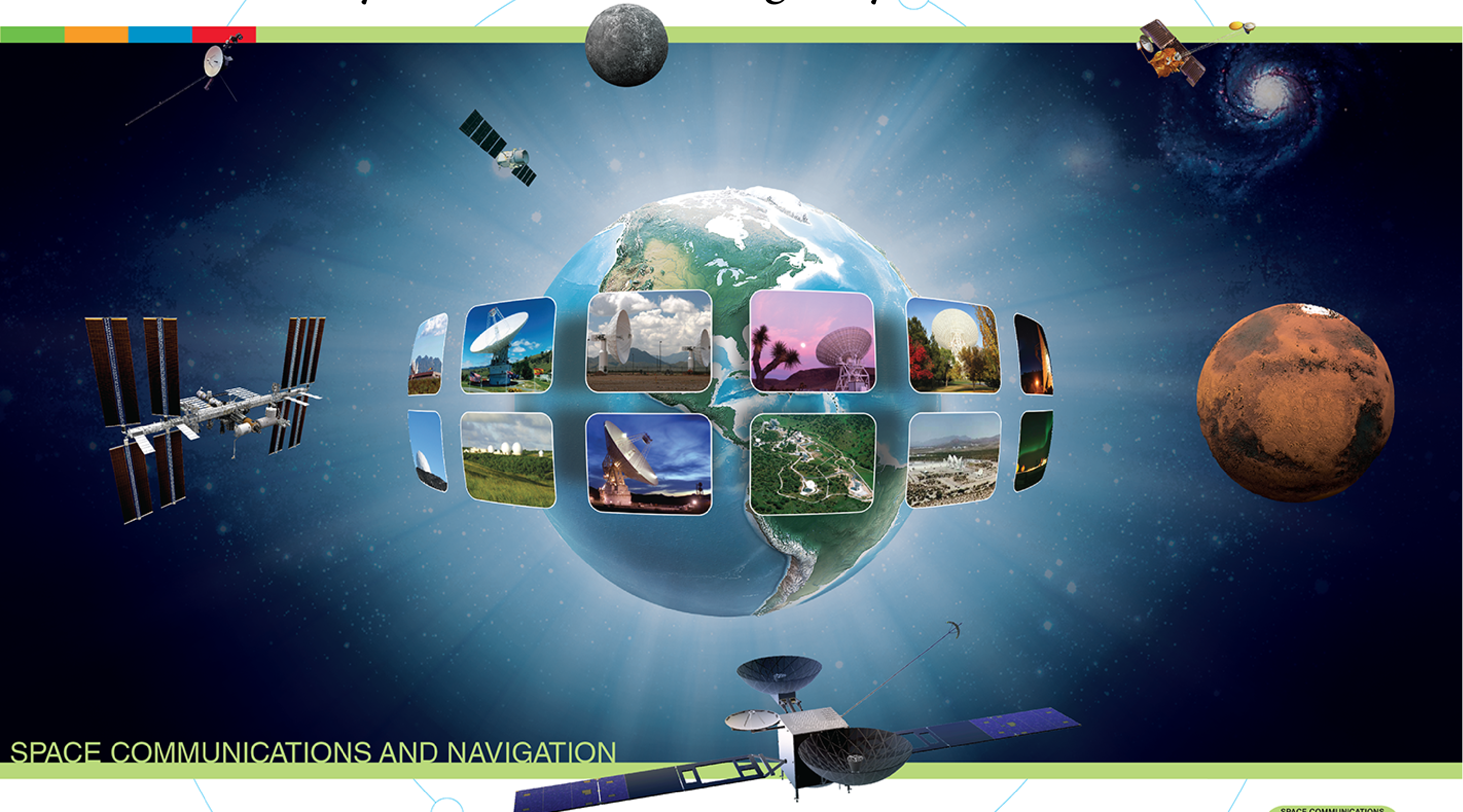
Future Standardization of Space Telecommunications Radio System with Core Flight System

National Aeronautics and Space Administration



Joseph P. Hickey (ZIN Technologies, Inc.); Janette C. Briones, Rigoberto Roche, Louis M. Handler, and Steven Hall (NASA Glenn Research Center)

www.nasa.gov

SPACE COMMUNICATIONS SCaN AND NAVIGATION

Introduction

This project aimed to provide interoperability between two existing NASA software architectures:

The Core Flight System (cFS)

and

Space Telecommunications Radio System (STRS)

