F` Software Framework

A Small Scale Component Framework for Space

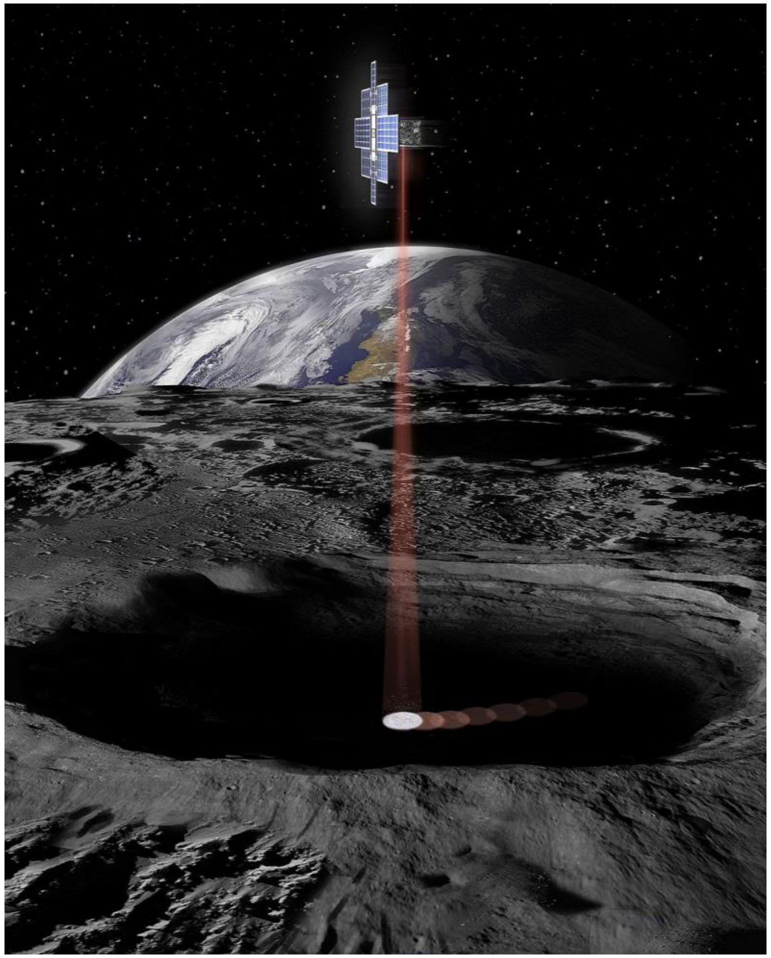
Jet Propulsion Laboratory,

California Institute of Technology

10/3/2020

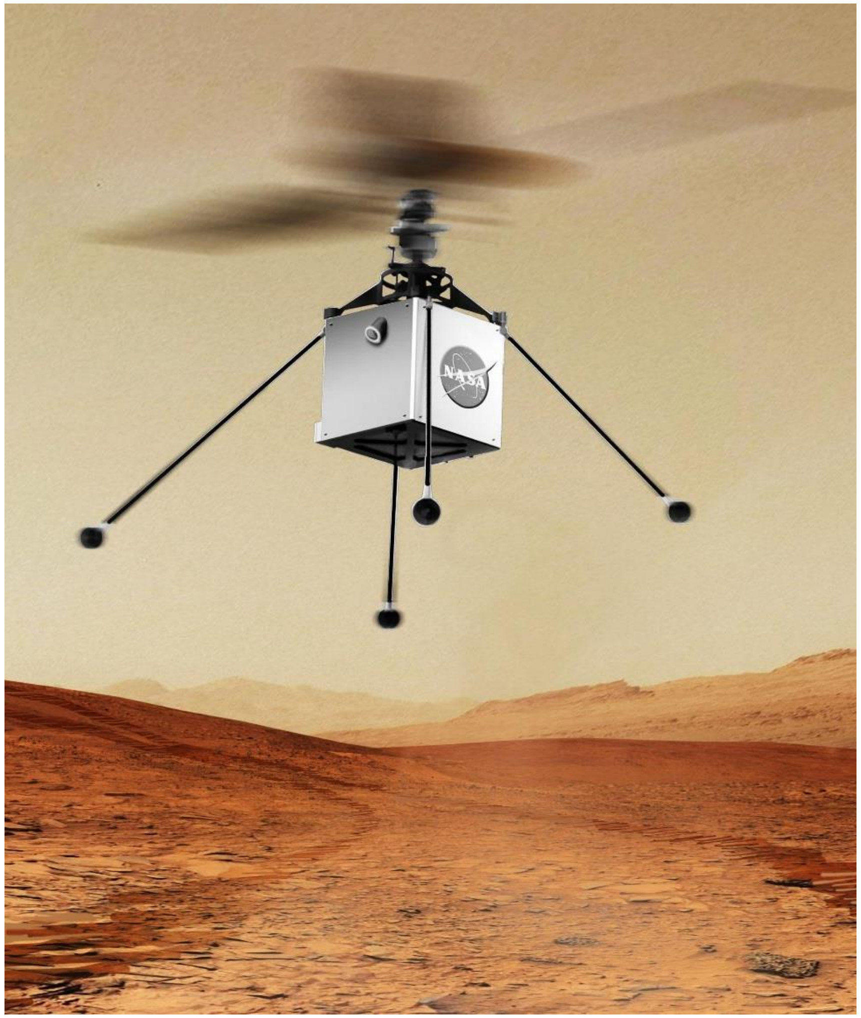
What is F´?

- Embedded Systems Framework
- Flight Software for Small-Scale Projects
  - Cubesats, Deployables, Instruments
- Provides:
  - Framework Autocoder
  - Standard Components and Abstractions
  - Modeling, Testing, and Basic Ground Support



Why F´?

- Standard Components and Abstractions
- Flight Heritage
- Designed For Modularity and Reuse
- Compact Code, Efficient Implementation
- Scalability Features
- Unit Testing Framework



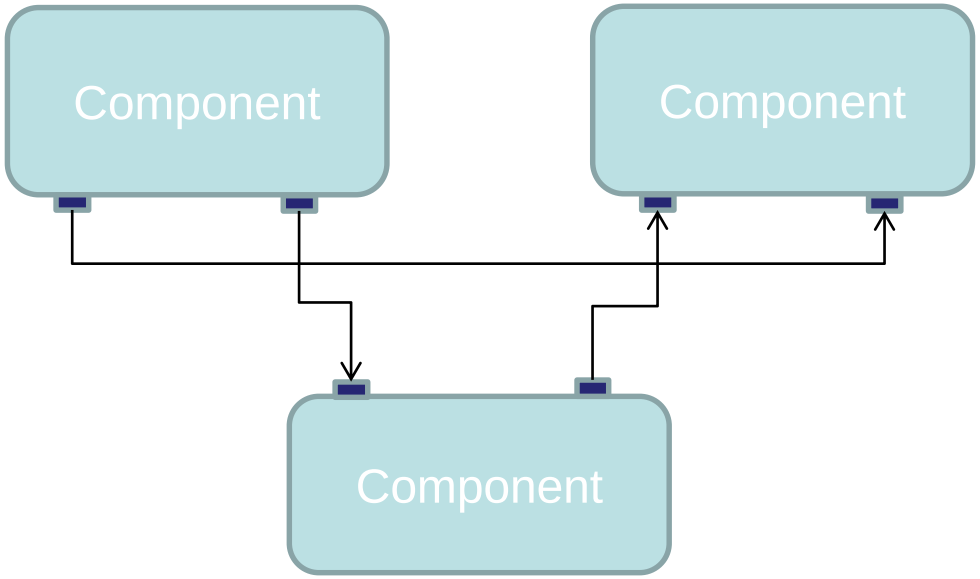
Background

- F` was developed as part of a technology development task at JPL
  - Explore new flight hardware
  - Explore new software approaches
  - Targeted at smaller projects like instruments, Cubesats, and Smallsats
  - Sparser processor resources
    - Lower amounts of memory, storage, and processor computing power
    - Microcontrollers, small embedded processors, radiation-hardened avionics
  - Clusters of interconnected processors
- Goals were to show:

Goal	Explanation
Reusability	Frameworks and adaptations readily reusable
Modularity	Decoupled and easy to reassemble
Testability	Components easily isolated for testing
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.

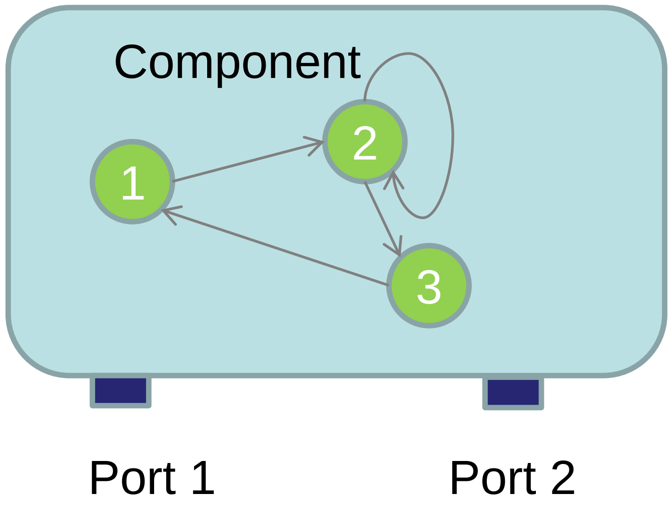
F`: A Component Architecture

- Definition: The F` Component Architecture is a design pattern based on an architectural concept combined with a software 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”
- Uses code generation to produce code to implement common framework patterns and facilities
  - Developer specifies in XML (soon to develop DSL)
- Developer writes implementation classes to implement interfaces.



