CubeSat Flight Software Workshop

F Prime Architecture

Timothy Canham
Flight Software Engineer
June 3, 2019

Jet Propulsion Laboratory
California Institute of Technology

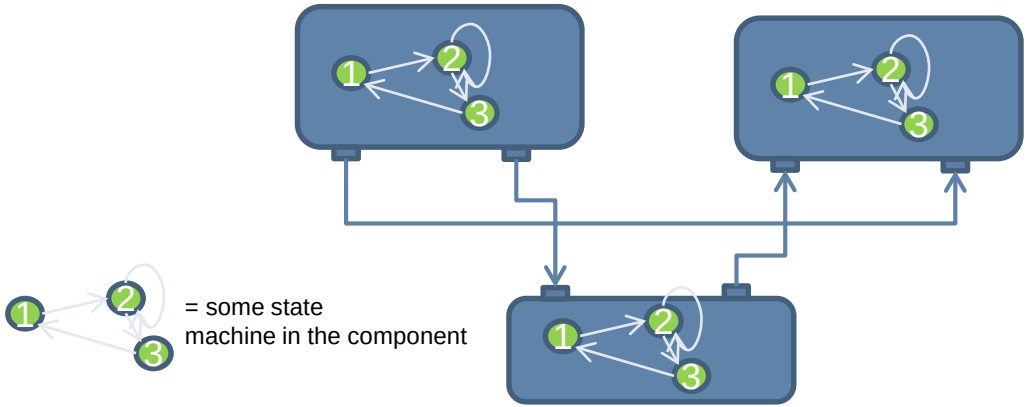
Background

- F` was developed as part of a technology task at JPL
  - Explore new flight hardware
  - Explore new software approaches
  - Targeted at smaller projects like instruments, Cubesats, and Smallsats
  - Sparser processor resources (e.g. 2MB memory, 128K program space)
    - TI MSP430, ARM-M*, LEON3
  - 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.

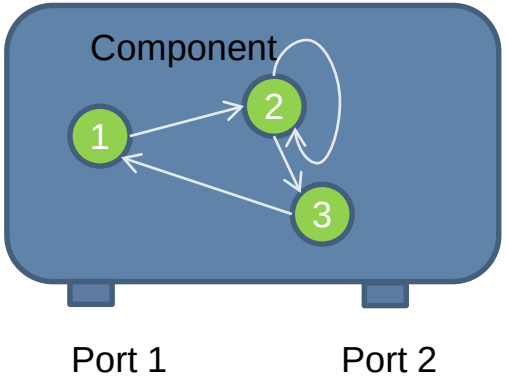
F`: A Component Architecture

- 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 logic
  - Developer specifies in XML
- Developer writes implementation classes to implement interfaces.



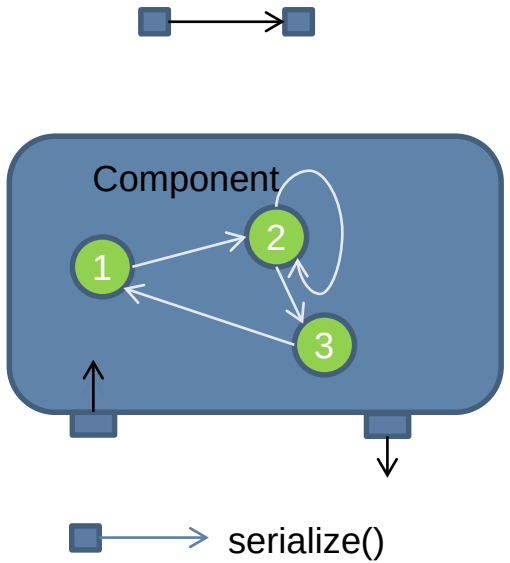
Characteristics of Components

- 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
- Context for threads
- Executes commands and produces telemetry

