cFS Basecamp Application Developer’s Guide

Version 2.2

January 2025

Document Audience and Prerequisites

- **Objectives**
  - Describe how to develop apps using cFS Basecamp’s Application C Framework (APP_C_FW)
  - Basecamp apps use object-based design patterns and deviate from some of the design patterns in the cFS Application Developer’s Guide **cFE/docs/cFE Application Developers Guide.md at main · nasa/cFE**. However they adhere to the cFS APIs and are portable across cFS instantiations.

- **Intended Audience**
  - Software developers (typically flight software) that want to understand and/or develop Basecamp style cFS applications

- **Prerequisites**
  - Basic understanding of the cFS architecture. See Basecamp’s cFS Overview and cFS Framework documents
  - Familiarity with the cFS Application Developer’s Guide is helpful but not required
  - C programming experience
  - Access to a Linux platform with an internet connection, if you want to do the “Hello World” exercises

Read Before You Proceed!!

- Basecamp app designs do not follow all NASA app design conventions described in the *NASA Application Developer’s Guide*

- Basecamp serves as a cFS educational platform and its app design provides functionality required by the educational work flows
    - For example, users can create new cFS projects targets by selecting and adding apps from a github “app store”

- Why learn the “Basecamp way”?
    - Basecamp’s application framework design is based on sound software engineering principles
    - Having a second perspective on designing apps encourages developers to think about their designs with rationale for their decisions

- Basecamp is a testament to the fact that the cFS Framework provides an API and does not dictate application designs

Basecamp App Framework & Design Patterns

- **Basecamp Apps use an object-based design using an Application C Framework (APP_C_FW)**
- **Telecommands and telemetry messages are defined in Electronic Data Sheets**
  - Includes constants and types
- **JSON files are used for tables**
  - APP_C_FW supports table management and uses FreeRTOS's coreJSON parser
  - Each app writes it's own table load/dump functions
- **All apps contain a JSON initialization parameter file**
  - Contains app deployment configurations that can be resolved at runtime
  - C header files only contain parameters that must be defined during compilation
  - Build tools can modify confgurations such as message IDs that facilitates Basecamp's app plu 'n play model
- **Reset command functionality is app specific and goes beyond counters**
  - Reset state should be contained in routine telemetry for verification

Some Preliminary Systems Thoughts…

- This document explains the mechanics of designing and implementing a single app

- Designing an app suite for a cFS target is a systems engineering endeavor because the apps typically provide functionality for multiple subsystems (communcaitons, payload, etc.)

- System app design concepts are introduced in Basecamp’s cFS Framework document and fully addressed in Basecamp’s Systems Engineering course

Document Approach

- This type of document is challenging because you often need to know multiple pieces of information in parallel, but not in depth, and then need to spiral through the topics again going into more detail

- The *"Hello App" coding templates* and *app_c_demo* sections are intended to help with this situation by introducing concepts without too much detail so the following sections can go into much more detail

- Note NASA's File Manager app FM was redesigned using Basecamp's framework-based design to create Basecamp's FILE_MGR app. FILE_MGR's design is described in its documentation and it may be helpful readers that are familiar with NASA's app design approach to understand the APP_C_FW approach

This is a work in progress and not all sections are complete
The 🚧 symbol is used to indicate a work in progress

Outline

1. Hello App Designs
2. Demo App Overview
3. Electronic Data Sheets
4. Framework-based App Design
5. Demo App Detailed Design
6. Design Patterns

Appendix A: Design Notation

Appendix B: Coding Conventions

Section 1

Hello App Designs

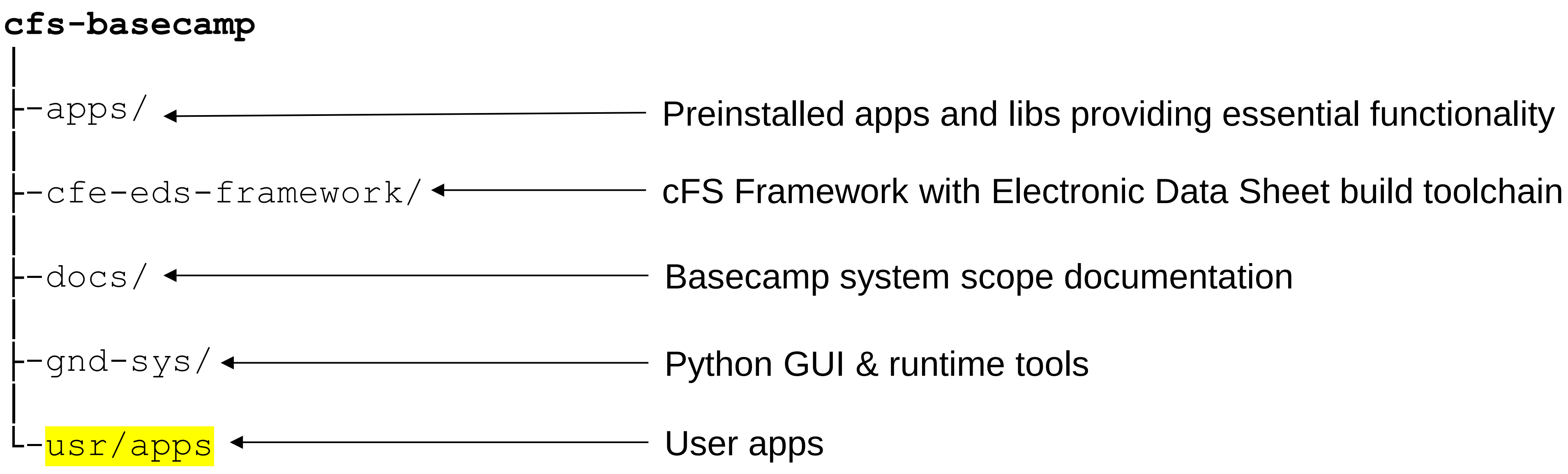
Introduction

- **Basecamp includes a series of “Hello Apps” that are generated from templates**

- **Each app introduces an app design feature and includes hands-on coding exercises**

- **This section provides design information that supplements the “Hello App” coding templates**
  - This keeps all of the app design information in a single document and each coding template document contains information that helps with the coding exercises
  - Design and coding concepts introduced in this section are explained in greater detail in later sections

Hello Apps Source Code Directory

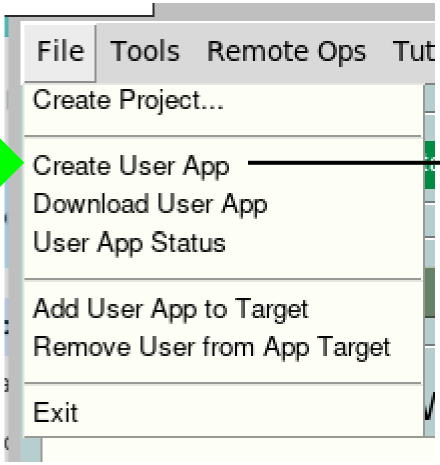


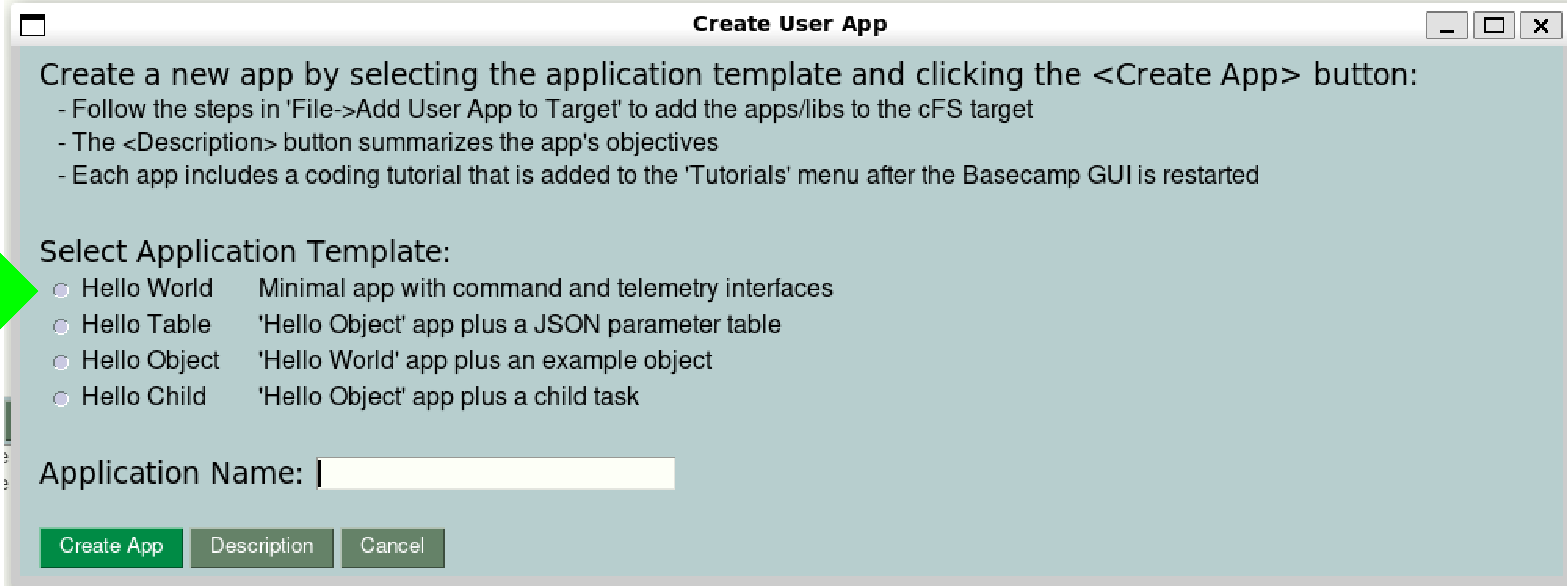
```
cfs-basecamp
|
├-apps/                  ← Preinstalled apps and libs providing essential functionality
|
├-cfe-eds-framework/     ← cFS Framework with Electronic Data Sheet build toolchain
|
├-docs/                  ← Basecamp system scope documentation
|
├-gnd-sys/               ← Python GUI & runtime tools
|
└-usr/apps               ← User apps
```

A new directory in **cfs-basecamp/usr/apps** is created for each Hello App’s source code

Creating “Hello Apps”

- **Select “Create App” in the File dropdown menu to access the ”Hello App” templates**





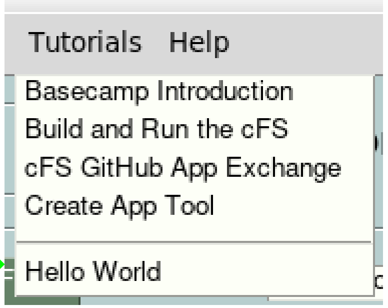
Create a new app by selecting the application template and clicking the <Create App> button:
- Follow the steps in 'File->Add User App to Target' to add the apps/libs to the cFS target
- The <Description> button summarizes the app's objectives
- Each app includes a coding tutorial that is added to the 'Tutorials' menu after the Basecamp GUI is restarted

Select Application Template:

○	Hello World	Minimal app with command and telemetry interfaces
○	Hello Table	'Hello Object' app plus a JSON parameter table
○	Hello Object	'Hello World' app plus an example object
○	Hello Child	'Hello Object' app plus a child task

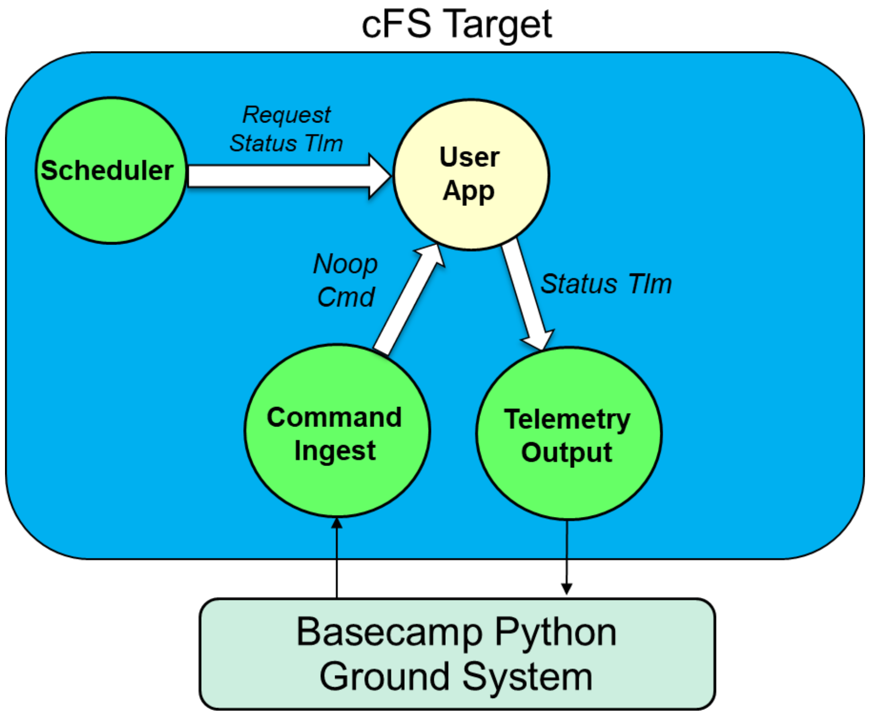
Application Name:

- **After an app is created and the Python GUI is restarted the coding tutorial will be listed in bottom section of the Tutorials dropdown menu**



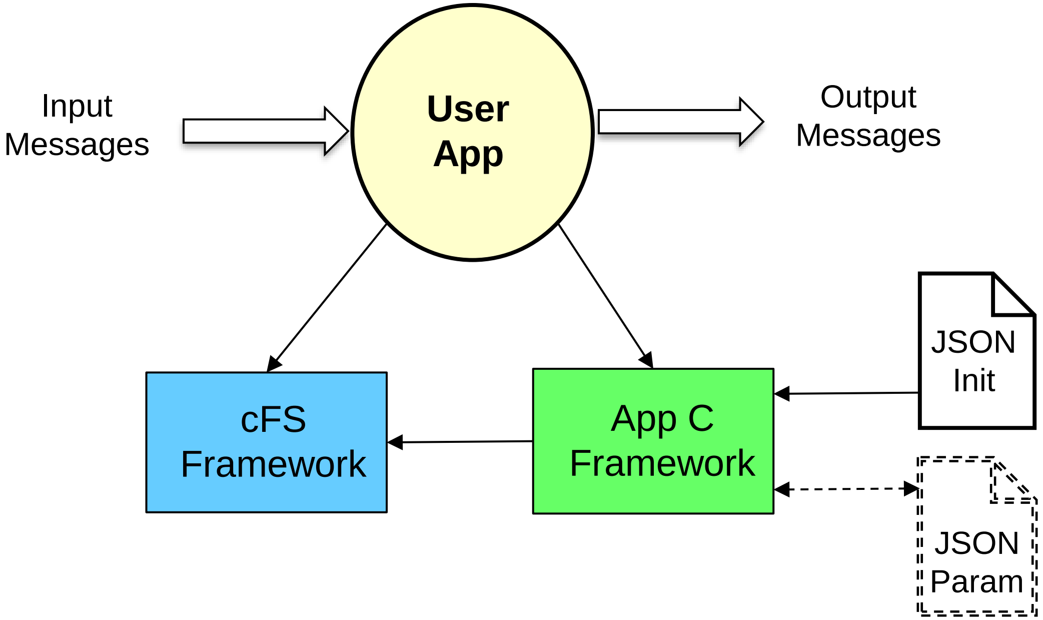
Application Runtime Context

The following diagram is from the cFS Framework document and shows an app's context with respect to the ops service app suite



- **The next slide describes the user app's context to serve as a starting point for developing apps**
- **Since the cFS term "housekeeping" is not descriptive for new developers it has been replaced with "status"**
  - In addition, many apps such as a controller that execute at a fixed frequency output state information at that frequency and don't reply on a separate "request status telemetry" message

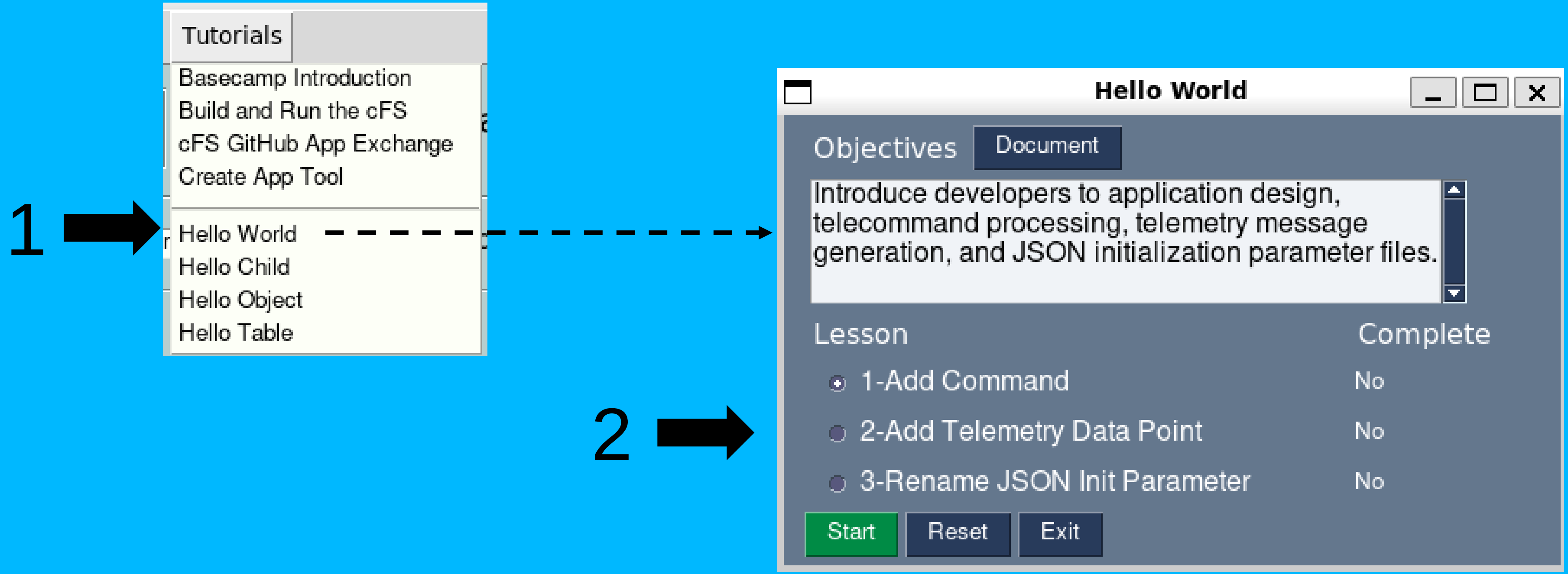
Application Context



- **This is the cFS Basecamp application context**
  - Apps may have additional interfaces such as an app-specific library
  - When developing apps it’s good practice to draw the app’s detailed context to clearly define and understand it’s interfaces
- **Input and output messages can either be command or telemetry messages**
  - This flexibility allows apps to work in groups to provide mission functionality
- **Apps make function calls to the cFS Framework APIs**
- **Apps make function calls to Basecamp’s App C Framework (APP_C_FW) API**
- **Every app has a JSON initialization parameter file and optionally one or more JSON parameter files**
  - JSON files are managed and parsed using APP_C_FW services

cFS Basecamp Exercises: Hello World

1. Read through this section for a basic understanding of the Hello World app
2. From the Tutorial dropdown list select “Hello World” and do the Lessons
    - Refer back to these slides as needed to deepen your understanding



Introduce developers to application design, telecommand processing, telemetry message generation, and JSON initialization parameter files.

Lesson	Complete
● 1-Add Command	No
○ 2-Add Telemetry Data Point	No
○ 3-Rename JSON Init Parameter	No

Minimal App Functionality

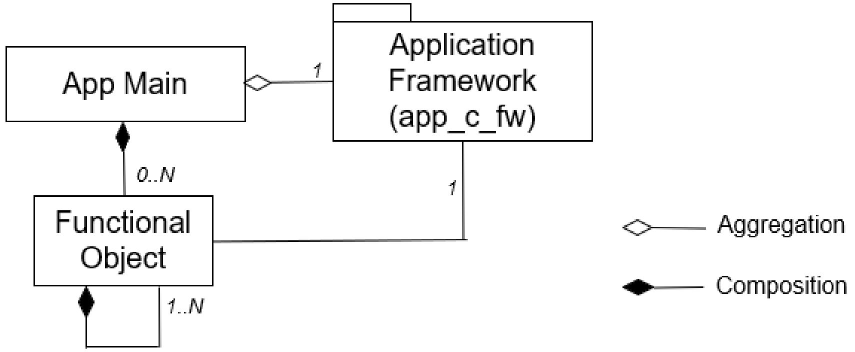
- **Basecamp's "Hello World" app template implements the minimal functionality required by an app**
  - Create a Software Bus "Pipe" and register to receive messages
  - Accept command messages and execute command-specific functions
  - Output status telemetry
- **The following functionality and design/code patterns evolved based on NASA/Goddard's experience with Low Earth Orbit (LEO) satellites**
  - If the app successfully initializes, send an event message identifying the app version
    - Provide evidence that each app has successfully started and it's the expected version
  - Provide command valid and command invalid counters in periodic status telemetry
    - Allows the ground operators to confirm that a command was received and processed with either a successful or unsuccessful outcome
  - Send a "housekeeping" (i.e.status) telemetry message at a constant periodic rate
    - Housekeeping is a NASA/Goddard colloquial term. From a telemetry message perspective it means status. From a periodic execution perspective it's time when an app can do "housekeeping chores" like check if a new table is available.
    - Includes command counters so they can be checked after sending a command
  - Provide a "No Operation (NOOP)" command that increments the command valid counter and sends an event message containing the app version
    - Allows the ground operators to confirm the communication path to an app is operational and that the app is functioning properly
  - Provide a "Reset Counters" command that clears the command counters

Basecamp Minimal App Functionality

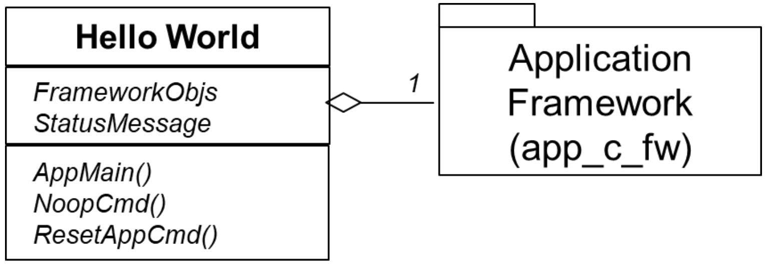
- **Basecamp apps include the NASA/Goddard design patterns with a few additions**
- **Basecamp apps use Basecamp's Application C Framework (APP_C_FW)**
  - Provides application services and utilities that support object-based designs
  - Developers can focus on developing app functional objects
- **Define command and telemetry messages using Electronic Data Sheets**
- **Use a JSON initialization parameter file that defines deployment configurations**
  - Many mission and platform configurations traditionally defined in C header files are defined in this initialization file
  - Read during an app's initialization
- **APP_C_FW Command Manager**
  - Apps register each object's command functions with the Command Manager
  - When a command message is received, Command Manager peroms validation and calls the corresponding command function
- **The Reset Counter command is called a Reset App and has a broader scope than just resetting counters**
  - The Reset App command results in an app's status being reset to an app-specific default state
  - Each object within an app provides a reset function that is called
  - If a status item is affected by the reset command then it should be included in a periodic telemetry message so the new status can be verified

Application Design Overview

- **This is a brief introduction into Basecamp's app design framework to provide context for the coding exercises**
  - Complete detailed design descriptions are provide in later sections
- **Here's a Unified Modeling Language (UML) representation of an app's architecture**