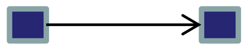
Characteristics of Components

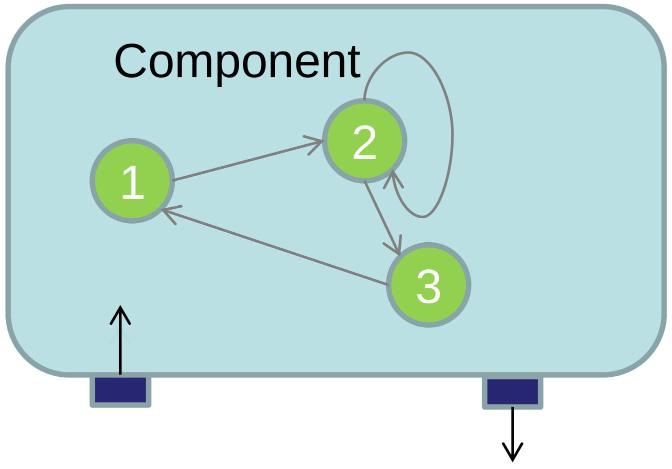
- Encapsulates behavior
- Components are not aware of other components
- Localized to one compute context
  - e.g. physical processor, user process, etc
- Interfaces are via strongly typed ports
  - Ports are formally specified interfaces
  - There are no direct calls to other components
- Context where software threads run
- Where flight/ground commands, telemetry and parameters are defined

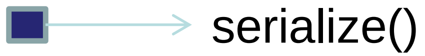


Characteristics of Ports

- Encapsulates typed interfaces in the architecture.
  - Implemented as C++ class with one interface method
- Point of interconnection in the architecture
- Ports are directional; there are input and output ports
  - Direction is direction of *invocation*, not necessarily data flow. Ports can retrieve data
- Ports can connect to 3 things:
  - Another typed port
    - Call is made to interface method on attached port
  - A component
    - Incoming port calls call component registered callback
  - A serialized port
    - Port serializes call and passes as data buffer (more to come)
- All arguments in the interface are serializable, or convertible to a data buffer. There are built-in types supported by the framework; developer can write custom types. (see later slides)
- Ports can have return values in limited cases
  - Only able to return data when component has synchronous interface (see definition later)
  - Serializable connections are defined as only output data flow
- Pointers/references are allowed for performance reasons
  - User beware!
- Multiple output ports can be connected to a single input port, but only one port connection for each output port.

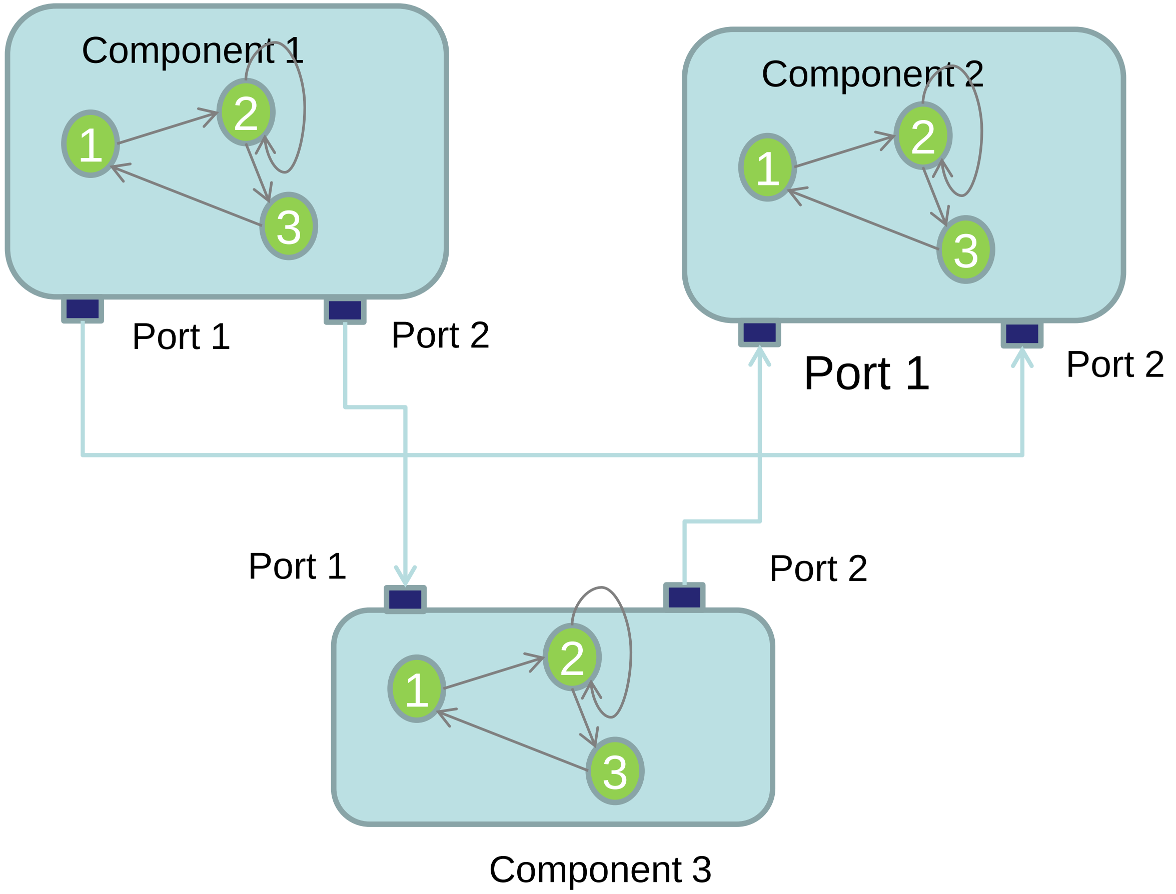






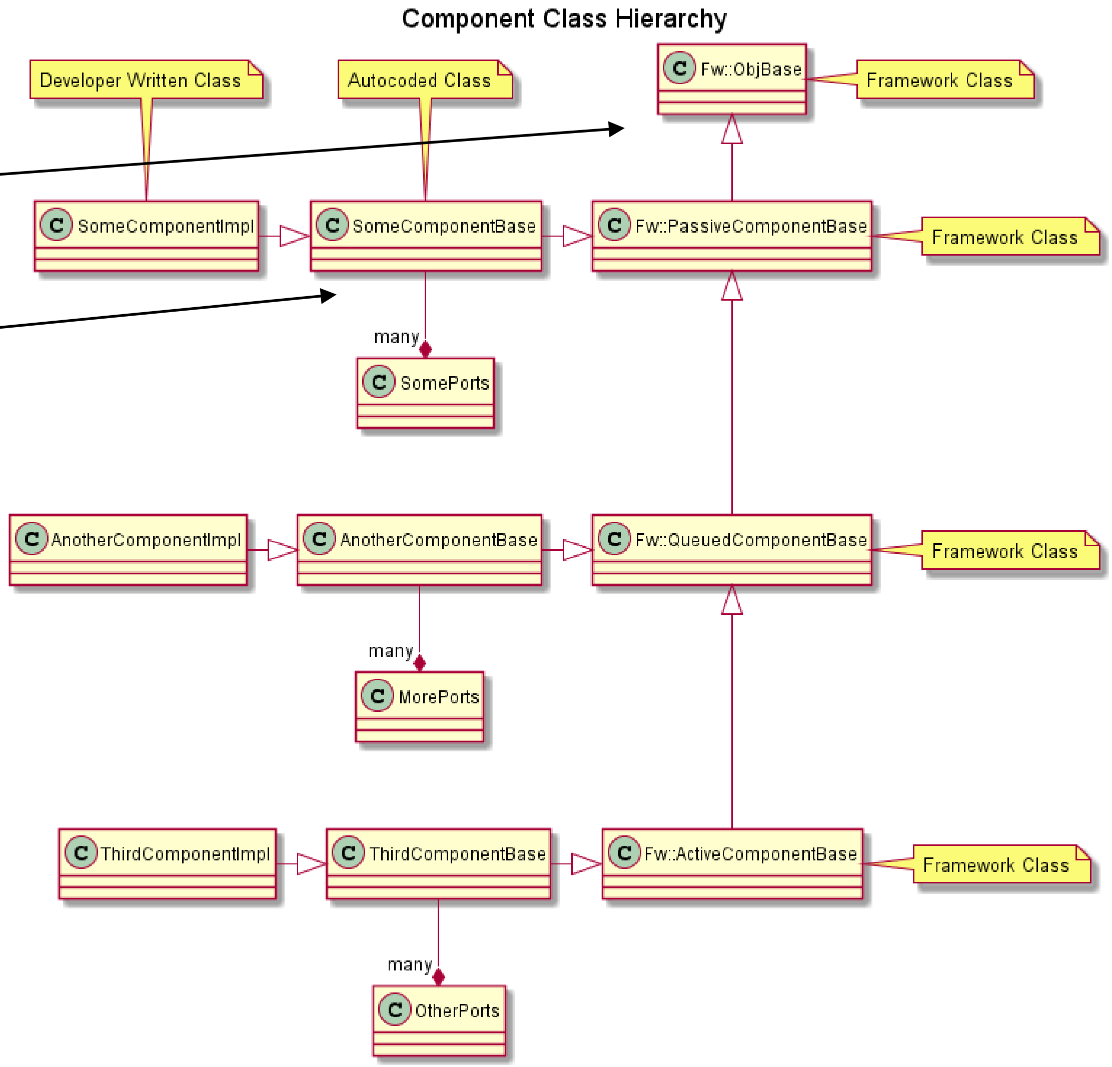
A Component Topology

- Components are instantiated at run time
- They are then connected via ports into a *Topology*, or a specific set of interconnected components
- There are no code dependencies between components, just dependencies on port interface types
- Alternate implementations can easily be swapped
  - Simulation versions, alternate drivers, etc.



