Jet Propulsion Laboratory
California Institute of Technology

F Prime Cheatsheet

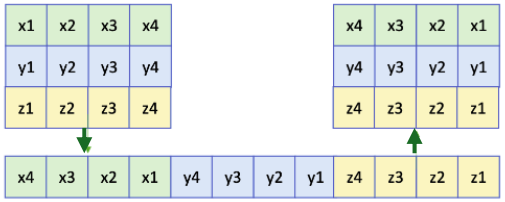
fprime.jpl.nasa.gov/cheatsheet

Terminology and Concepts (1)

FPP (1-1) – A Domain-Specific Language (DSL) that is used to model F Prime, and F Prime auto codes .hpp and .cpp files based on an initial .fpp model, so the user has less C++ boilerplate code to write. It also autogenerates a test harness.

Serialization (1-2) – taking a specific set of typed values or function arguments and writing them in a standard way (Big Endian) into a data buffer

- Queued components have queues where port calls, commands and their arguments are serialized



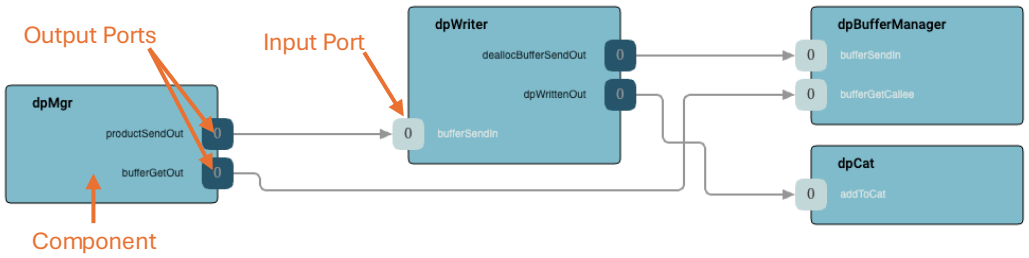
Example serialization into and out of a queue

Core Constructs (1-3)

1. **Components** – Basic building blocks of the flight software

Component Kind	Has Queue	Has Thread	Typical Work Type
Passive	No	No	Cyclic, Helper
Queued	Yes	No	Cyclic
Active	Yes	Yes	Event-Driven, Background

   - **Cyclic work** – addresses hard deadlines, work is performed on a repeating schedule driven by a Rate Group, e.g. send a control update every 10ms
   - **Event-Driven work** – timely work lacking hard deadlines, e.g. send command
   - **Background work** - work without timing requirements (soft deadlines)
2. **Ports** – Used to connect components together through the passing of data.
   Ports must be of the same data type. Ports may be of the following kinds:
   1. **Synchronous (sync)** - immediately invokes a function without a queue
   2. **Asynchronous (async)** - call is placed in a queue and dispatched on a thread that empties the queue
   3. **Guarded** - call directly invokes handler, but only after locking a mutex shared by all guarded ports in the component
3. **Topology** - The wiring model for a deployment lives in topology.fpp and does two jobs:
   1. Declares instances of components with unique base IDs
      (Note: one component class can have many instances, each with its own priority, queue depth, instance number, etc.)
   2. Connects ports between those instances



Example Topology Diagram (from Visualizer tool)

F Prime Terminal Commands (2)

```
1. fprime-bootstrap project
2. fprime-util new --component
3. fprime-util impl
4. fprime-util generate -f
5. fprime-util build
6. fprime-util new --deployment
7. fprime-util new --subtopology
8. fprime-util visualize
9. fprime-gds
```

Rate Groups (1-4)

Driven by a system clock source, divides ticks into slower rates (e.g., 1000Hz → 100Hz, 10Hz, 1Hz).

