F` (F Prime)

User’s Guide

Prepared By

Timothy Canham

May 24, 2017

Jet Propulsion Laboratory

Pasadena, CA

Copyright (C) 2009-2017 California Institute of Technology. ALL RIGHTS RESERVED.

Any commercial use must be negotiated with the Office of Technology Transfer at the California Institute of Technology.

Change Log

May 24, 2017	Release 1.0

Contents

1 Introduction

The F' (F Prime) Software Framework was developed at JPL with the goal of providing software for flight systems that would be memory efficient and reusable. It consists of a component framework utilizing code generation to generate the commonly used patterns in the architecture. The framework is meant to be delivered as a package; the source, build system, code generators, GUI and an example are ready to be built and run on systems commonly used at JPL. This user's guide will show how to acquire, build and run an example as well as directions for applying the software to a particular use. For more details on the architecture itself, read the F` Architecture Description document located in the *docs* folder of the distribution. It will explain the terms used to describe the example.

2 Acquiring the Software

Major releases of the software are archived on the NASA public GitHub. In order to clone the F` Git repository, a user account on GitHub is required. To create an account, go the webpage:

https://github.jpl.nasa.gov

Once the account has been created, the F` distribution can be cloned with:

```
git clone https://github.jpl.nasa.gov/ISF-Development/isf.git
```

3 System Requirements

The distribution requires that a certain set of tools be installed on the host where it is being built. This section will list the tools as well as requirements for each host.

3.1 Tools

3.1.1 Python

The code generators for F` require Python > 2.4x to run as well as additional packages. To make installing these packages easier, the python package "pip" can be used with the pip requirements file `Gse/bin/requirement.txt` (see pip documentation).

3.1.2 Eclipse CDT

Although Eclipse is not required to build and run the software, Eclipse CDT project files have been created and provided with the source. Eclipse provides a convenient environment for editing and building the software. The Eclipse CDT environment can be downloaded from http://www.eclipse.org.

3.2 Hosts

3.2.1 Linux

In addition to the tools listed above, the Linux build requires the following tools (these should already be there for a typical installation):

- GCC/G++ >= 4.3.2 (/usr/bin/gcc)
- gmake
- libxml and libxslt

For Ubuntu, additional necessary modules can be added by executing `mk/os-pkg/ubuntu-packages.sh`.

3.2.2 Windows 10

For Windows development, an installation of Cygwin is required to execute the example. There are two possible build environments for the Cygwin build: Cygwin itself and Eclipse.

3.2.2.1 Cygwin Build Environment

To download the Cygwin environment, go to the following web site:

http://www.cygwin.com

When installing Cygwin, the following packages should be installed. The package versions were the ones available at the time that the guide was written, but newer versions should work. The 32-bit version of Cygwin was found to have fewer issues running the environment.

- gcc-core
- gcc-g++
- gccmakedep
- libstdc++
- make
- python
- python-lxml
- python-setuptools
- git
- python-tkinter
- X11

A script that can be run to add the necessary packages can be found in `mk/os-pkg/cygwin-pkgs.sh`.

3.2.3 Mac OS

The demo has been tested on Mac OS. In order to compile the code, the XCode development environment must be installed. The LLVM compiler will be installed with Xcode as well as git. Packages similar to the ones above must be installed.

4 Building and Running the Demo

A demonstration application has been provided with the software. Directions for executing the software are in section 4.2.

4.1 Overview

The demonstration application consists of the following components:

- A rate group driver passive component, which cycles at 1 Hz. This component takes the 1 Hz base clock and distributes messages to the rate group components to run at their prescribed rate.
- Three rate group active component instances running at 1Hz, 0.5Hz, and 0.25Hz that are cycled by the rate group driver.
- A driver active component that accept buffers. Normally a driver would interact with hardware, but for this demo it simply loops data coming to the input port back to an output port. The driver is connected to the 1Hz rate group to show how a driver can be periodically run to check for hardware events. The driver also has an "interrupt" port connected to the rate group driver to drive it at 1Hz. The "1Hz" interrupt in the demo is achieved by executing a loop with 1 second delays.
- A "producer" queued component (SendBuff) that is connected to the 0.5Hz rate group and produces data that gets sent to the driver component.
- A "consumer" active component (RecvBuff) that accepts buffers from the driver component. Since the driver component loops back buffers, every time the producer component sends a buffer, the consumer component will receive it.
- A passive uplink/downlink component that receives telemetry from other components for downlinking and receives uplinked commands. It uses TCP/IP sockets to communicate with ground software. For this demo, the ground software is not present.
- An active telemetry component that accepts channelized telemetry from the other components. It is connected to the uplink/downlink interface component, and sends data based on the 4Hz rate group.
- An active logger component that receives logging output from components and sends it to the uplink/downlink component.
- A passive text logger component that receives the text version of the logging events and prints it to standard output.
- An active command dispatcher component that receives uplinked commands from the uplink/downlink component, decodes the command identifier (opcode), and forwards it to the component that executes the command.
- An active parameter database component that stores parameter values for the consumer and producer components.
- A time source passive component that reads Linux time and returns it to the caller. The time is used as time tags for events and telemetry sent by the components.
- A command sequencer to execute a set of commands.
- A health component to verify the active components are still running.
- Some signal generator components to generate test data.

Details for each component can be seen in the component SDD (Software Design Documents) in the `docs` subdirectory for each component as well as the Ref application (TODO).

4.2 Building the Demo

The demo can be run on one of the supported hosts described in Section 3.

1) Change to the "`Ref`" (AKA "Reference Application) directory in the location where the Git repository was cloned.
2) Type "`make gen_make`" from the prompt. Output similar to the following should appear:

```
> make gen_make
Generating Makefiles in /somepath/fprime-sw/mk/makefiles
Makefile generation complete.
```

   The source directories contain files names "mod.mk" which are scanned by the "gen_make" script to generate a make file to perform the build. See section XXXX for details on how the make system operates.

3) Type "`make`" from the prompt. The code should build to completion.

4) To install the command/telemetry GUI dictionary files, type:

   ```
   make dict_install
   ```

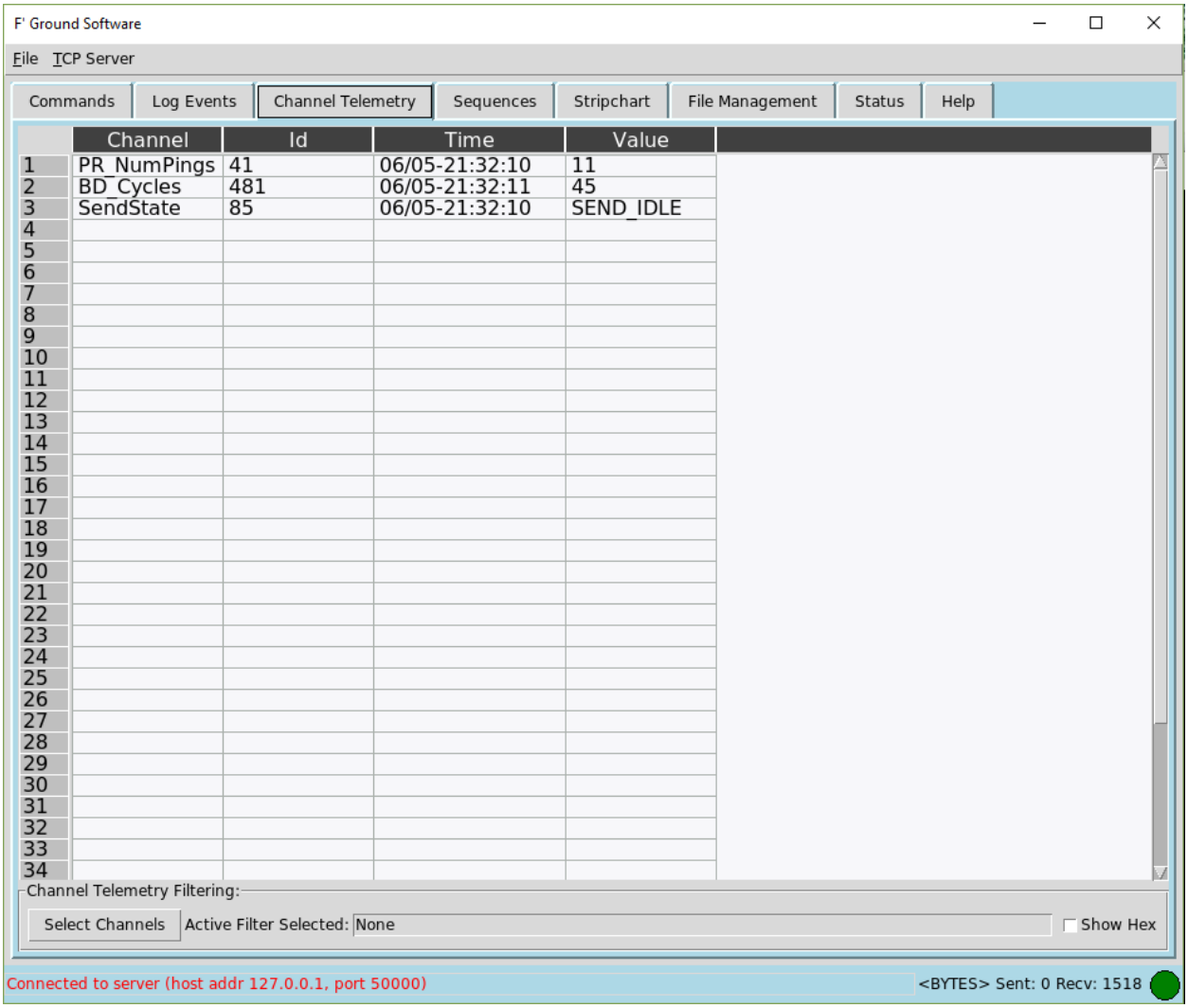
5) Typing "`make help`" will list other make targets that are available.

4.3 Running the Demo

The demo requires a system running an X server to display the GUI. To run the demo, execute the following script:

```
Ref/script/run_ref.sh
```

This will bring up the command and telemetry GUI seen below.



Another window will show the standard output of the reference application.

The user can enter commands and view telemetry by selecting the various tabs. Once the GUI is exited, the script will shut down the window containing the application.

5 A Tour of the Source Tree

The following directories constitute the demonstration code. The directories are a mix of tools, framework code and demonstration code. Details on each module can be seen in the docs/sdd.md (or html) subdirectory. When the code generator is run in a particular directory, it will generate files with the suffix "Ac.hpp and Ac.cpp". These files are not described since they are considered build products. They are automatically incorporated into the build by the build system. The following files are produced by the code generator:

Source	Generates	Description
<Name>SerializableAi.xml	<Name>SerializableAc.hpp(.cpp)	Autocoded serializable files
<Name>PortAi.xml	<Name>PortAc.hpp(.cpp)	Autocoded port files
<Name>ComponentAi.xml	<Name>ComponentAc.hpp(.cpp)	Autocoded component files
<Name>TopologyAi.xml	<Name>TopologyAc.hpp(.cpp)	Autocoded topology files

5.1 Autocoders

The Autocoders directory contains the scripts that are used to generate source for the components. It is implemented as a set of Python 2.x scripts that process the XML files used to describe the various entities in the system. The directory also contains the C language version of the HSM (Hierarchical State Machine) framework that has been flown on several missions at JPL. The HSM is not needed by the framework or the code generation, but is available for use by developers writing component adaptations if they wish to use HSM for implementing state machines. Developers do not need to study the implementation of the Autocoder in order to utilize it. The following directories are of interest to developers:

5.1.1 Templates

This directory has examples of the XML files that the developer would write for each entity in the system. Each file will be covered in detail in subsequent sections. Their function in the architecture is described in the architecture document. The files are as follows:

- Example2SerializableAi.xml – An example of a Serializable type
- ExampleSerializableAi.xml – An example of a more complicated serializable
- ExamplePortAi.xml – An example of a Port type
- AnotherPortAi.xml – A second port definition
- ExampleComponentAi.xml – An example of a Component type
- ExampleType.hpp(.cpp) – Not XML, but an example of a user written serializable
- ExampleComponentImpl.hpp(.cpp) – An example of a user written component class

The user can copy these files and use them as a basis for their own XML files.

5.2 Docs

The Docs directory contains documentation related to the design and usage of the F` framework.

5.3 Fw

The *Fw* directory is the location of framework code and base classes. This code should not be modified (with one exception, see *Cfg* in section 5.3.1) by developers using the framework. The code generation relies on the types declared to construct the entities in the architecture.

5.3.1 Cfg

The *Cfg* directory contains a header file (*Config.hpp*) that is used to configure various properties of the architecture. The developer can modify the file to tune the architecture for the requirements of a particular deployment environment. The contents are described in section 9.

The file *AcConstants.ini* contains a set of values for variables used in the code generator. This file follows the Python ConfigParser syntax which is based on Windows *.ini* files. Using this file allows component features like opcodes and port numbers to be changed without modifying the component XML itself. See the component XML specification in section 6.6.3.

5.3.2 Types

The Types directory contains basic types and other base classes used in the architecture.