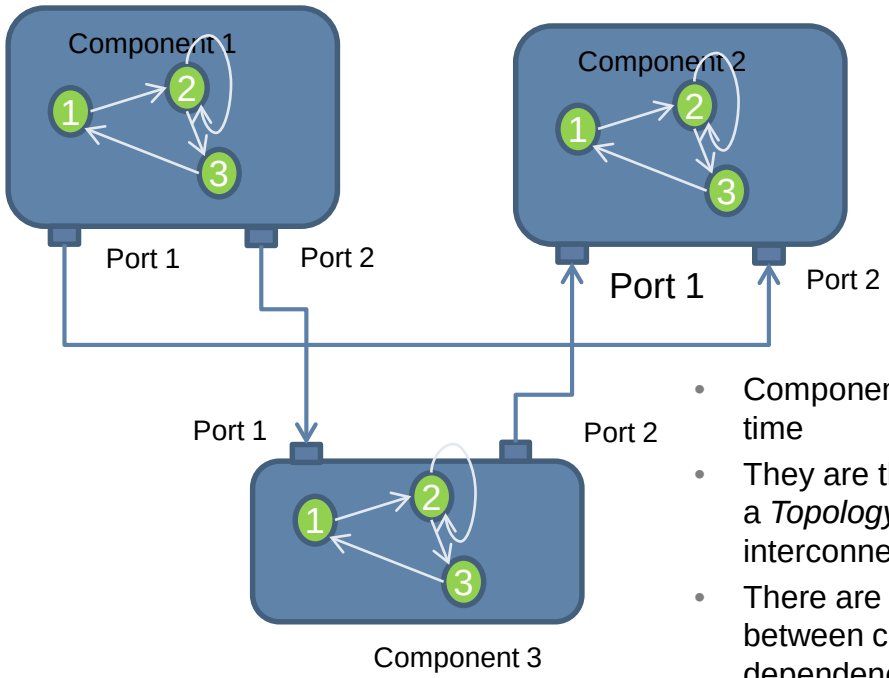


Characteristics of Ports

- Encapsulates typed interfaces in the architecture.
  - Think 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 method on attached port
  - A component
    - Incoming port calls call component provided 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; user can write custom types. (see later slides).
- Ports can have return values, but that limits use
  - Only return data when component has synchronous interface
  - No serializable connections since serialization passes a data buffer but does not return one (see later slides for explanation)
- Pointers/references allowed for performance reasons
- Multiple output ports can be connected to a single input port



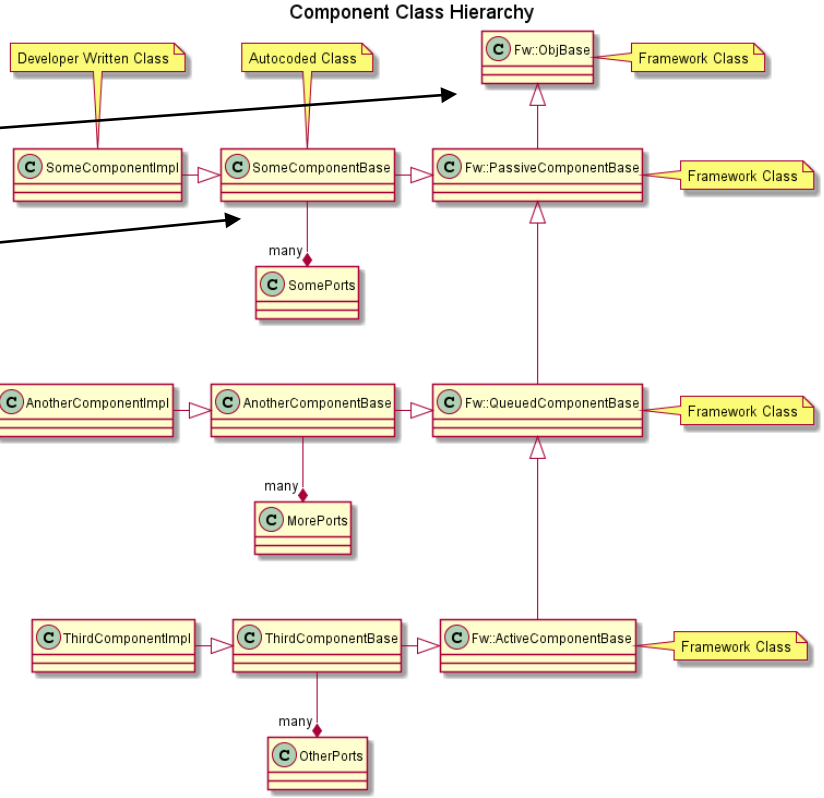
A Component Topology



- 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
  - E.g. simulation versions

