INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

The United Nations University

SMR/774 - 24

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

DESIGNING AN INEXPENSIVE REAL-TIME SYSTEM: A CASE STUDY

Ravindra KARNAD
Five D-Electronic Technical Services
No.25 Anniyappa Garden
6th Main 10th Cross
Thippasandra
560 075 Bangalore
INDIA

DESIGNING
AN INEXPENSIVE REAL-TIME SYSTEM:
A CASE STUDY

17th October 1994
Trieste

Ravindra Karnad
INDIA

At the tail-end of this course, let us recall some of the basics of real-time systems:

- Any system will process it's inputs and respond with certain outputs within a reasonable ***response time*** .
- But a ***real-time system*** is one which ***must satisfy explicit response-time constraints*** else it may be considered to have FAILED!
- Some systems can tolerate a delayed response with a degraded performance and are called ***soft real-time*** systyems.
- Others can tolerate only marginal degradation (i.e. can afford to have a delayed response once in a while) are called ***firm real-time*** systems.
- Systems which just cannot tolerate a delyed response are ***hard real-time systems***

In these systems depending upon:

- the ***cost constraints***,
- the ***complexity*** of the processing required and
- and the ***time available to process*** these inputs before the response can be made available

we could decide about the sophistication of the system in terms of the :

* Word size and Speed of the CPU
* Type of OS

These decisions would result in a trade-off between cost and effort:

- Ready made systems are ***high cost*** with ***low development effort***
- Do-it-yourself systems are ***cheap*** but ***development intensive.***

We could identify three categories in this range of solutions giving a different mix of ***cost*** and ***performance:***

- **High-end:** Example:A PC running a real-time OS with off-the-shelf I/O cards with device-driver.
- **Medium:** Example: A PC with a RTOS but a do-it-yourself card and driver.
- **Low:** Small embedded system with it's own RT Kernel

We shall look at a small real-time application which employs the low-end solution:

- ***Small embedded system*** with 8 bit microprocessor/microcontroller.
- A do-it-yourself ***Real-time Kernel*** .
- ***Cross-development*** of assembly language software on PC .

We shall construct a simple ***hypothetical real-time problem*** for which we examine the requirements of computation and then go on to define the requirements of the real-time kernel. Consider the following problem:

On a machine a carriage performs a reciprocatory motion between two fixed limits. As the carriage reaches either limit a signal is received from a sensor. (However due to mechanical vibration and over-shoot the signal from the sensor is not "clean "). After every cycle of the oscillation, an actuator is to be turned on for 50ms. The velocity of the carriage, and the sensor determine the following :
The period of Oscillation is about 300ms
The duration of the signal from the sensor is about 60ms
The bounce on the sensor lasts for 15ms.
The pressure of the oil in the system is to be logged at 100ms intervals in and displayed every 1 sec.

The **first step** in the design is to ***identify the number of tasks or processes*** that would have to be supported by the software. In the example above, the processes are:

- The scanning of the sensors.....P1
- The turning on and off of the actuator.....P2
- The data logging of the pressure.....P3

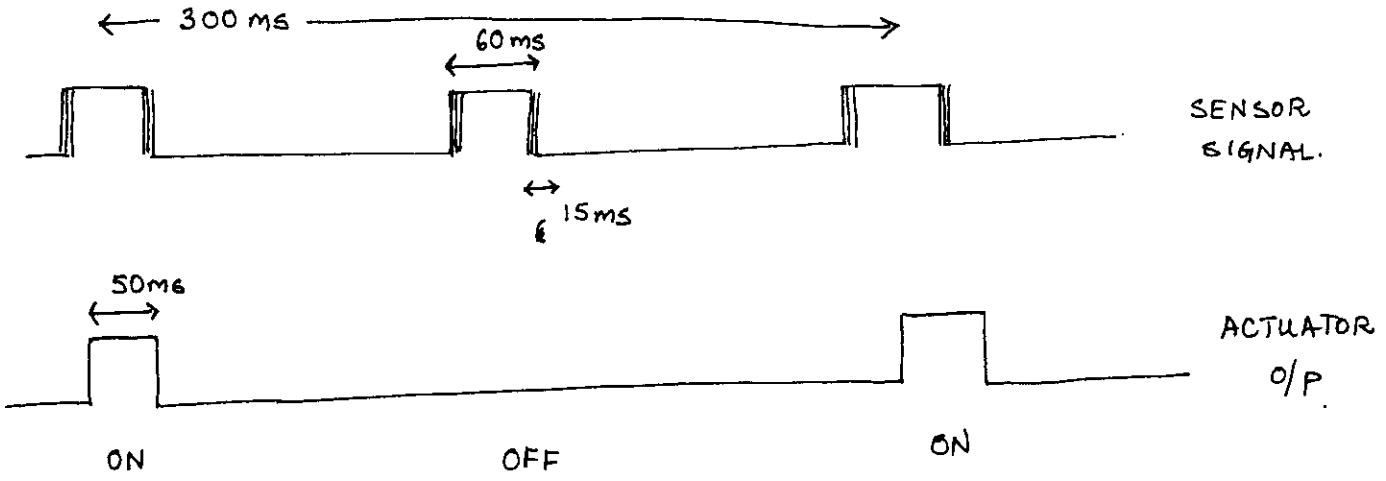
It must be recognised that these are concurrent processes.