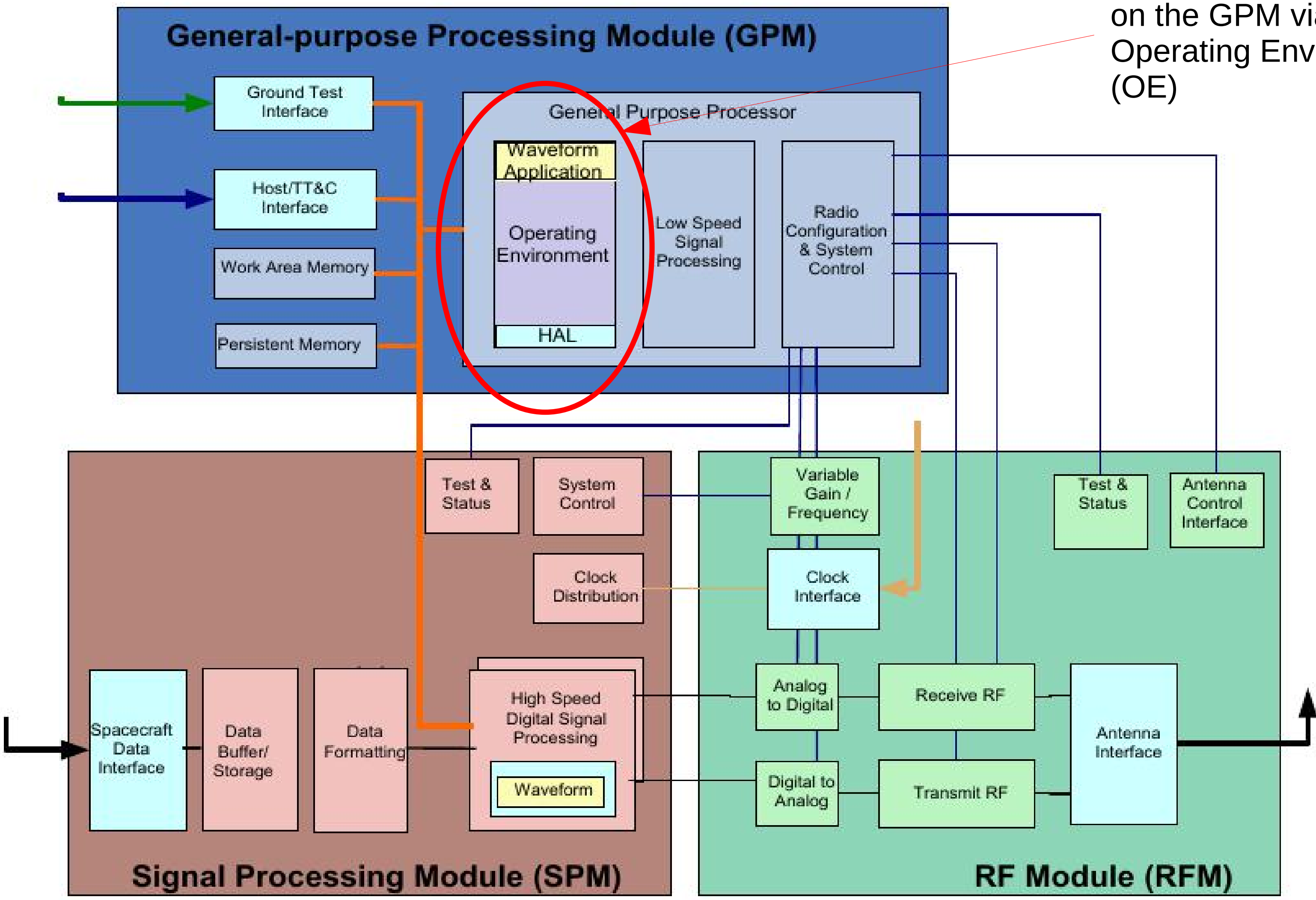
STRS: What Is It?

Space Telecommunications Radio System

- An architecture for Software Defined Radios (SDR)
  - A conventional radio has modulation/demodulation and processing logic built into its hardware design
  - An SDR shifts most of the logic into software & FPGAs
- SDR’s are *highly reconfigurable*
  - Accommodates advances in technology
  - Modulation techniques can be adapted on-the-fly
  - Enables cognitive radio concepts

SDRs are commonplace in commercial and military industries.

STRS: SDR Design



SDR applications execute on the GPM via the STRS Operating Environment (OE)

