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

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/643 - 24

SECOND COLLEGE ON
MICROPROCESSOR-BASED REAL-TIME CONTROL -
PRINCIPLES AND APPLICATIONS IN PHYSICS
5 - 30 October 1992

AN INEXPENSIVE (LOW-LEVEL) IMPLEMENTATION FOR REAL-TIME CONTROL

A Case Study

R. KARNAD
FIVE-D Electronic Technical Services
102, Ganesh Appts
10th Mallswaram
Bangalore 560 003
India

AN INEXPENSIVE (LOW-LEVEL) IMPLEMENTATION FOR REAL-TIME CONTROL

A CASE-STUDY

Ravindra Karnad

INTRODUCTION:

Many applications of control,despite being real-time critical, may not warrant the use of sophisticated real-time operating systems and could just as well be implemented using simple but effective techniques of system design.

This would be especially true in developing countries where projects, more often than not, operate on a tight budget and one must use the most economic solution for a given application.

In this case-study we look at an implementation of a controller for a numerically-controlled machine. To keep things simple we shall not go into the details of the entire system but only examine some of the functions of the machine which would serve as good examples of real-time critical tasks.

We shall also see how these tasks are effectively handled by very simple primitives which are most suitable for low-level real-time programming. (By "Low-level" we mean programs written in assembly language with a small scheduler and no real OS as such)

Throughout this discussion we shall (intentionally) avoid any reference to any particular microprocessor family to highlight the fact that the same techniques could be employed with virtually any microprocessor.

THE APPLICATION:

The machine is a honing machine used to grind the inner surfaces of cylindrical components. The cylindrical component is placed vertically and the abrasive stones are mounted on a honing head which enters the cylindrical component. The honing head has two simultaneous motions: vertical oscillation between the extremes of the cylinder as well as rotation about the vertical axis. The abrasive stones thus trace out a helical path along the inner surface of the component. In addition, the stones can be given small movements (in a few micrometers) in the radial direction so that the amount of material removed by the stones can be controlled and hence the diameter of the component is controlled to within a few microns.

As far as the controller is concerned, the basic functions boil down to doing the following ***tasks:***

1. Accepting commands from the operator
2. Controlling the vertical movement of the carriage.
3. Sensing the vertical position of the carriage.
4. Controlling the outward motion of the stones with stepper-motor
5. Sensing the actual motion of the stepper motor.
6. Monitoring the rotational speed of the spindle
7. Miscellaneous functions like turning ON the coolant etc

   Etc.

For the proper functioning of the machine, not only should these tasks be carried out in the correct sequence but also accomplished within the given time constraints imposed by the system.

Consider tasks #2 and #3 from the above list. Task #3 would sense the vertical position and would detect the reversal point of the carriage. Task#2 must respond fast enough to cause the carriage to reverse well before it could cause any damage to either the component or the honing head (which costs a few thousand dollars). At the same time the controller must also be able to respond to an operator's command or in fact do each of the other tasks within it's own time constraint. This makes it amply clear that the controller must be able to handle many ***concurrent tasks*** with ***real-time constraints*** .

THE IMPLEMENTATION (AN OVERVIEW):

We shall see how we can arrive at a simple mechanism which permits us to have many (low-level) programs concurrently running so that the controller is able to handle such a real-time load and effectively perform it's job of controlling the functions of the machine. In brief, the system design will follow the following steps:

First, the overall functions of the controller are listed out as tasks (as has been done above.)

Next, for each of the tasks we determine some timing parameters which affect it's real-time response. This would include:

* How often this task must be executed,
* The amount of computation to performed each time and
* The time within which this computation is to be performed.

Next, each task is broken down into a finite-state machine. This simplifies, to a great extent, not just the programming effort but also the effort in de-bugging the system, as well as incorporating changes, updates and even radically new features.

Having identified the keyboard task, we can now estimate the following real-time parameters:

