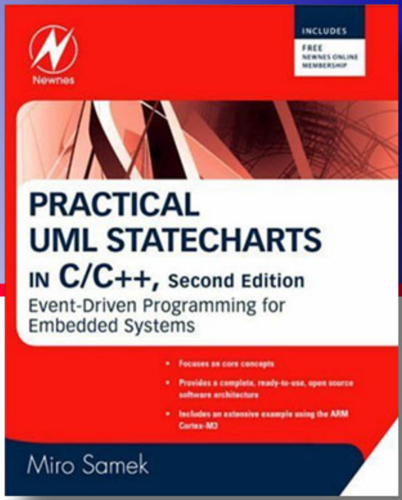
innovating embedded systems



Application Note

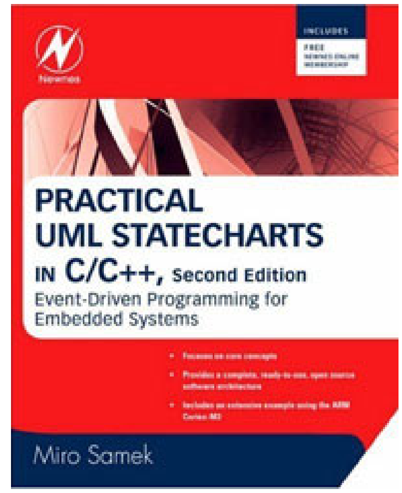
Inheriting State Machines with QP™ 4.x

Document Revision C
September 2009

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

Table of Contents

1 Introduction



Traditional Object-Oriented Programming (OOP) prescribes how to inherit attributes and refine individual class methods (virtual functions in C++) to use polymorphism. But how do you inherit entire state machines?

The issue is tricky, because a state machine is a system of interrelated states and transitions rather than a just a group of virtual functions. The challenge is to keep intact the numerous relationships among the hierarchical states and transitions in the process of inheriting and refining the derived state machine. The relations among state machine elements come in two flavors: (1) states refer to other states as superstates, and (2) transitions refer to states as targets of the transition. The challenge is not to break these relationships in the derived state machines.

As described in Chapter 4 of "*Practical UML Statecharts in C/C++, Second Edition*" [PSiCC2], the representation of state handler functions in QP/C++ 4.x has changed compared to the version 2.x published in the book "*Practical Statecharts in C/C++*" [PSiCC1]. This change impacts the rules for inheriting entire state machines, so the guidelines described in Chapter 6 of *Practical Statecharts in C/C++* do not apply in QP/C++ 4.x. This Application Note addresses this issue by describing how to inherit state machines with the new QP 4.x.

NOTE: This Application Note uses the C++ code for illustrating the concepts, because most likely the C++ version will be extended via inheritance. However, the accompanying code to this Application Note contains also the C code. The differences between C and C++ are only syntactical, and both versions rely essentially on the same techniques for adapting the state models for inheritance.

This Application Note explains two approaches to inheriting state machines. The first, much safer approach, is not to change the state machine structure at all. In this case, the complete state machine topology (the network of hierarchical states and transitions) is completely defined in the state machine superclass. The subclasses inherit this state machine structure and override only the actions and guard conditions that are declared as virtual functions in the base class. This way of inheriting state machines is an example of the widely used **Template Method** design pattern described in the "Design Patterns: Reusable Elements of Object-Oriented Software" book [GoF 95]. This approach is illustrated by the inheritance of Mine state machines from the "Fly 'n' Shoot" game described in Chapter 1 of [PSiCC2].

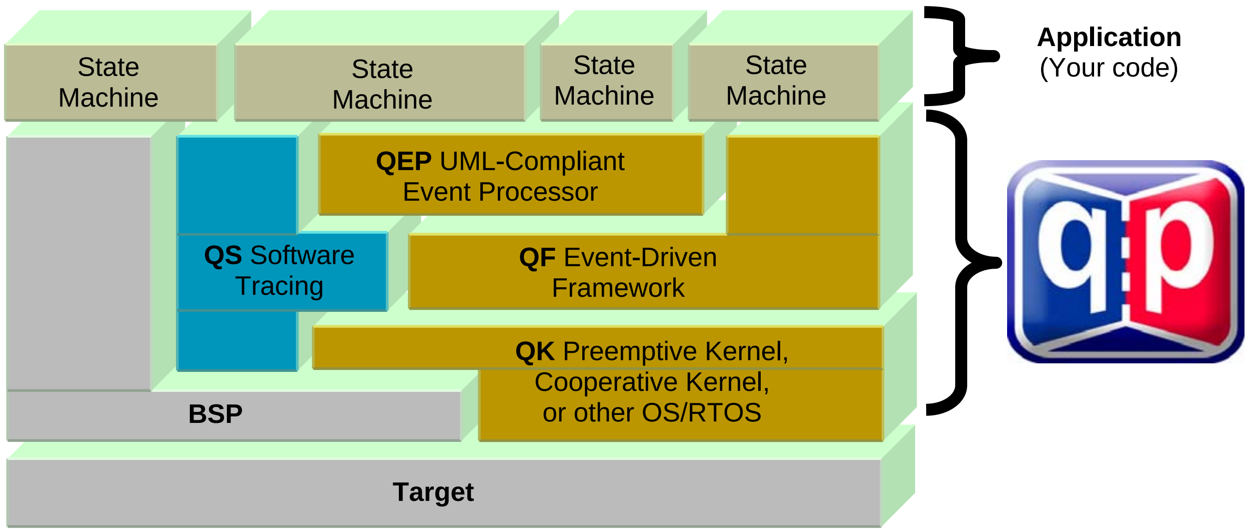
The second method of inheriting state machines allows changing the structure of the inherited state machine by adding states and transitions. This **Subtyping Method** is illustrated by the refinement of the Calculator state machine described in Chapter 3 of [PSiCC2].

1.1 About QP™

QP™ is a family of very lightweight, open source, state machine-based frameworks for developing event-driven applications. QP enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (UML statecharts) directly in C or C++ **without big tools**. QP is described in great detail in the book "*Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*" [PSiCC2] (Newnes, 2008).

As shown in Figure 1, QP consists of a universal UML-compliant event processor (QEP), a portable real-time framework (QF), a tiny run-to-completion kernel (QK), and software tracing instrumentation (QS). Current versions of QP include: QP/C™ and QP/C++™, which require about 4KB of code and a few hundred bytes of RAM, and the ultra-lightweight QP-nano, which requires only 1-2KB of code and just several bytes of RAM. The Linux port described in this Application Note pertains to QP/C and QP/C++.

Figure 1 QP components and their relationship with the target hardware, board support package (BSP), and the application



QP can work with or without a traditional RTOS or OS. In the simplest configuration, QP can completely **replace** a traditional RTOS. QP can manage up to 63 concurrently executing tasks structured as state machines (called active objects in UML).

