be used other than catastrophic events.
- All the unused interrupt vectors should be filled with a pointer to a null routine. During debug, insert a breakpoint in all such routines.

If the maximum interrupt rate is known a performance analyzer should be used to measure the maximum time in the ISR. If the time exceeds the fastest interrupts, there's very likely chance that a latent problem is waiting to pounce.

3. Inter-Process Communication

It is crucial to identify and implement a proper Inter-Process Mechanism. Real Time Systems supports two type of communication across thread boundary one is lightweight where calling operations across thread boundary is permitted and the other is heavyweight where hardware enforces segmented address spaces for threads. To a first time user the simplest way to communicate between threads is to directly call operations across thread boundary. This is possible in lightweight system but heavyweight systems this would not be possible. However crossing thread boundaries would complicate the resolution of association amongst threads. Mutual exclusion and reentrancy issues would also come in picture. However these problems can be sorted out by using the following Inter-Process Mechanisms

3.1. Semaphore

The following guidelines should be followed when using semaphores:

1. Protect every shared & global data structures, resources & devices with Semaphores

2. Make sure for release the semaphore (or any resource) under any condition.

3. Do a code walk through and check the order of wait-release calls. E.g. in pSOS, if calls are reversed, by mistake - sm_v & sm_p, it is difficult to catch, sometimes not covered in test, system may fail eventually.

4. Check for FIFO/Priority option, danger of priority inversion when 2 or more task waiting for semaphore or message queue.

3.2. Message Queue

The following guidelines should be followed when using Message Queues:

1. Use Message Queue to pass information between tasks.

2. Make separate message queue for each task to receive.

- *Consider max no of messages while defining a queue length.*

3. Message from Low priority Task to Higher Priority Task: Often message queue & task can be replaced by a function call.

4. Message from Higher priority(H.P) Task to Lower Priority(L.P) Task: HP task can out-produce task LP's consumption in short bursts. Use Message queue of sufficient length.

5. Queue size > Maximum number of messages that task HP can out-produce Task HP.

6. Unlimited message length queue can be used, but consider danger of memory consumption!!!

3.3. Events

The following guidelines should be followed when using Events.

1. Synchronize task with Events.

2. Events are very useful when tasks are required to get ready individually on combinations of signals.

3. Take care for under-counting / over-counting. Events can not count.

4. Avoid Events for 'Bursty' signals.

5. Do not use when there is a danger of setting an already set flag.

3.4. Deadlock

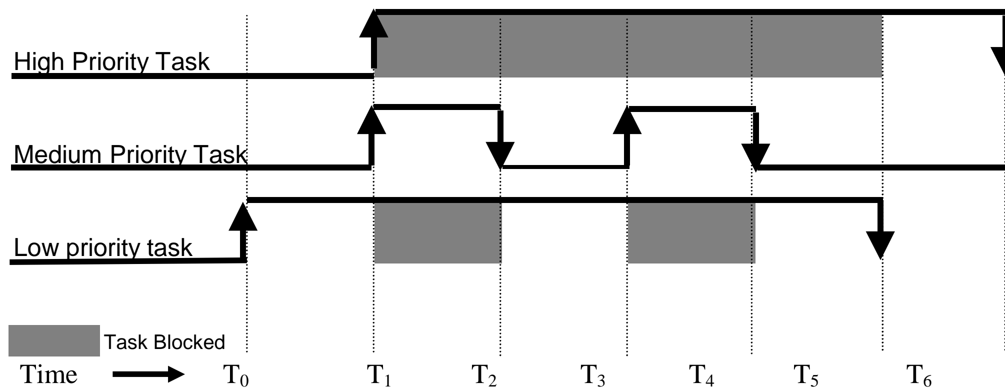
Deadlocks should be avoided in Real time systems because their existence in the system brings in system overhead, latency and therefore affects performance and predictability. They can be avoided in the following ways :

1. Avoid getting all the resources in same order if it is required to get more than one resource.

2. Avoid circular dependency between tasks and resources

3.5. Priority Inversion

Priority inversion occurs when a high priority task is blocked waiting for a semaphore that is held by a low priority task. The situation is illustrated in the following figure. Here 3 tasks : high priority, medium priority and low priority tasks exists. Low becomes ready first (indicated by rising edge) and shortly thereafter takes the semaphore. Now, when high priority task gets ready it must block (indicated by the shaded region) until low is done with their shared resource. The problem arises when a medium priority task which does not require any access to that resource gets to preempt the low priority task and runs denying the high priority task any processor time.



