Event Driven Embedded Software and UML State Charts

Matthew Eshleman
covemountainsoftware.com

Background - Matthew Eshleman

- Nearly 20 years of embedded software engineering, architecture, and project planning
  - Delivered 30+ products with many more derivative projects. Millions of people have used my code.
  - MSEE, BEE, Georgia Tech
- Learn more: http://covemountainsoftware.com/consulting

Agenda

- What do I mean by Event Driven and Hierarchical state machines?
- Alternatives
- Pros and Cons
- Overview of UML State Charts (Hierarchical state machines)
- Demo/Examples: Qt, Custom

Once upon a time…

- Vacation Road Trip
- Reading Samek’s 1st Edition
- Eureka!

What is “Event Driven”?

- The software is designed to process events in order, to completion. The architecture is “event centric.”
- No polling loops.
- No global variables/flags that can change in the middle of logic.
- Herb Sutter, "*Prefer Using Active Objects Instead of Naked Threads*”.
  - “Active Object” is event driven.

What is Event Driven?

What is Event Driven? - Event Sources

Example of Event **Sources**:

- Button press
- Tuner lock status
- Network connection status
- Battery status
- Sensor Event (Temperature change, etc)
- Algorithm Generated Event
    - Ratings Violation, Signal Exceeds Threshold
- Error Event

What is Event Driven? - Frameworks

Example of Frameworks:

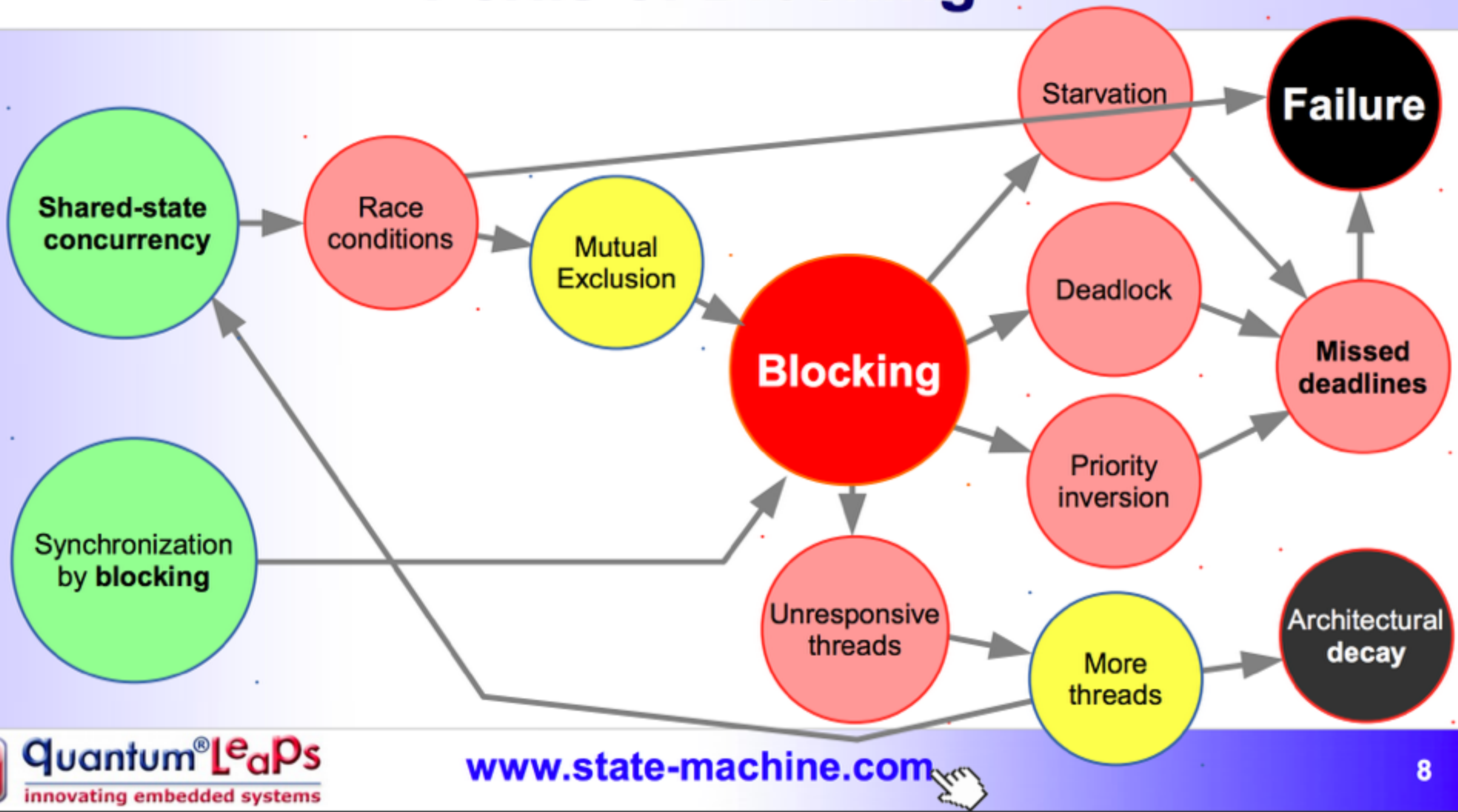
- Custom
  - RTOS Threads and Queues. Manually inject events into queues, some thread waiting on the queue receives and processes the event.
- QP Active Objects: http://www.state-machine.com/
- Qt: Signals and Slots
- C#: Events and Delegates
- The choice of framework will define how the consumer will be designed and coded and how events are created and received.

