Hierarchical State Machines
(UML Statecharts)
Hierarchical State Machines (UML Statecharts)

• Current state of the art in state machine theory and notation
  → Reuse of behavior through state nesting (inheritance-type reuse)
  → Reuse of behavior through sub-machines (procedure-type reuse)
  → Many other features...

"IT MAY NOT BE A PERFECT WHEEL, BUT IT'S A STATE-OF-THE-ART WHEEL."
Lab Project: “blinky-button”

- Make the Blinky AO handle also the BTN1 press, which should light up LED1 for \( \frac{1}{2} \) seconds
  - Just like blinky-freertos version-1 lab project

- Don’t need to add another AO (!)
  - Blinky AO is **responsive** to events
  - Unlike blinky-freertos thread, which was NOT responsive due to blocking
State-Transition Explosion Problem

- Traditional Finite State Machines inflict repetitions
  → no mechanism to reuse behavior (show stopper for FSMs)
Hierarchical State Nesting

- Hierarchical state nesting captures common behavior in superstates → reuse through “programming by difference” (inheritance)
Lab Project: Hierarchical “blinky-button”

```c
QState Blinky_active(Blinky * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case BTN1_DEPRESSED_SIG: {
            GPIO_PinOutSet(LED_PORT, LED1_PIN);
            QTimeEvt_armX(&me->btn1TimeEvt, BSP_TICKS_PER_SEC/2U, 0U);
            status = Q_HANDLED();
            break;
        }
        case BTN1_TIMEOUT_SIG: {
            GPIO_PinOutClear(LED_PORT, LED1_PIN);
            status = Q_HANDLED();
            break;
        }
        case default: {
            status = Q_SUPER(&QHsm_top);
            break;
        }
    }
    return status;
}
```

```c
QState Blinky_on(Blinky * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case BTN1_DEPRESSED_SIG: {
            GPIO_PinOutSet(LED_PORT, LED1_PIN);
            QTimeEvt_armX(&me->btn1TimeEvt, BSP_TICKS_PER_SEC/2U, 0U);
            status = Q_HANDLED();
            break;
        }
        case BTN1_TIMEOUT_SIG: {
            GPIO_PinOutClear(LED_PORT, LED1_PIN);
            status = Q_HANDLED();
            break;
        }
        case default: {
            status = Q_SUPER(&Blinky_active);
            break;
        }
    }
}
```
Transition Execution Sequence / RTC Semantics

In the UML
→ Evaluate g()
→ Exit source states a(); b();
→ Transition action t();
→ Enter target states c(); d(); e();

In QP
→ Evaluate g()
→ Transition action t();
→ Exit source states a(); b();
→ Enter target states c(); d(); e();
Hierarchical State Machine Semantics
Lab Project: QHsmTst

C:\qp\qpc\examples\win32\qhsmtst\make
 gcc -MM -MT dbg/qhsmtst.o -g -ffunction-sections -fdatasections -O -Wall -W -I. -I./../include -I./../ports/win32 -DPQ_API_VERSION=9999 qhsmtst.c > dbg/qhsmtst.o
gcc -g -ffunction-sections -fdatasections -O -Wall -W -I. -I./../include -I./../ports/win32 -DPQ_API_VERSION=9999 c main.c > dbg/main.o
erg -g -ffunction-sections -fdatasections -O -Wall -W -I. -I./../include -I./../ports/win32 -DPQ_API_VERSION=9999 c qhsmtst.c > dbg/qhsmtst.o
erg -g -ffunction-sections -fdatasections -O -Wall -W -I. -I./../include -I./../ports/win32 -DPQ_API_VERSION=9999 c main.c > dbg/main.o
er -g -ffunction-sections -fdatasections -O -Wall -W -I. -I./../include -I./../ports/win32 -DPQ_API_VERSION=9999 c ../include/qstamp.c > dbg/qstamp.o
gcc -Wl,-Map,dbg/qhsmtst.map --cref --gc-sections -L./../ports/win32/dbg -o dbg/qhsmtst.exe dbg/main.o dbg/qhsmtst.o dbg/qstamp.o -lpq