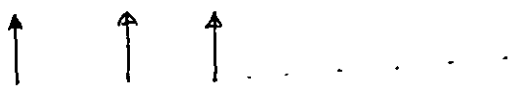
The **second step** is to ***estimate the response-time requirements*** of the system. Here this has to be done ***for each process*** that has been identified. In our problem, the scanning of the sensors is to be done fast enough to debounce and recognise the signal within 60ms. for the process P1. Likewise the process P2 is to be active only once in every period of oscillation. The process P3 has to be activated once every 100ms.

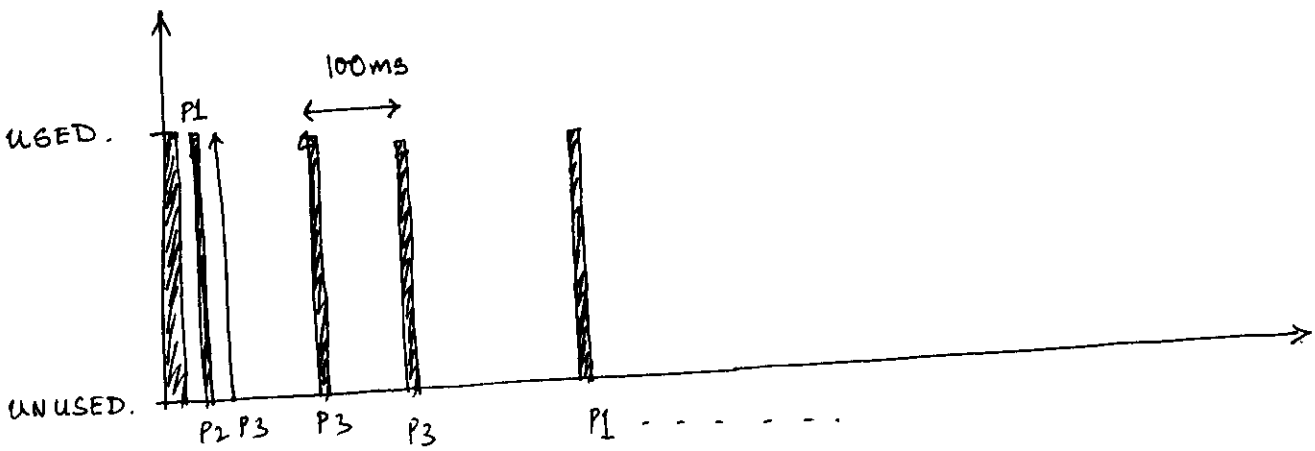
It is interesting and important to recognise that although the three processes are concurrent, for each of these processes, there is an ***idle-time.*** During this idle-time the ***process has no need for the CPU.***

It is this feature of most processes that is exploited by the kernel to run many concurrent processes and to keep the ***utilisation of the CPU*** to the level that is actually required.

This has great relevance to low-power applications where the CPU can be put into a ***power-down mode*** when no process needs the CPU.







PROCESSOR OCCUPANCY.

Having identified the processes and their real-time needs, we can in the **third-step** look at whether these processes need to communicate with each other. In this case P1 needs to inform P2 when a complete cycle of oscillation is over.

A small ***real-time kernel*** is called for which provides simple mechanisms for:

- ***Scheduling*** one of several concurrent processes
- ***Switching*** from one process to another with the correct ***context***
- Providing a simple mechanism of ***inter-process communication***

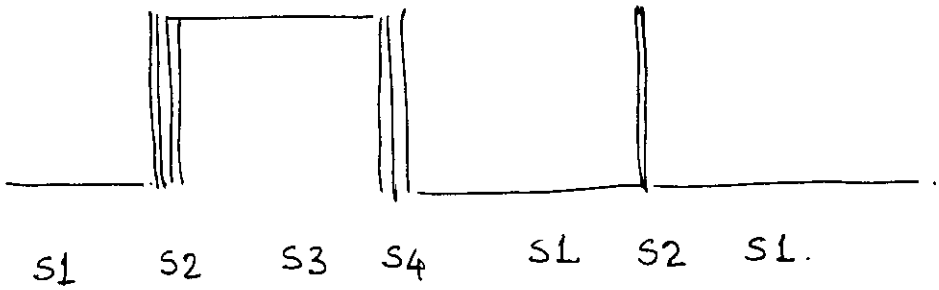
The first two tasks are done by the ***scheduler*** and the ***despatcher*** of the kernel.