The Ideal Event Consumer

- For me, an ideal event consumer is an “Active Object” with an internal hierarchical state machine

Superloop issues

- Often processing global variables that are themselves modified in interrupt service routines or perhaps other threads
- Blocking issues. What if you must “wait 100ms” for some hardware event. sleep(100)?
  - Oops. Other events/changes are now delayed as well.
- Often devolve into spaghetti as complexity increases.

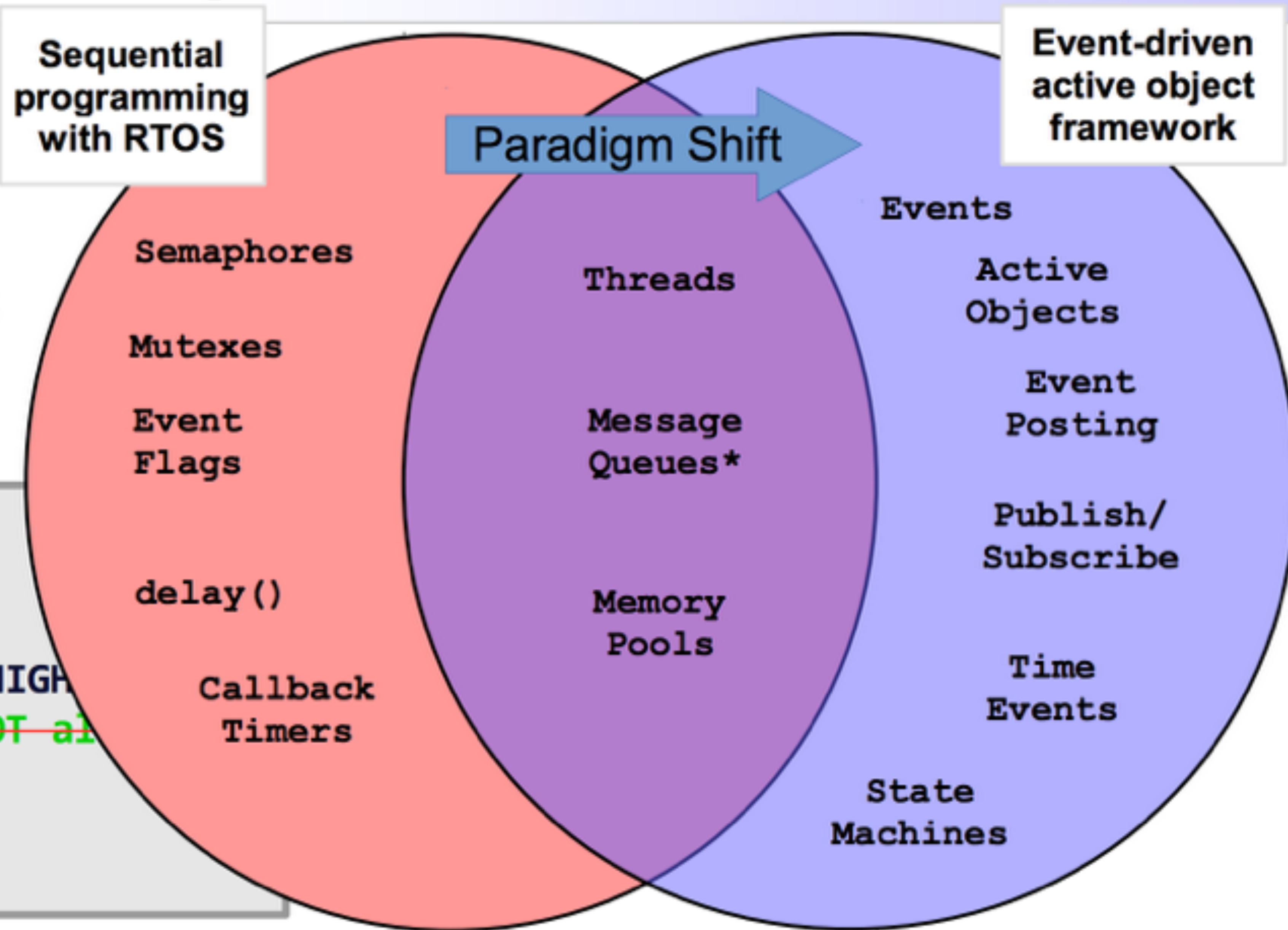


*slide used with permission [4]

Paradigm Shift: Sequential → Event-Driven

- No blocking
  - → No use for most RTOS mechanisms!

```
void thread_blink() {
    pinMode(LED_PIN, OUTPUT);
    while (1) {
        digitalWrite(LED_PIN, HIGH
        RTOS_delay(1000); // NOT al
        ...
    }
}
```



*slide used with permission [4]

Event Driven Pros

- Product specifications are often defined using event language: “If THIS happens, then do THAT”. Using an event driven architecture paired with state machines results in code that maps nicely to specifications.
- Queuing events guarantees the order of events. This is a common flaw in many “super loop” style code, where global variable flags are often used to indicate some event.
- The code is much easier to read, understand, maintain.
- Great opportunity to save power and battery life.
- Test Driven Development is easier.

Event Driven Cons

- Queues
  - How deep should the event queue be?
  - Oversized queues waste memory
- Timing - what happens during a burst of events?
  - Especially if your system/specifications have true real time constraints, the event driven approach may need to be re-considered. Consider a locomotive brake controller…

Compare and Contrast
(super loop with global state, multiple global variables)

```
switch (State) {
case RX:
  appData.rxCount++;
  appData.flags.bit.rxedPacket = 1;

  if (BufferSize > 0) {
    if (strncmp((const char*) Buffer, (const char*) PongMsg, 4) == 0) {
      led = !led;
      appData.RssiValueSlave =
                     ((uint16_t) Buffer[sizeof(PongMsg)])
                     + (((uint16_t) Buffer[sizeof(PongMsg)
                                          + 1]) << 8);
    } else {
      pRadio->Rx(RX_TIMEOUT_VALUE);
    }
  }
case TX:
  //etc
```

