

State of mind

State machine design; as easy as telling the time. By **Anders Holmberg**.

State machines are often used to solve certain kinds of problems where the control flow can be pictured as moving through a set of distinct states. There are a number of different definitions and practical descriptions of what a state machine is. At a theoretical level, a state machine is a kind of automata that responds to some kind of stimuli by changing state.

Texas Instruments has recently introduced the eZ430 Chronos development kit, which is based on a traditional sports watch design. This features pulse monitoring, a three axis accelerometer and a pressure sensor, all enclosed in an attractive housing.

The watch has a number of different modes, or states, where the events mentioned above can take on a different meaning. For example:

- Stop watch
- Set time and date
- Activated radio mode for heart rate monitoring
- Wake up alarm
- Different display modes
- Exercise energy estimation

We can quickly list a number of abstract events that can be interesting to an application built on this platform.

- *Button presses.* The watch has five buttons and the original firmware reacts to all of these. It also assigns meaning to a long [2s] press on two of these buttons and can also detect and generate repeating or continuous button pressing that generates a series of button events.

- *Timer events.* A number of timers can be running and generate periodic events. A stopwatch, for example, probably needs accurate timer events at a rather high speed to keep the display updated. Heart rate is also polled periodically, at least on the application level. The expiry of user defined alarms or the crossing of user defined thresholds – for example, heart rate – can be viewed as abstract events that should

have an impact on the application's behaviour.

An application like this, using a relatively resource constrained mcu, is often based on the 'fat interrupt routine' idiom, where most of the application logic is spread out over a number of interrupt functions. For example, the main timer interrupt function for the eZ430 Chronos original firmware is close to 180 lines, including single line comments and blank lines; the function drives quite a lot of the application functionality by way of function calls and manipulation of global data.

This way of working has a lot of advantages. It is, for example, easy in this case to control the different low power modes of the MSP430 mcu, because the only thing the main loop has to take care of is to do some low priority processing and then go back to the appropriate low power mode.

However, a potential drawback is that application logic is spread over a number of different modules and it can thus be difficult to trace how different functionality interacts during a debug session or when new functionality is to be added.

A state of mind

Another way of breaking down this type of application is to organise it around a main loop, where most of the high level logic is taken care of, and to simplify the interrupt routines as far as possible to only detect and report events to the application. In a state/event oriented application, a state machine approach can then be used to