C:\qp\qpc\examples\win32\qhsmtst\dbg\qhsmtst.exe
QHsmTst example, built on Jun 12 2017 at 16:33:29,
QP: 5.9.2.
Press ESC to quit...
top-INIT;s2-ENTRY;s2-ENTRY;s2-ENTRY;s21-ENTRY;s211-ENTRY;
>g: s21-g;s211-EXIT;s21-EXIT;s1-ENTRY;s1-ENTRY;s11-ENTRY;
>i: s1-I;
>a: s1-A;s11-EXIT;s1-EXIT;s1-ENTRY;s1-ENTRY;
>d: s1-D;s11-EXIT;s1-EXIT;s1-ENTRY;s1-ENTRY;
>d: s11-D;s11-EXIT;s1-ENTRY;
>c: s1-C;s11-EXIT;s1-EXIT;s2-ENTRY;s2-ENTRY;s21-ENTRY;s211-ENTRY;
>e: s-E;s211-EXIT;s21-EXIT;s2-ENTRY;s2-ENTRY;s11-ENTRY;
>e: s-E;s11-EXIT;s1-ENTRY;s11-ENTRY;
>c: s1-C;s11-EXIT;s1-ENTRY;s2-ENTRY;s2-ENTRY;s21-ENTRY;s211-ENTRY;
>i: s2-I;
>i: s-I;
>?: Bye, Bye!
Coding HSMs

- You can code HSMs manually with QP (as you’ve been doing so far)
  → Repetitious and error prone 😞

- OR

- You can press the “Generate Code” button (F7) in QM
  → Guaranteed to match your design 😊
Lab Project: “blinky-button” with QM
Designing a Non-Trivial HSM (Calculator)

- Problem specification:
  - Parse numerical expressions of the type: operand operator operand equals ...
  - The user can provide any symbol at any time
  - Allow chaining of expressions
  - Allow negative numbers (‘-’ button)
  - Handle the ‘OFF’ button
  - Handle the ‘ON/C’ button (Cancel)
  - Handle the ‘CE’ button (Cancel Entry)
  - Ignore the ‘MR’, ‘M+’, ‘M-’, and ‘%’ buttons for now
High-Level Design

→ Realize the primary function (the primary use case)
→ Analyze and fix the behavior by discovering states
→ Scrutinize events for the right level of granularity
Actively Look for Reuse

• Look for repeated transitions and place them in invented superstates
Elaborate High-Level Composite States

- States “operand1” and “operand2” need substates
Refine the Behavior (negative numbers)

- $2 \times -2 = \ldots$

- $-2 \times 2 = \ldots$
Final Touches

- Factor out “ready” state
- Make “negated” substates of “operand”
- Add “error” state
- Add CE transitions
Lab Project: “calc1”

- `qpc\examples\win32\calc1`

```
ON Command Prompt
C:\qpc\examples\win32\calc1>dbg\calc1.exe
Calculator example, QEP version: 5.9.2
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] = 2 begin-EXIT;ready-EXIT;operand1-ENTRY;int1-ENTRY;
[ 2] = - int1-EXIT;operand1-EXIT;opEntered-ENTRY;
[ 2] = - opEntered-EXIT;operand2-ENTRY;negated2-ENTRY;
[ 2] = - negated2-EXIT;int2-ENTRY;
[ 2] = int2-EXIT;operand2-ENTRY;ready-ENTRY;result-ENTRY;
[ 3] = result-EXIT;ready-EXIT;on-EXIT;final-ENTRY;
```
Submachines

- How to reuse the common internal structure of “operand1” and “operand2”?
- Needed reuse of different type than inheritance state nesting
- Need procedure-type reuse
  → **Submachines** (reusable composite state, like a procedure)
  → **Submachine states** (specific context of use, like a procedure call)
Lab Project: “calc1_sub” step1
Lab Project: “calc1_sub” step2

- Remove “1” from all states’ names. Add **entry points** and **exit points**
Lab Project: “calc1_sub” step3

Errors occurred during code generation. Please check the log window for details.