1.2 Licensing QP™

The **Generally Available (GA)** distribution of QP™ available for download from the www.state-machine.com/downloads website is offered with the following two licensing options:

- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file `GPL.TXT` included in the packaging of every Quantum Leaps software distribution. The GPL *open source* license allows you to use the software at no charge under the condition that if you redistribute the original software or applications derived from it, the complete source code for your application must be also available under the conditions of the GPL (GPL Section 2[b]).

- One of several Quantum Leaps commercial licenses, which are designed for customers who wish to retain the proprietary status of their code and therefore cannot use the GNU General Public License. The customers who license Quantum Leaps software under the commercial licenses do not use the software under the GPL and therefore are not subject to any of its terms.

For more information, please visit the licensing section of our website at: www.state-machine.com/licensing.

2 Changes between QP v2.x and QP 4.x

The representation of state handler functions in QP/C++ 4.x has changed compared to the version 2.x published in the book *Practical Statecharts in C/C++* [PSiCC1]. The state handler functions are no longer true member functions of the `QHsm` subclasses, but rather simply **static** member functions. The following table contrasts the two approaches:

QP/C++ 2.x ([PSiCC1])	QP/C++ 4.x ([PSiCC2])

<pre>class Calc : public QHsm {
protected:
    void initial(QEvent const *e);
 	
    QSTATE calc(QEvent const *e);
 	
    QSTATE ready(QEvent const *e);
 	
    QSTATE result(QEvent const *e);
 	
    . . .
};</pre>
	<pre>class Calc : public QHsm {
	protected:
	    static QState initial(**Calc *me**,
	                           QEvent const *e);
	    static QState calc(**Calc *me**,
	                           QEvent const *e);
	    static QState ready(**Calc *me**,
	                           QEvent const *e);
	    static QState result(**Calc *me**,
	                           QEvent const *e);
	    . . .
	};</pre>

<pre>QSTATE Calc::result(QEvent const *e)
 	
{
    switch (e->sig) {
        case Q_ENTRY_SIG:
            dispState("result");
            eval();
            return (QSTATE)0;
        }
    }
    return (QSTATE)&Calc::ready;
}</pre>
	<pre>QState Calc::result(**Calc *me**,
	                    QEvent const *e)
	{
	    switch (e->sig) {
	        case Q_ENTRY_SIG: {
	            me->dispState("result");
	            me->eval();
	            return Q_HANDLED();
	        }
	    }
	    return Q_SUPER(&Calc::ready);
	}</pre>

As you can see, the new approach (QP/C++ 4.x) uses static state handlers and emulates the "this" calling convention by explicitly providing the "me" pointer (just like the QP/C version). As shown in the bottom part of the table, in the state handler definition you use the "me" pointer to access the true members of the derived state machine (`Calc` in this case).

The new approach is without a doubt less elegant than the old one. Conceptually, state handlers **are** members of the state machine class and they should be coded as such.

2.1 The Rationale

However — and here is where the rubber hits the road — the users of the earlier versions of the QP/C++ have filed too many alarming reports from the trenches where the "elegant" approach either had very lousy performance, or did not work altogether. For example, some compilers used over 30 machine instructions to de-reference a pointer-to-member-function and only 3 to de-reference a regular pointer-to-function. Needless to say, 3 machine instructions should do the job (see also Chapter 3 of [PSiCC2]).

As it turns out, too many C++ compilers, especially in the embedded C++ cross-compilers, simply don't support pointers-to-member-functions well. As explained in the sidebar on page 75 of [PSiCC1], other C++ features such polymorphism and multiple inheritance, compound the complexity of pointers-to-member-functions. Pointers-to-member-functions seem often to be just an afterthought implemented very inefficiently in the compiler, if at all. To avoid inefficiencies and portability issues, QP/C++ 4.x does **not** to use pointers-to-member-functions, but simply static methods that don't use the "this" calling convention.

3 Template Method Example

The illustration of the Template Method is based on the “Fly ‘n’ Shoot” game example described in Chapter 1 of [PSiCC2]. The “Fly ‘n’ Shoot” game offers a meaningful opportunity for reusing the Mine behavior. As you recall, the game has two types of Mines (Mine1 and Mine2), which behave similarly, but not quite the same. As it turns out, both Mine state machines have exactly the same structure and differ only by the actions executed by the state machines. This is exactly the situation you can address with the Template Method.

NOTE: The code for the Template Method is located in the directory:
`<qpcpp>\examples\80x86\dos\ tcpp101\l\game2\.`

3.1 The Mine Superclass Design

Figure 2 shows the hierarchy of the state machine classes. The Mine state machine base class captures the common behavior of mines. This base class uses virtual functions for actions and guard conditions that are dependent on the type of the mine. These virtual functions are called in the state machine of the Mine superclass. The subclasses of Mine, such as Mine1 and Mine2, override the virtual functions and thus provide different behavior.

Figure 2 The Mine state machine abstract state machine class and its two subclasses

