# UML Statecharts at $10.99
# by Miro Samek

***Article published on DrDobbs.com in May 2006***

Too many embedded developers believe that the Unified Modeling Language (UML) is all about using big tools and that the UML concepts, such as the advanced form of state machines (UML statecharts), are just too heavy for smaller embedded microcontrollers.

In this article I describe a method and software for implementing UML statecharts in C, small enough to fit a low-end 8-bit microcontroller. More specifically, I present a nontrivial UML statechart example that runs on the USB Toolstick from Silicon Laboratories (see Photo 1) with room to spare. The USB Toolstick is a complete evaluation board for the C8051F300 microcontroller (256 bytes of RAM, 8KB of Flash, 11 pins). The Toolstick is available on-line from the Silicon Labs website for just $10.99. Actually, you can even start without the Toolstick, because I provide a Toolstick software emulation that uses the same state machine code, but runs on a Windows PC.

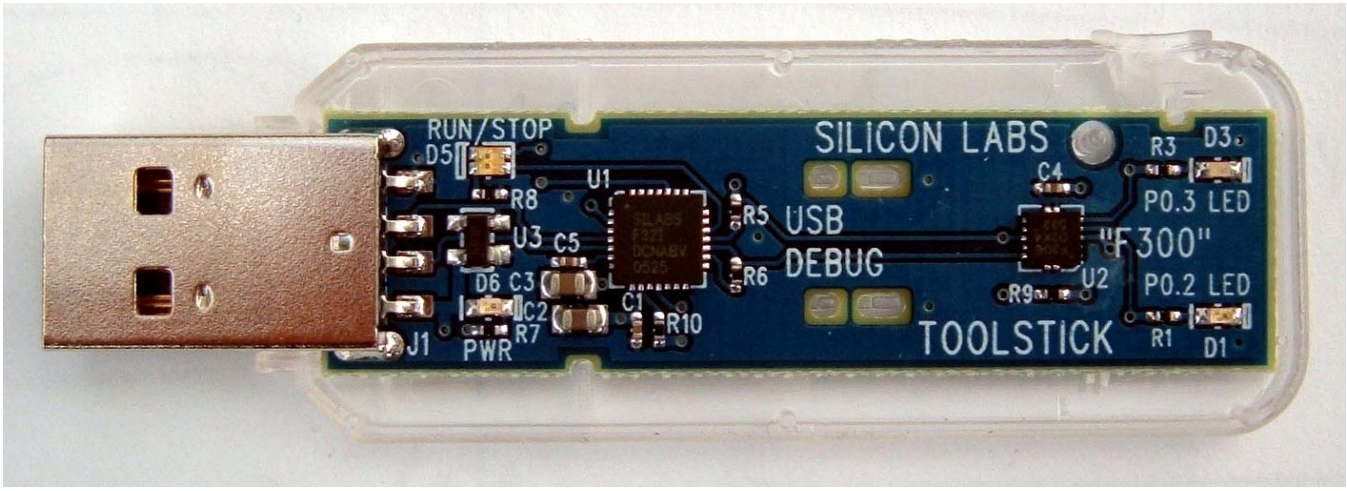


**Photo 1—***The USB Toolstick from Silicon Labs combines the USB debugger (left part) and the C8051F300 microcontroller (the smaller chip to the right). The "F300" controls the two LEDs at the right edge of the Toolstick (green LED at the top and red LED at the bottom).*

I begin with a very light introduction to statecharts followed by a statechart design example. Next I give a step-by-step guide for coding the designed statechart in portable C. Finally, I show how to deploy multiple concurrent state machines on the USB Toolstick. I assume that you are somewhat familiar with the traditional state-transition diagrams, as well as basic real-time concepts, such as event queues, interrupt processing, and non-preemptive scheduling.

## 1. Why Bother?

The legitimate question is “Why should you even bother using advanced UML statecharts in a low-end 8-bitter?” After all, these parts are so small that no big programs can fit into them anyway, let alone designs requiring UML statecharts.

Well, I think that a lot can go wrong in 8KB, or even 4KB of code. All these small microcontrollers are archetypal event-driven systems that must constantly react to events. In general, the reaction to an event depends both on the event type and, more importantly, on the current execution context. For example, if you press a button of an electronic watch, the watch probably reacts quite differently when it is in the timekeeping mode, compared to the same button pressed in the setting mode.

From the programming point of view, the dependency on the context often leads to convoluted, deeply nested if-else “spaghetti” code. The “spaghetti” results from capturing the various bits and pieces of the relevant event history (the context) in multitude of variables and flags, and then setting, clearing, and testing these variables in complex expressions scattered throughout the code.

Finite State Machines (FSMs) offer a much better alternative because they make the reactions to events explicitly dependent on the execution context, represented as “state”. By recognizing the importance of the context upfront, FSMs become very powerful “spaghetti reducers” that drastically cut the number of execution paths through the code, simplify the conditions tested at each branching point, and simplify transitions between different modes of operation [1]. In doing this, state machines eliminate many variables and flags used to store the context, and replace all that cram with a single “state variable”. Therefore the resulting application typically requires less RAM than the original “spaghetti”, which is a big deal for a small 8-bitter.