They are as follows:

File	Usage
BasicTypes.hpp	Defines portable built-in data types and common macros.
Assert.hpp(.cpp)	Defines macros to declare an assertion in C++ code.
CAssert.hpp	Defines macros to declare an assertion n C code.
StringType.hpp(.cpp)	Declares a string base class
Serializable.hpp(.cpp)	Declares the Serializable base classes and helper functions
PolyType.hpp(.cpp)	Describes a serializable polymorphic type class that can be used to uniformly store different types.
EightyCharString.hpp(.cpp)	An eighty character string available for general usage if the developer does not wish to write one.
InternalInterfaceString.hpp(.cpp)	A string class used by internal interfaces when a string argument is specified.

5.3.3 Obj

The Obj directory contains class declarations and implementations for the root base class in the architectural framework. FwObjBase.hpp contains the declaration for the object base class. In addition, it contains a declaration for an object registry class. An object registry is an optional feature that allows all objects to be registered as they are created. It is a way to keep track of which objects have been created as well as perform some common actions such as printing a string representation of each object. Components and ports use the object class as a base class. The files and their descriptions are as follows:

File	Description
ObjBase.hpp(.cpp)	Declaration for object base class.
SimpleObjRegistry.hpp(.cpp)	An implementation of a simple object registry. This registry simply stores created object pointers in an array and calls their toString() method when asked.

5.3.4 Port

The Port directory contains the base classes for ports. The files and their descriptions are as follows:

File	Description
PortBase.hpp(.cpp)	Port base class. Contains methods and attributes common to all ports.
InputPort.hpp(.cpp)	Input port base class. Derived from Port base class. Contains methods and attributes common to all input ports.
OutputPort.hpp(.cpp)	Output port base class. Derived from Port base class. Contains methods and attributes common to all output ports.
InputSerializePort.hpp(.cpp)	Input serialize port class. Typed output ports can be connected to input serialize ports. The typed output port will serialize its arguments prior to invoking the port.
OutputSerializePort.hpp(.cpp)	Output serialize port class. Output serialize ports can be connected to typed input port. The serialize port passes the typed port a buffer representing the serialized arguments, which the typed port deserializes.

5.3.5 Comp

This directory contains the class declarations for the various kinds of components. These classes act as base classes for components created by the code generation and are not directly used by developers. The files are as follows:

File	Description
PassiveComponentBase.hpp(.cpp)	The base class for unthreaded passive components. These components have no thread of execution associated with them. This class derives from the object base class.
QueuedComponentBase.hpp(.cpp)	The base class for queued components, which have a message queue but no thread. It is derived from the passive component base class.
ActiveComponentBase.hpp(.cpp)	The base class for active components. Active components have a thread of execution as well as a message queue. It is derived from the queued component class.

5.3.6 Cmd

This directory contains XML and class declarations used to generate code for command interfaces to components. The xml generates port classes in the normal way via the code generator. The code generator then uses those generated classes as special command input ports for components that define commands in their component XML. Since the ports themselves are generated in the same way as any other port, they can be used by developers in other components that process commands such command dispatcher. The files in this directory are as follows:

File	Description
CmdPortAi.xml	XML description of a command port.
CmdResponsePortAi.xml	XML description of a command response port
CmdRegPortAi.xml	XML description of a command registration port
CmdArgBuffer.hpp(.cpp)	A class used by the command port that is a data buffer containing the serialized form of the command arguments.
CmdString.hpp(.cpp)	A string class used by the command code generator for string arguments.
CmdPacket.hpp(.cpp)	A class representing an encoded command packet that contains a command opcode and arguments. The code generator does not depend on this class, so it can be modified or not used.

5.3.7 Tlm

This directory contains XML and class declarations used to generate code for channelized telemetry interfaces for components. Channelized telemetry has historical been a snapshot in time of a set of data. Every value of that data is not necessarily stored permanently, but is sampled. The xml generates port classes in the normal way via the code generator. The code generator then uses those generated classes as special telemetry output ports for components needing telemetry. Since the ports themselves are generated

in the same way as any other port, they can be used by developers in other components that process telemetry such as a telemetry buffer for downlinking telemetry. The files in this directory are as follows:

File	Description
TlmPortAi.xml	XML description of a telemetry port
TlmBuffer.hpp(.cpp)	A data buffer class that represents the serialized form of the telemetry channel
TlmString.hpp(.cpp)	A string class used by the telemetry code generator when a string is the telemetry channel
TlmPacket.hpp(.cpp)	A notional class representing an encoded telemetry packet that contains a telemetry channel identifier and serialized value. The code generator does not depend on this class, so it can be modified or not used.

5.3.8 Log

This directory contains XML and class declarations used to generate code logging (event) interfaces for components. Developer implementation code sends log events to capture all the events of interest in a system as they happen. Other components serve to store events for forwarding to a ground interface or test software. An XML definition for telemetry ports is defined which the code generator then uses as special logging output ports for components. Since the ports themselves are generated in the same way as any other port, they can be used by developers in other components that process logging such as a logging history. The files in this directory are as follows:

File	Description
LogPortAi.xml	XML description of a logging port
LogTextPortAi.xml	XML description of an optional text logging port. This port generates a string representing the logged event. The code generator generates code to use both kinds of logging ports, but the text log can be disabled.
LogBuffer.hpp(.cpp)	A data buffer class that represents the serialized form of the log entry
LogString.hpp(.cpp)	A string class used by the log code generator when a string is the telemetry channel
TextLogString.hpp(.cpp)	A string class used the text log interface to pass strings
LogPacket.hpp(.cpp)	A notional class representing an encoded log packet that contains a log entry identifier and serialized set of values. The code generator does not depend on this class, so it can be modified or not used.

5.3.9 Prm

This directory contains XML and class declarations used to generate parameter interfaces for components. Parameters are values are meant to be stored in non-volatile storage that affect various properties of the software. Parameters are loaded at run time and given to components on request. An XML definition for a parameter port is used by code generator to create special parameter output ports for components. Since the ports themselves are generated in the same way as any other port, they can be used by developers in other components that provide parameters. The files in this directory are as follows:

File	Description
PrmPortAi.xml	XML description of a parameter port
PrmBuffer.hpp(.cpp)	A data buffer that represents a serialized parameter
PrmString.hpp(.cpp)	A string class used by the parameter code generator when a string is the parameter value.
PrmPacket.hpp(.cpp)	A notional class representing an encoded parameter packet that contains a parameter identifier and serialized value. The code generator does not depend on this class, so it can be modified or not used.

5.3.10 Time

This directory contains XML and class declarations used to generate time interfaces for components. The time interface port is created by the code generator as a source of time for time-tagging telemetry samples and log events. Since the ports themselves are generated in the same way as any other port, they can be used by developers in other components that provide time from whatever sources are present in the system. The files in this directory are as follows:

File	Description
TimePortAi.xml	XML description of a time port
Time.hpp	A class containing a time value that represents the time when the port was invoked.

5.3.11 Com

This directory contains definitions for a communication port. This port could be used as an interface to components that send and receive data to ground or test software. The files are as follows:

File	Description
ComPortAi.xml	XML description of a communication port
ComBuffer.hpp(.cpp)	A data buffer class used to represent a packet of serialized communication data.
ComPacket.hpp(.cpp)	A data packet class representing data from one of the telemetry or command types. The specific packet types are derived classes.

5.4 Svc

The intent of this directory is to provide a set of components implementing services that would be useful for a flight application. The service layer in software is traditionally the layer that provides mechanism for executing the software and managing data. The components, ports and other types are examples of how the architecture can be applied. The component example implementations are very simple; flight versions would most likely be more sophisticated. The way development is done is that the developer will define the components and their properties in XML. The code generator will generate C++ classes that encapsulate the features of the component. The developer will then write a class the derives from those generated classes and implement the port methods. For these directories, each file will not be described, but a higher-level description of what each directory contains will be given instead. The descriptions are as follows:

5.4.1 ActiveLogger

This directory contains a component XML description and implementation for an active component that accepts serialized log events. The input port accepting log entries puts them in a message queue for the component thread. The component thread calls the port handler in the derived class written by the developer. In this case, the handler simply takes the log entry, serializes it into a communications buffer, and sends it out via the output communications port. There are commands for filtering the event levels.

5.4.2 ActiveRateGroup

This directory contains a component implementation of an active rate group. In real-time programming, a rate group is a thread of execution that does a sequential set of operations that execute cyclically and are required to complete by a certain deadline. In this case, this component provides the thread for the rate group and sequentially calls a set of output ports that would be connected to other components doing the specific operations that are required.

5.4.3 CmdDispatcher

This directory contains an implementation of a command dispatcher. The commands are received in serialized form from another component. The command identifier (opcode) is deserialized and an output port is looked up that is matched with that identifier. The port is then invoked with the serialized command arguments. After executing the command, the component responds back to the dispatcher via a response port and reports on the outcome of the command. The dispatcher then calls a status output port if connected in the event there was something else such as a sequencer waiting for a result. During initialization, a command registration port is called by components to match the identifier with the port the component is executing commands from. The component also implements some NO_OP commands.

5.4.4 CmdSequencer

This directory contains an implementation of a command sequencer. A sequence file uploaded to the system contains a set of commands that are executed in order with optional time points. The sequencer waits until the current command is complete before executing the next in the sequence. A failed command terminates the sequence.

5.4.5 CmdRecord

This directory specifies the port used to pass command buffers between the SequenceFileLoader and the SequenceRunner components.

5.4.6 Cycle

This directory specifies the port used to drive the ActiveRateGroup components. The port passes a time stamp indicating when the cycle was started.

5.4.7 Fatal

The directory specifies a port used to pass a notification that a FATAL event has occurred. It is currently produced by the ActiveLogger component when it receives a FATAL event from a component.

5.4.8 GndIf

The directory contains just the XML definition of a component that could be used to send and receive communications packets. The uplink port would be connected to the command dispatcher for executing commands and the downlink port would be connected to by components collecting downlink telemetry like logs and channelized data. A derived class implementing a TCP/IP socket version can be seen in SocketGndIf.

5.4.9 Hub

A hub is a pattern for communicating between computing nodes. A hub component is a component that contains input and output serialized ports. As mentioned in section 5.3.4, when a typed port is connected to a serialized port, the port arguments are serialized and passed as a data buffer to the serialized port. Likewise, when a data buffer is sent from a serialized port to a typed port, the data is deserialized back in to the typed arguments of the port. A hub component takes the serialized data passed to it and sends it via a communication channel to a hub on a remote node. The remote hub takes that data and calls a serialized output port connected to a typed port of the same type as the original port. That allows components to be interconnected across computing nodes without any modifications to the components themselves. This particular hub is implemented as an active hub, which means that the serial buffers from the incoming ports are put in a message queue and then sent out the *DataOut* ports on the thread of the component. The data output ports are in the form of a generic com buffer. The data output ports would be connected to a component that manages the communication hardware. Likewise, incoming data on the *DataIn* port is queued for the component thread, which sends it out via a serialized output port.

5.4.10 LinuxTime

This component is an adaption of the time source component specified in the *Time* directory. In this case, it is a time source that makes a Linux system call for the Linux demo.

5.4.11 PassiveConsoleTextLogger

This is an adaptation of the *PassiveTextLogger* component that simply takes the text version of the log and prints it to the standard output.

5.4.12 PassiveRateGroup

This is a rate group with the same implementation as the active rate group in section 5.4.2, with the exception that the component does not have a thread. The thread that calls the input run port will be used to call all the output ports. This component is not currently used in the reference application.

5.4.13 PassiveTextLogger

This defines a base class for a component that prints the text version of events. It executes on the thread of the caller. The implementation classes are elsewhere, such as PassiveConsoleTextLogger.

5.4.14 PolyDb

This component implements a database of PolyType entries. The intent is for this to be used as a database of values being used by different components in the system. Some components submit new values they have gathered, and other components retrieve the ones they use.

5.4.15 PolyIf

This contains the definition for a PolyPort, or a port that passes a PolyType. It is used to set and get values for the PolyDb component.

5.4.16 PrmDb

This component implements storage for parameters. It implements the framework setPrm and getPrm ports. Parameter values are stored as a table based on parameter ID. It reads a file during initialization that contains the parameter values and loads them into memory. Subsequent calls to getPrm will get the loaded file. Parameter values in the components can be updated by command and saved to PrmDb. The PrmDb component can be commanded to save the updated values to a file.

5.4.17 RateGroupDriver

This component takes a primary clock tick in the system and divides it down to drive output ports. Constructor arguments define the divisors for each port. The output ports are meant to be connected to the input ports of rate groups to drive them at the correct rate.

5.4.18 Sched

This directory contains the definition of a scheduler port that is used by the rate group components and rate group members.

5.4.19 SequenceFileLoader

This directory contains a component that loads a set of command buffers from a file and passes them to the SequenceFileRunner component.

5.4.20 SequenceRunner

The directory contains a component that will sequence through a series of command buffers passed through its CmdRecord port.

5.4.21 SocketGndIf

This directory contains a notional uplink/downlink component that communicates with ground software via a TCP/IP socket. It would be connected to telemetry sources and the command dispatcher.

5.4.22 Time

This directory contains the XML definition for the time source base class. A time source is necessary for time-tagging the telemetry and log events in components. Various implementations that derive from this base class will provide time.

5.4.23 Tlm

This directory defines a passive telemetry storage component base class. It has as an input port for the telemetry buffers sent by components. It has an output port to send the telemetry packets to a ground interface component like the one in section 5.4.21. It has a scheduler input port so that it can be executed periodically on a rate group to send the stored telemetry to the ground interface.

