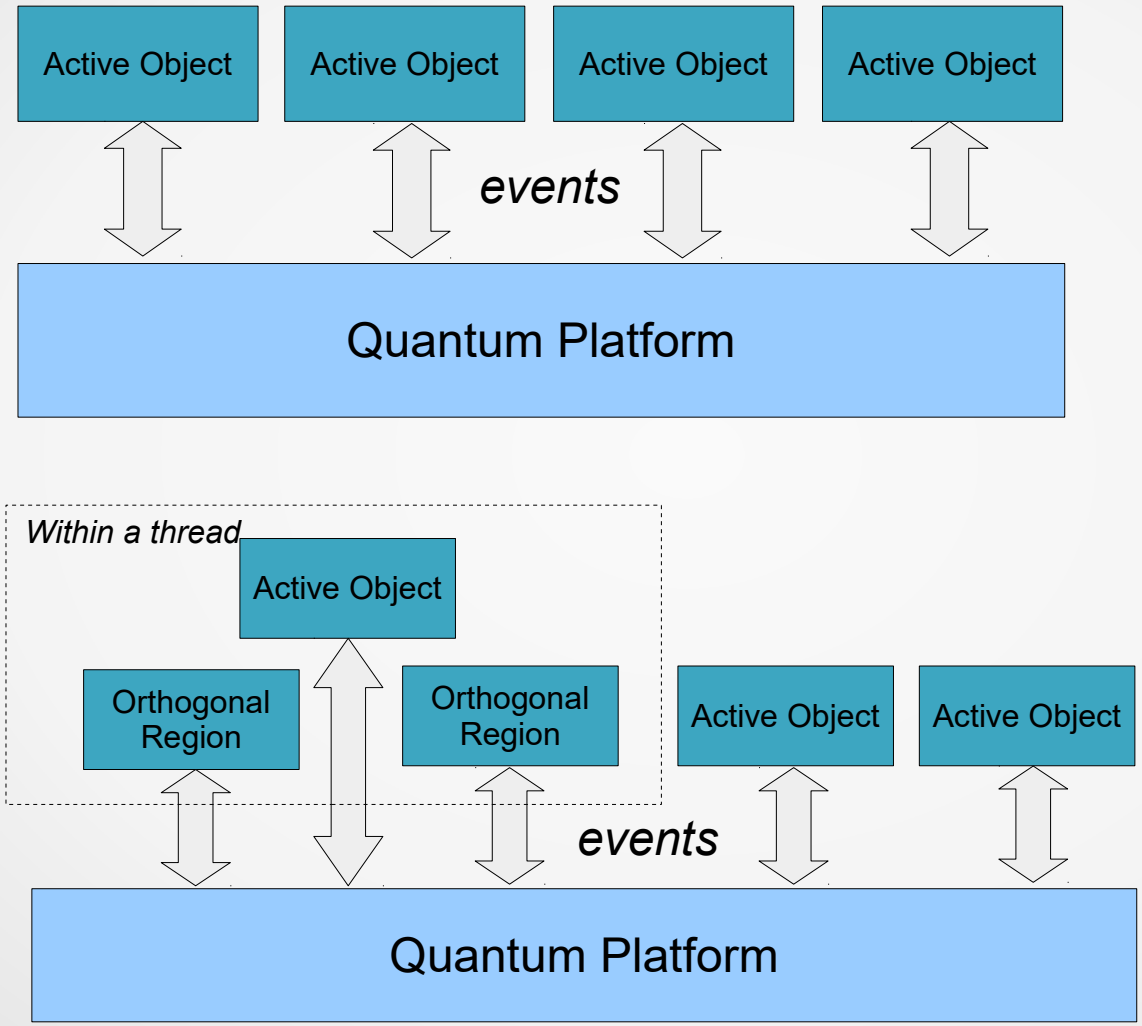
