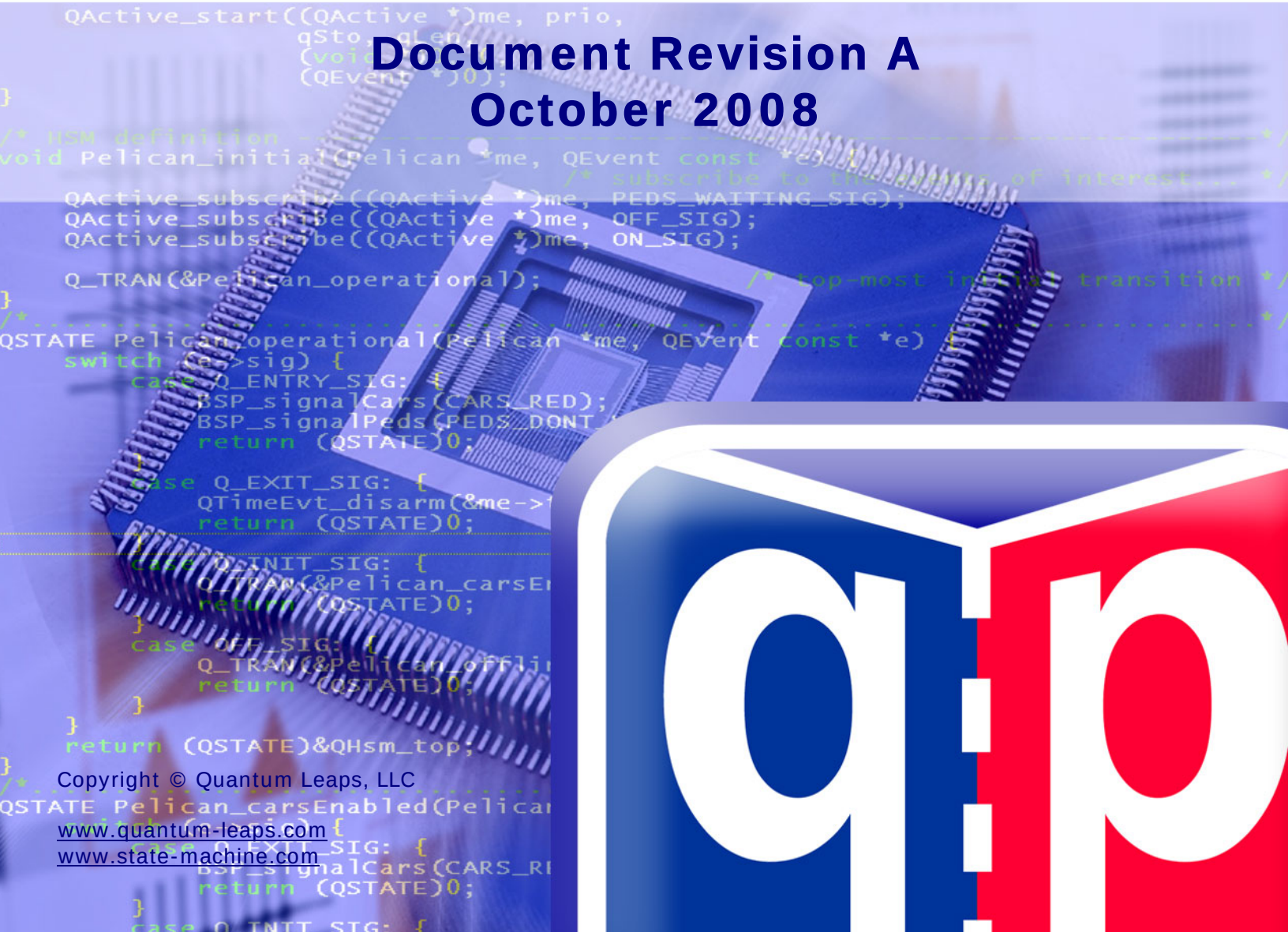
Quantum™ LeaPs
innovating embedded systems

Design Pattern
State-Local Storage



Document Revision A
October 2008

www.quantum-leaps.com
www.state-machine.com

Table of Contents

Design Pattern: State-Local Storage

1 Intent

Reduce memory footprint of state machines by introducing the state-local variable scope

2 Problem

In the standard UML state machine formalism all states of a state machine share the same set of variables (extended-state variables). Consequently, the memory required by a state machine is the aggregate of all extended-state variables needed in all states, even though only one state configuration[1] can be active at any given time. The aggregate memory footprint of all extended-state variables might be a problem for memory-constrained applications with a large number of states.

Figure 1 Embedded devices with user interfaces requiring a large number of states.

[1] When dealing with hierarchically nested states and orthogonal regions, the simple term "current state" can be quite confusing. In a hierarchical state machine (HSM), more than one state can be active at once. If the state machine is in a leaf state that is contained in a composite state (which is possibly contained in a higher level composite state, and so on), all the composite states that either directly or transitively contain the leaf state are also active. Furthermore, because some of the composite states in this hierarchy might have orthogonal regions, the current active state is actually represented by a tree of states starting with the single top state at the root down to individual simple states at the leaves. The UML specification refers to such a state tree as state configuration [OMG 07].

For example, a growing number of embedded systems have a user interface (UI) consisting of a small LCD display and a few buttons (see Figure 1). A UI of an MP3 player, a low-end cell-phone, or a wearable medical device can easily consist of hundreds of "screens", whereas each "screen" naturally corresponds to one or more states of the underlying state machine. In most such cases the UI code dominates the complexity of the embedded software. Many of these "screen" states require storing some state-specific information as long as the state is active. However, only one "screen" can be active at a time, so the memory occupied by the extended-state variables for one state can be reused for another state **safely**. Even though the amount of data per state is typically small, the aggregate of all these "extended-state" variables over the relatively large number of states can exhaust the available RAM quite quickly.

3 Solution

The concept of **state-local storage** (SLS) allows a state machine designer to reduce the memory footprint of a state machine by providing variables local to states. As the state machine transitions from one state to another, the SLS mechanism automatically **overlays** the extended-state variables for the target state configuration on top of the no-longer needed variables for the source state configuration. Thus, the concept of SLS enables a state machine to operate in less memory than the standard non-overlaid approach.

3.1 State-Local Storage Implementation with QEP™

In the usual state machine implementation with the QEP™ hierarchical event processor, a state machine is represented as a subclass (direct or indirect) of the `QHsm` base class. States correspond to static state-handler functions of this subclass. All state-handler functions access the same set of "extended-state" variables by means of the "`me`" pointer argument [PSiCC2 08].

To apply state-local storage, this standard implementation can be modified to use a **different** class for each state or a group of states. The state-specific classes allow declaring the local data for each state as well as state-handler functions that take "`me`" pointers of the corresponding state-class type. The result is that as the state machine transitions from state to state, the type of the "`me`" pointer automatically changes to reflect the different data members available to the state-handler functions, even though all "`me`" pointers point to the same address in memory (memory overlaying).

In hierarchical state machines (HSMs), the concept of state-local storage must of course take into consideration state nesting. Specifically, for the QEP implementation to work, the hierarchy structure of the state classes **must** replicate the hierarchy of state nesting.

NOTE: The standard UML transition execution sequence requires executing actions associated with the transition *after* exiting the source state configuration, but before entering the target state configuration. This is a big problem for the State-Local Storage concept, because the actions associated with transitions can access neither state-local variables of the source state configuration (already destroyed), nor the target state configuration (not yet created).

In contrast, as described in Chapter 2 of [PSiCC2], the transition sequence of the QEP™ hierarchical event processor executes the actions associated with the transition entirely in the context of the **source** state, that is, *before* exiting the source state configuration. This allows transition actions to operate on the state-local variables of the source state configuration.

