Instrument Software Framework (ISF)

A Small Scale Component Framework for Space

Timothy Canham, Jet Propulsion Laboratory
Garth Watney, Jet Propulsion Laboratory
Leonard Reder, Jet Propulsion Laboratory

12/11/2013

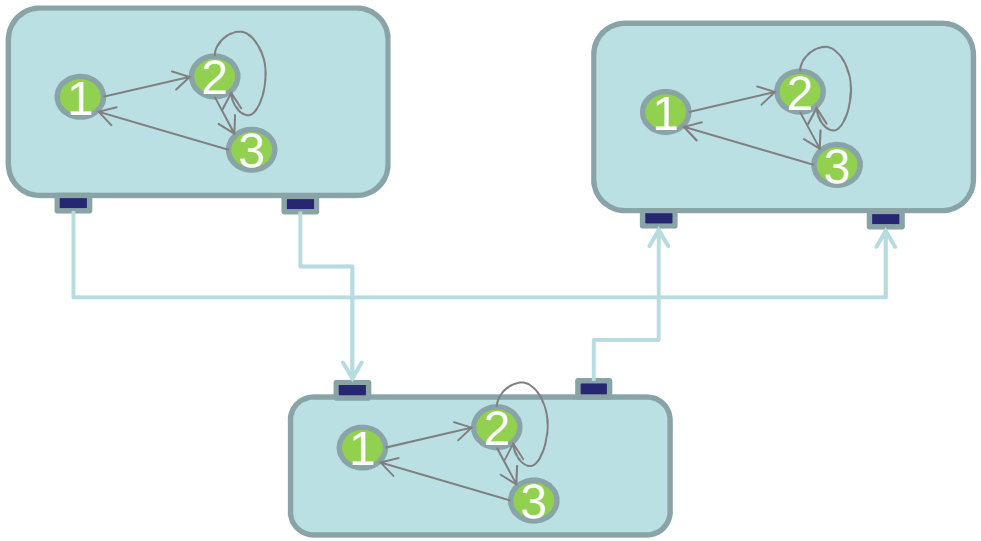
Background

- ISF was developed as part of a technology task at NASA JPL.
  - Explore new flight hardware
  - Explore new software approaches
  - Targeted at smaller projects like instruments, Cubesats, and smaller Smallsats
  - Sparser processor resources (e.g. 2MB memory, 128K program space)
    - TI MSP430, ARM-M*, LEON3
  - Farms of smaller interconnected processors
- Goals were to show:

Goal	Explanation
Reusability	Frameworks and adaptations readily reusable
Modularity	Decoupled and easy to reassemble
Testability	Easily tested in isolation
Adaptability	Should be adaptable to new contexts and bridge to inherited
Portability	Should be portable to new architectures and platforms
Usability	Should be easily understood and used by customers
Configurability	Facilities in the architecture should be scalable and configurable
Performance	Architecture should perform well in resource constrained contexts. Should be very compact.

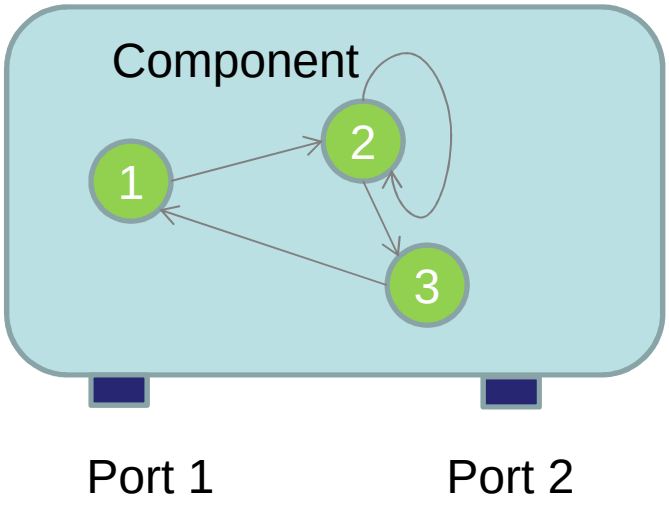
ISF: A Component Architecture

- The ISF Component Architecture is a design pattern based on an architectural concept combined with an architectural framework.
- Not just the concepts, but framework classes and tools are provided for the developer/adapter.
- Implies patterns of usages as well as constraints on usage.
- Centered around the concept of “components” and “ports”
- Leverages code generation for framework classes



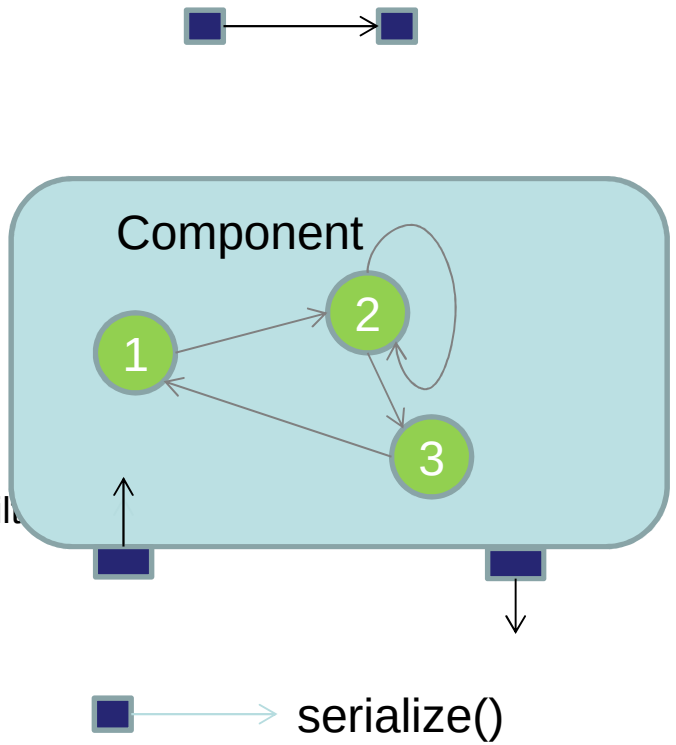
Characteristics of Components

- Encapsulates behavior
- Components are not aware of other components
- Localized to one compute context
- Interfaces are via strongly typed ports
  - Ports are formally specified interfaces
  - No direct calls to other components
- Consists of three kinds: Passive, Queued and Active
  - Explanation coming