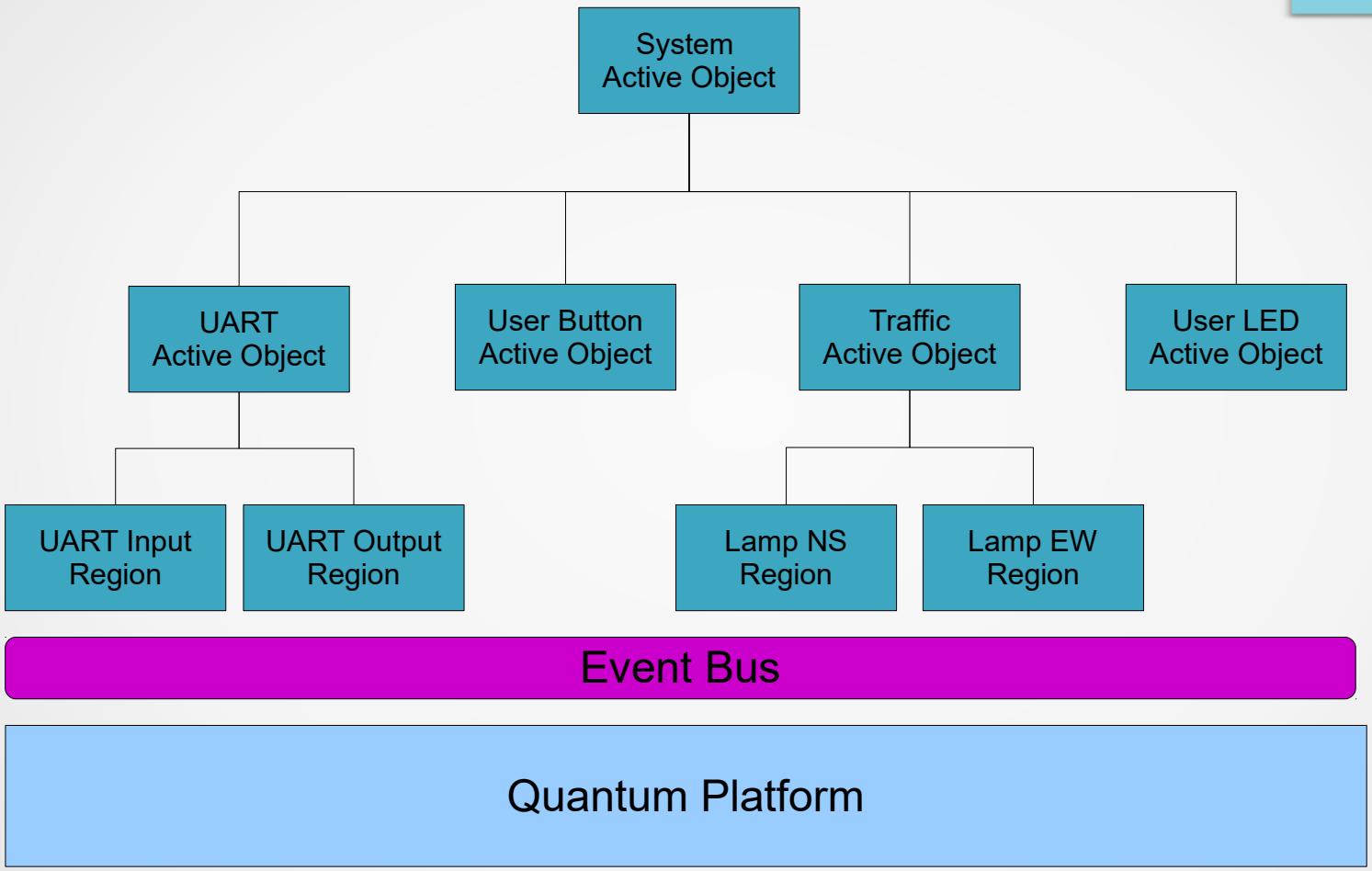
EMBSYS 110, Spring 2017 Week 6