Component Type Hierarchy

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

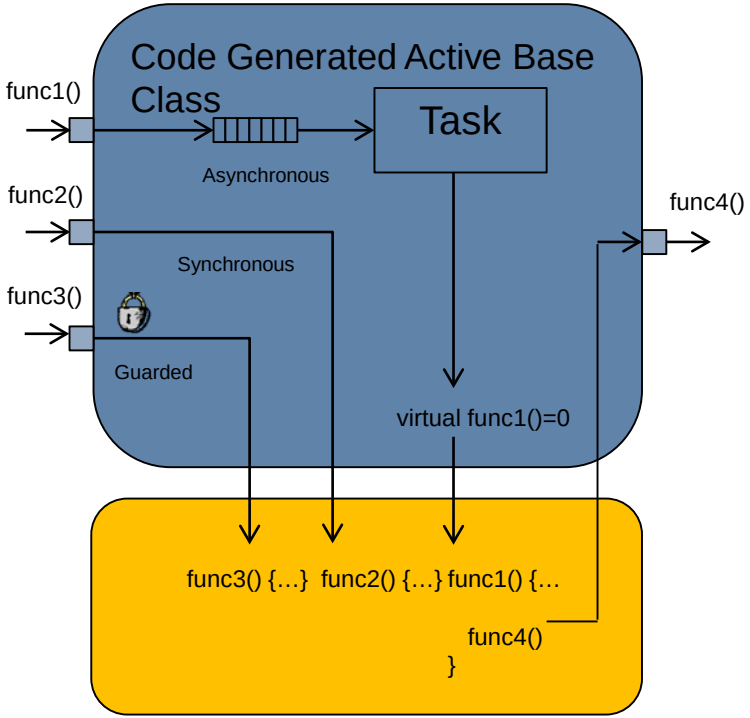


Component Types

- User specifies type of component in XML. Types are:
- Passive Component
  - No thread
  - Port interface calls are made directly to user derived class methods
- Queued Component
  - No thread
  - A queue is instantiated, and asynchronous port calls are serialized and placed on queue.
  - Implementation class makes call to base class to dispatch calls to implementation class methods for asynchronous ports
    - Can be made from any implementation class function
    - Thread of execution provided by caller to a synchronous port
- 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 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. 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 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 and thread to make sense.
- Designer needs to be aware of how all the different call kinds interact (e.g. reentrancy)
- Output ports are invoked by calling generated base class functions from the implementation class.



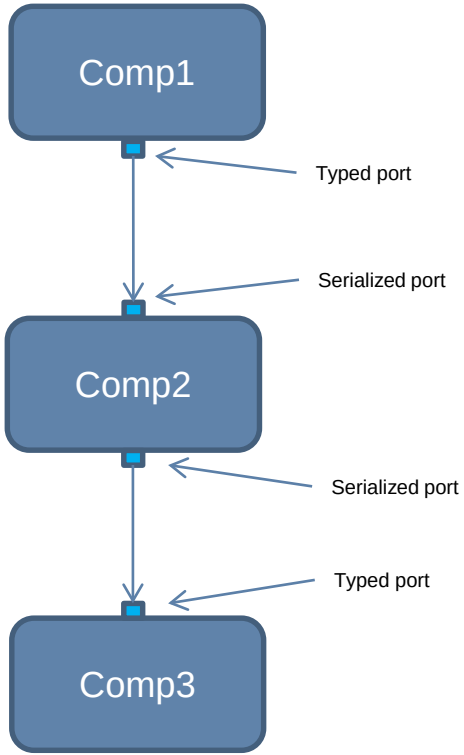
Developer Written Implementation Class

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
- Port calls and commands and their arguments are serialized and placed on message queues in components
- Command arguments and telemetry values are passed and encoded/decoded into this form
- Components that store and pass data can use this form and not require knowledge of underlying types
- User can define arbitrary interface argument types and framework automatically serializes
- User can define complex types in XML and code generator will generate classes that are serializable for use internally and to and from ground software

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.
- Component XML specifies arguments and types.
- Data for service is passed in serialized form.
- The code generator parses arguments and types and generates code to convert arguments to serialized data or vice versa.
- Calls implementation functions with deserialized arguments (commands) or provides base class functions to implement calls (telemetry, events and parameters)
- Code generator uses port types that are specified in XML themselves. These ports can be then used in components that provide interfaces for transporting or storing data for those services.