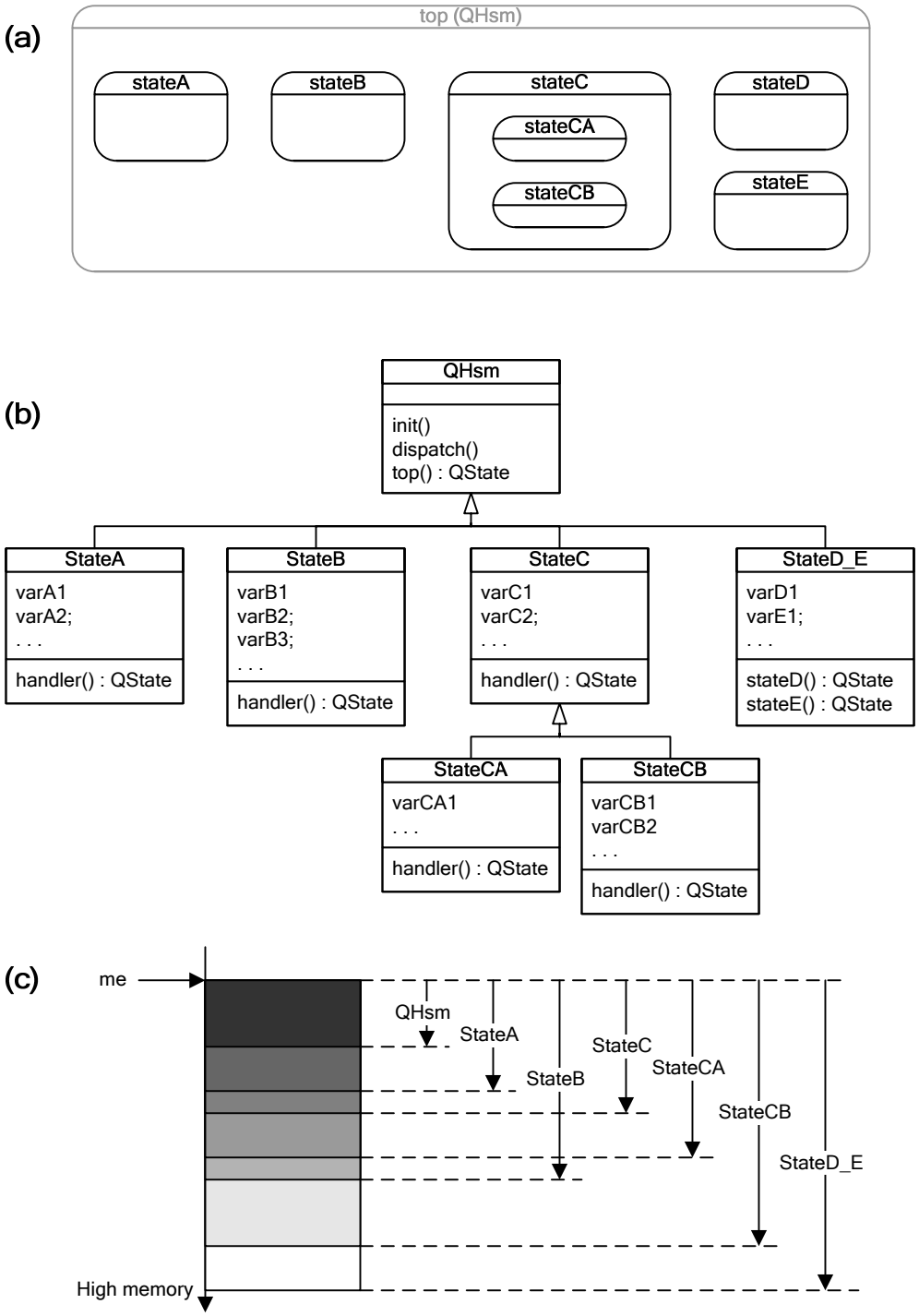


Figure 2 Hierarchical state machine (a), the corresponding state-class diagram (b), and the memory layout of a state machine instance (c).

Figure 2 shows graphically the SLS implementation with QEP. The class diagram in Figure 2(b) shows the state-class hierarchy corresponding to the hierarchical state machine from Figure 2(a). Please note that a state-class might represent more than one state. For example, class `StateD_E` corresponds to both "stateD" and "stateE". Panel (c) in Figure 2 shows the memory layout of the state machine class. All state instances are overlaid at the same memory address indicated as the

"me" pointer in Figure 2(c). This memory address is subsequently used by all state-handler functions of the state machine. Note, however, that the type of the "me" pointer used be different state-handler functions can be different, which means that the state handlers have different views of the same physical memory (memory overlaying).

3.2 Example User Interface Application

The example application for the State-Local Storage state pattern is found in the directory <qpcpp>\examples\80x86\dos\tcpp101\l\sls\ for QP/C++, <qpc>\examples\80x86\dos\tcpp101\l\sls\ for QP/C, and <qpn>\examples\80x86\tcpp101\sls\ for QP-nano, respectively. You can execute the application by double-clicking on the file SLS.EXE file in the dbg\ subdirectory. The example demonstrates several aspects of using SLS for bigger applications, including the use of "Orthogonal Component" and "Transition to History" state patterns. Figure 3 shows the three screens generated by the SLS.EXE application.

The SLS example consists of a user interface to numerical computations, such as computation of the Standard Deviation (Screen 1) and Linear Regression (Screen 2). The application provides also context-sensitive Help (Screen 0).

The SLS example starts with Screen 1, in which it computes Standard Deviation of an open-ended number of samples $\{x_i\}$. The User enters the sample x_i into the provided edit-box by means of the standard keyboard. The application parses the input and allows only valid floating-point numbers (e.g., --000.23..04 would be an invalid number). After typing in the number, the User presses ENTER to add the sample to the data set. After each data sample the system updates the number of samples (n), the average ($<x>$), and the two variance estimators (σ_n and σ_{n-1}). The User can cancel the last entry by pressing the 'E' key (Cancel Entry). The User can also cancel the whole data set of n-samples by pressing the 'C' key (Cancel). The User can navigate to the next screen by pressing the DOWN-arrow key and to the previous screen by pressing the UP-arrow key.

Screen 2 performs the Linear Regression computation, in which a straight line $y = a*x + b$ is fitted to n-samples of $\{(x_i, y_i)\}$ pairs. The User enters x_i and y_i data sample, whereas the application allows only valid floating-point numbers. After each data sample the system updates the number of samples (n), the direction coefficient (a), and the offset (b). The User can cancel the last entry (both x_i and y_i) by pressing the 'E' key (Cancel Entry). The User can also cancel the whole data set of n-samples by pressing the 'C' key (Cancel). The User can navigate to the next screen by pressing the DOWN-arrow key and to the previous screen by pressing the UP-arrow key.

On any screen, the User can request help, by pressing the 'F1' key, which opens the Help screen (Screen 0). The help is context-sensitive in that it offers a different text when activated from Screen 1 (Standard Deviation), and different when invoked from Screen 2 (Linear Regression). The Help screen always returns to the last active screen (History mechanism).