5.4.24 TlmChan

This directory contains an adaptation of the *Tlm* base class in section 5.4.23. In this adaptation, the telemetry is stored in an array of telemetry buffers based on the telemetry ID. The storage is double-buffered. When the scheduler port is invoked, the component switches the active array to non-active and starts copying the non-active array to the packet output port. If incoming telemetry calls happen during the copy operation, they are placed in the active array.

5.5 Os

This directory contains classes that abstract operating system features. This allows the components that are code generated to not be dependent on a particular operating system. The architecture is dependent on these classes. The subdirectories contain implementations of the class for different operating systems. Not all operating systems will implement all classes. The classes are as follows:

5.5.1 Task

This class represents a task (AKA thread) in an operating system process. It has methods for starting, ending, waiting and suspending tasks. This class is used by active components.

5.5.2 Queue

This class represents a message queue. It has methods for creating, writing to, reading from and destroying queues. This class is used by queued components.

5.5.3 Mutex

This class represents a mutex. It is used to guard critical sections of data and code. It is used by components that have guarded ports (see architecture description).

5.5.4 File

This class represents a file. It is used to abstract away various operating system implementations of file I/O.

5.5.5 InteruptLock

5.6 Drv

The Drv directory contains some hypothetical device driver components and types. It is part of the example code. The architecture does not depend on source code in this directory. There is no particular significance to the name; rather it was selected to represent how a developer might organize their code. The subdirectories are as follows:

5.6.1 BlockDriver

This represents a hardware driver that accepts buffers of data to send to a device, and sends buffers that it receives from the device. Since there is no real hardware behind the driver, the driver takes any incoming data buffers from the input port and sends them out the output port.

5.6.2 DataTypes

This directory contains the port and data buffer types used by the driver and components using the driver.

5.6.3 Pci

This directory contains a class that has been used on other projects to interface with a PCI bus. Subdirectories contain various implementations of the class for different hardware targets used in the past. This class could be used by driver component implementations.

5.7 Ref

This directory contains a reference application. Components here represent what an adapter might do when writing application-specific logic. An adapter would use the framework layers, drivers and services that are meant to be reusable along with application components for a particular task.

5.7.1 SendBuffApp

This passive component represents a part of the application that sends data to a consumer. It runs periodically in a rate group and sends a packet upon command. It uses the "driver" described in section 5.6.1.

5.7.2 RecvBuffApp

This active component represents a part of the application that receives data from a sender, in this case SendBuffApp. It receives a buffer from the driver.

5.7.3 Top

This is the "topology" module. This is where all the components are instantiated and connected together, and the active components are started. It is also the location of the C *main()* function entry point. Each deployment (see section 6.2) will have a module similar to this.

6 Developing Software

Before developing components, it is helpful to read the architectural description to understand the concepts behind the architecture. It can be found in docs/Architecture/F` Architecture.docx.

The software development process for F` consists of defining the various architectural elements in XML, using those elements at the appropriate places in the architecture, and writing derived component classes that do the project specific logic. The code generation tools will produce the appropriate C++ classes to implement the architecture. The user writes derived component classes that implement the component interfaces. The build system supplied with the framework invokes the code generator with the correct dependencies and compiles the output together with the user.

As described in the architectural document, the layering in the architecture is such that the order of definition should start with Serializable types, then Ports, then Components, and finally Topologies. From a practical perspective, the types that are shared between ports and components should be defined in separate libraries so the code can be shared without creating linker dependencies.

6.1 Modules

A module is defined as a set of files that are compiled to form a library. The build system provides a configuration file for each module that lists the files that are to be compiled. A template for this file can be found in `mk/make_templates/mod.mk`. The `mod.mk` file should be placed in each directory where code is located that is part of the module. The contents of the `mod.mk` file is as follows:

Variable	Description
SRC	Contains all the source files that are built for all targets for the modules. All the XML files as well as C and C++ files that are portable go in this variable, separated by spaces. Multiple lines can be separated by the continuation character (\).
SRC_XXX	Contains source that is particular to a specific build. The values of XXX are defined by the different builds supported by the make system. To determine the different values, look at the detailed description of the make system in Section XXX .
SUBDIRS	A list of subdirectories that contain further mod.mk files. These subdirectories can contain code to test the module. Details in writing and compiling test code can be found in section 8.
COMPARGS, COMPARGS_XXX	A list of compiler flags specific to the module or target. See the make system description.
TEST_XXX	Variables used for test code. See the section on test code.

Table 1 - mod.mk variable descriptions

6.2 Deployments

A deployment is defined as a set of module libraries that are linked together to form an executable for a particular purpose or location. An example is that a project may have multiple computing nodes each with their own executable that communicate with each other to perform a set of tasks. Each of the individual executables would be considered a deployment. Modules can be shared amongst any number of deployments. The collection of modules that form a deployment are typically set by the software architect. For a small project with only one processor, typically only one deployment would be defined. The file that specifies the deployments and the modules that consist of them are specified in `mk/configs/modules/modules.mk`. The variables and values must follow GNU make variable syntax. The contents of `modules.mk` are as follows:

Variable	Description
DEPLOYMENTS	Contains a list of deployment names.
<DEPLOYMENT>_MODULES	For each deployment name, these variables define the set of modules used to build the deployment. It can contain modules or other variables that define a set of modules. For modules, the values should be the path to the module directories relative to the root of the source tree.
Other variables…	Other variables can be defined that follow gnu make syntax rules. A common use of these is to group modules together that are commonly used by multiple deployments.

6.3 Basic Types

A number of basic types are defined and are used throughout the architecture and are usable within the XML specifications as well as user-written code. They are defined in the header file *Fw/Types/BasicTypes.hpp*. The types are:

Type	Description
I8	8-bit signed integer
U8	8-bit unsigned integer
I16	16-bit signed integer
U16	16-bit unsigned integer
I32	32-bit signed integer
U32	32-bit unsigned integer
F32	32-bit IEEE floating point number (float)
F64	64-bit floating point number (double)
I64	64-bit signed integer
U64	64-bit unsigned integer
bool	C++ Boolean type

Table 2 - Architecture Basic Types

These types are used in XML specifications. Not all types are available on all processor architectures. Which types are available is a configurable feature of the architecture and is typically set by compiler arguments.

6.4 Adaptive Types

The representation of some types change their size depending on the processor architecture. For instance, on a modern Intel or ARM processor, the size of the C type *int* may be 64 bits, while a microcontroller may use 8 bits. Not all of the types listed in section 6.3 may even be available. For this reason, some macro definitions of native types can be used for variables that don't require a particular size. For example, an index in a *for* loop could use these. They can also be found in *Fw/Types/FwBasicTypes.hpp*. The types are:

Type	Description
NATIVE_INT_TYPE	A signed integer
NATIVE_UINT_TYPE	An unsigned integer
POINTER_CAST	If a pointer needs to be cast to an integer for storage or passing through a function call, this can be used to make sure the pointer doesn't lose some of the address bits. As an example, if a pointer were cast and stored in a U32 on a 64-bit processor, the pointer value would lose the upper 32 bits. If it is stored in a POINTER_CAST, it will retain all 64 bits.

6.5 Polymorphic Type

A polymorphic class type is defined in *Fw/Types/PolyType.hpp*. This class is meant to store the value for a variety of built-in types. The type stored can be changed during runtime by reassigning a value of a variable of different type. PolyType can be passed through ports and serialized.

6.5.1 Setting Values

The PolyType object can have a value assigned to it via the constructor or the *equals* operator:

```
U32 val =10;
PolyType pt(val);
U32 val2 = 13;
pt = val2;
```

6.5.2 Getting Values

The value stored in the PolyType object can be retrieved two ways:

1. Cast the object to the type of the value:
   `U32 val = (U32)pt;`
2. Use the "get()" method.
   ```
   U32 val;
   pt.get(val);
   ```

In both cases, if the type being retrieved does not match the stored type, the code will assert.

6.5.3 Checking Values

The PolyType instance has "isXXX" functions for checking what type is being stored, where "XXX" is the name of the type.

6.6 Defining XML Types

Serializable types, ports, and components are defined in XML files. Those files are parsed by the code generator and files containing C++ class declarations are created. This section will describe the syntax of the XML files.

6.6.1 Serializable

Serializables are the types that are passed between components in the architecture and are used for ground data services such as commands, telemetry, events and parameters (see the component description for what these are). A serializable type is a complex or basic type that can be converted to a data buffer that contains the values of an instance of the type. This data buffer can be passed around the system (and to ground software) in a generic way and reconstituted into the original type as needed. Basic C/C++ types like `int` and `float` are supported automatically in the architecture, but the user is also allowed to define arbitrary complex types that can be serialized. This is done in one of two ways, either by hand-coding a class or allowing the code generator to generate the class. These types can then be used in XML specifications of ports and components.

6.6.1.1 XML Specified Serializable

In most cases, a complex type is representable as a collection of simple types. This is analogous to a C struct with data members. The code for serializing and deserializing a struct is straightforward, so a code generator is provided the will create the classes for a serializable struct. The user specifies the members in XML, and the C++ class is generated. The types of the members can be basic types or other serializables, either XML or hand-coded. In order for the build system to detect that a file contains the XML for a serializable type, the file must follow the naming convention <SomeName>SerializableAi.xml. An example of this can be found in `Autocoders/templates/ExampleSerializableAi.xml`. The XML tags and attributes to use for the serializable are as follows:

Tag	Attribute	Description
serializable		The outermost tag the indicates that a serializable is being defined
serializable	namespace	The C++ namespace for the serializable class (optional)
serializable	name	The class name for the serializable
serializable	typeid	An integer uniquely identifying the type. Optional, generated from hash otherwise. Should be unique across the application.
comment		Used for a comment describing the type. Is placed as a Doxygen compatible tag in the class declaration
members		Starts the region of the declaration where type members are specified
member		Defines a member of the type
member	type	The type of the member. Should be one of the types defined in Table 2, ENUM, string, an XML specified serializable, or a user-written serializable.
member	name	Defines the member name
member	size	Specifies that the member is an array of the type with the specified size.
member	format	Specifies a format specifier when displaying the member
enum		Specifies an enumeration when the member type=ENUM
enum	name	Enumeration type name
item		Specifies a member of the enumeration
item	name	Specifies the name of the enumeration member
item	value	Assigns a value to the enumeration member. Member values in the enumeration follow C enumeration rules if not specified.
item	comment	A comment about the member. Becomes a Doxygen tag.
import_serializable_type		Imports an XML definition of another serializable type for use in the definition.
include_header		Includes a C/C++ user-written header for member types.

Table 3- Serializable XML Specification

6.6.1.1.1 Constraints

6.6.1.1.1.1 Ground Interfaces

Serializables used for defining the interfaces for the ground system must be fully specified in XML files. Members of the serializables cannot be hand-coded types since the ground system analyzes the XML to determine the types for displaying and archiving the data. (See section 6.6.3 for component ground interface specifications).

6.6.1.2 Hand-coded Serializable

A hand-coded serializable is useful in the case where the type is not easily expressed as a collection of basic types or if the user is using an external library with types already defined. A user may also wish to attach special processing functions to a class. The pattern for writing C++ serializable classes is to declare a class that is derived from the framework base class `Fw::Serializable`. That type can be found in `Fw/Types/Serializable.hpp`. An example can be found in `Autocoders/templates/ExampleType.hpp`. The method is as follows:

1) Define the class. It should be derived from Fw::Serializable.
2) Declare the two base class pure virtual functions, `serialize()` and `deserialize()`. Those functions provide a buffer as a destination for the serialized form of the data in the type.
3) Implement the functions. The buffer base class that is the argument to those functions provides a number of helper functions for serializing and deserializing the basic types in the table above. Those functions are specified in the `SerializeBufferBase` class type in the same header file. For each member that the developer wishes to save, call the serialization functions. The members can be serialized in any order. In the worst case, a raw buffer version is provided for just doing a bulk copy of the data if serializing individual members is not feasible. The deserialization function should deserialize the data in the order it was serializaed.
4) Add an enumeration with a single member named SERIALIZED_SIZE. The value of that member should be the sum of the sizes of all the members. It can be done with the sizeof() function that is available to all C/C++ compilers. Optionally, a type identifier can be added so that a sanity check can be done when the data is deserialized. That type should be serialized, and then checked against the enumeration when it is deserialized.
5) Implement copy constructors and equal operators. If the type is passed by value, the developer should write these functions if the default isn't sufficient.
6) Implement any custom features.

Once the class is completed, it can be included in port definitions.

6.6.1.2.1 Constraints

6.6.1.2.1.1 Ground Interfaces

Hand-coded user types cannot be used in XML definitions for types that will be sent to and from the ground system, since the ground system requires all type definitions to be in XML. If a developer wants to use a type that has special functions in a ground interface, it can be defined with the members only in XML and generated by the code generator. Then the developer can define a derived class with the behavior.