Component Type Hierarchy

- Hierarchy consists of:
  - core framework classes
  - generated classes that implement architecture features
  - Developer written classes that implement interface handlers and project-specific logic

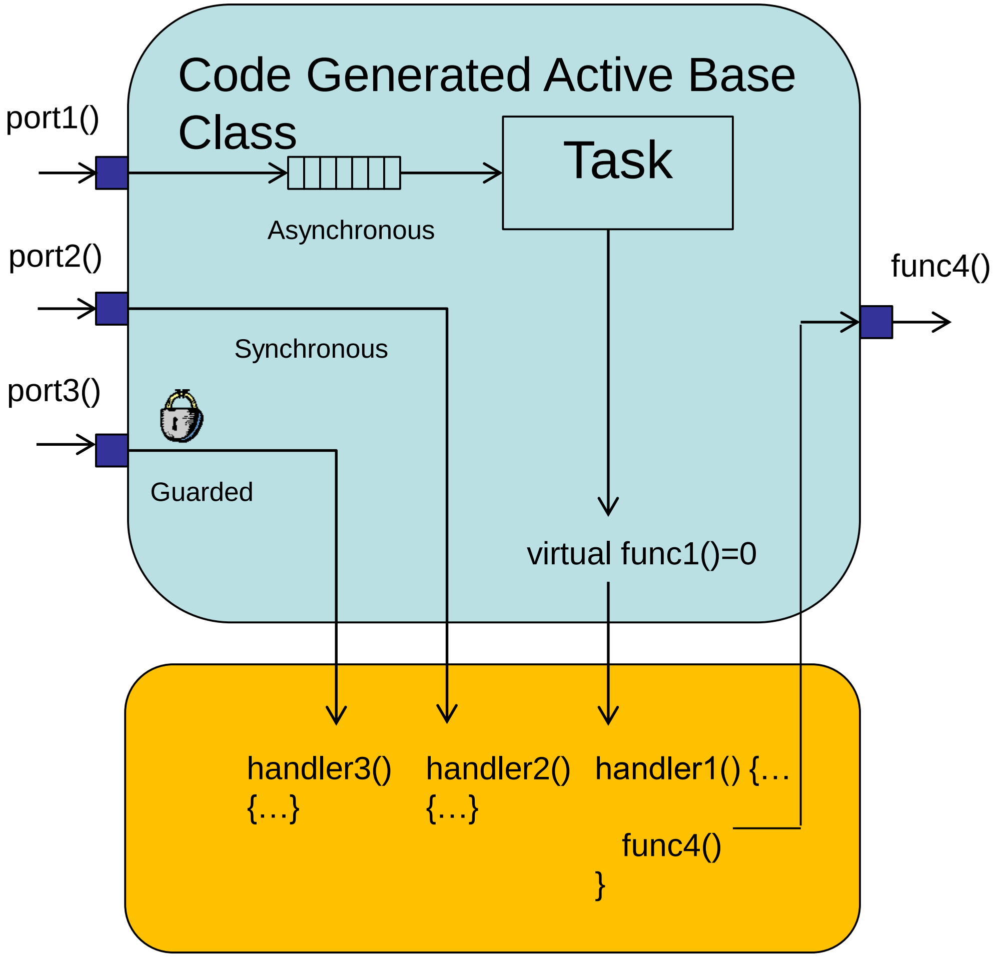


Component Types

- User specifies type of component in XML. Types are:
- Passive Component
  - No thread
  - Input port interface calls are made directly to implementation class handler methods on the thread of the caller of the input port.
- Queued Component
  - No thread, adds a message queue
  - Asynchronous port calls are placed on component message queue.
  - Implementation class calls method in base class on the thread of another caller to dispatch any messages pending in the queue.
- Active Component
  - Component has thread of execution as well as queue
  - Thread dispatches port calls from queue as it executes based on thread scheduler
- Calls to output port are on thread of implementation functions
  - Thread making call is dependent on port type (see next slide)

Port/Component Characteristics

- The way incoming port calls are handled is specified by the component XML.
- Input ports can have three attributes:
  - Synchronous – port calls directly invoke derived functions without passing through queue
  - Guarded – port calls directly invoke derived functions, but only after locking a mutex shared by all guarded ports in component
  - Asynchronous – port calls are placed in a queue and dispatched on thread emptying the queue.
- A passive component can have synchronous and guarded ports, but no asynchronous ports since there is no queue. Code executes 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 and at least one asynchronous port for the queue to make sense. Code executes on the thread of the caller.
- An active component can have all three varieties, but needs at least one asynchronous port for the queue and thread to make sense. Code executes on the thread of the caller or component depending on port type.
- Developer needs to be aware of how all the different call kinds interact (e.g. reentrancy)
- Output ports are invoked on the thread of handlers by calling methods in the generated base class.



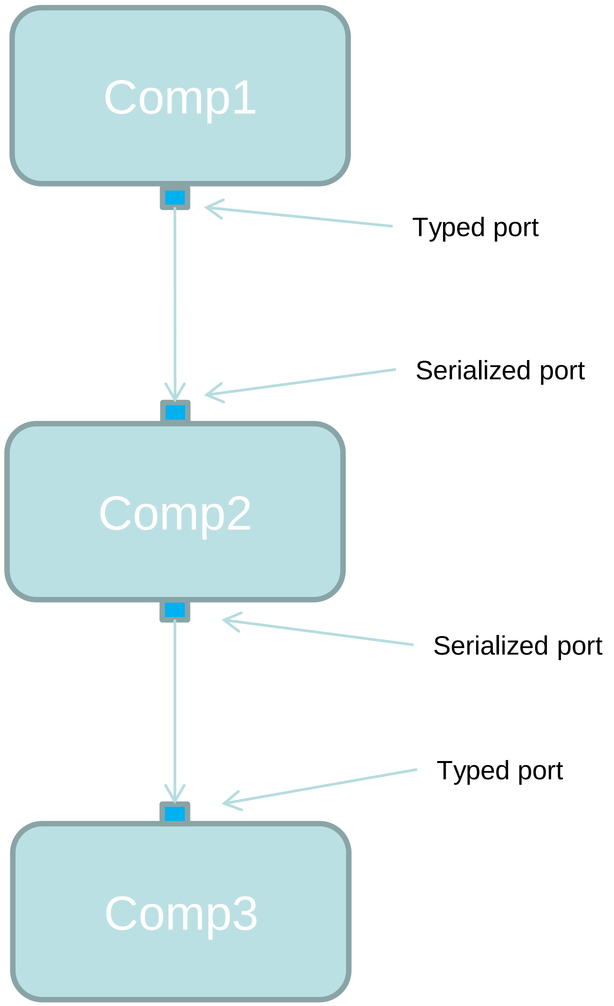
Developer Written Implementation Class

Serialization

- Serialization is a key concept in the framework
- Definition: Taking a specific set of typed values or function arguments and converting them in an architecture-independent way into a data buffer
- Asynchronous port calls and commands and their arguments are serialized and placed on message queues in components
- Commands, telemetry, and parameters are specified as serializables for transport to and from ground software
- User can use built-in types or define complex types in XML and code generator will generate serializable classes that are usable internally as well as interfaces to and from ground software
- User can also implement arbitrary serializable types with a set of required methods and the framework automatically handles them

Serialization Ports

- A special optional port that handles serialized buffers
- 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 (almost).
  - For input port, calling port detects connection and serializes arguments
  - For output port, serialized port calls interface on typed port that deserializes arguments
  - Not supported for ports with return types
- Useful for generic storage and communication components that don't need to know type
  - Allows design and implementation of C&DH (command and data handling) components that can be reused.



Commands, Telemetry, Events and Parameters

- The code generator provides a method of implementing commands, telemetry, events (AKA EVRs), and parameters for interface with a ground system.
- Component XML specifies arguments and types.
- Data for service is passed in serialized form.
- The generated code decodes serialized commands and invokes implementation handlers with the correct arguments. Telemetry and events are emitted by calling typed functions from implementation code.
- Port types for flight/ground interface are defined in normal XML, so they can be used by the code generator for implementing component-level handlers or generically by other components used to implement flight/ground functions.

Commands

- Component command XML specifies:
  - Opcode, mnemonic, and arguments
    - Arguments can be any built-in type or developer defined enumeration
  - Synchronization attribute
    - Sync, async, or guarded
    - Same meaning as ports
      - Async can specify message priority
- Implementation class implements handler for each command
  - Base class deserializes arguments from argument data buffer
- Implementation class reports a command status when done
  - Can be immediately in handler or deferred until command is complete later
  - Component must report a status to clear command dispatcher tracking and advance sequences
- Autocoder automatically adds ports for registering commands, receiving commands and reporting an execution status.