Log Console:

ERROR C101> ${SMs::Calc::SM::operand} : submachine
INFO> Code generation ended (time elapsed 0.008149)
INFO> 1 file(s) generated, 2 file(s) processed, 2 error(s), and 0 warning(s)
INFO> Entire model: C:\qp\pc\examples\win32\calc1\calc2.qm
ERROR C100> ${SMs::Calc} : empty submachine
ERROR C101> ${SMs::Calc::SM::operand} : submachines allowed only in subclasses of QMachine
State Machine Classes in QP

1. `QHsm` - `abstract` class
   - `state : QHsmAttr`
   - `init()`
   - `dispatch()`

2. `QActive` - `abstract` class
   - `eQueue`
   - `osObject`
   - `thread`
   - `prio`
   - `start()`
   - `post()`

3. `QMsm` - `abstract` class
   - `init()`
   - `dispatch()`

4. `QMActive` - `abstract` class
   - `start()`
   - `post()`

5. `QTimeEvt` - class
   - `ctr`
   - `postIn()`
   - `postEvery()`
   - `disarm()`
   - `rearm()`

6. `Ship` - `Missile` - `Tunnel`

7. `Mine1` - `Mine2` - `Mine3` - `Mine4`

“Fly ‘n’ Shoot”
Game application
1. Change Calc superclass to QMsm

2. Change Calc_ctor() to call QMsm_ctor()

3. Change the code generation license
Lab Project: “calc1_sub” step5

make and run
Guidelines for Designing HSMs

- Don’t Repeat Yourself (DRY principle)
  → Use state hierarchy to eliminate repeated transitions
  → Use submachines to eliminate repeated sub-state structures
  → Use entry/exit actions instead of repeating the same actions on transitions

- Prefer state entry/exit actions over actions on transitions (Moore machines over Mealy machines)
  → Entry/exit actions provide guaranteed initialization and cleanup on any transition path (safer design, easier to maintain)

- Prefer modeling and automatic code generation over manual coding
  → Take advantage of graphical modeling. Eliminate manual labor.
Pitfalls to Avoid in HSMs (1)

- Polling or blocking inside SM violates RTC semantics!
  → un-blocking is an alternative way of **delivering events** in the middle of RTC

```c
QState MyHSM_stateA(MyHsm * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case MY_EVENT_SIG: {
            // do something...
            OSTimeDly(OS_TICKS_PER_SEC / 2);
            // do something...
            status = Q_HANDLED();
            break;
        }
        . . .
    }
}
```

“Timeout event”
Pitfalls to Avoid in HSMs (2)

- Indirect sharing memory through events
  - e.g. passing pointers to **shared** buffers inside events

```c
QState MyHSM_stateA(MyHsm * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
    case SPI_DATA_SIG: {
        uint8_t *buf = Q_EVT_CAST(SPIEvt)->pBuf;
        // process data in buf...
        status = Q_HANDLED();
        break;
    }
    ...
}
```

```
void SPI_IRQHandler(void) {
    for (i = 0; i < SPI_BUF_LEN; ++i)
        g_spiBuf[i] = SPI_REG; // global buffer
    // post an event
    SPIEvt *evt = Q_NEW(SPIEvt, SPI_DATA_SIG);
    evt->pBuf = g_spiBuf;
    QACTIVE_POST(evt, (void *)0);
}
```
Pitfalls to Avoid in HSMs (3)

- State handlers have **read-only** access to the current event
  - Don’t write to the current event, especially after (down)casting
  - Use the `Q_EVT_CAST()` macro to down-cast current event

```c
QState MyHSM_stateA(MyHSM * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case MY_EVT1_SIG: {
            ((MyEvt1 *)e)->foo = 0x1234; /* WRONG! */
        }
        . . .
    }
    return status;
}
```
Pitfalls to Avoid in QP HSMs (4)

- Events are weakly typed in QP and you need to (down)cast them
  → Event type to cast must be unambiguously determined by the event signal

```c
QState MyHSM_stateA(MyHsm * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case MY_EVENT_1_SIG: {
            me->foo = Q_EVT_CAST(MyEvent2)->foo; /* WRONG?! */
            . . .
            status = Q_HANDLED();
            break;
        }
        . . .
    }
    return status;
}
```
Summary

- Hierarchical State Machines have mechanisms for **reusing** behavior
  - State nesting (inheritance-type reuse)
  - Submachines (procedure-type reuse)

- The most constructive element of the UML
  - Graphical modeling and automatic code generation

- Beautifully complement the concurrency model of Active Objects
  - Don’t use state machines without an active object framework!