STRS: What It Defines

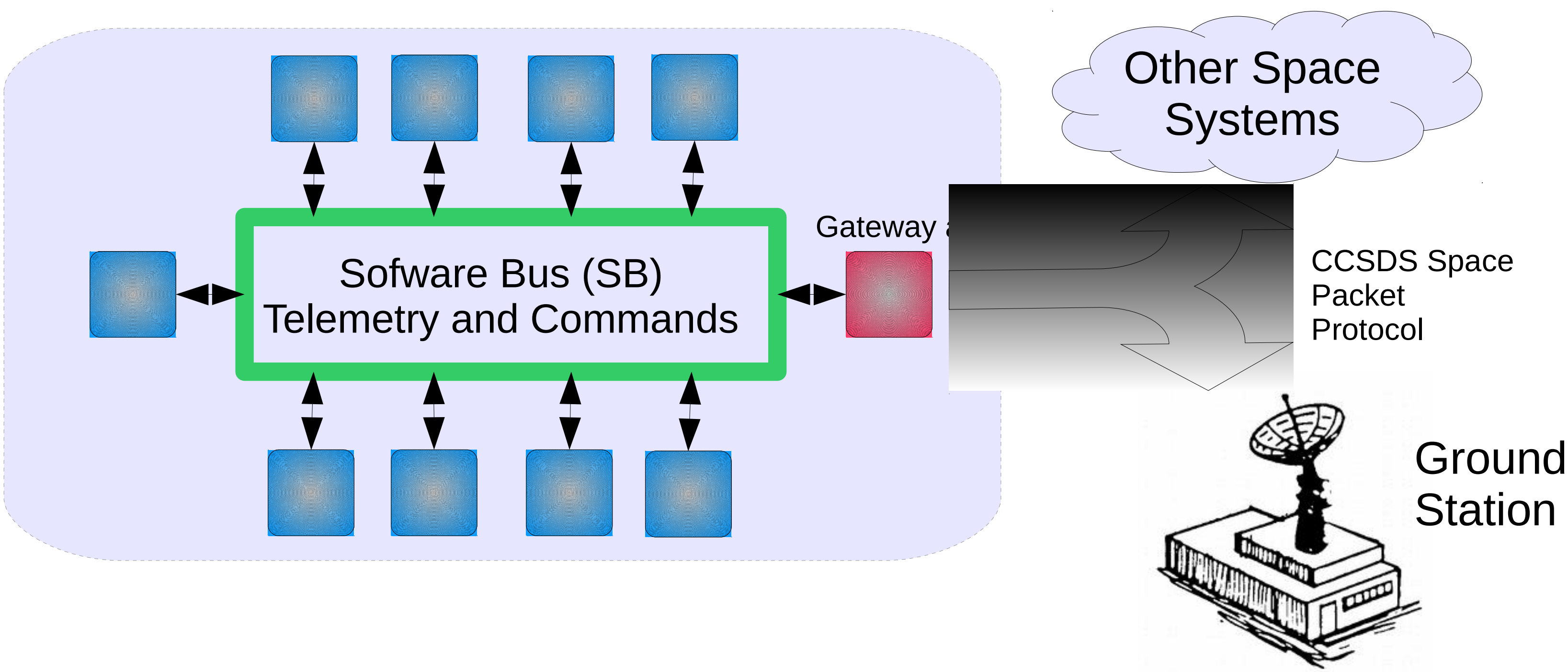
STRS defines an API for initialization, configuration, and data exchange between SDR components:

- Allows encapsulation of functionality.
- Allows multiple vendors to work on different parts of the radio at once
- Allows updates to one part not to affect the other parts of the radio
- Allows portability of SDR logic: Software design and implementation processes may be leveraged to lower risk and increase reliability

Publicly published and released in **NASA-STD-4009**
http://strs.grc.nasa.gov

cFS: What Is It?

cFS is a general-purpose flight software framework based on a collection of modular *"apps"* that primarily communicate using a message passing architecture called the *"Software Bus"*



- The Software Bus may be extended to exchange commands and telemetry with other systems/processors, which may or may not be based on cFS.
- CCSDS standard 133.0-B-1 space packet protocol (with secondary command/telemetry header) is used for all messages, internal and external.