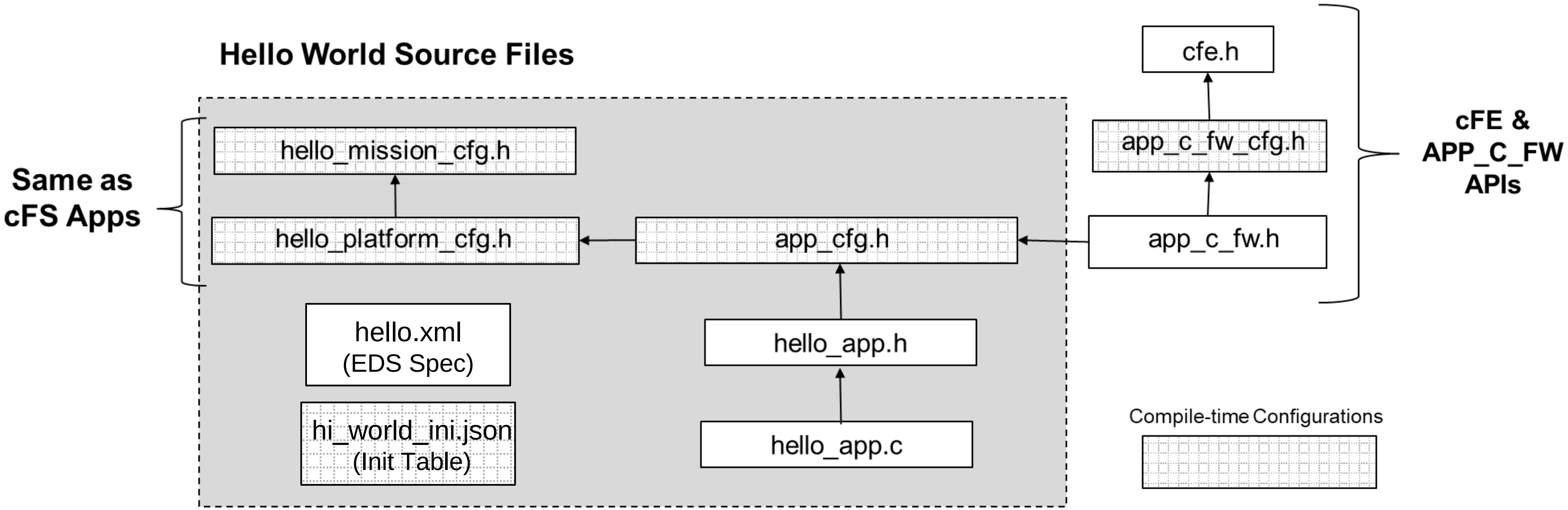
  - The "App C Framework" is a package available to all apps
  - Apps are composed of zero or more objects
  - Objects are composed of zero or more objects
- **The Hello World app is the simplest app with no objects**



Hello World Directory Structure

```
cfs-basecamp/usr/apps/hi_world
├-eds/
│ └-hi_world.xml . . . . . . . . Electronic Data Sheet spec defining app’s interface
│
├-fsw/
│ ├-mission_inc/
│ │ └-hi_world_mission_cfg.h . . Configurations for every app instance in a mission
│ ├-platform_inc/
│ │ └-hi_world_platform_cfg.h. . Configurations for each app platform instantiation
│ ├-src/
│ │ ├-app_cfg.h  . . . . . . . . Configurations intrinsic to the app
│ │ ├-hi_world.c
│ │ └-hi_world.h
│ └-tables/
│   └-cpu1_hi_world_ini.json . . Configurations that are processed during app’s initialization
│
├-CMakeLists.txt
└-hi_world_ini.json  . . . . . . App spec used in automated app store processes
```

Hello World Source Files



- The next slide describes the role of each file
- The build tools create header files (not shown) from the EDS spec
- Each Basecamp app defines it's own app_cfg.h file that defines the initialization table parameters and serves as a centralized point for other configuration header files including some generated from the EDS spec (not shown)

Hello World Source Files

Header File	Purpose
hello_mission_cfg.h	Analogous to cFS app mission config header in scope. Only contains parameters that must be defined during compilation, otherwise they should be in hi_world_ini.json
hello_platform_cfg.h	Analogous to cFS app platform config header in scope. Only contains parameters that must be defined during compilation, otherwise they should be in hi_world_ini.json
app_cfg.h	Every Basecamp app has a header with this name. Configurations have an application scope that define parameters that shouldn't need to change across deployments. If they need to change across deployments then they should be in mission/platform config files.
app_c_fw.h	Defines the API for the Application C Framework by including all of the framework component public header files
app_c_fw_cfg.h	Defines platform-scoped configuration parameters for the framework. The defaults should accommodate most deployments. The configurations must meet the needs of all apps on a platform sharing the framework.
cfe.h	Defines the cFE API and included by the framework so Basecamp definitions can build on cFE definitions.
hello_app.h	Demo app’s “class structure” that’s serves as the container of the object hierarchy
hello.xml	Electronic Data Sheet (EDS) specification for primarily application message definitions
hi_world_ini_ini.json	Configuration parameters that are read by the app when it initializes

Hello World App Main Loop Execution

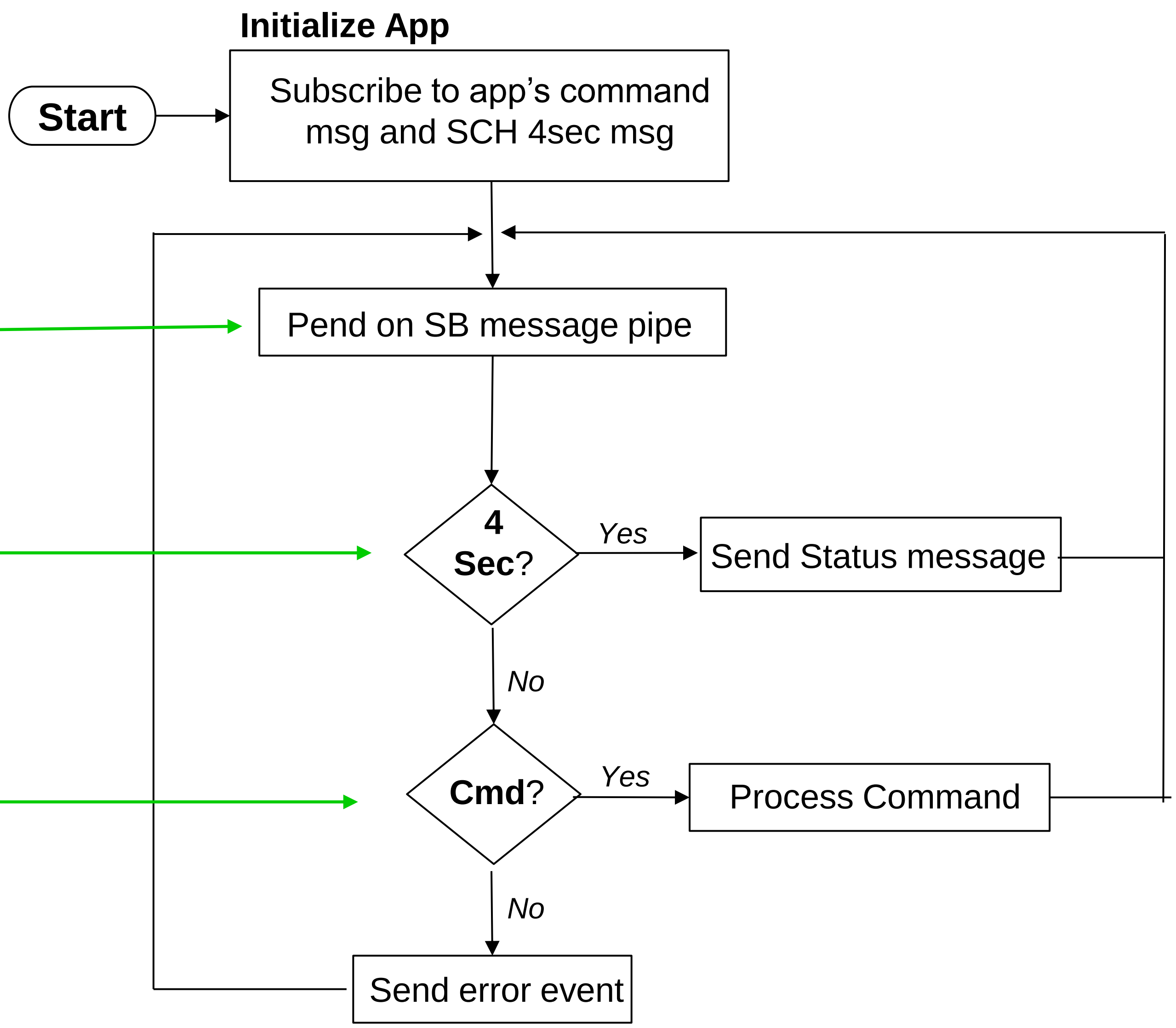
Suspend execution until a message arrives on app's pipe

Periodic *4 second* message from SCH app

- Send status telemetry message
- "Housekeeping cycle" convenient time to perform non-critical functions

Process commands

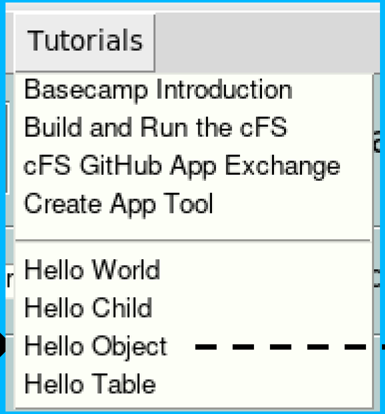
- Commands can originate from ground or other onboard apps
- Apps typically subscribe to one ground command message that uses unique function codes for each command



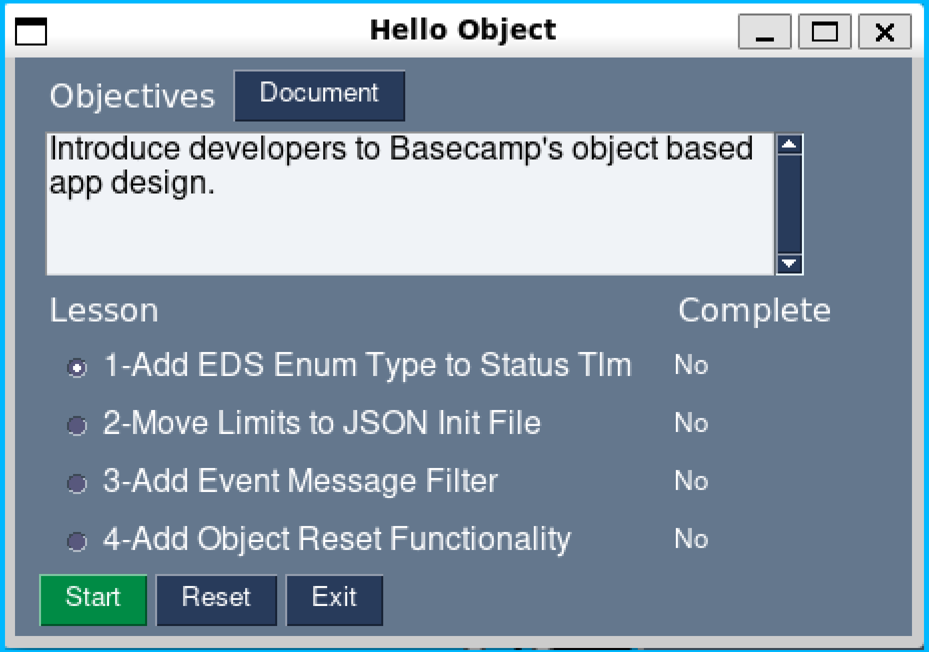
cFS Basecamp Exercises: Hello Object

1. Read through this section for a basic understanding of the Hello Object app
2. From the Tutorial dropdown list select “Hello Object” and do the Lessons
    - Refer back to these slides as needed to deepen your understanding

1



2



Introduce developers to Basecamp's object based app design.

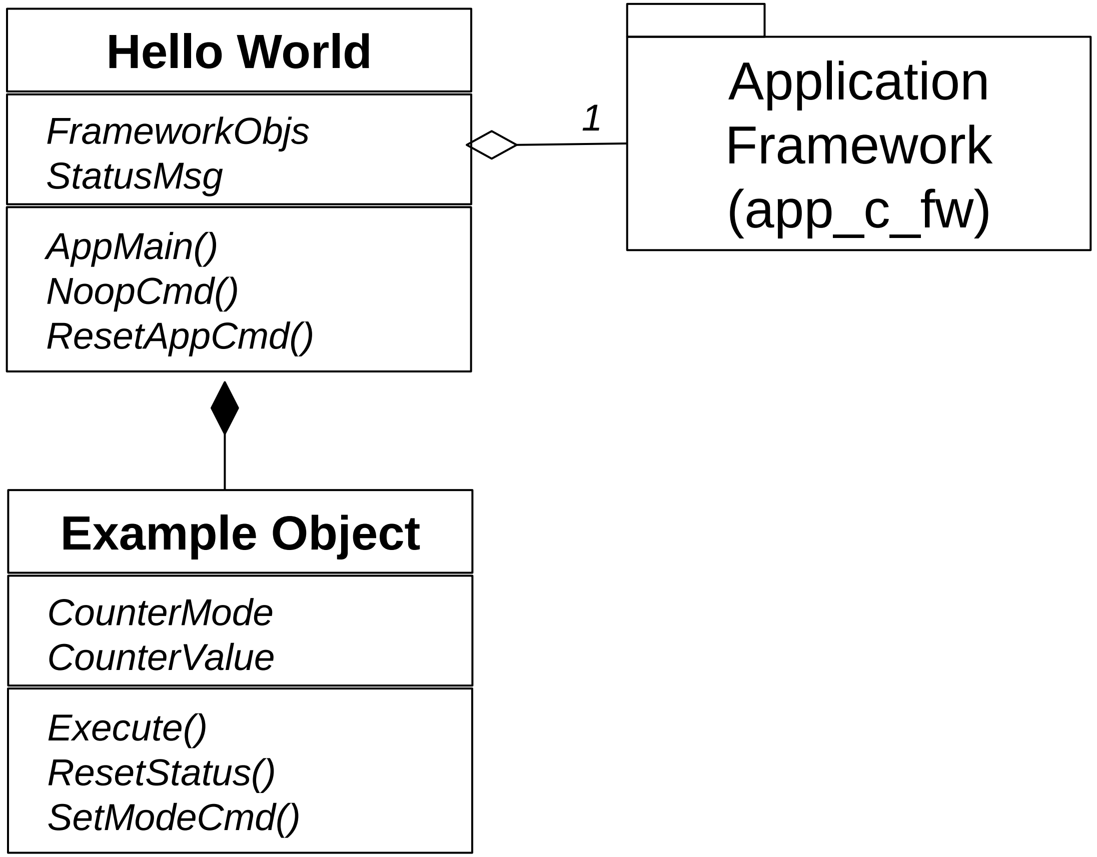
Lesson	Complete
● 1-Add EDS Enum Type to Status Tlm	No
○ 2-Move Limits to JSON Init File	No
○ 3-Add Event Message Filter	No
○ 4-Add Object Reset Functionality	No

Hello Object App Functionality

- **The Hello Object app adds an example object to the Hello World app**
  - The Hello World coding exercise additions are not part of the Hello World app baseline

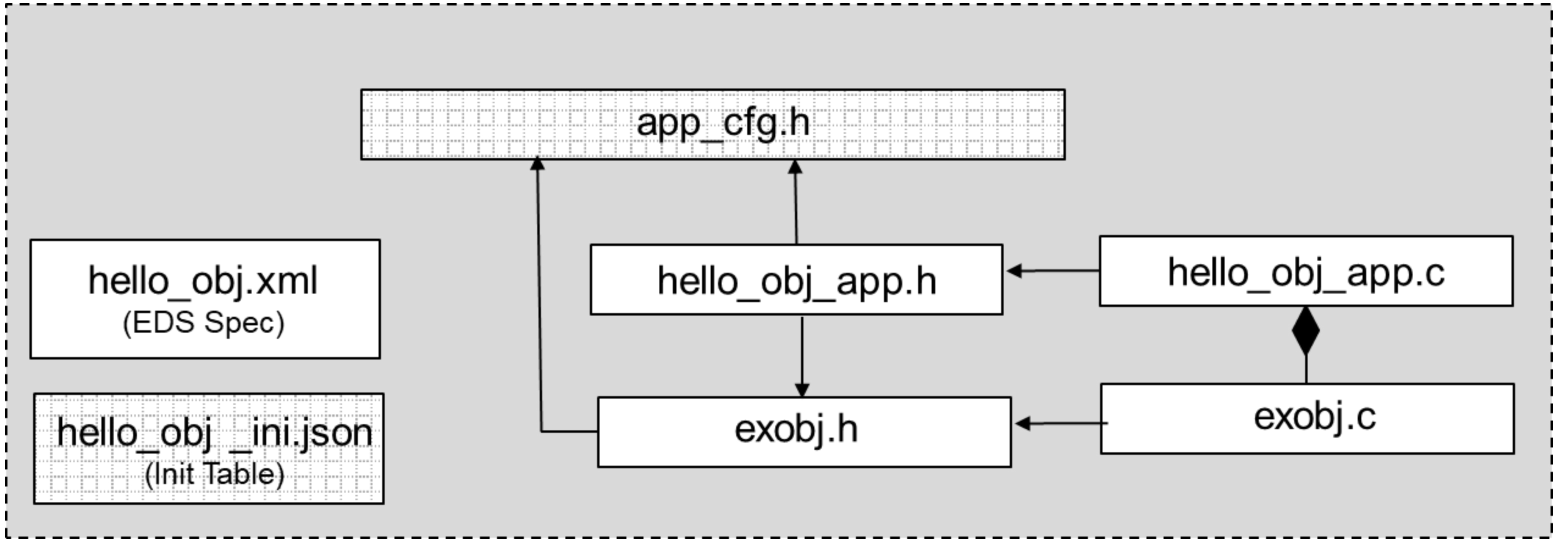
- **The example object performs the following functions**
  - Provides an up/down counter that can either be in an increment or decrement mode
  - Provides a command to set the counter mode
  - Defines lower and upper counter limits
  - The counters ‘wrap around’ using the limits
    - In increment mode when the upper limit is reached the counter value is set to the lower limit
    - In decrement mode when the lower limit is reached the counter value is set to the upper limit
  - The counter runs at 1Hz
  - The counter defaults to increment mode starting at the low limit
  - The current counter value and counter mode are in the status telemetry message

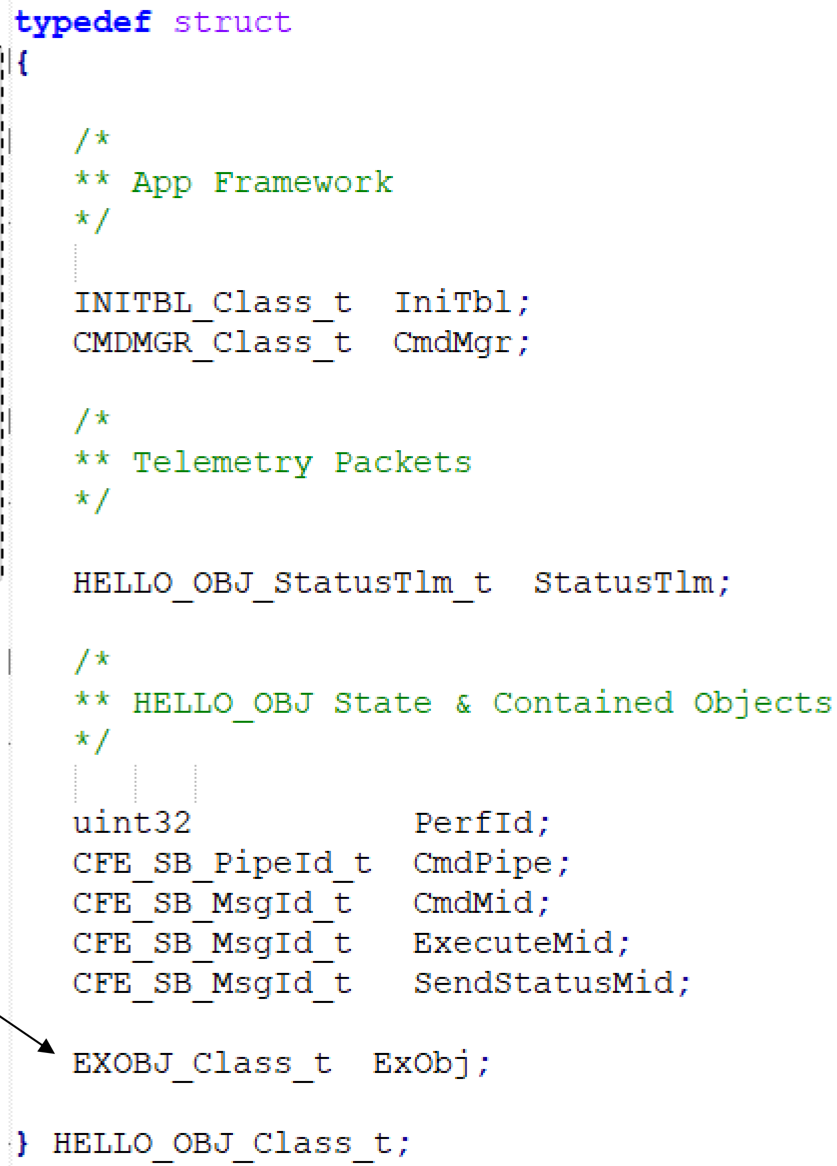
Hello Object Design



Hello Object Source Files

App Source Files





```
typedef struct
{

   /*
   ** App Framework
   */

   INITBL_Class_t  IniTbl;
   CMDMGR_Class_t  CmdMgr;

   /*
   ** Telemetry Packets
   */

   HELLO_OBJ_StatusTlm_t  StatusTlm;

   /*
   ** HELLO_OBJ State & Contained Objects
   */

   uint32            PerfId;
   CFE_SB_PipeId_t   CmdPipe;
   CFE_SB_MsgId_t    CmdMid;
   CFE_SB_MsgId_t    ExecuteMid;
   CFE_SB_MsgId_t    SendStatusMid;

   EXOBJ_Class_t  ExObj;

} HELLO_OBJ_Class_t;
```

- app_cfg.h has additional 'standard' includes that are not shown, see TBD for details
- Hello_obj includes exobj.h so it can declare an instance of EXOBJ in its class data