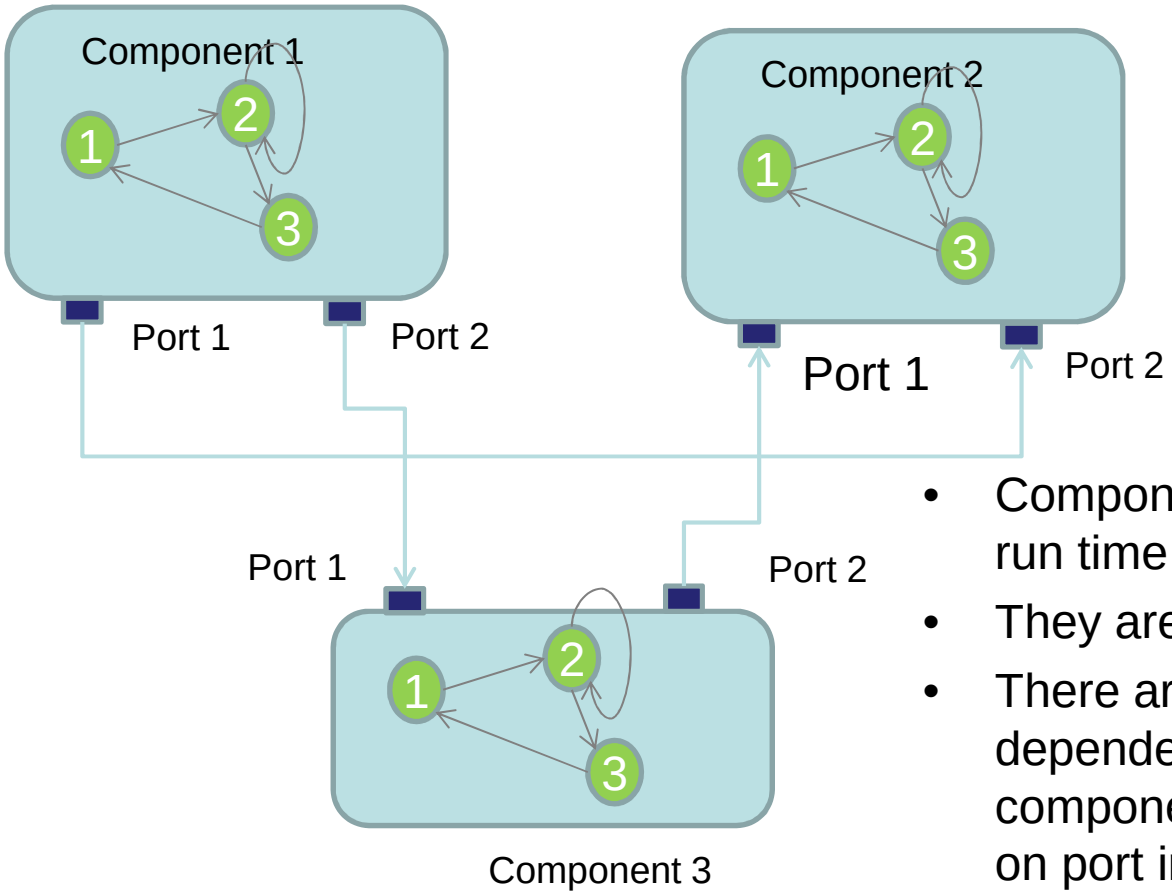


Characteristics of Ports

- Encapsulates typed interfaces in the architecture.
- Point of interconnection in the architecture.
- Ports are directional; there are input and output ports
- Ports can connect to 3 things:
  - Another typed port
    - Call is made to method on attached port
  - A component
    - Incoming port calls call component provided callback
  - A serialized port
    - Port serializes call and passes to generic serialized interface (more to come)
- All types in the interface call must be serializable. Built in types are supported; user types must derive from Serializable and implement serialization methods.
- Ports can have return values, but that limits use
  - Only with synchronous/locked ports
  - No serialization
- Pointers/references allowed for performance reasons

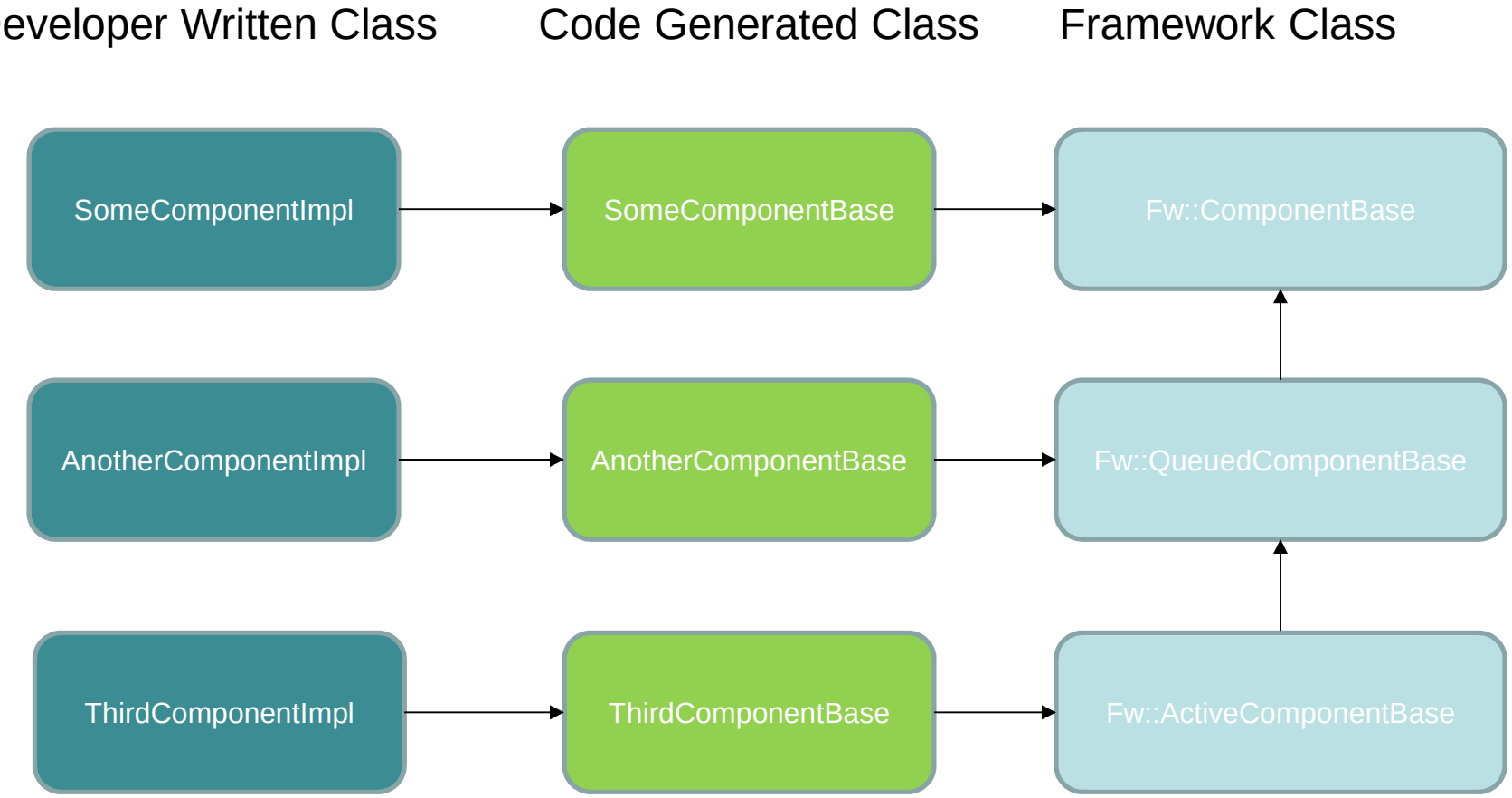


A Component Topology



- Components are instantiated at run time
- They are then connected via ports
- There are no symbolic dependencies between components, just dependencies on port interface types
- Alternate versions, such as simulation, can be substituted