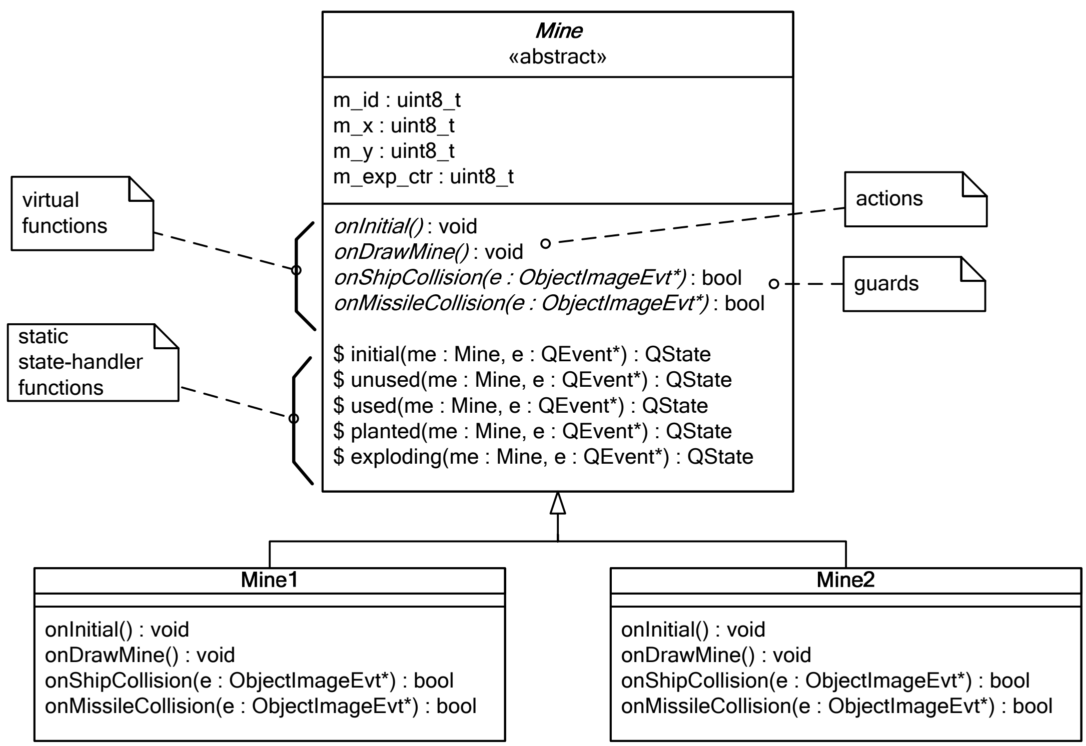
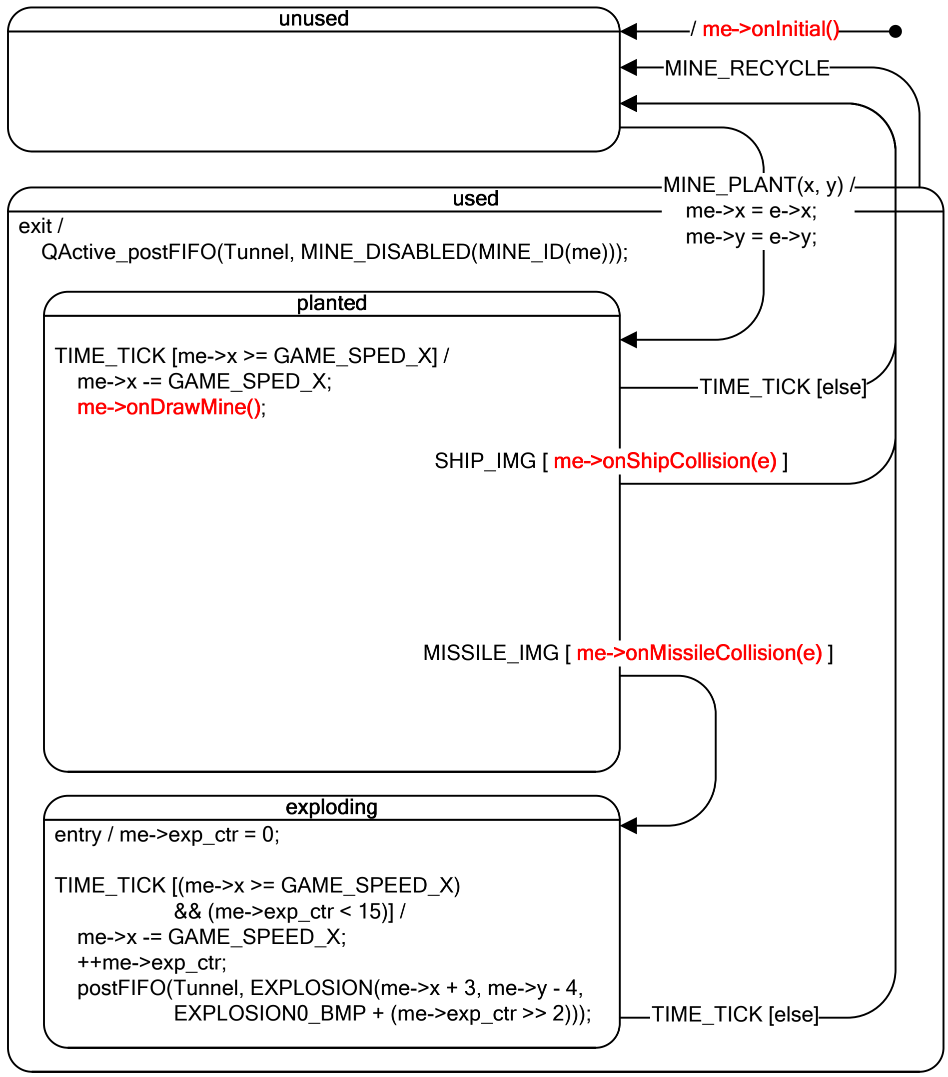


Figure 3 shows the state machine of the Mine base class, which calls the virtual actions and guard conditions.

Figure 3 Mine state machine that calls the virtual actions and guards (shown in bold and red)



3.2 The Mine Superclass Implementation

Listing 1 shows the declaration of the `Mine` state machine class. The declaration is exactly the same as any other hierarchical state machine class in QP, except that the Mine class declares a few **virtual functions** to be overridden by the sublcasses. In fact these are pure-virtual functions, because the Mine base class does not provide any implementation for these functions.

Listing 1 Declaration of the Mine state machine base class for inheritance (file mine.h)

```
#ifndef mine_h
#define mine_h
```



```
class Mine : public QHsm {                              // extend the QHsm class
protected:
    uint8_t m_id;
    uint8_t m_x;
    uint8_t m_y;
    uint8_t m_exp_ctr;

public:
    Mine(uint8_t id) : QHsm((QStateHandler)&Mine::initial), m_id(id) {}

protected:
    virtual void onInitial(void) = 0;
    virtual void onDrawMine(void) = 0;
    virtual uint8_t onShipCollision(ObjectImageEvt const *e) = 0;
    virtual uint8_t onMissileCollision(ObjectImageEvt const *e) = 0;

private:
    static QState initial  (Mine *me, QEvent const *e);
    static QState unused   (Mine *me, QEvent const *e);
    static QState used     (Mine *me, QEvent const *e);
    static QState planted  (Mine *me, QEvent const *e);
    static QState exploding(Mine *me, QEvent const *e);
};

#endif                                                               // mine_h
```

Listing 2 shows the definition of the `Mine` state machine class. The highlighted code corresponds to the invocations of the virtual actions and guards. When you compare the `Mine` state machine implementation with the original implementation of `Mine1` and `Mine2` from Chapter 1 of [PSiCC2] book, you will see that the `Mine` state machine code is nearly identical except that the few virtual calls simply encapsulate those actions or guards that depend on the mine type.

NOTE: A call of the type: `me->foo()` is subject to **late-binding** (polymorphism).