6.6.2 Port

Ports are the interfaces between components. Components invoke functionality in other components by invoking methods on their output ports instead of invoking the components directly. Input ports invoke methods defined by the components and registered at run time. Ports are fully specified in XML files. There is no developer written code, although the XML specifications can include argument types defined by the user as long as they are serializable as described in section 6.6.1.2. The process of defining a port is to specify the arguments to the method in the XML file. In order for the build system to detect that a file contains the XML for a port type, the file must follow the naming convention <SomeName>PortAi.xml. An example of this can be found in `Autocoders/templates/ExamplePortAi.xml`. The XML tags and attributes are as follows:

Tag	Attribute	Description
interface		The outermost tag the indicates that a port is being defined
interface	namespace	The C++ namespace for the port class (optional)
interface	name	The class name for the port
comment		Used for a comment describing the port. Is placed as a Doxygen compatible tag in the class declaration
args		Optional. Starts the region of the declaration where port arguments are specified.
arg		Defines an argument to the port method
arg	type	The type of the argument. Should be one of the types defined in Table 2, ENUM, "string", an XML specified serializable, or a user-written serializable. A "string" type should be used if a text string is the argument. If the type is "Serial", the port is an untyped serial port that can be connected to any typed port for the purpose of serializing the call. See the Hub pattern in the architectural description document for a usage.
arg	name	Defines the argument name
arg	size	Specifies the size of the argument if it is of type "string".
arg	pass_by	Optional. Specifies if the argument should be passed by reference or pointer. Default is to pass by value. Values can be "value","pointer", or "reference." This is provided as an optimization to avoid unnecessary copies. When arguments to ports with references are detected by components, the argument is serialized the same way as an argument that is passed by value. If it is passed by pointer, the pointer value is copied and not the contents of the type instance pointed to by the pointer. This carries the usual responsibility for understanding the scope and lifetime of the memory behind the pointer, as the pointer value may be held by another component past the duration of the port call.
enum		Specifies an enumeration when the argument type=ENUM

enum	name	Enumeration type name
item		Specifies a member of the enumeration
item	name	Specifies the name of the enumeration member
item	value	Assigns a value to the enumeration member. Member values in the enumeration follow C enumeration rules if not specified.
item	comment	A comment about the member. Becomes a Doxygen tag.
return		Optional. Specifies that the method will return a value.
return	type	Specifies the type of the return. Can be the types defined in Table 2.
return	pass_by	Specifies whether the argument should be passed by value or by reference or pointer.
import_serializable_type		Imports an XML definition of another serializable type for use in the definition.
include_header		Includes a C/C++ user-written header for argument types.

Table 4- Port XML Specification

6.6.2.1 Constraints

The following constraints are on port XML definitions:

6.6.2.1.1 Serialization

Port calls are serialized for various reasons by components, among them copying data to put on a message queue. A port call cannot be serialized if it contains return values, since the serialized arguments only are passed to the connected port or component and no return data is provided.

6.6.3 Components

Components are the places in the architecture where all the behavior resides. Incoming port calls deliver data and actions to be acted on, and output ports are used to invoke functions outside the component. In addition, the components define the interfaces used to communicate with ground software. Components can be used for a variety of purposes such as hardware drivers, state machines, device managers, and data management. The process of defining a component is to specify the ports and ground data interfaces in XML. The developer then implements a derived class which inherits from the base class and implements the interfaces as methods in the class (see section 6.7). In order for the build system to detect that a file contains the XML for a component type, the file must follow the naming convention <SomeName>ComponentAi.xml. An example of this can be found in `Autocoders/templates/ExampleComponentAi.xml`. The XML tags and attributes are as follows:

Tag	Attribute	Description
component		The outermost tag the indicates that a component is being defined
component	namespace	The C++ namespace for the component class (optional)
component	name	The class name for the component

<table>
<tr><td>component</td><td>kind</td><td>The type of component. Can be "passive","queued", or "active". A passive component has no thread or queue. A queued component has a message queue, but no thread. A component on another thread must invoke a synchronous input interface (see below) to get messages from the queue. An active component has a thread, and unloads its message queue as the thread is scheduled and as port invocation messages arrive.</td></tr>
<tr><td>component</td><td>modeler</td><td>When the attribute is "true", the autocoder does not automatically create ports for commands, telemetry, events, and parameters. If it is "true", those ports must be declared in the port section with the "role" attribute. See description below.</td></tr>
<tr><td>import_dictionary</td><td></td><td>Imports a ground dictionary defined outside the component XML that conforms to the command, telemetry, event and parameter entries below. This allows external tools written by projects to generate dictionaries.</td></tr>
<tr><td>import_port_type</td><td></td><td>Imports an XML definition of a port type used by the component.</td></tr>
<tr><td>import_serializable_type</td><td></td><td>Imports an XML definition of a serializable type for use in the component interfaces.</td></tr>
<tr><td>comment</td><td></td><td>Used for a comment describing the component. Is placed as a Doxygen compatible tag in the class declaration.</td></tr>
<tr><td>ports</td><td></td><td>Defines the section of the component definition where ports are defined.</td></tr>
<tr><td>port</td><td></td><td>Starts the description of a port.</td></tr>
<tr><td>port</td><td>name</td><td>Defines the name of the port.</td></tr>
<tr><td>port</td><td>data_type</td><td>Defines the type of the port. The XML file containing the type of the port must be provided via the import_port_type tag.</td></tr>
<tr><td>port</td><td>kind</td><td>Defines the synchronicity and direction of the port. The port can be of the following kinds:
<table>
<tr><th>Kind</th><th>Attributes</th></tr>
<tr><td>sync_input</td><td>Invokes the derived class methods directly.</td></tr>
<tr><td>guarded_input</td><td>Invokes the derived class methods after locking a mutex shared by all guarded ports and commands.</td></tr>
<tr><td>async_input</td><td>Creates a message with the serialized arguments of the port call. When the message is dequeued, the arguments are deserialized and the derived class method is called.</td></tr>
<tr><td>output</td><td>Port is an output port that is invoked from the logic in the derived class.</td></tr>
</table></td></tr>
</table>

<table>
<tr><td>port</td><td>priority</td><td>The priority of the invocation. Only used for asynchronous ports, and specifies the priority of the message in the underlying message queue if priorities are supported by the target OS. Range is OS dependent.</td></tr>
<tr><td>port</td><td>max_number</td><td>Specifies the number of ports of this type. This allows multiple callers to the same type of port if knowing the source is necessary. Can be specified as an Fw/Cfg/AcContstants.ini file "$variable" value.</td></tr>
<tr><td>port</td><td>full</td><td>Specifies the behavior for async ports when the message queue is full. One of "drop","block", or "assert," where "assert" is the default</td></tr>
<tr><td>port</td><td>role</td><td>Specifies the role of the port when the modeler = true</td></tr>
<tr><td>commands</td><td></td><td>Optional. Starts the section where software commands are defined.</td></tr>
<tr><td>commands</td><td>opcode_base</td><td>Defines the base value for the opcodes in the commands. If this is specified, all opcodes will be added to this value. If it is missing, opcodes will be absolute. This tag can also have a variable of the form "$variable" referring to values in Fw/Cfg/AcContstants.ini.</td></tr>
<tr><td>command</td><td></td><td>Starts the definition for a particular command.</td></tr>
<tr><td>command</td><td>mnemonic</td><td>Defines a text label for the command. Can be alphanumeric with "_" separators, but no spaces.</td></tr>
<tr><td>command</td><td>opcode</td><td>Defines an integer that represents the opcode used to decode the command. Should be a C compilable constant.</td></tr>
<tr><td>command</td><td>kind</td><td>Defines the synchronicity of the command. The command can be of the following kinds:
<table>
<tr><th>Kind</th><th>Attributes</th></tr>
<tr><td>sync</td><td>Invokes the derived class methods directly.</td></tr>
<tr><td>guarded</td><td>Invokes the derived class methods after locking a mutex shared by all guarded ports and commands.</td></tr>
<tr><td>async</td><td>Creates a message with the serialized arguments of the port call. When the message is dequeued, the arguments are deserialized and the derived class method is called.</td></tr>
</table></td></tr>
<tr><td>command</td><td>priority</td><td>Sets command message priority if message is asynchronous, ignored otherwise.</td></tr>
<tr><td>command</td><td>full</td><td>Specifies the behavior for async commands when the message queue is full. One of "drop","block", or "assert," where "assert" is the default</td></tr>
<tr><td>args</td><td></td><td>Optional. Starts the region of the declaration where</td></tr>
</table>

		command arguments are specified.
arg		Defines an argument in the command.
arg	type	The type of the argument. Should be one of the types defined in Table 1, ENUM, "string", or an XML specified serializable. A "string" type should be used if a text string is the argument.
arg	name	Defines the argument name
arg	size	Specifies the size of the argument if it is of type "string".
enum		Specifies an enumeration when the argument type=ENUM
enum	name	Enumeration type name
item		Specifies a member of the enumeration
item	name	Specifies the name of the enumeration member
item	value	Assigns a value to the enumeration member. Member values in the enumeration follow C enumeration rules if not specified.
item	comment	A comment about the member. Becomes a Doxygen tag.
telemetry		Optional. Specifies the section that defines the channelized telemetry.
telemetry	telemetry_base	Defines the base value for the channel IDs. If this is specified, all channel IDs will be added to this value. If it is missing, channel IDs will be absolute. This tag can also have a variable of the form "$variable" referring to values in Fw/Cfg/AcContstants.ini.
channel		Starts the definition for a telemetry channel.
channel	id	Specifies a numerical value identifying the channel
channel	name	A text string with the channel name. Cannot contain spaces.
channel	data_type	Specifies the type of the channel. Should be one of the types defined in Table 1, ENUM, "string", or an XML specified serializable. A "string" type should be used if a text string is the argument.
channel	size	If the channel type is "string," specifies the size of the string.
channel	abbrev	An abbreviation for the channel. Needed for AMPCS.
channel	update	If the channel should be always updated, or only on change. Values are "always" or "on change"
channel	format_string	A format string specifier for displaying the channel value.
comment		A comment describing the channel.
enum		Specifies an enumeration when the channel type=ENUM
enum	name	Enumeration type name
item		Specifies a member of the enumeration

item	name	Specifies the name of the enumeration member
item	value	Assigns a value to the enumeration member. Member values in the enumeration follow C enumeration rules if not specified.
item	comment	A comment about the member. Becomes a Doxygen tag.
parameters		Optional. Specifies the section that defines parameters for the component.
parameters	parameter_base	Defines the base value for the parameter IDs. If this is specified, all parameter IDs will be added to this value. If it is missing, parameers IDs will be absolute. This tag can also have a variable of the form "$variable" referring to values in Fw/Cfg/AcContstants.ini.
parameters	opcode_base	Defines the base value for the opcodes in the parameter set and save commands. If this is specified, all opcodes will be added to this value. If it is missing, opcodes will be absolute. This tag can also have a variable of the form "$variable" referring to values in Fw/Cfg/AcContstants.ini.
parameter		Starts the definition for a parameter
parameter	id	Specifies a numeric value that represents the parameter
parameter	name	Specifies the name of the parameter
parameter	data_type	Specifies the type of the parameter. Should be one of the types defined in Table 1, ENUM, "string", or an XML specified serializable. A "string" type should be used if a text string is the argument.
parameter	size	Specifies the size of the parameter if it is of type "string"
parameter	default	Specifies a default value for the parameter if the parameter is unable to be retrieved from non-volatile storage. Only for built-in types.
parameter	comment	A comment describing the parameter.
parameter	set_opcode	Command opcode used to set parameter
parameter	save_opcode	Command opcode used to save parameter
enum		Specifies an enumeration when the parameter type=ENUM
enum	name	Enumeration type name
item		Specifies a member of the enumeration
item	name	Specifies the name of the enumeration member
item	value	Assigns a value to the enumeration member. Member values in the enumeration follow C enumeration rules if not specified.
item	comment	A comment about the member. Becomes a Doxygen tag.
events		Optional. Specifies the section that defines events for the component.