Component Type Hierarchy

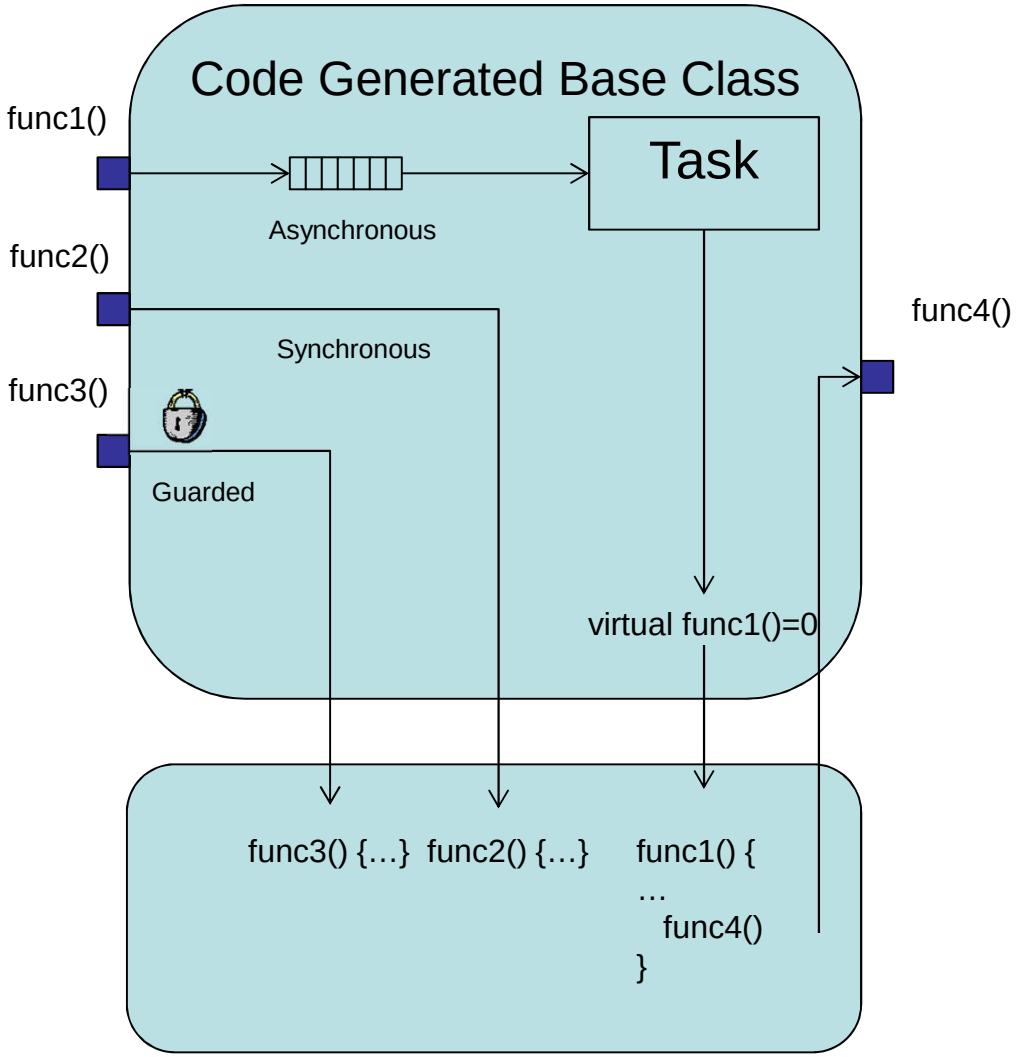


Component Types

- Passive Component
  - No thread
  - Port calls are made directly to user derived class
- Queued Component
  - No thread
  - A queue is instantiated, and port calls are serialized and placed on queue.
  - Derived class makes call to base class functions to dispatch calls
    - Thread of execution provided by caller to this component
- Active Component
  - Component has thread of execution as well as queue
  - Thread dispatches calls from queue as it receives cycles
- In all cases, calls to output port are on thread that invokes derived class functions

Port Characteristics

- The way incoming port calls are handled is configured by the code generator.
- Input ports can have three characteristics:
  - Synchronous – port calls directly invoke derived functions without passing through thread
  - Guarded – port calls directly invoke derived functions, but only after getting a lock shared by all guarded ports in component
  - Asynchronous – port calls are invoked on thread of active components
- Output ports are called synchronously from implementer's functions
- Serialized ports can accept serialized data instead of typed calls.
  - Any port can connect to a serialized port
  - When the connection is detected, the typed port serializes that data prior to invoking port
  - Serialized port doesn't need to know anything about type of calling port
  - Enables a whole class of generic transport and storage classes that increase the reusability of components