Events

- Component event XML specifies:
  - ID, name, severity and arguments
    - Arguments can be any built-in type or XML complex type
    - Can define enumerations
  - Format specifier string
    - Used by ground software and optional on-board console to display message with argument values
    - Follows C format specifier syntax
- Code generated base class provides method to call for each event with typed arguments
  - Provides stronger type checking at compile time than MER/MSL EVR macros
  - Called by implementation class
  - Automatically time-tags event at moment of method invocation
- Code generator automatically adds ports for retrieving time tag and sending event
  - Two independent ports for sending events
    - A binary version with serialized arguments for transport to ground software
    - A text port that sends a string version of the event (using the format specifier) that can be sent to a console
      - Can be globally disabled via architecture configuration macros to save execution time and code space

Telemetry

- Component telemetry XML specifies channels that have:
  - ID, name, and data type
    - Data type can be any built-in type or external XML complex type
    - Can define enumerations
  - Format specifier string
    - Used by ground software and optional on-board console to display message with argument values
- Code generated base class provides function to call for each channel with typed argument
  - Called by implementation class
- Code generator automatically adds ports for retrieving time tag and sending channelized data

Parameters

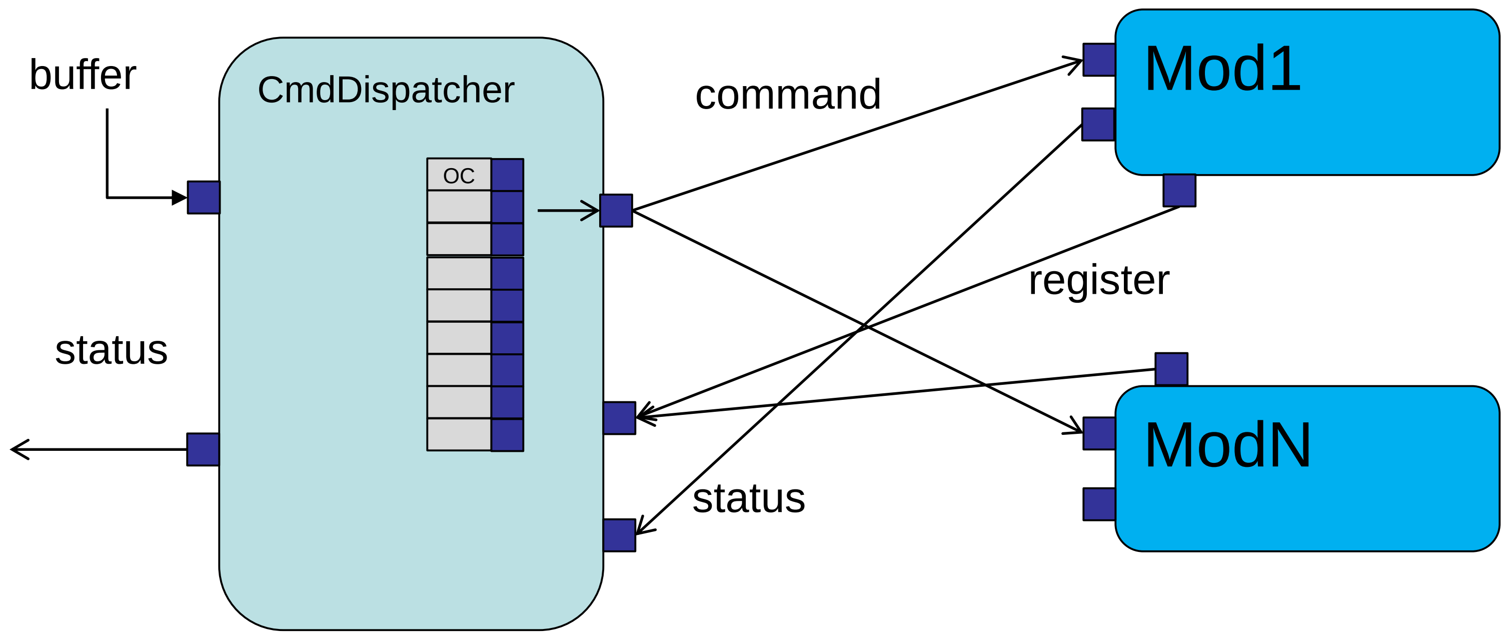
- Parameters are traditional means of storing non-volatile state
  - Framework provides code generation to set and save, but developer must write a component to load and save parameters from a particular non-volatile storage
    - File-based implementation provided with distribution
- Component XML specifies parameters that have:
  - ID, name, and data type
    - Data type can be any built-in type
    - Can define enumerations
  - Optional default value
    - In the event the parameter cannot be retrieved, assigns default value to parameter
- Code generator generates commands for setting and saving parameter values
- Code generator generates optional overridable virtual method for providing notification to implementation class when values are updated.
- Code generator automatically adds ports for retrieving and saving parameters
  - When parameters are saved by commands at the component level, the "save" port sends the parameter value to the storage component. The storage component is responsible for when and how to save parameter value
- During initialization, a public method in the base class is provided to load the parameters and store copies in private component data
- Code generated base class provides function to call for each parameter to retrieve stored copy
  - Implementation class can call whenever parameter value is needed
  - Implementation class cannot modify the parameter value, only commands can

Core Components

- A set of core components has been matured over the lifetime of F’
  - Some subjected to JPL class B flight processes
    - Reviews, inspections, full coverage unit tests and static code analysis
  - Some contributed back to F’ core repository by projects
- Provides out-of-the box implementations so that projects don’t have to start from the beginning
- Common command and data handling components
- Flight/ground interface components
- Drivers
- Helper components

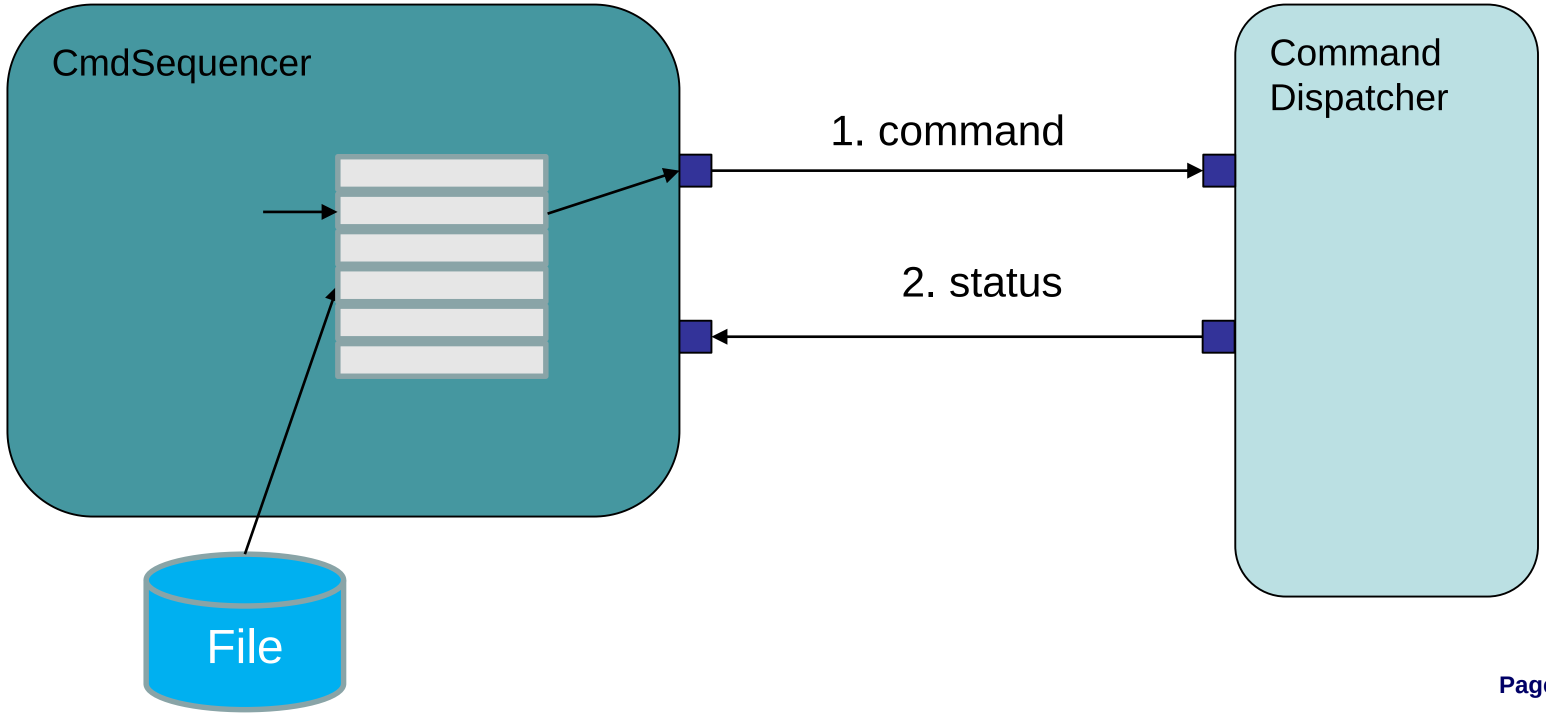
Command Dispatcher (Svc/CmdDispatcher)

- Command dispatcher is an active component
- Components register their command opcodes
- Command dispatcher receives buffer from ground or sequencer containing command and arguments
- Command opcode is extracted, and opcode lookup is made
- Component is invoked to execute commands it registered
- Component responds with completion status (pass, fail) when command complete