- **Passive** – executes synchronously in driver's thread, use for hard real-time work, detects slips if work overruns
- **Active** – executes asynchronously on separate thread, use when jitter is acceptable, won't block other rate groups

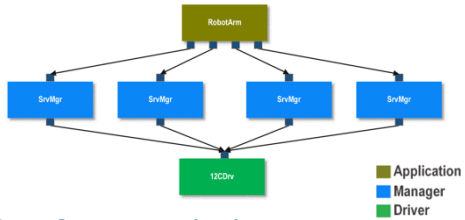
CMake (1-5) – F Prime's build system of choice

- Handles FPP autocoding and dependencies
- To add a module, create CMakeLists.txt calling `register_fprime_module()`
- Then add the directory via `add_fprime_subdirectory()` in the parent CMakeLists.txt.

Application-Manager-Driver (1-6)

Layered software architecture

1. Application – mission functionality
2. Manager(s) – manages a peripheral
3. Driver – hardware interface

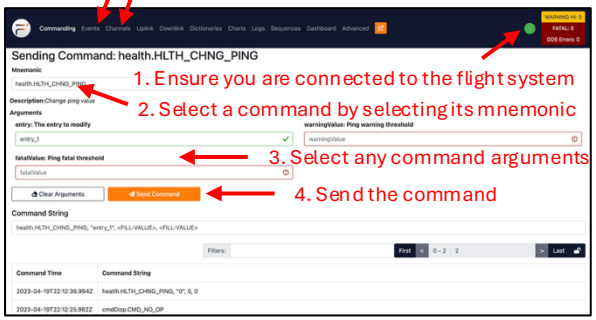


Data Constructs (1-7)

Real time communication primitives between components and the ground

1. **Commands** (opcode, mnemonic, arguments, synchronization)
2. **Events** (ID, name, severity)
3. **Channels** (ID, name, data type, update (optional))
4. **Parameters** (ID, name, data type, default value)

F Prime GDS UI - Sending a Command (3)



F Prime Core Components (4)

Svc (service) components are generic components of a flight software stack provided by the F Prime framework, so users do not have to reinvent them.
See table below for Svc component descriptions.

Component	Description
Command Dispatcher	Dispatches uplinked commands to components
Command Sequencer	Executes a set of commands uplinked as a file
Telemetry Processing	Gets sampled telemetry from components for downlink
Event Logging	Gets asynchronous event notifications from components for downlink
File Uplink/Downlink	Send and receive files
File Manager	File management including listing, moving and deleting files
Parameter Database	Database to store non-volatile parameter values for components
Value Database	Allows components to exchange run-time data values
Rate Group Execution	Allows a set of components to run sequentially at a certain rate
Uplink/Downlink Framing	Generic components for framing data for telecommunications
Buffer Management	Pre-allocates a set of buffers for dynamic memory requirements for components

New Project Workflow (5)

1. **Prerequisites:** `python3 -m venv fprime-venv && source fprime-venv/bin/activate && pip install fprime-bootstrap`
2. **Create new project** with `fprime-bootstrap project` in the terminal
   Note: Pick one namespace for the entire project upon creation.
3. **Create a new component** in the Components folder using `fprime-util new --component` (see (1-1) for component kind selection)
   1. Edit the <Component Name>.fpp file
   2. `fprime-util impl` to turn .fpp into C++ stubs
   3. Replace original .cpp and .hpp files with new templates and fill in implementation of functions
4. **Create a new deployment** in the project directory using `fprime-util new --deployment`
   1. Define component instances in instances.fpp and assign unique base IDs

```
# 3.1)  Example active component instance in instances.fpp:
instance mySensor: MyProject.Sensor base id 0x0300 \
  queue size 10 \
  stack size 64 * 1024 \
  priority 50
```

   2. Declare instances in topology.fpp (see (1-1-3))

```
# 3.2) Example component instance in topology.fpp:
Open in Diagram: MyDeployment
topology MyDeployment {
  instance mySensor
```