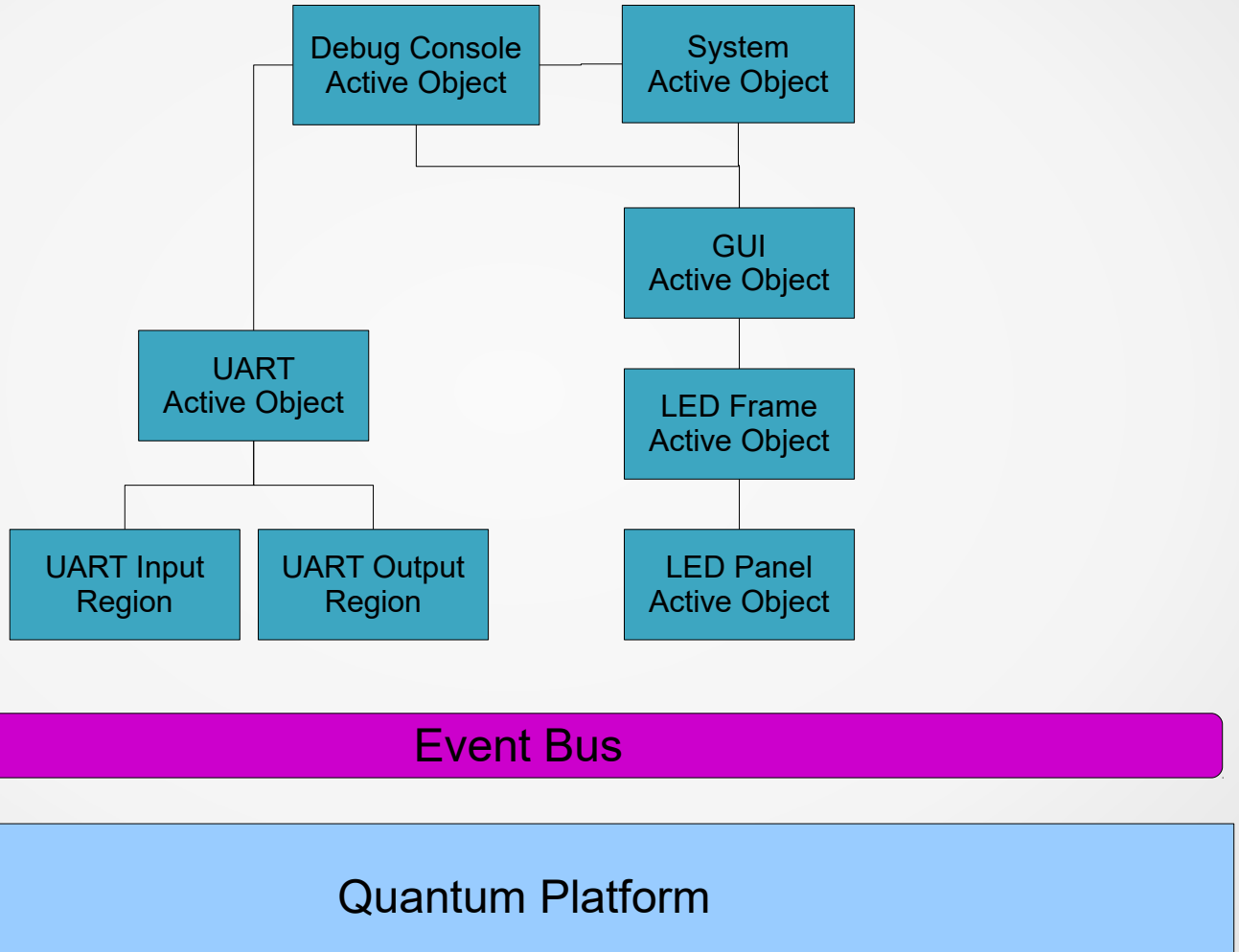
- Week 6
  - Review of System Partitioning
  - Event Interface
  - Robustness and error handling strategies.
  - Design by contract.
  - Debugging, profiling and testing.