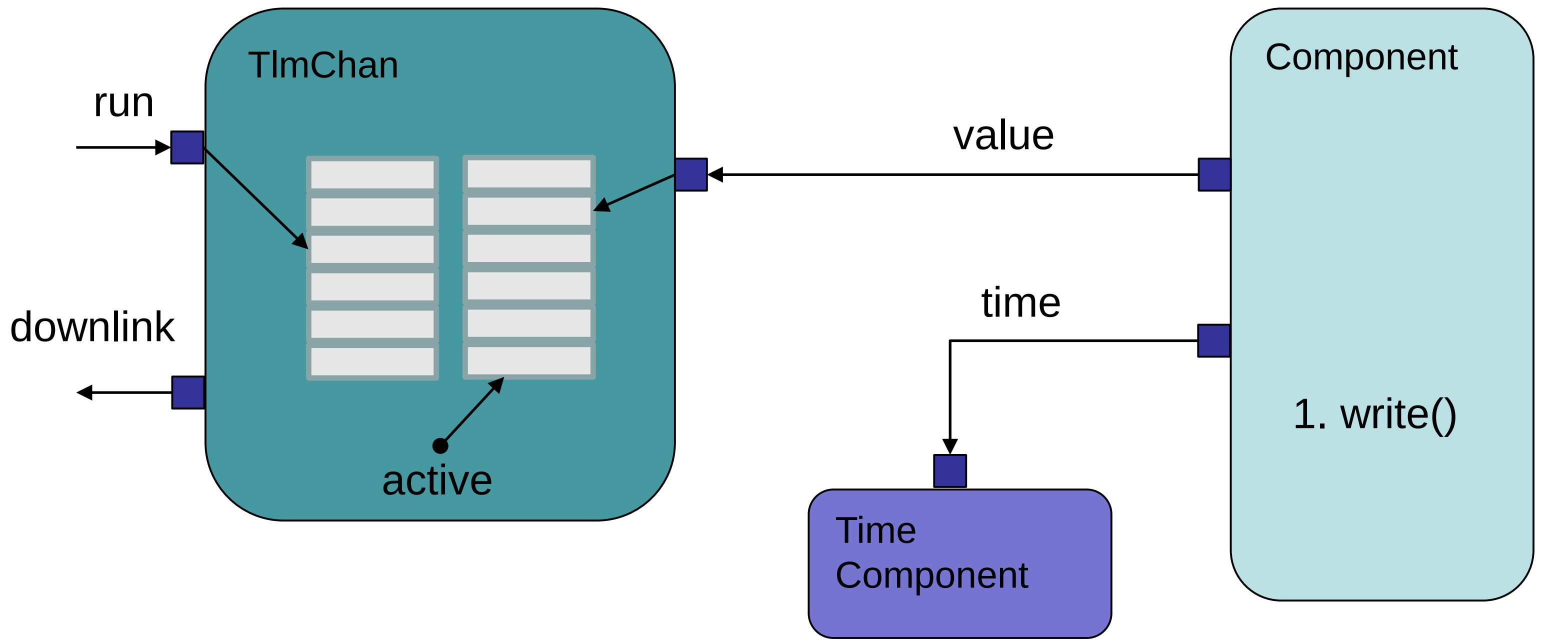
Command Sequencer (Svc/CmdSequencer)

- Command sequencer loads file from file system
- Sends command and waits for response for each command in the file
- A failed response terminates the sequence, successful response moves to the next command
- Active component



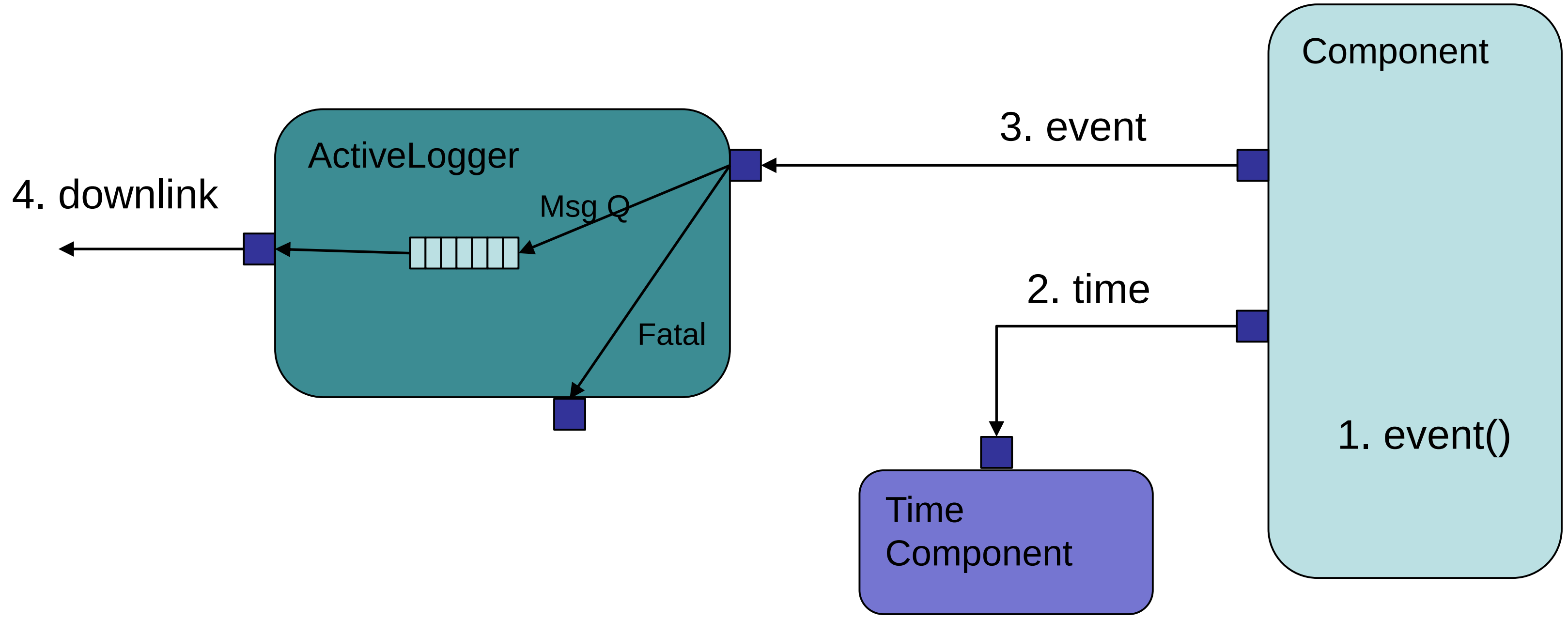
Channelized Telemetry (Svc/TlmChan)

- Telemetry database has double-buffered array of telemetry buffers
- Component writes updated value with time tag to telemetry database component
- Telemetry database writes value to active buffer
- Run port is called periodically by rate group. Swaps active buffer
- Run call copies updated values to downlink
- Active component



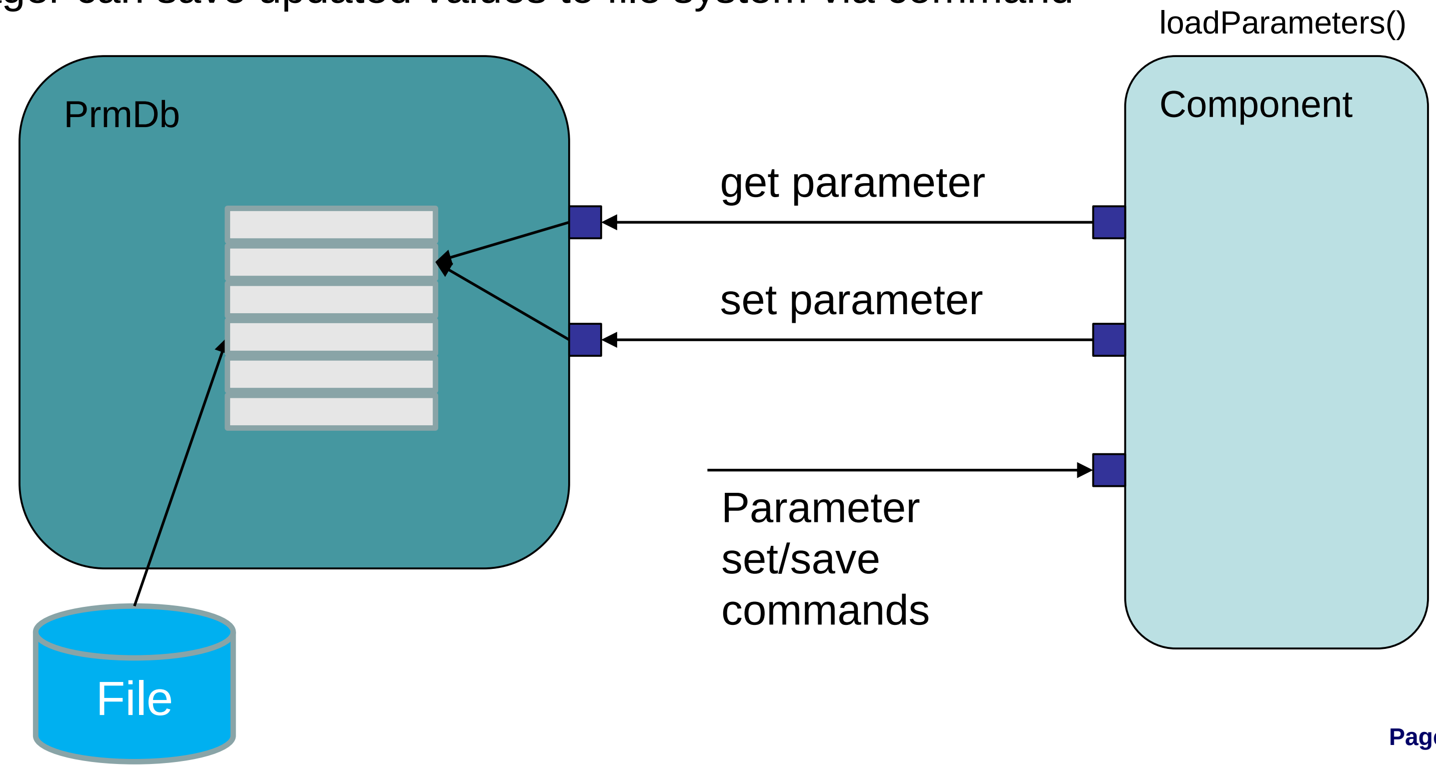
Event Log (Svc/ActiveLogger)