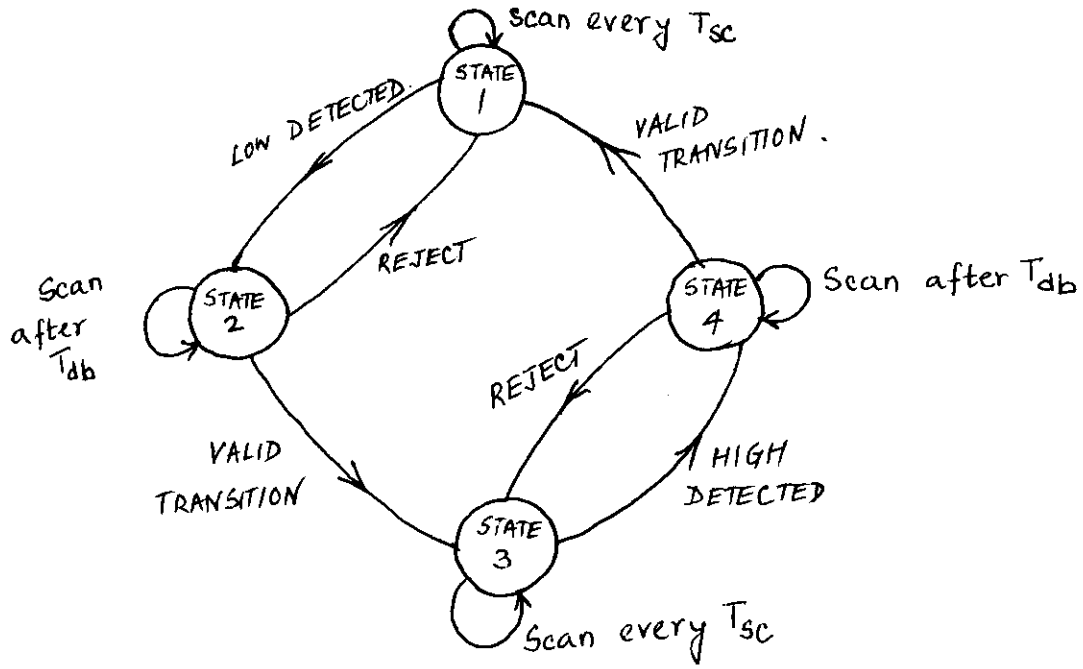
* How often must the process of the Key-board task run ?
* How fast must it respond ?

We can safely assume that the machine operator (who will be typically slower than the fastest typist) can press no more than (say) 5 keys/sec. This ofcourse means that there would be a duration of atleast 200 msec between two key-depressions. It would be reasonable to assume (One must keep making reasonable assumptions or a good guesses whenever data is absent) that of this 200 msec, the finger of the operator is on the key for atleast 100 msec and the next 100 msec it is travelling to the next key. This will guarantee that each key is depressed for atleast a minimum of 100 msec. This means that it is sufficient to scan the entire keyboard once every 100msec. The scanning operation, as explained above, would involve driving a low on each column and scanning each row for a logic-low. This could be done in a few 100 micro seconds. This also indicates that the percentage of time that the key-board task uses the processor is very small (much less than 1%). This simple arithmetic reassures us that the keyboard task can be very comfortably handled by the processor.

Before we can start identifying the modules which constitue the keyboard process, we can first look at the ***state-machine model*** of the keyboard task. The state-diagram shown on the following page indicates that at any given time the keyboard task is in any one of four states.

For each state we need to identify the following:

* What is the action performed in that state
* Which events cause a transition from this state
* To which state/states can transitions be made.



The state-machine approach permits us to break-up each task into small independent pieces of code as ***structured modules*** .(Where each module corresponds to *one state* of *one task).* Each task would run as a ***process.*** *(Remember that a process is a program in execution).* Depending upon the state in which the task is, the appropriate module for that task would be picked up for execution. (The finite-state machine approach also enables us to write extremely small pieces of code which could be tested to ensure that they run to completion within a reasonably small time.)

Last, but not least, we need a scheduler which will :

* Allocate processor time to different processes
* Provide a mechanism for communication between processes
* Perform the time-keeping function of processes.

To illustrate these concepts, we shall consider just two simple tasks: the *keyboard task* and the *sensor task* and see how these are implemented as state-machines and made to run concurrently in our system and how they interact with other tasks.

THE KEY-BOARD TASK:

The function of the key-board is all too familiar and is just a part of the man-machine interface of a system. In this application, the "keyboard" is only a set of command keys for the operator as well as a numeric keypad for entry of the machine parameters. Although we could have added a dedicated key-board encoder, we shall refrain from doing so and adopt the policy of *not adding any extra hardware if the processor has enough time to spare to do the job.*

For the purpose of our discussion we shall assume that we have a mechanism of scanning the keys of our operator-console through a few lines of our PIA: The keys are arranged on a typical matrix and the processor drives a logic-low on a given column and scans for a logic-low on a row. If no key has been pressed, the processor will read a high. However, if a key (let's say it is on Col #1, Row#2) has been depressed, the logic-low that has been driven on col#1 will be returned as a low on row#2 when the processor scans it. The processor then knows the co-ordinates of the key that has been pressed. So much for the mechanism of the keyboard scanning, let us now turn to the *real-time* aspects of this task.