Listing 2 Definition of the Mine state machine (file mine.cpp)

```
#include "qp_port.h"
#include "bsp.h"
#include "game.h"
#include "mine.h"

Q_DEFINE_THIS_FILE

//............................................................................
QState Mine::initial(Mine *me, QEvent const *) {
    . . .
    me->onInitial();              // customized initialization for subclasses
    return Q_TRAN(&Mine::unused);
}
//............................................................................
QState Mine::unused(Mine *me, QEvent const *e) {
    switch (e->sig) {
        case MINE_PLANT_SIG: {
```

```
            me->m_x = ((ObjectPosEvt const *)e)->x;
            me->m_y = ((ObjectPosEvt const *)e)->y;
            return Q_TRAN(&Mine::planted);
        }
    }
    return Q_SUPER(&QHsm::top);
}
//............................................................................
QState Mine::used(Mine *me, QEvent const *e) {
    switch (e->sig) {
        case Q_EXIT_SIG: {
            // tell the Tunnel that this mine is becoming disabled
            MineEvt *mev = Q_NEW(MineEvt, MINE_DISABLED_SIG);
            mev->id = me->m_id;
            AO_Tunnel->postFIFO(mev);
            return Q_HANDLED();
        }
        case MINE_RECYCLE_SIG: {
            return Q_TRAN(&Mine::unused);
        }
    }
    return Q_SUPER(&QHsm::top);
}
//............................................................................
QState Mine::planted(Mine *me, QEvent const *e) {
    uint8_t x;
    uint8_t y;
    uint8_t bmp;

    switch (e->sig) {
        case TIME_TICK_SIG: {
            if (me->m_x >= GAME_SPEED_X) {
                me->m_x -= GAME_SPEED_X;                // move the mine 1 step
                                         // tell the Tunnel to draw the Mine
                me->onDrawMine();                      // customized in subclasses
            }
            else {
                return Q_TRAN(&Mine::unused);
            }
            return Q_HANDLED();
        }
        case SHIP_IMG_SIG: {
            if (me->onShipCollision((ObjectImageEvt const *)e)) {
                // go straight to 'disabled' and let the Ship do the exploding
                return Q_TRAN(&Mine::unused);
            }
            return Q_HANDLED();
        }
        case MISSILE_IMG_SIG: {
            if (me->onMissileCollision((ObjectImageEvt const *)e)) {
                return Q_TRAN(&Mine::exploding);
            }
            return Q_HANDLED();
        }
    }
    return Q_SUPER(&Mine::used);
```

```
}
//............................................................................
QState Mine::exploding(Mine *me, QEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            me->m_exp_ctr = 0;
            return Q_HANDLED();
        }
        case TIME_TICK_SIG: {
            if ((me->m_x >= GAME_SPEED_X) && (me->m_exp_ctr < 15)) {
                ObjectImageEvt *oie;

                ++me->m_exp_ctr;              // advance the explosion counter
                me->m_x -= GAME_SPEED_X;            // move explosion by 1 step

                // tell the Game to render the current stage of Explosion
                oie = Q_NEW(ObjectImageEvt, EXPLOSION_SIG);
                oie->x   = me->m_x + 1;                       // x of explosion
                oie->y   = (int8_t)((int)me->m_y - 4 + 2);    // y of explosion
                oie->bmp = EXPLOSION0_BMP + (me->m_exp_ctr >> 2);
                AO_Tunnel->postFIFO(oie);
            }
            else {
                return Q_TRAN(&Mine::unused);
            }
            return Q_HANDLED();
        }
    }
    return Q_SUPER(&Mine::used);
}
```

3.3 Subclassing The Mine Superclass

In the Template Method of inheriting state machines the subclasses of the Mine state machine class do not define any additional states or transitions, but rather simply specialize the virtual functions declared in the superclass. The following Listing 3 and Listing 4 show the declarations/definitions of the Mine1 and Mine2 subclasses, respectively.

NOTE: The subclasses `Mine1` and `Mine2` capture only the differences from the common superclass `Mine`.

Listing 3 Declaration/Definition of the Mine1 state machine subclass (file mine1.cpp)

```
    #include "qp_port.h"
#include "bsp.h"
#include "game.h"
#include "mine.h"

Q_DEFINE_THIS_FILE

// local objects -------------------------------------------------------------
class Mine1 : public Mine {                                 // extend the Mine class
public:
```

```
    Mine1(void);

protected:
    virtual void onInitial(void);
    virtual void onDrawMine(void);
    virtual uint8_t onShipCollision(ObjectImageEvt const *e);
    virtual uint8_t onMissileCollision(ObjectImageEvt const *e);
};

static Mine1 l_mine1[GAME_MINES_MAX];                // a pool of type-1 mines

                                // helper macro to provide the ID of this mine
#define MINE_ID(me_)    ((me_) - l_mine1)

//............................................................................
QHsm *Mine1_getInst(uint8_t id) {
    Q_REQUIRE(id < GAME_MINES_MAX);
    return &l_mine1[id];
}
//............................................................................
Mine1::Mine1(void) : Mine(MINE_ID(this)) {                           // the ctor
}
//............................................................................
void Mine1::onInitial() {
    . . .
}
//............................................................................
void Mine1::onDrawMine(void) {
    ObjectImageEvt *oie = Q_NEW(ObjectImageEvt, MINE_IMG_SIG);
    oie->x   = m_x;
    oie->y   = m_y;
    oie->bmp = MINE1_BMP;
    AO_Tunnel->postFIFO(oie);
}
//............................................................................
uint8_t Mine1::onShipCollision(ObjectImageEvt const *e) {
    uint8_t x   = (uint8_t)e->x;
    uint8_t y   = (uint8_t)e->y;
    uint8_t bmp = (uint8_t)e->bmp;

    // test for incoming Ship hitting this mine
    if (do_bitmaps_overlap(MINE1_BMP, m_x, m_y, bmp, x, y)) {
                                         // Hit event with the type of the Mine1
        static MineEvt const mine1_hit(HIT_MINE_SIG, 1);
        AO_Ship->postFIFO(&mine1_hit);

        return 1;                             // report collision with the Ship
    }
    else {
        return 0;                                 // no collision with the Ship
    }
}
//............................................................................
uint8_t Mine1::onMissileCollision(ObjectImageEvt const *e) {
    uint8_t x   = (uint8_t)e->x;
    uint8_t y   = (uint8_t)e->y;
```



```
    uint8_t bmp = (uint8_t)e->bmp;

    // test for incoming Missile hitting this mine
    if (do_bitmaps_overlap(MINE1_BMP, m_x, m_y, bmp, x, y)) {
                            // Score event with the score for destroying Mine1
        static ScoreEvt const mine1_destroyed(DESTROYED_MINE_SIG, 25);
        AO_Missile->postFIFO(&mine1_destroyed);
        return 1;                              // report collision with the Missile
    }
    else {
        return 0;                                  // no collision with the Missile
    }
}
```