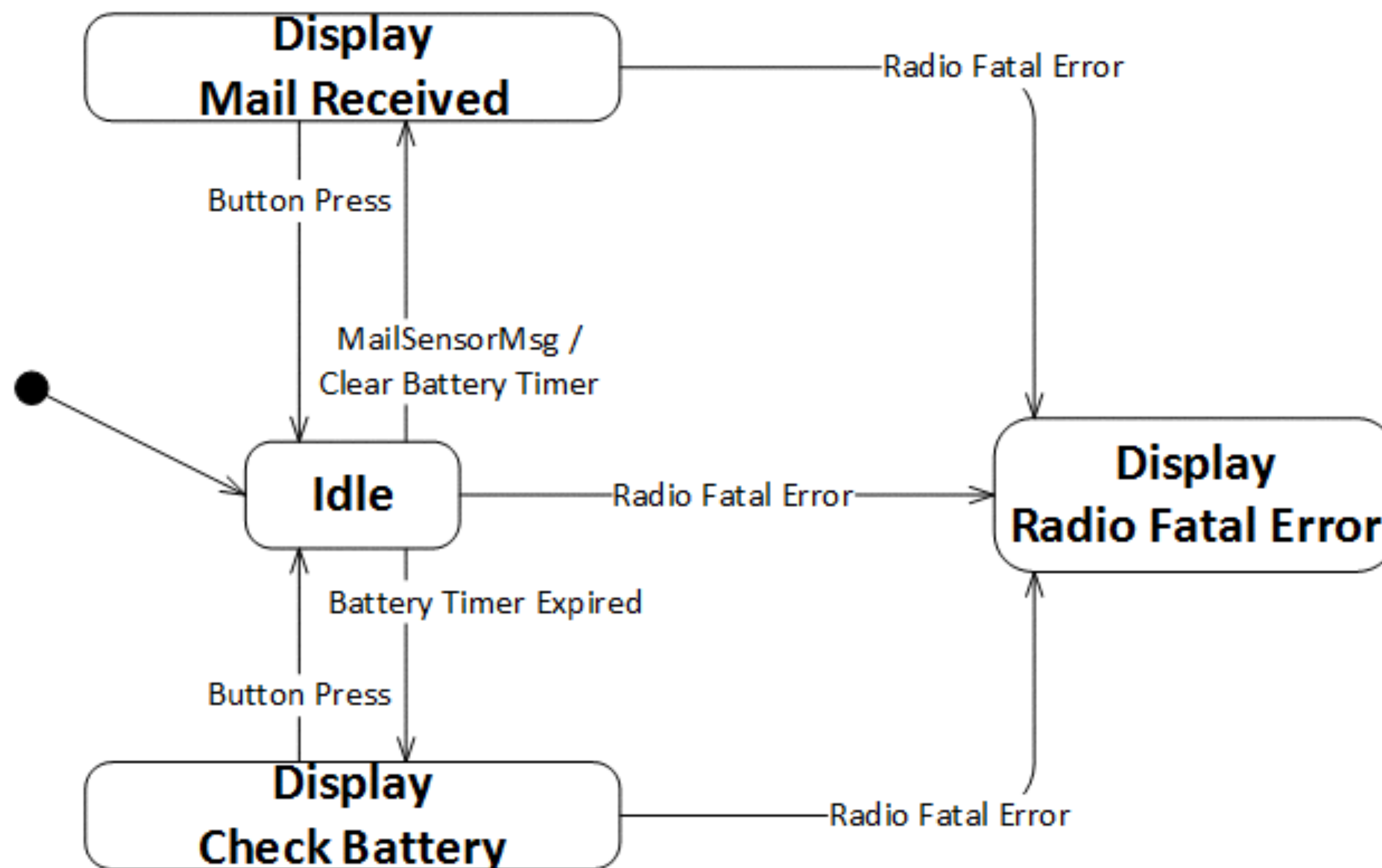
- Lots of global variables
- Globals modified in ISR context
- (not shown) hard coded delays

Compare and Contrast (Event driven state machine)