But I’m probably not saying here anything new. The classical automata theory has been around since dirt. However, it is also widely known that the traditional FSMs have a nasty tendency called “state and transition explosion”. The number of states and transitions necessary to represent a system tends to grow much faster than the complexity of the system itself because the traditional FSM formalism imposes repetitions [2]. For example, a FSM model of a simple 4-operation calculator might have some 15 states. Every one of these states needs to handle the ‘C’ (CANCEL) event, because the user must be able to cancel a computation and start over at any stage. In a traditional FSM you have no choice but to repeat the essentially identical CANCEL transition some 15 times. Similarly, you have to repeat at least a few times the ‘CE’ (CANCEL_ENTRY) transition, the ‘=’ (EQUALS) transition, and many others. Needless to say, the resulting code is not just bloated, but it also is full of impossible to maintain repetitions, all of which renders the whole formalism hardly useable. Please note that the complexity level at which FSMs start to “explode” is quite low. Apparently, traditional state machines are already in trouble to handle the complexity of a 4-operation calculator, a small home appliance, or even a more advanced digital watch. These are all application areas for low-end microcontrollers.

To become truly useable, the classical automata theory needs a mechanism for sharing reusing and transitions across many states. The formalism of *statecharts*, invented originally by David Harel [3] and adapted subsequently into virtually all modern methodologies, including the UML, adds exactly such a mechanism. By allowing hierarchical nesting of states, statecharts provide a very efficient way of sharing behavior, so that the complexity of a statechart no longer explodes but grows linearly with the complexity of the modeled system. Obviously, formalism like this is a blessing to embedded developers because it makes the state machine approach truly applicable to real-life problems [2].

Sidebar: **Reuse of Behavior in UML Statecharts**

The most important innovation of statecharts over the classical FSMs is the introduction of *hierarchically nested states* (that's why statecharts are also called hierarchical state machines). The semantics associated with state nesting (shown in Figure 1(a)) are as follows: If a system is in the nested state "s11" (called the substate), it also implicitly is in the surrounding state "s1" (called the superstate). This state machine will attempt to handle any event in the context of state "s11", which conceptually is at the lower level of the hierarchy. However, if state "s11" does not prescribe how to handle the event, the event is not quietly discarded as in a traditional "flat" state machine; rather, it is automatically handled at the higher level context of state "s1". This is what is meant by the system being in state "s11" as well as "s1" at the same time. Of course, state nesting is not limited to one level only, and the simple rule of event processing applies recursively to any level of nesting.

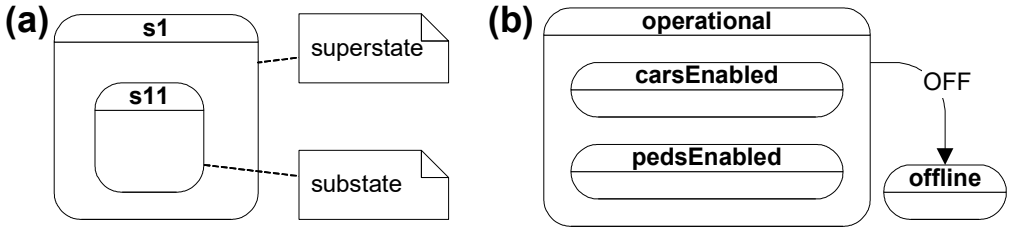


**Figure 1—***(a) UML notation for hierarchically nested states; (b) UML state diagram of a PELICAN (PEdestrian LIght CONtrolled) crossing, in which states "carsEnabled" and "pedsEnabled" share the common transition OFF to the off state.*

As you can see, the semantics of hierarchical state decomposition facilitate sharing of behavior through *programming-by-difference*. The substates need only define the differences from the superstates. A substate can easily reuse the common behavior from its superstate(s) by simply ignoring commonly handled events, which are then automatically handled by higher-level states. In this manner, the substates can share all aspects of behavior with their superstates. For example, in a state model of a PELICAN (PEdestian LIght CONtrolled) crossing shown in Figure 1(b), states "carsEnabled" and "pedsEnabled" share a common transition OFF to the "offline" state, defined in their common superstate "operational".

State nesting goes hand-in-hand with another feature of statecharts, which is the provision of guaranteed initialization and cleanup of nested states. Every state in a UML state machine can have optional entry actions, which are executed upon entry to the state, as well as optional exit actions, which are executed upon exit from the state. Entry and exit actions are associated with states, not transitions. Regardless of how a state is entered or exited, all of its entry and exit actions must be executed.

To be compatible with the *programming-by-difference* principle, the order in which entry actions are executed must always proceed from supertates to substates, because substates rightfully expect to be responsible only for the differences from the initialization already performed by the superstates. By the same argumentation, the order in which exit actions are executed must be exactly reversed, that is, exit actions must be executed from the innermost substates to superstates.

UML statecharts have a very compelling graphical notation, which preserves the general form of the traditional state-transition diagrams. The following Figure 2 summarizes the most important elements of the UML statechart notation [4].