Listing 4 Declaration/Definition of the Mine2 state machine subclass (file mine2.cpp)

```
#include "qp_port.h"
#include "bsp.h"
#include "game.h"
#include "mine.h"

Q_DEFINE_THIS_FILE

// local objects ---------------------------------------------------------------
class Mine2 : public Mine {                                // extend the Mine class
public:
    Mine2(void);

protected:
    virtual void onInitial(void);
    virtual void onDrawMine(void);
    virtual uint8_t onShipCollision(ObjectImageEvt const *e);
    virtual uint8_t onMissileCollision(ObjectImageEvt const *e);
};

static Mine2 l_mine2[GAME_MINES_MAX];                      // a pool of type-2 mines

                                // helper macro to provide the ID of this mine
#define MINE_ID(me_)    ((me_) - l_mine2)

//..............................................................................
QHsm *Mine2_getInst(uint8_t id) {
    Q_REQUIRE(id < GAME_MINES_MAX);
    return &l_mine2[id];
}
//..............................................................................
Mine2::Mine2(void) : Mine(MINE_ID(this)) {                                // the ctor
}
//..............................................................................
void Mine2::onInitial() {
    . . .
}
//..............................................................................
void Mine2::onDrawMine(void) {
    ObjectImageEvt *oie = Q_NEW(ObjectImageEvt, MINE_IMG_SIG);
```

```
    oie->x   = m_x;
    oie->y   = m_y;
    oie->bmp = MINE2_BMP;
    AO_Tunnel->postFIFO(oie);
}
//............................................................................
uint8_t Mine2::onShipCollision(ObjectImageEvt const *e) {
    uint8_t x   = (uint8_t)e->x;
    uint8_t y   = (uint8_t)e->y;
    uint8_t bmp = (uint8_t)e->bmp;

    // test for incoming Ship hitting this mine
    if (do_bitmaps_overlap(MINE2_BMP, m_x, m_y, bmp, x, y)) {
                                   // Hit event with the type of the Mine1
        static MineEvt const mine2_hit(HIT_MINE_SIG, 2);
        AO_Ship->postFIFO(&mine2_hit);
        return 1;                          // report collision with the Ship
    }
    else {
        return 0;                              // no collision with the Ship
    }
}
//............................................................................
uint8_t Mine2::onMissileCollision(ObjectImageEvt const *e) {
    uint8_t x   = (uint8_t)e->x;
    uint8_t y   = (uint8_t)e->y;
    uint8_t bmp = (uint8_t)e->bmp;

    // test for incoming Missile hitting this mine
    // NOTE: Mine type-2 is nastier than Mine type-1.
    // The type-2 mine can hit the Ship with any of its
    // "tentacles". However, it can be destroyed by the
    // Missile only by hitting its center, defined as
    // a smaller bitmap MINE2_MISSILE_BMP.
    //
    if (do_bitmaps_overlap(MINE2_MISSILE_BMP, m_x, m_y, bmp, x, y)) {
                        // Score event with the score for destroying Mine2
        static ScoreEvt const mine2_destroyed(DESTROYED_MINE_SIG, 45);
        AO_Missile->postFIFO(&mine2_destroyed);

        return 1;                       // report collision with the Missile
    }
    else {
        return 0;                           // no collision with the Missile
    }
}
```

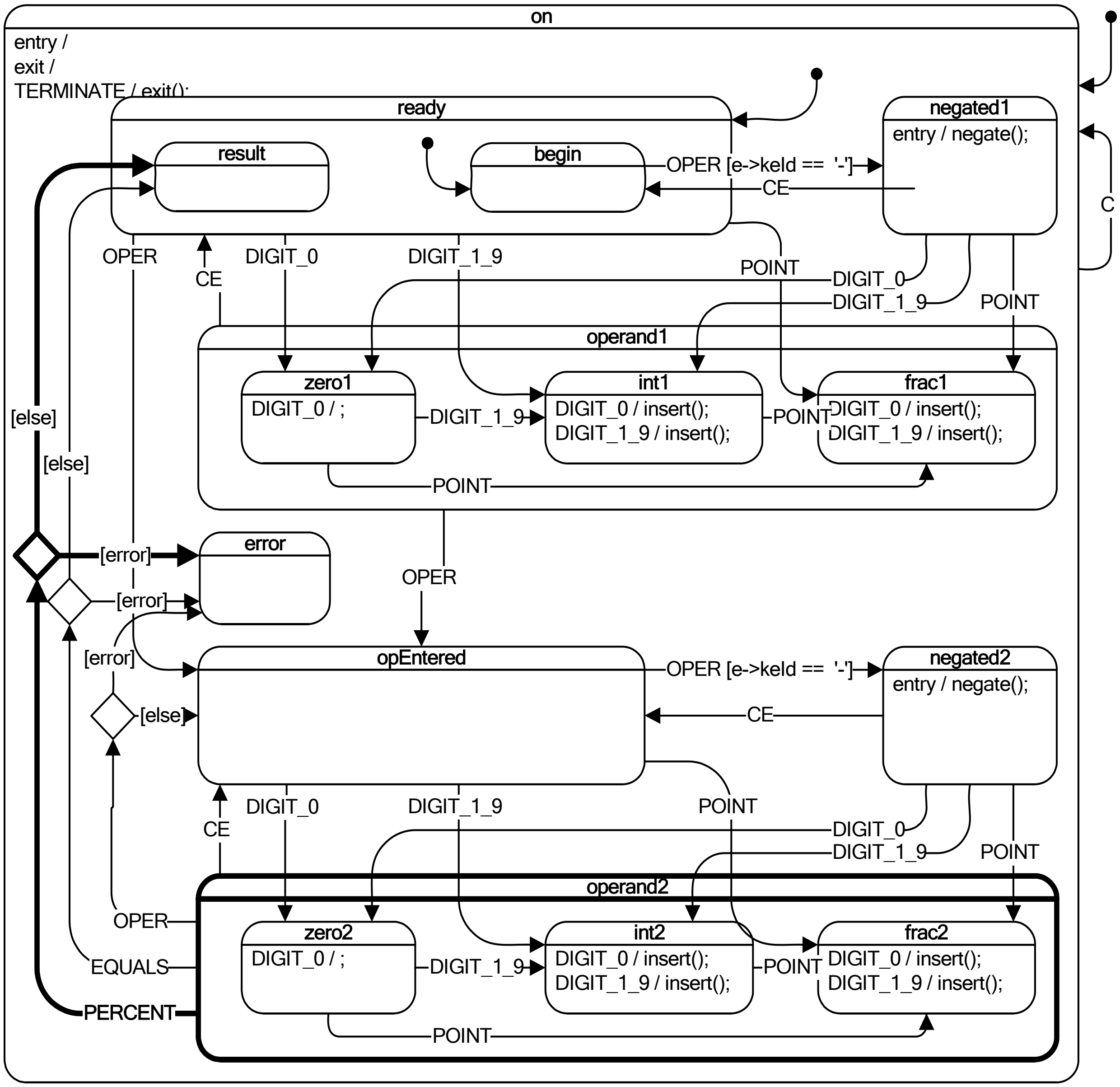
3.4 Testing the Code

The executable for the example is found in the directory `<qpcpp>\examples\80x86\dos\tcpp101\l\game2\dbg\game.exe`. This version of the "Fly 'n' Shoot" game should behave exactly as the original described in Chapter 1 of [PSiCC2].

