ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA012

INTERRUPTING A STATE MACHINE

K. V. L. Baker, ISIS Neutron and Muon Source, Didcot, UK

Abstract

At the ISIS Pulsed Neutron and Muon Source [1] we interact with a variety of types of beamline systems for controlling the environment of samples under investigation. A state machine is an excellent way of controlling a system which has a finite number of states, a predetermined set of transitions, and known events for initiating a transition. But what happens when you want to interrupt that flow? An excellent example of this kind of system could be a field ramp for a magnet, this will start in a "stable" state, the "ramp to target field" event will occur, and it will transition into a state of "ramping". When the field is at the target value, it returns to a "stable" state. Depending on the ramp rate and difference between the current field and the target field this process could take a long time. If you put the wrong field value in, or something else happens external to the state machine, you may want to pause or abort the system whilst it is running you will want to interrupt the flow through the states. This paper will detail a solution for such an interruptible system within the EPICS [2] framework.

WHAT IS A STATE MACHINE?

A state machine can be defined in an electronics context as “a device which can be in one of a set number of stable conditions depending on its previous condition and on the present values of its inputs” [3]. An event is something that happens, it may be a timeout or value is reached, or it may be that a button is pressed. This event will in turn trigger a transition from one state to another – there is a change in the inputs for the state machine. Most practicable state machines are finite state machines – that is they have a finite number of states and transitions. Infinite state machines are plausible, but impractical, as such the use of the term state machine will typically refer to a finite state machine.

One of the simplest examples of a state machine is something which can be on or off, such as a light. Most light circuits will have a switch of some variety, typically one that is open or closed. If the switch is open the light is off. Pressing the switch to closed is an event which will trigger the turn light on transition, and the light will then settle into the on state. Figure 1 shows a simple state diagram for the on/off state of the light.

The complexity is already apparent, for example: if there is no power then the light cannot be on, if the lamp has broken then it can be off in the on state, if the switch develops a fault then the state might not be alterable from the present one, and so on. The more complex the system, the longer the list of states that can occur.

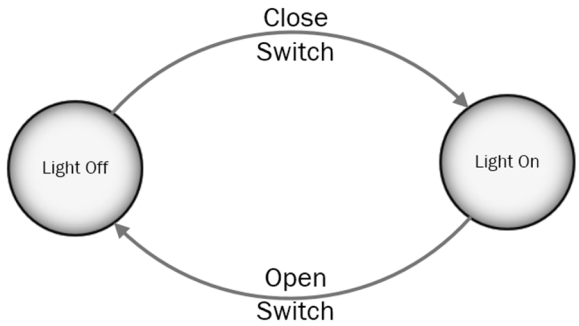

Figure 1: Simple light state diagram.

This electronics concept is also applicable to programming, especially of systems which interact. If, rather than a light, the button was to switch on a large scale lighting system that can only be run at night which requires water cooling. To allow the on state to be achieved the switch must now be closed, the water flow must be of a suitable rate and we need to know that the present time is between sunset and dawn. Whilst this set of criteria can be fed into a circuit and considered at an electronics level, this example is well suited to a software solution. It is straightforward to determine sunset and dawn by looking it up from a trusted online source, and this can then be defined as an appropriate action to consider in the state machine.

WHAT IS AN INTERRUPT ?

Typically an interrupt can be defined as a break or stop of something continuous [4]. This is true in an electronics or computing context as well, where an interrupt is often a signal used to break the flow of code through a processor. This can then be thought of as a specific type of event, one which might alter the flow more rapidly than waiting for a state to complete. An interrupt is in essence an input which is treated in a prioritised fashion.

If we take the complex light situation above, then the clock ticking over so that dawn has occurred will immediately mean a need to switch the light off. But, where is that time being checked?

Each loop of the on state could go to that external source to get the time of dawn and sunset for the current date, and check the present time to see if it is night or not, then trigger a transition to the off state.

A better programming solution would be for the on state to go and get the dawn and sunset times each day and store them in variables to compare, so it doesn’t need to keep spending time getting that data as well as checking on everything else.

An ideal programming solution would be to have a separate process collect the dawn and sunset times once a day,

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA012

keep comparing the present time against those stored values, and provide a simple state of "day" or "night" to the on state to use to trigger appropriate transitions.