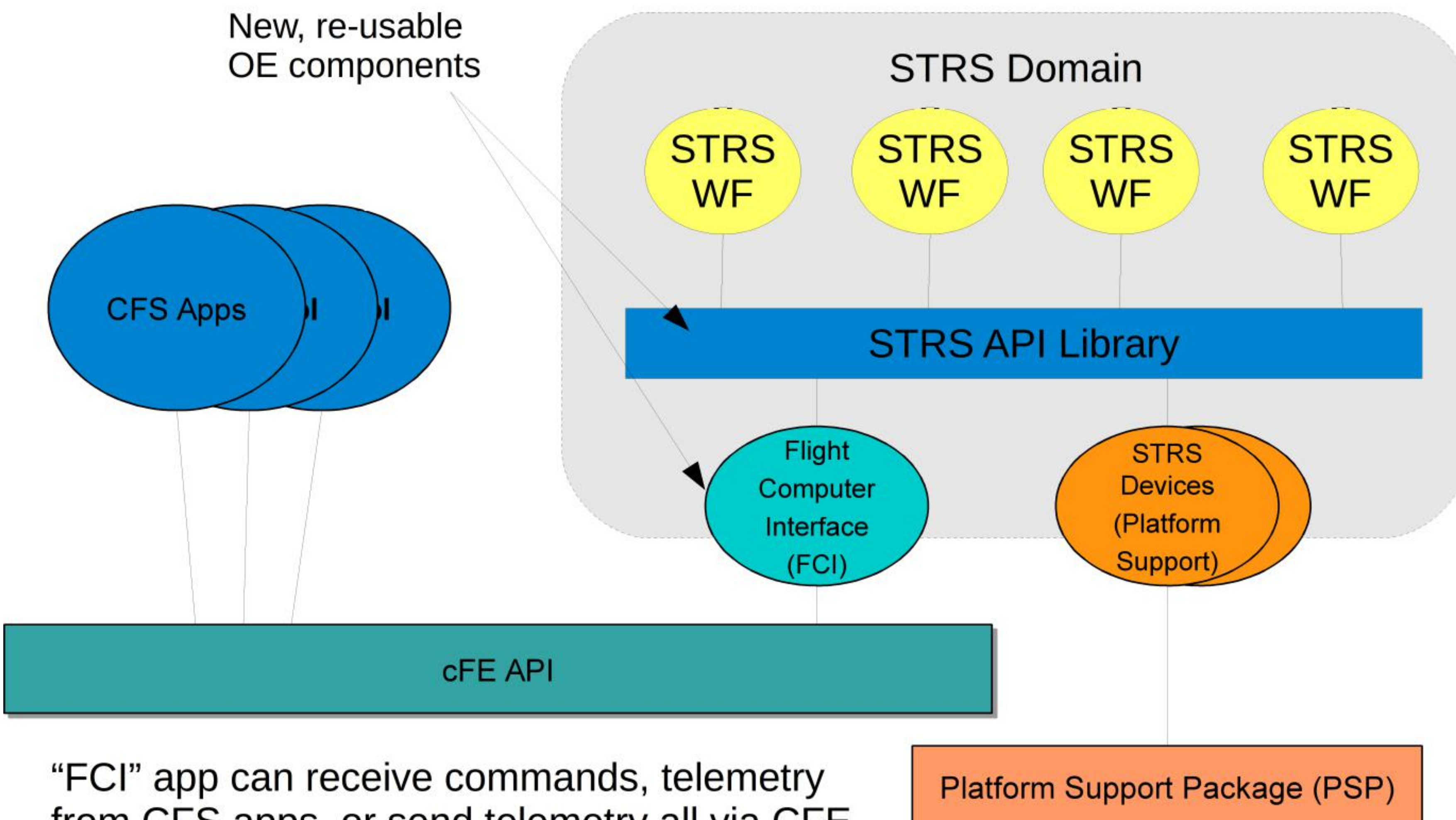
cFS + STRS...

Both technologies have considerable investment:

- CFS is used across NASA for flight software:
  - Many missions, past & future: Morpheus, LADEE, GPM, RBSP, MMS, LRO, Orion, EVA, GHAPS, etc.
  - Many cFS compatible apps have been developed
- STRS Waveform Repository:
  - Contains multiple reusable waveforms
  - OE for JPL, Harris SDRs

It is desirable to leverage both sets of existing applications and have them inter-operate

Putting Them Together

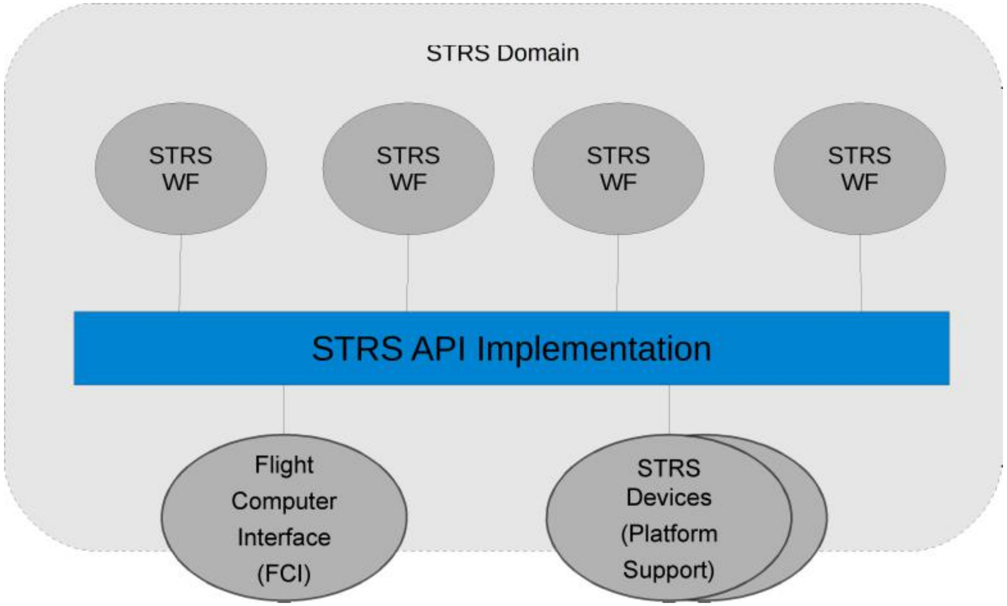


"FCI" app can receive commands, telemetry from CFS apps, or send telemetry all via CFE Software Bus