System Partitioning (Review)



System Partitioning (Review)




Debug Console
Active Object
System
Active Object
GUI
Active Object
UART
Active Object
LED Frame
Active Object
UART Input
Region
UART Output
Region
LED Panel
Active Object
Event Bus
Quantum Platform

System Partitioning (Review)

- Some interesting references:
- Large Hadron Collider:
- https://accelconf.web.cern.ch/accelconf/ica07/PAPERS/RPPB21.PDF

Event Interface

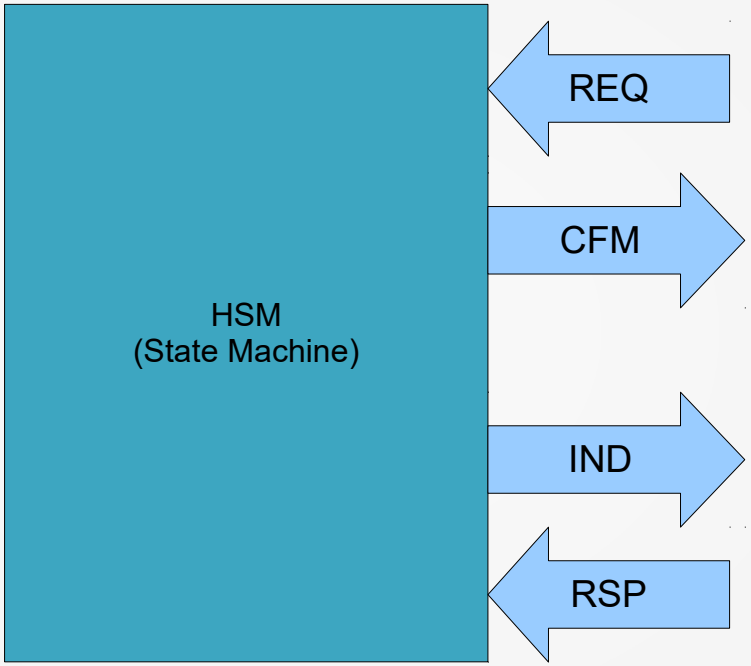
- QP or other event frameworks do not dictate how events are defined. It provides flexibility. Polite way of saying they don’t care.
- Easy to end up with an ad-hoc or loose set of events.
- Active objects (or in traditional RTOS terms *threads*) are asynchronous. They interact just like network nodes using network protocols.
- Events, therefore, should be defined like messages/packets of a protocols.

Event Interface

- We adopted the following common semantics:
  - REQ and CFM (request and confirmation)
  - IND and RSP (indication and response)
- They form pairs with matching sequence numbers.
  - CFM acknowledges REQ.
  - RSP acknowledges RSP.
- Naming convention:

  <***OBJ_NAME***>_<***EVENT_TYPE***>_REQ and so on.

  For example: USER_LED_PATTERN_REQ, UART_IN_CHAR_IND
- Question – why use sequence number? (Race conditions.)
- Acknowledgment is like returning status from a function. The norm is to check with exception allowed. (Not the other way round!)



Event Interface/Statechart Design

- Each HSM exposes an event interface. It describes the services it provides.
- How an service is fulfilled is hidden. Only care about the results. (Just like… :)
- Max timeout is part of the REQ. Otherwise how does the requesting object know how long to wait? Guaranteed to send a confirm within max timeout.
- Upon timeout, requesting object should handle the timeout exception case.
- Redundancy in timeout handling? Done in both requesting and requested object. Why is it necessary? Different teams work on different components. No assumption, no trust. Why does a computer program hang?

Event Interface/Statechart Design

- When designing a statechart, it is important to handle every request in every state! (So for some indications as well.)
- It sounds challenging. But thanks to state hierarchy!
- What if you missed one state? Thanks to timeout in requesting object! Now we see the importance of redundancy.
- Examples of the need to handle a request in an unexpected state? Actually a common cause for race conditions! Statechart exposes all these cases so we can see them!

Event Interface/Statechart Design

- Note the use of internal events as reminders. Useful for
  - Breaking up a long chain of actions.
  - Factorizing common actions.
  - Just to make a statechart look neater!
- Protocol reference. Q.921 LAPD/HDLC and BLE.
- Summary of statechart design pattern so far
  - **Use well-defined event interface (REQ/CFM/IND/RSP).**
  - **Use sequence number.**
  - **Always check for timeout.**
  - **Handle all requests in all states.**

Robustness

- A software system is robust when it can cope with not just normal situations but also unexpected ones.
- One that crashes upon unexpected user input or network packets is NOT robust. (For example, my laptop crashes when I close the lid and reopen it quickly! Other examples?)
- One that hangs or simply becomes unresponsive for a long time is NOT robust. (Example?)
- How about one that handles unexpected situations in an ad-hoc (or not precisely defined) manner? A “low level programmer” (recall Harel’s paper) may end up making decisions on whether *or* how to handle an error status returned from a called function, which can be retrying for N times, using different parameters, or simply returning an error.