<table>
<tr><td>events</td><td>event_base</td><td>Defines the base value for the event IDs. If this is specified, all event IDs will be added to this value. If it is missing, event IDs will be absolute. This tag can also have a variable of the form "$variable" referring to values in Fw/Cfg/AcContstants.ini.</td></tr>
<tr><td>event</td><td></td><td>Starts the definition for an event.</td></tr>
<tr><td>event</td><td>id</td><td>Specifies a numeric value that represents the event.</td></tr>
<tr><td>event</td><td>name</td><td>Specifies the name of the event.</td></tr>
<tr><td>event</td><td>severity</td><td>Specifies the severity of the event. The values can be:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>DIAGNOSTIC</td><td>Software debugging information. Meant for development.</td></tr>
<tr><td>ACTVITY_LO</td><td>Low priority events related to software execution.</td></tr>
<tr><td>ACTVITY_HI</td><td>Higher priority events related to software execution.</td></tr>
<tr><td>COMMAND</td><td>Events related to command execution. Should be reserved for command dispatcher and sequencer.</td></tr>
<tr><td>WARNING_LO</td><td>Error conditions that are of low importance.</td></tr>
<tr><td>WARNING_LO</td><td>Error conditions that are of critical importance.</td></tr>
<tr><td>FATAL</td><td>An error condition was encountered that the software cannot recover from.</td></tr>
</table></td></tr>
<tr><td>event</td><td>format_string</td><td>A C-style format string to print a message corresponding to the event. Used for displaying the event in the command/data handling software as well as the optional text logging in the software. (See section 9.12).</td></tr>
<tr><td>comment</td><td></td><td>A comment describing the event.</td></tr>
<tr><td>args</td><td></td><td>Starts the region of the declaration where event arguments are specified.</td></tr>
<tr><td>arg</td><td></td><td>Defines an argument in the event.</td></tr>
<tr><td>arg</td><td>type</td><td>The type of the argument. Should be one of the types defined in Table 1, ENUM, "string", or an XML specified serializable. A "string" type should be used if a text string is the argument.</td></tr>
<tr><td>arg</td><td>name</td><td>Defines the argument name</td></tr>
<tr><td>arg</td><td>size</td><td>Specifies the size of the argument if it is of type "string".</td></tr>
<tr><td>internal_interfaces</td><td></td><td>Optional. Specifies an internal interface for the component. Internal interfaces are functions that can be called internally</td></tr>
</table>

		from implementation code. These functions will dispatch a message in the same fashion that asynchronous ports and commands do. The developer implements a handler in the same way, and that handler is called on the thread of an active or queued component. Internal interfaces cannot be specified for a passive component since there is not message queue. A typical use for an internal interface would be for an interrupt service routine.
internal_interface		Specifies an internal interface call.
internal_interface	priority	Sets internal interface message priority if message is asynchronous, ignored otherwise.
internal_interface	full	Specifies the behavior for internal interfaces when the message queue is full. One of "drop","block", or "assert," where "assert" is the default
comment		A comment describing the interface.
args		Starts the region of the declaration where interface arguments are specified.
arg		Defines an argument in the interface.
arg	type	The type of the argument. Should be one of the types defined in Table 1, ENUM, "string", or an XML specified serializable. A "string" type should be used if a text string is the argument.
arg	name	Defines the argument name
arg	size	Specifies the size of the argument if it is of type "string".

Table 5- Component XML Specification

6.6.3.1 Constraints

The following constraints are on components:

6.6.3.1.1 Passive Components

Passive components cannot have asynchronous input ports or commands, since there is no queue for messaging.

6.6.3.1.2 Queued Components

Queued components must have at least one synchronous or guarded input port or synchronous command so that a calling thread can use the port to retrieve port call messages from the message queue.

Queued components must have at least one asynchronous input port, command or internal interface or there would be no purpose in making the component queued.

6.6.3.1.3 Active Components

Active components must have at least one asynchronous input port, command or internal interface or there would be no purpose in making the component active or have a message queue.

6.6.3.1.4 Command and Telemetry Interfaces

The component XML is parsed by the command and telemetry system in order to construct the data storage and display application. For this reason, only the built-in types in Table 2 or types represented in XML can be used.

6.6.4 Command/Telemetry Ports

When declaring commands, telemetry, events and parameters for a component, the code generator automatically creates the correct set of ports for those interfaces. Those ports are based on normal XML definitions of ports, so those port types can be used as normal ports in other components. As an example, if a component needs to implement commands, the XML would declare those commands as shown in Table 1. Another component needs to act as a command dispatcher to send commands to that component, so the dispatcher component would add an output port of the command type. The dispatcher is handling commands as a generic port invocation rather than deserializing the command arguments as the generated code does in the component that declares the commands. This allows writing components that do processing of commands and telemetry generically without having to treat command and telemetry ports as special cases. The port definition XML files can be included in component XML definitions in the same manner as other ports. The following table provides a list of the types, file names, and descriptions of the ports used for the command and telemetry ports.

Port Type	XML File	Description
Commands		
Command	Fw/Cmd/CmdPortAi.xml	A port that passes a serialized command to a component.
Command Response	Fw/Cmd/CmdResponsePortAi.xml	A port that passes the completion status of a command.
Command Registration	Fw/Cmd/CmdRegPortAi.xml	A port used to request registration of a command. Used during initialization to tell a command dispatcher where to send specific opcodes.
Telemetry		
Telemetry	Fw/Tlm/TlmPortAi.xml	A port that passes a serialized telemetry value.
Time	Fw/Time/TimePortAi.xml	A port that returns a time value for time stamping the telemetry.
Events		
Log	Fw/Log/LogPortAi.xml	A port that passes a serialized event.
LogText	Fw/Log/LogTextPortAi.xml	A port that passes the text form of an event. Can be disabled via configuration of the architecture.
Time	Fw/Time/TimePortAi.xml	A port that returns a time value for time stamping the telemetry.
Parameters		
Parameter	Fw/Prm/PrmPortAi.xml	A port that returns a serialize parameter value

6.6.5 Topologies

The topology (or interconnection) of components can be implemented by interconnecting them manually (see section 6.8) or by specifying them via a Topology XML file. An example of this can be seen in `Ref/Top/RefTopologyAppAi.xml`. The file will be processed by the autocoder if it has the ending `AppAi.xml`. The XML specification for topologies is as follows:

Tag	Attribute	Description
deployment		The outermost tag the indicates that a topology is being defined
assembly		Alternate declaration for "deployment"
deployment	name	The name of the deployment
import_component_type		Imports the XML file that defines a component used in the topology.
instance		Defines a component instance.
instance	name	Name of the component instance. This instance name must match a component object declared in the C++ code.
instance	namespace	C++ Namespace of component implementation type.
instance	type	C++ type of implementation class
instance	base_id	The starting ID value for commands, events and telemetry for this instance of the component. Used to construct dictionary for ground system.
instance	base_id_window	A bookkeeping attribute that the modeler uses to space out the base_id values. It can be omitted if the base_ids are spaced enough to cover the id range in the component.
connection		Defines a connection between two component ports
connection	name	Name of connection
source		Defines the source of the connection
source	component	Defines source component. Must match an instance in instance section above.
source	port	Defines source port on component
source	type	Source port type
source	num	Source port number if multiple port instances
target	component	Defines target component. Must match an instance in instance section above.
target	port	Defines target port on component
target	type	Target port type
target	num	Target port number if multiple port instances

The autocoder will output a source file and header file following the convention <Deployment Name>AppAc.cpp and .hpp. The header file contains a function named

`construct<architecture>Architecture.` This function will call all the connection methods to connect the components. The file requires a header `Components.hpp` in the directory where the XML is defined that has declarations for the implementation class instances in the connections.

6.6.5.1 Constraints

The XML specification for the component requires static declaration of component instances that can be referred to by their object name. If components are instantiated in other ways such as a heap, the manual method can be used.

6.7 Implementation Classes

The code generator takes the XML definitions in the previous section and generates C++ base classes. The developer writes classes that derive from those base classes and implement the project-specific logic. For input ports and commands, the base classes declare pure virtual methods for the derived class to implement. If a developer forgets to implement these functions, the compilation of the code will fail. For output ports, telemetry channels, events, and parameters, the base class provides methods for the base class to call.

Depending on the kind of the component, the virtual calls will be made on the thread of the component itself or the thread of a component calling a synchronous or guarded port.

An example of an implementation class can be seen in *Autocoders/templates/ExampleComponentImpl.cpp/.hpp.*

6.7.1 Ports

6.7.1.1 Input port calls

The pure virtual function to implement a port call is derived from the name of the port declaration in the component XML. The function is declared in the protected section of the class and has the following naming scheme:

<port name>_handler(NATIVE_INT_TYPE portNum, <argument list>) = 0;

where:

<port name> = The name given to the port in the "name=" tag in the port section of the XML.

portNum = If XML writer has defined multiple ports, this allows the developer to know which port was invoked. The value is the port instance indexed to zero. In the event the "max_number" attribute is not specified (i.e. a single input port), this value will be zero.

<argument_list> = The list of arguments specified in the "args" section of the port definition XML.

6.7.1.2 Output port calls

The base class function for outgoing port calls is derived from the name and type of the port declaration in the component XML. The function is declared in the protected section of the class and has the following naming scheme:

<port name>_out(NATIVE_INT_TYPE portNum, <argument list>);

where:

<port name> = The name given to the port in the "name=" tag in the port section of the XML.

portNum = If XML writer has defined multiple ports, this allows the developer to specify which port to invoke. The value is the port instance indexed to zero. In the event the "max_number" attribute is not specified (i.e. a single output port), this value should be set to zero.

<argument_list> = The list of arguments specified in the "args" section of the port definition XML.

The call will invoke the port methods define on whatever component the component is interconnected with. If those ports are defined as synchronous or guarded, the other component's logic will execute on the thread of the call.

If the port is not connected and is called, the code will assert. If the design calls for ports that are optionally connected, the connection status can be checked before calling via this function:

isConnected _<port name>_OutputPort(NATIVE_INT_TYPE portNum);

6.7.1.3 Port number calls

A method in the base class can be called to get the number of ports available. The method has the following naming scheme:

NATIVE_INT_TYPE getNum_<port name>_<direction>Ports(void);

where:

<port name> = The name given to the port in the "name=" tag in the port section of the XML.

<direction> = The direction of the port, *Input* or *Output*.

The developer can use this to automatically scale the code to the number of ports specified in the XML. If the port output function is called with a *portNum* value greater than the number of ports minus one, the code will assert.

6.7.2 Commands

The pure virtual function to implement a command is derived from the mnemonic in the command declaration in the component XML. The function is declared in the protected section of the class and has the following naming scheme:

<mnemonic>_cmdHandler(FwOpcodeType opcode, U32 cmdSeq, < argument list>)= 0;

where:

<mnemonic> = The mnemonic string of the command given in the "mnemonic=" tag in the command section of the XML.

<argument_list> = The list of arguments specified in the "args" section of the command definition XML.

When the command has been completed, a command response method must be called in the base class to inform the dispatcher of the command that it has completed. That function call is as follows:

void cmdResponse_out(FwOpcodeType opCode, U32 cmdSeq, Fw::CommandResponse response);

The *opcode* and *cmdSeq* values passed in by the function should be passed to the command response function as well as a status indicating the success or failure of a command. The opcode is specified in the XML and *cmdSeq* will be set by the command dispatcher to track where the command is in a sequence of commands. If more information about a failure is needed, an event should be specified and called with the additional information (see section 6.7.4). If a command takes a number of steps and the call to the command dispatch function does not complete the command, the opcode and command sequence should be stored for a later call to the command response function.

6.7.3 Telemetry

A telemetry channel is intended to be used for periodically measure data. It is a snapshot in time, and all values may not be permanently recorded and sent to the command and data handling software. The code generator generates a base class function for each telemetry channel defined in the XML. The developer calls this to write a new value of the telemetry being stored. The function is declared in the protected section of the class and has the following naming scheme:

tlmWrite_<channel name>(<type>& arg);

where:

<channel name> = The name given to the port in the "name=" tag in the "channel" section of the XML.

<type> = The non-namespace qualified type of the channel as specified in the "data_type" = tag in the XML.

The argument is always passed by reference to avoid copying. The call internally adds a timestamp to the value. There is a method *getTime()* in the base class if the developer wishes to use a time value for other purposes.

6.7.4 Events

Events are intermittent and are all recorded to reconstruct a series of actions or events after the fact. The code generator generates a base class function for each event defined in the XML. The developer calls whenever the event to be recorded happens. The function is declared in the protected section of the class and has the following naming scheme:

log_<severity>_<event name>(<event arguments>);

where:

<severity> = the value of the "severity" attribute in the XML for the event

<event name> = the name of the event given in the "name" attribute in the XML.

<event arguments> = the argument list of the event

The call internally adds a timestamp to the event. There is a method *getTime()* in the base class if the developer wishes to use a time value for other purposes.

6.7.5 Parameters

Parameters are values that are stored non-volatilely and are ways to influence the behavior of the software without requiring software updates. During initialization, the parameters are retrieved and stored in the component base class for later use be the developer's derived class. If for some reason the parameters cannot be retrieved, the default value specified in the XML is returned. The function is declared in the protected section of the class and has the following naming scheme:

<parameter type> paramGet_<parameter name>(Fw::ParamValid& valid);

where:

<parameter type> = the type of the parameter specified by the "data_type" tag in the XML.

<parameter name> = the name of the parameter given in the "name" attribute in the XML

The parameter value is returned by reference to avoid copying the data. The "valid" value should be checked after the call to see what the status of the parameter value is. The possible values of the status and their meanings are:

Value	Meaning
Fw::PARAM_UNINIT	The code to attempt to retrieve the value was never called. This is most likely an error in forgetting to call the *loadParameters()* public function for the component during software initialization.
Fw::PARAM_VALID	The parameter was successfully retrieved.
Fw::PARAM_INVALID	The parameter was not successfully retrieved and no default was specified.
Fw::PARAM_DEFAULT	The parameter was not successfully retrieved but a default was provided.

Table 6- Parameter Retrieval StatusValues

A virtual method is defined in the base class:

void parameterUpdated(FwPrmIdType id);

By default this method does nothing, but the developer can override the method if the implementation needs a notification if a parameter value is updated. It is called each time a parameter is updated.

6.7.6 Internal Interfaces

When internal interfaces are specified in the component XML, a function is generated that can be called by implementation code to dispatch a message for a message loop. The function has the following name:

<internal interface name>_internalInterfaceInvoke(<arguments>);

A handler function definition is also defined for the function that will be called when the internal interface message is dispatched. The function has the following name:

<internal interface name>_internalInterfaceHandler(<arguments>);

The function is defined as a pure virtual to make sure it is implemented.

6.7.7 Message Pre-hooks

When asynchronous ports or commands are specified, the code generator defines functions that can be called prior to dispatching the message. This provides a lightweight mechanism to do some work before the message is dispatched. The function is defined as a virtual (not pure) function with a default implementation which is empty. The implementer can override the function with an alternate version.