Hello Object App Main Loop Execution

Suspend execution until a message arrives on app's pipe

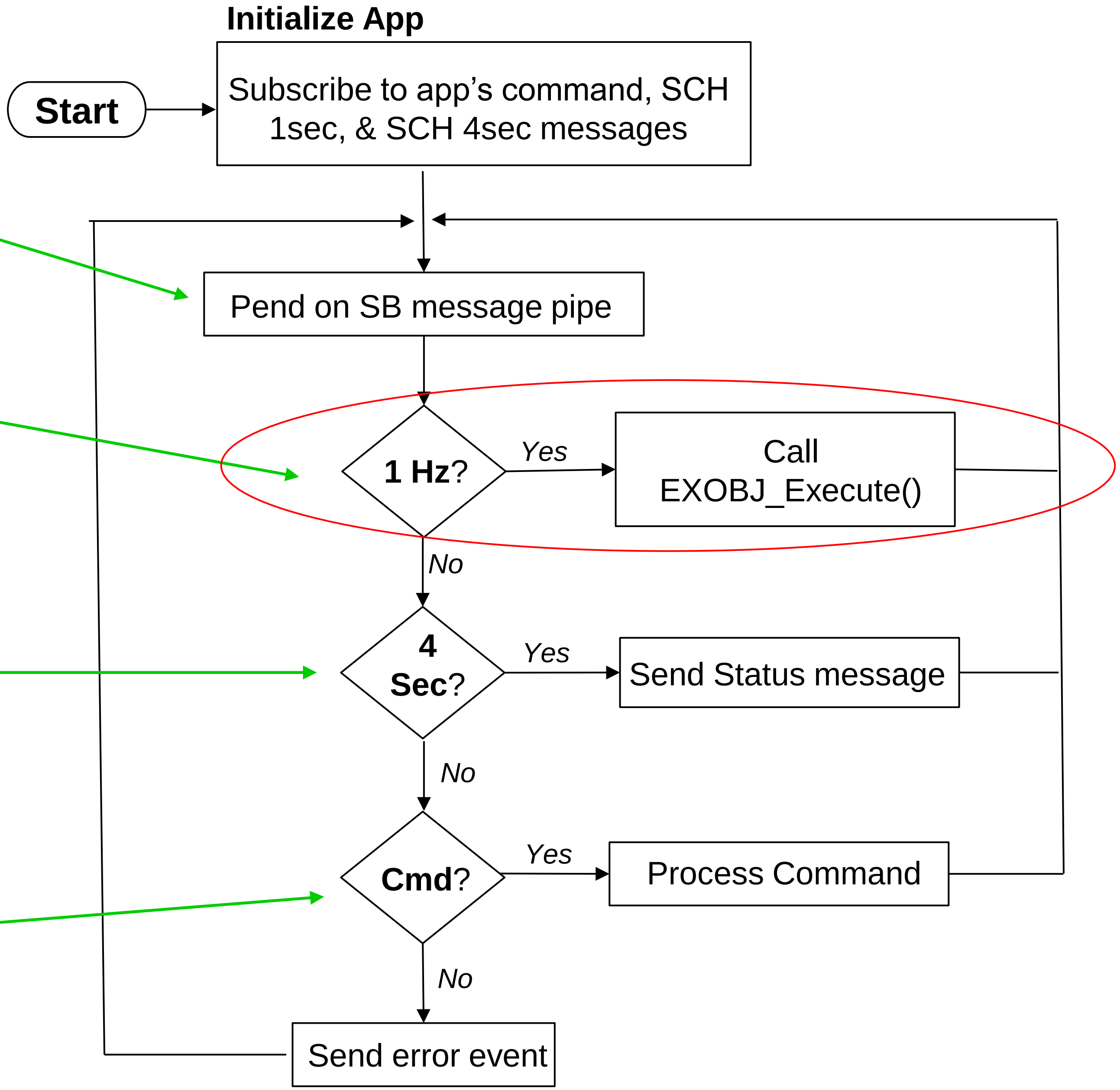
Periodic *1Hz* message from SCH app (added to Hello World)

Periodic *4 second* message from SCH app

- Send status telemetry message
- "Housekeeping cycle" convenient time to perform non-critical functions

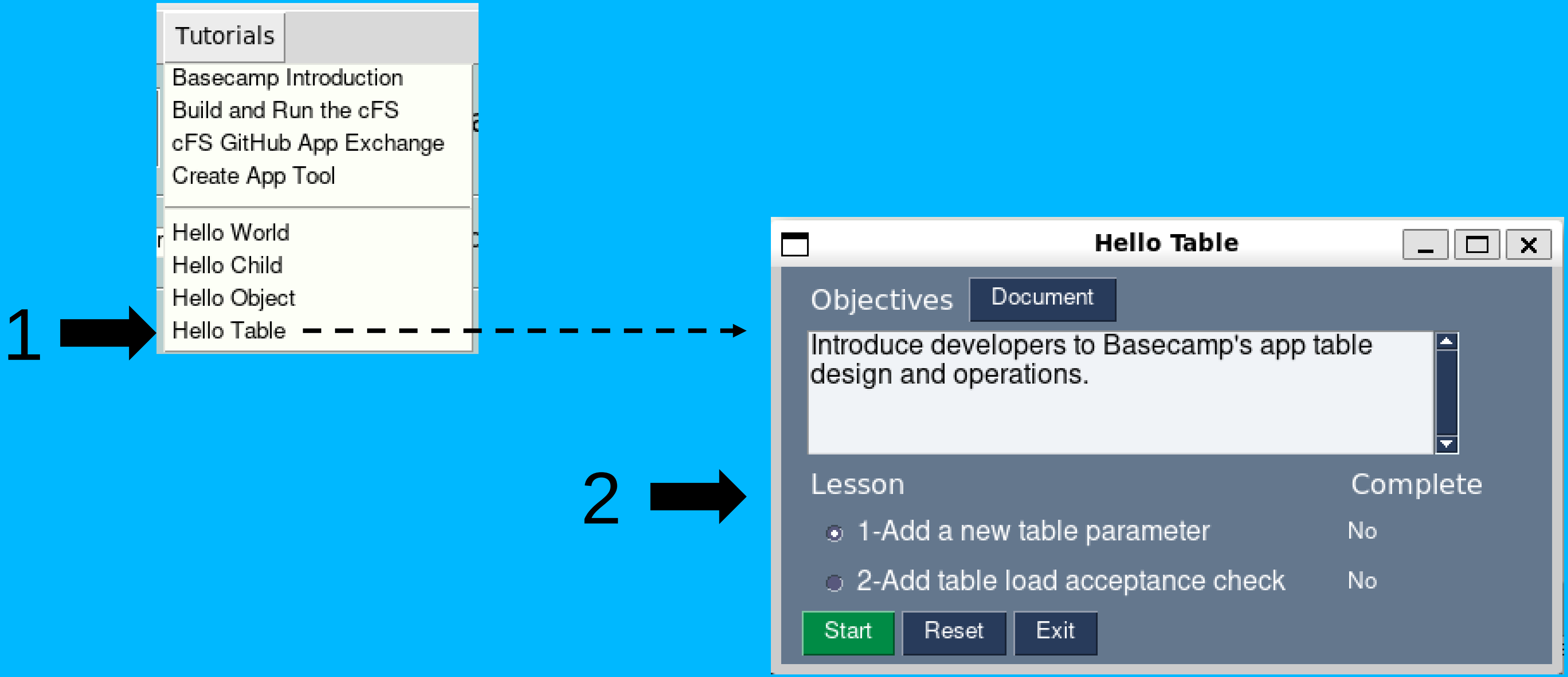
Process commands

- Commands can originate from ground or other onboard apps



cFS Basecamp Exercises: Hello Table

1. Read through this section for a basic understanding of the Hello Table app
2. From the Tutorial dropdown list select “Hello Table” and do the Lessons
    - Refer back to these slides as needed to deepen your understanding



Introduce developers to Basecamp's app table design and operations.

Lesson	Complete
● 1-Add a new table parameter	No
○ 2-Add table load acceptance check	No

Table Introduction

- **Basecamp's JSON tables serve the same purpose as cFS binary tables**
  - Tables are a collection of related parameters that could potentially change during runtime

- **If the app only has a couple of parameters then parameter command(s) may be easy to use than a table**

- **If there's a very low chance of a parameter changing during runtime and restarting the app is acceptable if the parameter ever needs to change then the app's JSON initialization file may be suitable**

- **The APP_C_FW table service uses the same JSON parser as app init parameter service**
  - Table parsing includes support for floating point parameters in addition to Integers and Strings

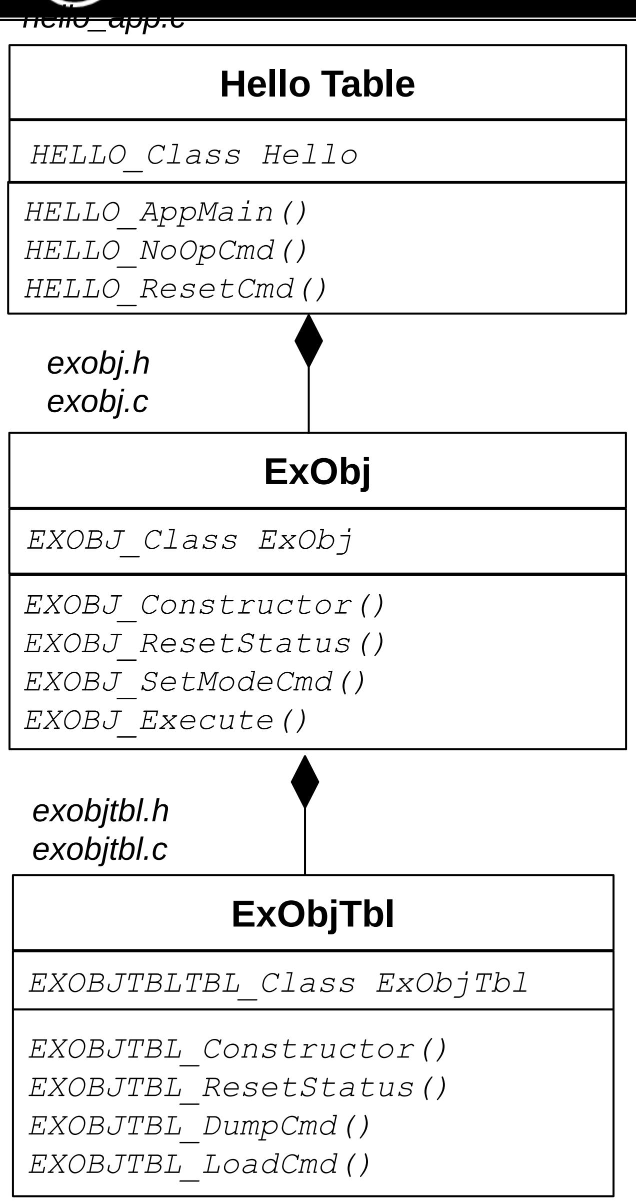
- **The object that owns the table object has the option to provide a table validation function**
  - This function is called as part of the Table Load command and the table values will not be used if validation fails

Hello Table App Functionality

- **Functional modifications to Hello Object**
  - The EDS-defined Counter Mode type is in the status telemetry message (retained from coding lesson)
  - The EXOBJ_Execute() event message is defined as a DEBUG event and an event filter allows the first 8 events to be published (retained from coding lesson)
  - The App reset command resets the event filter. EXOBJ does not have any reset behavior.
  - The counter limits are defined in a new parameter

- **Additional functionality**
  - The increment and decrement modes have separate low and high limits
  - The Set Mode command sends the limits in an information event message
  - A Table Load command reads/parses a JSON table file and loads the new parameters values into variables
  - A table load callback acceptance function, owned by EXOBJ, is called when a new table is loaded. The default functionality is to accept the table and send an event message. A coding lesson adds functionality to the acceptance function.
  - A Table Dump command creates a JSON table file using the parameters values from variables

Hello Table Design (1 of 2)



- **The App C Framework is an object-based design written in C**
- **Apps are constructed as an aggregation of objects**
  - Hello Table contains one Example Object (ExObj)
  - ExObj contains one Example Object Table (ExObjTbl)
  - The object hierarchy can be as wide or deep as needed
- **The key roles of the main app are to**
  - Read the app’s JSON initialization configuration file
  - Initialize contained objects and register their commands
  - Manage the main control loop
- **Contained objects implement the ‘business logic’**
  - ExObj increments a counter during each execution cycle
  - ExObj’s Set Mode command supports increment and decrement
  - ExObjTbl defines the counter’s lower and upper limits

Hello Table Design (2 of 2)

- **The app_c_fw TBLMGR object is owned by the app's main object and is constructed prior to constructing objects owned by the app that need to register a table**
  - EXOBJ owns and constructs the table

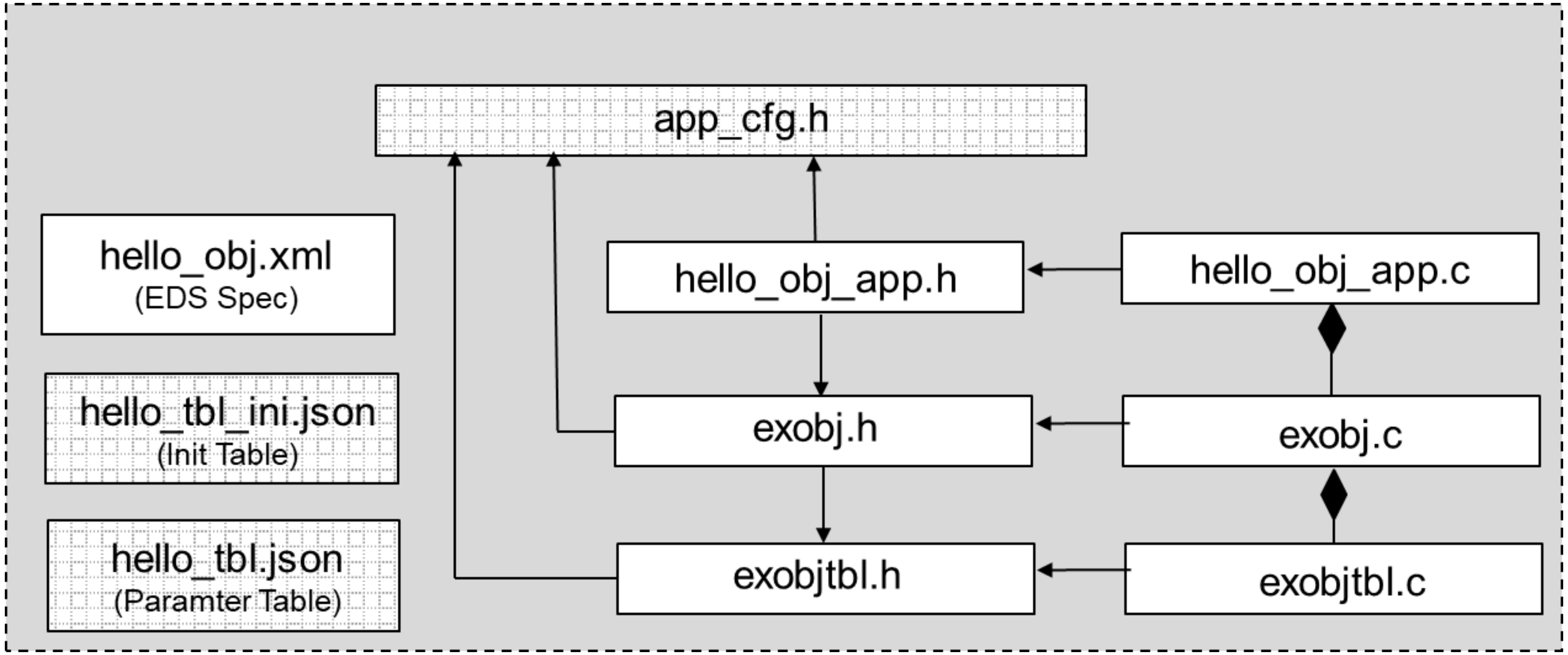
- **The default table name is defined in an app's init table**
  - The init parameter name is defined in app_cfg.h

- **app_c_fw defines common command codes for the table load and dump commands**
  - All apps that have tables with use the same command codes just like the Noop and Reset commands

- **By convention, apps with tables report the last table action and action status in their status telemetry**

Hello Table Source Files

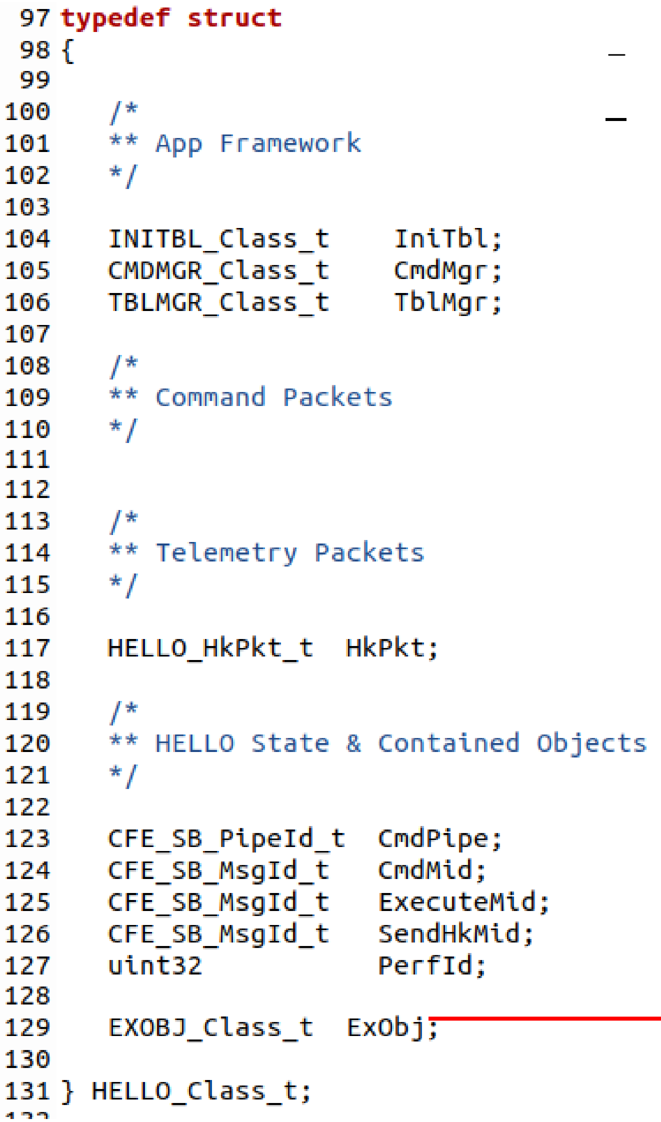
App Source Files



Hello Table Header Files

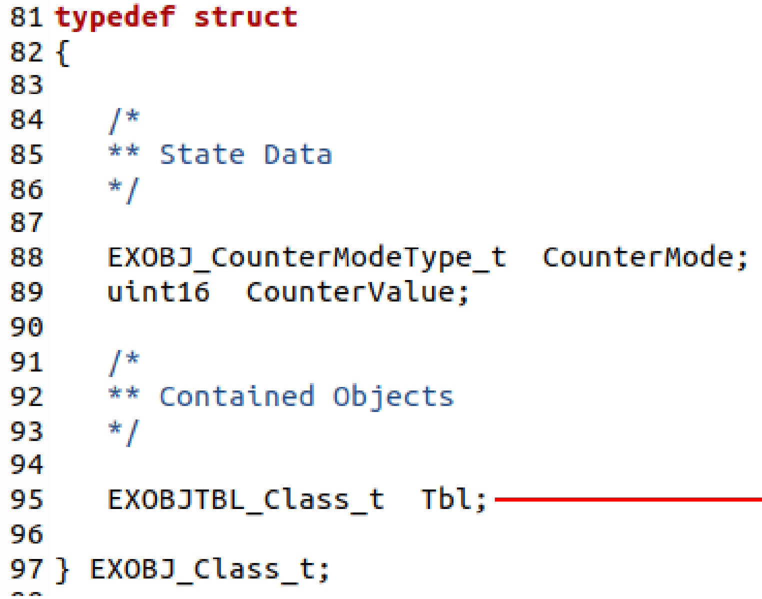
- **Use a variation of the ‘singleton” design pattern**
  - Object constructors passed reference to owner’s storage
  - `void EXOBJ_Constructor(EXOBJ_Class_t *ExObjPtr, ...);`
  - EXOBJ uses a static variable to store pointer so subsequent EXOBJ function (i.e. method) calls don’t require a pointer to be passed

hello_table.h



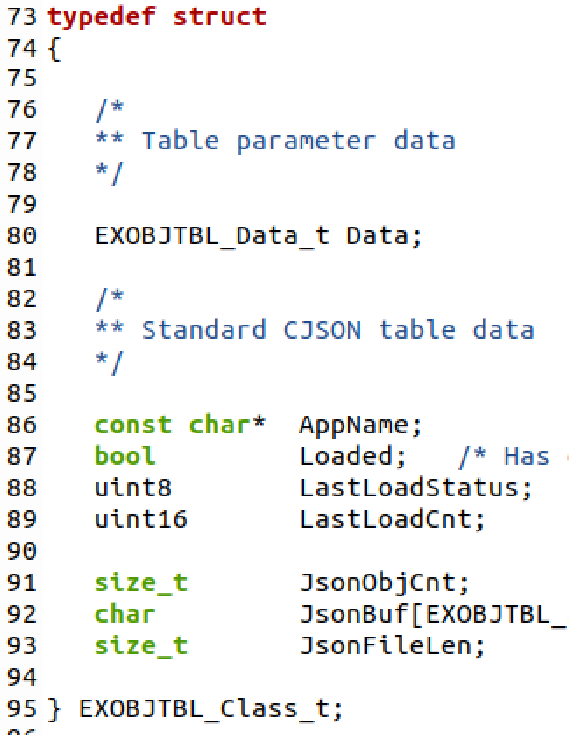
```
typedef struct
{

   /*
   ** App Framework
   */

   INITBL_Class_t    IniTbl;
   CMDMGR_Class_t    CmdMgr;
   TBLMGR_Class_t    TblMgr;

   /*
   ** Command Packets
   */


   /*
   ** Telemetry Packets
   */

   HELLO_HkPkt_t  HkPkt;

   /*
   ** HELLO State & Contained Objects
   */

   CFE_SB_PipeId_t  CmdPipe;
   CFE_SB_MsgId_t   CmdMid;
   CFE_SB_MsgId_t   ExecuteMid;
   CFE_SB_MsgId_t   SendHkMid;
   uint32           PerfId;

   EXOBJ_Class_t  ExObj;

} HELLO_Class_t;
```

exobj.h



```
typedef struct
{

   /*
   ** State Data
   */

   EXOBJ_CounterModeType_t  CounterMode;
   uint16  CounterValue;

   /*
   ** Contained Objects
   */

   EXOBJTBL_Class_t  Tbl;

} EXOBJ_Class_t;
```

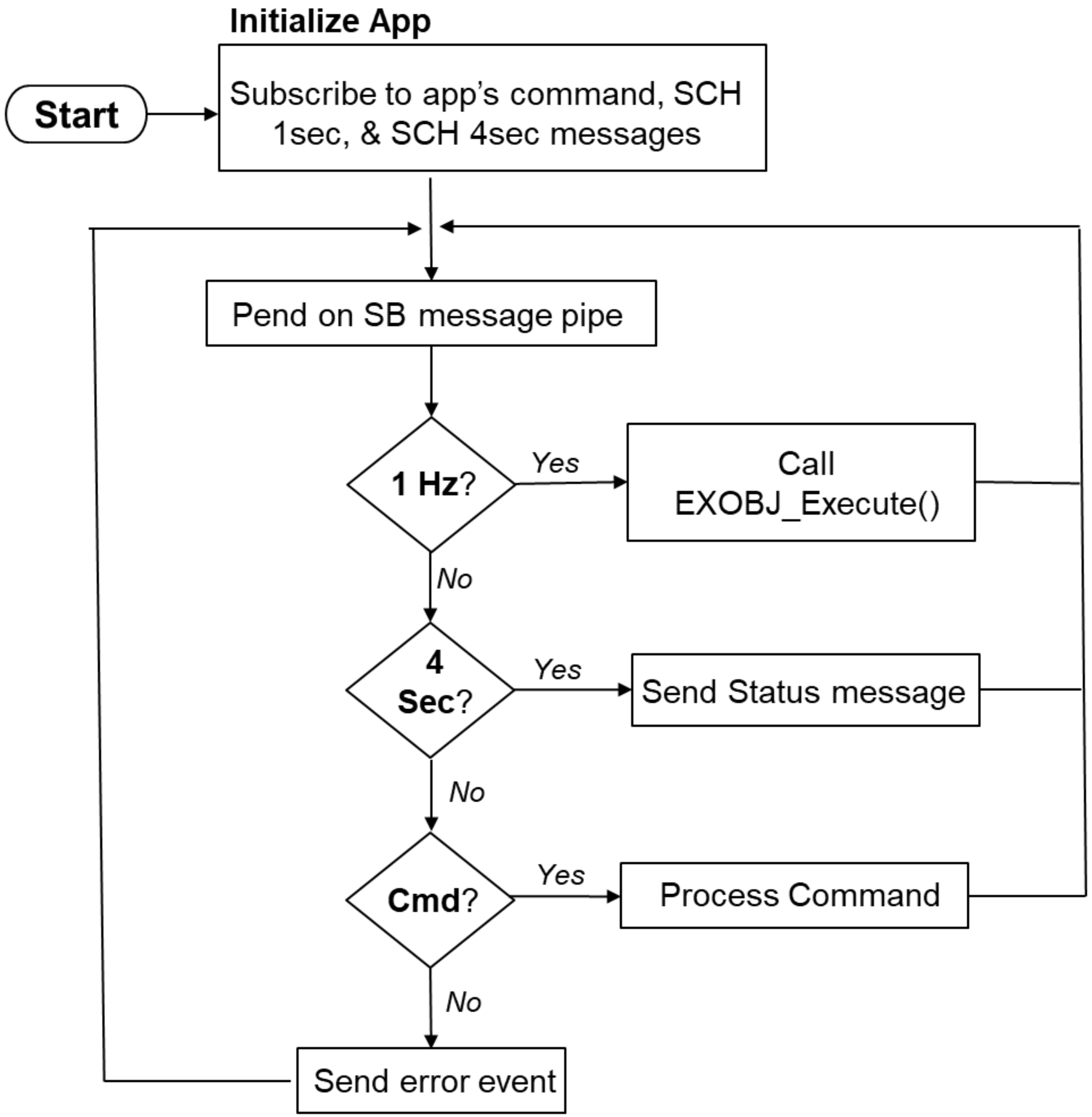
exobjtbl.h



```
typedef struct
{

   /*
   ** Table parameter data
   */

   EXOBJTBL_Data_t Data;

   /*
   ** Standard CJSON table data
   */

   const char*  AppName;
   bool         Loaded;   /* Has
   uint8        LastLoadStatus;
   uint16       LastLoadCnt;

   size_t       JsonObjCnt;
   char         JsonBuf[EXOBJTBL_
   size_t       JsonFileLen;

} EXOBJTBL_Class_t;
```