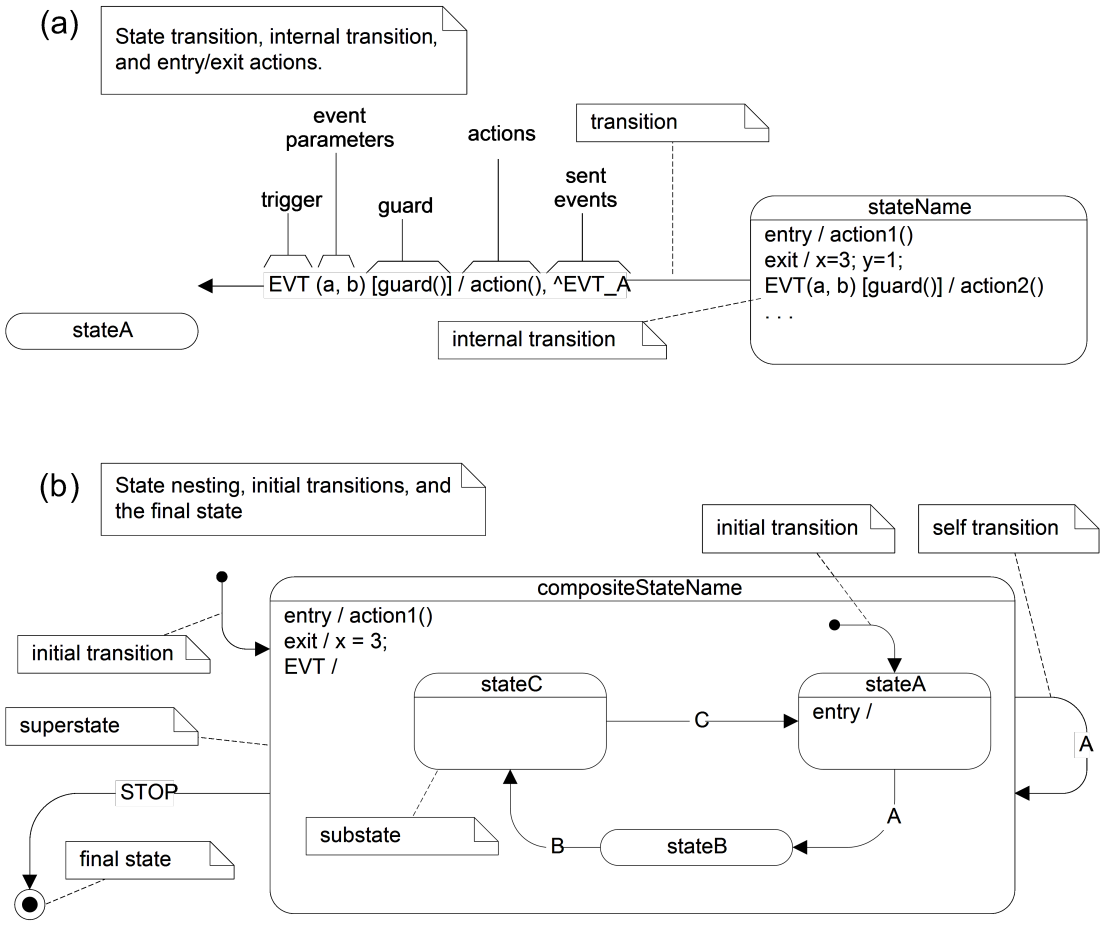


**Figure 2—***Main elements of the UML statechart notation.*

## 2. Designing a Statechart

In view of the very limited capabilities of the Toolstick I was rather constrained with the choice of a compelling example. Basically, the Toolstick can only blink its two LEDs (see Photo 1), or at most change the LED intensity using the built-in PWM generators. To me this resembled the operation of a PEdestrian LIght CONtrolled (PELICAN) crossing that I've once used in one of my earlier articles [2]. Here, I have improved and expanded the example to demonstrate the various aspects of designing a non-trivial statechart.

Before I begin, I need to provide you with a quick problem specification. The PELICAN crossing controller starts with cars enabled (green light for cars) and pedestrians disabled (don't-walk signal for pedestrians). To activate the traffic light change, a pedestrian must push the button at the crossing, which generates the PEDS_WAITING event. In response, the cars get the yellow light, which after a few seconds changes to red light. Next, pedestrians get the walk signal, which shortly thereafter changes to the flashing don't-walk signal. When the don't-walk signal stops flashing, cars get the green light again. After this cycle, the traffic lights don't respond to the PEDS_WAITING button press immediately, although the button "remembers" that it has been pressed. The traffic light controller always gives the cars a minimum of several seconds of green light before repeating the traffic light change cycle. One additional feature (coming late into the project) is that at any time an operator can take the PELICAN crossing offline (by providing the OFF event). In the "offline" mode, the cars get a flashing yellow and pedestrians flashing don't-walk signal. At any time the operator can turn the crossing back online (by providing the ON event).

Due to the hierarchical character of statecharts, you can approach the design from the top down or bottom up. In this design walkthrough I will use a combination of both approaches.