- Component sends time-tagged event to Event Log component
- Event log component places event on message queue. Thread of Event Log component then sends downlink packet with event
- FATAL event immediately invokes FATAL handler port on thread of calling component for special handling if desired
- Active component



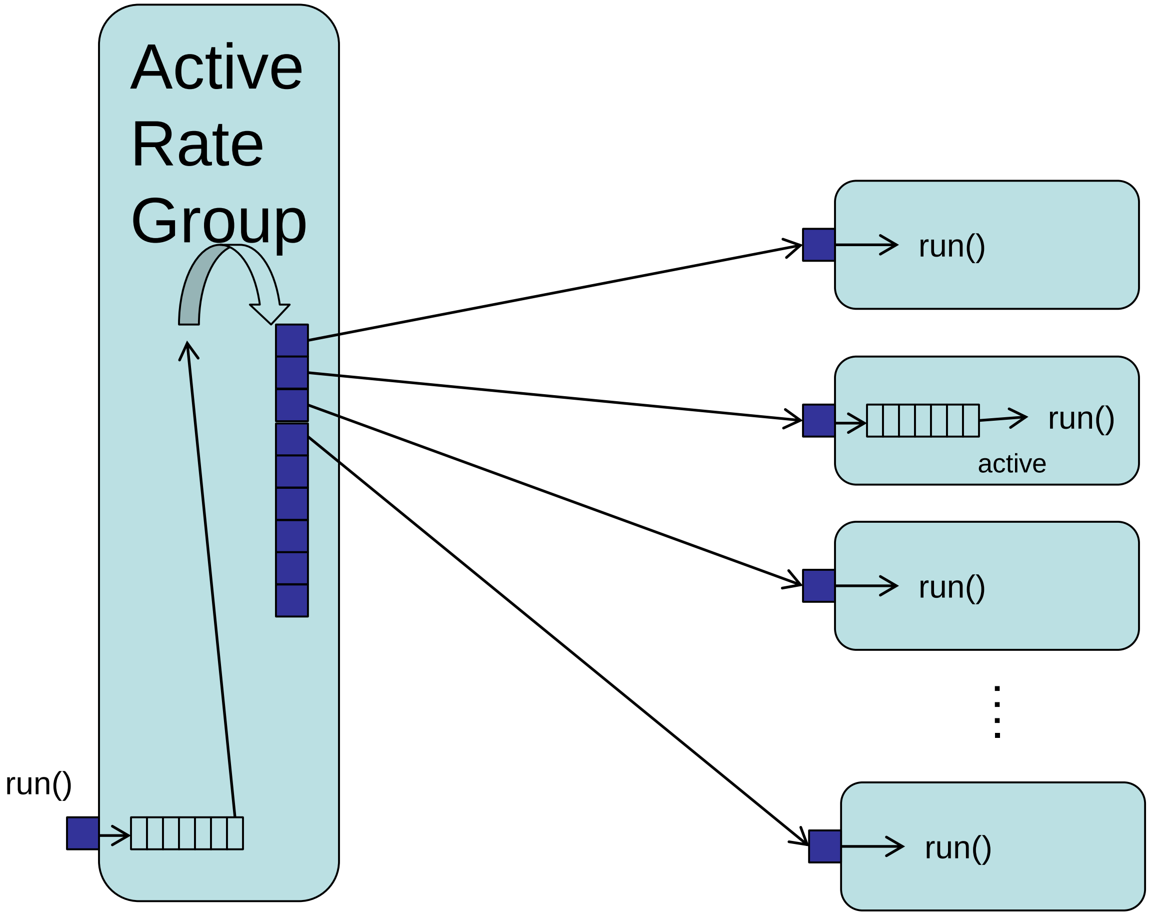
Parameter Database (Svc/PrmDb)

- Parameter Database loads file containing parameters from file system during initialization
- Initialization subsequently calls *loadParameters()* on all components with parameters.
- Commands to components can update and set parameter values via port call to parameter manager
- Manager can save updated values to file system via command



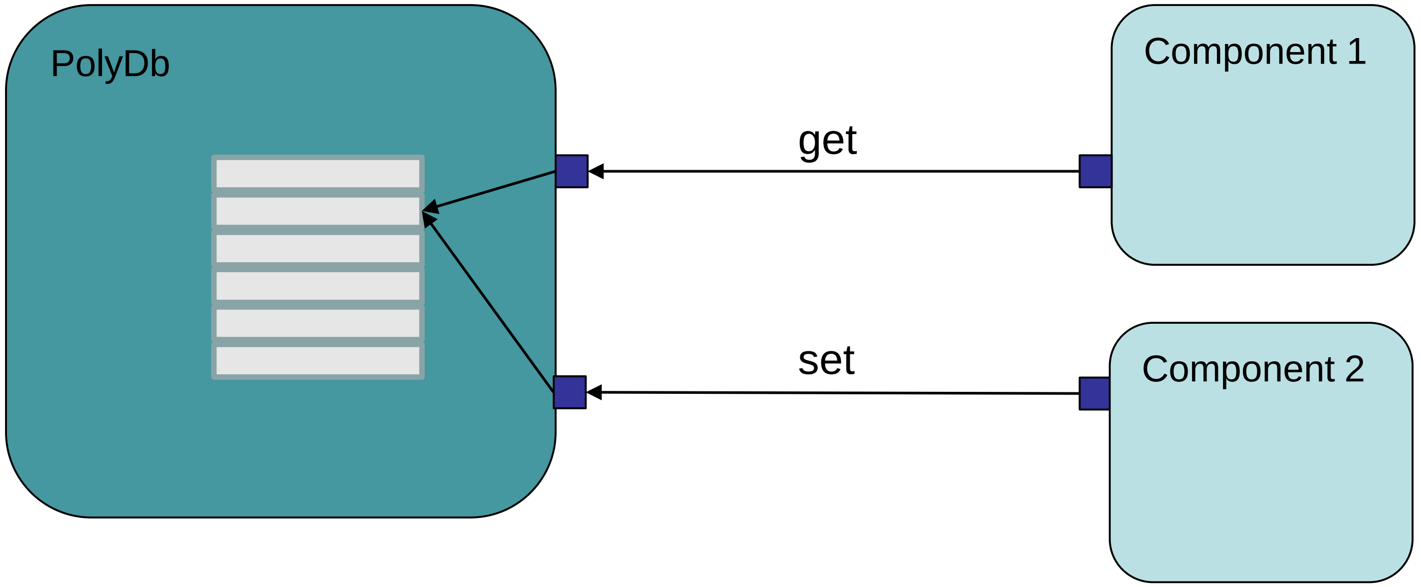
Rate Group (Svc/ActiveRateGroup)

- Rate group is a array of run() ports
- Gets a call to asynchronous run port driven by project specific timing component
- Calls ports in 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



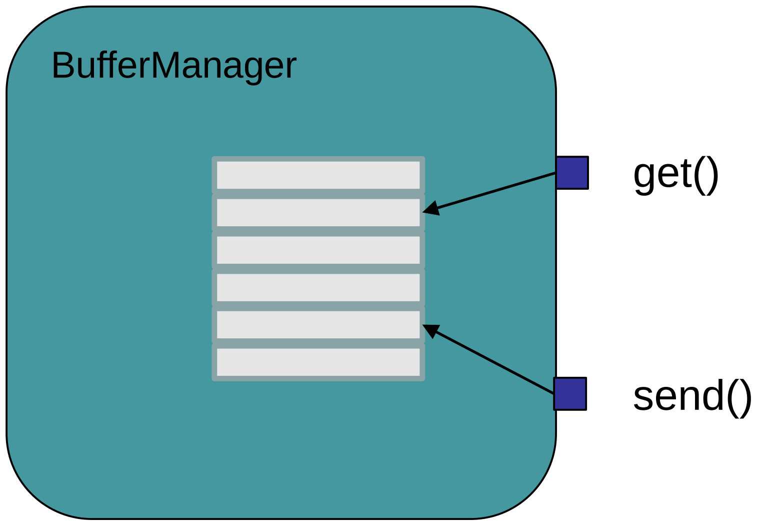
Polymorphic Database (Svc/PolyDb)

- Used to pass values around internally between components
- Uses Fw/PolyType class to allow a range of types to be stored
  - Enforces value type written is value type read
- Stored as an array accessed by integer index
  - Project assigns meaning to values stored with index