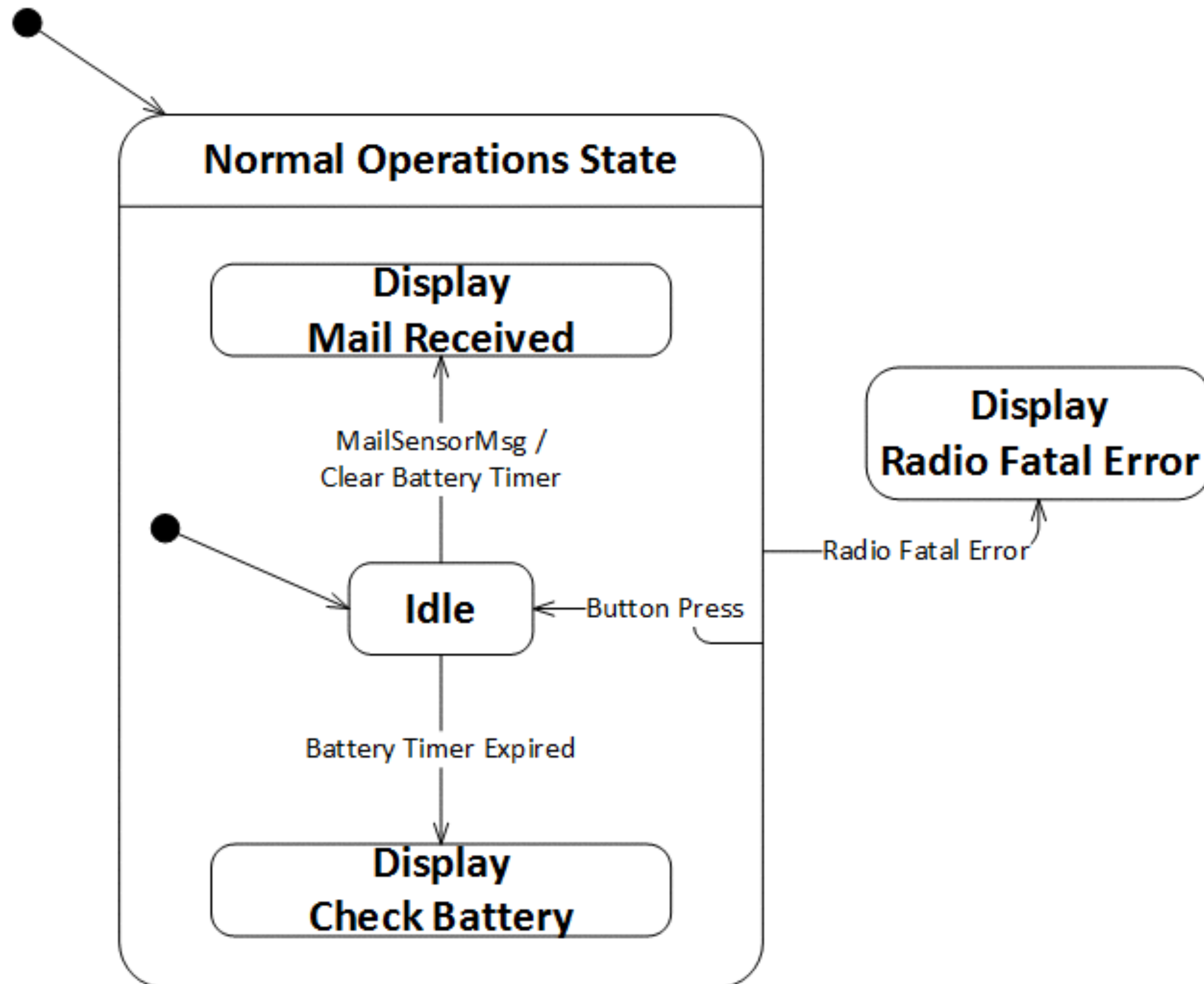
```
Statemachine::SmHandler_ Statemachine::Idle(Statemachine*, const Event event)
{
    switch (event)
    {
        case Event::ENTER_STATE:
            //CUT clear OLED
           break;
        case Event::SENSOR_MSG_RXD_OK:
            return DisplayMailReceived;
            break;
        case Event::EXCEED_DAYS_WITHOUT_SENSOR_MSG:
            return DisplayCheckSensorBattery;
            break;
        case Event::HEARTBEAT:
            m_seconds_without_mail_sensor_msg++;
            //CUT heartbeat OLED display
            CheckTooManyDaysWithoutSensor();
            break;
        default:
            break;
    }
    return Idle;
}
```

Tutorial on UML State Charts

State Machines - Flat State Machine



State Machines - Same Behavior - Hierarchical



Demo

- Qt Demo
    - source: https://github.com/mattheweshleman/qt-state-machine-demo
- Mail Box Alert Project Demo

Single class implementation (Samek style)

- Single class/object represents a single state machine, where each state is implemented as a static method.