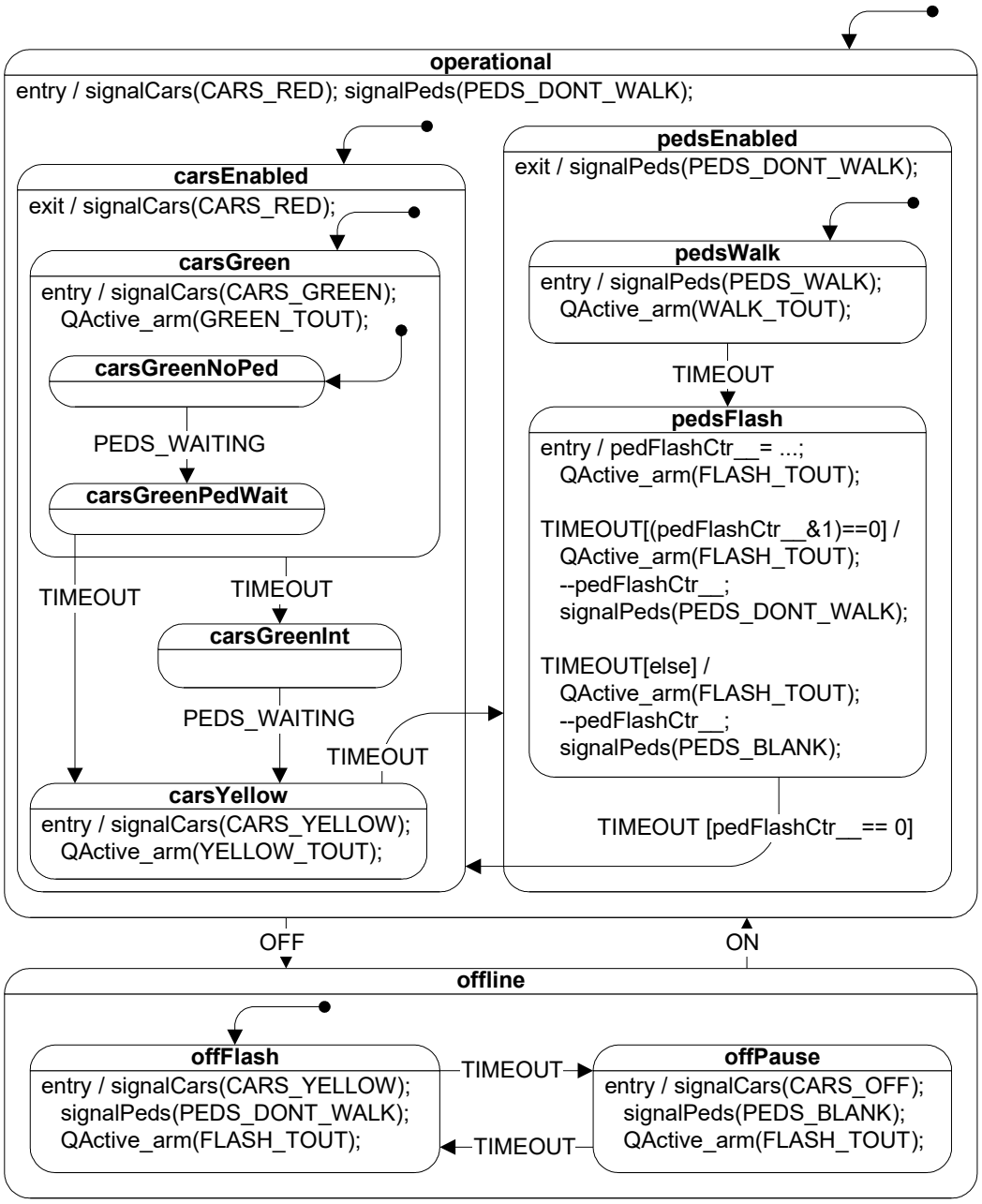


**Figure 3—** *The complete PELICAN crossing statechart.*

The limitation on the number of pictures in this article does not allow me to show a series of progressively elaborated statecharts, which would be perhaps the most educational method of explaining the design process. Instead, Figure 3 shows the complete PELICAN crossing statechart that I'll gradually explain in the text.

I start the design with just two states: "carsEnabled" and "pedsEnabled". This pair of states realizes the main function of the PELICAN crossing, which is to alternate between enabling cars and enabling pedestrians. Obviously, both states need some substates to realize the details of the specification, but I ignore this at first. At this stage, I only make sure that the design guarantees mutually exclusive access to the crossing, which is the main safety concern here. Please note that the exit action from the "pedsEnabled" state disables pedestrians, and the exit action from "carsEnabled" disables cars. Now, due to the guarantee of cleanup, these exit actions will be executed whichever way the states happen to be exited, so I can be sure that the pedestrians have always don't-walk signal outside the "pedsEnabled" state and cars have the red light outside the "carsEnabled" state.