With reference to the state-diagram of the keyboard, we note the following:

STATE 1: * Processor keeps scanning keyboard every T_{sc} msec
* If a low is detected on any scanned row, change state to 1 else remain in state 1

STATE 2: * Scan the same matrix point after T_{db} (the de-bounce interval)
* If the scan is still low, it is a valid transition so decode the key and change to state 3
* else the transition is rejected as noise so return to state 1

STATE 3: * Keep scanning the same key every T_{sc} msec.
* If the key is still depressed, stay in State 3
* If a scan is returned as high, change to state 4

STATE 4: * Scan the same key after T_{db} msec
* If the scan is high , the key has been released so return to state 1 else reject the transition as noise and stay in state 3

From the above description of the states, the following points are clear:

* Firstly, the keyboard process needs to be activated only once in several milliseconds.
* Depending upon the state, a unique piece of code is to be picked up for execution. This piece of code is extremely simple and can be tested for correctness without difficulty.

Before, we turn to another task and it's implemntation, let us look at the scheduler which would be used to run these tasks concurrently.

THE SCHEDULER AND IT'S DATA STRUCTURES:

At the heart of the system is a simple scheduler which schedules the various concurrent tasks. In this case, we have adopted the non-pre-emptive scheduling method. In this scheme when it is time for a given task to run, it runs upto completion and is not interrupted by other tasks.

Each task will determine the time duration after which it needs to be re-activated. This value of time (usually in milli-seconds, but sometimes also in seconds) is written into a timer location. A real-time interrupt will constantly decrement this timer value until it reaches a value of zero. The

scheduler maintains a list of timers, one for each tasks. The first task whose timer has expired will be picked up and it's process will be activated. Since each task has been modelled as a finite-state-machine, whenever the timer expires, the scheduler would first find out the current state of the task and then pick-up the appropriate state-routine for execution. The small pieces of code which constitute the state routines of the various tasks, are written so that they terminate quickly after just analysing the current inputs and detmining whether or not a state-transition has to be made and if so, the new state number.

The scheduler keeps track of various tasks, their state numbers and the timers with the help of the following data structures:

* A list of active tasks.
* A list of timers
* A list of states
* A list of jump-tables

The list of active tasks simply tell the scheduler which tasks are curently active and which have been suspended (or killed). It would then look up the list of timers to check if any of the timers have expired (i.e. become = 0) . Let us say that timer # *j* in the list of timers has expired. This indicates that it is now the turn of task # j to have it's process running. The scheduler then looks up the current state of task # j (say it is 'm') and then with this computes the offset in the jump table corresponding to task # j. This would then enable the scheduler to pass on the control to the appropriate state routine of the appropriate task. This is typically done by a call-to-subroutine instruction followed by a jump. After the state routine has been executed, control is returned to the scheduler by a return-from-subroutine type of instruction. The scheduler then continues to scan the list of timers to see if any task needs attention.

The basic time-keeping function of keeping time for all the tasks is performed by a real-time-interrupt which can occur every 1 ms or so. This interrupt simply decrements the timer of every active process. In processors where it is possible to wait-for-interrupt the scheduler may just wait for this tick of the clock before checking the list of timers. This is because, the timers are decremented only at every tick of the clock (i.e. every time a real-time-interrupt occurs) and thus no timer can expire in-between interrupts. In some applications it may be possible to go into a power-down mode between the ticks .

Recall that the keyboard task would run it's process for just a few 100 micro seconds and that the task would itself need to be activated once every

100 ms or so. This shows that the processor occupancy for this task is as low as 0.003 to 0.005 . In simpler terms, this means that this task is keeping the processor busy for only 0.3% to 0.5% of the time. Thus in applications where power consumption is critical, it is evident how useful it would be to power-down the processor between ticks and when no task needs attention.

A careful calculation of response times would enable one to squeeze the maximum processing power of the processor in the given application. Whether or not the processor can still take-on more concurrent tasks would ofcourse depend upon the (peak) processor occupancy and whether this would affect the response of the system to the existing tasks as well as the new ones that are to be added.