4 Subtyping Method Example

The illustration of the Subtyping Method is based on the Calculator example described in Chapter 3 of [PSiCC2]. The basic Calculator state machine is subtyped to add percentage calculations, which requires overriding one state and adding one transition, as shown in Figure 4.

Figure 4 Refined Calc2 statechart. The added/refined elements are shown in bold



The refined calculator shown in Figure 4 "knows" how to handle the percentage calculations of the form x + y% gives z (e.g., price + sales tax gives total), where the '+' operator can be replaced by '-', '*', or '÷'. The Calc2 statechart illustrates a nontrivial refinement to the original Calc (see Chapter 2 in [PSiCC2]), because it involves refining an existing state "operand2" by adding to it a transition. The problem is that the state "oper-and2" is already involved in many relationships. For example, it is the superstate of "zero2", "int2", and "frac2", as well as the source of transitions triggered by signals EQUALS, OPER, and CE. The question is: Can you override just the "operand2" state handler without breaking all the relationships in which it already takes part?

4.1 Calc1—Preparing a State Machine Class for Inheritance

The state machine structure coded with fixed addresses of static state-handler functions is not flexible enough to allow for overriding selected state-handler functions in the subclasses. To allow the needed flexibility, the state machine must be coded with addresses of state handler functions that can be modified.

Every state machine class that you intend to extend via inheritance must be prepared to be inherited. The preparation means that it will be declared and coded slightly differently, to allow greater flexibility in overriding state machine elements.

4.1.1 Declaring the Base Calc1 State Machine Class

Listing 5 shows `Calc1` class, which is a modified version of the original `Calc` class described in Chapter 2 in [PSiCC2]. The Calc1 class is prepared for inheritance by adding static "state-variables" shown in bold. Additionally, data members are declared as protected instead of private, so that they are accessible in the subclasses of `Calc1`.

NOTE: The state machine base class can also declare virtual functions to be uses as actions within the state machine. The late-binding mechanism works in the subclasses because the static state-handler functions invoke the class methods via the "`me`" pointer.

Listing 5 Declaring Calc1 state machine base class for inheritance.

```
#ifndef calc1_h
#define calc1_h

enum Calc1Signals {
    C_SIG = Q_USER_SIG,
    CE_SIG,
    DIGIT_0_SIG,
    DIGIT_1_9_SIG,
    POINT_SIG,
    OPER_SIG,
    EQUALS_SIG,
    OFF_SIG,

    MAX_CALC1_SIG              // offset for adding signals used in subclasses
};

struct CalcEvt : public QEvent {
    uint8_t key_code;                                        // code of the key
};

// Calculator HSM class for inheritance -------------------------------------
class Calc1 : public QHsm {
protected:
    double  m_operand1;                                 // the value of operand 1
```



```
    uint8_t m_operator;                                    // operator key entered

public:
    Calc1(void) : QHsm((QStateHandler)&Calc1::initial) {}              // ctor

protected:
    static QState initial(Calc1 *me, QEvent const *e);  // initial pseudostate

    static QState on       (Calc1 *me, QEvent const *e);      // state-handler
    static QState error    (Calc1 *me, QEvent const *e);
    static QState ready    (Calc1 *me, QEvent const *e);
    static QState result   (Calc1 *me, QEvent const *e);
    static QState begin    (Calc1 *me, QEvent const *e);
    static QState negated1 (Calc1 *me, QEvent const *e);
    static QState operand1 (Calc1 *me, QEvent const *e);
    static QState zero1    (Calc1 *me, QEvent const *e);
    static QState int1     (Calc1 *me, QEvent const *e);
    static QState frac1    (Calc1 *me, QEvent const *e);
    static QState opEntered(Calc1 *me, QEvent const *e);
    static QState negated2 (Calc1 *me, QEvent const *e);
    static QState operand2 (Calc1 *me, QEvent const *e);
    static QState zero2    (Calc1 *me, QEvent const *e);
    static QState int2     (Calc1 *me, QEvent const *e);
    static QState frac2    (Calc1 *me, QEvent const *e);
    static QState final    (Calc1 *me, QEvent const *e);

    static QStateHandler   state_on;                          // state-variable
    static QStateHandler   state_error;
    static QStateHandler   state_ready;
    static QStateHandler   state_result;
    static QStateHandler   state_begin;
    static QStateHandler   state_negated1;
    static QStateHandler   state_operand1;
    static QStateHandler   state_zero1;
    static QStateHandler   state_int1;
    static QStateHandler   state_frac1;
    static QStateHandler   state_opEntered;
    static QStateHandler   state_negated2;
    static QStateHandler   state_operand2;
    static QStateHandler   state_zero2;
    static QStateHandler   state_int2;
    static QStateHandler   state_frac2;
    static QStateHandler   state_final;
};

#endif                                                              // calc1_h
```

NOTE: The `Calc1` state machine in Listing 5 provides "state-variables" for all its states. However, you can choose to provide "state-variables" only for states intended for overriding in the subclasses.

4.1.2 Defining the Base Calc1 State Machine Class

Listing 6 shows the definition of the `Calc1` base class. At the top of the listing, you see the definitions and initialization of the static state-variables. Subsequently, you don't hard-code the ad-dresses of state handler methods in the initial transitions, regular transitions, or return statements in the state handlers. That way, all these address (targets of various transitions and superstates) can be changed by modifying the state-variables, such as the pointer to function `Calc1::state_operand2`, for example.