QUEUED STATE MACHINE OVERVIEW

A Queued State Machine (QSM) is a design pattern. Within software engineering a design pattern is a language independent reusable template that can be used to define how the code should be structured. The QSM is a common pattern used in LabVIEW [5], using a combination of design patterns to create the desired behaviour [6].

The command, action queue or event queue design pattern, and similarly the event-driven programming paradigm used in other languages provide something similar to a queued state machine. The queue will typically have a first in, first out (FIFO) array that will take a list of actions to undertake, or events to process. The actions or events will then be dequeued (removed from the queue) and actioned in the same order that they were enqueued (added to the queue).

For a queued state machine, you queue the list of states. A state differs from an action by being a stable condition until a transition occurs, whilst an action is an event that occurs and is then forgotten. This continuation of action, such as ramping a value, provides a "ramping" state.

USE CASES AT ISIS

There are a number of places where state machines can come in handy for control systems, and one of the key ones that has been considered recently at ISIS is that of magnets. From a beamline controls point of view, these will typically consist of a power supply unit (PSU) at the very least. The PSU will begin in a "stable" state, where the output and target are the same value, there is nothing changing. If a new target is requested, then the system will enter a ramping state. In this ramping state the output is changing at a defined rate. Once the output and target are the same value then the system can return to the stable state.

The issue is that whilst ramping things can change, and waiting for it to reach the target might not be practical, or potentially possible. If you are ramping from 0 Amps to 100 Amps at 0.0001 Amps/second then it will take more than 11 days. If you actually wanted to go to 1 Amp, which would take about 3 hours, then there is a big difference in elapsed time due to the mistake. The 22 days it would take to undertake that first ramp and then get back to the actually desired 1 Amp could be used in a much better way than waiting for ramps to finish.

Aborting that ramp would limit the delay. This means the ramping state now needs to consider the following events: are target and output the same; and whether or not an abort has been requested. If an abort was requested there may be some other tasks to undertake, such as setting the target to the present output. You could monitor the abort in a separate process, as with the time of day previously described, and simply set the target to the present output when the abort is selected, which should transition the state from ramping to stable.

However, a pause scenario might require something different again. If you pause you want to be able to resume from where you are when you requested the pause to your initial target, so you need to transition from ramping to paused. From paused, a resume will take you to back to ramping, and an abort will take you to stable, setting the target to the present output. Figure 2 shows a flow chart for the ramping state to decide on the transition to use.

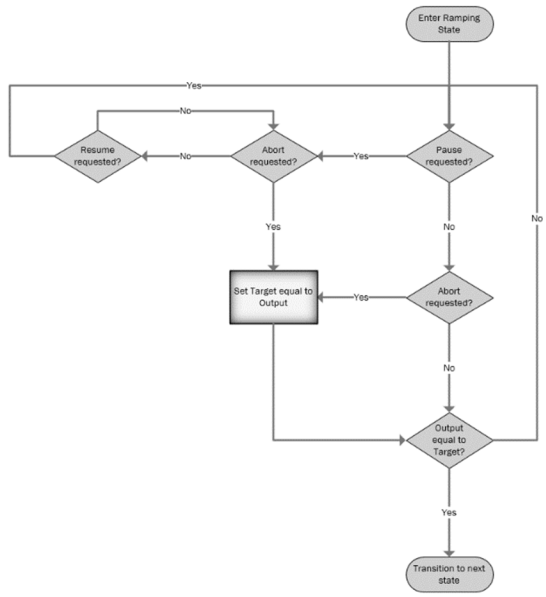

Figure 2: Flow chart for a ramping state.

This starts to lead to obvious functions, for example an abort function, which can be called by different transitions. The system can be complicated further by various modes of operation. A magnet may be superconducting, and an abort may require better handling of switch temperatures and the current in leads that just setting the output and target to the same values.

If the ramp you are using goes from one side of zero to the other (for example, negative to positive), then depending on the PSU it may mean that the ramp has to go to zero, change the direction of the magnet, and then continue. This transition logic could be in the hands of the firmware or the control system. If it is the former, then the control system doesn't need to be concerned about anything, and can just send ramp targets as it wishes. If the firmware doesn't handle a transition to zero itself, then there are a number of states that can be dictated at the start of the ramp, namely "ramp to 0", "change direction", and "ramp to target". With a superconductor having reached target then the leads to the switch may also need to be ramped, adding to the complexity of the states and transitions.