The function name for ports is as follows:

void <port name>_preMsgHook(NATIVE_INT_TYPE portNum, <port arguments>);

The values of the port arguments are passed to the function.

The function name for commands is as follows:

void <command mnemonic>_preMsgHook(FswOpcodeType opcode, U32 cmdSeq);

It does not provide the arguments for the command since they are not extracted until the command message is processed.

6.7.8 Initialization Code

6.7.8.1 Constructor

The component framework has the option of storing component names for component interconnection testing and tracing. This is enabled or disabled via the class naming configuration discussed in section 9.2. The macro that indicates whether or not the naming is enabled is *FW_OBJECT_NAMES*. The developer should define and implement two alternate constructors, one that takes a name argument and one that does not. As seen in the example, the only difference between the two constructor implementations is that the base class constructor needs to be passed the name argument. The user can add any custom constructor code as well, but at this stage the component base class is not initialized so no port calls should be made.

6.7.8.2 Initialization

Each component base class has an *init()* function that must be called before interconnecting components. If the component is queued or active, a queue depth argument must be provided. In addition, there is an optional instance argument if the component is going to be instanced more than once. This function can be called from a parallel *init()* function in the derived class.

6.7.8.3 Task Preamble/Finalizer

Active components provide a preamble prototype for code that can be run once before the thread blocks waiting for port invocations and a finalizer prototype for code that runs when the component exits the

message loop. These two functions are called on the thread of the active component. They are declared as virtual functions in C++, so they are not required. The preamble function is named *preamble(void)* and the finalizer *finalizer(void)*. They can be used to do one-time activities such as data structure initialization and cleanup.

6.8 Constructing the Topology

The executing software consists of a set of interconnected components executing on the threads of the active components or driven by other events in the system such as hardware interrupts or timing sources. This section will describe the steps necessary to get the software up and running.

6.8.1 Instantiating the Components

The constructors in the component base classes have been designed so that the components can be instantiated using whatever memory model the developer wishes. They can be created statically, on the heap, or on the stack. As described in section 6.7.8.1, the constructor has either a name argument or none at all. The developer's derived class constructors may have extra arguments that are particular to that application. If classes are declared statically, the developer should keep in mind uncertainties about execution ordering.

6.8.2 Initializing the Components

As discussed in section 6.8.2, each component has an init() call that initializes the component base classes. This call should be made after instantiating the components. For queued and active components, the queue size is passed. The queue should be sized based on an understanding of task priorities and message traffic between the components.

6.8.3 Interconnecting the Components

The components in the software are interconnected by connecting the ports of the components together. Ports are connected by passing pointers to input ports to the output ports that are calling them. Mathods are generated in the component base classes to get input port pointers and pass them to output ports. The following sections describe the connections of different port types.

6.8.3.1 Interface Ports

Interface ports are the regular ports that are used to connect components together. For each port type and name on a component, the method naming scheme is as follows:

Get input port pointer:

<PortType> get_<port name>_InputPort(NATIVE_INT_TYPE portNum);*

where:

<PortType> = The full port type specified in the "data_type" attribute in the definition of the port in the component XML.

<port name> = The name of the port in the "name" attribute of the port definition.

The *portNum* argument to the method should be set to zero if there is only one instance of the port.

Set output port pointer

The value of the input pointer retrieved via the method in the last section is given to an output port of the same type by calling:

*void set_<port name>_OutputPort(NATIVE_INT_TYPE portNum, <PortType>*port);*

where:

<PortType> = The full port type specified in the "data_type" attribute in the definition of the port in the component XML.

<port name> = The name of the port in the "name" attribute of the port definition.

The *portNum* argument to the method should be set to zero if there is only one instance of the port.

There is a second overloaded version of the method to set an output port when the input port being passed to it is a Serialized port:

*void set_<port name>_OutputPort(NATIVE_INT_TYPE portNum, Fw::InputSerializePort *port);*

This is an alternate way of using ports in a more generic fashion that is covered in section XXXXX.

6.8.3.2 Command Ports

As discussed in section 6.6.4, the code generator will create the correct set of command-related ports for a component that has commands defined. For that component, the functions used to get or set command-related port pointers have standard names. The names are as follows:

Get command input port:

Fw::InputFwCmdPort get_CmdDisp_InputPort(void);*

Set command status port:

void set_CmdStatus_OutputPort(Fw::InputCmdResponse_Port port);*

Set command registration port:

void set_CmdReg_OutputPort(Fw::InputCmdRegPort port);*

For the component(s) that are connected to those ports, they would use the normal methods for accessing the port pointers as described in the last section.

6.8.3.3 Telemetry Ports

The standard port accessor functions for telemetry are as follows:

Set telemetry output ports

void set_Tlm_OutputPort(Fw::InputTlmPort port);*

Set time output ports

void set_Time_OutputPort(Fw::InputTimePort port);*

6.8.3.4 Event Logging Ports

The standard port accessor functions for logging events are as follows:

Set logging output ports

void set_Log_OutputPort(Fw::InputLogPort port);*

void set_TextLog_OutputPort(Fw::InputFwLogTextPort port);*

Set time output ports

void set_Time_OutputPort(Fw::InputFwTimePort port);*

Note that the *set_Time_OutputPort()* call is shared with the telemetry ports. It can be called once for both.

6.8.3.5 Parameter Ports

The standard port accessor functions for parameters is as follows:

void set_ParamGet_OutputPort(Fw::InputFwPrmGetPort port);*

void set_ParamSet_OutputPort(Fw::InputFwPrmSetPort port);*

6.8.4 Registering Commands

The pattern for dispatching commands is for a user-implemented command dispatch component to connect an output command port to the input command port for each component servicing commands. Internally, the dispatcher would map the set of opcodes for a particular component to the port that is connected to that component. To aid that process, the code generator creates a command registration function when there are commands specified for a component. It takes as an argument the port number on the dispatcher component that is connected to the component's command port. The registration port should be connected to the dispatcher's registration port as described in section 6.8.3.2. Then the *regCommands()* method can be called in the component, which will invoke the registration port for each of the opcodes defined. This method should be called for each component that has commands.

6.8.5 Loading Parameters

Section 6.8.3.5 describes how to connect a parameter output port to a component providing parameter storage. After the two components are connected, the base class method *loadParameters()* can be called. That method will request the values of all parameters for that component via the parameter port. From then on they will be available to the implementation class for use. Although it is common to only read parameters at software initialization, there is nothing that prevents a re-read after the software is running by invoking the *loadParameters()* call again in the event the parameter storage was updated.

6.8.6 Starting Active Components

The last action in constructing the topology is to start the tasks for any active components. The *start()* method is found in the *ActiveComponentBase* base class in *Fw/Comp/FwActiveComponentBase.hpp*. The arguments and their meanings are as follows:

Argument	
identifier	A thread-independent value that is used to identify activities of the thread. Should be unique in the system.
priority	The execution priority of the task. 0 = low priority, 255 = high priority
stackSize	The size of the stack given to the task.

Table 7 - Active Component Start() arguments

As mentioned in section 6.7.8.3, the functions *preamble()* and *finalizer()* will be run once before and after the loop waiting for port invocations.

7 Utilities

The framework provides a number of services and utilities for the developer. The service classes are also used by the code generator. They are as follows:

7.1 OS Libraries

The framework provides OS services abstraction classes to allow developers to write code in a more portable fashion. Implementations of these classes are provided for Unix variants (POSIX), Mac OS and VxWorks. Additional ports can be done by providing additional implementations as a new OS is added.

7.1.1 Tasks

Tasks are called threads under Unix variants, and tasks under VxWorks. A developer may wish to start tasks to wait on an event or a resource that is not a port invocation. The OsTask class definition can be found in *Os/OsTask.hpp*. The OS implementations are found in various subdirectories. Active components use these classes to implement their features. The methods of the class are:

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td>start()</td><td>Starts the task. The arguments are:
<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>name</td><td>String name of the task.</td></tr>
<tr><td>identifier</td><td>A user-selected unique integer identifying the task.</td></tr>
<tr><td>priority</td><td>The priority of the task. 0 is lowest, 255 is highest. NOTE: Opposite to VxWorks priorites. VxWorks adaptation reverses the values internally.</td></tr>
</table>
</td></tr>
</table>

	stackSize — The size of the stack, in bytes. **routine** — The function that will be called in the new task context. **arg** — An argument passed to the thread. The *arg* argument to the routine will be set to this value. **cpuAffinity** — In SMP systems, specify a processor core to run task. A value of -1 means no preference.
getIdentifier()	Returns the task identifier passed in the *start()* function. Useful when you have a collection of tasks to iterate over.
delay()	Suspends execution of the task for the specified number of milliseconds.
getNumTasks()	Returns the number of active tasks in the system.
suspend()	Suspends the execution of the task. Not available on all operating systems.
resume()	Resumes execution of the task after a suspend() was issued. Not available on all operating systems.
wasSuspended()	Returns a Boolean to indicated whether or not the task was suspended via the suspend() call, or if it was suspended due to some other issue such as an exception.
isSuspended()	Check to see if the task is currently suspended.
setStarted()	Tells the task object that the task routine was called. Should be called from the routine registered in the start() call.
isStarted()	Returns true if the task routine was started successfully. Set by setStarted().
registerTaskRegistry()	Allows a task registry to be passed for registering the task. This is a static call that should be called only once to register a singleton. See section 7.1.2 for a description.

7.1.2 Task Registry

Task registries are meant to be used to track all Task instances in the system. The concept is that this registry would contain pointers to all the instances and be able to iterate over them and perform actions. The file *Os/Task/Task.hpp* provides a base class definition of a registry with the *addTask()* and *removeTask()* pure virtual function definitions. Developers would write their own derived classes to implement a registry that is appropriate for their system.

7.1.3 Mutexes

The class definition can be found in *Os/Mutex.hpp*. The functions *lock()* and *unlock()* lock and unlock the mutex. The mutex is unlocked after construction.

7.1.4 Message Queues

The class definition can be found in *Os/Queue.hpp*. This class implements message queues. The methods are as follows:

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td>create()</td><td>Creates the message queue. The arguments are as follows:
<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>name</td><td>A string identifying the queue. If the OS supports it, it will be used to name the queue.</td></tr>
<tr><td>depth</td><td>The number of messages that a queue will support before dropping messages.</td></tr>
<tr><td>msgSize</td><td>Maximum size of a message.</td></tr>
<tr><td>block</td><td>A flag to indicate whether or not message receive calls should block.</td></tr>
</table></td></tr>
<tr><td>send()</td><td>Send a message on the queue. The arguments are as follows:
<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>buffer</td><td>Buffer to send. This version of the function takes a reference to a SerializeBufferBase instance. This is used by the framework to send serialized port calls.</td></tr>
<tr><td>priority</td><td>The priority of the message. The</td></tr>
</table></td></tr>
<tr><td>send()</td><td>Send a message on the queue. The arguments are as follows:
<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>buffer</td><td>Buffer to send. This version of the function takes a pointer to a byte buffer.</td></tr>
<tr><td>priority</td><td>The priority of the message. Depending on the underlying OS implementation, the messages are received in priority order.</td></tr>
</table></td></tr>
<tr><td>receive()</td><td>Receive a message from the queue. The arguments are as follows:
<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>buffer</td><td>Serialized buffer to put received message in. This version of the function takes a reference to a SerializeBufferBase instance. This is used by the framework to receive serialized port calls.</td></tr>
<tr><td>priority</td><td>The priority of the received message.</td></tr>
</table></td></tr>
<tr><td>receive()</td><td>Receive a message from the queue. The arguments are as follows:
<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>buffer</td><td>Byte buffer to put received message in. This version of the function takes a pointer to a byte buffer.</td></tr>
<tr><td>capacity</td><td>The capacity of the receiving buffer. If it is not large enough for the message, the message will not be written and the function will</td></tr>
</table></td></tr>
</table>

		return with an error.
	actualSize	The size of the received message. Will not exceed capacity.
	priority	The priority of the received message.
getNumQueues	Returns the number of message queues created in the system.	

7.1.5 Interval Timer

The class definition can be found in *Os/IntervalTimer.hpp*. An interval timer is a facility that starts and stops the measurement of passage of time in the system. It is not an expiration timer that signals when it is complete. It is used to measure execution times of or to timestamp software activities. The methods and their descriptions are:

Method	Description
start()	Saves the start time of the timer.
stop()	Saves the stop time of the timer.
getDiffUsec()	Returns the time difference between start and stop in microseconds.
getDiffUsec()	Overloaded version that takes two times, subtracts them, and returns the difference in microseconds.
getDiffNsec()	Version that takes two times, subtracts them, and returns the difference in nanoseconds.
getDiffRaw()	Returns the difference between two raw time values in raw time.
getSumRaw()	Returns the sum of two raw time values in raw time.
nanoSecPerClockTick()	Returns the number of nanoseconds per clock tick.
toNanoSec()	Returns the number of nanoseconds passed since the hardware clock was zero.
getRawTime()	Gets the current raw time value. Can be used for time-tagging events.

