Using Communicating State Machines to Design an Interrupt Driven Task Schedular

Bill Stoddart

September 11, 2001

Abstract

A number of Forth systems have been equipped with a minimalistic but extremely effective task schedular. Design methods and scheduling theory seem to inevitably lead to cumbersome and far less elegant solutions. In this paper we present a mathematical model of the classical Forth multi-tasker in the form of communicating state machines. The aim is to suggest that such a technique can lead to the sort of elegant and minimalistic design which is typical of Forth, whilst also providing the possibility of formal validation. Our model includes the multi-tasker itself along with abstract protocol schemas for the behaviour of tasks and hardware devices, needed to specify the correct behaviour of these with respect to interrupts.

Key words: Forth, Communicating State Machines, Event Based Models.

Introduction

We consider the design of a task switching mechanism for a microprocessor system which needs to respond to events in its environment. The system is interfaced to its environment by a number of i/o devices which issue interrupts when they detect that an event has occurred. Each i/o device is managed by a separate task. E.g. a particular task might manage the input from a temperature sensor, receiving an interrupt when a changed reading occurs. The function of the interrupt routine is to signal that there is some work for the task which manages the device. When the task is subsequently scheduled the temperature is read and any necessary action taken. Of course, in our model we avoid this kind of specific detail, and just model the general scenario of interrupt routines which provoke the subsequent scheduling of the associated task.

The word "design" is used here in the sense of "to indicate, to draw, to form a plan of" (Chambers 20th C Dictionary) rather than with its alternative connotation of "to contrive" (ibid.). The discovery of the beautiful mechanism described here is entirely due to Charles Moore.

Modelling events associated with the task schedular

The state machines we use to build our models change state on the occurrence of an "event". We think of events as instantaneous (so to model somethong with duration we would need start and finish events).

The tasks run on a single processor and are dispatched by a round robin scheduler. The system can run any number of tasks (subject to memory and performance constraints) but we can illustrate its operation with two tasks A and B.

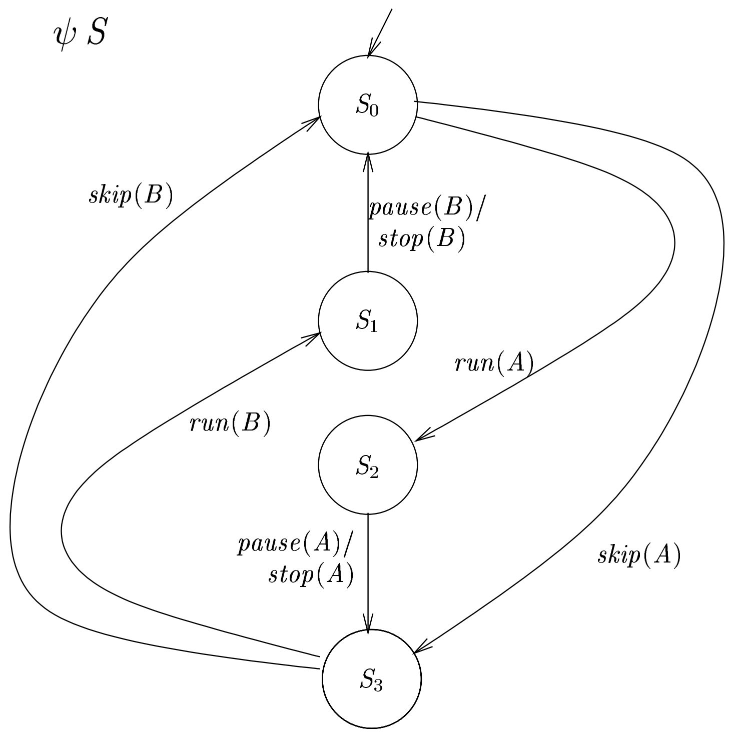

Figure 1: Task Scheduler running tasks A and B

Figure 1 models the operation of the scheduler. From its initial state S_0 it may either run task A or skip task A. If it skips A, it becomes ready to run or skip B. If task A is run, it must subsequently either *pause* or *stop* to allow scheduling of the next task.

This view of the schedular leaves out important details, such as what determines whether a task should run or be skipped. We add these by introducing additional state machines. Each task is modelled in terms of a status flag, an i/o device interrupt protocol and an execution protocol. We detail those for task A.