Listing 6 Definition of the Calc1 state machine base class

```
#include "qp_port.h"                          // the port of the QP framework
#include "bsp.h"                                     // board support package
#include "calc1.h"

// state variables -----------------------------------------------------------
QStateHandler Calc1::state_on        = (QStateHandler)&Calc1::on;
QStateHandler Calc1::state_error     = (QStateHandler)&Calc1::error;
QStateHandler Calc1::state_ready     = (QStateHandler)&Calc1::ready;
QStateHandler Calc1::state_result    = (QStateHandler)&Calc1::result;
QStateHandler Calc1::state_begin     = (QStateHandler)&Calc1::begin;
QStateHandler Calc1::state_negated1  = (QStateHandler)&Calc1::negated1;
QStateHandler Calc1::state_operand1  = (QStateHandler)&Calc1::operand1;
QStateHandler Calc1::state_zero1     = (QStateHandler)&Calc1::zero1;
QStateHandler Calc1::state_int1      = (QStateHandler)&Calc1::int1;
QStateHandler Calc1::state_frac1     = (QStateHandler)&Calc1::frac1;
QStateHandler Calc1::state_opEntered = (QStateHandler)&Calc1::opEntered;
QStateHandler Calc1::state_negated2  = (QStateHandler)&Calc1::negated2;
QStateHandler Calc1::state_operand2  = (QStateHandler)&Calc1::operand2;
QStateHandler Calc1::state_zero2     = (QStateHandler)&Calc1::zero2;
QStateHandler Calc1::state_int2      = (QStateHandler)&Calc1::int2;
QStateHandler Calc1::state_frac2     = (QStateHandler)&Calc1::frac2;
QStateHandler Calc1::state_final     = (QStateHandler)&Calc1::final;

// HSM definition ------------------------------------------------------------
QState Calc1::initial(Calc1 *me, QEvent const * /* e */) {
    BSP_clear();
    return Q_TRAN(state_on);
}
//............................................................................
QState Calc1::operand2(Calc1 *me, QEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            BSP_message("operand2-ENTRY;");
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            BSP_message("operand2-EXIT;");
            return Q_HANDLED();
        }
        case CE_SIG: {
            BSP_clear();
            return Q_TRAN(state_opEntered);
        }
        case OPER_SIG: {
            if (BSP_eval(me->m_operand1, me->m_operator, BSP_get_value())) {
                me->m_operand1 = BSP_get_value();
                me->m_operator = ((CalcEvt const *)e)->key_code;
                return Q_TRAN(state_opEntered);
            }
```

```
            else {
                return Q_TRAN(state_error);
            }
            return Q_HANDLED();
        }
        case EQUALS_SIG: {
            if (BSP_eval(me->m_operand1, me->m_operator, BSP_get_value())) {
                return Q_TRAN(state_result);
            }
            else {
                return Q_TRAN(state_error);
            }
            return Q_HANDLED();
        }
    }
    return Q_SUPER(state_on);
```

NOTE: The `Calc1` state machine in Listing 6 uses “state-variables” in all transitions and superstate designations. However, you can limit the flexibility by using “state-variables” only for states that are intended for overriding. You can use the fixed addresses (e.g., `&Calc1::result` for the “result” state) for all states that you do **not** allow to override.

4.2 Calc2—Deriving a State Machine

Once you prepared a base statechart, you can easily derive from it and override the state machine elements that the base has exposed for such derivation, as shown in Listing 7.

4.2.1 Declaring the Derived Calc2 Statechart

Listing 7 Deriving Calc2 statechart from Calc1 base class

```
#ifndef calc2_h
#define calc2_h

#include "calc1.h"

enum Calc2Signals {
    PERCENT_SIG = MAX_CALC1_SIG,
    MAX_CALC2_SIG
};

class Calc2 : public Calc1 {                            // Calc2 state machine
public:
    Calc2(void);                                                      // ctor

protected:
    static QState operand2 (Calc2 *me, QEvent const *e);
    static QStateHandler   state_operand2;
};

#endif                                                             // calc2_h
```

The extended calculator needs one more signal to the `Calc1` signals, which is added in the enumeration `Calc2Signals`. The statechart `Calc2` inherits from `Calc1`, and declares only one state handler `operand2`, which will override the state handler from the base class. Additionally, the derived statechart `Calc2` declares the static state variable `state_operand2` for further derivation.

4.2.2 *Defining the Derived `Calc2` Statechart*

Listing 8 Definition of the Calc2 state machine

```
#include "qp_port.h"
#include "bsp.h"                                         // board support package
#include "calc2.h"

#include <stdlib.h>

Q_DEFINE_THIS_FILE

// state variables -----------------------------------------------------------
QStateHandler Calc2::state_operand2  = (QStateHandler)&Calc2::operand2;

// Ctor definition -----------------------------------------------------------
Calc2::Calc2(void) : Calc1() {
                                          // substitute all overridden states...
(1)     Calc1::state_operand2 = state_operand2;
}
//............................................................................
QState Calc2::operand2(Calc2 *me, QEvent const *e) {
    switch (e->sig) {
        case PERCENT_SIG: {
            double operand2 = BSP_get_value();
            switch (me->m_operator) {
                case KEY_PLUS:
                case KEY_MINUS: {
                    operand2 = me->m_operand1 * operand2 / 100.0;
                    break;
                }
                case KEY_MULT:
                case KEY_DIVIDE: {
                    operand2 /= 100.0;
                    break;
                }
                default: {
                    Q_ERROR();
                    break;
                }
            }
            if (BSP_eval(me->m_operand1, me->m_operator, operand2)) {
                return Q_TRAN(state_result);
            }
            else {
                return Q_TRAN(state_error);
```



```
                }
            }
        }
(2)     return Q_SUPER(&Calc1::operand2);          // let Calc1 handle other events
    }
```

Note first important aspect of Listing 8 is (1) changing of the state-variable `Calc1::state_operand2` to point to the state-handler function of the **subclass** `Calc2`. This change automatically updates all the relationships between the “operand2” state and other states in the Calc1 state machines.

The second vital aspect is that the overriding state handler `Calc2::operand2()` defines only the differences from the original state-handler `Calc1::operand2()`. In fact, the derived state-handler function `Calc2::operand2()` delegates handling all events except `PERCENT_SIG` to the base class state handler `Calc1::operand2()` by returning it as the superstate (2)..

4.3 Testing the Derived Statechart

Once you prepared a base statechart, you can easily derive from it and override the state machine elements that the base has exposed for such derivation, as shown in Listing 7.