Commands

- Component command XML specifies:
  - Opcode, mnemonic, and arguments
    - Arguments can be any built-in type or external XML complex type
    - Complex type can be a single argument
    - Can define enumerations
  - Synchronization attribute
    - Sync, async, or guarded
    - Same meaning as ports
      - Async can specify message priority
- Implementation class implements function for each command
  - Framework code deserializes arguments from argument data buffer
- 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
    - Complex type can be a single argument
    - 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 function to call for each event with typed arguments
  - Provides stronger type checking at compile time than MER/MSL EVR macros
  - Called by implementation class
- 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

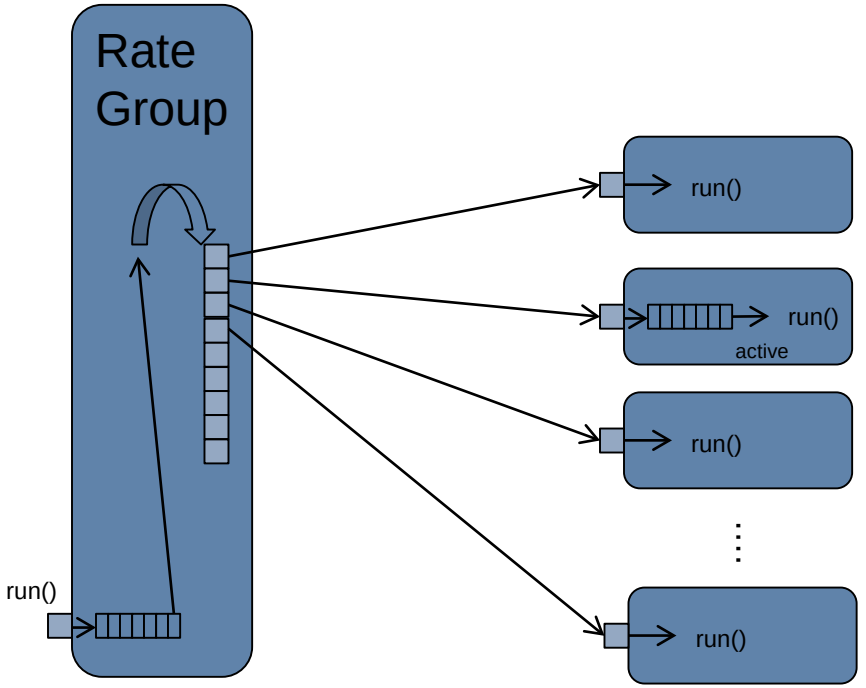
- Parameters are traditional means of storing non-volatile state
  - Framework provides code generation to manage, but user must write specific storage component
- Component XML specifies parameters that have:
  - ID, name, and data type
    - Data type can be any built-in type or external XML complex type
    - Can define enumerations
  - Optional default value
    - In the event the parameter cannot be retrieved, assigns default value to parameter
- Code generator automatically adds port for retrieving parameters
- During initialization, a public method in the class is called that retrieves the parameters and stores copies locally
  - Can be called again if parameter is updated
- Code generated base class provides function to call for each parameter to retrieve stored copy
  - Implementation class can call whenever parameter value is needed

Architectural patterns

- Over time, tested C&DH components can be developed that implement typical non-mission-specific flight functions that are specified in the XML
  - Commands, Events, Telemetry, Parameters
- Design them so small-scale projects can live with sufficient implementations out of the box
- Includes:
  - Command dispatcher
  - Command sequencer
  - Event log (binary and console)
  - Telemetry database
  - File-based parameter storage and updating
  - Active rate groups
- Define interfaces for facilities that would support the existing component implementations
  - Uplink/Downlink packet types
    - Input/Output of ground system and C&DH components
  - Uplink/Downlink ports
    - Project would have specific uplink downlink hardware