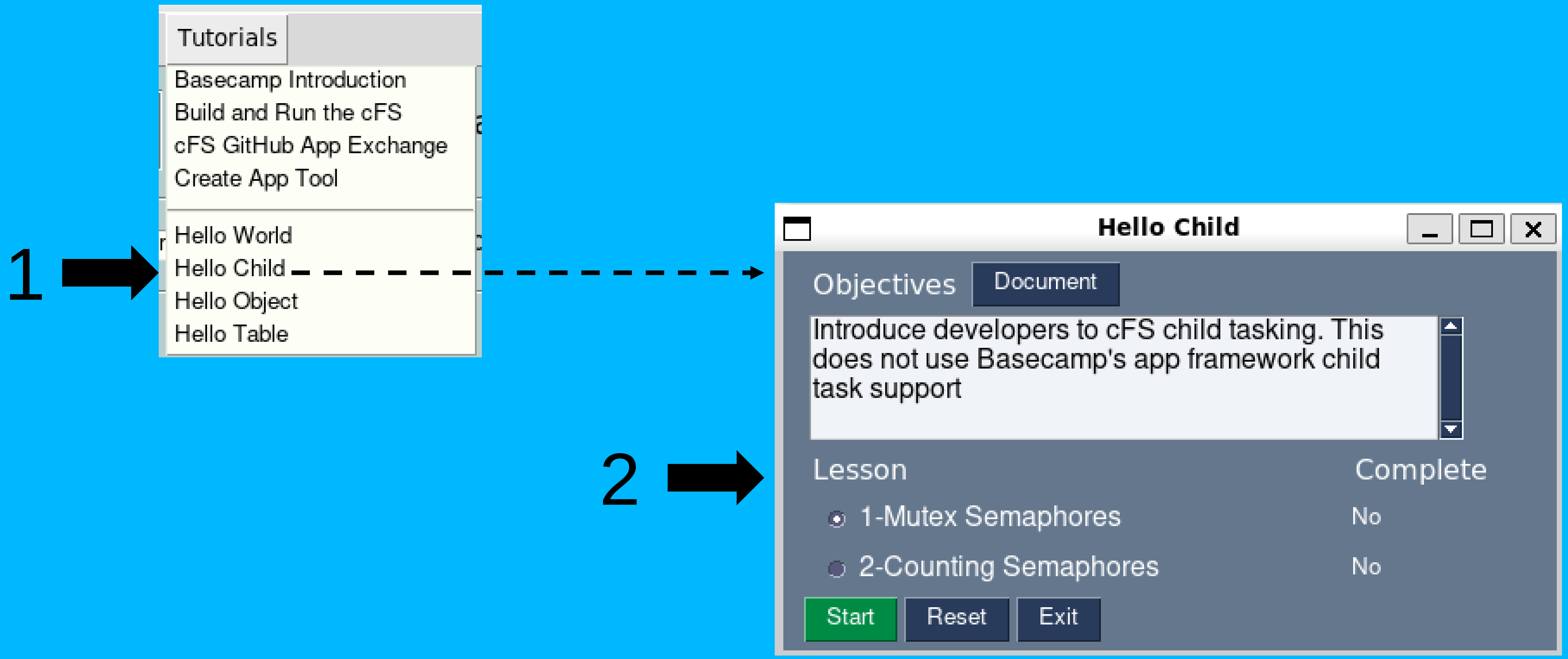
Hello Table App Main Loop Execution

Same logic as Hello Object



cFS Basecamp Exercises: Hello Child

1. Read through this section for a basic understanding the Hello Child app
2. From the Tutorial dropdown list select “Hello Child” and do the Lessons
    - Refer back to these slides as needed to deepen your understanding



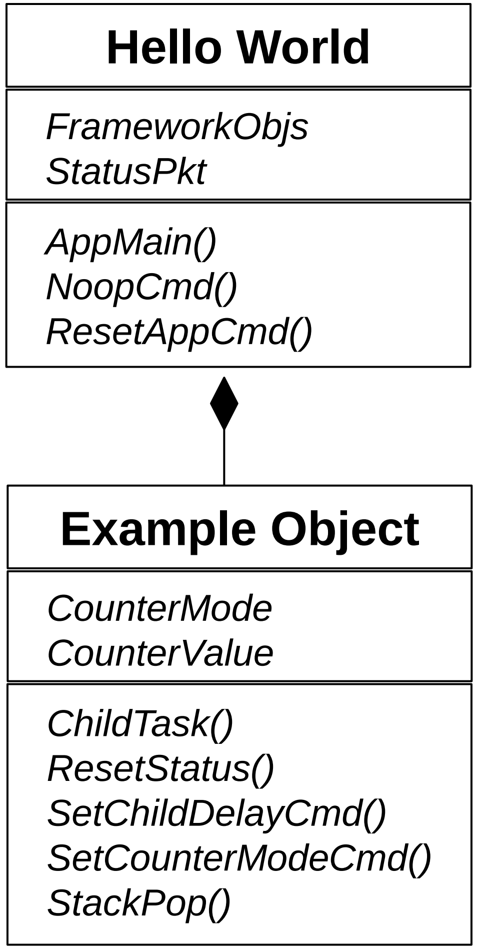
Introduce developers to cFS child tasking. This does not use Basecamp's app framework child task support

Lesson	Complete
● 1-Mutex Semaphores	No
○ 2-Counting Semaphores	No

Child Task Introduction

- **cFS apps can create one of more child tasks to perform functions that run in an execution thread separate from the parent app**
  - The execution thread implementation is specific to the cFS target platform
- **Child resource are owned by the parent app**
- **Parent and child share memory address space**
  - Use semaphores to prevent simultaneous memory access conflicts
- **Convention is to create child tasks when the parent app initializes**
  - System initialization timing is usually less stringent and dynamic resource management is minimized when the system is operational
  - Helps establish a deterministic operational state for realtime systems that have strict timing requirements
- **Common use cases**
  - Low priority CPU intensive background tasks, e.g. File Manager and Checksum apps
  - High priority, typically short duration, e.g. MQTT Gateway app
  - See the child task 'Use Cases' and examples to help guide your decision

Hello Child Design



Hello World

FrameworkObjs
StatusPkt

AppMain()
NoopCmd()
ResetAppCmd()

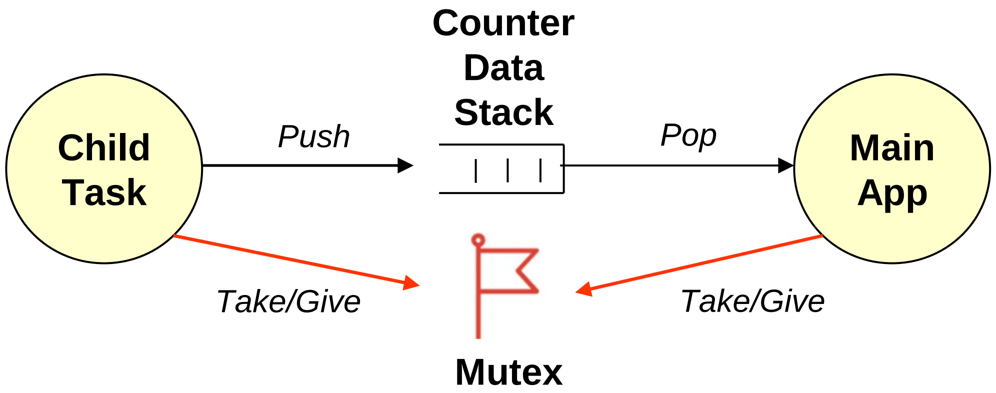
Example Object

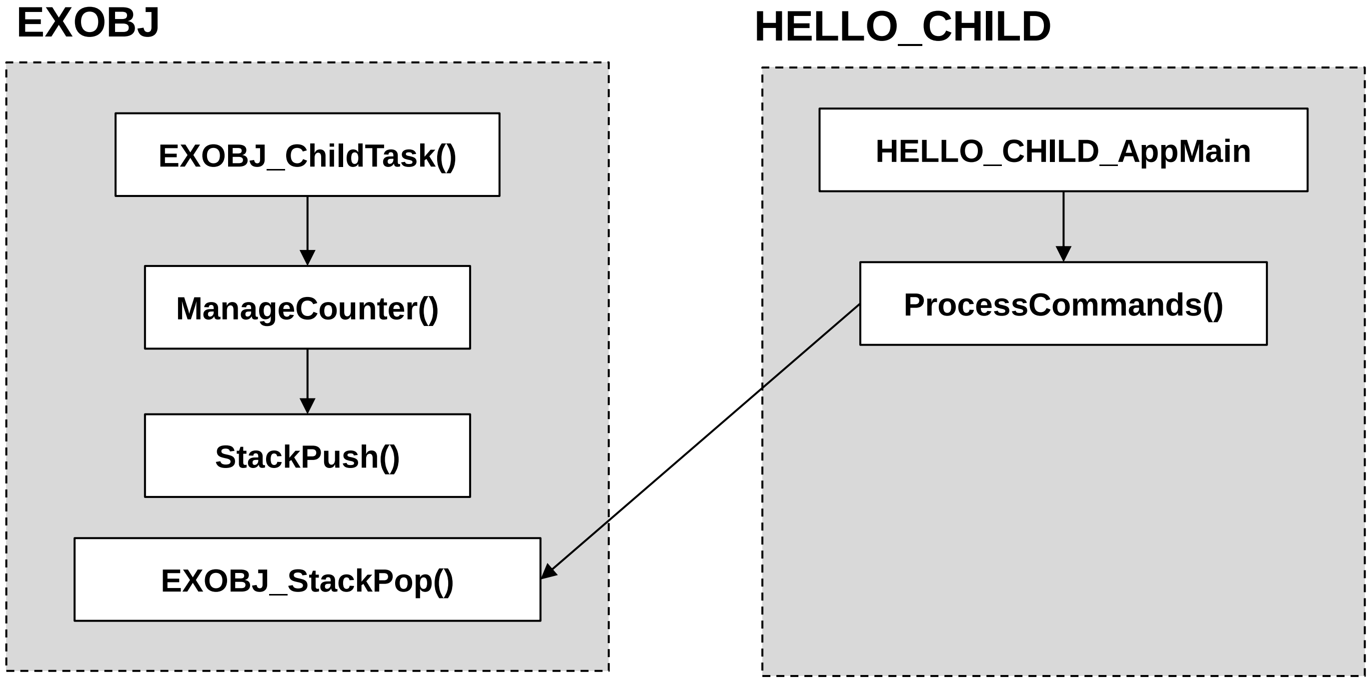
CounterMode
CounterValue

ChildTask()
ResetStatus()
SetChildDelayCmd()
SetCounterModeCmd()
StackPop()

- **The Hello Child app has the same objects as the Hello Object app however the public interface has changed to accommodate running the Execute() function as a child task**
  - EXOBJ_ChildTask() replaces EXOBJ_Execute()

Hello Child App Design (2 of 2)





- Coding lesson 1 adds the stack functionality
- A Mutex Semaphore is used to coordinate access to the shared *Counter Data Stack*
- The cFS naming convention is to prefix global functions with the object's name

Section 2

Demo App Overview

Demo App Introduction

- **The Demo app APP_C_DEMO is included in Basecamp's default app suite**
- **Basecamp's built-in tutorials use APP_C_DEMO to help user's quickly learn how to use Basecamp**
- **APP_C_DEMO's features and design provide a fully functional non-trivial app that**
  - Goes beyond the simple *Hello* apps that are designed to teach a specific feature of cFS apps
  - Is easy for users to quickly understand and operate
  - Has enough complexity so it can be used illustrate most Basecamp operational features and use a large percentage of the APP_C_FW framework
  - Note APP_C_DEMO functions are designed to help teach app development concepts and may not be practical for a space mission
- **This section describes APP_C_DEMO from an operational perspective so users can use APP_C_DEMO to learn Basecamp's features**
  - Section 4 uses the Demo app as examples for developing with the APP_C_FW
  - Section 5 describes the Demo app's design

APP_C_DEMO Functions

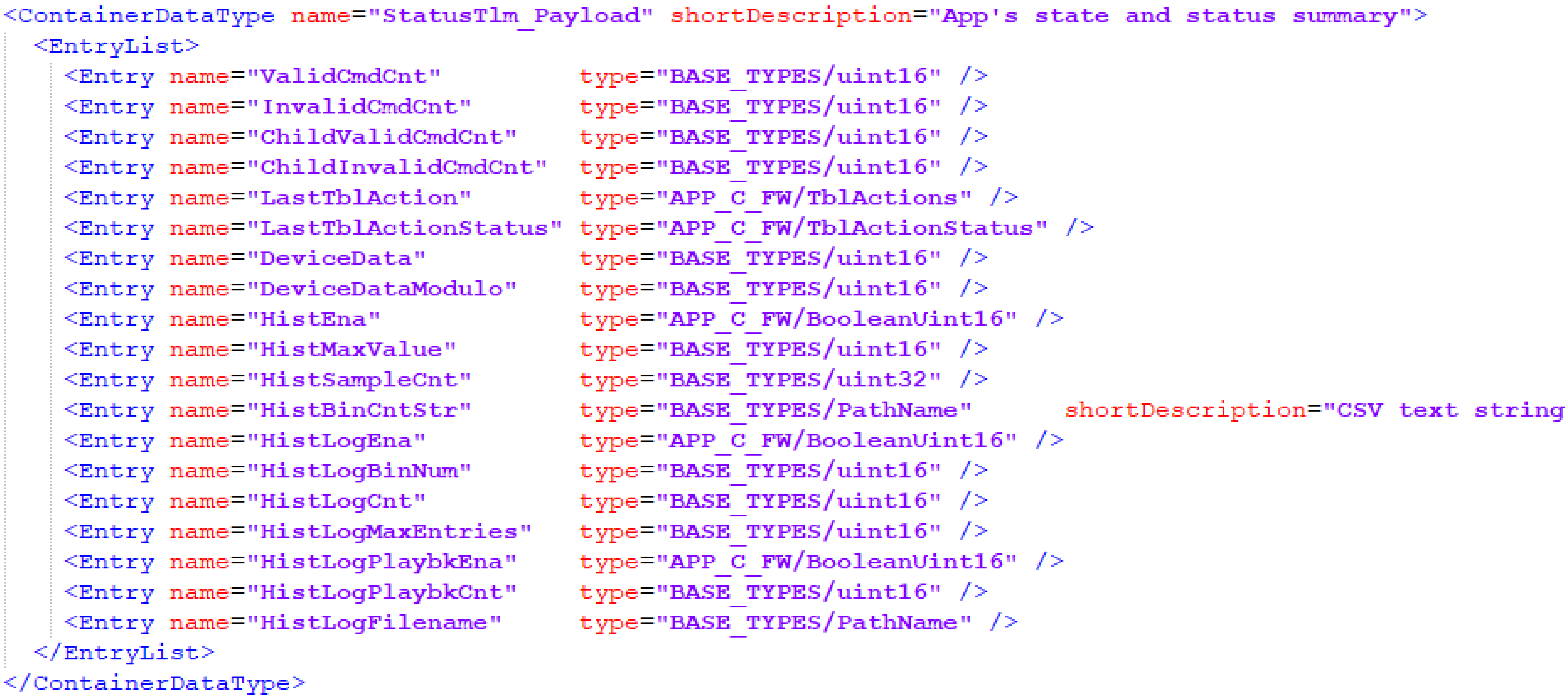
- **APP_C_DEMO computes a histogram for a randomly generated integer designated as “Device Data”**
  - The histogram records how many times randomly generated data samples fall within a set of data ranges (bins)
  - The histogram JSON table defines the histogram bin limits
- **The following commands control the app’s functionality**
  - *Start Histogram*
    - Start computing a histogram that is reported in the Status Telemetry message
  - *Stop Histogram*
  - *Start Histogram Log*
    - Create a log file containing time-stamped entries for a command-specified bin up to a command-specified number of entries
  - *Stop Histogram Log*
  - *Start Histogram Log Playback*
    - Loop through the histogram log file sending each entry in a telemetry message
  - *Stop Histogram Log Playback*

Status Telemetry

- **The status telemetry message is sent at a 1Hz rate**
  - The random *Device Data* is generated at 1Hz so each value is telemetered
  - The Demo app uses the Scheduler app’s (KIT_SCH) 1Hz message published on the Softwrae Bus to determine when to run
  - Demo app’s JSON initiialization file defines the Scheduler app’s Topic ID (superset of message ID) used for it’s execution rate

- **The status telemetry message is defined using an Electronic Data Sheet specification that is shown on the next slide**
  - Including valid/invalid command counters is standard practice because it allows operators to receive feedback on commands
  - Note any state information affected by a command is included

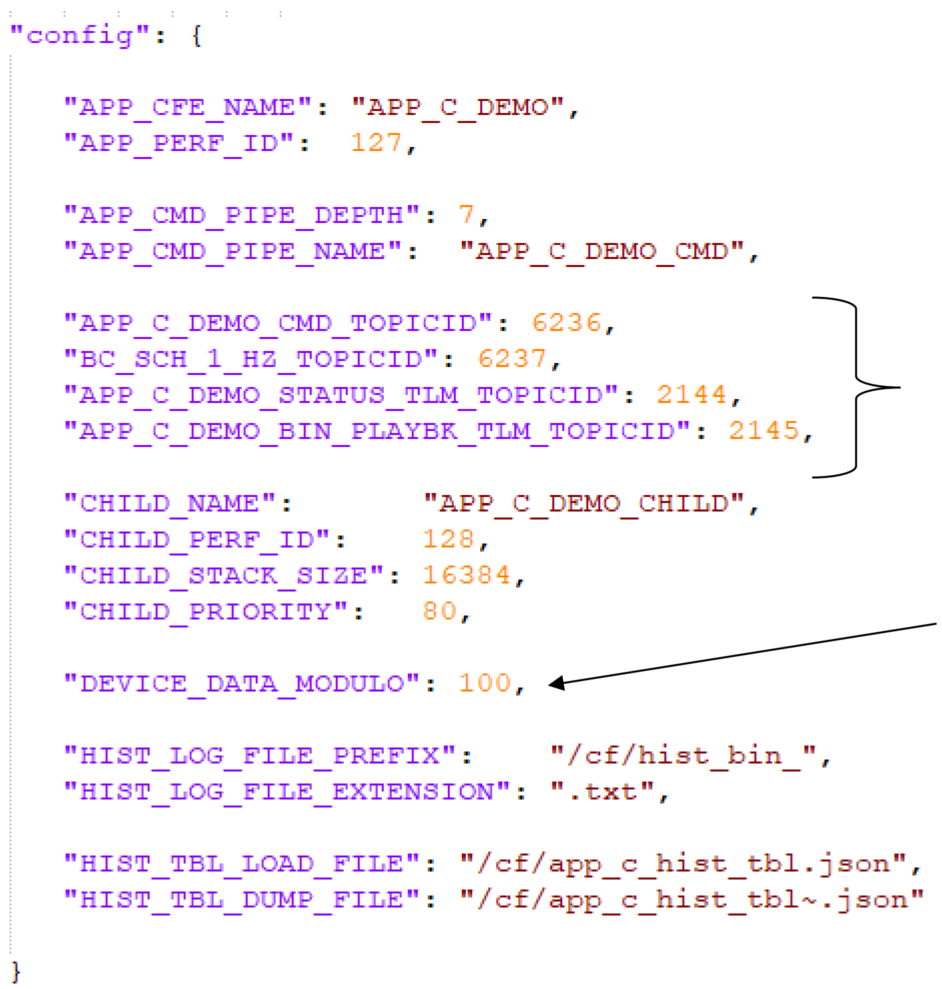
Status Telemetry Electronic Data Sheet Definition



```
<ContainerDataType name="StatusTlm_Payload" shortDescription="App's state and status summary">
  <EntryList>
    <Entry name="ValidCmdCnt"         type="BASE_TYPES/uint16" />
    <Entry name="InvalidCmdCnt"       type="BASE_TYPES/uint16" />
    <Entry name="ChildValidCmdCnt"    type="BASE_TYPES/uint16" />
    <Entry name="ChildInvalidCmdCnt"  type="BASE_TYPES/uint16" />
    <Entry name="LastTblAction"       type="APP_C_FW/TblActions" />
    <Entry name="LastTblActionStatus" type="APP_C_FW/TblActionStatus" />
    <Entry name="DeviceData"          type="BASE_TYPES/uint16" />
    <Entry name="DeviceDataModulo"    type="BASE_TYPES/uint16" />
    <Entry name="HistEna"             type="APP_C_FW/BooleanUint16" />
    <Entry name="HistMaxValue"        type="BASE_TYPES/uint16" />
    <Entry name="HistSampleCnt"       type="BASE_TYPES/uint32" />
    <Entry name="HistBinCntStr"       type="BASE_TYPES/PathName"     shortDescription="CSV text string
    <Entry name="HistLogEna"          type="APP_C_FW/BooleanUint16" />
    <Entry name="HistLogBinNum"       type="BASE_TYPES/uint16" />
    <Entry name="HistLogCnt"          type="BASE_TYPES/uint16" />
    <Entry name="HistLogMaxEntries"   type="BASE_TYPES/uint16" />
    <Entry name="HistLogPlaybkEna"    type="APP_C_FW/BooleanUint16" />
    <Entry name="HistLogPlaybkCnt"    type="BASE_TYPES/uint16" />
    <Entry name="HistLogFilename"     type="BASE_TYPES/PathName" />
  </EntryList>
</ContainerDataType>
```

JSON Initialization File

- **All deployment configurations are defined in an app's JSON file unless the configuration must be part of the build process in which case they are defined in C header files.**



```
"config": {

   "APP_CFE_NAME": "APP_C_DEMO",
   "APP_PERF_ID":  127,

   "APP_CMD_PIPE_DEPTH": 7,
   "APP_CMD_PIPE_NAME":  "APP_C_DEMO_CMD",

   "APP_C_DEMO_CMD_TOPICID": 6236,
   "BC_SCH_1_HZ_TOPICID": 6237,
   "APP_C_DEMO_STATUS_TLM_TOPICID": 2144,
   "APP_C_DEMO_BIN_PLAYBK_TLM_TOPICID": 2145,

   "CHILD_NAME":       "APP_C_DEMO_CHILD",
   "CHILD_PERF_ID":    128,
   "CHILD_STACK_SIZE": 16384,
   "CHILD_PRIORITY":   80,

   "DEVICE_DATA_MODULO": 100,

   "HIST_LOG_FILE_PREFIX":    "/cf/hist_bin_",
   "HIST_LOG_FILE_EXTENSION": ".txt",

   "HIST_TBL_LOAD_FILE": "/cf/app_c_hist_tbl.json",
   "HIST_TBL_DUMP_FILE": "/cf/app_c_hist_tbl~.json"

}
```

Messages IDs (EDS Topic IDs) used by Demo app. Values populated during cFS target build process

Modulo is a mathematical operator that returns the remainder after dividing one number by another. A modulo operator applied to a randomly generated number will return a value from 0 to 99.