Modelling the effect of the task status flag

The decision as to whether a task will be run or skipped by the scheduler is made according to the setting of the task's status flag. The status flag for A is shown in figure 2.

In its initial state SA_0 the status flag permits the event $skip(A)$ but blocks $run(A)$. In state SA_1 the status flag has the opposite effect, blocking $skip(A)$

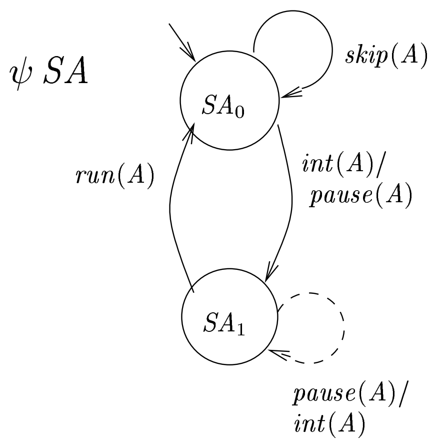

Figure 2: Task A status flag

and allowing $run(A)$. It can change from SA_0 to SA_1 by the occurrence of either an interrupt for task A (event $int(A)$) or by a pause in the execution of task A (event $pause(A)$). When task A is run, (event $run(A)$) its status flag is reset to state SA_0.

Shared events

Communication between state machines is via shared events. The rule is as follows: where two or more state machines have the same event in their repertoire, that event can only occur when all such machines are ready to engage in it. So for $skip(A)$ to occur we need the schedular (machine S) to be in state S_0 and the status flag for A (machine SA) to be in state SA_0. The effect of the event $skip(A)$ will be to take machine S to state S_3 and to leave machine SA in state SA_0.

A state machine which could, at some point, engage in some event but is not currently in a state where it can do so at the moment, will block that event in the sense that it will certainly prevent it being the next event to occur in the system. We have to be very carefull when using this modelling technique not to give the such a power of veto to a component which could not realistically exercise it. For example our task status flag does not have the power to block the occurrrence of an interrupt. On the other hand it does have the power to block either the running or the skipping of its associated task. We will return to this point later on.

From state SA_1 the status flag can engage in either $pause(A)$ or $int(A)$ but will remain in the same state. This pair of transitions is represented with a dotted line, which represents an expectation that these transitions are not expected to occur within the context of the overall system, but must be included here because it is not the job of the status flag to block them, and that would be the effect of their omission under our interpretation of shared events.

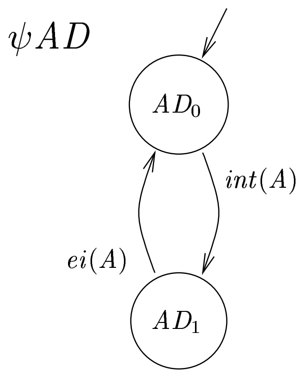

Figure 3: The i/o device serviced by task A

The interrupt protocol for i/o devices

Figure 3 defines the interrupt protocol to be followed by task A's i/o device. After the event $int(A)$ a further $int(A)$ is impossible until interrupts are enabled for the device (event $ei(A)$). On the other hand there is no restriction placed on the interrupts from other devices (since, for example, we interpret $int(B)$ as a completely different event from $int(A)$).

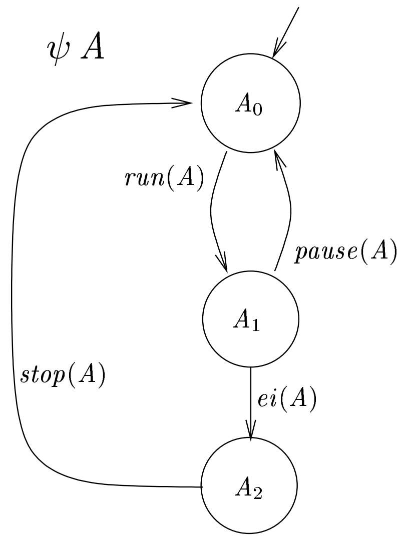

Figure 4: Execution schemata for task A

The execution protocol for tasks

Figure 4 defines the execution protocol of task A. From its initial non-executing state A_0 the task may be run (event $run(A)$. It returns to the non-executing state either via a pause (when it has more work still to do but other tasks are to have access to the processor) or by re-enabling the interrupts for its associated i/o device and stopping (when it has completed its work).