Robustness

- Since there may be little coordination among programmers about these “low-level” details (and even if there is it may not be clearly defined), the overall or global system reaction to such unexpected situations is usually not easy to figure out. (What happens when I reopen the lid quickly?)
- Discussion – How about C++ exception? (try, catch, thow, etc.) See PSCiCC.
- An embedded real-time system (a.k.a. reactive system) has to be robust, since by definition it has to react to all kinds of asynchronous input from the “world”, which can be in any order, with various timings, expected or not.

Error Handling

- Statechart is a nature fit for specifying error handling. Why?
- Statechart **can** *precisely*, *concisely* and *completely* describe system behaviors, which can include how it reacts to error situations.
- Note the word "**can**" above. A designer (you) still have to think about any possible error situations and how they should be handled.
- What's different now?

Error Handling

- Differences:
  - Instead of thinking about error handling when you are coding, you think about it when doing a proper design.
  - You can focus on logical behaviors (or interactions) rather than coding syntax.
  - A statechart shows error handling behaviors explicitly and allows them to be easily visualized and communicated.
  - Hierarchical states make default error handling easy.
  - State transition makes it easy to “jump” back to retry or restart. (Compare to loops.) In other words it is flexible to control the flow.

Error Handling

- Entry and exit actions guarantee proper initialization and cleanup (synchronous). Important to ensure resources are freed when exiting a state due to errors. Once specified in an an exit action, it applies to all cases when a state is exit. (Think about calling “fclose()” on every error.)
- Timer is convenient to use. (Compare to checking tick count in a loop.) Typical pattern is:
  - Starting timer when entering wait state.
  - Stopping timer when exiting from wait state.
  - Handling timeout event in wait state.

Error Handling

- PWS and event semantics ensure safety constraints are met. Examples:
  - When exiting an operational state (e.g. Scanning) in the ***whole*** layer, it is guaranteed all ***parts*** (e.g. Motor, RadioBeam, etc) are shutdown.
  - How? Via STOP_REQ and matching CFM.
  - After stopping all parts, it is guaranteed all resources previously allocated for them will not be accessed any more and will be safe to delete (a shared FIFO, file handle, etc.)
- Asynchronous non-blocking event-driven paradigm allows any exception conditions to be detected and handled as soon as *required*.

Error Handling Strategies - Assert

- Decide which errors to handle and which *not*.
  - *“Not”* does not mean to ignore the error.
  - It means firing an *assert*, which typically causes reboot.
  - It should also log an error.
  - Reboot is a drastic but effective measure to recover from an otherwise unrecoverable error.
  - Reboot is better than having the system run erratically or being unresponsive. It brings the system to fail-safe state.

Error Handling Strategies - Assert

- Assert is to verify a condition or assumption that must be true unless there is a software bug or an *unlikely* hardware malfunction.
- This is called “design by contract”. See PSCiCC.
- Discussion – Does a hardware malfunction trigger assertion? Yes, it simplifies design (less error cases to handle). No, it should continue to run to for diagnosis. Consider external components vs internal blocks.
- Discussion – Do you disable assert for production release?
-

Error Propagation

- Do not assert for these cases:
  - Invalid user input.
  - Invalid data packet.
  - Error status from another software components (CFM/IND event with failure). Treat them as black boxes. Though they may not fail today, they may fail in the future. Make no assumption.
  - Invalid REQ events from another software components. Same rational as above. It includes REQ arriving in an unexpected or inconvenient state. (asynchronous system).
  - Error conditions detected in certain hardware components (e.g. external ones that are more likely to fail).