Histogram JSON Parameter File

JSON parameter files are equivalent to cFS tables.

In cFS nomenclature file read and write operations are called table loads and dumps, respectively.

This tabkle defines the number of histogram bins and data value ranges for each bin. Since it’s a table it can be changed while the app is running.

JSON Parameter Definitions

```
"bin-cnt": 5,
  "bin": [
     {
       "lo-lim":  0,
       "hi-lim": 19
     },
     {
       "lo-lim": 20,
       "hi-lim": 39
     },
     {
       "lo-lim": 40,
       "hi-lim": 59
     },
     {
       "lo-lim": 60,
       "hi-lim": 79
     },
     {
       "lo-lim": 80,
       "hi-lim": 99
     }
  ]
```

Section 3

Electronic Data Sheets

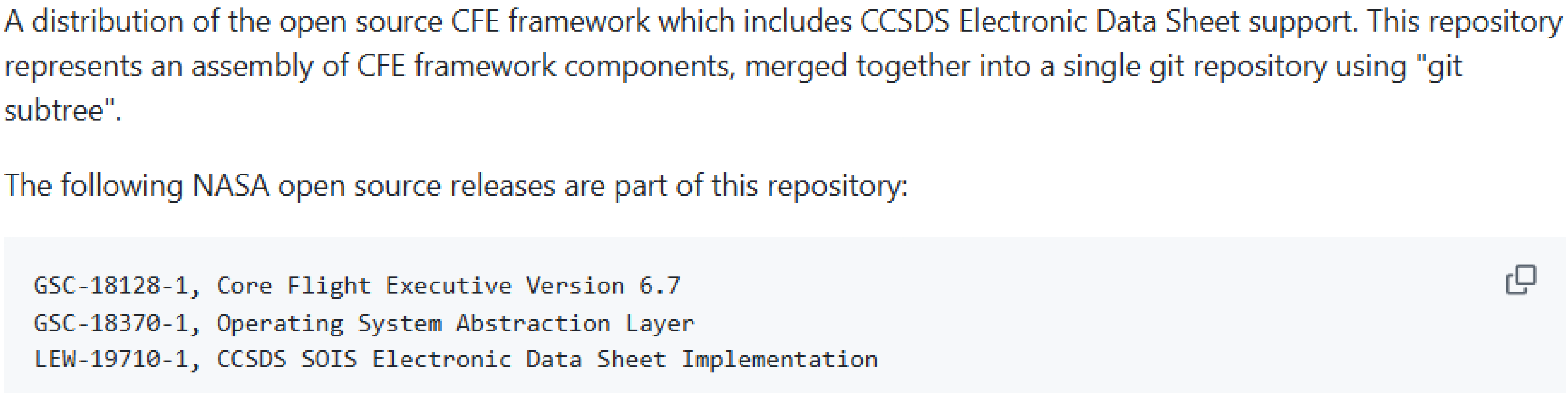
What are Electronic Data Sheets?

- **CCSDS Electronic Data Sheets (EDS) are formal, machine-readable specifications for the interfaces of hardware and software used in space missions**
  - CCSDS 876.0-B-1 Spacecraft Onboard Interface Services--XML Specification for Electronic Data Sheets
- **EDS specifies black box view of interfaces**
  - Data formats, conversions, limits, exchange protocols (state machines)
  - It is not an Interface Control Document (ICD) meaning it does not specify how a system or mission will use the device or software
- **EDS tools are required to translate the XML specifications into artifacts that can be used by the system that is interfacing to component with the EDS specification**

Basecamp EDS Toolchain

- **As of December 2024, NASA's cFS github main branch does not include an EDS toolchain**
  - NASA has stated that EDS is part of their technical roadmap and EDS spcifications for the cFE services are part of the current github main branch

- **Basecamp uses the following cFS tech branch for its EDS toolchain:**
  - https://github.com/jphickey/cfe-eds-framework



A distribution of the open source CFE framework which includes CCSDS Electronic Data Sheet support. This repository represents an assembly of CFE framework components, merged together into a single git repository using "git subtree".

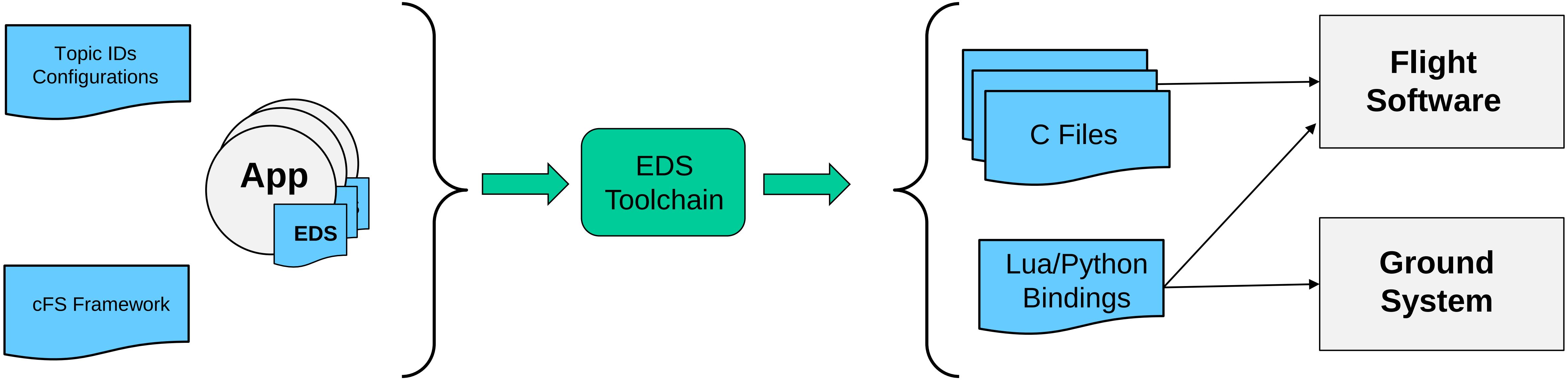
The following NASA open source releases are part of this repository:

```
GSC-18128-1, Core Flight Executive Version 6.7
GSC-18370-1, Operating System Abstraction Layer
LEW-19710-1, CCSDS SOIS Electronic Data Sheet Implementation
```

Key EDS Directories & Files

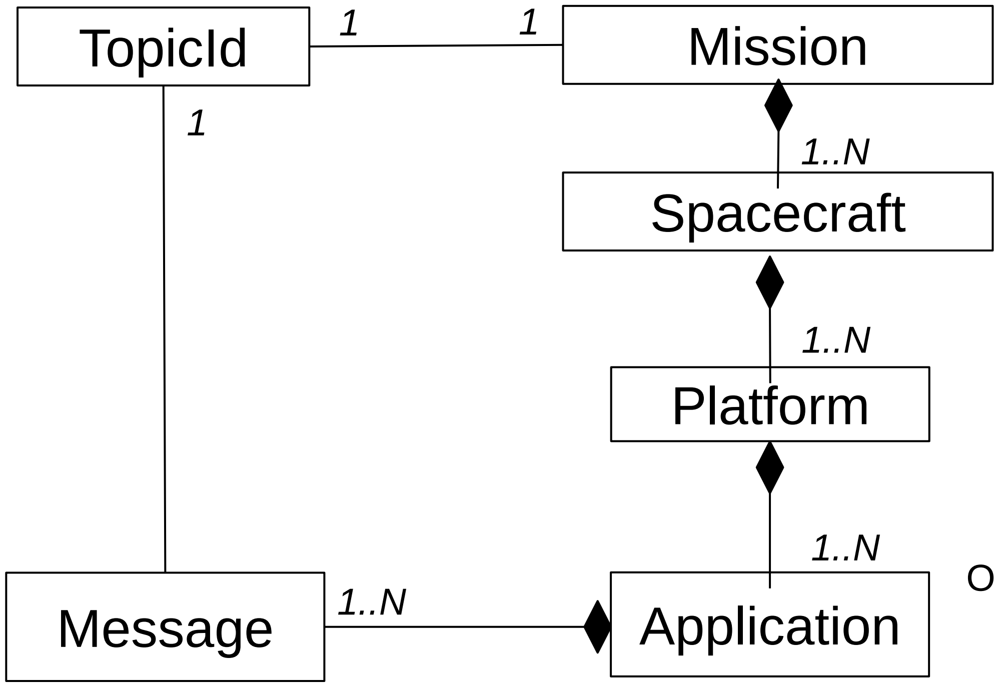
```
cfs-basecamp
├-apps/appc_fw/eds . . . . . . . app_c_fw.xml - App framework type definitions
│
├-cfe-eds-framework/ . . . . . . cFS Framework with Electronic Data Sheet build toolchain
│ ├-basecamp_defs  . . . . . . . Target definition folder
│ │ └-eds/ . . . . . . . . . . . config.xml - Assign values to cFE configuration parameters
│ │                              cfe-topicids.xml - Defines command&telemetry message topic IDs
│ │
│ └-cfe/modules/
│ │ ├-core_app/eds/. . . . . . . base_types.xml - Defines basic numeric
│ │ └-*/eds  . . . . . . . . . . [module].xml -
│ │
│ └-tools/eds/cfecfs/edsmsg/eds. ccsds_spacepacket.xml - Define CCSDS book 133 Space Packet Protocol
│                                cfe_hdr.xml - cFE usage of CCSDS space packets
│
└-usr/apps/*/. . . . . . . . . . Base directory for user apps
  └-eds/ . . . . . . . . . . . . [app].xml - Every app has an EDS spec
```

Electronic Data Sheet Workflow



- Command and telemetry definitions are defined using EDS
- The EDS Toolchain produces ground and flight code from the EDS
- **The ground and flight code must be generated from the same EDS definitions!**
  - This is straightforward when running a cFS target that is hosted on the same platform as Basecamp's GUI and communicating using 127.0.0.1 (localhost)
  - For remote cFS targets the build & deploy processes must ensure consistency

Topic IDs (Deployment Model?)



Is there a mission ID? No

Each spacecraft has an ID
Single spacecraft deployment with potentially multiple CPUs
All CPUs belong to s/c

Platform = Category of hardware & OS e.g. Raspberry Pi
Can have more than one processor that shares a platform
Each processor has an ID
Platform tuning for each platform
Old school: Force each CPU to be its own platform configuration

- *Topic IDs are a combination of Message IDs with a CPU instance ID*
- *Multiple instances of the same app on a CPU is not supported*

Basecamp Topic ID Management

- **Topic IDs are defined cfe-topicids.xml**
  - `<Define name="APP_C_DEMO_CMD_TOPICID" value="${CFE_MISSION/TELECOMMAND_BASE_TOPICID} + 28"/>`

- **App EDS spec's use this value to initialize their topic ID**
  - `<Variable type="BASE_TYPES/uint16" readOnly="true" name="CmdTopicId" initialValue="${CFE_MISSION/APP_C_DEMO_CMD_TOPICID}" />`
  - ${} values are populated by the EDS toolchain

- **Basecamp adds a topicid tool to the cFS make system that is invoked using 'make topicids'**
  - This tool reads topic ID values from cfe-topicids.xml and populates values in the app JSON init configuration files
  - app_c_demo_ini.xml: "config": { … "APP_C_DEMO_CMD_TOPICID": 6236, …}

cFS EDS Conventions

- **EDS is a standard however the cFS EDS definitions and toolchain add conventions on how the standard is used**
  - These slides identify categories of conventions but do itemize them
  - When writing a new app, copying an existing app's EDS spec is the easiest way to get started

- **The following files (listed on the previous EDS directory slide) define cFS constants and types that should be used**
  - base_types.xml, ccsds_spacepacket.xml, cfe_hdr.xml
  - app_c_fw.xml definitions should be used if you are writing Basecamp style apps

- **The EDS toolchain creates C headers and writes them into cfs-basecamp/cfe-eds-framework/build/inc**

- **The derivation of names in C are often a combination of infomration from the EDS**
  - For example demo app's no opperation command code in app_c_demo_eds_cc.h is #define APP_C_DEMO_NOOP_CC
  - This name comes from the EDS Package name APP_C_DEMO, ContainerDataType name="Noop“ and the toolchain's hardcoded '_CC' suffix convention

Basecamp EDS Conventions

- **Basecamp command payload container type names are defined as [CommandName]_CmdPayload and the command container type names are [CommandName]**
  - This differs from the NAsA naming convention command container type names as [CommandName]Cmd
  - The Cmd suffix is not used because its redundant with in the context of a complete EDS command string that has the format [app]/Application/CMD/[CommandName]. This occurs because one Topic Id is used for many commands that are uniquely referenced using command function codes
- **Basecamp telemetry payload container type names are defined as [TelemetryName]Tlm_Payload and the telemetry container type names are [TelemetryName]Tlm**
- **EDS is an app level specification so EDS names do not follow Basecamp's object-based design naming standards**
  - Type definitions are prefixed with the app name and are not refined to the object level
- **Add #include "<app>_eds_typedefs.h" to app_cfg.h to make EDS defined types available to every apppbject**
  - Global type definition inclusion increases object coupling and reduces information hiding
- **#include "<app>_eds_cc.h" in app's main c file**

Section 4

Framework-based App Design

Introduction

- **The basecamp C Application Framework is light-weight object-based framework for writing cFS applications in C**
  - The framework library is named app_c_fw which will be used as this document’s shorthand notation

- **What does object-based mean?**
  - Applications are a composition of objects where an object is the bundling of data and functions (aka methods) that implement a single concept that is identified by the object’s name
  - Coding idioms implement the object oriented (OO) concepts rather than trying to create artificial OO constructs implemented in C
  - Even enforcing a couple of software engineering principles** such as the Single Responsibility and Open/Closed principles can result in significant improvements

- **The demo app APP_C_DEMO is a non-trivial app that is part of the default cFS target**
  - Uses many of app_c_fw’s features
  - This guide uses it as a reference app implementation to illustrate how app_c_fw is used

Motivation

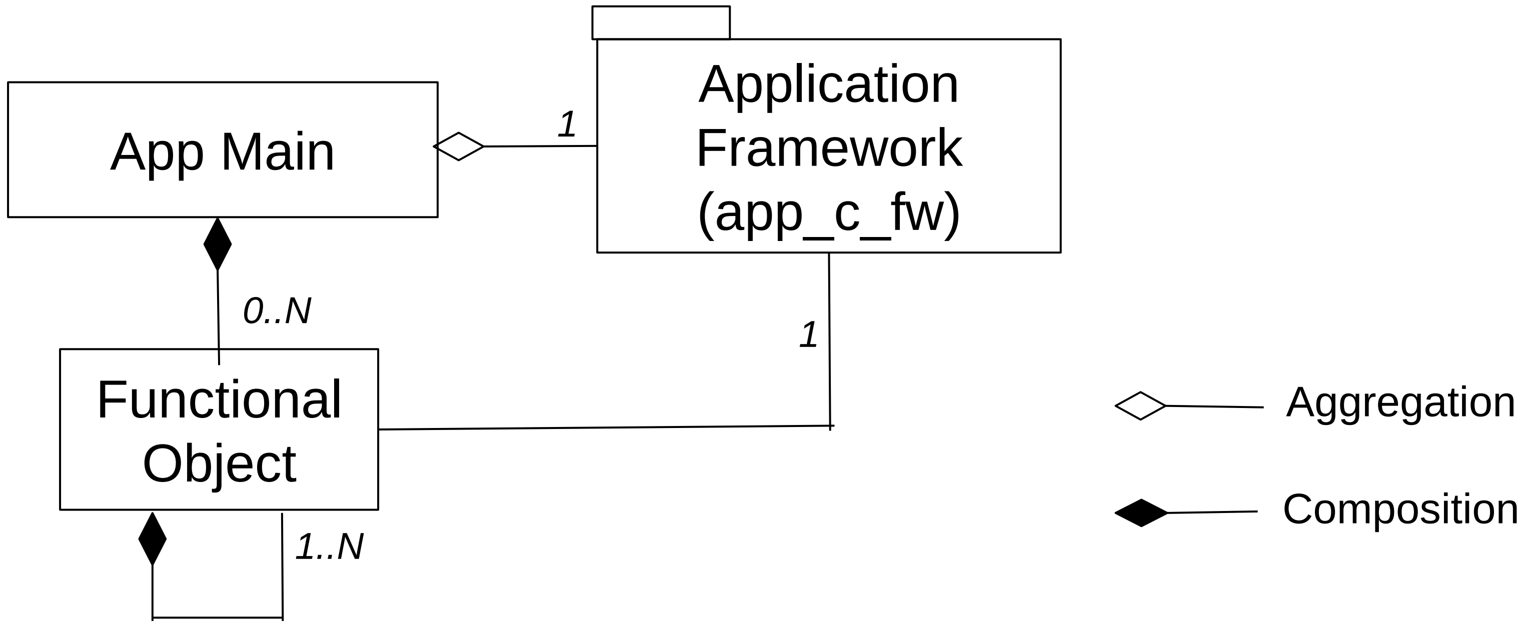
- **Since the cFS is a message-based system many apps have a common control and data flow structure**

- **A common object-based framework helps enforce a modular design that has many benefits**
  - Increased code reuse across apps which increases reliability and reduces testing
  - Common app structure reduces learning curve when adopting new apps and simplifies maintenance
  - The framework supports the app features/interfaces required by the Basecamp app package specification which allows apps to be published and exchanged

- **Decreases coupling and increases cohesion**
  - They are not easy to measure/verify and often reveal themselves during maintenance so when you make a change observe how the change is manifested
    - Is it localized?
    - How many components are impacted?
    - Are details encapsulated behind an API?
  - See File Manager app’s documentation refactor analysis section for how Basecamp’s design conventions can improve these attributes

Framework-based Application Architecture

Here’s the top-level application design represented in Unified Modeling Language (UML)



- **Aggregation** represents a relationship where the contained object (unfilled diamond connector) can exist independent of the owner
  - Each app ‘has a’ an app_c_fw

- **Composition** represents a relationship where the contained cannot exist independent of the owner
  - functional objects exists to provide behavior and functionality and they only exist within the context of the application

Functional Objects

- **Functional objects implement mission requirements**
  - App behavior and functions

- **Apps are composed of 0 to N functional objects**
  - These objects are organized in a tree-like structure

- **An app with no functional objects is the notorious "Hello World" app described in Section 1**

Functional Object Design

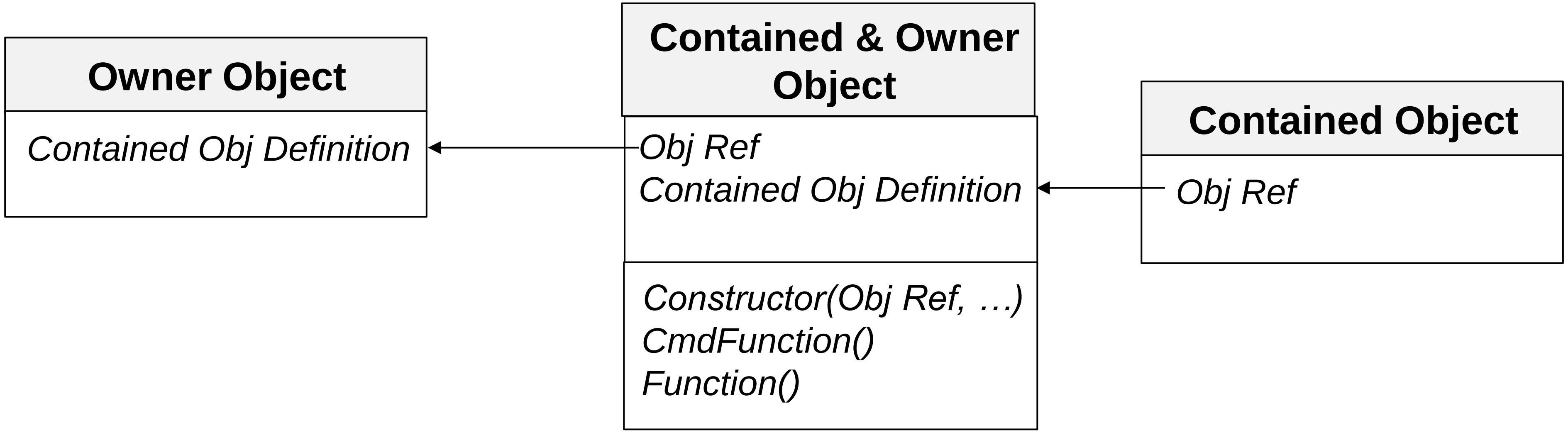
- **An objected oriented design analysis approach called Class-Responsibility-Collaboration (CRC)[1] can be helpful when designing an app's functional objects**
  - The descriptions below use the analogy of building a modular home with construction blocks
  - **Class[2]**
    - Defines the type of construction block like a window, door, roof, etc.
  - **Responsibility**
    - Defines what each construction block should do
    - A window should let in light, optionally open/close, etc.
  - **Collaboration**
    - What other construction blocks are needed
    - A window block collaborates with a wall block
  - The Application Framework Library can be thought of as supplying services like water and electricity

- **If you can easily name your classes then you've probably decomposed the design space into reasonable components**
  - The class properties (data) and methods(functions) should be intrinsic to the scope and responsibilities of the class

1. Class-Responsibility-Collaboration Card - GeeksforGeeks
2. Basecamp uses objects which will be explained in a few slides

Object Composition Relationships

- **The figure below shows the object composition model**



- **Owner objects define the data for objects they contain and pass a reference to the contained object's constructor**
- **Contained objects store a reference to the owner's instance data**
  - Analogous to the object oriented Singleton[1] design pattern without any wrapper protection
  - Typically only one object exists within an app

[1] Design Pattern - Singleton Pattern

Object Design Conventions (1 of 2)

- **Public object functions (or methods) fall into two categories**
  - Command functions are executed when the parent app receives a command message on the software bus that contains the function’s command function code
    - Command functions are registered by the main app during initialization
  - All other public functions are called by the main app or by other objects during their execution
  - Both types of functions may execute within the app’s context or an app child task context
  - Command functions are part of the app’s public message interface
  - Non-command public functions define the object’s public interface within an app

- **Objects can generate telemetry messages**

- **Objects can create Software Bus pipes**
  - An app’s control flow is determined by all of the SB pipes

Object Design Conventions (2 of 2)

- **Relative event message ID numbering is used within each object**
  - Ranges of IDs used by each object are managed at the application level

- **Table objects only manage a JSON parameter table and do not contain other objects**
  - They are covered in the Table Manager section

- **The App Object model balances simplicity with ‘design space’ coverage**
  - Most apps can follow the basic design pattern, so the benefits of a common app design and reuse are realized, however developers should not feel constrained by the model if it doesn’t fit a particular situation