Listing 9 Event loop for the Calc2 state machine

```
#include "qp_port.h"                        // the port of the QP framework
#include "bsp.h"                                    // board support package
#include "calc2.h"

#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>

// Local objects -------------------------------------------------------------
static Calc2 l_calc;                                  // instantiate Calculator2

//............................................................................
void main() {

    cout << "Calculator2 example, QEP version: "
         << QEP::getVersion() << endl
         << "Press '0' .. '9'     to enter a digit\n"
            "Press '.'            to enter the decimal point\n"
            "Press '+'            to add\n"
            "Press '-'            to subtract or negate a number\n"
            "Press '*'            to multiply\n"
            "Press '/'            to divide\n"
            "Press '=' or <Enter> to get the result\n"
            "Press 'c' or 'C'     to Cancel\n"
            "Press 'e' or 'E'     to Cancel Entry\n"
            "Press <Esc>          to quit.\n\n";

    l_calc.init();                              // trigger initial transition

    for (;;) {                                                   // event loop
```



```
CalcEvt e;                                         // Calculator event

BSP_display();                                     // show the display

e.key_code = (uint8_t)getche();                  // get a char with echo
cout << ": ";

switch (e.key_code) {
    case 'c':                              // intentionally fall through
    case 'C': {
        ((QEvent *)&e)->sig = C_SIG;
        break;
    }
    case 'e':                              // intentionally fall through
    case 'E': {
        ((QEvent *)&e)->sig = CE_SIG;
        break;
    }
    case '0': {
        ((QEvent *)&e)->sig = DIGIT_0_SIG;
        break;
    }
    case '1':                              // intentionally fall through
    case '2':                              // intentionally fall through
    case '3':                              // intentionally fall through
    case '4':                              // intentionally fall through
    case '5':                              // intentionally fall through
    case '6':                              // intentionally fall through
    case '7':                              // intentionally fall through
    case '8':                              // intentionally fall through
    case '9': {
        ((QEvent *)&e)->sig = DIGIT_1_9_SIG;
        break;
    }
    case '.': {
        ((QEvent *)&e)->sig = POINT_SIG;
        break;
    }
    case '+':                              // intentionally fall through
    case '-':                              // intentionally fall through
    case '*':                              // intentionally fall through
    case '/': {
        ((QEvent *)&e)->sig = OPER_SIG;
        break;
    }
    case '%': {                                   // new event for Calc2
        ((QEvent *)&e)->sig = PERCENT_SIG;
        break;
    }
    case '=':                              // intentionally fall through
    case '\r': {                                              // Enter key
        ((QEvent *)&e)->sig = EQUALS_SIG;
        break;
    }
    case '\33': {                                               // ESC key
        ((QEvent *)&e)->sig = OFF_SIG;
```

```
                    break;
                }
                default: {
                    ((QEvent *)&e)->sig = 0;                    // invalid event
                    break;
                }
            }

            if (((QEvent *)&e)->sig != 0) {            // valid event generated?
                l_calc.dispatch(&e);                            // dispatch event
            }
        }
    }
```

Listing 9 shows the event loop for `Calc2`, which is very similar to the original `Calc` event loop. The only differences are the instantiation of `Calc2` object instead of `Calc1` and generation of the PERCENT event.

The Listing 10 below shows a test of Calc2 run for the following computations:

```
100 + 8%
100 - 8%
100 * 8%
100 / 8%, and
100 / 0%, which causes a divide-by-zero error
```

Listing 10 Test run of Calc2

```
Calculator2 example, QEP version: 4.0.00
Press '0' .. '9'     to enter a digit
Press '.'            to enter the decimal point
Press '+'            to add
Press '-'            to subtract or negate a number
Press '*'            to multiply
Press '/'            to divide
Press '=' or <Enter> to get the result
Press 'c' or 'C'     to Cancel
Press 'e' or 'E'     to Cancel Entry
Press <Esc>          to quit.

on-ENTRY;on-INIT;ready-ENTRY;ready-INIT;begin-ENTRY;
[         0] 1: begin-EXIT;ready-EXIT;operand1-ENTRY;int1-ENTRY;
[         1] 0:
[        10] 0:
[       100] _:
[       100] 8:
[      1008] &:
[      1008] c: int1-EXIT;operand1-EXIT;on-EXIT;on-ENTRY;on-INIT;ready-ENTRY;read
y-INIT;begin-ENTRY;
[         0] 1: begin-EXIT;ready-EXIT;operand1-ENTRY;int1-ENTRY;
[         1] 0:
[        10] 0:
[       100] -: int1-EXIT;operand1-EXIT;opEntered-ENTRY;
[       100] 8: opEntered-EXIT;operand2-ENTRY;int2-ENTRY;
```

```
[        8] %: int2-EXIT;operand2-EXIT;ready-ENTRY;result-ENTRY;
[       92] c: result-EXIT;ready-EXIT;on-EXIT;on-ENTRY;on-INIT;ready-ENTRY;ready
-INIT;begin-ENTRY;
[        0] 1: begin-EXIT;ready-EXIT;operand1-ENTRY;int1-ENTRY;
[        1] 0:
[       10] 0:
[      100] *: int1-EXIT;operand1-EXIT;opEntered-ENTRY;
[      100] 8: opEntered-EXIT;operand2-ENTRY;int2-ENTRY;
[        8] %: int2-EXIT;operand2-EXIT;ready-ENTRY;result-ENTRY;
[        8] c: result-EXIT;ready-EXIT;on-EXIT;on-ENTRY;on-INIT;ready-ENTRY;ready
-INIT;begin-ENTRY;
[        0] 1: begin-EXIT;ready-EXIT;operand1-ENTRY;int1-ENTRY;
[        1] 0:
[       10] 0:
[      100] /: int1-EXIT;operand1-EXIT;opEntered-ENTRY;
[      100] 8: opEntered-EXIT;operand2-ENTRY;int2-ENTRY;
[        8] %: int2-EXIT;operand2-EXIT;ready-ENTRY;result-ENTRY;
[     1250] c: result-EXIT;ready-EXIT;on-EXIT;on-ENTRY;on-INIT;ready-ENTRY;ready
-INIT;begin-ENTRY;
[        0] ←: begin-EXIT;ready-EXIT;on-EXIT;final-ENTRY;
[        0] 1: begin-EXIT;ready-EXIT;operand1-ENTRY;int1-ENTRY;
[        1] 0:
[       10] 0:
[      100] /: int1-EXIT;operand1-EXIT;opEntered-ENTRY;
[      100] 0: opEntered-EXIT;operand2-ENTRY;zero2-ENTRY;
[        0] %: zero2-EXIT;operand2-EXIT;error-ENTRY;
[ Error 0 ] c: error-EXIT;on-EXIT;on-ENTRY;on-INIT;ready-ENTRY;ready-INIT;begin-
ENTRY;
[        0] ←: begin-EXIT;ready-EXIT;on-EXIT;final-ENTRY;
Bye! Bye
```


5 Related Documents and References

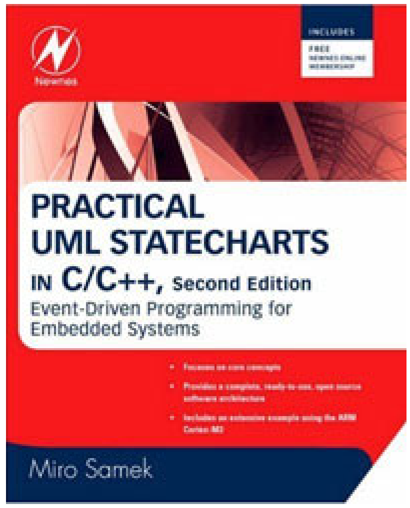
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[PSiCC1] "Practical Statecharts in C/C++: Quantum Programming for Embedded Systems", Miro Samek, CMP Books, 2002	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc.htm
[GoF 95] "Design Patterns: Reusable Elements of Object-Oriented Software", Erich Gamma at al. ("Gang of Four"), 1995	Available from most online book retailers, such as amazon.com.
[QP 08] "QP 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

6 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 Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*", by Miro Samek, Newnes, 2008