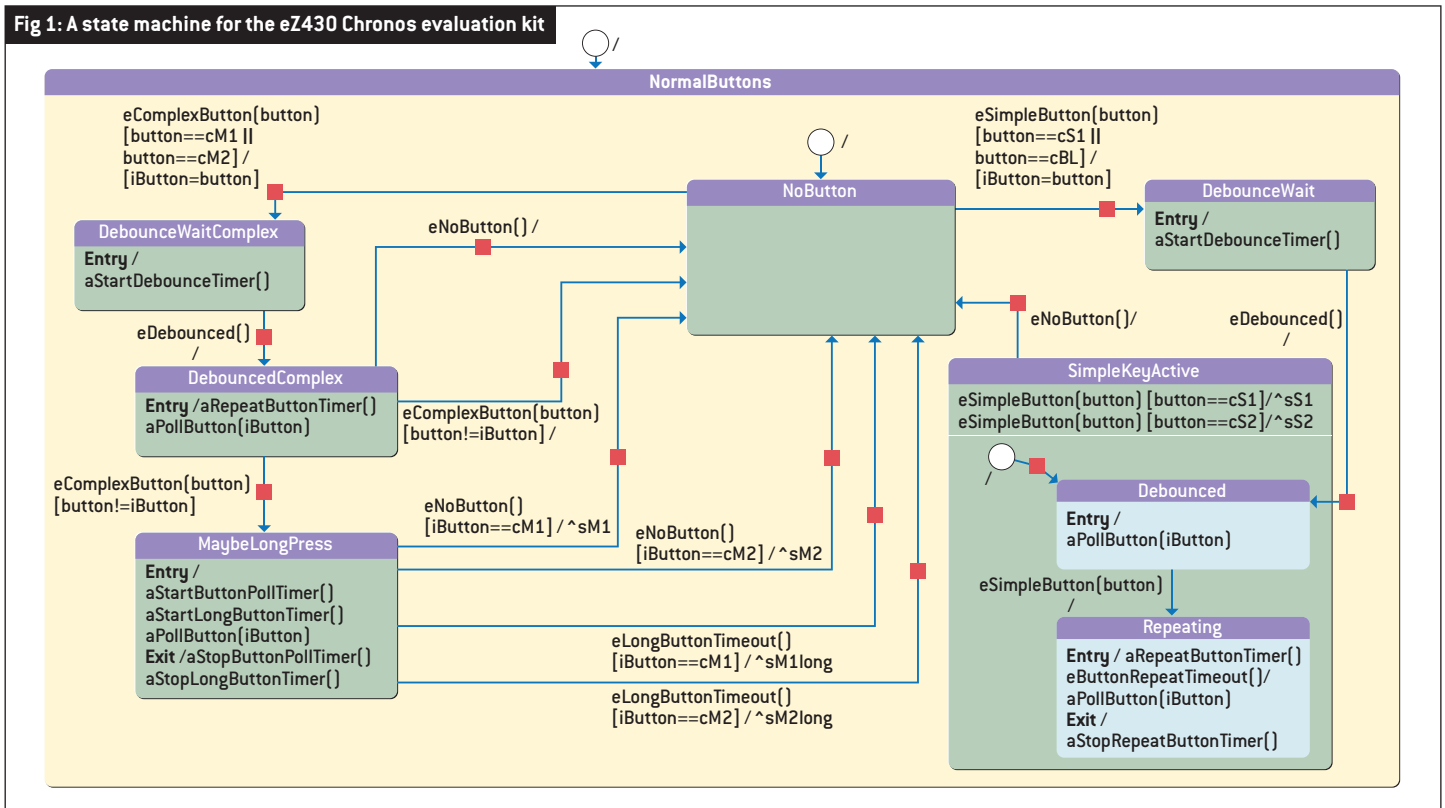


design and implement the logic.

The advantage is that it is possible to achieve a clean separation between input device drivers, application logic and output device drivers. The code for input detection and output handling could then easily be reused in different projects on the same hardware without the need to clean the code to remove application specific knowledge.

The drawback is the book keeping needed for a centralised state machine can become quite unwieldy as application complexity grows. However, help can be found by adopting a tool assisted state machine approach. Tools such as visualSTATE can help you develop an event driven state based application and bring a

Fig 1: A state machine for the eZ430 Chronos evaluation kit



number of benefits across the development process.

- *Increased productivity.* There is a learning curve but, in the end, productivity will go up as you spend less time on the book keeping in your code and focus on the functionality.
- *Increased quality.* The use of state machines is regarded as a semi formal method and, by treating a state/event oriented problem as a state machine problem at the design stage, you can reap the benefits of working in the more formalised setting presented by well defined state machine semantics.
- *A clean separation between the input/output part of your application and the application logic.* This simplifies application architecture and promotes reuse across different hardware platforms.

Example

As an example, we can look at a small model designed for the ez430 Chronos that takes care of button debouncing and deciding if a button press is a short or long press. Here, the state machine sends signals to the application part of the state machine to indicate what kind of button press it has just decoded.

The state *NoButton* is the start state, as indicated by the small circle in figure 1 (the initial state) and accompanying transition to the state.

In a real world application, performing the debouncing logic inside the state machine may not be the best, or even the obvious, choice. It nevertheless serves as an illustration of the power of hierarchical state machines.

In the original application code for the watch, this functionality is spread over a number of modules and the main logic for the decoding and debouncing is a complex switch statement spread over some 200 lines of code. While it contains high level application logic, that only serves to underline the point about complexity.

The debouncing state machine can react to three events – *eComplexButton*, *eSimpleButton* and *eNoButton*. The interrupt routine that reacts to button events simply determines if the interrupt is from one of the buttons that does not have a meaning assigned to a long press. In that case, an *eSimpleButton* event is generated. An *eComplexButton* event is generated for the two buttons that have different meaning for short and long presses. The state machine can, in turn, use a helper function *aPollButton()* to

determine if a certain button is still active. This function can, in addition to the other button events, generate the *eNoButton* event to indicate the recently pressed button is no longer active.

The state machine, like the original debouncing code, also uses a set of timers to control debouncing and to determine the length of button presses. When the state machine has detected one of the various button presses, it reports it to the rest of the state machine (not shown in Fig 1) by sending a signal – the notation can for example be seen on the transitions from state *MaybeLongPress* to *NoButton* ($\wedge sM1long$). The semantics of signals is that the signal is processed immediately after the state machine has processed the event that led to the generation of the signal.

Author profile:

Anders Holmberg is visualSTATE product manager for IAR Systems (www.iar.com).

ne For further information on any subject visit: www.newelectronics.co.uk