COMMUNICATIONS

Data Communication Between “Smart” Wearables

Software Signals Explained

Build a Frequency Counter

Control Theory for DIYers

PLUS
Vehicle Fuel-Use Monitor
  // Data Collection and Diagnostics
  // Real-Time Miles Per Gallon Data
  // Test Run Results
  // And More
UML State Machines

You can use Unified Modeling Language (UML), which is a standardized software specification language, to represent state machines. This article provides an understanding of state machines—both basic and enhanced—and how you can implement them in your software.

Once upon a time, I used to be a software developer. This confession may be hard for my regular readers to accept, but I was not always a hardware guy. Even as I plead guilty, I have an excuse: Back then, I was mainly involved in embedded system development, so I was not so far from the soldering iron.

When I started to work, it was the Stone Age—object-oriented programming and gigahertz processors had not yet been invented. Nevertheless, some of the good programming practices I learned are still current. One of my favorites is the state machine concept. There is simply no better way to go from specifications to a working piece of code!

Although I still use state machines in numerous projects, I didn’t plan to write an article about them until I read Miro Samek’s Practical UML Statecharts in C/C++. It was a good refresher. In this book, Samek, founder of Quantum Leaps, expertly explains state machines’ different variations and provides an efficient and complete implementation of its Quantum Platform (QP) framework.

This article’s focus is on basic and enhanced state machines. I’ll explain how to represent them in the ubiquitous Unified Modeling Language (UML) notation and, more importantly, how to code them. I will not use QP (which is open source but under license for commercial projects), but I will present a simpler, personal version based on the same concept.

HAVE A DRINK?

Most concepts are easier to understand with an example. Imagine a customer asked you to develop the firmware for a beverage dispenser system. As I must respect my French citizenship, let’s say it is a Cognac dispenser. The project’s goal is to help drinkers limit their intake, so they must enter a code on a keyboard to fill their glass (see Figure 1). I admit it’s a silly example, but it’s just an example.

To forbid access to drunken users, there are some specifications. The user must first press the Activate button then enter the code, which can include either digits or letters (with a specific button on the keypad to switch between numeric and alphabetic mode) then press the Enter button. There is also a Delete button to correct the last keyed digit. Finally, the correct code must be entered in less than 10 s or the full sequence must be restarted. A reasonably drunk user will probably fail this test. The system then activates the Cognac pump for 2 s and locks the dispensing for 30 s before accepting another request.

Figure 1—My example application is a beverage-dispensing machine that uses a small keypad (simulated by my PC’s numeric keypad) to enter an authorization code.
This seems like a simple project, but the specifications are not 100% clear. For example, how should the dispenser react if a wrong key sequence is entered (e.g., pressing Delete before any digit or pressing Activate twice)? Or, what if the user tries reentering a code while the system is dispensing? Clearly, the first step in such a project should be to transform the customer's expectations into a clear and complete specification. But how?

**UML STATE CHARTS**

This is where UML shines. UML is a standardized graphical method to rigorously specify and model any software or process. I'm not an UML expert, so I'm not going to provide a full description, especially as UML is actually a toolset with plenty of different modeling charts. However, UML includes a powerful state machine representation called UML state charts.

I used the GNOME Project's Dia open-source software to draw these diagrams. Figure 2 shows the basic version. Here is the underlying model: At any time step, the software is in a given state, represented by a rounded-corner box. The initial state (i.e., the state at which the system is at reset) is pointed to by a circle-ended arrow called the initial transition. In a given state, the software is awaiting a certain number of events, represented by arrows called transitions. When one of these events happens, the software may execute some actions and move to another state. It could also stay in the same state in the case of "local transitions." Lastly, each state can have some actions executed each time the state is entered or exited.

This may seem complicated, so an example may help. Figure 3 shows a UML representation of a simple state machine in charge of the beverage dispenser's pump control.

![Figure 2](image28x93_to_566x814)

**Figure 2**—The UML state chart is a graphical representation of a system that has different states and event-triggered transitions from state to state.

![Figure 3](image28x93_to_566x814)

**Figure 3**—The pump management's UML state chart includes only two states.

The pump management is very simple. A guard condition is a logical condition associated with a transition. The transition is taken when the corresponding event is triggered, but only if the guard condition is true. This enables you to add "if then else" structures in a state chart.

There is one more enhancement provided by UML state charts. In addition to such standard state machines, UML supports hierarchical state machines (HSMS). What does this mean? Some states can be grouped in a "super state," which can be reactive to events (Figure 4). Moreover, super states can also be grouped in larger super states in a hierarchical manner. They are used so several states can react the same way to some events.