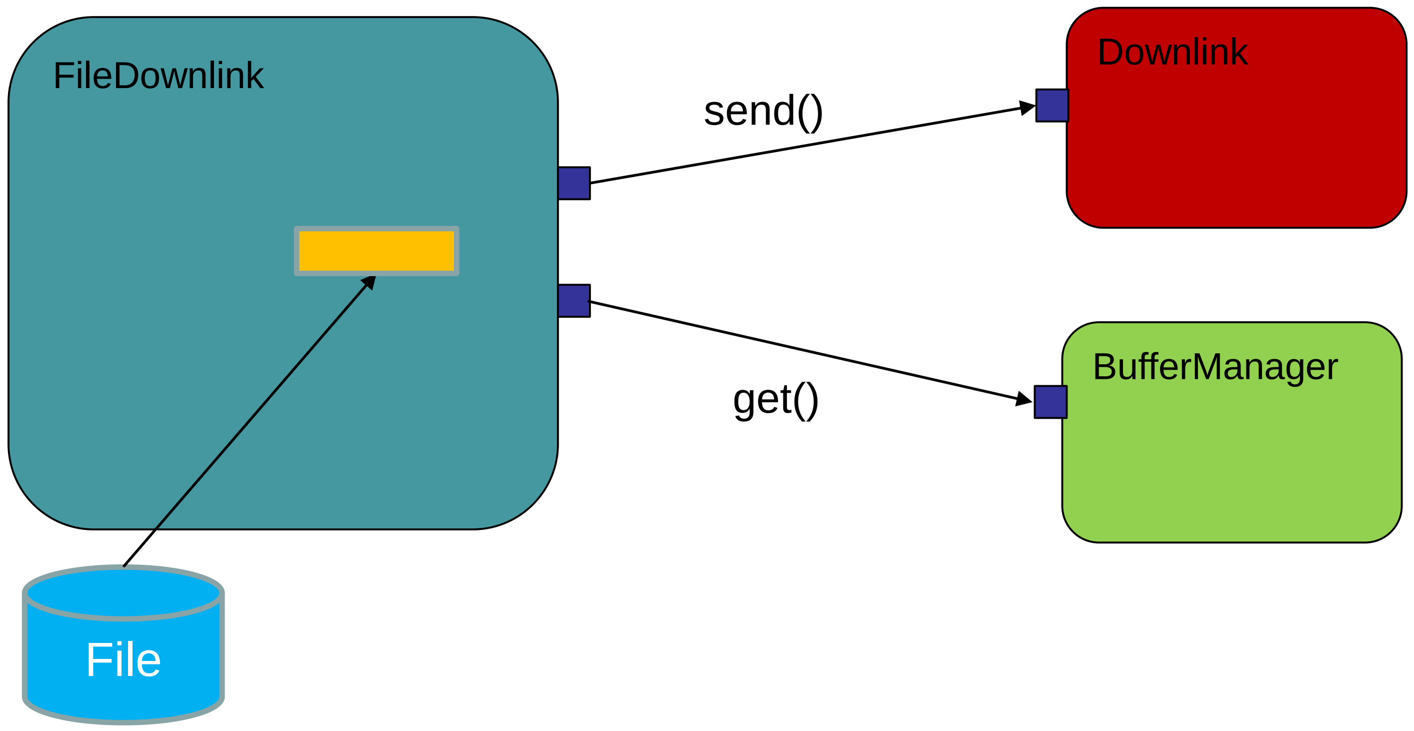
Buffer Management (Svc/BufferManager)

- Manages a pool of buffers for components
- Allocated at startup
- Variable sized buffers
- FIFO based storage
- Component gets() a buffer, uses it, and then sends() it when complete
- Buffer lifetimes managed as a system issue



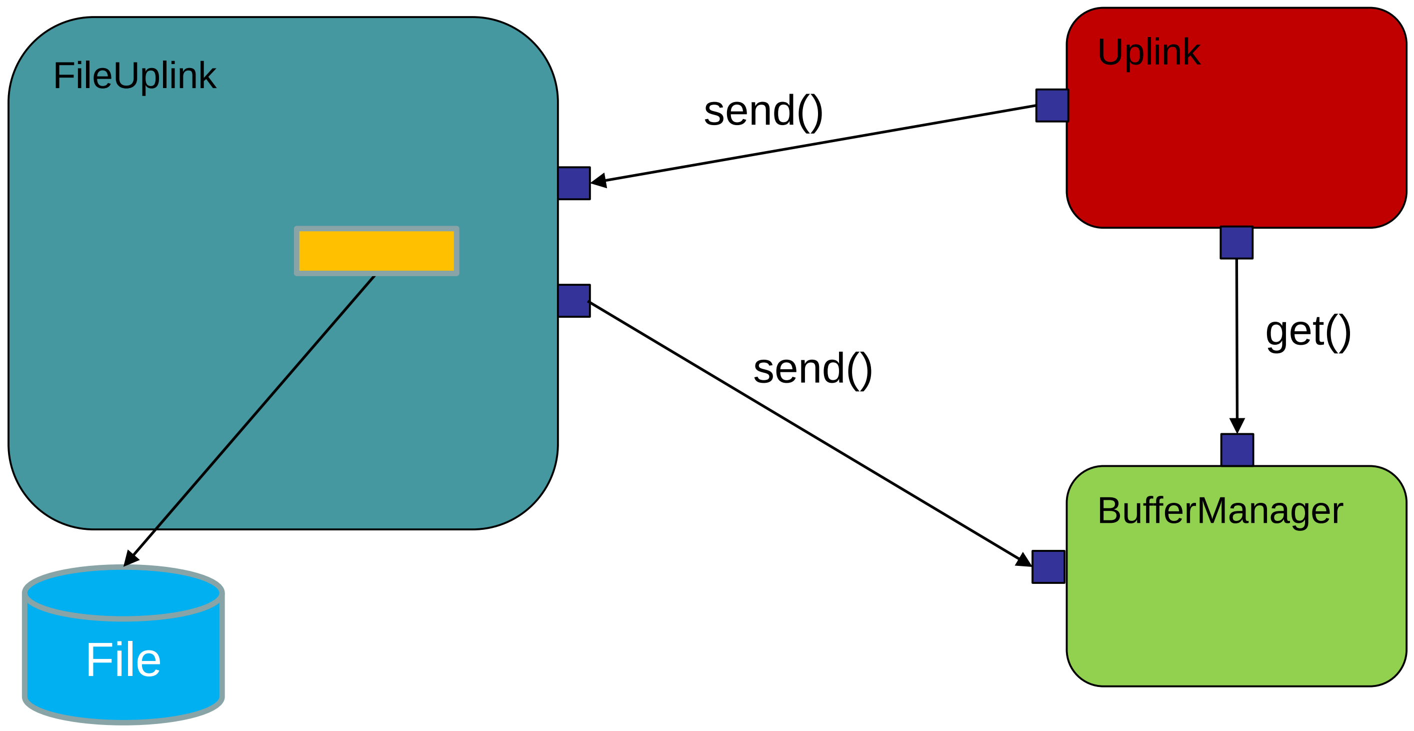
File Downlinking (Svc/FileDownlink)

- Sends the contents of a file to the ground system when commanded
- Gets buffers from BufferManager



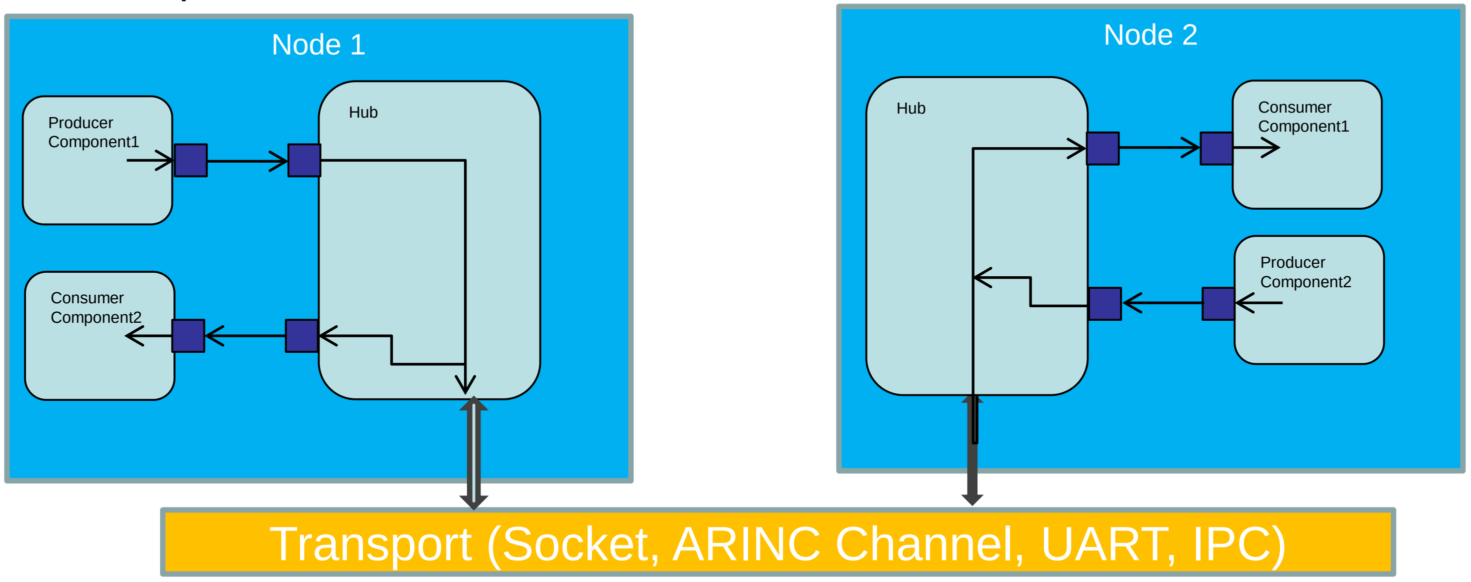
File Uplinking (Svc/FileUplink)