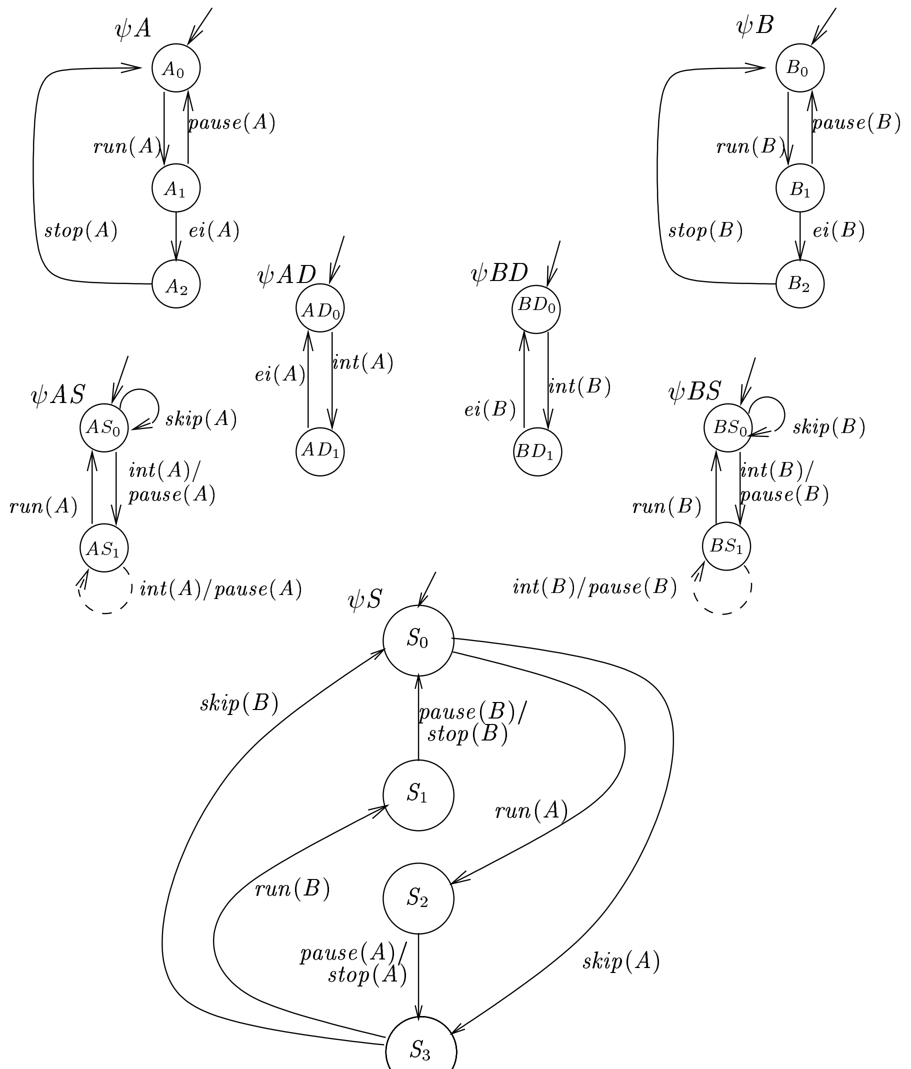

Figure 5: Tasks A and B with status flags, i/o devices and scheduler

The complete model and example traces

The complete model is shown in figure 5. The following example trace, in which a single interrupt from B's i/o device is handled, will illustrate its operation under conditions of light loading:

Event	Involving	State
		$\{A_0, AS_0, AD_0, S_0, B_0, BS_0, BD_0\}$
$skip(A)$	S, AS	$\{A_0, AS_0, AD_0, S_3, B_0, BS_0, BD_0\}$
$skip(B)$	S, BS	$\{A_0, AS_0, AD_0, S_0, B_0, BS_0, BD_0\}$
$int(B)$	BS, BD	$\{A_0, AS_0, AD_0, S_0, B_0, BS_1, BD_1\}$
$skip(A)$	S, AS	$\{A_0, AS_0, AD_0, S_3, B_0, BS_1, BD_1\}$
$run(B)$	S, B, BS	$\{A_0, AS_0, AD_0, S_1, B_1, BS_0, BD_1\}$
$ei(B)$	B, BD	$\{A_0, AS_0, AD_0, S_1, B_2, BS_0, BD_0\}$
$stop(B)$	S, B	$\{A_0, AS_0, AD_0, S_0, B_0, BS_0, BD_0\}$

The design becomes critical under conditions of heavy load, i.e. when interrupts occur more or less as soon as they are enabled. In this case an interrupt may arrive to activate a task which is still running in response to a previous interrupt. A key design decision here is that when a task is dispatched its status flag is reset to its initial state, allowing its interrupt routine to register an effect whilst the task is actually running. The following trace gives an illustration:

Event	Involving	State
		$\{A_0, AS_0, AD_0, S_0, B_0, BS_0, BD_0\}$
$skip(A)$	S, AS	$\{A_0, AS_0, AD_0, S_3, B_0, BS_0, BD_0\}$
$int(A)$	AS, AD	$\{A_0, AS_1, AD_1, S_3, B_0, BS_0, BD_0\}$
$int(B)$	BS, BD	$\{A_0, AS_1, AD_1, S_3, B_0, BS_1, BD_1\}$
$run(B)$	S, B, BS	$\{A_0, AS_1, AD_1, S_1, B_1, BS_0, BD_1\}$
$ei(B)$	B, BD	$\{A_0, AS_1, AD_1, S_1, B_2, BS_0, BD_0\}$
$int(B)$	BS, BD	$\{A_0, AS_1, AD_1, S_1, B_2, BS_1, BD_1\}$
$stop(B)$	S, B	$\{A_0, AS_1, AD_1, S_0, B_0, BS_1, BD_1\}$
$run(A)$	S, A, AS	$\{A_1, AS_0, AD_1, S_2, B_0, BS_1, BD_1\}$
$ei(A)$	A, AD	$\{A_2, AS_0, AD_0, S_0, B_0, BS_1, BD_1\}$
$int(A)$	AS, AD	$\{A_2, AS_1, AD_1, S_0, B_0, BS_1, BD_1\}$

At this point task A is running and action in response to interrupts is pending for both tasks. Under the assumption that no further interrupts occur and that tasks relinquish the processor via *stop*, the following event trace will subsequently occur:

$$\langle stop(A), run(B), ei(B), stop(B), run(A), ei(a), stop(A)\rangle$$

Now let us return to the unwanted event transitions of machines SA and SB, which are marked by dotted lines in our diagrams, and consider exactly why we do not want them to occur but must nevertheless include them in the behaviour of our machines. For example consider the transition:

$$SA_1 \xrightarrow{int(A)} SA_1$$

this is unwanted since it fails to register that an interrupt has occurred. It must be included because it cannot be the business of a status flag, which will be implemented as a program variable, to control whether an interrupt routine can occur. On the other hand it can very well be used, in an *if* statement, to control a choice, made by the scheduler, between running or skipping a task, and this is exactly its function in our model.

Conclusions

So what can we claim to achieved by having built an "Event Calculus" model? Firstly we hope we have expressed some design ideas in a form which is both formal and, we hope, accessible to a wide range of engineers who are not specialists in Formal Methods. We have also detailed the protocols required of interrupt routines and task executions: e.g. a task must enable the interrupts of its associated device before it does a *stop*, but must not enable interrupts before a *pause*. If we are going to implement a non-preemptive scheduler (in which *stop* and *pause* will be implemented as function calls made from the level of user code) the level of design detail in our model is not far from that required for coding. Note however, that at the level of abstraction used here no actual decision has been taken as to whether per-emptive or non-pre-emptive scheduling is to be used.

A final point is that when moving from design to implementation we will not seek to "implement" the individual state machines as such. They were just views relating to different aspects of the model. In other cases where the same technique is used however, our state machines will represent separate physical or software entities, possibly encapsulating data and communicating values as well as engaging in primitive synchronisations. In such cases implementation of a model may well be based on the state machine representations of its component parts.