How to avoid Priority Inversion:

1. List out all resources and tasks with priority that waiting for each resource.
2. Use priority-ceiling protocol/Priority Inheritance : In this protocol the Operating systems automatically increases the priority of the task with low priority to the priority of the high priority task as soon as the high priority task starts waiting for the semaphore/resource held by the low priority task.

3.6. Starvation

Starvation occurs when a task is completely denied processor time. This generally occurs due to bad priority settings. Taking care of while assigning priority could tackle this problem.

4. Dynamic Memory Management

Dynamic Memory is used at the cost of speed but it brings along with it some advantages as well viz. clarity and economy of memory. This is a critical part in Real time systems because they generally have very less memory space.

4.1. General Design guidelines

The basic things to be followed when Dynamic Memory Allocation is done is a follows

1. Automate check for memory leak
2. Take care of fragmentation while using variable size buffers.
3. Use Low watermark technique during debugging.
4. Optimize buffer size.

4.2. Suggestions

Although some real time operating systems support the C library functions *malloc* and *free* these functions should not be used because they introduce the word unpredictable into the system hence making them non-compliance to the basic idea of being Real Time.

Instead the Real time Operating system provides a buffer pools which have buffers of equal size. Systems calls to get a buffer and free a buffer are also provided.

5. A Case Study

This case has been taken from a real life experience which shows the importance of a good design in a real time system

5.1. The System

NASA (Nation Aeronautics and Space Agency) had sent a Mars Pathfinder mission which had landed on the Martian Surface on July 4th 1997. There was a great deal of coverage of the mission showing how the pathfinder landed onto the surface but soon mission started to go on the failure path when the spacecraft started experiencing system resets resulting in data losses

5.2. The Problem

The Pathfinder had an “Information Bus” which worked as a shared memory in the system. There were many task which used this bus for their activities. The access to this bus was though *mutex*.
The system had a low priority task called the Meteorological Task which used to acquire the mutex for the information bus , use the bus for its activities and then release the bus. However if an interrupt tried to schedule the Information Bus thread it would go to a blocked state because the mutex was held by the low priority Meteorological Task. The system had a medium priority task ,the Communication task which would run sometimes. It was possible for an interrupt to occur that would schedule the medium priority task (Communication task) to run while the high priority was still in blocked state.
The system had a watch dog that would run at intervals to check the system state. On observing that the high priority task was in a blocked state for a long time and a medium priority task running it would reset the system, consequently losing the data.

5.3. The Solution

On simulating similar system cases in the lab it was observed that the problem was because of PRIORITY INVERSION. It was concluded if the low priority task (Meteorological Task) inherited the priority of the high task(Information Bus task) it would not allow and medium priority task to preempt it. Finally the Priority Inheritance Protocol was implemented in the mutex and the code uploaded to the spaceship. No more resets occurred

6. References

L. Sha, R.Rajkumar and J.P. Lehoczky:
Priority Inheritance Protocols: An Approach to Real time Synchronization, IEEE Press
Phillip A Laplante:
Real-Time Systems Design and Analysis, IEEE Press
Jane W.S. LIU :
Real time Systems, Pearson Education
Bruce Powl Douglass:
Real-Time UML, Addison Wesley.
Gomaa, Hassan:
Software Design methods for concurrent and Real-Time Systems.
Developing pSOS System Applications, Integrated Systems, Inc, 1996.

Website references:
http://www.kohala.com
http://www.esconline.com