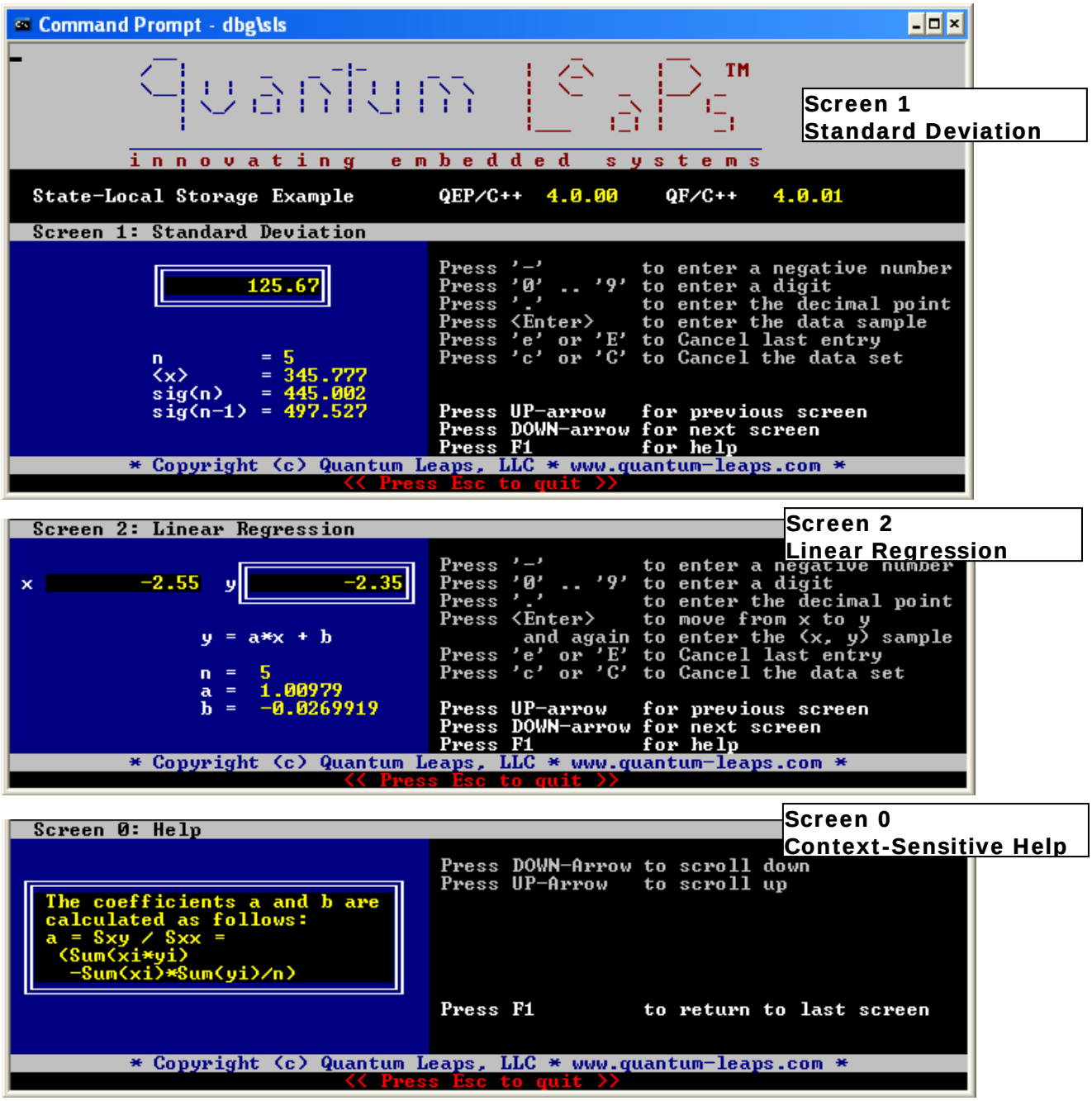


Figure 3 User Interface example with State-Local Storage.

Figure 4 shows the hierarchical state machine of the SLS example. The state machine has the "canonical structure" recommended in the "Ultimate Hook" design pattern (www.state-machine.com/-devzone/patterns.htm). This canonical structure consists of the "UI_top" superstate that implements the main functionality in its submachine, and the "UI_final" state that performs application shutdown in the entry action. This design guarantees that all substates of "UI_top" inherit the QUIT transition, and that the exit actions of the whole active state configuration will be executed upon the application shutdown.

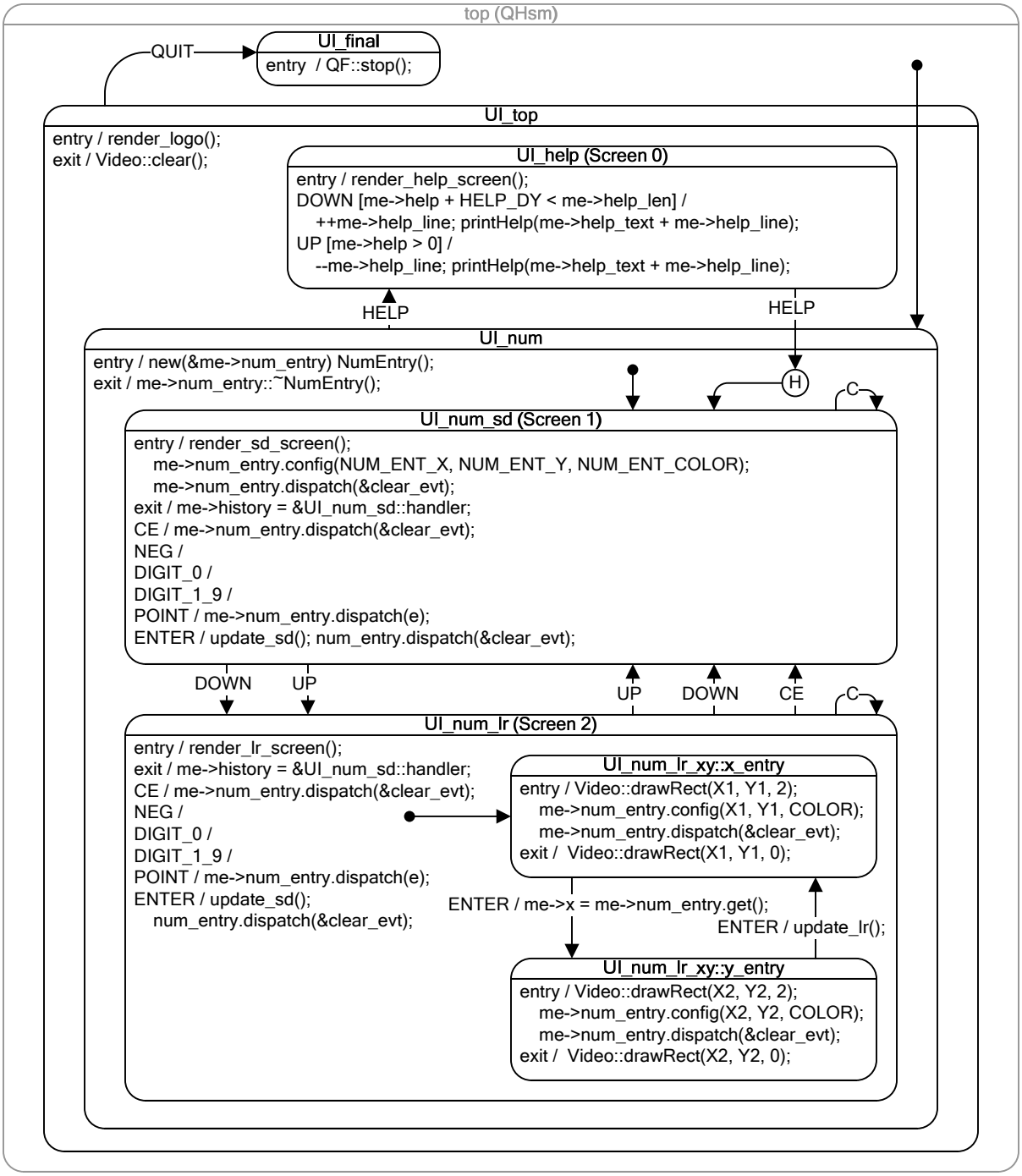


Figure 4 State machine of the SLS example.

The internal submachine of the "UI_top" superstate consists of the states "UI_help" for displaying help and "UI_num" for numerical computations. The most important purpose of the "UI_num" superstate is to instantiate and finalize the `num_entry` orthogonal component. The `num_entry` component is a state machine responsible for parsing the entered numbers and allowing only valid entries. All substates of "UI_num" have access to this component, because it is defined at the level of the "UI_num" state.

The "UI_num_sd" substate handles Standard Deviation computation (see Figure 3, Screen 1). This state needs additional extended-state variables to store the number of samples (n), the sum of samples (Σx_i), and the sum of squares (Σx_i^2). Based on these variables, for each new sample the action for the ENTER event computes the average ($<x> = \Sigma x_i / n$) as well as two variance estimators ($\sigma_n = sqrt(\Sigma x_i^2/n - <x>/n))$ and ($\sigma_{n-1} = sqrt(n/(n-1)*\sigma_n)$.