- **See Appendix B for Object coding conventions**

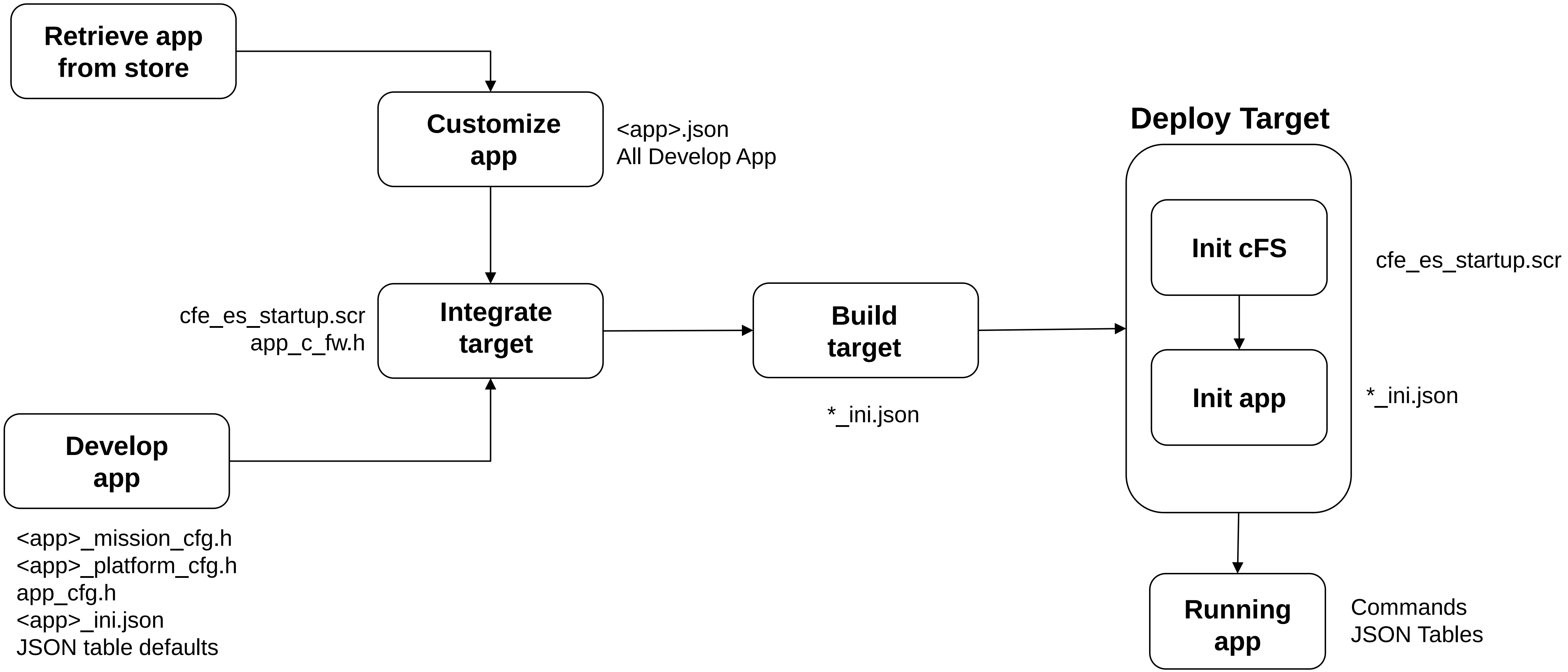
Configuration Strategy

- **Configuration options are an important sotware reusability attribute**
  - Basecamp augments the cFS options to facilitate it’s app store workflow

- **Configuration parameter attributes**
  - Scope: Range of influence
  - Workflow timeframe: When the parameter is set and used in the workflow

- **The next slide identifies configuration file relationships with the Basecamp cFS target workflow**
  - This slide is followed by slides that define scope and purpose of each configuration file

Workflow & Configuration Files



Configuration File Scope & Purpose (1 of 2)

Configuration	Scope & Purpose
cfe_es_startup.scr	Target-scoped file that specifies libraries and apps that are loaded by the cFE during initialization. The file can be customized during the target integration activity and is used during the target
app_c_fw_cfg.h	Defines platform-scoped configuration parameters for the framework. The defaults should accommodate most deployments. The configurations must meet the needs of all apps on a platform sharing the framework.
<app>.json	Used by App Store apps to define cFS target integration parameters that include the default cfe_es_startup.src entry.
<app>_mission_cfg.h	Defines mission-scoped application configurations. These configurations apply to every app deployment on different platforms within a single mission. Analogous to cFS app mission config header in scope. Only contains parameters that must be defined during compilation, otherwise they should be in app’s JSON init file.
<app>_platform_cfg.h	Defines platform-scoped application configurations. Analogous to cFS app platform config header in scope. Only contains parameters that must be defined during compilation, otherwise they should be in app’s app_cfg.h or JSON init file.

Configuration File Scope & Purpose (2 of 2)

Configuration	Scope & Purpose
app_cfg.h	Every Basecamp app has a header with this name. Configurations have an application scope that define parameters that shouldn't need to change across deployments. If they need to change across deployments then they should be in mission/platform config files.
<app>_ini.json	Defines configuration parameters that can be defined at runtime. For example, command pipe name, command pipe depth, and command message identifier.
Commands & JSON Tables	The decision whether to define parameters in a table versus as command parameters has multiple factors including how the parameter is used by the app in its processing and on the operational scenarios that may dictate the need for variations in the parameter. This is discussed in discussed in the app_c_demo description.

Framework Overview

Framework Overview

- **The Application Framework is a cFS library called APP_C_FW that provides common app services and utilities.**

- **Framework services are designed as objects which means they have state information that is stored in a structure.**
  - The structure typedef naming convention is X_Class_t where X is the name of the service.
  - App Main objects declare an instance of the service object and a pointer to this instance is passed as the first parameter of service functions.
  - If an App's Functional Object requires access to a service, then the App Main object passes its instance pointer to the functional object.

- **Framework services must be coded as reentrant because they are shared by all the basecamp apps.**
  - Reentrant coding is a technique that allows a function to be Interrupted and called again by another execution thread before it finishes executing with the first thread.

Application Framework Services

Component	Source File	Description
Initialization Table	inittbl	Reads a JSON file containing key-value definitions and provides functions for accessing these values
Command Manager	cmdmgr	Provides a command registration service and manages dispatching commands
Core JSON	cjson	Provides an interface to the FreeRTOS coreJSON library that simplifies managing and parsing JSON files. Table Objects use CJSON for managing JSON parameter table files.
Table Manager	tblmgr	Provides a table registration service and manages table loads and dumps
Child Task Manager	childmgr	Provides a framework allowing commands and callback functions to execute within a child task
State Reporter	staterep	Manages the generation of a periodic telemetry packet that contains Boolean flags. Provides and API for app objects to set/clear states. Often useful to aggregate fault detection flags into a single packet that can be monitored by another application.

Each service is documented in later sections

Framework Utilities

- **Utilities are collections of related functions**
  - They are stateless and only operate on function parameters

- **Fileutil provides utilities that simplify file management**

– FileUtil_GetFileInfo() is a commonly used function that validates the filename string and returns the file state as either invalid, nonexistent, open, closed or it's a directory. See FILE_MGR for example calls.

- **Pktutil provides general utilities used with command and telemetry packets**

– For example, the KIT_TO app uses PktUtil_IsPacketFiltered() to determine whether a telemetry packet should be output.

JSON Initialization Configuration File Service

Initialization Configuration Introduction

- **JSON initialization configuration files define application deployment configurations**
  - If a configuration parameter impacts a data structure, then it must be defined in a header file at the appropriate scope

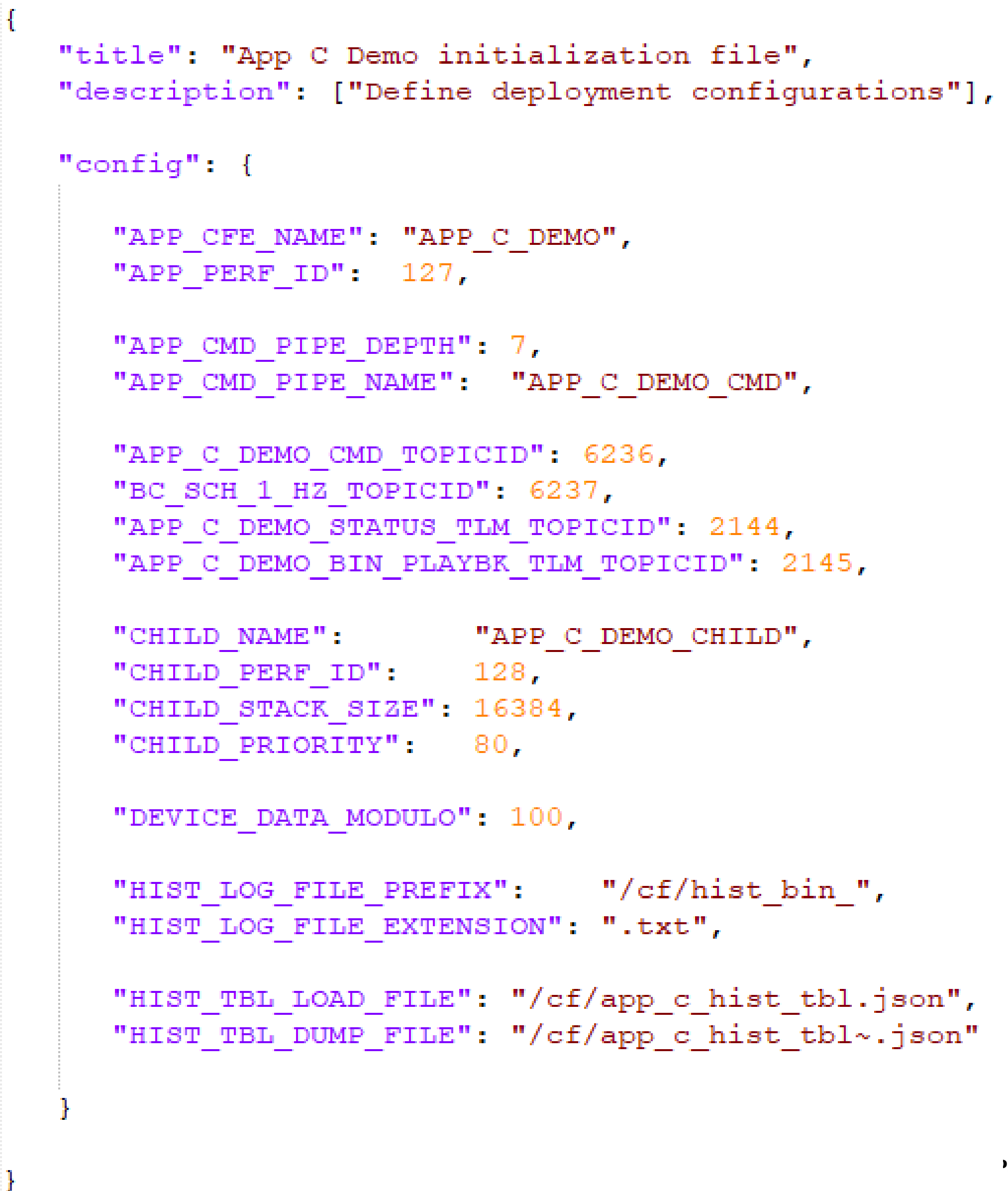
- **Advantages of using JSON files read during initialization include**
  - Text files are human and computer friendly
  - Separate tables can be defined in the “_defs” directory for each CPU target
  - Tools to manipulate the files can easily be written since JSON has wide language support
  - In a running system, an app can be restarted with a new table

- **Basecamp’s “make topicds” tool replaces message ID initialization configurations with the EDS-defined message ID values**

Initialization Configuration File

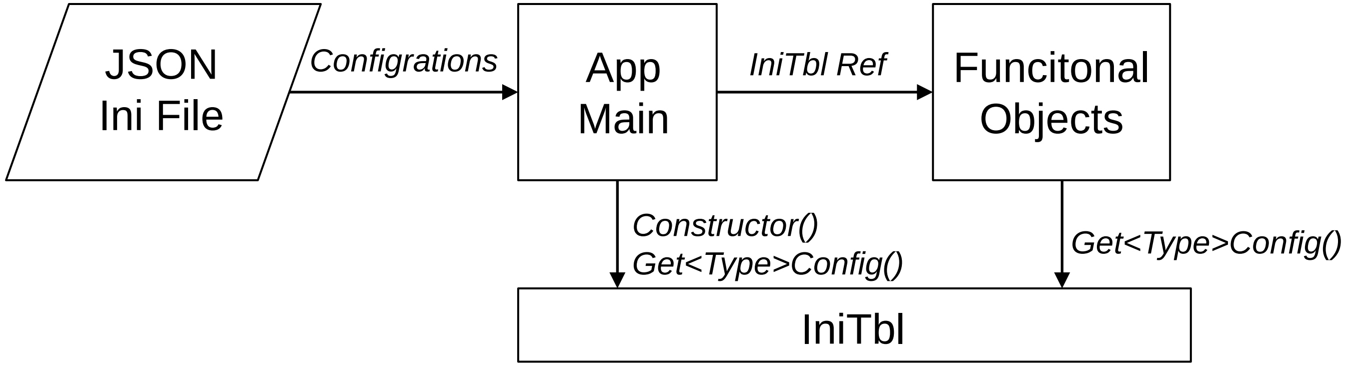
- **File is read in during application initialization**
  - JSON table filename is defined in app's xxx_platform_cfg.h
- **Description is an array of strings that is used for parameter comments**
- **"config" JSON object contains the key-value pair definitions**
- **Keys are defined in each app's app_cfg.h**
- **Supports 32-bit unsigned integers, floats and strings**
- **Topic Identifiers (*_TOPICID) are defined in this file. The names must match the names used in the Electronic Data Sheet file that defines the project's topic IDs.**

app_c_demo_ini.json



```
{
   "title": "App C Demo initialization file",
   "description": ["Define deployment configurations"],

   "config": {

      "APP_CFE_NAME": "APP_C_DEMO",
      "APP_PERF_ID":  127,

      "APP_CMD_PIPE_DEPTH": 7,
      "APP_CMD_PIPE_NAME":  "APP_C_DEMO_CMD",

      "APP_C_DEMO_CMD_TOPICID": 6236,
      "BC_SCH_1_HZ_TOPICID": 6237,
      "APP_C_DEMO_STATUS_TLM_TOPICID": 2144,
      "APP_C_DEMO_BIN_PLAYBK_TLM_TOPICID": 2145,

      "CHILD_NAME":       "APP_C_DEMO_CHILD",
      "CHILD_PERF_ID":    128,
      "CHILD_STACK_SIZE": 16384,
      "CHILD_PRIORITY":   80,

      "DEVICE_DATA_MODULO": 100,

      "HIST_LOG_FILE_PREFIX":    "/cf/hist_bin_",
      "HIST_LOG_FILE_EXTENSION": ".txt",

      "HIST_TBL_LOAD_FILE": "/cf/app_c_hist_tbl.json",
      "HIST_TBL_DUMP_FILE": "/cf/app_c_hist_tbl~.json"

   }

}
```

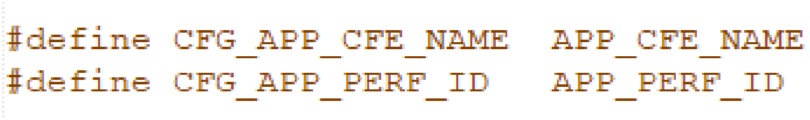
JSON Initialization Architecture



The following slides describe the coding steps for implementing a JSON initialization table

Define Initialization Parameters (1 of 2)

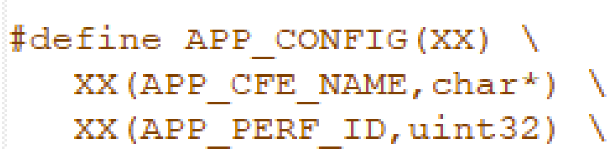
1a. Define configurations in app_cfg.h



```
#define CFG_APP_CFE_NAME   APP_CFE_NAME
#define CFG_APP_PERF_ID    APP_PERF_ID
```

• • •

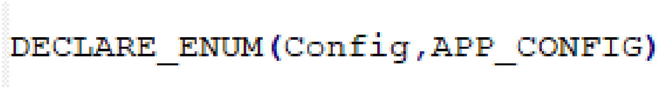
Define macros using the naming CFG_XXX, where XXX is the same name used in the JSON initialization file



```
#define APP_CONFIG(XX) \
   XX(APP_CFE_NAME,char*) \
   XX(APP_PERF_ID,uint32) \
```

• • •

Add the XXX definition to APP_CONFIG macro
Declare types as uint32, float or char*

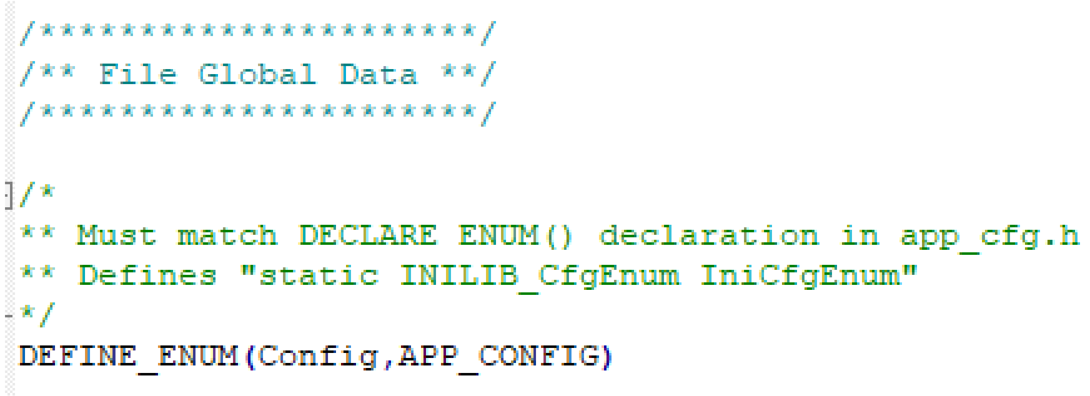


```
DECLARE_ENUM(Config,APP_CONFIG)
```

Declare parameter enumerations using DECLARE_ENUM()

Define Initialization Parameters (2 of 2)

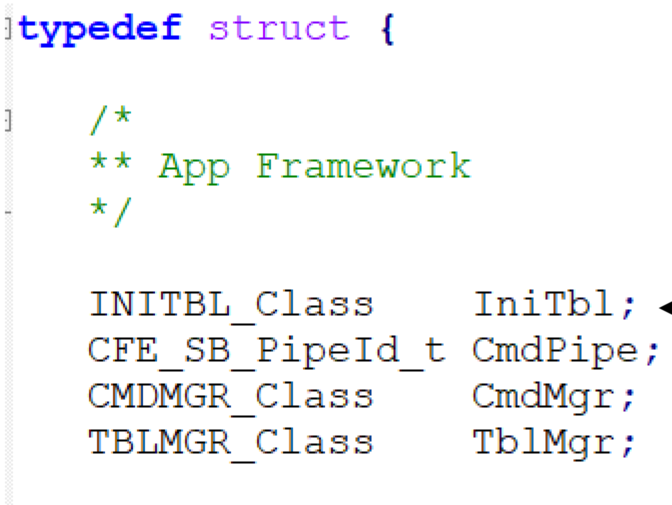
1b. App Main File: Define the initializations parameter enumerations



```
/**********************/
/** File Global Data **/
/**********************/

/*
** Must match DECLARE ENUM() declaration in app_cfg.h
** Defines "static INILIB_CfgEnum IniCfgEnum"
*/
DEFINE_ENUM(Config,APP_CONFIG)
```

The user doesn’t need to know the details

1c. App Main File: Define IniTbl object in app’s main class



```
typedef struct {

   /*
   ** App Framework
   */

   INITBL_Class    IniTbl;
   CFE_SB_PipeId_t CmdPipe;
   CMDMGR_Class    CmdMgr;
   TBLMGR_Class    TblMgr;
```

IniTbl Definition

Use the App Initialization Table

2a – Construct INITBL in the app's initialization function

```
INITBL_Constructor(&AppCDemo.IniTbl, APP_C_DEMO_INI_FILENAME, &IniCfgEnum)
```

2b – Retrieve parameter values using CFG_XXX macro and `INITBL_Get<Type>Config()`

```
CFE_SB_CreatePipe(&AppCDemo.CmdPipe, INITBL_GetIntConfig(INITBL_OBJ, CFG_APP_CMD_PIPE_DEPTH),
                  INITBL_GetStrConfig(INITBL_OBJ, CFG_APP_CMD_PIPE_NAME));
```

Notes

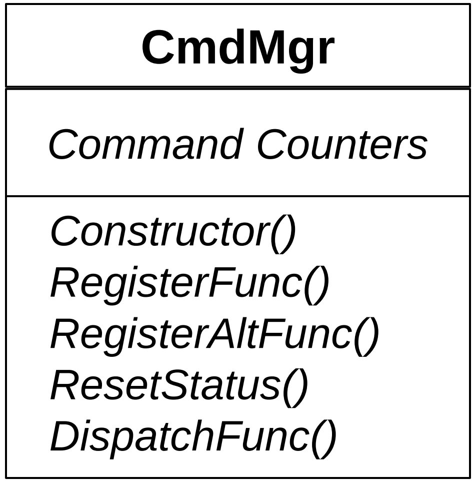
– The app'sJSON filename must be added to the "FILELIST' in targets.cmake

– If a parameter is used in multiple locations create storage for it at the most local scope possible and initialize the storage in the appropriate constructor function. See app_c_demo's performance ID.

– Since message IDs are variables, a switch statement with message ID cases statements. An if-else construct will be needed. See app_c_demo_app.c's ProcessCommands().