In the next step, I concentrate on the internal structure of the “pedsEnabled” state. The job of the “pedsWalk” substate is to display the walk signal and to time out. The job of the “pedsFlash” substate is to turn the don’t-walk signal on and off. All actions are triggered by the TIMEOUT events, which are generated by the timer object associated with the state machine. The function QActive_arm() arms the timer for a one-shot delivery of the TIMEOUT event in the specified number of clock ticks. In the substate “pedsFlash”, the TIMEOUT event triggers two *internal transitions* and one regular transition leading out of the state. Internal transitions in UML are different from regular transitions, because the internal transitions never cause execution of state exit or entry. Internal transitions have also a distinctive notation that is similar to the entry and exit actions (see also Figure 2). The two internal transitions in state “pedsFlash” have different guard conditions (the Boolean expressions in the square brackets), which means that they are enabled only if the conditions in the square brackets evaluate to TRUE. The guard conditions are based in this case on the internal counter pedFlashCtr_ that controls the number of flashes of the don’t-walk signal.

In the following step, I elaborate the internal structure of the “carsEnabled” state. The most interesting problem here is to guarantee the minimum green light for cars before enabling pedestrians. Upon entry to the “carsGreen” substate, the timer is armed to expire after the minimum green light time. When the PEDS_WAITING event arrives before the expiration of this timeout, the active state is “carsGreenNoPed”, and the state machine transitions to the substate “carsGreenPedWait”, which has the purpose of “remembering” that a pedestrian is waiting. When the minimum green light time expires in the “carsGreenPedWait” state, it triggers the TIMEOUT transition to the “carsYellow” state, which after another timeout transitions out of “carsEnabled” state to open the crossing to pedestrians. However, if the PEDS_WAITING event does not arrive before the minimum green light timer expires the state machine will be in the “carsGreenNoPed” state that does not prescribe how to handle the TIMEOUT event. Per the semantics of state nesting, the event is passed to the higher-level state, that is, to “carsGreen”, which handles the TIMEOUT event in the transition to “carsGreenInt” (interruptible green light).

At this point, the statechart accomplishes the main functionality of the PELICAN crossing. The design progressed top-down, by gradually elaborating the inner structure of hierarchical states. However, you can also design statecharts in the bottom-up fashion. In fact, this is the best way to add the last feature: the “offline” mode of operation.

The “offline” mode of operation is added simply by enclosing the whole state machine elaborated in the previous steps inside the superstate “operational” that handles the transition OFF to the “offline” state. Please note how the state hierarchy ensures that the transition OFF is inherited by all direct or transitive substates of the “operational” superstate, so regardless in which substate the state machine happens to be, the OFF event always triggers transition to “offline”. Now, imagine how difficult it would be to make such a last-minute change to a traditional, non-hierarchical FSM.

The PELICAN crossing is ready now, but we still have a big problem of actually generating the external events to the PELICAN state machine, such as PED_WAITING, ON, and OFF. The actual PELICAN crossing hardware will provide a push button for generating the PED_WAITING event, as well as the ON/OFF switch for the to generate the ON/OFF events, but the Toolstick has no external input (see Photo 1). For Toolstick, we need to simulate the pedestrian/operator in a separate state machine. This is actually a good opportunity to demonstrate how to combine many state machines that collectively deliver the intended functionality of the application. Please refer to the accompanying code for the implementation of the straightforward Pedestrian state machine.

## 3. Coding a Statechart in C

Contrary to widespread misconceptions, you don't need big code-synthesizing tools to translate UML statecharts to efficient and highly maintainable code. This section explains step-by-step how to hand-code the PELICAN crossing statechart from Figure 3 in portable C.

The implementation strategy I'm going to use is straightforward, because most of the complexities in managing state hierarchy and executing correct exit and entry actions in state transitions are handled by a generic "event processor" from Quantum Leaps, LLC, called QP-nano. In fact, QP-nano is more than just an event processor; it is a complete platform for executing concurrent state machines. Besides the optimized, hierarchical event processor, QP-nano provides also event passing mechanism, event queuing, time event generation (timers), and a simple non-preemptive, prioritized scheduler to execute state machines in run-to-completion (RTC) fashion. All these services require about 1300 bytes of code (ROM) and just a few bytes of RAM on the 8051.

### 3.1 Representing Events

QP-nano has been specifically designed for small systems with very limited RAM. In this minimal version events are represented as structures containing the byte-wide enumerated type of the event, such as TIMEOUT or PED_WAITING, which in the UML is called the *signal*. Optionally, QP-nano allows every event to have a single scalar event parameter. Event parameters are very useful to convey the quantitative information associated with the event. For example, an ADC conversion might generate an event with the signal ADC_READY and the parameter containing the value produced by the ADC.

### 3.2 Coding States

Each state in QP-nano is represented as a C function called a state handler function. The job of QP-nano is to invoke these state handler functions in the right order to process events according to the UML semantics.

A state handler function is a regular C function that takes the state machine pointer as the argument and returns a pointer to another state handler function, which is typedef-ed as QSTATE in the QP-nano header file qpn.h. You need to structure your state handler functions such that they return the pointer to the superstate handler, if they don't handle the current event, or a NULL-pointer, if they handle the current event. QP-nano uses this information to "learn" about the nesting of states to process events hierarchically and correctly execute state transitions.