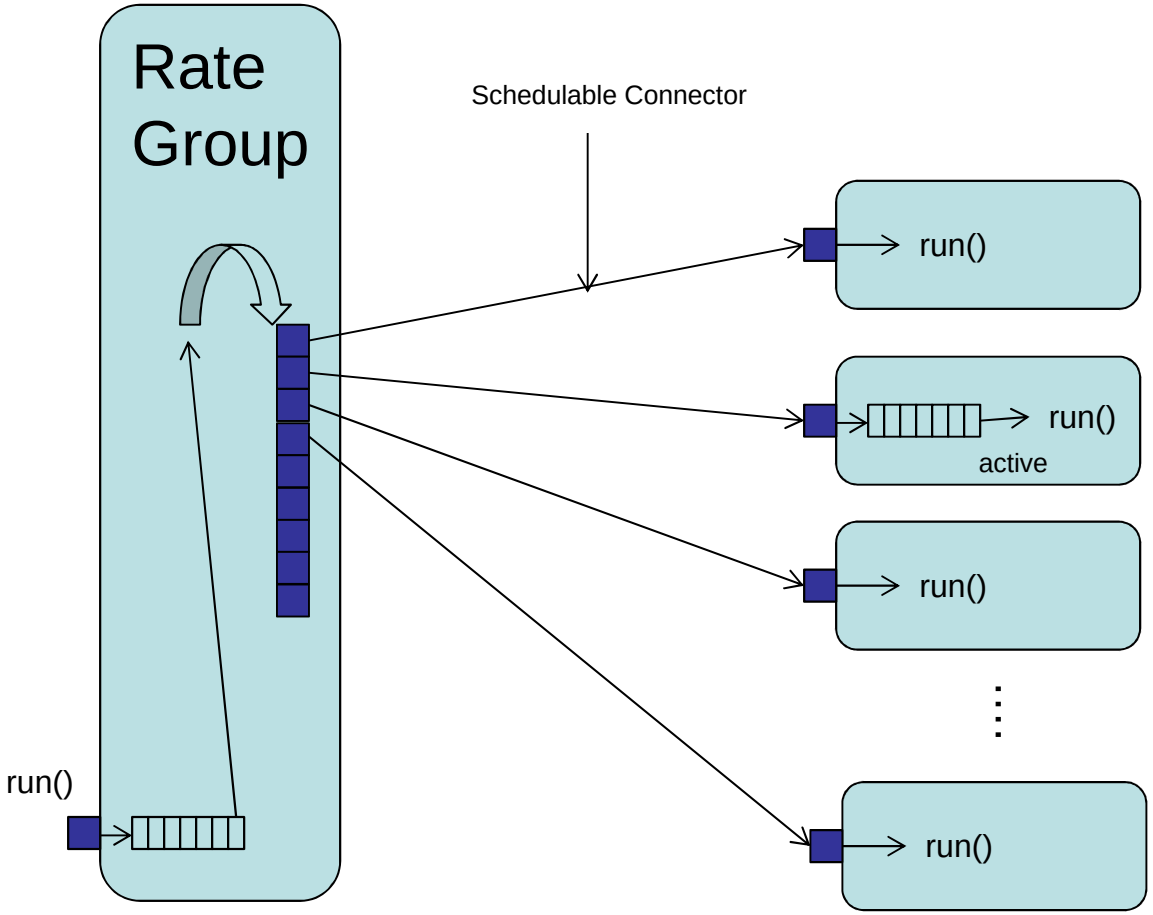


Serialization

- Serialization is a key concept in the framework
- Serialized forms of interface calls are used for message queues and to move across memory spaces (more to come).
- Data storage in the generic components can be in the form of serialized values
- Provides a uniform way of passing and storing data
- Allows flexibility in the kinds of data stored.
  - Logging, telemetry and parameters can have arbitrary types stored without having to know anything about the type
- User types can be serialized by deriving from Serializable base type and implementing serialization functions.
  - Code generation for simple structures
- Serialized into Serialization buffers which are passed around
  - Differing buffer sizes and types depending on context

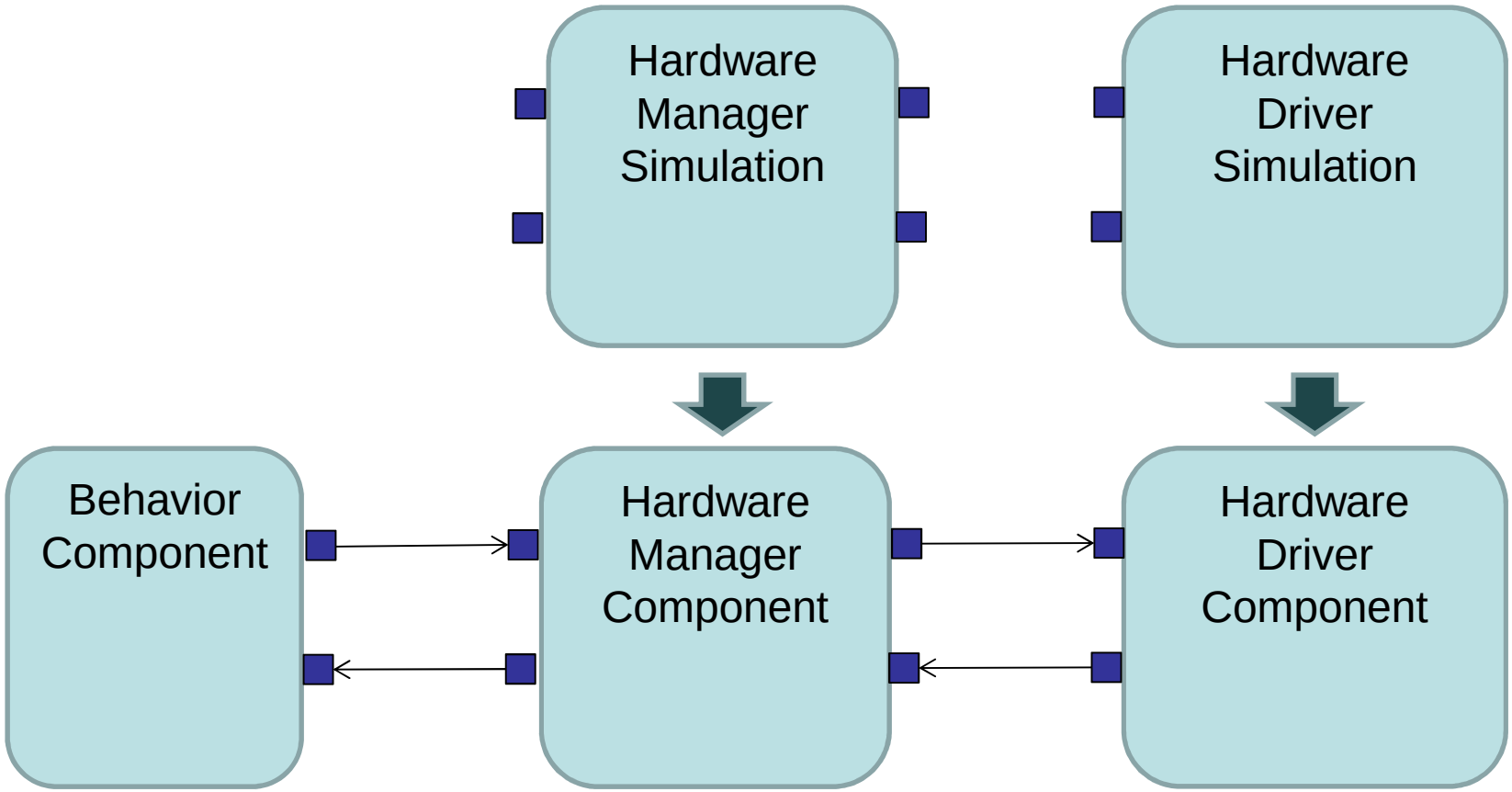
Component Pattern - Rate Group

- Rate group is a container of run() ports.
- Has an ordered list for run order
- Since is a list of run ports, doesn't know (or care) which destinations are in active components or not
- Rate Group is an active component