7.1.6 Watchdog Timer

The class definition can be found in *Os/WatchdogTimer.hpp.* A watchdog timer schedules a callback at the specified time in the future. It is a one-shot timer; it needs to be rescheduled each time. The methods and their descriptions are:

Method	Description	
startTicks	Starts the timer and gives the expiration in units of system ticks. The arguments are:	
	Argument	**Description**
	delayInTicks	Ticks to delay. OS/platform specific.
	p_callback	Function to call when timer expires.
	parameter	Value passed to callback
startMs	Starts the timer and gives the expiration in units of milliseconds. The arguments are:	

	Argument	**Description**
	delayInMs	Milliseconds to delay. Could be rounded up to the next system timer interval in ticks.
	p_callback	Function to call when timer expires.
	parameter	Value passed to callback
restart()	Restart the timer using the value passed to *startTicks()* or *startMs()*. If a different expiration time is desired, the start functions should be called instead.	
cancel()	Cancel the timer in progress. The callback will not be called.	

7.1.7 Interrupt Lock

The class definition can be found in *Os/InterruptLock.hpp*. An interrupt lock prevents interrupts from preempting the execution of code execution. It can be used as a very lightweight mutex on platforms that support interrupt locking, but should be used carefully as it is not subject to priorities like a conventional mutex. In addition, it defers interrupts that could be time critical so the code executed between the locking and unlocking should be very short in duration. This is a portable interface, but the user should be aware of the behavior for each OS. The methods and their descriptions are:

Method	Description
lock()	Lock interrupts
unlock()	Unlock interrupts
getKey()	Returns the interrupt lock key, if used by the OS. This is typically not needed.

7.1.8 File

The class definition can be found in *Os/File.hpp*. The File class attempts to abstract the most common file functions to a class interface to make porting code that does file accesses easier. The methods and their descriptions are:

Method	Description	
open()	Open the file with the given filename and mode.	
seek()	Move the current location of the file to the offset. If absolute = true, move it relative to the beginning of the file, otherwise relative to the current location.	
read()	Reads data from the file into a buffer. The arguments are as follows:	
	Argument	**Description**
	buffer	Destination buffer for data
	size	Size to read. When read is complete, size is modified to actual size.
	waitForFull	If true, wait until full size is read, continuing reads when signals are encountered. If end-of-file is encountered, will

<table>
<tr><td></td><td><table><tr><td></td><td>return with less than full amount requested.</td></tr></table></td></tr>
<tr><td>write()</td><td>Writes data to the file from a buffer. The arguments are as follows:<table><tr><th>Argument</th><th>Description</th></tr><tr><td>buffer</td><td>Source buffer for data</td></tr><tr><td>size</td><td>Size to write. When write is complete, size is modified to actual size.</td></tr><tr><td>waitForDone</td><td>If true, wait until full size is written, continuing writes when signals are encountered.</td></tr></table></td></tr>
<tr><td>close()</td><td>Close the file. The file is automatically called by the destructor, but this method provides a way to manually close it as well.</td></tr>
<tr><td>getLastError()</td><td>Returns the last error value. Meant to abstract errno.</td></tr>
<tr><td>getLastErrorString()</td><td>Returns a text description of the error for the last file operation. Meant to abstract strerror();</td></tr>
</table>

7.1.9 Directory

The class definition can be found in *Os/Directory.hpp*. The class consists of a set of static functions to do basic directory operations. The methods and their descriptions are:

Method	Description
create	Create a directory at the path location specified. Directory separators are OS specific, but typically "/". If the leading directory separator is omitted, it will be created relative to the current working directory.
remove	Remove the directory at the path specified.
changeTo	Change the working directory to the specified path.

7.1.10 Log

This class definition can be found in *Os/Log.hpp*. It is an interface to a system logging facility. It is meant to abstract the VxWorks logging facility. The methods and their descriptions are:

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td>logMsg</td><td>Log a text message. The arguments are:<table><tr><th>Argument</th><th>Description</th></tr><tr><td>fmt</td><td>Format string to fill and print</td></tr><tr><td>a1 to a6</td><td>Values to be printed. They will be inserted into the format string based on the format string specifiers. The type of the argument is POINTER_CAST (normally an integer), but a typical usage is to cast the values to display to POINTER_CAST and allow the format string extraction to get the correct value. This can include strings (char*), but</td></tr></table></td></tr>
</table>

		the location in memory that holds the string must persist since the format string may be printed on another task in the system.

7.2 Other Utilities

7.2.1 Assert

The framework provides (and uses) asserts to perform run-time checks for software errors. They are a method for verifying conditions that should be true unless there is a software or processor error.

7.2.1.1 Usage

The definition for the framework provided assert can be found in *Fw/Types/Assert.hpp*. The user calls the FW_ASSERT macro with up to six arguments provided. The arguments can consist of any of the types defined in section 6.3. The arguments to the assert call are of type PolyType (see 6.5). There are constructors provided for all the basic types, so the assert can be called with the arguments directly. There will be an implicit temporary PolyType instance created automatically by the compiler.

```
void myfunc(U32 someArg, F32 someVal) {
    // check the value
    FW_ASSERT(someArg < 10,someArg,someVal);
    …
}
```

In the above example, *someArg* is checked to be below a value of 10. If that is not true, the assert will execute and store the values of *someArg* and *someVal* as well as the file and line number. The PolyType instance does not have to be declared explicitly but is done implicitly by converting the argument to a temporary instance of PolyType. It can be assumed that the call to FW_ASSERT will not return.

7.2.1.2 Hook

By default, when FW_ASSERT is called the framework prints a message about the location of and arguments to the macro and then calls the C `assert()` function. Alternatively, the framework also provides a function that allows the registration of a user-defined handler. The handler is `registerAssertHook` and can be found in *FwAssert.hpp*. The assert hook will be called with a string representing the text of the assert. The user implements a derived class that implements the `reportAssert()` pure virtual method, and does whatever project specific logic is required.

7.2.1.3 Configuring

In some cases, it is desirable to remove all asserts once the code is mature. See section 9.4 on how to configure asserts to remove them.

8 Unit Testing

The component architecture supports a pattern for unit testing components by providing the ability to generate counterpart test components from the same XML files that are used to generate the components being tested. In addition, the build system supports building standalone test binaries with only the unit test

code and supporting modules. An example of a unit test can be found in *Autocoders/templates/test/ut*, which tests the component in *Autocoders/templates*. The Linux native target will be used for the example. The procedure for writing a unit test is as follows:

8.1 Generate the Unit Test Component

The first step is to generate the test component.

8.1.1 Generating the Test Component

The build system has specific targets for generating test components. From the module directory where the XML file for the component resides, type:

```
make testcomp
```

This will generate the test component files. They have following names:

TesterBase.hpp(.cpp) – Tester base class. Defines all the routines for checking component interfaces.

GTestBase.hpp(.cpp) – Another base class that adds more checking using the GoogleTest framework. This class is derived from the base class in TesterBase.hpp.

Tester.hpp(.cpp) – A class that automatically connects all the ports between the component and the test component and initializes the components.

If the XML files for a component are changed, the target should be re-run to regenerate the test code.

8.1.2 Move the Test Component Files

Since the component directory itself is meant to be compiled into an overall deployment executable, the test component source files need to be moved to a directory where they can be compiled into a standalone unit test. The build system has some standard targets defined if the unit test files are placed in a specific directory. For a given module, place the generated test component files in *<module directory>/test/ut*. In the module directory, edit the file *mod.mk* and add an entry *SUBDIRS = test*. In the *<module directory>/test* directory, add another *mod.mk* file with the only entry being *SUBDIRS = ut*. Finally, the unit test directory will have a *mod.mk* which has contents similar to the following:

```
TEST_SRC = TesterBase.cpp GTestBase.cpp Tester.cpp TestComponentImpl.cpp

TEST_MODS = <Component under test directory> Fw/Cmd Fw/Comp Fw/Port Fw/Prm
Fw/Time Fw/Tlm Fw/Types Fw/Log Fw/Obj Os Fw/Com
```

The *TEST_SRC* variable defines files locally compiled to the unit test executable. For the regular components in the module directory, placing the XML definitions in the SRC variable in mod.mk file will automatically cause the XML files to be generated into C++ files which are then compiled. For the unit tests, the generated test component files must be manually added to the *TEST_SRC* variable to be compiled. Any other test source code that is used can also be added to *TEST_SRC*.

The *TEST_MODS* variable defines the modules that the unit test executable will link to. The entries are the directory names where the modules reside relative to the root of the source code. The module with the

component being tested must also be listed. The modules for the framework itself need to be included. The following table lists the TEST_MODS entries needed:

TEST_MODS entry	When Needed
Fw/Obj	Always needed; has object base
Fw/Comp	Always needed; has component base classes
Fw/Port	Always needed; has port base classes
Fw/Types	Always needed; has framework type definitions and implementations
Os	Need if an active component is being tested
Fw/Cmd	Needed if the component has commands
Fw/Tlm	Needed if the component has telemetry
Fw/Log	Needed if the component has events
Fw/Prm	Needed if the component has parameters

When the contents of *mod.mk* in the *test/ut* directory have been added, invoke the target "make gen_make" from the deployment directory to make the build system aware of the new files.

8.2 Write a Derived Class

Writing test code for the generated test component follows the same pattern as a regular component. The user defines and implements a class that is derived from the generated test component base class. The difference is that the port invocation directions are reversed, since the ports are reversed. In addition, special methods are generated that help the developer invoke commands, telemetry, events and parameter settings. The developer should derive from the class defined in *GTestBase.hpp* or from the class defined in *TesterBase.hpp* if they don't wish to use GoogleTest.

8.2.1 Ports

Since the ports are reversed, the functions generated to test the port connections are reversed as well. They follow the same pattern as the ports in the regular components – input ports generate pure virtual functions in the test component base class that must be overridden, and output ports are base class member functions that can be called. The naming convention is the same as the regular components. They have the form:

from_<port name>_handler(...);

invoke_to_<port name>(...);

8.2.2 Commands

The test component code generator generates functions in the base class for each command. The function has arguments for each of the arguments specified in the command XML. When the developer wishes to test a command the function can simply be invoked with the desired values. The functions have the following naming convention:

sendCmd_<command mnemonic>_cmd(...arguments)

When the command is invoked and the component being tested reports the command status via the command status function, the *cmdResponseIn()* virtual method will be called in the test component.

8.2.2.1 Command History

8.2.2.1.1 Clearing History

Command history can be cleared by calling:

this->clearHistory();

8.2.2.1.2 Checking History

ASSERT_CMD_RESPONSE_SIZE(n) – check number of command responses

ASSERT_CMD_RESPONSE() – check the contents of the command response

8.2.3 Telemetry

Since the component being tested calls methods to report telemetry and those end up invoking a telemetry output port, the test component base class defines a set of virtual handler functions for each telemetry value. The developer implements the functions to check the telemetry values. The naming convention for the virtual functions is:

tlmInput_<channel name>(Fw::Time &timeTag, <channel type> arg);

8.2.3.1 Telemetry History

8.2.3.1.1 Clearing Telemetry History

Telemetry history can be cleared by calling:

this->clearTlm();

8.2.3.1.2 Checking History

The total number of telemetry channels received can be checked by calling:

ASSERT_TLM_SIZE(n);

The number of receipts of a particular channels can be checked by calling:

ASSERT_TLM_<channel name>_size(n);

The contents of an channel can be checked by calling:

ASSERT_TLM_<channel name>(index,<args>);

The index is used to check when more than one channel of a particular type has been received.

8.2.4 Events

8.2.4.1 Event History

8.2.4.1.1 Clearing Events

Event history can be cleared by calling:

this->clearEvents();

8.2.4.1.2 Checking History

The total number of events can be checked by calling:

ASSERT_EVENTS_SIZE(n);

The number of receipts a particular event can be checked by calling:

ASSERT_EVENTS_<event name>_size(n);

The contents of an event can be checked by calling:

ASSERT_EVENTS_<event name>(index,<args>);

The index is used to check when more than one event of a particular type has been received.

8.2.5 Parameters

Parameter values can be staged for the component under test. They should be staged prior to calling the *loadParameters()* function. The parameter functions are of the form:

void paramSet(<parameter value>,Fw::ParamValid valid);

8.3 Connecting the Test Components

The component being tested and the component under test are interconnected in the same way that components are in applications. The test component has input and output ports that are mirrors of the corresponding ports on the component being tested. They can be interconnected in the way described in section 6.8.3. A C *main()* function should be supplied as an entry point.

8.4 Testing Active Components

Passive or Queued components can be tested by calling the ports that exercise the functionality of the component since they have no thread. For Active components, normally a thread is started to empty the message queue for asynchronous ports. For unit testing, running a thread can be problematic because the component thread can stall while the thread running the test continues, making it harder to get results of code that runs on the thread. Instead, the user can follow the following procedure to empty the message queue without spawning a thread:

1) When declaring the derived implementation class (see section 6.7) for the component, declare a "friend" class for the class. Friend classes are allowed to access data and methods for a class that are not declared public.

2) Give the test component (see section 8.2) class name the same name as the friend class declared in the first step.
3) The following functions that normally execute on the thread of the component can be manually invoked from the test component:
    - preamble() – executes the preamble code of the implementation class.
    - finalizer() – executes the finalizer code of the implemention class
    - doDispatch() – A message is placed in the component message queue whenever an asynchronous input port or command is invoked. A subsequent call to *doDispatch()* retrieves that message from the message queue and invokes the handler in the implementation class.