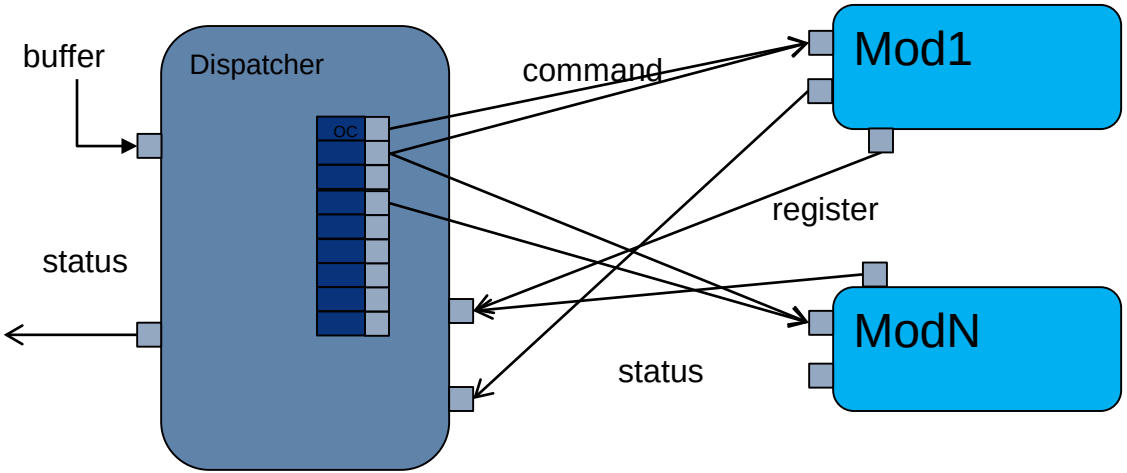
Example Component Pattern - Rate Group

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



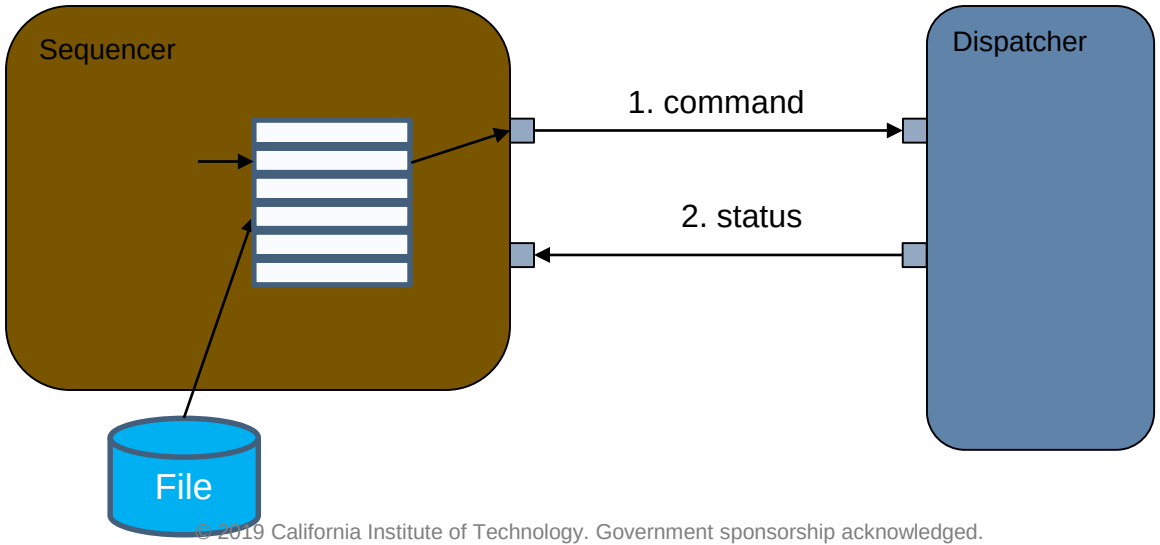
Example Component Pattern - Command Dispatcher

- Command dispatcher receives raw buffer containing command and arguments
- Command opcode is extracted, and lookup is made
  - Table maps opcode to port
  - Multiple opcode entries per component
- Argument buffer is passed to component
- Command dispatcher is a passive component



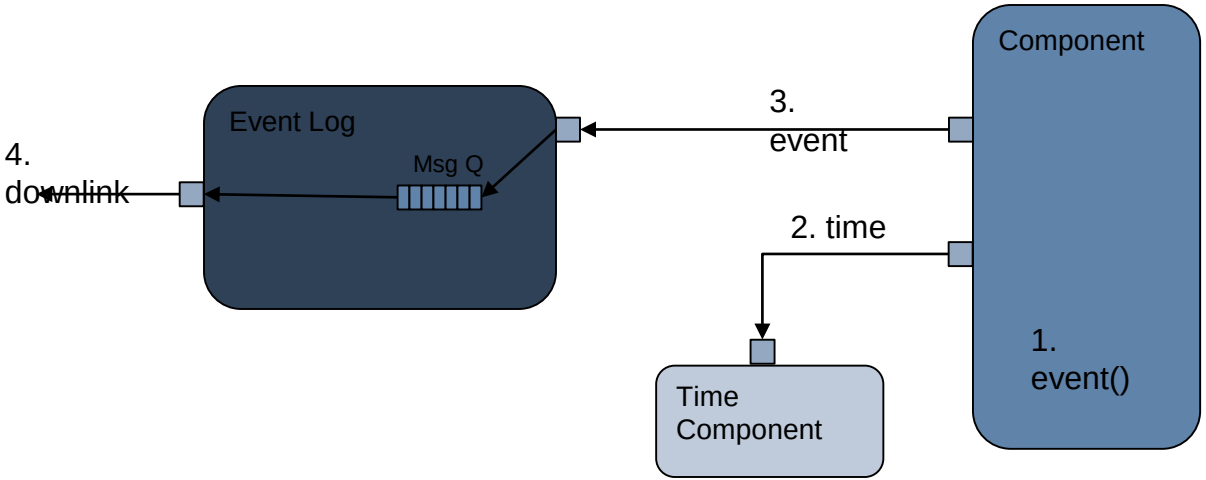
Example Component Pattern - Command Sequencer