To determine what elements belong a given state handler function, you first need to look up the state in the diagram and follow around the state's boundary. You need to implement all transitions originating at the boundary, any entry and exit actions defined directly in this state, as well as all internal transitions enlisted directly in the state. Additionally, if there is an initial transition embedded directly in the state, you need to implement it as well. You don't worry about any substates nested in the given state. These substates are implemented in their own state handler functions.

Take for example the state "carsGreen" shown in Figure 3. This state has one transition TIMEOUT originating at its boundary, an exit action and the initial transition to the substate "carsGreenNoPed". The state "carsGreen" nests directly inside "carsEnabled".

```
(1) QSTATE Pelican_carsGreen(Pelican *me) {
(2)     switch (Q_SIG(me)) { /* switch on signal of the current event */
(3)         case Q_ENTRY_SIG: {                       /* entry action */
                QActive_arm((QActive *)me, CARS_GREEN_MIN_TOUT);
                BSP_signalCars(CARS_GREEN);
(4)             return (QSTATE)0;                    /* event handled */
            }
(5)          case Q_INIT_SIG: {
(6)             Q_INIT(&Pelican_carsGreenNoPed);/* initial transition */
(7)             return (QSTATE)0;                    /* event handled */
            }
(8)         case Q_TIMEOUT_SIG: {
(9)             Q_TRAN(&Pelican_carsGreenInt);    /* state transition */
```

```
(10)             return (QSTATE)0;                    /* event handled */
                }
            }
(11)        return (QSTATE)&Pelican_carsEnabled;     /* the superstate */
    }
```

**Listing 1—***State-handler function for the "carsGreen" state.*

Listing 1 shows the state handler function Pelican_carsGreen() corresponding to the PELICAN state "carsGreen". The state handler takes only one argument: the state machine pointer; Pelican* in this case (1). By convention, I always name this argument "me". (If you are familiar with C++, you'll recognize that "me" corresponds to the "this" pointer in C++.) Generally, every state handler is structured as a big switch that discriminates based on the signal of the current event. To reduce the number of arguments of the state handler function, QP-nano stores the current event in the state machine object pointed to by the "me" pointer. For convenience, QP-nano provides the macro Q_SIG() to access the signal of the event (2). Each case is labeled by an enumerated signal and terminates with "return (QSTATE)0". Returning a zero-pointer from a state handler informs the event processor that the particular event *has* been processed. On the other hand, if no case executes, the state handler exits through the final return statement, which returns the pointer to the superstate handler function (QSTATE)&Pelican_carsEnabled in this case (11). Please note that the final return statement from a state handler function is the *only* place where you specify the hierarchy of states. Therefore, this one line of code represents the *single point of maintenance* for changing the nesting level of a given state.

At the label (3) in Listing 1, you can see how to code the **entry action**. QP-nano provides a reserved signal Q_ENTRY_SIG (and also Q_EXIT_SIG for exit actions) that the event processor sets in the state machine before calling the appropriate state handler function to execute the state entry actions. Therefore, to code a state entry action, you provide a case statement labeled with signal Q_ENTRY_SIG, enlist all the actions you want to execute upon the entry to the state, and terminate the actions with "return (QSTATE)0" (4). Coding an **exit action** is identical, except that you provide a case statement labeled with signal Q_EXIT_SIG.

Every composite state (a state with substates) can have its own initial transition, which in the diagrams is represented as an arrow originating from a black ball. For example, state "carsGreen" in Figure 3 has such a transition to the substate "carsGreenNoPed". QP-nano provides a reserved signal Q_INIT_SIG that the event processor sets in the state machine before calling the state handler function to execute the initial transition. At the label (5) of Listing 1 you can see how to code the **initial transition**. You provide a case statement labeled with signal Q_INIT_SIG, enlist all the actions you want to execute upon the initial transition, and then designate the target substate with the Q_INIT() macro (6). You terminate the case statement with "return (QSTATE)0", which informs the event processor that the initial transition has been handled (7).

Finally, at the label (8) in Listing 1, you can see how to code a regular **state transition**. You provide a case statement labeled with the triggering signal (Q_TIMEOUT_SIG in this case), enlist the actions, and then designate the target state with the Q_TRAN() macro provided by QP-nano (9). You terminate the case statement with "return (QSTATE)0", which informs the event processor that the event has been handled (10).

And this is about all you need to know to code any state (Note: to conserve stack space, QP-nano can handle up to 4 levels of state nesting). The PELICAN crossing source code (pelican.c) accompanying this article provides more examples, such as coding **internal transitions** and transitions with **guards**.