The "UI_num_lr" substate handles Linear Regression computation (see Figure 3, Screen 2). This state requires entering two numbers per sample (x_i, y_i). The job can be accomplished with just one `num_entry` parser orthogonal component (inherited from the "UI_num" superstate), because the user enters one number at a time. However, the state must "remember" which edit box (the x-edit-box or the y-edit-box) has the input-focus. This is accomplished by two substates "UI_num_lr_xy::x_entry" and "UI_num_lr_xy::y_entry", respectively.

NOTE: The addition of state-classes opens new dimension for managing state names. The name of the state becomes now a combination of the state-class name (e.g., `UI_num_lr_xy`) and the state-handler function name (e.g., `x_entry()`). As usual in UML, in a diagram you can abbreviate the names as long as they remain unambiguous in the given context.

Figure 5 shows the `NumEntry` state machine associated with the `num_entry` orthogonal component of the "UI_num" superstate. The job of the `NumEntry` state machine is to parse on-the-fly the digits entered by the user so that only valid numbers can be entered. Additionally, the `NumEntry` state machine is responsible for managing the input buffer and rendering the buffer contents to the screen.

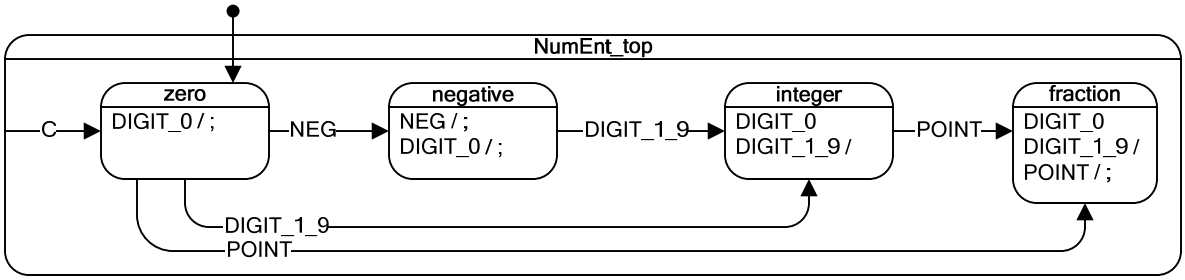


Figure 5 State machine of the NumEntry orthogonal component.

The `NumEntry` component can be configured to render the input buffer at given coordinates on the screen and to use a given color. That way, the `NumEntry` component can be associated with any number of numerical edit-boxes on the screen. For example, the `NumEntry` component is reconfigured from the x-edit-box to the y-edit-box as the input-focus shifts from x to y on the Linear Regression screen.

Figure 6 shows the UML class diagram of the SLS example. The inheritance relationships among state classes **must** be consistent with the state nesting of the state diagram (see Figure 4). For example, state "UI_num_sd" nests inside state "UI_num", so the class `UI_num_sd` must inherit from the class `UI_num`. However, not every state in the diagram needs to be represented by a separate state-class. States that don't need significant state-local storage don't need their own classes. For example, substates "UI_num_lr_xy::x_entry" and "UI_num_lr_xy::y_entry" are represented together by one state-class `UI_num_lr_xy`.

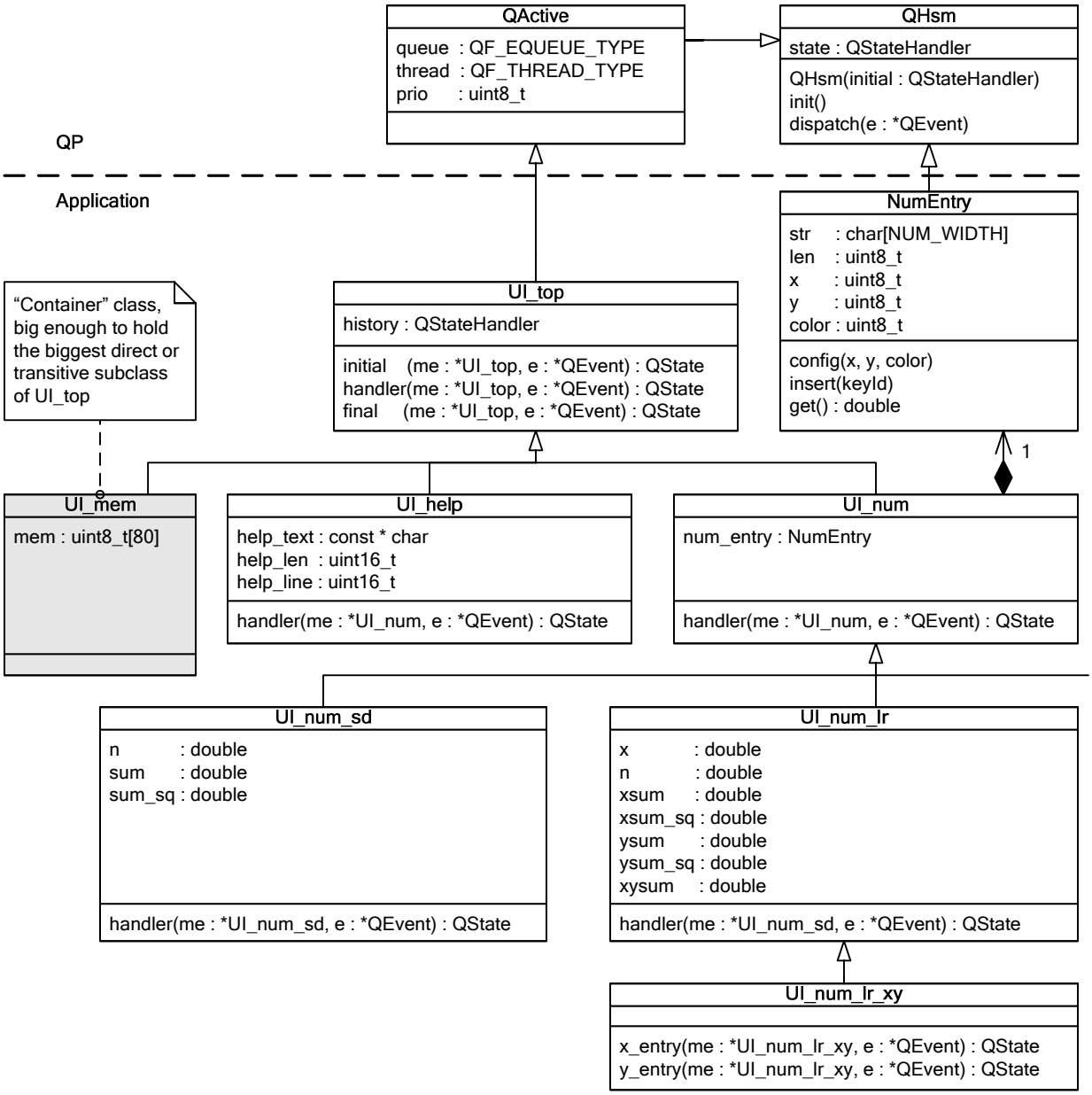


Figure 6 State-classes for the UI state machine.

The class diagram in Figure 6 shows also the `UI_mem` subclass of `UI_top` that does not directly correspond to any state. The purpose of this class is to provide a "container" big enough to hold the biggest subclass of `UI_top`. The "container" class `UI_mem` is the only one actually instantiated, and then all other subclasses of `UI_top` are overlaid in this memory, as described in Section 3.1.

4 Sample Code for QP/C++

The sample code for QP/C++ is found in the directory <qpcpp>\examples\80x86\dos\tcpp101\l\sls\. The code has been compiled with the legacy Borland Turbo C++ 1.01 compiler available for free download from http://dn.codegear.com/article/21751. The example application contains the Turbo C++ project files SLS.PRJ, SLS-REL.PRJ, and SLS-SPY.PRJ for building the Debug, Release, and SPY versions, respectively.