8.5 Writing the Tests

Unit tests typically consist of using the test component to invoke enough of the component functionality such that required behavior is verified and a desired level of code coverage is achieved. This can be accomplished by writing a set of individual public methods in the test component for each test scenario. The Google Test framework (https://code.google.com/p/googletest) is distributed as part of the code repository to provide a method of executing the tests and verifying the result. To use it, add the module *gtest* to the *TEST_MODS* variable in the *test/ut/mod.mk* file as described in section 8.1.2.

8.6 Building the Unit Test

The build system supplies targets for building unit test executables. From the module directory (*not <module_dir>/test/ut*), type: "make ut". Any modules that the unit test depends on will automatically be built.

8.7 Running the Unit Test

The unit test binary is placed in a binary directory in the *test/ut* directory. For the Linux example, it would be *linux-linux-x86-debug-gnu-4.7.2-bin*. The binary can be run directly from the command line or invoked via a special build system target. One of the envisioned purposes for the standard unit test targets is to be able to run them as a part of a suite of regression tests. Some unit tests may require that test data be generated prior to execution or may require special command line arguments. For this reason, the unit test target does not directly run the binary but instead runs a script with an expected name. For Linux, this script is *<module_dir>/test/ut/runtest_LINUX*. Users write this script and do any setup and teardown necessary as well as running the binary. Since binary directory names are likely to change over time due to new compiler configurations, the build system passes the binary output directory as the first argument to the script. Users can append that argument to the unit test path to run the binary and not have to edit the script when a new compiler version is added. A template for this script is provided in *mk/make_templates/runtest_LINUX*. Once the script is added, from the module directory, type "make run_ut"

9 Configuring the Architecture

The architecture has a number of features that can be enabled or disabled depending on how resource-constrained the target environment is or what features are desired. The configuration is done by defining

C macro values that activate or disable code. Default values are defined in *Fw/Cfg/FwConfig.hpp*. Users can change these values or override them using compiler flags if different settings are desired for different deployments. Users can also use the macros in their own code if they have code that is dependent on that feature.

9.1 Supported Types

The architecture is designed to be portable to different processor architectures. Some architectures such as small microcontrollers do not support the full range of types. The architecture supports a non-sized integer type named NATIVE_INT_TYPE. NATIVE_INT_TYPE is recommended for code (e.g. loop variables) where a particular size is not needed in order to make it more portable. An additional type NATIVE_UINT_TYPE is available for unsigned variables. These types are defined in *Fw/Types/BasicTypes.hpp* and use compiler macros for sizing. If compilers other than GNU and LLVM (MacOS) are desired, additional code and macros have to be added to specify which types are available. The macros used to add or remove the different typed are as follows:

Macro	Definition
FW_HAS_64_BIT	The architecture supports 64-bit integers
FW_HAS_32_BIT	The architecture supports 32-bit integers
FW_HAS_16_BIT	The architecture supports 16-bit integers
FW_HAS_F64	The architecture supports 64-bit double-precision floating point values

9.2 Object Naming

The architecture can store names for each object created. This is useful when using object registries or tracing to see what objects exist and how they interact. The object naming does increase the per-object storage and code size, so in a resource-constrained environment disabling this feature might be desirable. This macro should be used in developer implementation classes to call the correct constructor in the code generated base classes (See section 6.7.8.1). The macros related to this feature are as follows:

Macro	Definition
FW_OBJECT_NAMES	Enables storage of the name as well as code to retrieve it.
FW_OBJ_NAME_MAX_SIZE	Sizes the buffer used to store the object name. If the size of the string passed to the code generated component base classes is larger than this size, the string will be truncated. FW_OBJECT_NAMES must be set for this to have any effect.

9.3 Object Registry

An object registry is a class that holds a list of framework component and port objects. The registry can be used to list all the objects, or call common functions on all objects. A base class for the object registry is defined in *Fw/Obj/ObjBase.hpp*, and a simple implementation can be found in *Fw/Obj/SimpleObjRegistry.hpp*. The macros to configure this feature are:

Macro	Definition
FW_OBJECT_REGISTRATION	Enables object registries.
FW_OBJ_SIMPLE_REG_ENTRIES	The size of the array in the simple object registry used to store objects. FW_OBJECT_REGISTRATION must be set.
FW_OBJ_SIMPLE_REG_BUFF_SIZE	The size of the buffer used to store object names in the simple registry. FW_OBJECT_REGISTRATION must be set.

9.4 Object to String

The framework port and object classes have an optional *toString()* method. This method by default returns the instance name of the object, but toString() is defined as a virtual method so a developer class can override this and provide custom information. The macros to configure this feature are:

Macro	Definition
FW_OBJECT_TO_STRING	Enables the *toString()* method. FW_OBJECT_NAMES must be enabled for this to have any effect.
FW_OBJ_TO_STRING_BUFFER_SIZE	Defines the buffer size used to store *toString()* results.
FW_SERIALIZABLE_TO_STRING	Defines a toString() method for code generated serializable types to display the value. See section 6.6.1.1 for serializables.
FW_SERIALIZABLE_TO_STRING_BUFFER_SIZE	Defines the size of the string that holds the toString() result for code generated serializables.

9.5 Asserts

The assert feature is described in section 7.2.1. The assert can be configured in the following ways:

<table>
<tr><th>Macro</th><th>Definition</th></tr>
<tr><td>FW_ASSERT_LEVEL</td><td>Sets the level of reporting for the asserts. Here are the values:
<table>
<tr><th>Value</th><th>Definition</th></tr>
<tr><td>FW_NO_ASSERT</td><td>Asserts turned off. The code to check the condition is not compiled. Some developers prefer this once the code has been tested to regain some processing performance.</td></tr>
<tr><td>FW_FILEID_ASSERT</td><td>An integer value for the file where the assert occurs (as opposed to __FILE__). This saves a lot of code space since no file name is stored. The make system supplies this to the</td></tr>
</table></td></tr>
</table>

		compiler by hashing the file name. A list of hash values for each compiled file can be found in *mk/hash*.
	FW_FILENAME_ASSERT	The __FILE__ macro is used in the assert to tell which file the assert occurred in.
FW_ASSERT_TEXT_SIZE	The size of the buffer used to store the text of the assert.	

9.6 Port Tracing

When components are interconnected, it is often useful to trace the set of invocations through components and ports. The port base class has a *trace()* call that is invoked by the derived port classes whenever the port is invoked. The *trace()* call calls *Os::Log::logMsg()* with the name of the port. Calling the port base class method *setTrace()* turns global tracing on and off. Individual ports can be have tracing turned on and off by calling the *overrideTrace()* method on the port instance. The macros to configure this feature are:

Macro	Definition
FW_PORT_TRACING	Enable port tracing.

9.7 Port Serialization

As discussed in the architectural document, a port type (*Input/OutputSerializePort*) exists that has no interface type, but instead receives (or sends) a serialized form of the port invocation for the attached port. The primary pattern for this is to invoke components on remote nodes. The code generator generates code in each port that will serialize or deserialize the invocation if it detects that it is connected to a serializing port. If development is for a single node, this feature can be disabled to reduce the code size. The macros to configure this feature are:

Macro	Definition
FW_PORT_SERIALIZATION	Enables port serialization

9.8 Serializable Type ID

As described in Section 6.6.1, serializable types can be defined for use in the code. When objects of those types are serialized, an integer representing the type ID can be serialized along with the object data. This allows the type to be determined later if only the serialized form is available. Turning off this feature will lower the amount of data moved around for a given object when it is serialized. The macros to configure this feature are:

Macro	Definition
FW_SERIALIZATION_TYPE_ID	Enables serializing the type ID. If this is not defined, the type ID will not be stored when serializing.
FW_SERIALIZATION_TYPE_ID_BYTES	Defines the size of the serialization ID. Fewer bytes means that less data storage is needed, but also limits the

	number of types that can be defined.

9.9 Queue Name

The Os::Queue class stores a queue name as private data. The macros to configure this feature are:

Macro	Definition
FW_QUEUE_NAME_MAX_SIZE	Defines the size of the string buffer that stores the queue name.

9.10 Task Name

The Os::Task class stores a task name as private data. The macros to configure this feature are:

Macro	Definition
FW_TASK_NAME_MAX_SIZE	Defines the size of the string buffer that stores the task name.

9.11 Command Buffers

The arguments for each command are serialized into a buffer which is passed to the components for deserializing. The buffer must be large enough to serialize the arguments for the largest command in the system. If the command argument buffer is too small, the code will assert when attempting to store the argument data. Commands can optionally have a string argument which is specified to be a particular number of characters. A buffer is defined to store a maximum size of string arguments across the system. The macros to configure these features are:

Macro	Definition
FW_CMD_ARG_BUFFER_MAX_SIZE	Defines the size of the command argument buffer.
FW_CMD_STRING_MAX_SIZE	Defines the maximum size of a string argument in the system. Cannot be greater than FW_CMD_ARG_BUFFER_MAX_SIZE.

9.12 Telemetry Buffers

Telemetry values are serialized by the component into a buffer and sent via an output telemetry port to another component for storage. The buffer must be large enough to serialize the value of the largest telemetry type in the system. Telemetry channels can optionally have a string value which is specified to be a particular number of characters. A buffer is defined to store a maximum size of string telemetry values across the system. The macros to configure these features are:

Macro	Definition
FW_TLM_BUFFER_MAX_SIZE	Defines the size of the telemetry buffer.
FW_TLM_STRING_MAX_SIZE	Defines the maximum size of a telemetry string value in the system. Cannot be greater than FW_TLM_BUFFER_MAX_SIZE.

9.13 Parameter Buffers

Parameter values are deserialized by the component from a buffer received from a parameter management component. The buffer must be large enough to serialize the value of the largest parameter type in the system. Parameters can optionally be a string value which is specified to be a particular number of characters. A buffer is defined to store a maximum size of string parameter values across the system. The macros to configure these features are:

Macro	Definition
FW_PRM_BUFFER_MAX_SIZE	Defines the size of the parameter buffer.
FW_PRM_STRING_MAX_SIZE	Defines the maximum size of a parameter string value in the system. Cannot be greater than FW_PRM_BUFFER_MAX_SIZE.

9.14 Logging Buffers

The arguments for each event are serialized into a buffer which is created by the components and sent via a logging port to the component handling events. The buffer must be large enough to serialize the arguments for the largest event in the system. If the event argument buffer is too small, the code will assert when attempting to store the argument data. Events can optionally have a string argument which is specified to be a particular number of characters. A buffer is defined to store a maximum size of string arguments across the system. The macros to configure these features are:

Macro	Definition
FW_LOG_BUFFER_MAX_SIZE	Defines the size of the log buffer.
FW_LOG_STRING_MAX_SIZE	Defines the maximum size of a log string value in the system. Cannot be greater than FW_LOG_BUFFER_MAX_SIZE.

9.15 Text Logging

Event functions that are called are turned into two output port calls. One is a binary port that is used to store the event to be transported to ground software or a testing interface external to the software. The component also takes the format string specified in the XML and populates it with the event arguments, and calls an output port with a readable text version of the event. This is meant to be used for a console interface so the user can see, in text form, the same events being stored for transmission. A component with the text logging input port can be used to display the text. A very simple implementation of this can be seen in *Svc/PassiveConsoleTextLogger*. In a resource-constrained environment or in a flight implementation where the console is not viewable, the text formatting and extra code can consume an undesirable number of processor cycles. For this reason, the text logging can be turned off via a macro. This compiles out the code and format strings for text logging. The macros to configure text logging are as follows:

Macro	Definition

FW_ENABLE_TEXT_LOGGING	Enables or disables text logging.
FW_LOG_TEXT_BUFFER_SIZE	Defines the maximum size of a text string representation of the event.

9.16 Ground Interface Tuning

Different ground systems require different sizes of parameters stored in the downlink packets. Macros defined in *FwConfig.hpp* allow different projects to change the size of the parameters to suite the ground system. These types are defined via macro instead of using *typdef* so that the compiler can override it on a target/deployment basis. The types that can be modified are as follows:

Type	Description
FwPacketDescriptorType	The first field in a downlink or uplink packet indicates what packet type is being encoded. (See Fw/ComPacket.hpp) This sets the size of that field.
FwOpcodeType	This type defines the storage size for command opcodes
FwChanIdType	This type defines the storage size for telemetry channel IDs
FwEventIdType	This type defines the storage size for event IDs
FwPrmIdType	This type defines the storage size for parameter IDs
FwBuffSizeType	This type defines the storage size for the field that indicates how many bytes are stored for a buffer type like a string
FwEnumStoreType	This type defines the storage size for enumerations defined in XML declarations for components.

9.17 Time Base

The F` time tags have a field that specifies the time base of the time tag. A time base is defined as a clock in the system correlated with a known epoch. It is often the case that when a system is being initialized it does not always have access to a clock correlated to surface operations. It can transition through several time bases (processor, radio, Earth) on the way to becoming fully operational. The *TimeBase* type defines the set of clocks in the system that can produce a time tag. It lets users of the system see which clock was used when time tagging telemetry.

10 Modeling

Much of the XML can be generated via a MagicDraw plug-in that takes SysML models and emits the XML. The modeling is a topic that will be documented in the future but at the writing of the version of the User's Guide help from the ISF team is required.