- 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, passed response moves to the next command
- Active component



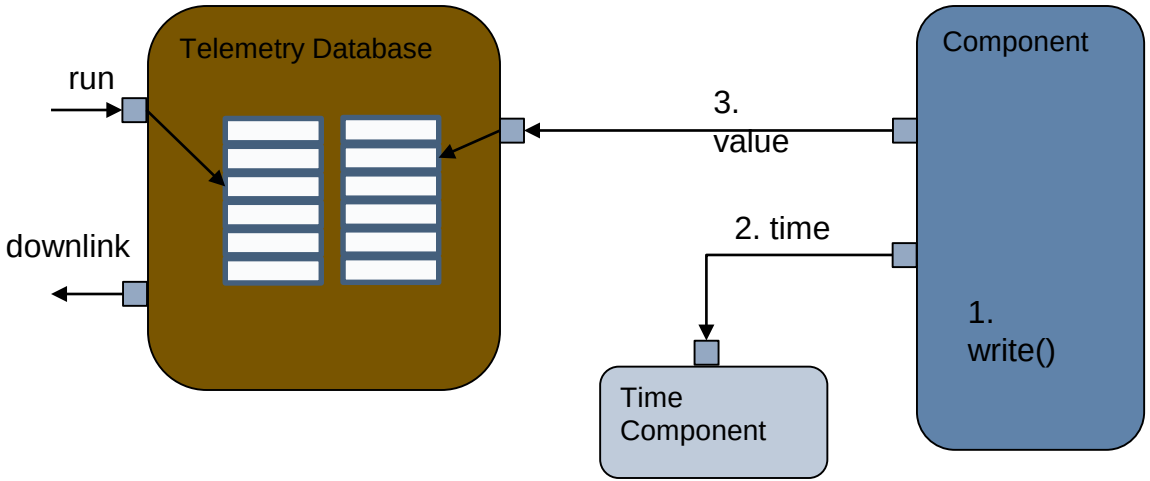
Example Component Pattern – Event Log

- Component implementation calls function to generate event
- Base class function (code-generated) retrieves time tag from time source component (project-specific).
- Component sends event to Event Log component
- Event log component places event on message queue. Thread of component then sends downlink packet with event
- Active component