Error Propagation

- How to handle errors that do not trigger assert?
  - Retry. Upon failed CFM from the lower layer (the *part*), retry for a number of times.
  - Propagation. Return failed CFM to the upper layer (the *whole*) with reason and source of failure.
  - Report the error to the user which can be a human operator or another machine on the network. Examples include resource not available, hardware failure, etc. It is up to the external user to decide how to handle it (retry, diagnosis, etc.)
  - Handle the error as explicitly defined in the statechart (app specific).
- The above is done recursively until the error event is propagated to the topmost state machine.

Case Study - UartAct

- *UartAct* stands for UART Active Object.
- It is a *part* to System. It is a *whole* of the composed UartIn and UartOut (parts).
- Note the use of timers in wait states.
- Note the propagation of start and stop errors to the upper layer.
- Check start and stop requests are handled in all states.
- When UartAct returns to the Stopped state, it is guaranteed UartIn and UartOut are both stopped as well. Safe to free shared resources such as FIFO’s.

Case Study - UartAct

- Design review
  - What is the purpose of UART_OUT_FAIL_IND and UART_IN_FAIL_IND?
  - Are they handled in all possible states?
  - If not (e.g. starting), how to fix it?
  - (defer...)
- A subtle but important note on the asymmetry between starting and stopping paths:
  - It favors stopping for fail-safe design. A stop request is expected to always succeed (assert otherwise).
  - A stop request is automatically deferred if it cannot be conveniently be handled in a transient state.

Case Study - UartAct

- A start request may fail (as discussed above). It can be handled with retries, propagation, or by reporting to the end user.
- Example of start request failure – rapid stop-start requests back-to-back. The state machine is in Stopping state when it receives a start request.
- How would you handle it? It can be in the middle of stopping others components.
- A simple way is to defer it like stop request. However we don’t want to defer both stop and start request. Why?
- Here we just return failure (let upper layer retry). In reality it shouldn’t happen if the upper layer (whole) always waits for stop CFM before restarting (no rapid stop-start sequence above). Again we are extra cautious as we are not making assumption!
- Example – Console command to restart entire system.

Design by Contract

- Use assert to verify input parameters and other invariant.
- Extreme useful in catching bugs.
- See example uses in QP.
- Better to keep them in production code and trigger system reboot as recovery.

Debugging, Profiling and Testing

- Debugging with interactive debug console which is itself an HSM (advantages?).
- Debug level run-time setting.
- Debug on/off run-time setting per HSM.
- Debug console can show the current states of all HSM’s. Global state.
- Debug console can start and stop individual HSM’s. Useful for testing and diagnosis.
- It’s not trivial to be able to repeatedly start and stop a module (HSM). Compare with a free-running RTOS thread. A good test for leakage.

Debugging, Profiling and Testing

- Profiling with printf. Macros help:
  - Log entry and exit actions of every state.
  - Log events (with time-stamp) handled in every state.
  - Keep track of the current state of every HSM.
- Rely on an efficient UART output (DMA, non-blocking, zero-copy, ISR-safe). We will use it as an example when discussing optimization.
- Symbol based. Pros and cons. Compare with QSPY.
- Use GPIO for timing measuring and profiling. Example of IMU sampling. Useful for checking processor load and response time.
- CPU utilization measurement using counter in idle loop (recall UCOS).
- Keep statistics (e.g. watermark for event pools, queues, or maximum latency between interrupts). Report in console periodically or upon command.

Debugging, Profiling and Testing

- Testing – Most unit test framework focuses on functional model. That is the setup, testing and tear-down are done via synchronous function calls.
- They are useful for testing helper classes through their functional API. Examples include a FIFO class, a data formatting class, etc.
- But they *may* not fit the event driven model.
- What is more useful/interesting is automated integration testing. Most tricky issues arise from the interaction among asynchronous state machines, or from interacting with real hardware.

Debugging, Profiling and Testing

- Sum of testing modules individually (unit testing) !=

  Testing the sum of modules together (integration testing)
- How good is a unit test of an IMU driver that mock out the SPI bus and the real IMU chip?
- Compare to protocol tester. It needs to send a request and wait for the confirmation. It may take multiple steps.
- It also needs to verify the states of the related HSM’s.
- Error injection to simulate error conditions.