STRS API Library

New “clean room” implementation of STRS API

- Provides C implementation for the STRS-defined API calls
  - Minimal “smarts” – only a dispatcher to other entities that must be defined outside the library. Nothing CFS-specific.
- Provides STRS defined headers:
  - STRS.h
  - STRS_APIs.h
  - STRS_ApplicationControl.h
  - STRS_ComponentIdentifier.h
  - STRS_ControllableComponent.h
  - STRS_LifeCycle.h
  - STRS_PropertySet.h
  - STRS_Sink.h
  - STRS_Source.h
  - STRS_TestableObject.h
  - STRS_Device.h

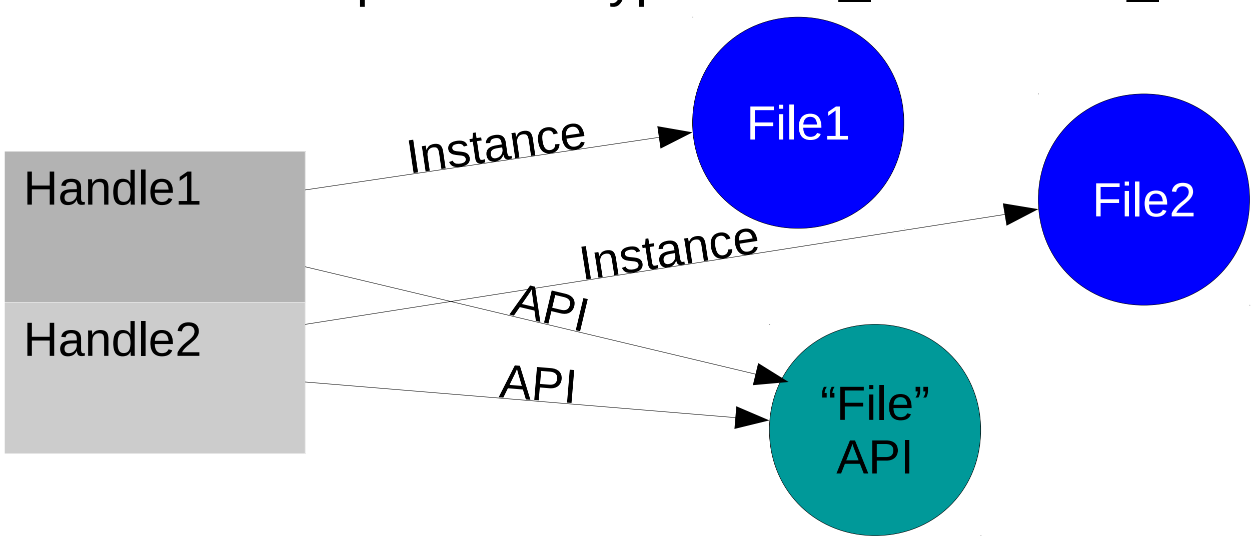


STRS Handle Management

Objective: The OE library manages a global lookup table for all STRS handle IDs.

Internal table contains:

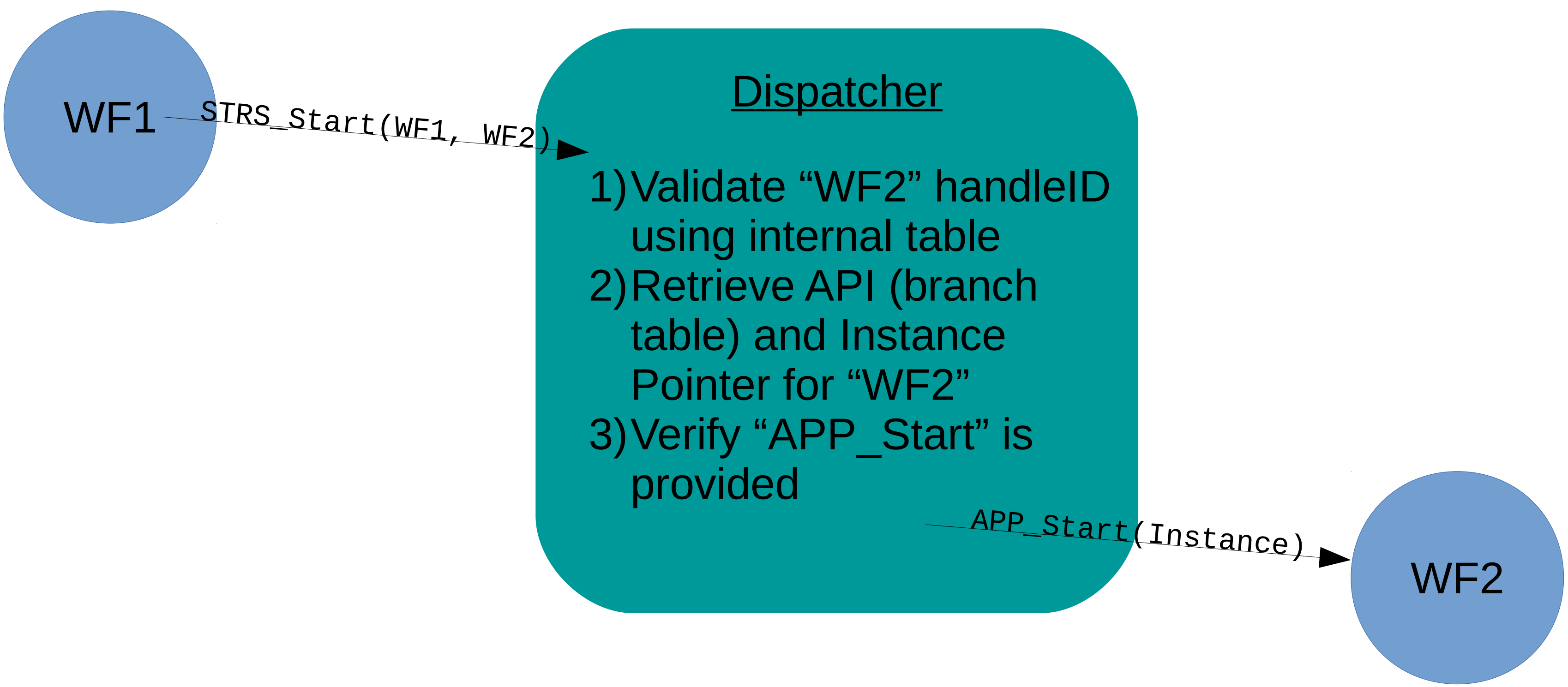
- STRS API validity mask: Which STRS API calls are allowed.
  - This restricts from calling e.g. `STRS_MessageQueueDelete()` on a non-queue object, or `STRS_FileClose()` on a queue, etc.
- Pointer of type `STRS_API_t*` to API structure or "Branch Table" containing specific implementations of APP API calls for the object.
- Object-specific instance pointer of type `STRS_Instance_t*`



STRS Dispatcher

Objective: Provide core "dispatcher" functions for STRS APIs.

- All core functions are implemented in pure C (like other CFS libraries)
- Uses "branch table" approach to servicing STRS API calls
  - All handles are equal, **no special treatment of any ID**.
  - Any special behavior is in the *implementation,* not in the dispatcher.



STRS Files and Queues

Objective: Provide suitable implementations for “File” and “Queue” functions

File and Queue operations loosely map to existing APP API calls:

- “APP_Instance” can create a wrapper object
- “APP_Initialize” can obtain the underlying resources (filehandle, etc)
- “APP_ReleaseObject” can release the resource
- “APP_Read” and “APP_Write” serve the normal purpose

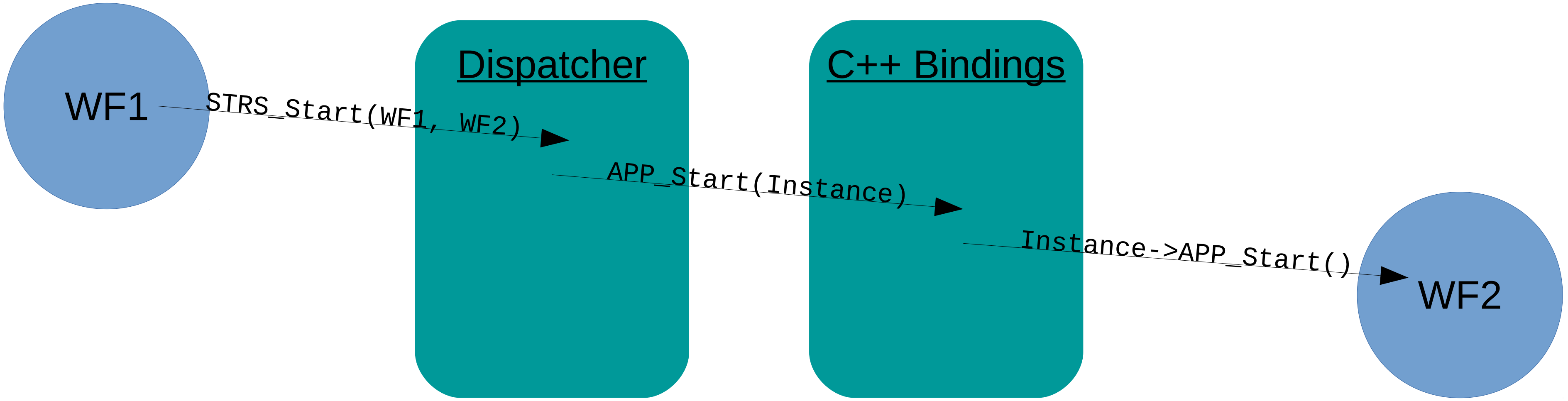
Any unique properties of special handles can be embedded entirely within the underlying implementation functions:

- `STRS_FileOpen()` creates an STRS handle using the File API
- `STRS_MessageQueueOpen()` creates an STRS handle using the Queue API
- The “Validity Mask” implemented in the OE ensures that a user cannot directly call other STRS APIs on these types of handles, such as `STRS_Initialize()`, even though it may implement the APP call.