With the number of interrupts, events to monitor, and transition points to consider each state is repeating quite a bit of code. For example, whilst ramping towards 0, we are continuously asking, are we at 0, has an abort been requested, has a pause been requested, has another interrupt occurred? Upon reaching 0 we have to know have we just ramped down the leads for a superconductor, or are we

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA012

about to change the direction and go to a new value, or is the system at the requested target? This next step then needs to know the history and a lot more about the state of the system.

This is where a design pattern, such as a queued state machine, can come in handy.

QUEUED STATE MACHINE IN STATE NOTATION LANGUAGE

At ISIS our control system for beamlines, IBEX, is based on EPICS. Because state machines are useful there is a system designed for use within EPICS to programme state machines, the State Notation Language and Sequencer [7]. An SNL programme is constructed of a number of concurrently running finite state machines called state sets. It has built-in functions for interacting with EPICS. It is possible to call C code directly from states, and the syntax is very similar to C. Which state in a state set to run is decided by the when clause. If the logic within a when clause evaluates to True that state is enacted. This description allows a slightly different, and useful language to be employed for this discussion. Rather than considering a finite state machine, a state set can be considered, with a when clause being the trigger to run a state, rather than considering triggers to exit states.

The queued state machine can be considered as a number of state sets interleaving with each other. There is the main state set that runs the states for what is being controlled, the when clause driven by a single variable, a state set monitoring the interrupts, and a state set managing the queue.

The state set monitoring the interrupts has to be a tight loop. After startup, it does nothing more than monitor those values, and do the least amount of work possible to make the other state sets change their behaviours. The queue state set does nothing more than manage the queue, and the main state set has all the logic and does all the work.

Abstracted Example

Let us consider a fairly simple abstracted system. The system can be started, paused, resumed, and aborted.

The system can be run in Mode A, where a start requires going through states 1, 2 and 3; Mode B, where a start necessitates states 2, 4, and 1 to be run in that order; or Mode C which runs states 1, 3, 4 and 2. Any pause or abort has to be acted on quickly. Whilst this is a simplified abstraction, it should be possible to see how the queue could be helpful in a more complex system without too much imagination.

The main state set in this abstracted example would consist of states for being idle, start, pause, abort, state 1, state 2, state 3, and state 4. As we are dealing with an abstraction we can assume that the states handle themselves, in reality there may be more interaction between them and they could need to respond to certain interrupts as well.

The idle state will be just that, idle, there will be no actions undertaken here. Start will look at the mode of the system, and queue up the states to pass through. The pause state will pay attention to whether or not the system has been started, and on a second request, or resume, it will allow the system to continue as if there had been no interruption. Abort will clear the queue of states, and return the system to an idle state. This state set would be controlled by a single variable to represent each state, an enum would be ideal, and the state to run will be whatever is contained in that variable.

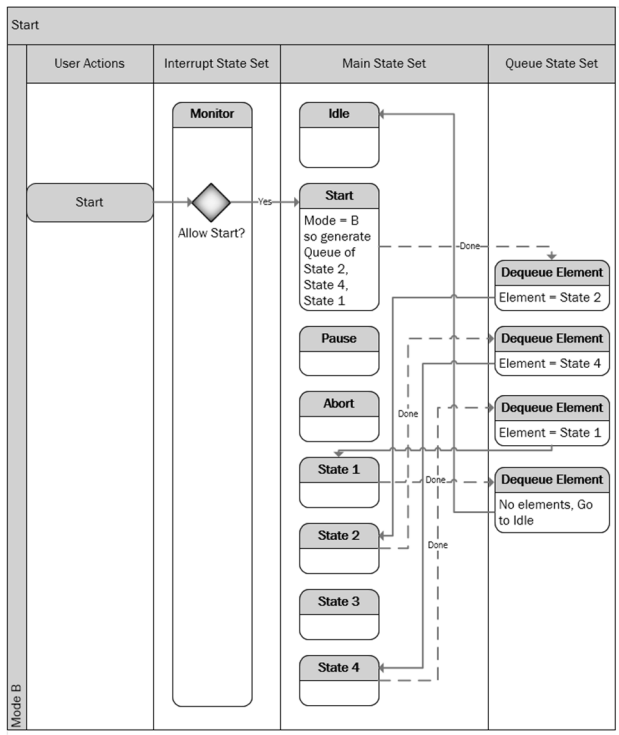

Figure 3: Start in mode B of QSM.