Command Management Service

Command Manager Overview



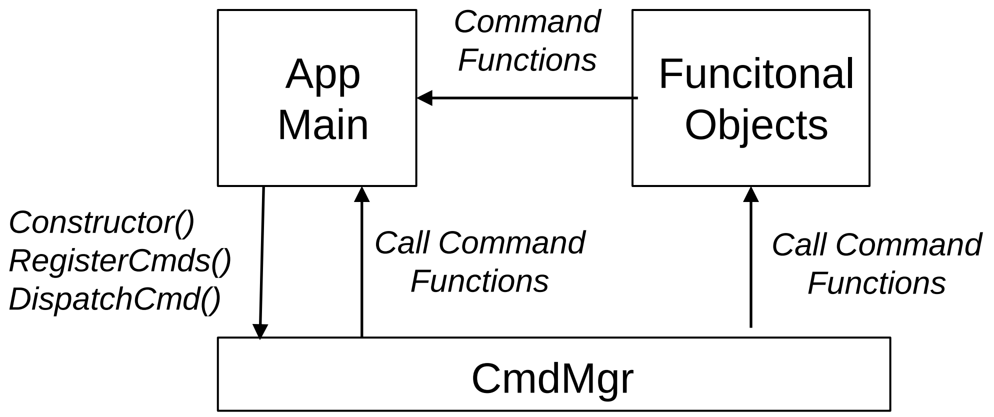
CmdMgr

Command Counters

Constructor()
RegisterFunc()
RegisterAltFunc()
ResetStatus()
DispatchFunc()

- **Provides a command registration service and manages dispatching commands**
- **Performs command length and checksum validations prior to calling the registered command**
  - App developers focus on implementing and testing app functionality
- **Supports “alternate” command concept that means the command counters are not incremented**
  - Useful when onboard commands are sent between apps and incrementing the command counters could confuse ground operation’s monitoring
- **Does not manage the SB command pipe calls**
  - Allows the app to determine whether to poll or pend on the command pipe
  - Keeps CmdMgr’s role and responsibilities focused on command validation and dispatching

Command Manager Architecture



The next slides describe the coding steps for instantiating and using Command Manager

Main App Coding Steps to use Command Manager

1. **Define a CmdMgr object in the app's class structure**

```
CMDMGR_Class     CmdMgr;
```

2. **Construct the CmdMgr object in the app's init function**

```
CMDMGR_Constructor(CMDMGR_OBJ);
```

3. **Register commands in the app's init function**

```
CMDMGR_RegisterFunc(CMDMGR_OBJ, OSK_C_DEMO_TBL_LOAD_CMD_FC,
                    TBLMGR_OBJ, TBLMGR_LoadTblCmd, TBLMGR_LOAD_TBL_CMD_DATA_LEN);
```

4. **Dispatch commands in the app's SB command pipe processing**

```
if (MsgId == OskCDemo.CmdMid) {
   CMDMGR_DispatchFunc(CMDMGR_OBJ, CmdMsgPtr);
}
```

5. **Reset CmdMgr in the app's reset command processing**

```
CMDMGR_ResetStatus(CMDMGR_OBJ);
```

Notes

- **The app_c_fw EDS defines function codes for common app commands**
  - Noop, Reset and optional Table Load/Dump

- **Every app should have a noop command**

- **CmdMgr maintains valid and invalid command counters**
  - These should be in routine telemetry so operators can verify command execution
  - Ground scripyts can perform autonomous command verification
  - Command counters are incremented after a command function executes, the effect of a command may need to be verified later. For example, a spacecraft slew command may be initially process successfully but the slew could still fail

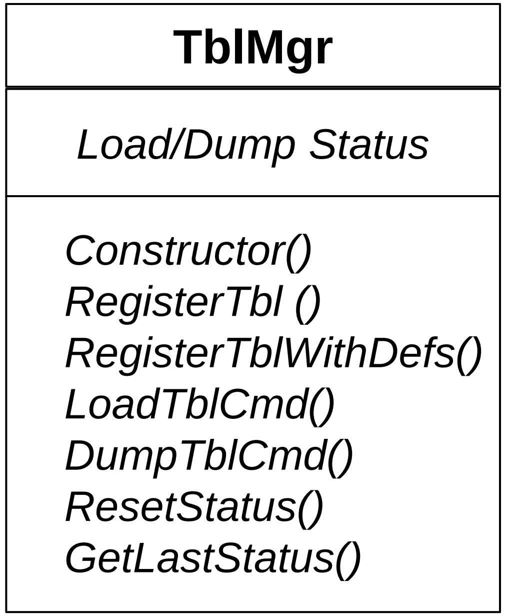
JSON Parameter Table Service

JSON Parameter Tables

- **Objectives**
  - Provide a text-based table service
  - Create a consistent application JSON table management operational interface
  - Comply with basecamp's object-based application Facilitate consistent application designs that abstract complexities, minimize application developer learning curves and simplify maintenance

- **Rationale**
  - cFS binary tables require an added layer of ground processing for translating between binary tables and human readable/writable text
  - JSON is a popular language for defining configuration parameters that is supported by popular scripting languages and only requires a text editor for reading/writing

Table Manager Overview



TblMgr

Load/Dump Status

Constructor()
RegisterTbl ()
RegisterTblWithDefs()
LoadTblCmd()
DumpTblCmd()
ResetStatus()
GetLastStatus()

- **Tables are defined in JSON text files**
- **Tables are parsed using the FreeRTOS core-JSON parser**
- **Provides a table registration service and manages loads and dumps**
- **The main app’s initialization function registers the *LoadTblCmd()* and *DumpTblCmd()* with Command Manager**

Table Manager Architecture

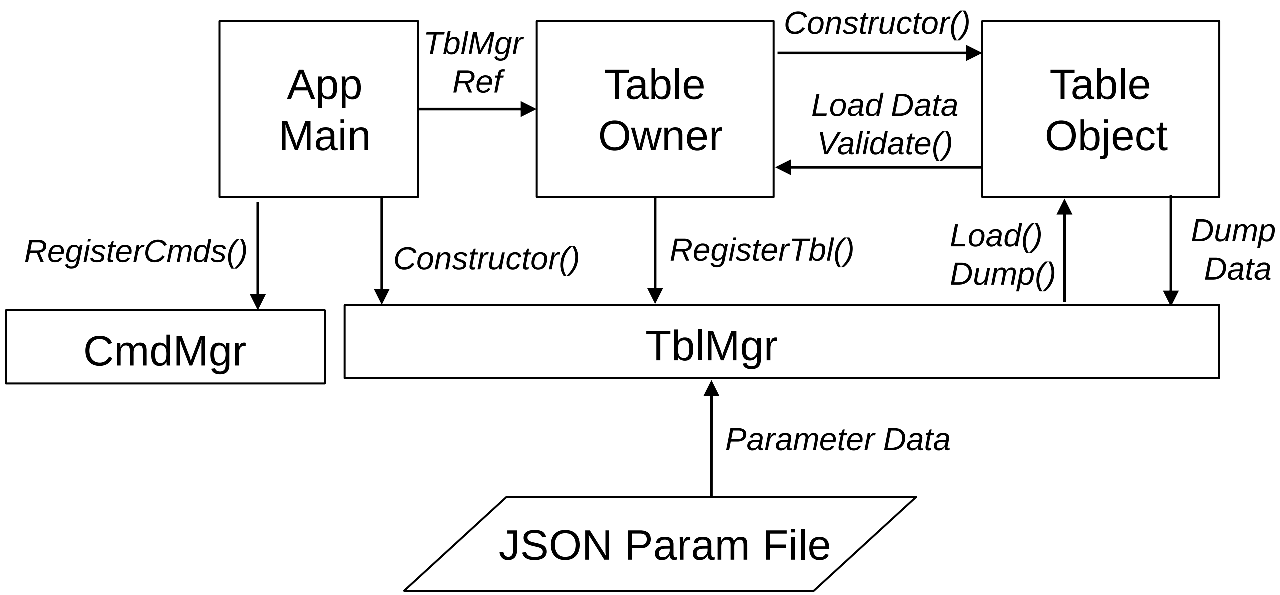
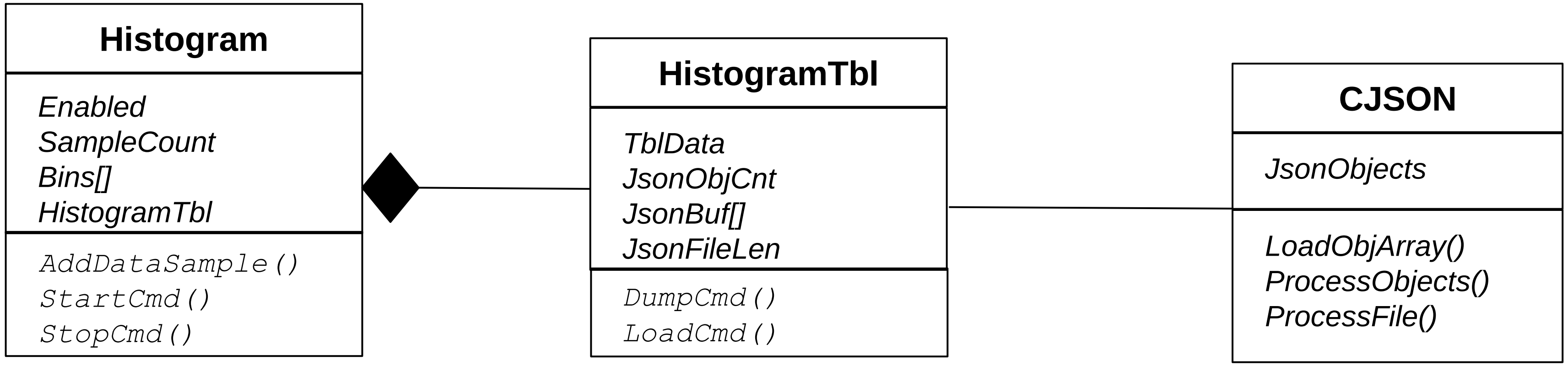


Table Manager Object Design



- ***Histogram* is the parent of *HistogramTbl* so it contains an instance of *HistogramTbl***
- ***HistogramTbl***
  - *TblData* stores table load data and its contents are copied to *Histogram's* instance if the table load is successful
  - *LoadCmd()* and *DumpCmd()* are TblMgr callback functions that control the load/dump processes. They are registered with *TblMgr* by the app's init function
  - *LoadJsondata()* is a callback function used by CJSON__ProcessFile() that copied data from the JSON file into *TableLoadDataBuf*
- ***CJSON provides a simple API for using CORE-JSON to manage tables***
  - *CJSON manages the JSON files and CORE-JSON works with character buffers*
  - *ProcessFile()* validates the JSON file and calls the user supplied callback function to copy data into it's table load buffer
- **A separate table object is defined for each JSON parameter table**

Demo App Table Example (1 of 5)

Main App Object

app_c-demo.h

Declare a Table Manager object in the class structure:

```
   TBLMGR_Class_t    TblMgr;
```

app_c-demo.c

Optionally define the following macro to improve readability since a pointer to the TblMgr object is frequently referenced.

```
   #define  TBLMGR_OBJ   (&(AppCDemo.TblMgr))
```

In the app's initialization function perform the following steps:

1. Construct the table manager object

```
   TBLMGR_Constructor(TBLMGR_OBJ, INITBL_GetStrConfig(INITBL_OBJ, CFG_APP_CFE_NAME));
```

2. Construct the object that owns the table.

```
   HISTOGRAM_Constructor(HISTOGRAM_OBJ, INITBL_OBJ, TBLMGR_OBJ);
```

3. Register the table load and dump command functions. Note that the table manager's functions are registered and not the table object's functions. Table commands contain a table ID parameter so the table manager can call the appropriate table object function. Table IDs are assigned when the owner of a table object registers the table with the table manager.

```
   CMDMGR_RegisterFunc(CMDMGR_OBJ, APP_C_DEMO_LOAD_TBL_CC, TBLMGR_OBJ, TBLMGR_LoadTblCmd,
                       sizeof(APP_C_DEMO_LoadTbl_CmdPayload_t));
   CMDMGR_RegisterFunc(CMDMGR_OBJ, APP_C_DEMO_DUMP_TBL_CC, TBLMGR_OBJ, TBLMGR_DumpTblCmd,
                       sizeof(APP_C_DEMO_DumpTbl_CmdPayload_t));
```

Demo App Table Example (2 of 5)

Main App Object (cont)

app_c-demo.c (cont)

4. Call table manager’s reset function in the app’s reset command function.

```
TBLMGR_ResetStatus(TBLMGR_OBJ);
```

5. Optionally send table status in telemetry. Telemetry content and rates are tuned for each mission. The demo app includes table manager’s last attempted action and the status of that action. If an app has more than one table `TBLMGR_GetLastTblStatus()` returns the status of the last table operated upon.

```
const TBLMGR_Tbl_t *LastTbl = TBLMGR_GetLastTblStatus(TBLMGR_OBJ);
APP_C_DEMO_StatusTlm_Payload_t *Payload = &AppCDemo.StatusTlm.Payload;

Payload->LastTblAction       = LastTbl->LastAction;
Payload->LastTblActionStatus = LastTbl->LastActionStatus;
```

Demo App Table Example (3 of 5)

Table Owning Object

histogram.h

1. Declare a Table object in the class structure:

```
HISTOGRAM_TBL_Class_t  Tbl;
```

histogram.c

2. In the constructor, construct the table object and register it with the table manager. Pass the object table's load and dump command functions. These are called by the table manager's load and dump commands that are registered with Command Manager by the main app object. This example loads default table values from a file the which is typically done during registration.

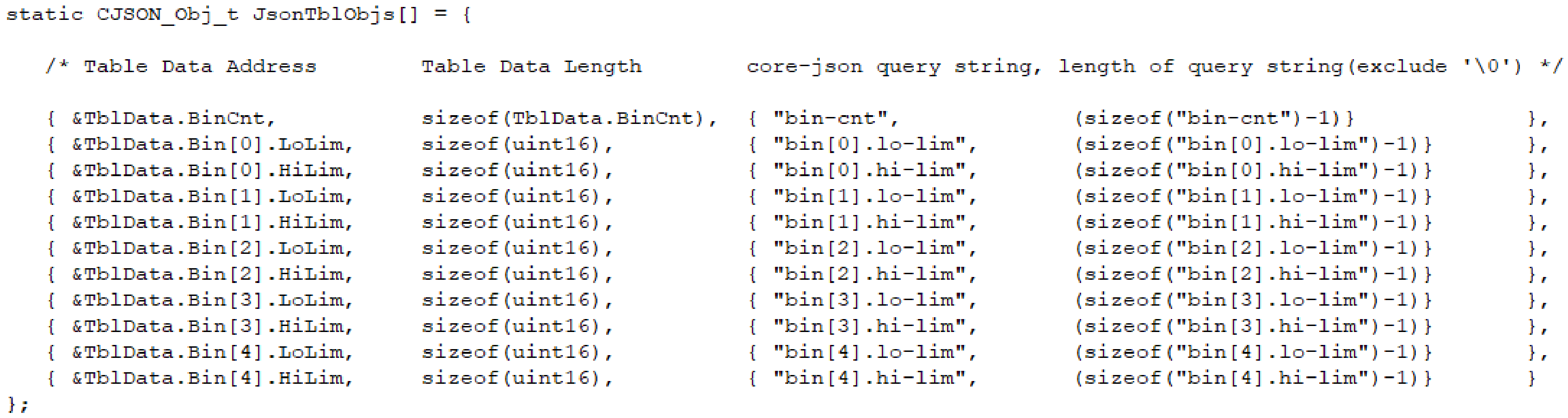
```
HISTOGRAM_TBL_Constructor(&Histogram->Tbl, AcceptNewTbl);

TBLMGR_RegisterTblWithDef(TblMgr, HISTOGRAM_TBL_NAME,
                          HISTOGRAM_TBL_LoadCmd, HISTOGRAM_TBL_DumpCmd,
                          INITBL_GetStrConfig(IniTbl, CFG_HIST_TBL_LOAD_FILE));
```

Demo App Table Example (4 of 5)

Table Object

histogramtbl.c

The load process is governed by a data structure that simplifies customization. Here are the key fields of demo app's table load data structure:



```
static CJSON_Obj_t JsonTblObjs[] = {

   /* Table Data Address        Table Data Length        core-json query string, length of query string(exclude '\0') */

   { &TblData.BinCnt,           sizeof(TblData.BinCnt),  { "bin-cnt",             (sizeof("bin-cnt")-1)}             },
   { &TblData.Bin[0].LoLim,     sizeof(uint16),          { "bin[0].lo-lim",       (sizeof("bin[0].lo-lim")-1)}       },
   { &TblData.Bin[0].HiLim,     sizeof(uint16),          { "bin[0].hi-lim",       (sizeof("bin[0].hi-lim")-1)}       },
   { &TblData.Bin[1].LoLim,     sizeof(uint16),          { "bin[1].lo-lim",       (sizeof("bin[1].lo-lim")-1)}       },
   { &TblData.Bin[1].HiLim,     sizeof(uint16),          { "bin[1].hi-lim",       (sizeof("bin[1].hi-lim")-1)}       },
   { &TblData.Bin[2].LoLim,     sizeof(uint16),          { "bin[2].lo-lim",       (sizeof("bin[2].lo-lim")-1)}       },
   { &TblData.Bin[2].HiLim,     sizeof(uint16),          { "bin[2].hi-lim",       (sizeof("bin[2].hi-lim")-1)}       },
   { &TblData.Bin[3].LoLim,     sizeof(uint16),          { "bin[3].lo-lim",       (sizeof("bin[3].lo-lim")-1)}       },
   { &TblData.Bin[3].HiLim,     sizeof(uint16),          { "bin[3].hi-lim",       (sizeof("bin[3].hi-lim")-1)}       },
   { &TblData.Bin[4].LoLim,     sizeof(uint16),          { "bin[4].lo-lim",       (sizeof("bin[4].lo-lim")-1)}       },
   { &TblData.Bin[4].HiLim,     sizeof(uint16),          { "bin[4].hi-lim",       (sizeof("bin[4].hi-lim")-1)}       }
};
```

The first two columns specify where the JSON data is loaded, address and length, and the last two columns are used by the core JSON parser to retrieve the data.

Demo App Table Example (5 of 5)

Table Object (cont)

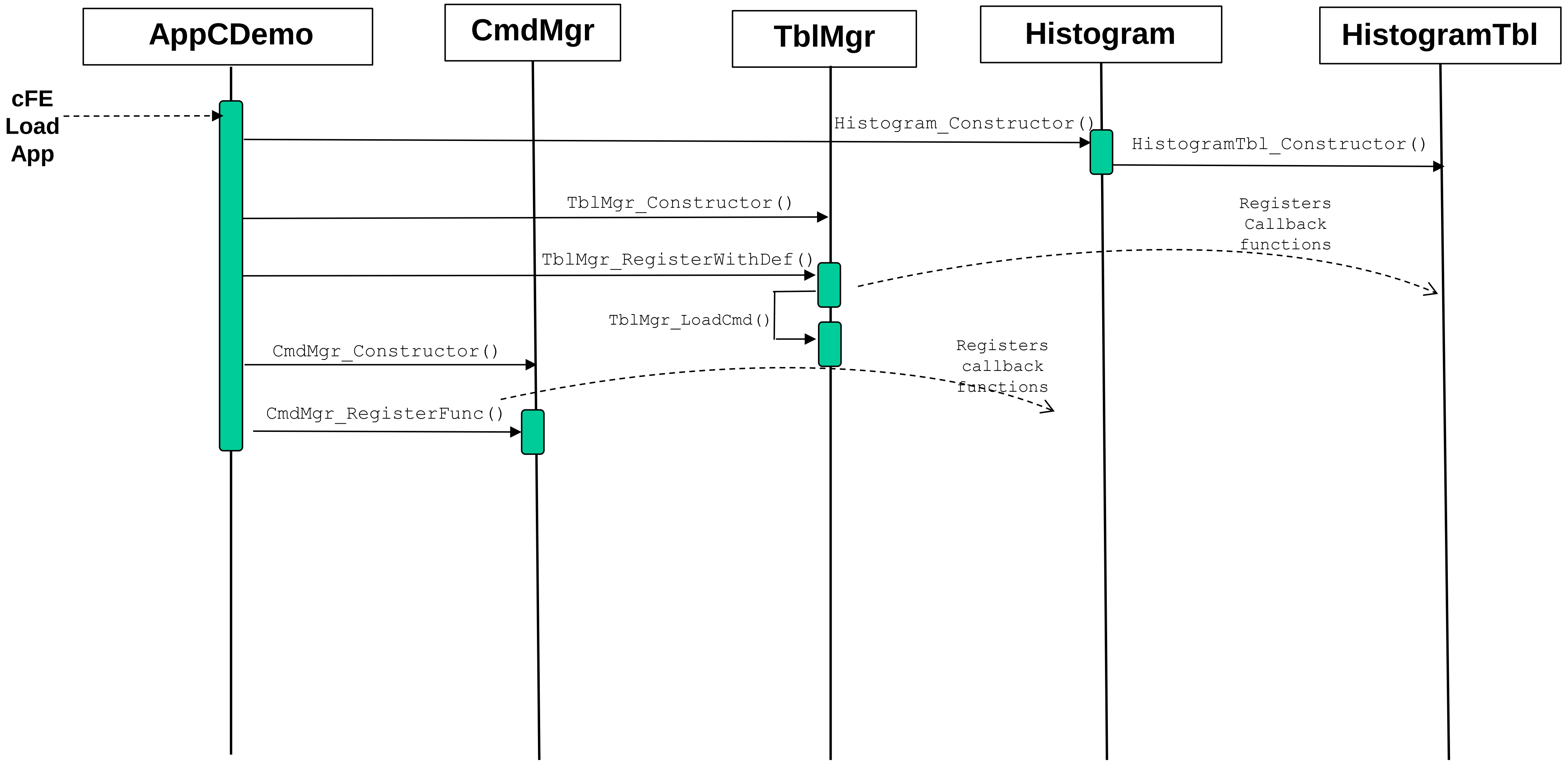
histogramtbl.c (cont)

Developers don’t have to directly invoke the JSON parser. Basecamp’s app framework’s CJSON object reduces the app developer’s code to the following function calls:

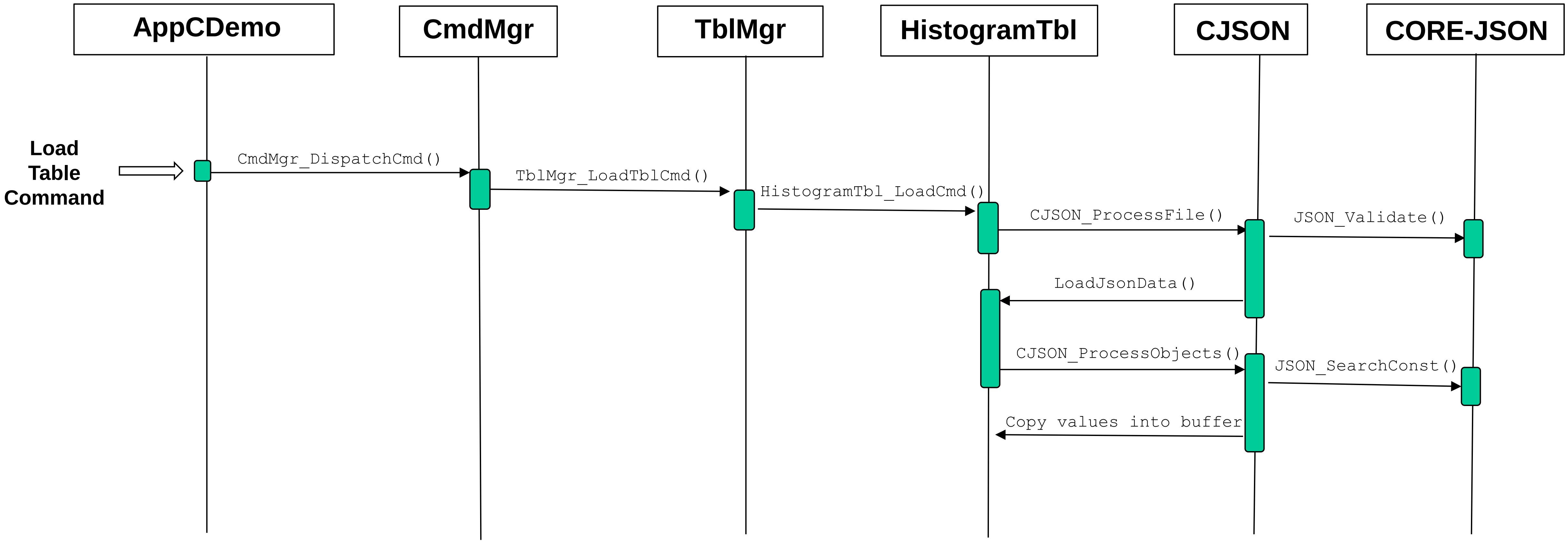
```
CJSON_ProcessFile(Filename, HistogramTbl->JsonBuf, HISTOGRAM_TBL_JSON_FILE_MAX_CHAR, LoadJsonData);
CJSON_LoadObjArray(JsonTblObjs, HistogramTbl->JsonObjCnt, HistogramTbl->JsonBuf, HistogramTbl->JsonFileLen);
```

CJSON_ProcessFile() validates the JSON and reads the contents into memory. The last parameter. LoadJsonData , is a pointer to a function and in this function is a call to CJSON_LoadObjArray(). This function processes the JSON from memory and loads the data into the table object using the JsonTblObjs[] described above.