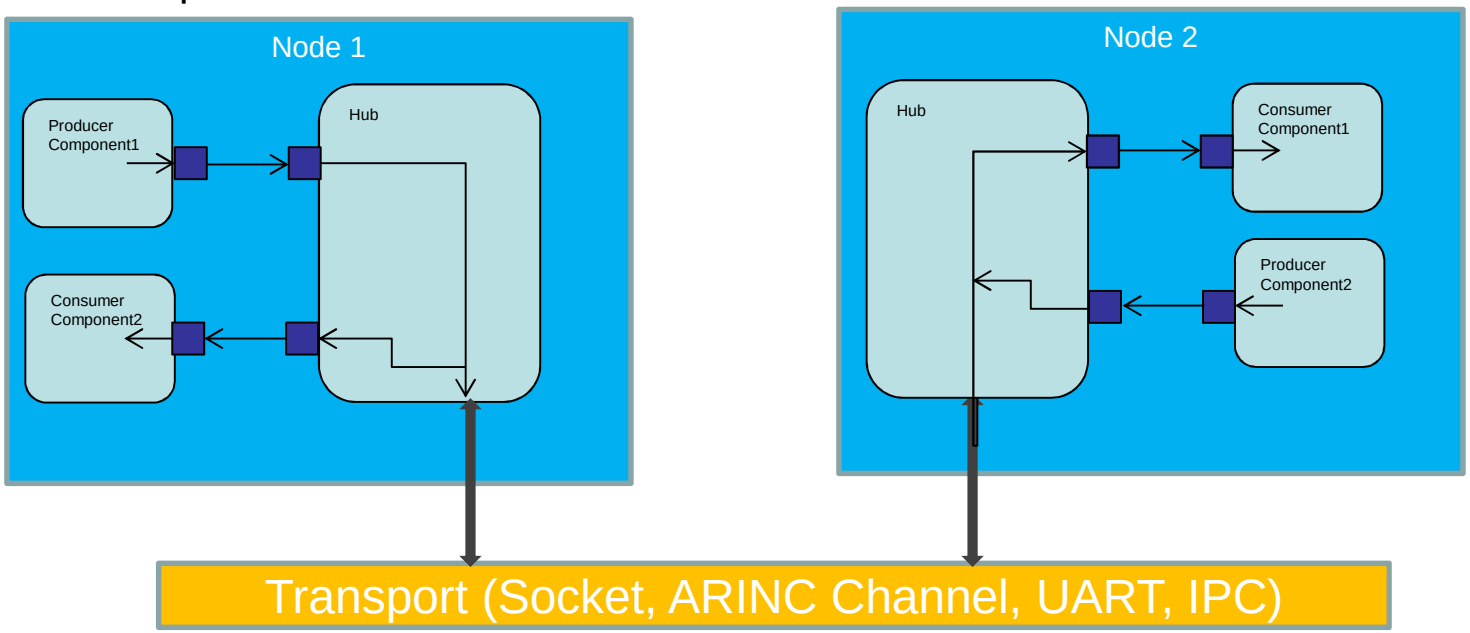
Simulation

- Simulation components or components that bridge to simulation can be substituted for components at whatever level makes sense.



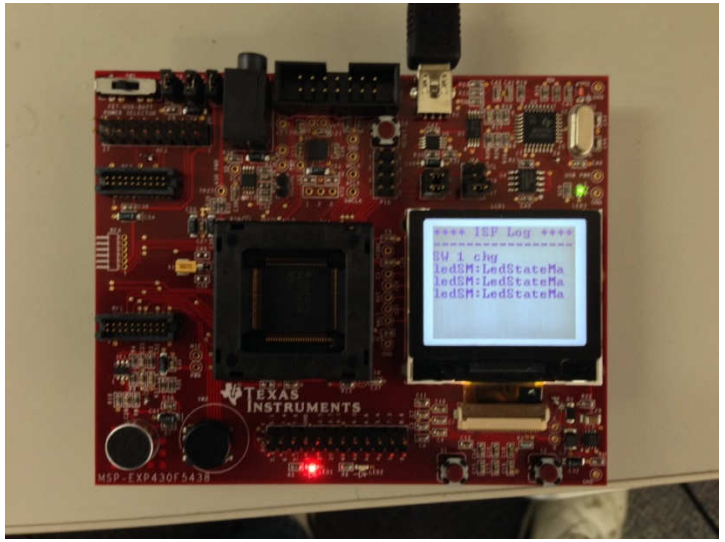
Multi-node

- Hub pattern
  - Hub is a component with multiple serialization input and output ports
  - Typed ports on calling components are connected to serialized ports (see earlier slides)
  - Each hub instance is responsible for connecting to a remote node
  - Input port calls are repeated to corresponding output ports on remote hub
  - Single point of connection to remote node, so central point of configuration for transport.



Small-scale Deployment

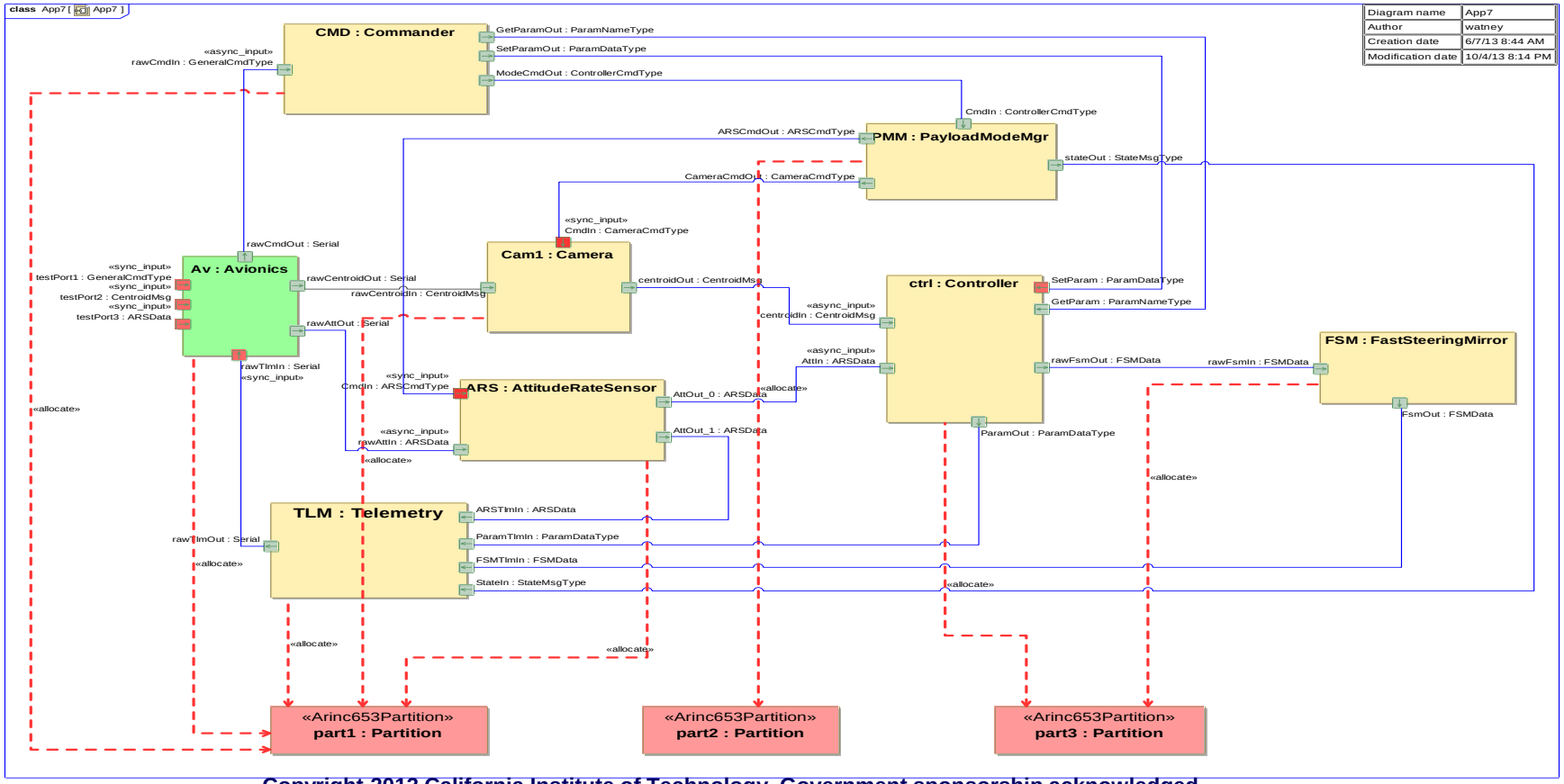
- INSPIRE, a Cubesat mission, is flying a TI MSP430F2618 microcontroller
  - 116K flash (for code), 8K RAM
- We ported ISF to a experimenter's board similar to the flight processor.
- We implemented a simple set of components that use the ISF as well as the QF (Quantum Framework)