### 3.3 Declaring State Machine Objects

While state handler functions specify the state machine behavior, and as such are represented in code only (ROM), they require a state machine object (RAM) to remember the current active state and the current event. These state machine objects are represented in QP-nano as C structures derived from the QActive structure, which is provided in QP-nano header file qpn.h. The "derivation of structures" means simply, that you need to literally embed the QActive structure as the first member of the derived structure. Listing 2 shows the declaration of the Pelican structure. By a convention, I always name the parent structure member "super_".

```
typedef struct PelicanTag Pelican; /* type definition for Pelican */
struct PelicanTag {
    QActive super_;                      /* derived from QActive */
```

```
    uint8_t pedFlashCtr__;    /* private pedestrian flash counter */
};
```

**Listing 2—***Declaration of the Pelican state machine "derived from" QActive structure.*

Looking at Listing 2, you should convince yourself that the "derivation of structures" simply means aligning the QActive object at the beginning of every Pelican object in memory. Such alignment allows treating every pointer to Pelican as a pointer to QActive at the same time, so any function designed to work with a pointer to QActive will work correctly if you pass to it a pointer to Pelican. In other words, all functions that QP-nano provides for QActive objects will work just fine for the derived Pelican (or Pedestrian) objects. You can think of this mechanism as single inheritance implemented in C.

Actually, when you look at the declaration of the QActive structure in the qpn.h header file, you will notice, that QActive itself is also derived from another structure QHsm. The QHsm structure represents a Hierarchical State Machine (HSM) and stores the current active state and the current event. QActive adds to this an event queue and a timer, which are both necessary elements of an independently executing state machine. In UML, such independently executing entities are called *active objects*, which explains the origin of the name QActive. The memory cost of QActive object is 6 to 10 bytes of RAM, depending on the size of the pointer-to-function and the configured size of the timer counter.

Of course, it doesn't matter what else you add in the derived structure after the "super_" member. In Listing 2, I've declared additionally the pedFlashCtr__ counter, which the Pelican state machine uses for counting the number of flashes of the "don't walk" signal in the "pedsFlash" state (Figure 3).

## 3.4 Initializing State Machine Objects

Initialization of hierarchical state machines requires some attention because you need to execute the top-most initial transition, which in general case can be quite involved. For example, the top-most initial transition in the PELICAN crossing statechart (Figure 3) consists of the following steps: (1) entry to "operational", (2) initial transition in "operational", (3) entry to "carsEnabled", (4) initial transition in "carsEnabled", (5) entry to "carsGreen", (6) initial transition in "carsGreen", and finally (7) entry to "carsGreenNoPed". Of course, you want QP-nano to deal with this complexity, which it can actually do, but in order to reuse the already implemented mechanisms, you need to execute the top-most initial transition as a regular transition.

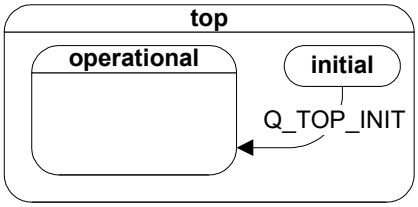


**Figure 4—***The top-most initial transition for the PELICAN state machine.*

Figure 4 illustrates how you do it. You need to provide a special state "initial" that handles the top-most initial transition as a regular state transition. The "initial" state is nested directly in the *top state*, which is the UML concept that denotes the ultimate root of the state hierarchy. QP-nano defines the top state handler function QHsm_top, which by default "handles" all events by returning NULL pointer. ("Handling" events in the top state means really silently discarding them, per the UML semantics.)

## 3.5 Configuring and Starting the Application

After you've coded all state machines, you need to tell QP-nano about them, so that it can start managing the state machines (or actually active objects) as components of the application.

QP-nano executes all active objects in the system in a run-to-completion (RTC) fashion, meaning that each active object completely handles the current event before processing the next one. After each RTC step, QP-nano engages a simple scheduler to decide which active object to execute next. The scheduler makes this decision based

on the priority statically assigned to each active object upon the system startup (priority-based scheduling). The scheduler always executes the highest-priority active object that has some events in its event queue.

```
(1) #include "qpn_port.h"                              /* QP-nano port */
    #include "bsp.h"                    /* Board Support Package (BSP) */
    #include "pelican.h"                    /* application header file */

    /*..........................................................*/
(2) static Pelican l_pelican;   /* statically allocate PELICAN object */
(3) static Ped     l_ped;    /* statically allocate Pedestrian object */

(4) static QEvent  l_pelicanQueue[1];        /* PELICAN's event queue */
(5) /* as the highest-priority task, Ped does not need event queue    */

    /*..........................................................*/
    /* CAUTION: the QF_active[] array must be initialized consistently
    * with the priority assignement in pelican.h
    */
(5) QActiveCB const Q_ROM QF_active[] = {
(6)     { (QActive *)0,          (QEvent *)0,    0                    },
(7)     { (QActive *)&l_pelican, l_pelicanQueue, Q_DIM(l_pelicanQueue)},
(8)     { (QActive *)&l_ped,     (QEvent *)0,    0 /* no queue */     }
    };
(9) uint8_t const Q_ROM QF_activeNum =
        (sizeof(QF_active)/sizeof(QF_active[0])) - 1;

    /*..........................................................*/
    void main (void) {
(11)    BSP_init();                           /* initialize the board */

(12)    Pelican_init(&l_pelican);  /* take the top-most initial tran. */
(13)    Ped_init(&l_ped, 15);      /* take the top-most initial tran. */

(14)    QF_run();                   /* start executing state machines */
    }
```

**Listing 3—***Static allocation of state machine objects and the main() function.*

Listing 3 shows how to configure and start the application. You customize QP-nano in the qpn_port.h header file, which contains extensive comments explaining all the options (1). Next, you statically allocate all active objects (2-3) as well as correctly sized event queues for them (4-5). Please note, that the highest-priority active object in the system, such as Pedestrian, might not need an event queue buffer at all, because the single event stored inside the state machine itself might be sufficient.