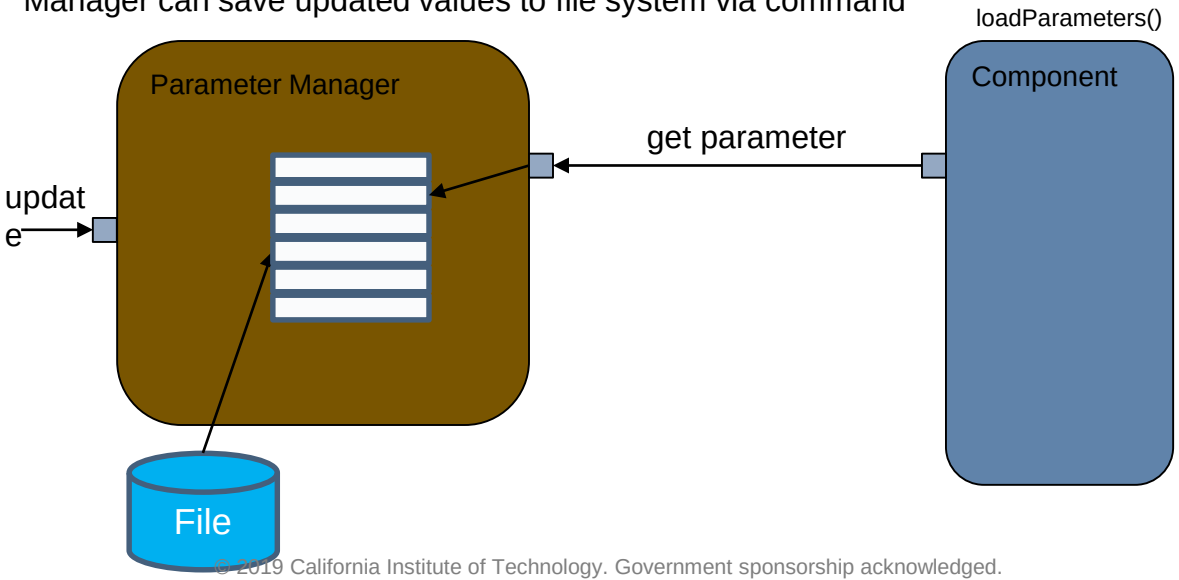
Component Pattern – Telemetry Database

- Telemetry database has double-buffered array of telemetry buffers
- Implementation class calls base class function with telemetry channel update
- Base class function retrieves time tag from time source component.
- Component writes updated value 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
- Passive component



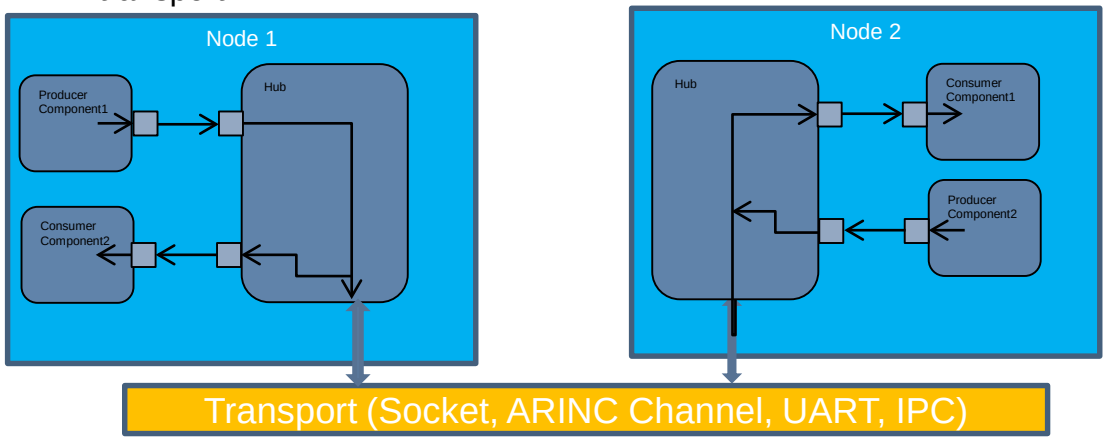
Component Pattern – Parameter Manager

- Parameter Manager loads file containing parameters from file system during initialization
- Initialization subsequently calls *loadParameters()* on all components with parameters. Can also be called after
- Uplinked parameter updates update the stored value of parameter
- Component can refresh parameters by implementing command to reload
- Manager can save updated values to file system via command



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.



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 really need
    - This is not data serialization but the use of serialized ports
  - Object naming/registry
  - Component connection tracing
  - Text logging
- For very compact processors with no OS, developers can choose non-active components

Status

- Framework released as open source
- Earlier JPL version flown on RapidScat, an ISS radar experiment
- Has been ported to:
  - Linux, MacOS, Windows (Cygwin), VxWorks, ARINC 653, RTEMS, Bare Metal (No OS)
  - PPC, Leon3, x86, ARM (A15/A7), MSP430
- Mature set of C&DH components
  - Following flight processes such as code inspections, static analysis, and full-coverage unit testing
- Version being developed as companion for JPL hardware project for Cubesat missions
  - Will include platform driver components and other peripherals
- Available on JPL GitHub:
  - https://github.jpl.nasa.gov/FPRIME/fprime-sw.git
- Hubs demonstrated on:
  - Sockets
  - ARINC 653 Channels
  - High-speed hardware bus between nodes
  - UARTs between nodes in an embedded system

Jet Propulsion Laboratory
California Institute of Technology

jpl.nasa.gov