Software Component	Memory Size	Description
ISF Framework	2.9K Flash	Includes base classes and types
ISF Adaptations	7K Flash	Includes auto-coded and user code for implementation.
Component Topology	1K heap	Components are dynamically allocated at startup

UML Modeling and Generation

- Research task at JPL wrote UML plug-ins to generate XML for components
- Modeled a JPL mission and demonstrated model-to-code process
- See other talk for details



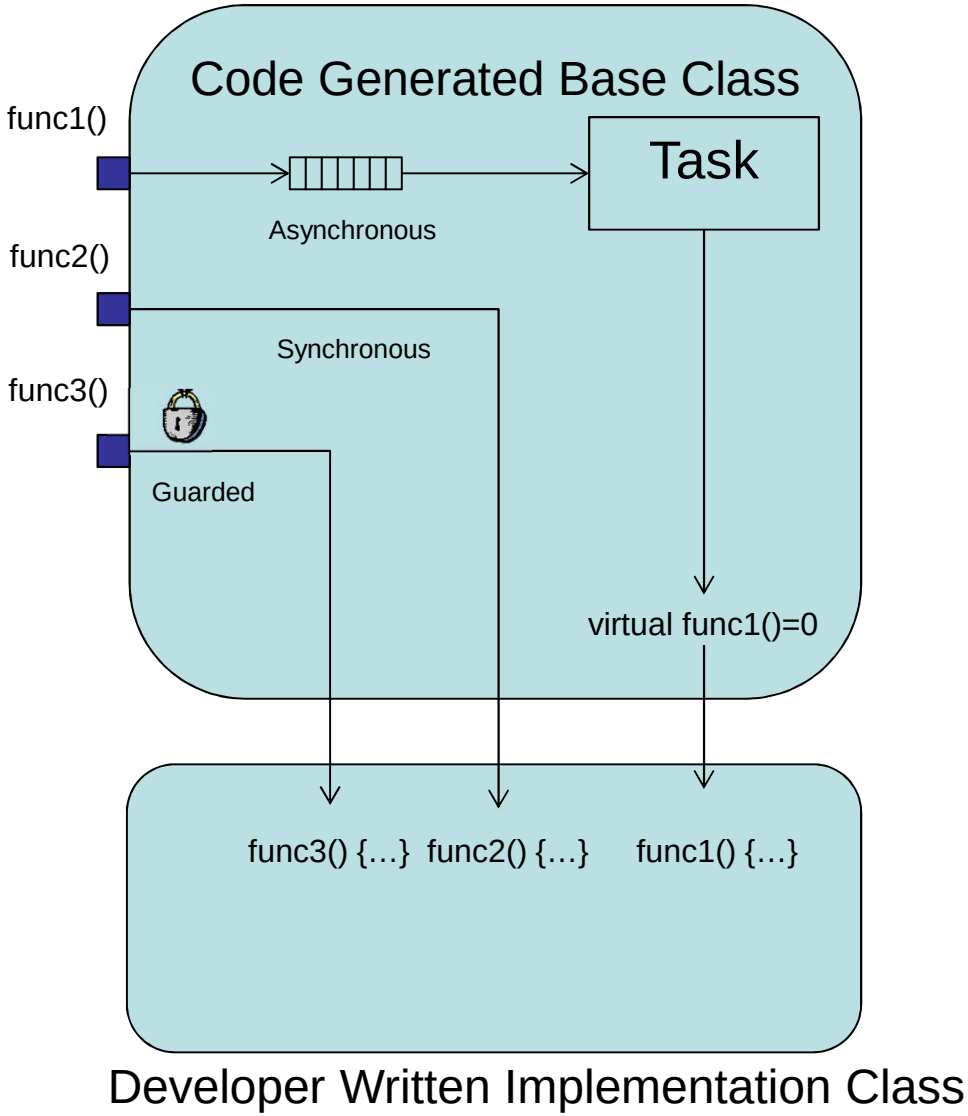
Status

- Rev 1.0 of the architecture is complete
- Rev 0.5 is being flown on RapidScat, an ISS radar experiment
- Has been ported to:
  - Linux, VxWorks, ARINC 653, No OS
  - PPC, Leon3, x86, ARM (A15/A7), MSP430
- Hubs demonstrated on:
  - Sockets
  - ARINC 653 Channels
  - High-speed flight interface
- In proposals for other projects at JPL
- Further work to do in “robustifying” the code generator for illegal combinations, etc.
- Add notion of commanding to framework rather than as an adaptation

Backup Slides

Input Port Characteristics

- The way incoming port calls are handled is configured by the code generator.
- Input ports can have three characteristics:
  - Synchronous – port calls directly invoke derived functions without passing through thread
  - Guarded – port calls directly invoke derived functions, but only after getting a lock shared by all guarded ports in component
  - Asynchronous – port calls are invoked on thread of active components
- A passive component can have synchronous and guarded ports, but no asynchronous ports since there is no queue. Calls execute on the thread of the calling component.
- A queued component can have all three port types, but it needs at least one synchronous or guarded port to unload the queue (see later slides), and at least one asynchronous port for the queue to make sense.
- An active component can have all three varieties, but needs at least one asynchronous port for the queue to make sense.
- Designer needs to be aware of how all the different call kinds interact