Even though the SLS example consists essentially of just one User Interface state machine, the source code is structured into several translation units (.CPP files). The goal is to demonstrate how to break up the state machine code to achieve possibly loose coupling among the various states ("screens"), so that they can be developed mostly independently and changes in one would not require massive recompilation in all others.

4.1 The Application Header File

The application header file is located in the file ui.h. This header file contains the elements that are shared across the whole application. For the SLS implementation, the application header file must expose the declarations of the higher-level state-classes, because they are necessary to derive the substate-classes. The explanation section following Listing 1 clarifies the main points of the application header file ui.h.

```
    #ifndef ui_h
    #define ui_h

    . . .

    //............................................................................
(1) class UI_top : public QActive {
    protected:
(2)     QStateHandler m_history;

    public:
(3)     UI_top(void) : QActive((QStateHandler)&UI_top::initial) {
        }

    protected:
(4)     static QState initial(UI_top *me, QEvent const *e); // initial pseudostate
(5)     static QState handler(UI_top *me, QEvent const *e);
(6)     static QState final  (UI_top *me, QEvent const *e);
    };
    //............................................................................
(7) class UI_mem : public UI_top {
    private:
(8)     uint8_t m_mem[80];    // maximum size of any substate (subclass) of UI_top
    };
    //............................................................................
(9) class UI_num : public UI_top {
    protected:
(10)    NumEntry m_num_entry;

    protected:
(11)    static QState handler(UI_num *me, QEvent const *e);
    };
(12) Q_ASSERT_COMPILE(sizeof(UI_num) < sizeof(UI_mem));

    //............................................................................
```

```
(13) class UI_num_sd : public UI_num {                          // standard deviation
     protected:
(14)     double m_n;
         double m_sum;
         double m_sum_sq;

     protected:
(15)     static QState handler(UI_num_sd *me, QEvent const *e);
     };
(16) Q_ASSERT_COMPILE(sizeof(UI_num_sd) < sizeof(UI_mem));

     //.....................................................................
     class UI_num_lr : public UI_num {                           // linear regression
     protected:
         double m_x;
         double m_n;
         double m_xsum;
         double m_xsum_sq;
         double m_ysum;
         double m_ysum_sq;
         double m_xysum;

     protected:
         static QState handler(UI_num_lr *me, QEvent const *e);
     };
     Q_ASSERT_COMPILE(sizeof(UI_num_lr) < sizeof(UI_mem));

     //.....................................................................
     class UI_help : public UI_top {
     protected:
         char const * const *m_help_text;
         uint16_t m_help_len;
         uint16_t m_help_line;

     protected:
         static QState handler(UI_help *me, QEvent const *e);
     };
     Q_ASSERT_COMPILE(sizeof(UI_help) < sizeof(UI_mem));

     //---------------------------------------------------------------------------
(17) extern QActive * const AO_UI;          // "opaque" pointer to UI Active Object

     #endif
```

Listing 1 Application header file `ui.h` for QP/C++.

(1) The top-level state-class `UI_top` of the UI state machine from Figure 6 inherits from `QActive`. This is the class that has all the responsibilities of the `QHsm` subclass in the standard QEP implementation.

(2) The top-level state defines the extended-state variable for storing the history of the "UI_num" substate (see "Transition to History" design pattern).

NOTE: The history variable must be defined at a higher level of hierarchy than the state that actually uses the history ("UI_num" in this case). This is because all the SLS variables for a given state disappear when this state is exited. Yet, the whole purpose of the history variable is to remember the last active substate after the superstate has been exited.

(3) The top-level state class must provide the usual constructor for the state machine.

(4) The top-level state class must provide the top-most initial pseudostate.

(5-6) The top-level state class provides all state-handler functions for states it represents.

(7) The UI_mem subclass of UI_top plays the role of the "container" for all state-classes comprising the UI state machine. The UI_mem class is the only one actually instantiated. All other state-classes are overlaid on top of the UI_mem instance. At this point it is important that the UI_mem class inherits from UI_top, because this guarantees that the UI_top() constructor is executed. This constructor correctly initializes the UI state machine.

(8) The UI_mem subclass size of the m_mem[] data member must be large enough, so that the UI_mem instance can indeed hold the largest state-class (typically the most derived substate).

NOTE: To make sure that none of the state-classes overruns the memory budget established by the UI_mem class, each state-class declaration is followed by a compile-time assertion.

(9) The UI_num class must derive from UI_top, because the "UI_num" is the substate of UI_top.

(10) The UI_num class contains the NumEntry state machine component ("Orthogonal Component" state pattern).

(11) The UI_num class declares its only state-handler function.

(12) The compile-time assertion makes sure that the UI_num state-class does not go over the memory budget established by the UI_mem class.

(13) The UI_num_sd state-class derives from UI_num, according to the state diagram.

(14) The UI_num_sd state-class declares State-Local Storage used only during the Standard Deviation computations.

(15) The UI_num_sd state-class declares its only state-handler function.

(16) The compile-time assertion makes sure that the UI_num_sd state-class does not go over the memory budget established by the UI_mem class.

(17) This global pointer represents the UI active object in the application and is used for posting events directly to the active object. The pointer is declared const, so that it can be placed in ROM. The active object pointer is "opaque", because it cannot access the whole active object, but only the part inherited from the QActive class.

4.2 The State Machine Implementation Modules

The implementation of the UI state machine can be sliced and diced into as many translation units as necessary. In the SLS example, the UI state machine is implemented in the following files: ui.cpp defines the UI_mem object and the "UI_top" state. The ui_num.cpp module defines the "UI_num" state and the "UI_num_sd" substate. The ui_nu_lr.cpp module defines the "UI_num_lr" state, as well as the "UI_num_lr::x_entry" and the "UI_num_lr::y_entry" substates. Finally, module ui_help.cpp implements the "UI_help" state.

The following Listing 2 shows the ui.cpp module, which defines the UI_mem object and the "UI_top" state.

```
#include "qp_port.h"                              // the port of the QP framework
#include "num_ent.h"
#include "ui.h"
#include "video.h"

// Local objects -------------------------------------------------------------
```

```
(1) static UI_mem l_ui;                     // instantiate the UI state machine...
                          // NOTE: this executes the ctor of the UI_top superstate

    // Global objects ----------------------------------------------------------
(2) QActive * const AO_UI = &l_ui;         // "opaque" pointer to UI Active Object

    // HSM definition ----------------------------------------------------------
(3) QState UI_top::initial(UI_top *me, QEvent const * /* e */) {
                                                        // subscribe to events...
        me->subscribe(QUIT_SIG);
                                                        // initialize the history
        me->m_history = (QStateHandler)&UI_num_sd::handler;

        return Q_TRAN(&UI_num::handler);
    }
    //.....................................................................
(4) QState UI_top::handler(UI_top *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                uint8_t c;
                Video::clearScreen(Video::BGND_BLACK);
                Video::clearRect( 0,  0, 80,  7, Video::BGND_LIGHT_GRAY);
                Video::clearRect( 0, 10, 80, 11, Video::BGND_LIGHT_GRAY);
                Video::clearRect( 0, 23, 80, 24, Video::BGND_LIGHT_GRAY);
                . . .
                Video::printStrAt(10, 23, Video::FGND_BLUE,
                  "* Copyright (c) Quantum Leaps, LLC * www.quantum-leaps.com *");
                Video::printStrAt(28, 24, Video::FGND_LIGHT_RED,
                    "<< Press Esc to quit >>");
                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                Video::clearScreen(Video::BGND_BLACK);      // clear the screen...
                return Q_HANDLED();
            }
            case QUIT_SIG: {
(5)             return Q_TRAN(&UI_top::final);
            }
        }
        return Q_SUPER(&QHsm::top);
    }
    //.....................................................................
    QState UI_top::final(UI_top *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                QF::stop();                             // stop QF and cleanup
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&QHsm::top);
    }
```

Listing 2 Module `ui.cpp`.