The interrupt state set monitors the start, pause/resume and abort requests. Should any of these change, then the associated main state is triggered, simply by setting the driving variable, and this state set quickly goes back to looking at the interrupting values. Should the interrupt be invalid, then this state set can issue the warning and ignore it, without delaying the main code to deal with the situation too much.

The queue state set will receive a list of states to go through from the start state, and as each state ends, will action the next state in the queue by setting the driving variable for the main state set, pausing and resuming as appropriate.

Figure 3 and Figure 4 illustrate some examples of the flows detailed above.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA012

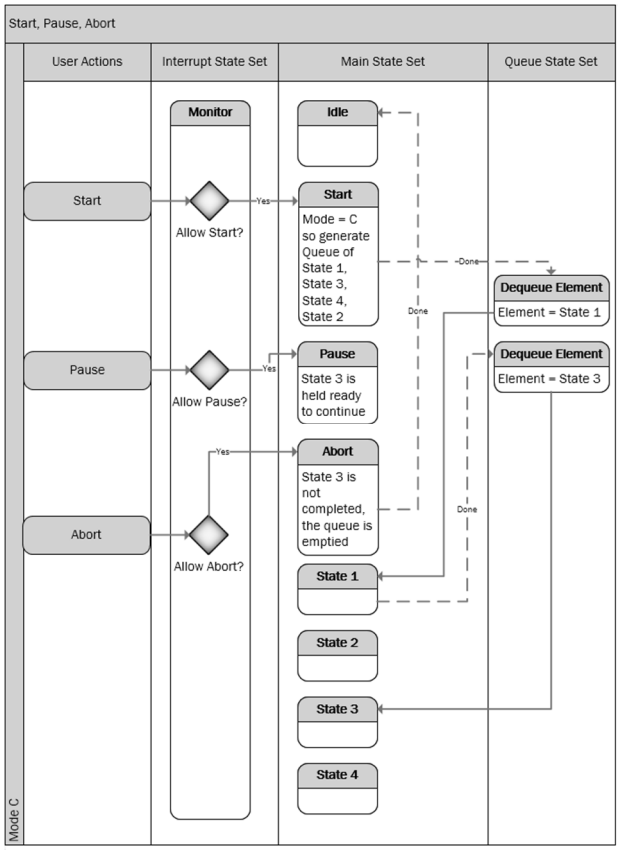

Figure 4: Start, pause, abort in mode C of QSM.

CONCLUSION

If you consider a ramp, the clauses which dictate the next state, or the state to run, become complicated. For example whilst ramping you have to check whether you are at your target, whether pause has been pressed, or whether an abort has been requested. Three checks aren't too bad, but if this were a ramp for a cryomagnet then consideration needs to be paid to whether or not the system is quenched, another trip has been seen, the temperature of the magnet, etc. If you then want to control your ramp rate as you ramp, going to specific targets, or even just changing direction at zero, you then have to start considering the end of a ramp step versus reaching your target, increasing the complexity further for your states and transitions. As such, running a system which can be interrupted allows for the complexity, without sacrificing readability or responsiveness.

A state machine is a powerful technique for keeping track of what a system is doing, whilst a queue allows you to order and re-order those states, whilst being able to interrupt them provides a more responsive system.

REFERENCES

[1] The ISIS Pulsed Neutron and Muon Source, http://www.isis.stfc.ac.uk/

[2] EPICS https://epics-controls.org/

[3] State Machine Definition, https://www.lexico.com/en/definition/state_machine

[4] Interrupt Definition, https://www.lexico.com/en/definition/interrupt

[5] LabVIEW, https://www.ni.com/en-gb/shop/labview.html

[6] LabVIEW, design patterns https://labviewwiki.org/wiki/Design_Patterns_Overview

[7] State Notation Language, https://www-csr.bessy.de/control/SoftDist/sequencer/