For example, imagine you have a complex system with lots of states. If a motor driver detects an overcurrent, you may want to switch power off. In a classic state machine, you will need to add a transition for this Overcurrent event on each state, which will be boring and, more importantly, will make software maintenance complex. With an HSMS, you simply enclose all relevant states in a super state and add only one transition from this super state to a MotorOff state when the Overcurrent event is triggered. So, in a nutshell, a super state is a kind of subprogram common to several states. **THE DISPENSER**

![Figure 4](image28x93_to_566x814)

**Figure 4**—Hierarchical state machines (HSMS) add the possibility of super states. This enables you to define a common behavior for several states. Guard conditions and entry/exit actions provide even greater flexibility.
Timer is then started and each pressed key is added to the current code word, as long as the maximum length is not achieved (using a guard condition). Each ALPHA key press switches between numeric and alphanumeric mode. The two corresponding states are enclosed in a super state, which manages the other events in a common way. Pressing Delete removes one byte from the current code. A timeout stops the key sequence and returns to the initial state. Pressing Enter either delivers the beverage or displays an error message, depending on whether or not the code was correct (once again, using guard conditions). As you see, using a super state drastically simplifies the design, as all common actions are described only once.

If you look closely, you will see that when the correct code is entered, the corresponding action is SendMsg(HSM.NEW_CODE);. This is how several state machines can work together. Here, this code sends an event to the pump-management state machine (see Figure 3).

Drawing this state chart is not straightforward. You must think twice about each way you want your design to function. However—and this is an important point—as soon as the state chart is on the table, the system is fully specified in an unambiguous way. You may or may not be happy with the result, but everyone will understand the diagram the same way. This is why UML—and state charts in particular—are great specification methods. If a customer accepts the state chart, there shouldn’t be any discussion at the software acceptance phase (as long as the customer knows UML and there aren’t any bugs in the software).

FROM UML TO C

Now you have a set of state machines, so how would you actually implement them in your software? You could use one of the automatic code generation tools available on the market, but let’s assume you want to do something yourself. There are several coding methods, which are all well described in Samek’s book. But, I agree with Samek that the most efficient and clean one, at least when using a classic language such as C on a small microcontroller, is to use pointers to functions.

Figure 6 provides an overall view of the concept: Each state machine has two data areas in RAM: A first-in, first-out (FIFO) queue to store the incoming events (with associated parameters if needed) and a pointer to a function, the “current state pointer.” This pointer simply indicates the address in memory of a function that manages the current state (so one function should be coded for each state). Then the main program executes a small state-machine manager function, which checks whether any state machine has a nonempty event queue (if some state machines must be more reactive than others, then a priority mechanism could be implemented there).

If a queue is not empty, the state machine manager simply calls the function pointed by the current state pointer, and passes it the first event of the queue as an argument. The state-handling function then implements a “switch/case” C structure and executes the corresponding actions depending on the event and taking care of the guard conditions. If a state change is needed, the current state pointer is simply overwritten.