(1) The `UI_mem` object is allocated statically, which makes it inaccessible outside of the .CPP file. Instantiation of the `UI_mem` object causes invocation of all the superclass constructors, including the `UI_top()` constructor. This latter constructor is essential for correct initialization of the UI active object.

NOTE: In C++, the `UI_top()` constructor is invoked automatically as part of the C++ runtime startup before the `main()` function is invoked. In C, however, you need to invoke the `UI_top()` constructor manually, right at the beginning of `main()`.

(2) Externally, the UI active object is known only through the "opaque" pointer `AO_UI`. The pointer is declared 'const' (with the const after the '*'), which means that the pointer itself cannot change. This ensures that the active object pointer cannot change accidentally and also allows the compiler to allocate the active object pointer in ROM.

(3) The `UI_top::initial` transition is defined in the standard way for the QEP event processor.

(4) The state-handler functions are defined in the standard way for the QEP event processor.

(5) The initial transition (as well as all other transitions) uses the fully-qualified names of the target state, which includes the state-class name followed by state-handler function name.

NOTE: In this case the "me" pointer changes its type as the state machine transitions from state to state. QEP event processor handles these type changes automatically and transparently.

The following Listing 3 shows the "UI_num" state handler function in the module `ui_num.cpp`.

```
    #include "qp_port.h"                              // the port of the QP framework
    #include "num_ent.h"
    #include "ui.h"
    #include "video.h"

    #include <stdio.h>
    #include <math.h>

(1) #include <new.h>                                          // for placement new

    //.....................................................................
    QState UI_num::handler(UI_num *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                                              // instantiate the state-local objects
(2)             new(&me->m_num_entry) NumEntry();
                                            // send object dictionaries for UI objects
                QS_OBJ_DICTIONARY(&me->m_num_entry);

                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                                               // destroy the state-local objects...
(3)             me->m_num_entry.NumEntry::~NumEntry();
                return Q_HANDLED();
            }
            case Q_INIT_SIG: {
                                  // take the initial transition in the component
(4)             me->m_num_entry.init();

                return Q_TRAN(&UI_num_sd::handler);
            }
            case HELP_SIG: {
                return Q_TRAN(&UI_help::handler);                  // Help screen
            }
        }
        return Q_SUPER(&UI_top::handler);
    }
```

Listing 3 The "UI_num" state handler function in the module `ui_num.cpp`.

(1) The include file <new.h> is required for the placement-new used later in this module.

NOTE: The legacy Turbo C++ 1.01 compiler does not provide the standard <new.h> header file. A minimal version of the new.h file with placement-new operator definition is provided in the application directory.

(2) The state-handler function for the "UI_num" state is responsible for explicit instantiating of all State-Local variables, which includes invoking the constructor of the m_num_entry component. In C++, the only way to invoke the constructor explicitly is by means of placement new, as shown here.

NOTE: Invoking the constructor for all SLS variables that have non-trivial constructors is critical, because the memory used by the SLS variables is reused by other states. This means that the SLS variables for a given state are typically in an uninitialized (random) state when the state is entered.

(3) The state-handler function for the "UI_num" state is responsible for explicit finalization of all State-Local variables that go out of scope. This includes invoking the destructor of the m_num_entry component, which in C++ can be accomplished as shown here.

(4) The state-handler function for the "UI_num" state is responsible for triggering the initial transition in the num_entry component.

5 Sample Code for QP/C and QP-nano

The sample code for QP/C is found in the directory <qpc>\examples\80x86\dos\tcpp101\l\sls\ and code for QP-nano is located in the directory <qpn>\examples\80x86\tcpp101\sls\. The code has been compiled with the legacy Borland Turbo C++ 1.01 compiler available for free download from http://dn.codegear.com/article/21751. The example application contains the Turbo C++ project files SLS.PRJ and SLS-REL.PRJ for building the Debug and Release versions, respectively. The QP/C implementation contains additionally the SLS-SPY.PRJ for building the SPY version.

The C code has very similar structure as the C++ code, except that class inheritance is modeled with "derivation of structures", as described in Chapter 1 of [PSiCC2].

NOTE: The SLS implementation in QP/C and QP-nano are essentially identical. The small differences between the QP/C and QP-nano implementations have only to do with general differences between QP and QP-nano.

5.1 The Application Header File

The application header file is located in the file ui.h. This header file contains the elements that are shared across the whole application. For the SLS implementation, the application header file must expose the declarations of the higher-level state-classes, because they are necessary to derive the substate-classes. The explanation section following Listing 4 clarifies the main points of the application header file ui.h.

```
#ifndef ui_h
#define ui_h

. . .
```

```
    typedef struct KeyboardEvtTag {
        QEvent super;                                   /* derived from QEvent */
        uint8_t key_code;                                   /* code of the key */
    } KeyboardEvt;

    /*.....................................................................*/
    typedef struct UI_topTag {
(1)     QActive super;                                  /* derive from QActive */
(2)     QStateHandler history;
    } UI_top;

(3) void   UI_top_ctor(UI_top *me);
(4) QState UI_top_initial(UI_top *me, QEvent const *e);  /* initial pseudostate */
(5) QState UI_top_handler(UI_top *me, QEvent const *e);        /* state handler */
(6) QState UI_top_final  (UI_top *me, QEvent const *e);        /* state handler */

    typedef struct UI_memTag {
(7)     UI_top super;                                    /* derive from UI_top */
(8)     uint8_t mem[80];   /* maximum size of any substate (subclass) of UI_top */
    } UI_mem;

    /*.....................................................................*/
    typedef struct UI_numTag {
(9)     UI_top super;                                    /* derive from UI_top */
(10)    NumEntry num_entry;                             /* orthogonal component */
    } UI_num;

(11) QState UI_num_handler(UI_num *me, QEvent const *e);
(12) Q_ASSERT_COMPILE(sizeof(UI_num) < sizeof(UI_mem));

    /*.....................................................................*/
    typedef struct UI_num_sdTag {                        /* standard deviation */
(13)    UI_num super;                                    /* derive from UI_num */
(14)    double n;
        double sum;
        double sum_sq;
    } UI_num_sd;

(15) QState UI_num_sd_handler(UI_num_sd *me, QEvent const *e);
(16) Q_ASSERT_COMPILE(sizeof(UI_num_sd) < sizeof(UI_mem));

    /*.....................................................................*/
    typedef struct UI_num_lrTag {                         /* linear regression */
        UI_num super;                                   /* derive from UI_num */
        double x;
        double n;
        double xsum;
        double xsum_sq;
        double ysum;
        double ysum_sq;
        double xysum;
    } UI_num_lr;

    QState UI_num_lr_handler(UI_num_lr *me, QEvent const *e);
    Q_ASSERT_COMPILE(sizeof(UI_num_lr) < sizeof(UI_mem));

    /*.....................................................................*/
    typedef struct UI_helpTag {
        UI_top super;                                   /* derive from UI_top */
        char const * const *help_text;
        uint16_t help_len;
        uint16_t help_line;
    } UI_help;

    QState UI_help_handler(UI_help *me, QEvent const *e);
```

```
    Q_ASSERT_COMPILE(sizeof(UI_help) < sizeof(UI_mem));

    /*--------------------------------------------------------------------------*/
(17) extern QActive * const AO_UI;       /* "opaque" pointer to UI Active Object */

(18) void UI_ctor(void);                 /* global ctor for the UI active object */

    #endif
```

Listing 4 Application header file `ui.h` for QP/C.

(1) The top-level state-class `UI_top` of the UI state machine from Figure 6 inherits from `QActive`. This is the class that has all the responsibilities of the `QHsm` subclass in the standard QEP implementation.

(2) The top-level state defines the extended-state variable for storing the history of the "UI_num" substate (see "Transition to History" design pattern).