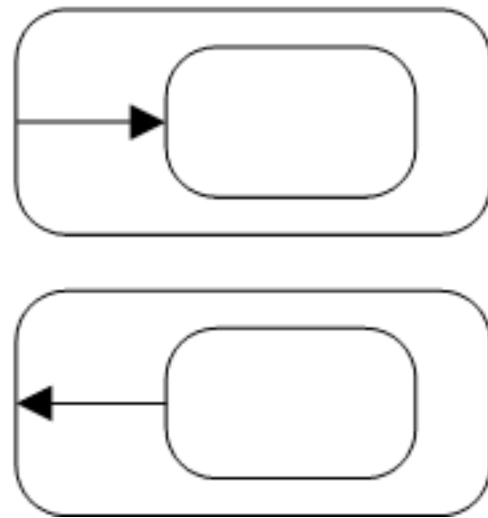
- Contrast with Qt and other designs, where each state is an instantiated object and the state machine itself is also an instantiated object.

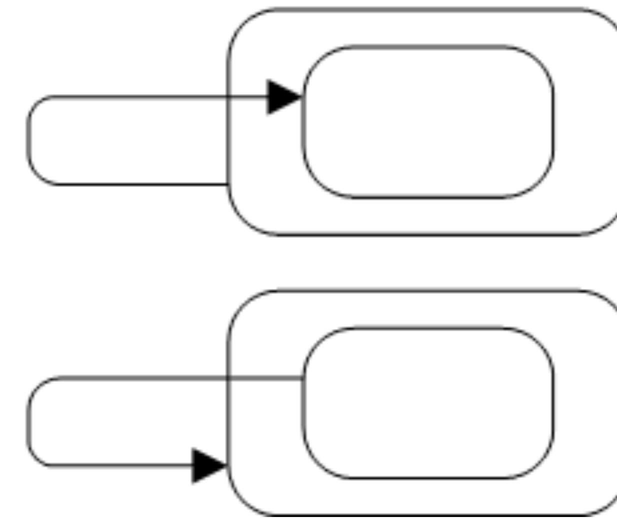
```
class BaseStationManager : public Statemachine
{
public:
  <<cut>>
  void Init();
  void ProcessEvent(const ManagerEvent& event);

private:
  <<cut>>
  static SmHandler NormalOperations(<<cut>>);
  static SmHandler    Idle(<<cut>>);
  static SmHandler    DisplayMailReceived(<<cut>>);
  static SmHandler    DisplayCheckSensorBattery(<<cut>>);
  static SmHandler DisplayRadioFatalError(<<cut>>);

  Handler mCurrent;
};
```

Note: Local vs External Transitions





- Pre UML 2.0, only “external transition” supported.
    - Think of “external transitions” as a type of reset event (they are often used as such).
- I naturally prefer “local transitions” and often I am bitten by this distinction.
- Many state machine frameworks only support “external transitions”, i.e. Qt’s, which is based on an older version of “State Chart XML”
    - Correction: most recent version of Qt now supports both transition types, but default is still external transition type.

https://en.wikipedia.org/wiki/UML_state_machine#/media/File:UML_state_machine_Fig8.png

Hierarchical State Machine Pros

- Code represents the system behavior more directly.
- Easy to maintain.
- [2] “They force you to understand your system behavior more clearly.”
- [1] “In other words, hierarchical state nesting enables programming by difference.”

Hierarchical State Machine Cons

- Might result in larger binaries.
- That is it! :-)

Resources

- [1] https://en.wikipedia.org/wiki/UML_state_machine
- [2] https://www.embeddedrelated.com/showarticle/723.php
- [3] http://www.state-machine.com/
- [4] http://www.state-machine.com/doc/Beyond_the_RTOS_Slides.pdf
- [5] http://doc.qt.io/qt-5/qstatemachine.html
- [6] https://www.w3.org/TR/scxml/
- [7] https://github.com/mattheweshleman/qt-state-machine-demo
- [8] https://github.com/mattheweshleman/mailbox-event-driven-demo

Thank you!

Any questions?

matthew@covemountainsoftware.com