**Figure 5**—This FSM can specify the beverage-dispenser keypad’s full functionality.

**Figure 6**—One good way to implement FSM-based software is by using an event queue and a pointer to function for each state machine of the system. The pointer contains the program memory address of the function to be executed in the current state when an event is triggered.
Listing 1—The type definitions and global variables should be easy to read. Five functions are enough to implement the full-state manager. Here, the three global variables are the event queues (one per state machine) with their accompanying pointers to the first and last used queue cells, one pointer to function per state machine, and one pointer to the idle-management function for more flexibility.

```c
typedef unsigned char HSM_T_HSMD; // HSM types
typedef unsigned char HSM_T_SIGNALID;
typedef unsigned long HSM_T_SIGNALPARAMETER;

typedef struct {
    HSM_T_SIGNALID SignalId;
    HSM_T_SIGNALPARAMETER SignalParameter;
} HSM_S_SIGNAL;

typedef enum {HSM_PRIORITY_HIGH=0, HSM_PRIORITY_MEDIUM, HSM_PRIORITY_LOW, HSM_UNUSED} HSM_T_PRIORITY;

typedef void (*HSM_T_STATEHANDLINGFUNCTION)(const HSM_T_SIGNAL Signal);
typedef void (*HSM_T_IDLEFUNCTION)(void);

// HSM global variables
extern HSM_T_SIGNAL HsmSignalOwa[4][HSM_FBF_SYSTEM][HSM_MSL_QQUEUE_LEN_TEYTH];
extern unsigned short HsmSignalQueueFirstFreePos[HSM_FBF_SYSTEM];
extern unsigned short HsmSignalQueueFirstUsedPos[HSM_FBF_SYSTEM];
extern HSM_T_STATEHANDLINGFUNCTION HsmCurrentState[HSM_PER_SYSTEM];
extern HSM_T_PRIORITY HsmPriority[HSM_PER_SYSTEM];

// Prototype declarations
void HsmInit(void);
void HsmDefineIdlefunction(HSM_T_IDLEFUNCTION idle_function);
void HsmSendMsg(HSM_T_HSMD msg_id, HSM_T_SIGNALID signal_id, HSM_T_SIGNALPARAMETER parameter);
void HsmRun(void);
```
void HsmRun(void)
{
    signed char hsm_to_proceed;
    HSM_FURENITY priority;
    unsigned char i;
    // Never returns
    while(TRUE)
    {
        // dispatch messages
        do
        {
            hsm_to_proceed=1;
            // Look for an HSM with a non empty message queue.
            // Starting with high priority HSM's.
            for(priority=HSM_PRIORITY_HIGH;
                (priority>HSM_PRIORITY_LOW) && (hsm_to_proceed==0);
                priority++)
            {
            for(i=0;(i<HSM.Profile.HSM_TO_PROCESS&&hsm_to_proceed==0);i++)
            {
                if(HsmPriority[i]==priority)
                {
                    if(HsmSignalQueueFirstFreePos[i]!=HsmSignalQueueFirstUsedPos[i])
                    {
                        hsm_to_proceed=1;
                    }
                }
            }
        }
        if(hsm_to_proceed==0)
        {
            // Execute HSM
            assert(HsmCurrentState[0][hsm_to_proceed]==(HSM_STATE)HANDLING_FUNCTION);
            (*HsmCurrentState+HsmSignalQueue[hsm_to_proceed])[
            HsmSignalQueue[0][hsm_to_proceed][hsm_to_proceed]]+=HSM.MSG.QUEUE_LENGTH;
            disable_interrups();
            while(hsm_to_proceed==0);
            // Call idle management function if it is defined
            if(HsmIdleFunction != (HSM_IDFUNCTION) NULL)
            (*HsmIdleFunction());
        }
    }
}

This is a simplified description, but I hope you followed.
Entry and exit actions are usually managed as specific events
passed to the state-management function and hierarchical
relationship as subroutine calls. Lastly, an idle-management
function could be called by the state machine manager when
all event queues are empty. This is a good place to switch the
processor in low-power mode and wait for an event. By the
way, the best way to generate events is through interrupt
routines. If a key is pressed or if any other asynchronous
event is generated, then the corresponding interrupt routine
could simply add an entry as the event FIFO (as long as it is
not already full).

Samek's QP framework form is a good implementation
of this concept and I strongly encourage you to have a look at
it. I preferred to code my own version, as I thought this was
the best way to actually understand what was going on. How-
ever, my implementation is quite simplified. For example, my
code only supports a simple parameter per event. It doesn't
support entry/exit actions for super states, and so forth.

Anyway, the full source code for my micro HSM code and
a demonstration based on the beverage dispenser example are
available on Circuit Cellar's FTP site. Feel free to use it and
adapt it for your application. There isn't any constraint as
long as you don't ask for support or cry if it doesn't work as
expected.

For those who may be reluctant to dig into the full code, I
closed some snapshots in this article. Listing 1 shows the types
and global variables definitions. For those who are not used to
understand and modify the software, I encourage you to do so, as this article provides only a light presentation. The code posted on the FTP site includes a full project you can run on your PC under Windows, thanks to Bloodshed Software’s free DEV-C++ tool suite. Timers and key events are simulated by a polling loop. The code displays some debug information on the screen and enables you to get a crude simulation of the dispenser (see Photo 1). And, no, sorry, it doesn’t actually deliver Cognac out of your PC USB connector.

WRAPPING UP

Here we are. Presenting UML state charts, and their coding in C in a short article was a tricky assignment, and you may feel frustrated without more detailed explanations. However, I hope you are interested enough to dig into that strong method and coding approach, either by playing with my example code or by reading Samek’s book. In any case, HMSIs are no longer on your darker side!

Robert Lacoste lives near Paris, France. He has 24 years of experience working on embedded systems, analog designs, and wireless telecommunications. He has won prizes in more than 15 international design contests. In 2003, Robert started a consulting company, ALCIOM, to share his passion for innovative mixed-signal designs. You can reach him at riacoste@alciom.com. Don’t forget to write “Darker Side” in the subject line to bypass his spam filters.

PROJECT FILES

RESOURCES