STRS API Implementation

Objective: Transparently support dispatching to waveforms implemented in C++ as well as C

- C++ bindings are provided using the same branch table
- A C++ class provides compatible (`extern "C"`) implementations of the C API, which in turn calls the C++ member function
- Dispatcher doesn't know the difference, nothing special is done
- Fully portable; nothing compiler specific, minimal `#ifdef` conditional compilation, and all C++ calling conventions are correctly adhered to.
- C++ is easily removed for targets that do not have C++ runtime libraries



Flight Computer Interface (FCI)

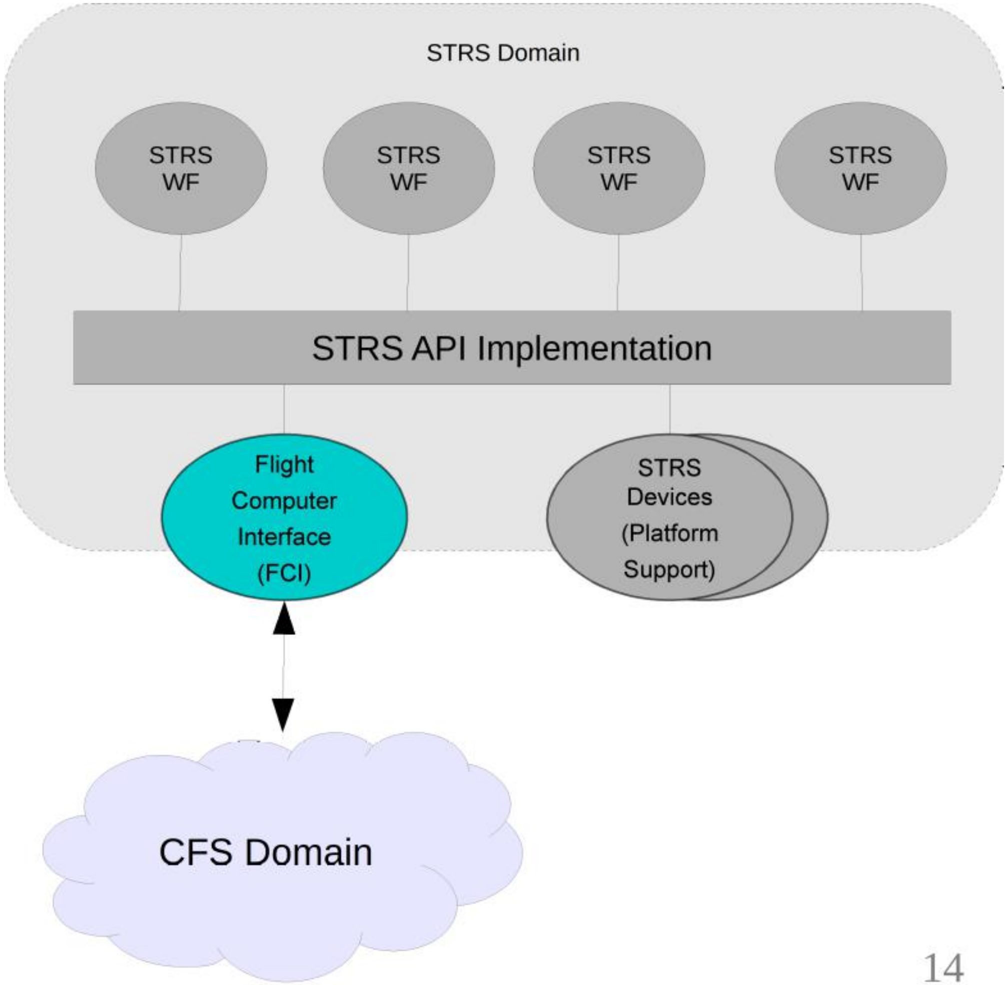
Dual personality: STRS and cFS application

On the cFS side:

- Has its own thread
- It can subscribe to anything on the CFE software bus.
- It can broadcast to the CFE software bus

On the STRS side:

- Instantiates required handle IDs:
  - STRS_ERROR_QUEUE
  - STRS_FATAL_QUEUE
  - STRS_WARNING_QUEUE
  - STRS_TELEMETRY_QUEUE
- Can make STRS calls
- Permits STRS applications to send or receive CFS software bus messages through STRS API