THE SENSOR TASK:

Consider yet another example of a task which runs concurrently with the keyboard task, the senosr task. This task keeps scanning the output of a proximity sensor which indicates that the carriage has reached it's extreme point and must reverse in direction. The output of the proximity sensor passes through some interface electronics from which the processor gets only a logic level: A logic HIGH indicates that the limit is NOT SENSED and a logic LOW indicates that the limit is SENSED.

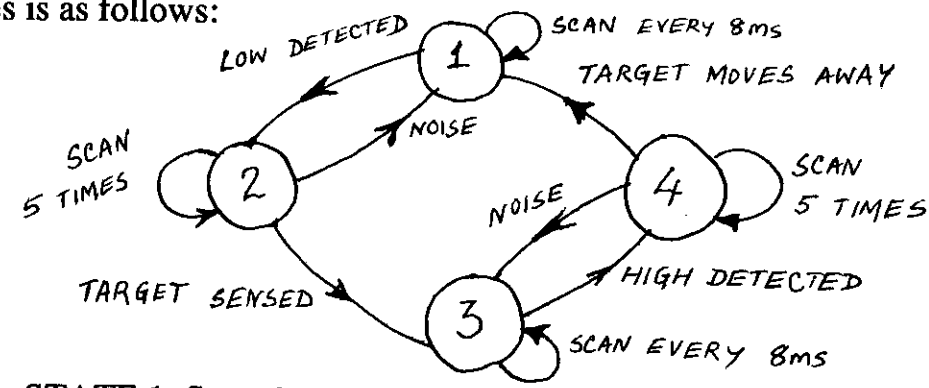
While this by itself is quite simple, we must bear in mind that the electrical signal travels a few metres through an environment which is electromagnetically speaking, quite polluted. Thus it is unwise to use the raw form of the signal as-such. In this case we shall perform a very elementary kind of filtering which invloves taking the majority value of an odd number of samples of the signal. This is done by taking about 5, 7 or 11 samples and then checking whether the signal was a LOW (or HIGH) for atleast 3,5 or 6 times. In this case we take 5 samples and take a 3-out-of-5 majority.

Before we describe the state-diagram of the sensor, we can once agin go through some simple arithmetic to determine if at all this task can be handled by the processor in software and if so, how often must this process run:

The sensor is designed to sense a portion of the carriage (called the target) which moves across the sensor at a speed of about 10 mts/min. The size of the target is such that when the target passes by at 10 mts/min, it will

take about 50 msec to completely cross the sensor. This implies that the output signal from the sensor will last for a minimum of 50 msec. During this window of 50 msec, we wish to accommodate atleast 6 samples: the first one to determine that there has been a change in state and the next 5 to take a majority count. This tells us that the sensor's output must be sampled atleast once every 7 or 8 m sec. Note that this time unlike the keyboard task, where the processor had to scan the whole matrix, here the processor has to scan just one single input line. The amount of computation is therefore extremely small and would last for 70 or 80 micro seconds. Since this task needs to run every 8 m sec even this task would have a processor occupancy of less than 0.01 or in other words, this task too would keep the processor busy for less than 1% of the time. In this manner it is clear how one can slowly add more concurrent tasks to utilise more of the processor's power.

This is implemented by the state diagram shown below. This is very similar to the state-diagram of the keyboard process and the description of the states is as follows:



STATE 1: Scan the sensor every 8 m sec. If the line is LOW, change to state 2 else remain in state 1

STATE 2: Scan the line every 8 m sec 5 times. If atleast 3 times the line has been LOW the target has been sensed so change to state 3 else change to state 1

STATE 3: Keep scanning the line every 8 msec. If a HIGH is sensed then change to state 4 else remain in state 3

STATE 4: Take a sample every 8 m sec 5 times. If the sensor output was HIGH atleast 3 times, then the target has moved away so chnge state to 1 else return to state 3

INTER-PROCESS COMMUNICATION:

The processes of different tasks would need to exchange information such as for example: the keyboard task must inform the other relevant tasks

which key has been pressed. Similarly the sensor task must inform the o9ther tasks that the carriage has reached a limiting position. This kind of information can be interchanged in a variety of ways. A simple alternative is to use a ***circular buffer***. Here there are two distinct pointers for every buffer: a write pointer and a read pointer. The task which *produces* the data uses the *write pointer* and the task which *consumes* the data uses the *read pointer*. A special character is written by the producer to mark the point where there has been a last entry.Whenever the consumer task reads this special character, it knows that it has consumed everything that has been produced . Since we do not employ pre-emptive scheduling conflicts between two tasks using the same data will not arise and semaphores are not essential.