NOTE: The history variable must be defined at a higher level of hierarchy than the state that actually uses the history ("UI_num" in this case). This is because all the SLS variables for a given state disappear when this state is exited. Yet, the whole purpose of the history variable is to remember the last active substate after the superstate has been exited.

(3) The `UI_top` state class must provide the usual constructor for the state machine.

(4) The `UI_top` state class must provide the top-most initial pseudostate.

(5-6) The `UI_top` state class provides all state-handler functions for states it represents.

(7) The `UI_mem` subclass of `UI_top` plays the role of the "container" for all state-classes comprising the UI state machine. The `UI_mem` class is the only one actually instantiated. All other state-classes are overlaid on top of the `UI_mem` instance.

NOTE: In C, the UI state machine (active object) must be instantiated explicitly by invoking the constructor `UI_ctor()` at the beginning of `main()`. The prototype of the `UI_ctor()` function is declared at the end of the `ui.h` header file.

(8) The size of the `mem[]` data member must be large enough, so that the `UI_mem` instance can indeed hold the largest state-class (typically the most derived substate).

NOTE: To make sure that none of the state-classes overruns the memory budget established by the `UI_mem` class, each state-class declaration is followed by a compile-time assertion.

(9) The `UI_num` class must derive from `UI_top`, because the "UI_num" is the substate of `UI_top`.

(10) The `UI_num` class contains the `NumEntry` state machine component ("Orthogonal Component" state pattern).

(11) The `UI_num` class declares its only state-handler function.

(12) The compile-time assertion makes sure that the `UI_num` state-class does not go over the memory budget established by the `UI_mem` class.

(13) The `UI_num_sd` state-class derives from `UI_num`, according to the state diagram.

(14) The UI_num_sd state-class declares State-Local Storage used only during the Standard Deviation computations.

(15) The UI_num_sd state-class declares its only state-handler function.

(16) The compile-time assertion makes sure that the UI_num_sd state-class does not go over the memory budget established by the UI_mem class.

(17) This global pointer represents the UI active object in the application and is used for posting events directly to the active object. The pointer is declared const, so that it can be placed in ROM. The active object pointer is "opaque", because it cannot access the whole active object, but only the part inherited from the QActive class.

(18) In C, the UI state machine (active object) must be instantiated explicitly by invoking the constructor UI_ctor() at the beginning of main().

5.2 The State Machine Implementation Modules

The implementation of the UI state machine is partitioned into modules (.C files) identically as the C++ version. The following Listing 5 shows the ui.c module, which defines the UI_mem object and the "UI_top" state.

```
    #include "qp_port.h"                        /* the port of the QP framework */
    #include "num_ent.h"
    #include "ui.h"
    #include "video.h"

    /* Local objects -----------------------------------------------------------*/
(1) static UI_mem l_ui;            /* the instance of the UI "container" object */

    /* Global objects ----------------------------------------------------------*/
(2) QActive * const AO_UI = (QActive *)&l_ui;    /* opaque pointer to the UI AO */

    /*..........................................................................*/
    void UI_ctor(void) {                                 /* the global UI ctor */
        UI_top_ctor((UI_top *)&l_ui);
    }
    /*..........................................................................*/
    void UI_top_ctor(UI_top *me) {
        QActive_ctor(&me->super, (QStateHandler)&UI_top_initial);
    }

    /* HSM definition ----------------------------------------------------------*/
(3) QState UI_top_initial(UI_top *me, QEvent const *e) {
        (void)e;           /* avoid the compiler warning about unused parameter */

                                                    /* subscribe to events... */
        QActive_subscribe((QActive *)me, QUIT_SIG);

        me->history = (QStateHandler)&UI_num_sd_handler;  /* initialize history */

        return Q_TRAN(&UI_num_handler);
    }
    /*..........................................................................*/
(4) QState UI_top_handler(UI_top *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                uint8_t c;
                Video_clearScreen(VIDEO_BGND_BLACK);
                Video_clearRect( 0,  0, 80,  7, VIDEO_BGND_LIGHT_GRAY);
                . . .
```

```
                Video_printStrAt(10, 23, VIDEO_FGND_BLUE,
                  "* Copyright (c) Quantum Leaps, LLC * www.quantum-leaps.com *");
                Video_printStrAt(28, 24, VIDEO_FGND_LIGHT_RED,
                    "<< Press Esc to quit >>");

                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                Video_clearScreen(VIDEO_BGND_BLACK);     /* clear the screen... */
                return Q_HANDLED();
            }
            case QUIT_SIG: {
(5)             return Q_TRAN(&UI_top_final);
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*.....................................................................*/
    QState UI_top_final(UI_top *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                QF_stop();                               /* stop QF and cleanup */
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&QHsm_top);
    }
```

Listing 5 Module `ui.c`.

(1) The `UI_mem` object is allocated statically, which makes it inaccessible outside of the .C file. Instantiation of the `UI_mem` object causes invocation of all the superclass constructors, including the `UI_top()` constructor. This latter constructor is essential for correct initialization of the UI active object.

NOTE: In C, the UI_mem instance is initialized in the `UI_ctor()` constructor function, which must be invoked explicitly right at the beginning of `main()`.

(2) Externally, the UI active object is known only through the "opaque" pointer `AO_UI`. The pointer is declared 'const' (with the const after the '*'), which means that the pointer itself cannot change. This ensures that the active object pointer cannot change accidentally and also allows the compiler to allocate the active object pointer in ROM.

(3) The `UI_top_initial` transition is defined in the standard way for the QEP event processor.

(4) The state-handler functions are defined in the standard way for the QEP event processor.

(5) The initial transition (as well as all other transitions) uses the fully-qualified names of the target state, which includes the state-class name followed by state-handler function name.

NOTE: In this case the "me" pointer changes its type as the state machine transitions from state to state. QEP event processor handles these type changes automatically and transparently.

The following Listing 6 shows the "UI_num" state handler function in the module `ui_num.c`.

```
    #include "qp_port.h"                         /* the port of the QP framework */
    #include "num_ent.h"
```

```
    #include "ui.h"
    #include "video.h"

    #include <stdio.h>
    #include <math.h>

    /*..........................................................................*/
    QState UI_num_handler(UI_num *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                                        /* instantiate the state-local objects */
(1)             NumEntry_ctor(&me->num_entry);
                                      /* send object dictionaries for UI objects */
                QS_OBJ_DICTIONARY(&me->num_entry);
                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                                         /* destroy the state-local objects... */
(2)             NumEntry_xtor(&me->num_entry);
                return Q_HANDLED();
            }
            case Q_INIT_SIG: {
                                  /* take the initial transition in the component */
(3)             QHsm_init((QHsm *)&me->num_entry, (QEvent *)0);
                return Q_TRAN(&UI_num_sd_handler);
            }
            case HELP_SIG: {
                return Q_TRAN(&UI_help_handler);                /* Help screen */
            }
        }
        return Q_SUPER(&UI_top_handler);
    }
```

Listing 6 The "UI_num" state handler function in the module ui_num.c.

(1) The state-handler function for the "UI_num" state is responsible for explicit instantiating of all State-Local variables, which includes invoking the constructor of the num_entry component.

NOTE: Invoking the constructor for all SLS variables that have non-trivial constructors is critical, because the memory used by the SLS variables is reused by other states. This means that the SLS variables for a given state are typically in an uninitialized (random) state when the state is entered.

(2) The state-handler function for the "UI_num" state is responsible for explicit finalization of all State-Local variables that go out of scope. This includes invoking the destructor of the num_entry component.

(3) The state-handler function for the "UI_num" state is responsible for triggering the initial transition in the num_entry component.

6 The QSPY Software Tracing Instrumentation

The SLS examples for QP/C and QP/C++ demonstrate how to use the QS software tracing to generate real-time trace of a running QP application. Normally, the QS instrumentation is inactive and does not add any overhead to your application, but you can turn the instrumentation on by defining the `Q_SPY` macro and recompiling the code.