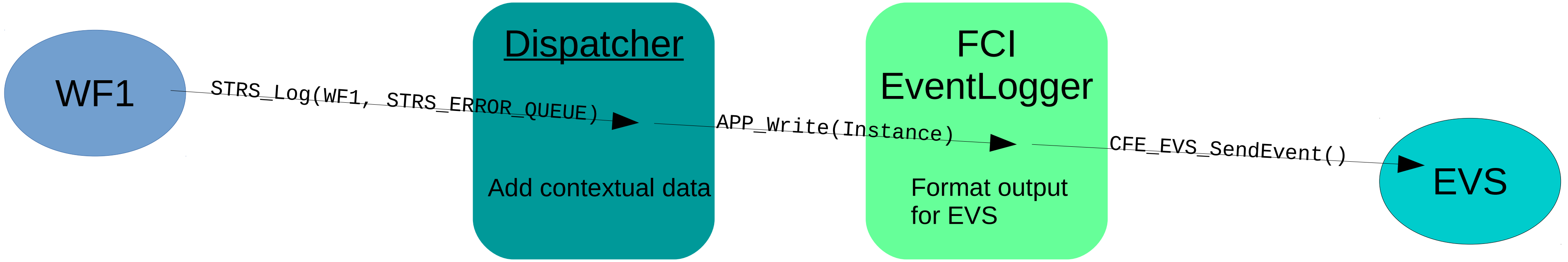
FCI: Log Queues

Objective: "Flight Computer Interface" (FCI) instantiates all required log objects within the OE:

- `STRS_ERROR_QUEUE`
- `STRS_FATAL_QUEUE`
- `STRS_WARNING_QUEUE`
- `STRS_TELEMETRY_QUEUE`

These handles all utilize an "EventLogger" API implemented within FCI.

- Only `STRS_Log()` is allowed on these handles (direct STRS_Write is restricted)
- Implementation of `APP_Write()` forwards the event message and contextual data to the CFE Event Services (EVS) subsystem
- Each STRS handle maps to a different CFE Event ID so each type of message can be identified in the resulting telemetry stream



FCI: Time Handles

Objective: "Flight Computer Interface" (FCI) instantiates an STRS handle to access the CFE "Mission Elapsed Time" (MET)

- "MET" is a monotonic clock provided by the CFE TIME subsystem.
  - This clock may be correlated with other clocks, such as UTC/earth time, using a "spacecraft time correlation factor" (STCF).
- This provides basis for `STRS_GetTime()` and `STRS_SetTime()`
  - STRS defines API calls only; it does not stipulate any particular clocks that must exist or how they operate
    - OE specifies the actual clocks and the handle name(s) it provides
  - MET access is provided via a normal STRS HandleID
    - STRS_GetTime() implemented as APP_Read()
    - STRS_SetTime() implemented as APP_Write()
    - Direct STRS_Read() / STRS_Write() on this handle are restricted

FCI: Software Bus Interface

Objective: FCI allows apps within the STRS domain to interact with cFS applications or vice versa

It is a common paradigm for cFS applications to accept application-defined commands sent from remote sources.

- FCI allows STRS API calls to be made using an interface that "looks and feels" like other CFS commands.
  - Allows use of existing CFS command generation tools to issue STRS API calls, including the web-based GUI.
  - Remote cFS apps are "just another STRS handle"
- Optional component; this feature could be easily removed if this functionality is not desired.

Proof-of-Concept: ASRP

- The Advanced Space Radio Platform (ASRP) is the incubator for the cFS + STRS combination.
  - Based on the Vadatech AMC516 hardware
  - cFS runs on the PowerPC host processor
- Multiple STRS devices implemented:
  - Local Bus (including FPGA loading and register access)
  - M-LVDS cross bar switch
  - Quad PLL

ASRP Waveforms

The following STRS waveforms are implemented on ASRP:

- Live sample capture
  - Configurable sized capture from live radio data
  - Implemented as simple STRS_Read / APP_Read call
- Spectral power density estimation
  - Implements P. Welch algorithm
    - Configurable number of segments, segment overlap, segment window function
    - Uses FFTW library on PowerPC for FFTs
  - Reads raw data via STRS_Read() from live sample capture.
- Web-based GUI for interactive use

Questions

Issues

- Synchronous vs. Asynchronous calls
  - Most STRS calls are synchronous
    - For instance, the data buffer on STRS_Read() is expected to be filled with valid data when the call returns
  - Most CFS operations are asynchronous
    - Sends a message on the software bus
    - “Fire and forget” – no replies
- Software Bus vs. STRS Pub/Sub
  - Although the CFS software bus is a publish/subscribe model, it requires all endpoints to be defined at compile time for subscription purposes.
  - STRS allows creation and deletion of endpoints at run time