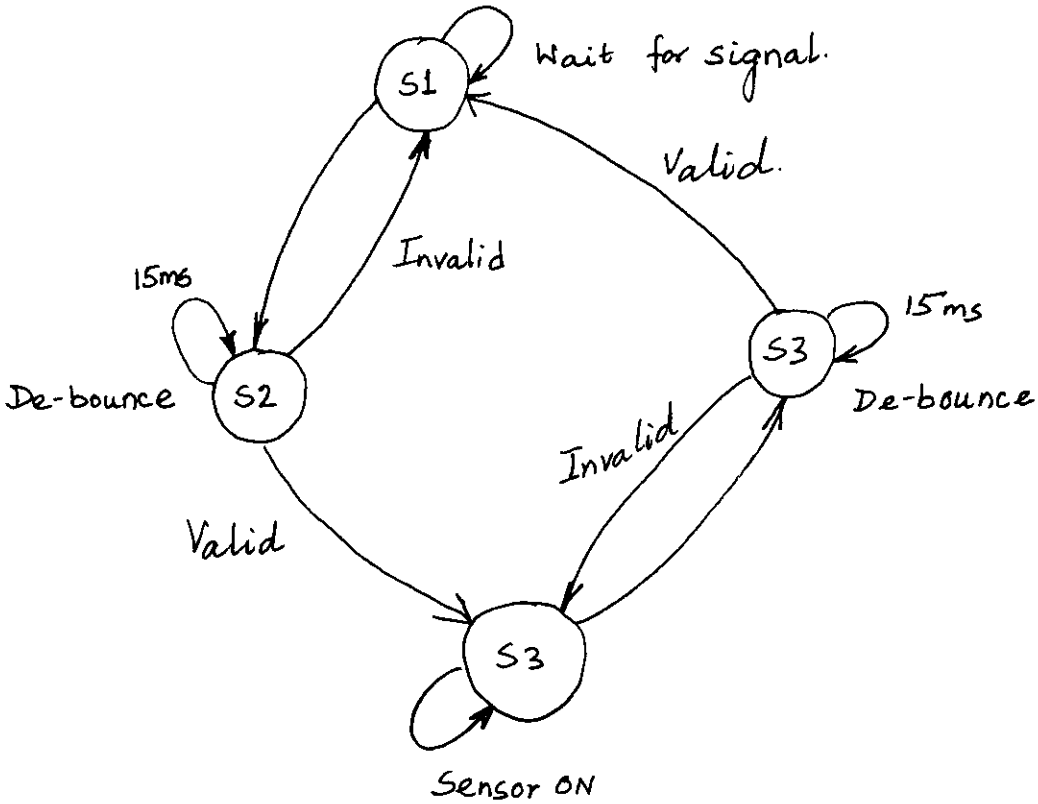
Before we look at how we build a simple kernel to perform these functions, let us look at a simple method of writing the software for each process:

One of the traditional methods of hardware design can be employed namely the ***Finite State Machine*** approach.

The FSM method has the following advantages:

- **Easy** for the despatcher **to save and retrieve the context** of a process.
- Each state has a **well defined entry and exit point** and hence a small self-contained routine called the state-rroutine.
- Easy to test and debug since state transitions can be forced and changed with ease. This makes it very easy to narrow down the culprit code even in a system which behaves erraticaly.
- Modifications and alterations are extremely simple.In most applications there will be a need to enhance features, or to alter specifications at a later date on later implementations. These would necessitate the addition of more processes and also alter the state diagram.


S1
S2
S3
S4
SL
S2
S1.


S1
Wait for signal.
Valid.
Invalid
15ms
De-bounce
S2
15ms
S3
De-bounce
Invalid
Valid
S3
Sensor ON

FSM for P1.

As an example of the FSM method we shall look at just the case of how the process P1 is implemted.

Since the problem states that there will mechanical bounce, we can identify four states of this process:

- S1: state where the sensor has not sensed any signal and is waiting for a signal.
- S2: when the sensor has sensed a signal and is waiting to debounce it.
- S3: where the sensor is in the "Sensed" condition
- S4: where the sensor is waitnig to be de-bounced again.

Note that in each of these 4 states, the processing required is extremely simple. and in some cases even trivial!

Having identified the processes and their real-time needs as well as drawing out the state diagrams for each process we can now proceed to build a small kernel which schedules processes, despatches them properly to the correct context and also provides a simple mechanism for IPC.

The kernel proposed here uses ***non-preemptive scheduling.*** This means that each process is allowed to run to completion and is not interrupted.

This goes well with the FSM method since the scheduler can permit each process to complete it's state-routine before despatching the next process.