   3. Add component to deployment's CMakeLists.txt (see (1-9))
   4. Wire port connections in topology.fpp inside the topology block

```
# Connection syntax:
sourceInstance.outputPort -> targetInstance.inputPort

# Standard component connections:
  # Time:
  timeGetOut -> systemTime.timeGetPort
  # Events:
  eventOut -> eventLogger.LogRecv
  # Telemetry:
  tlmOut -> telemetryChan.TlmRecv # (if component has telemetry)
  # Commands:
  cmdOut -> cmdDispatcher.cmdIn # (if component has commands)
```

   5. **Generate and build:** `fprime-util generate && fprime-util build`
5. **Run `fprime-gds`** from the project directory (see section (5))
   1. By default, the GDS runs on the local machine

Clone Existing Project (5-7): fprime-bootstrap clone <repo url>

Iterative Development Workflow (6)

1. Edit component code (.fpp → `fprime-util impl` → .cpp, .hpp)
   1. In component directory: `fprime-util generate -f && fprime-util build`
2. Update topology (add/modify instances, connections)
   1. In deployment directory: `fprime-util generate -f && fprime-util build`
3. Write/update tests (in component directory)
   1. `fprime-util generate --ut && fprime-util build -ut`
   2. `fprime-util check` (run tests)
4. Integration testing
   1. In deployment directory: `fprime-util build`
   2. `fprime-gds` (test with GDS UI)

- **Quick iteration:** Steps 1 & 3 (component-level)
- **Full integration:** Steps 2 & 4 (deployment-level)

C++ Autocoded Functions (7)

Functions generated by FPP to fill in in the component .cpp file

Suffix/prefix	User Action	Direction
`void <port>_handler(portNum, args…)`	implement	Something is calling into the component
`void <port>_out(portNum, args…)`	call	The component is reaching out
`void <MNEMONIC>_cmdHandler (opCode, cmdSeq, args…)`	implement	Command came in
`Void cmdResponse_out(opCode, cmdSeq, Fw::CmdResponse::OK)`	call	Send command result back
`log_<SEVERITY>_<EVENT_NAME> (args…)`	call	Publish an event
`tlmWrite_<CHANNEL_NAME>(value)`	call	Publish a telemetry value
`<Type>paramGet_<PARAM_NAME> (Fw::ParamValid& valid)`	call	Read a parameter
`void parameterUpdated (FwPrmIdType id)`	override	React to a param change
`bool isConnected_<port> _OutputPort (portNum)`	call	Guard before _out if the port is optional

Svc::BufferManager (8)

Safe dynamic memory allocation for embedded systems through a port call to a component designed to manage system memory

1. Call allocation port receiving an `Fw::Buffer`
2. Check validity of the buffer with `buffer.isValid()` and handle potential allocation failure
3. Use the allocated `Fw::Buffer`
4. Call deallocation port returning the `Fw::Buffer`

Primitive F' Type	Description
BOOL	C++ boolean
Ix	signed **x**-bit integer
Ux	unsigned **x**-bit integer
F32	32-bit floating point
F64	64-bit floating point

Configurable Integer Type	Description
FwIndexType	Port and small array indices
FwSizeType	Sizes
FwSignedSizeType	Signed sizes
FwAssertArgType	Arguments to asserts

Fw:: Type	Description
Fw::Time	Timestamp for events/telemetry
Fw::Buffer	Memory buffer for data transfer between components
Fw::CmdResponse	Command handler return status
Fw::Success	Binary result for operations (OK, FAILURE)
Fw::LogSeverity	Event severity
Fw::Enabled	Enable/disable state

Complex Types (1-8)

Example in topology.fpp:

```
# 1. Enum - typed enumeration
  enum SensorMode {
    IDLE = 0
    CALIBRATING = 1
    ACTIVE = 2 }

# 2. Array - fixed-length container
  array TelemetryBuffer = [10] F32

# 3. Struct - composite type
  struct SensorReading {
    timestamp: Fw.Time
    mode: SensorMode
    values: TelemetryBuffer
    $id: U32 }
```