- Gets the contents of a file from the ground system
- Uplink sends buffer with file contents
- Returns buffers to BufferManager



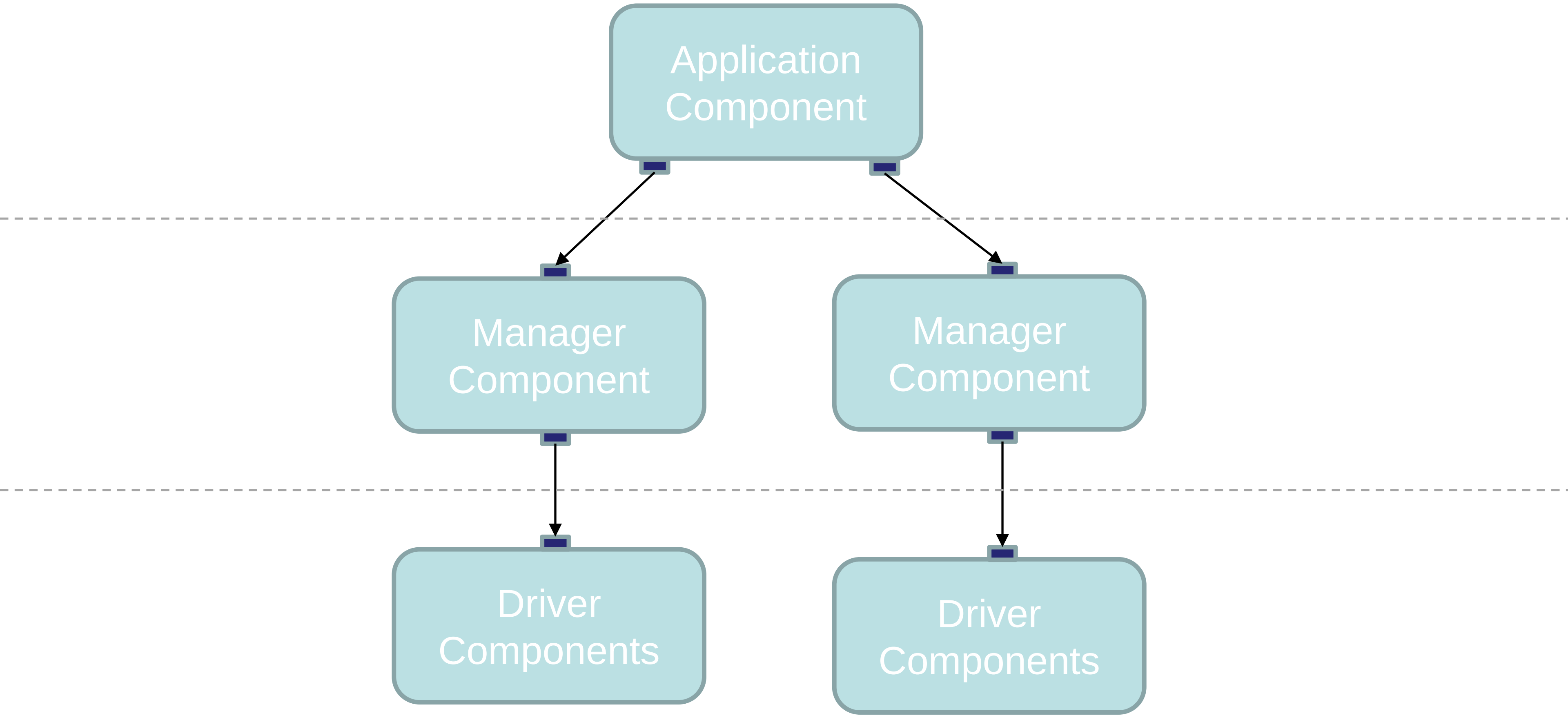
Multi-node

- 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.



Typical F´ Application Architecture

CALIFORNIA INSTITUTE OF TECHNOLOGY



Code Scaling

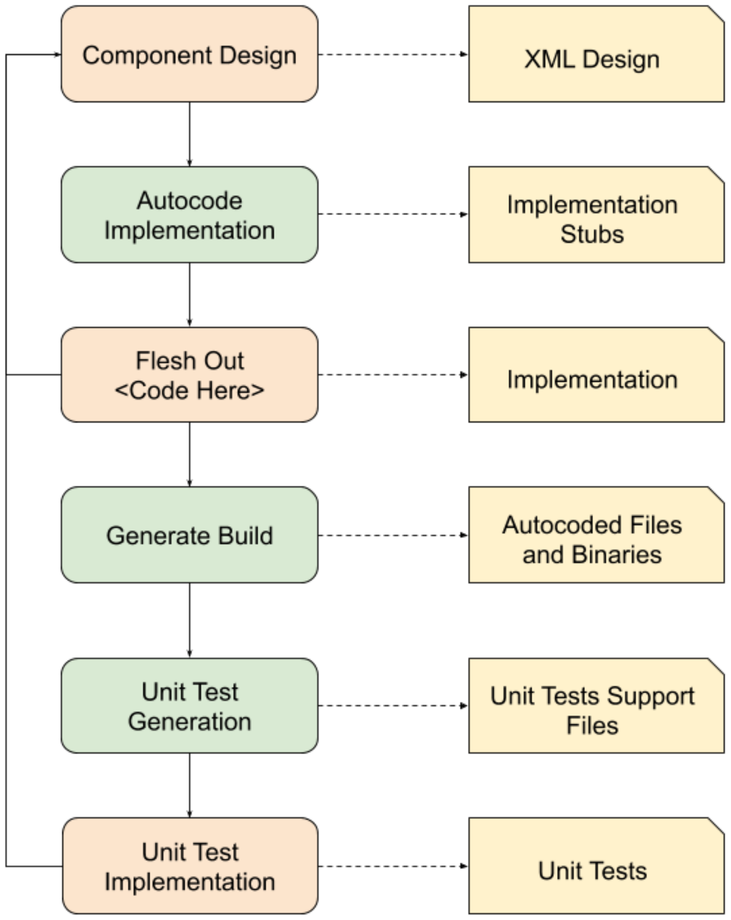
- Framework code is very compact
- Generated code is also compact
  - Demo application for TI microcontroller was about 15K
- Native type sizes can be configured
  - e.g. some microcontrollers have 16-bit/8-bit only support
- Features can be added or removed depending on resources
  - Object naming
  - Port execution tracing
  - Serialization of ports
    - Single node systems don't always need
    - This is not data serialization but the use of serialized ports
  - Object naming/registry
  - Component connection tracing
  - Text logging
- For bare-metal (i.e. no OS) processors, there is an OS emulation layer that allows reused of components with message queues and threads

Status

- Deployed on:
  - RapidScat (ISS radar experiment)
  - Asteria
  - Mars Helicopter
  - Lunar Flashlight
  - NeaScout
  - DOD Testbed
- Future Projects (In proposal process)
  - MSH (Mars Science Helicopter)
  - Cold Arm
  - OWLs
- Various university projects
- Has been ported to:
  - Linux, MacOS, Windows (Cygwin), VxWorks, ARINC 653, RTEMS, Bare Metal (No OS)
  - Snapdragon, PPC (RAD750), Leon3, x86, ARM (e.g. Raspberry Pi), MSP430
- Mature set of C&DH components
  - Following flight processes such as code inspections, static analysis, and full-coverage unit testing
- Avionics Platforms:
  - Sphinx
  - Sabertooth

Advantages of the F´ Architecture

- Autocoding:
  - Component stubs and UTs
  - Communication mechanics
- Rapid Development:
  - Rapidly swap components
  - Multiple topologies and organizations
- Standard Tooling:
  - Application design tools
  - System testing tools
- Standard patterns and best practices at application level



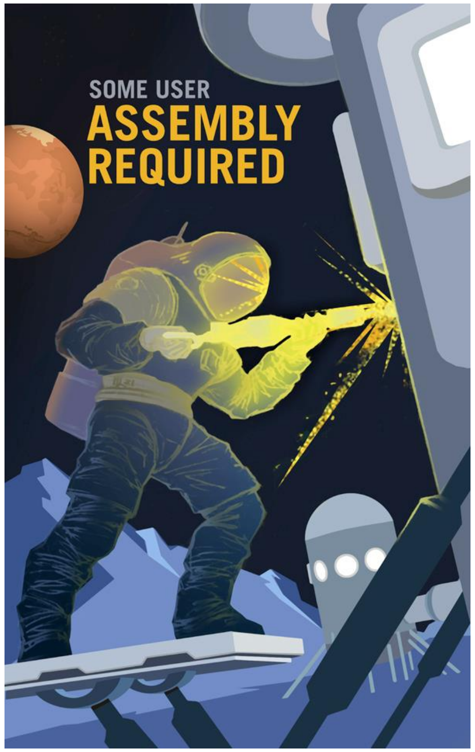
Open Source and F´

F´ Software Repository:
https://github.com/nasa/fprime

F´ Community Repositories:
https://github.com/fprime-community

F´ Internal Repository:
https://github.jpl.nasa.gov/FPRIME/fprime-sw

F´ Small Sat Introduction:
https://digitalcommons.usu.edu/smallsat/2018/all2018/328/



Credit: NASA/KSC