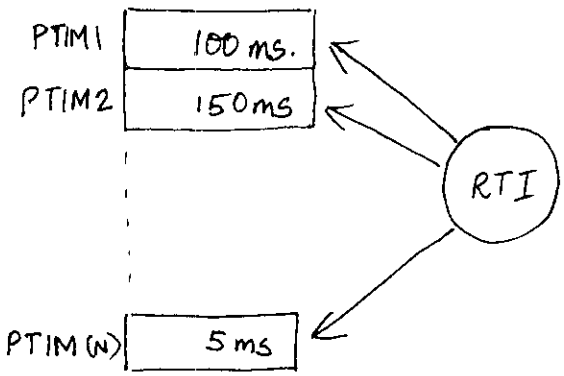
Each process must have a sense of time so that it can perform it's job within the specified time limits. This is done by a ***time-keeping function*** (essentially a tick of a 1ms clock) of the scheduler. Each process will have associated with it a **timer** which tells the scheduler when it must run next.

Additionally since we are following the FSM method, each process must have a **state** associated with it which tells the despatcher which state of the FSM the process must enter i.e. it must ***retrieve the correct context***

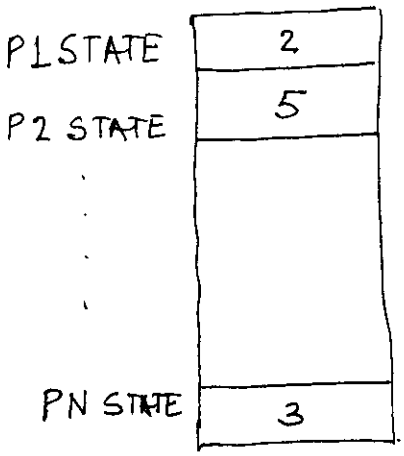
At start-up all process states and timers are initialised.

The kernel then sets up the concurrent processes and executes them as follows:

1. The **scheduler looks at the timers** of each process and the real-time interrupt counts down these timers. When a timer goes to zero, it is time to invoke this process.
2. The **despatcher then looks at the state number** of this process and calls up the appropriate routine.
3. The state routine of this **process performs it's task and terminates.** However before quitting it's state routine it updates it's **timer** to tell the scheduler **WHEN to invoke** it and also updates it's state number to tell the despatcher **WHERE to pass on control** when invoked.

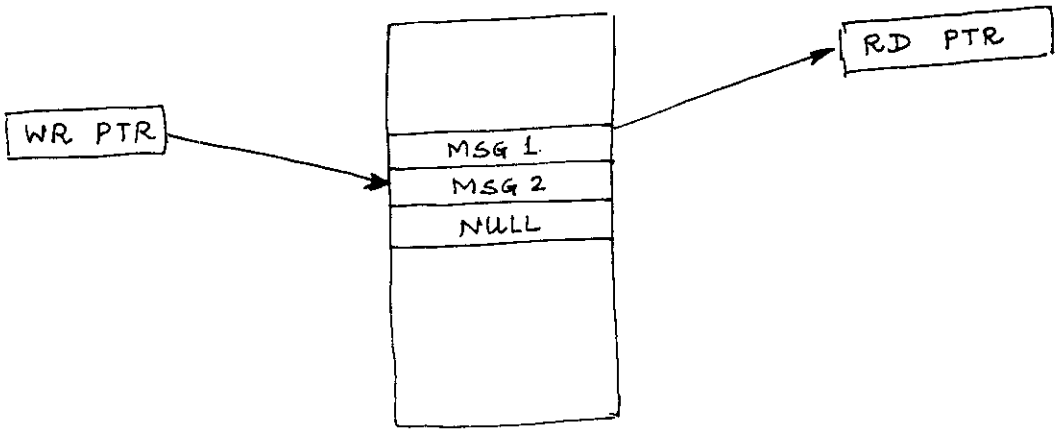


TIMER LIST




```
JMP   P1SØ
JMP   P1S1
      .
      .
JMP   P2SØ
JMP   P2S1.
      .
      .
      .
JMP   PNSN.
```

JUMP TABLES



Inter-process communication is implemented by a **circular ring buffer.** The reading and writing are done by using two separate pointers. A NULL character is put into the buffer at the end of the last written message.by the producer process. The consumer process reads upto the NULL. This method will work if the long term average of the reads and writes is the same. The size of the buffer depends upon the sudden bursts of messages produced by the producer process and is dependent upon the application.

In case the application needs to service other **device interrupts,** then the ISR must just do the bare minimum (like read the sampled data and put it in an internal buffer) and then convey this information to the necessary process. The concerned process can retrieve this information from the internal buffer when it next runs. The ISR could set the state and timer of the concerned process so that it processes this data in the next round of the scheduler.