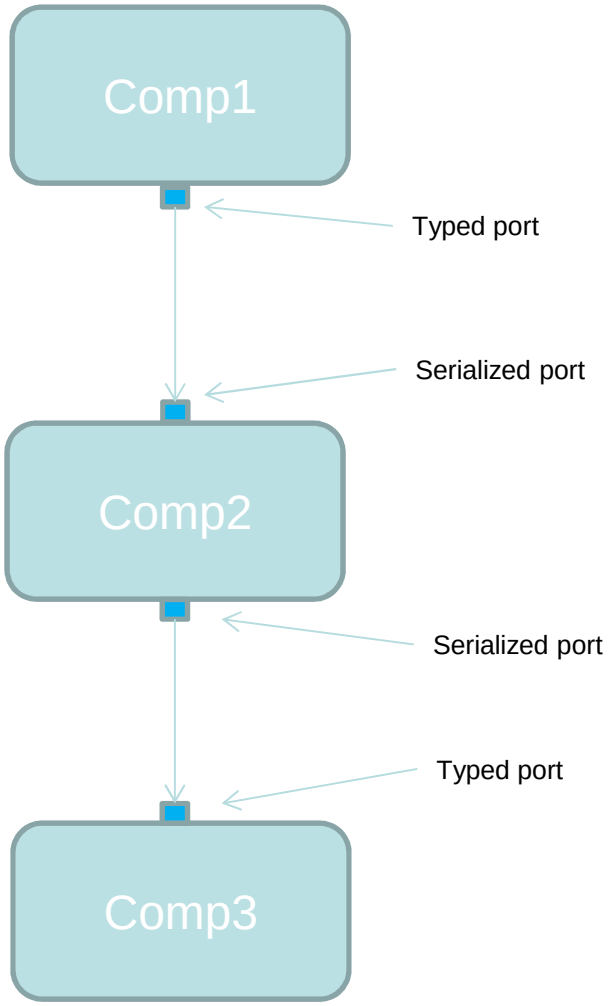


Serialization

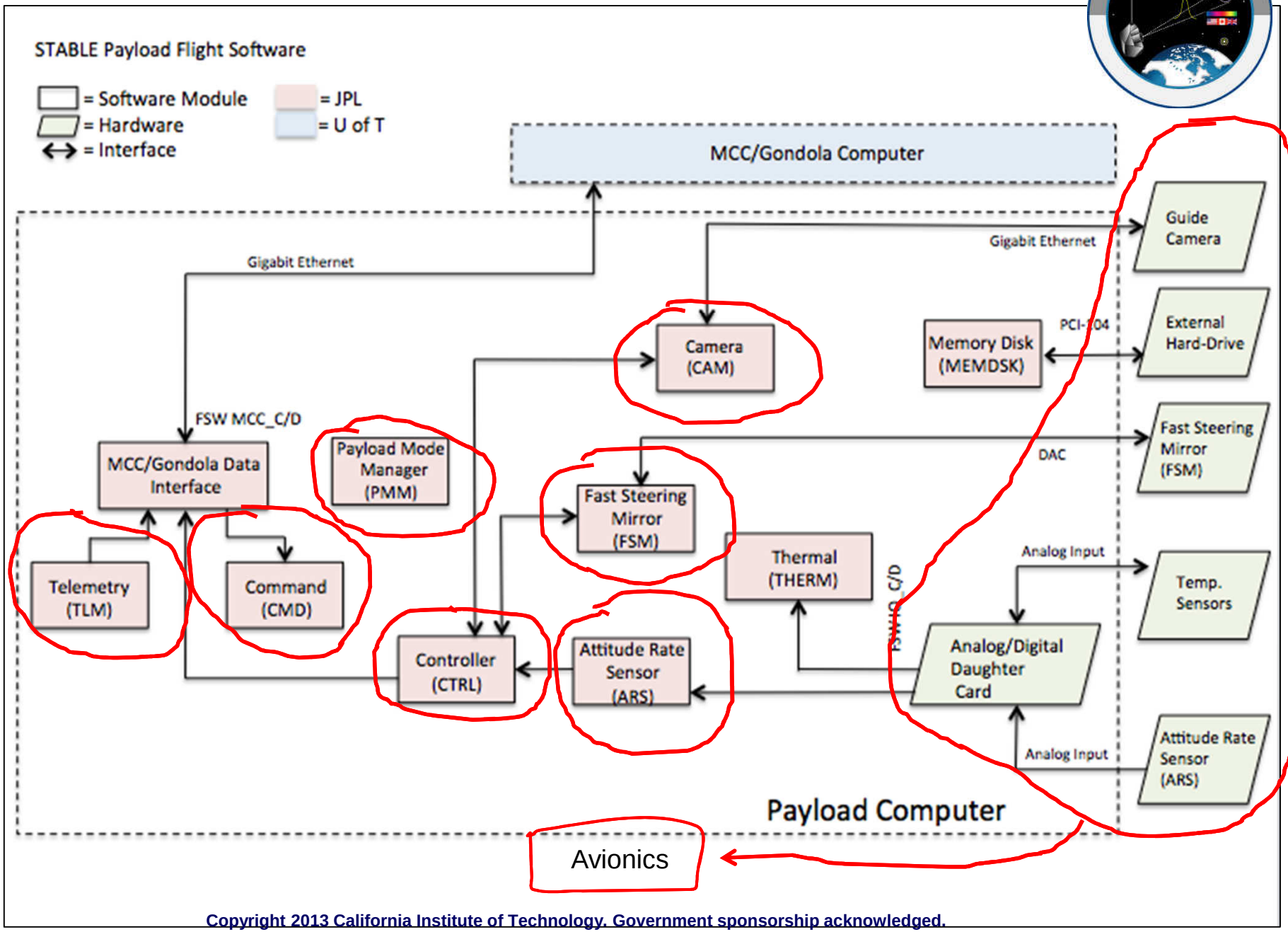
- Serialization is a key concept in the framework
- Serialized forms of interface calls are used for message queues and to move across memory spaces (more to come).
- Data storage in the generic components can be in the form of serialized values
- Provides a uniform way of passing and storing data
- Allows flexibility in the kinds of data stored.
  - Logging, telemetry and parameters can have arbitrary types stored without having to know anything about the type
- User types can be serialized by deriving from Serializable base type and implementing serialization functions.
  - Code generation for simple structures
- Serialized into Serialization buffers which are passed around
  - Differing buffer sizes and types depending on context