QS (Q-SPY) is a software tracing facility built into all QP components and also available to the application code. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS software tracing is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression (see Chapter 11 of [PSiCC2]).

State-Local Storage might complicate the use of the QS instrumentation. By nature, the SLS variables are overlaid in the same memory and chances are that more than one SLS variable can occupy exactly the same memory address (albeit at different times). This can lead to ambiguities in associating memory addresses to symbolic names in generation of the human-readable trace output.

Additionally, coding the object dictionary records (see Q-SPY description in Chapter 11 of [PSiCC2]) for SLS variable might be slightly more complicated. For example, to associate a symbolic name with the `NumEntry` orthogonal component, the embedded application must send the corresponding data dictionary record for the `num_entry` data member. However, the `num_entry` SLS variable is not available at the level of the `UI_top` state, which defines the top-most initial transition. The object dictionary record for the `num_entry` component can be generated only from the `UI_num` state-class. This means that this dictionary record will be generated every time the `UI_num` state is entered, not just once during the initial transient. Please consult the provided SLS example code for implementation details.

6.1 Invoking the QSPY Host Application

The QSPY host application receives the QS trace data, parses it and displays on the host workstation (currently Windows or Linux). For the configuration options chosen in this port, you invoke the QSPY host application as follows (please refer to the QSPY section in the “QP Reference Manual”):

```
qspy -c COM1 -b 115200
```

The following Figure 7 shows the human-readable output from the QSPY host application obtained from the SLS example code.

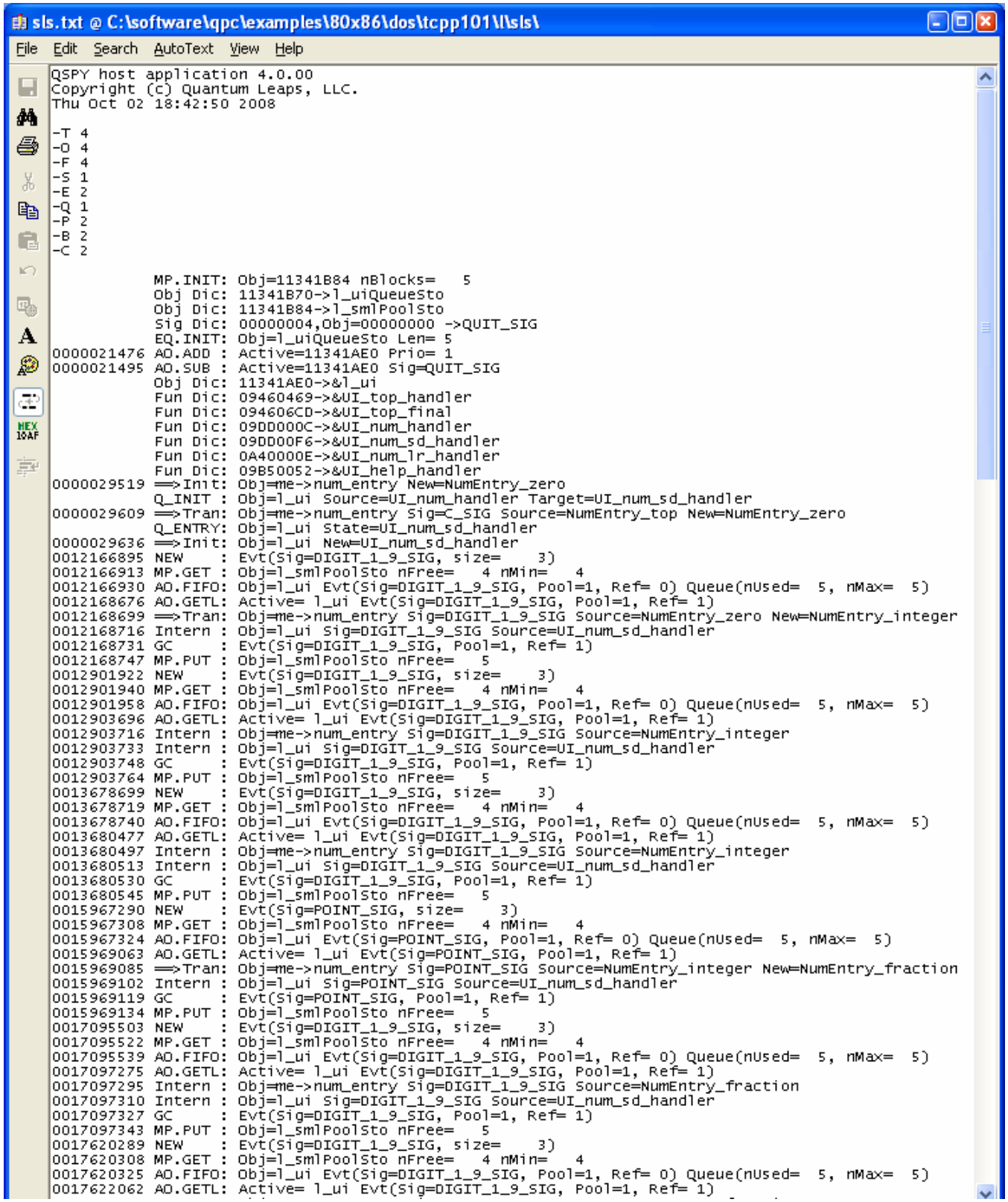


Figure 7 QSPY software trace output from the SLS example

7 Consequences

The State-Local Storage (SLS) state pattern has the following consequences.

- It defines the new **state-local scope** of variable lifetime. The lifetime begins upon the entry to a state and ends upon the exit from that state.
- The state-local variable lifetime scope allows reduction of the state machine footprint. The memory savings are proportional to the number of peer states that require different state-local variables.
- The hierarchy of the state-classes must be consistent with the state nesting. In the SLS implementation with the QEP event processor, the C or C++ compiler cannot check that the state-classes inheritance tree is indeed consistent with the state nesting.
- The state-handler functions are responsible for instantiating state-local variables in the entry actions and for finalizing them in the state exit actions.
- The QEP event processor handles automatically the memory overlaying during state transitions between any two hierarchically nested states.
- The QEP event processor also automatically changes the type of the "`me`" pointer passed to the various state-handler functions at different levels of the state hierarchy.
- State-local storage cannot work well with the fully UML-compliant transition execution sequence, which requires executing transition actions after exiting source state configuration, but before entering the target state configuration.
- The QEP event processor preserves the essential order of execution of exit actions from the source state configuration and entry to the target state configuration, but executes transition actions in the source context (before exiting the source). This execution sequence is much better for implementing SLS.

8 Known Uses

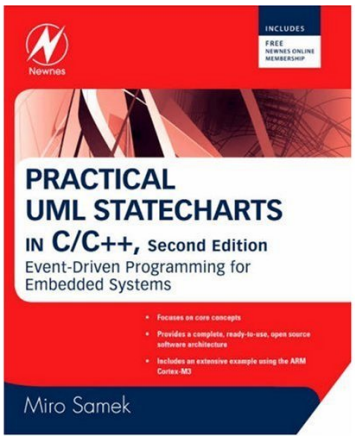
The Boost Statechart Library offers heavily templatized C++ implementation of UML statecharts that allows declaring state-local variables [Huber 07].

9 References

Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008, ISBN 0750687061	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QL DevZone 08] QP Developer's Zone, State Design Patterns, Quantum Leaps, LLC, 2008	http://www.state-machine.com/devzone/patterns.htm
[QP/C 08] "QP/C Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpc/
[QP/C++ 08] "QP/C++ Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpcpp/
[QP-nano 08] "QP-nano Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpn/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.state-machine.com/doc/-AN_QP_Directory_Structure.pdf
[Huber 07] Boost Statechart Library, Andreas Huber Dönni, 2007.	http://www.boost.org/doc/libs/1_36_0/libs/-statechart/doc/index.html

10 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA
+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)
e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com



"Practical UML State-charts in C/C++, Second Edition" (**PSiCC2**), by Miro Samek, Newnes, 2008, ISBN 0750687061