Next, at label (5) of Listing 3, you define and initialize a constant array QF_active[], in which you configure all the active objects in the system in the order of their relative priority. QP-nano has been carefully designed not to waste the precious RAM for any information available at compile time. The QF_active[] array is an example of such compile-time information and is allocated in the code-space by the Q_ROM modifier. Q_ROM is a macro that for the IAR 8051 compiler is defined as "__code" in qpn_port.h. Other Harvard-architecture processors can benefit from this scheme as well. Similarly, at label (9) of Listing 3, you define and initialize another compile-time variable QF_activeNum, which is the total number of active objects actually used in the system. Please note that the zero-element of the QF_active[] is unused, so the number of active objects in the application is the dimension of the QF_active[] array minus 1.

The main() function is remarkably simple. You call the board initialization (11), trigger all initial transitions in the active objects (12-13), and finally transfer the control to the QF_run() function, which implements the QP-nano scheduler running in an endless loop.

## 4. Deploying the Application on the Toolstick

Deploying the application on the Toolstick requires only providing the board-specific initialization and the time-tick interrupt, which must call the QP-nano function QF_tick() to generate the TIMEOUT events. The PELICAN example contains a small board support package (bsp.c) for the Toolstick, which has been modeled largely after the standard PWM_demo application that comes on the Toolstick CD.

The code accompanying this article contains an extensive README file explaining all the examples included and the usual workarounds for minor bugs in the demo tools and their incompatibilities. I just wanted to mention here that I ended up using the 4-KB KickStart™ 8051 compiler from IAR Systems, instead of the 2-KB demo version of the Keil 8051 compiler that comes with the Toolstick. Finally, because the memory footprint is of primary interest in this article, here are the numbers I've obtained with the IAR 8051 compiler optimized for size: QP-nano 1254 bytes of CODE, 1 byte of DATA; application totals: 2712 bytes of CODE, 16 bytes of DATA, 88 bytes of IDATA (including 64 bytes of stack), 1 byte of BIT memory.

## 5. Conclusions

UML-style state machines can help you produce efficient, maintainable, testable systems with well understood behavior, rather than creating the usual "spaghetti" code littered with convoluted ifs and elses. In this article I've demonstrated that the technique is applicable to quite small systems, starting from about 4KB of ROM, and some 128 bytes of RAM.

Contrary to widespread misconceptions, you don't need big UML tools to take full advantage of the hierarchical state nesting and the guaranteed initialization and cleanup of states, which are the most important features of UML statecharts. In fact, manual coding of a nontrivial PELICAN crossing statechart turned out to be a rather simple exercise in following just a few straightforward rules. The implementation technique based on an "event processor", such as QP-nano, results in concise, highly maintainable code that truly reflects the statechart structure without any repetitions. The resulting state machine representation in C is flexible, allowing even sweeping changes in the state machine structure to be accomplished quite easily, at any stage of the project.

Once you design a system with UML statecharts, you will not want to go back to the "spaghetti" code or even to the traditional RTOS. Welcome to the twenty-first century.

***Miro Samek*** *is the Founder and President of Quantum Leaps, LLC, a provider of real-time, state machine-based application frameworks for embedded real-time systems. He is the author of "Practical Statecharts in C/C++" (CMP Books, 2002), has written numerous articles for magazines, and is a regular speaker at the Embedded Systems Conference.*

## 6. Project Files

To download the code and additional files, go to state-machine.com/doc/articles.html .

## 7. Resources

[1] Miro Samek, “Back to Basics”, C/C++ Users Journal, December 2003.

[2] Miro Samek, “Déjà Vu”, C/C++ Users Journal, June 2003.

[3] David Harel, “Statecharts: A Visual Formalism for Complex Systems”, Science of Computer Programming, 8, 1987 (available online from http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf).

[4] Quantum Leaps, LLC, “Quick UML Reference” http://www.quantum-leaps.com/resources/goodies.htm#UML

## 8. Sources

**USB ToolStick Evaluation Kit**
Silicon Laboratories, Inc.
www.silabs.com/tgwWebApp/public/web_content/products/Microcontrollers/en/ToolStick.htm.

**Quantum Platform Nano (QP-nano)**
Quantum Leaps, LLC
http://www.state-machine.com/qpn

**IAR Embedded Workbench for 8051 KickStart™ version**
IAR, Inc.
http://supp.iar.com/Download/SW/?item=EW8051-KS4

**Borland Turbo C++ 1.01**
Borland Software Corporation
http://bdn.borland.com/article/0,1410,21751,00.html