Serialization Ports

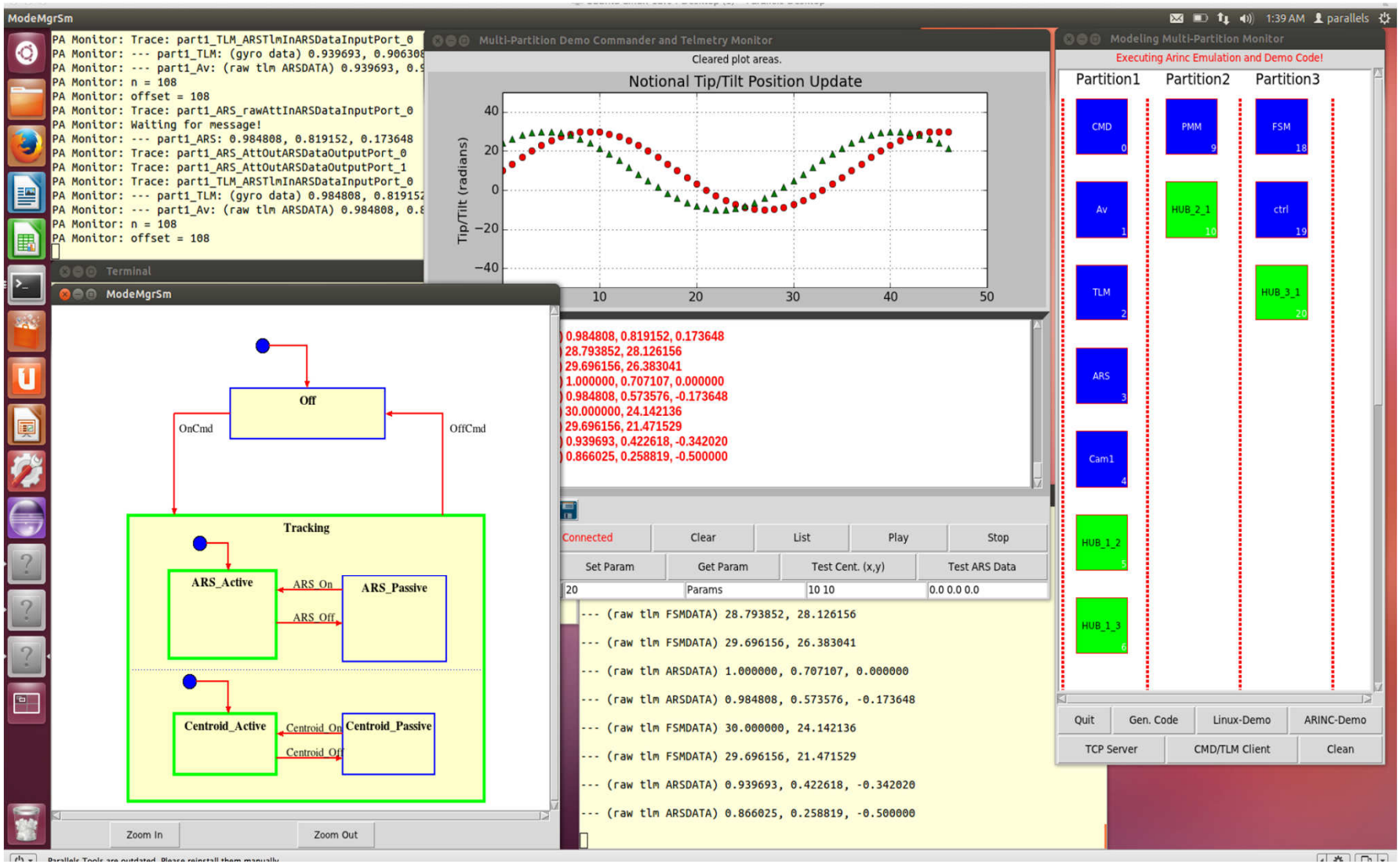
- A special, non-type specific port interface.
- Takes as input a serialized buffer when it is an input port, and outputs a serialized buffer when it is an output port.
- Can be connected to *any* typed port.
  - For input port, calling port detects connection and serializes call
  - For output port, serialized port calls interface on typed port that deserializes call.
  - Not supported for ports with return types
- Useful for generic storage and communication components that don't need to know type
  - Makes them much more reusable
  - e.g. Hubs (see later slides), telemetry, parameters, data products



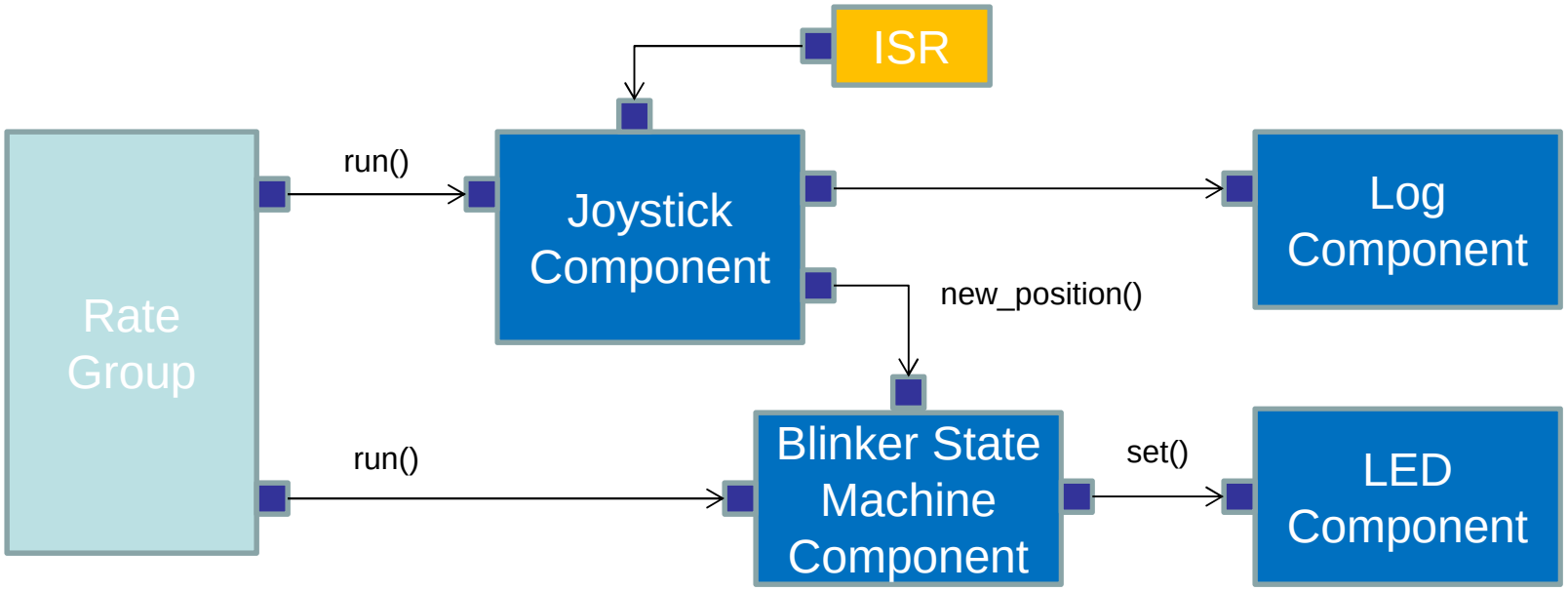
STABLE Instrument PDR



Stable Demo ARINC653 Emulation (three partition) Deployment



TI Microcontroller ISF Component Adaptation



- The rate group executes at ~2Hz
  - Calls the joystick and blinker state machine components
  - Blinks a second LED as a heartbeat
- ISR (Interrupt Service Routine) executes asynchronously when joystick is pushed. Position is reported to joystick component.
- When rate group executes joystick component, it checks for change in position. If it detects one, it reports it to the state machine component
- Depending on the position, the state machine either turns off LED, turns on LED, or blinks LED. The QF state machine on the next slide is implemented in this component.