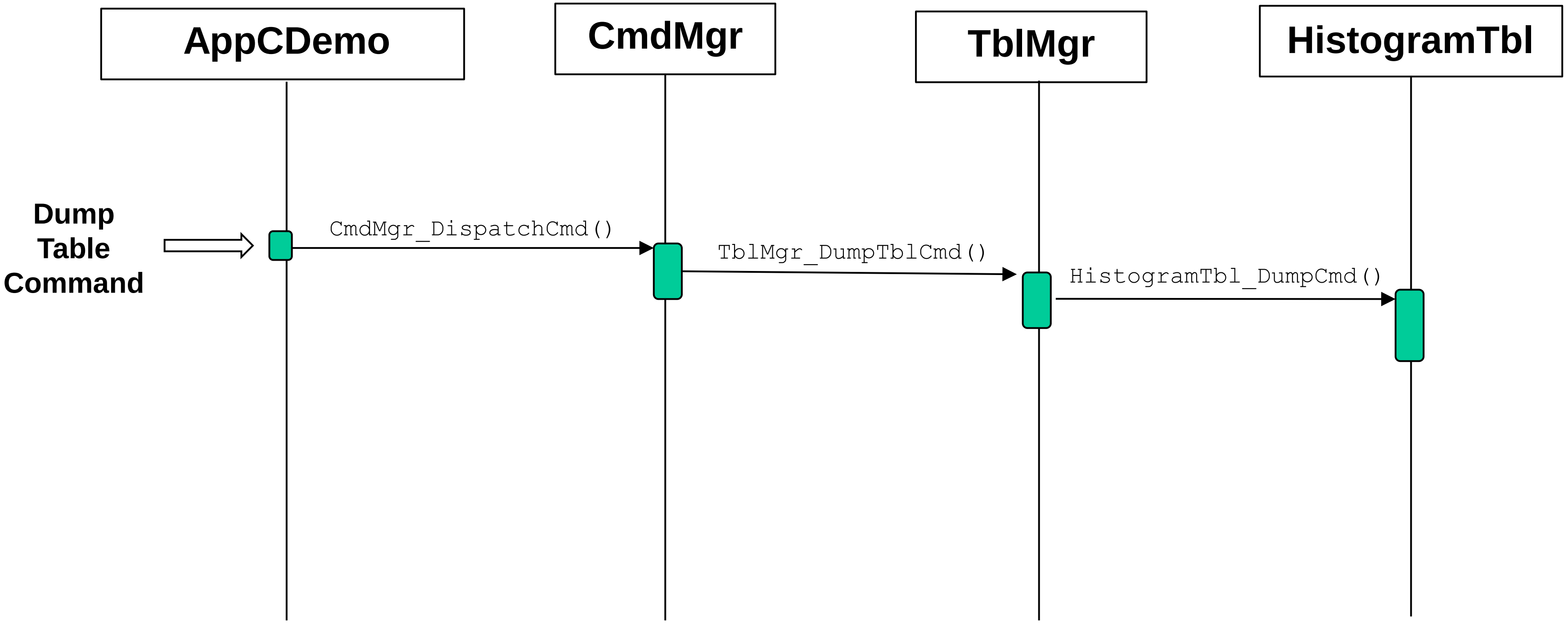
Table Initialization Sequence Diagram



Load Table Sequence Diagram



Dump Table Sequence Diagram



TODO

- Add JSON array example from KIT_SCH or KIT_TO
- Describe KIT_SCH and KIT_TO table load strategy combined with a command interface to load and dump individual array items
- Error handling conventions
  - Do not start the app if errors loading ini file definitions
  - Do start the app if a parameter table fails to load with the idea that the table could be loaded because the app is still functional at least from a basic running state so the parameter table can be loaded.
- EDS table name enumeration convention. Can't parameterize enum in app_c_fw EDS
- Expand on Hello Table design notes
- Operational procedures can also minimize risks. At Goddard tables are loaded and tested at FlatSat before they are loaded to the spacecraft. The concept of tables themselves carry risks because their contents can significantly impact the behavior of an app. For example, a value may cause a divide by zero.

- The best way to create a table object for your app is to copy an existing table object, change the global names and modify the load and dump functions. Note some apps like KIT_TO have commands that can modify the parameters that were originally loaded from a table. The table object owner should provide an interface for these commands because it owns the data that is being used.

- Apps may have more than one table. Table IDs are assigned based on the order of table registrations which is not the best design. See Basecamp Issue #103 Enhance table manager API · Issue #103 · cfs-tools/cfs-basecamp.

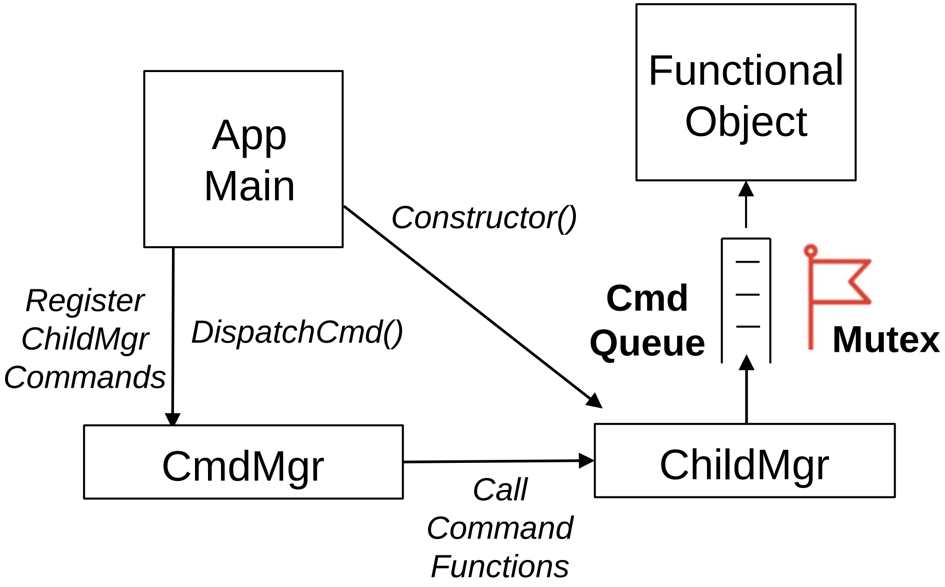
- Each app's JSON parameter table filename must be added to the "FILELIST' in targets.cmake.

Child Task Management Service

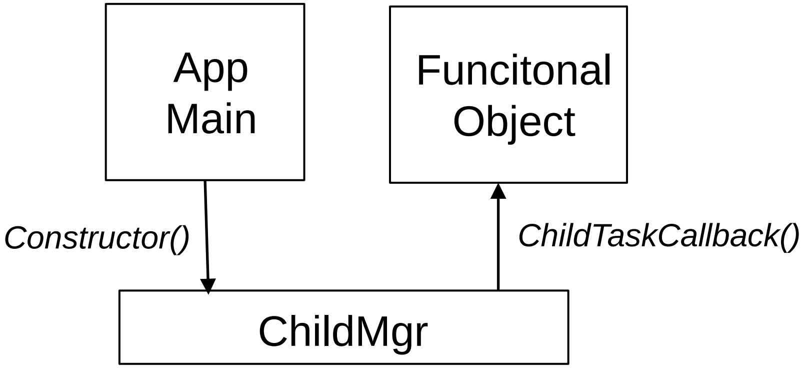
Child Task Manager Overview

- **Provides a common infrastructure for running contained objects within the context of a child task**
  - Balances ease of use, complexity, and scope of design problems that can be solved using the framework
  - It is <u>not</u> intended to provide a universal solution
- **Design considerations**
  - Main app should own the contained object that has functions that will run within a child task
  - App object functions running within a child task need to be designed with an awareness of how they manage utilizing the CPU and safely share data with the main task
- **Provides two mechanism for functions to run within a child task**
  1. Child task main loop pends indefinitely for commands
     - Note main app can send commands to perform child task functions synchronized with its execution
  2. Child task has an infinite loop that calls a user supplied callback function.
     - It is the callback function’s responsibility to periodically suspend execution

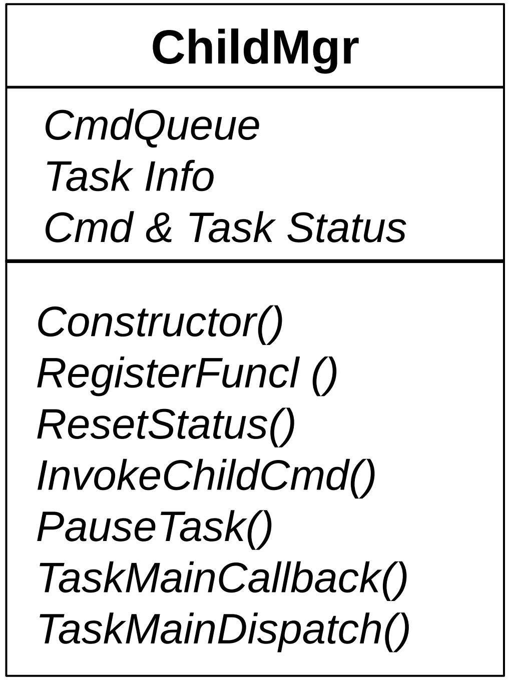
Child Task Processes Commands Architecture



Child Task Infinite Loop Architecture



Child Manager Functions



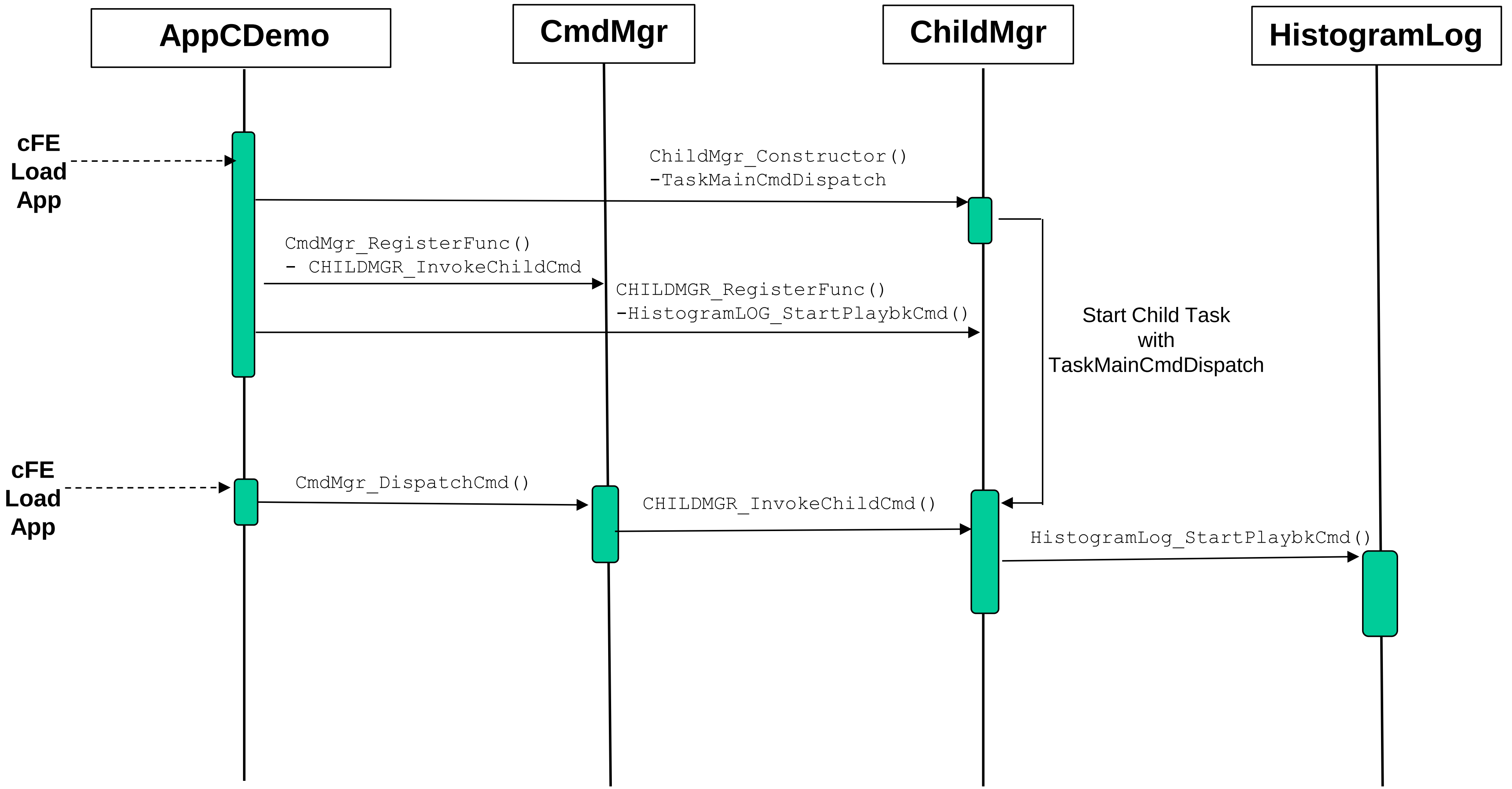
ChildMgr

CmdQueue
Task Info
Cmd & Task Status

Constructor()
RegisterFuncl ()
ResetStatus()
InvokeChildCmd()
PauseTask()
TaskMainCallback()
TaskMainDispatch()

- **Constructor()**
  - Creates child task and mutex semaphore for parent-child shared data
  - Configures main child task for command dispatch or infinite loop
- **RegisterFunc()**
  - Registers a command function
- **ResetStatus()**
  - Sets valid and invalid command counters to zero
- **InvokeChildCmd()**
  - The main app registers this function as the command dispatch function for every command that is executed by the child task. It copies the SB message into the child task's command queue and indicates that a command needs to be processed.
- **PauseTask()**
  - A utility function that can be used by a child task loop to pause these child tasks every n'th time it is called.
- **TaskMainCallback()**
  - Child task infinite loop that calls a callback function that was supplied to the constructor
- **TaskMainDispatch()**
  - Child task infinite loo that pends on the Command Queue semaphore

HistogramLog Start Playback Sequence Diagram



TBD

- **Add Coding steps**
- Add ChildMgr framework to app
- **Add child task init table parameters**
- Constructor child task

Core JSON

Service

- **Core JSON (CJSON)**
  - Provides an interface to the FreeRTOS coreJSON library that simplifies managing and parsing JSON files.
  - Table Objects use CJSON for managing JSON parameter table files. cjson (the backend for table processing) could also be considered a utility, it has state information
- **See JMSG processing as a non-Basecamp framework example**

Section 5

Demo App Detailed Design

Introduction

- **Section 2 introduced Basecamp's Demo app and should be read prior to this section**
  - It provides functional descriptions that serve as the app's requirements

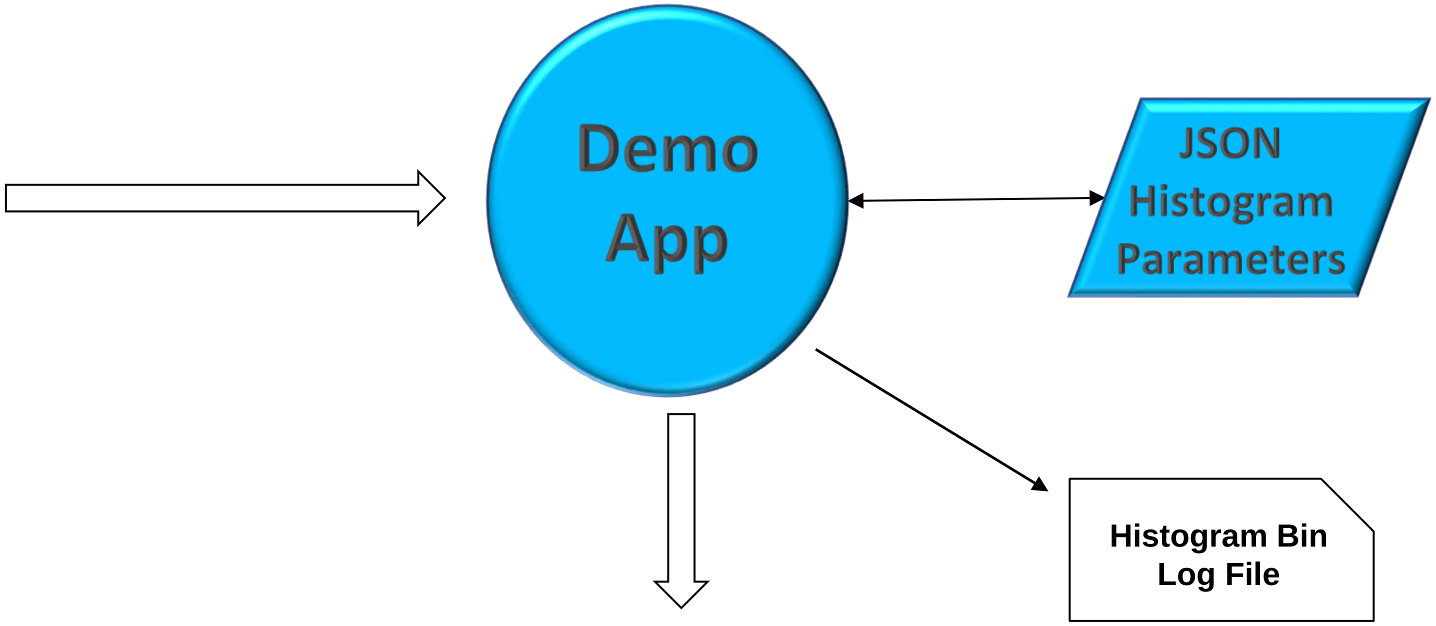
- **This section describes the Demo app's detailed design**
  - Appendix A describes Basecamp's design notation

- **The detailed design includes**
  - Context diagram
  - Activity diagram
  - Object diagrams
  - Sequence diagrams

Context Diagram

Commands

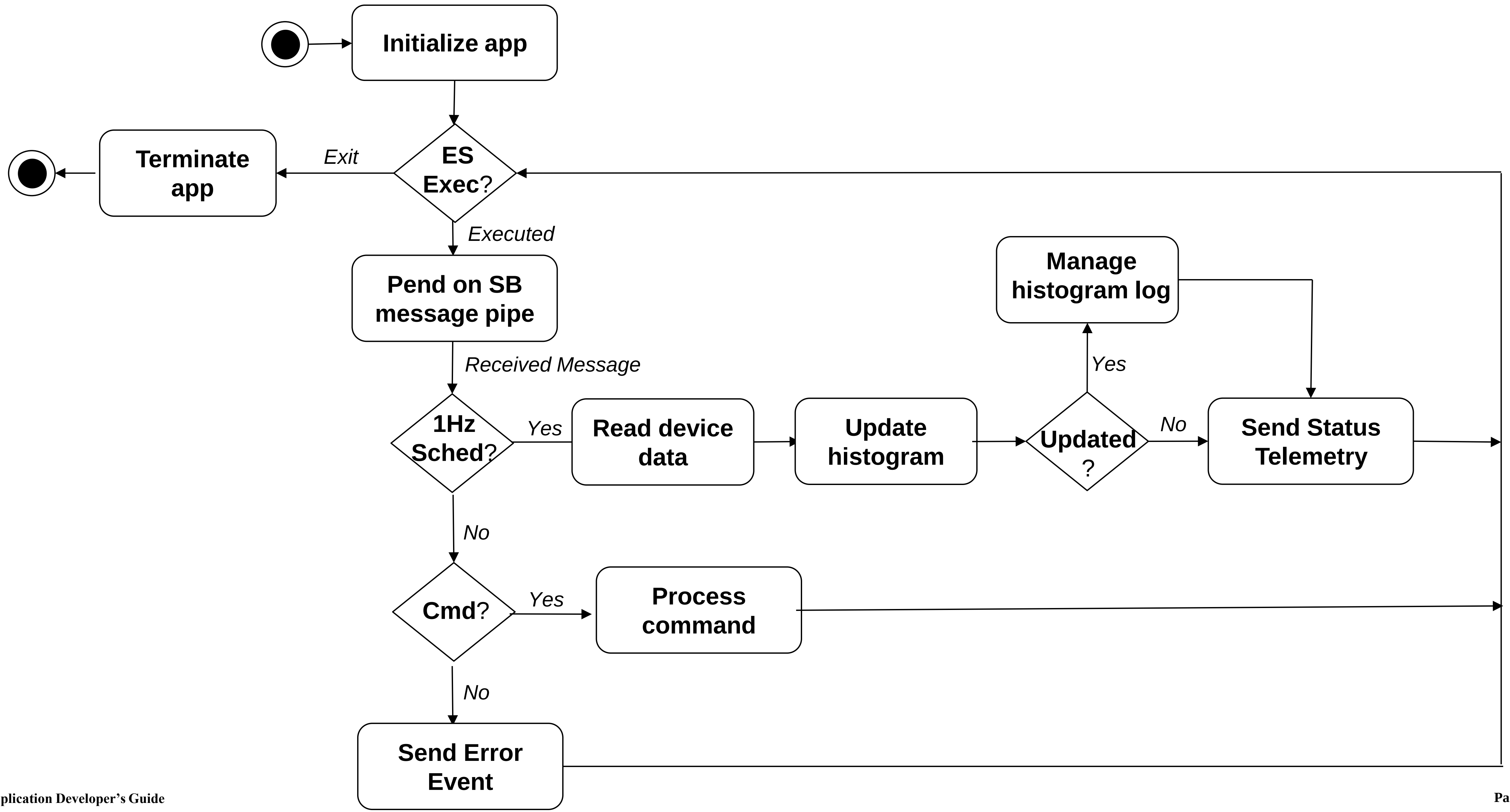
- Scheduler
  - 1 Hz
- Admin
  - Noop
  - Reset
  - Load Table
  - Dump Table
- Functions
  - Start Histogram
  - Stop Histogram
  - Start Histogram Log
  - Stop Histogram Log
  - Start Histogram Log Playback
  - Stop Histogram Log Playback



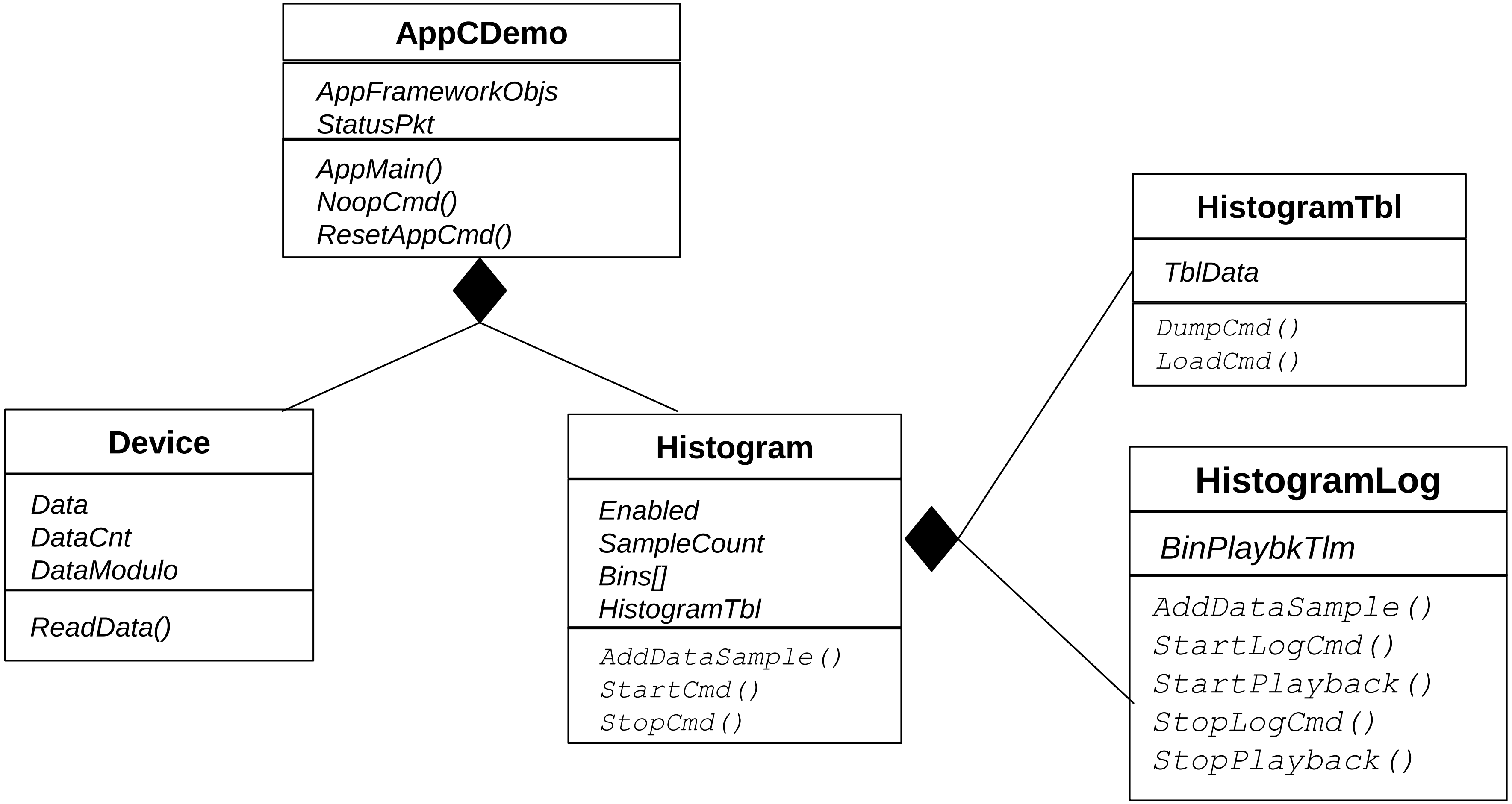
Telemetry

- Status
- Histogram Log File Playback

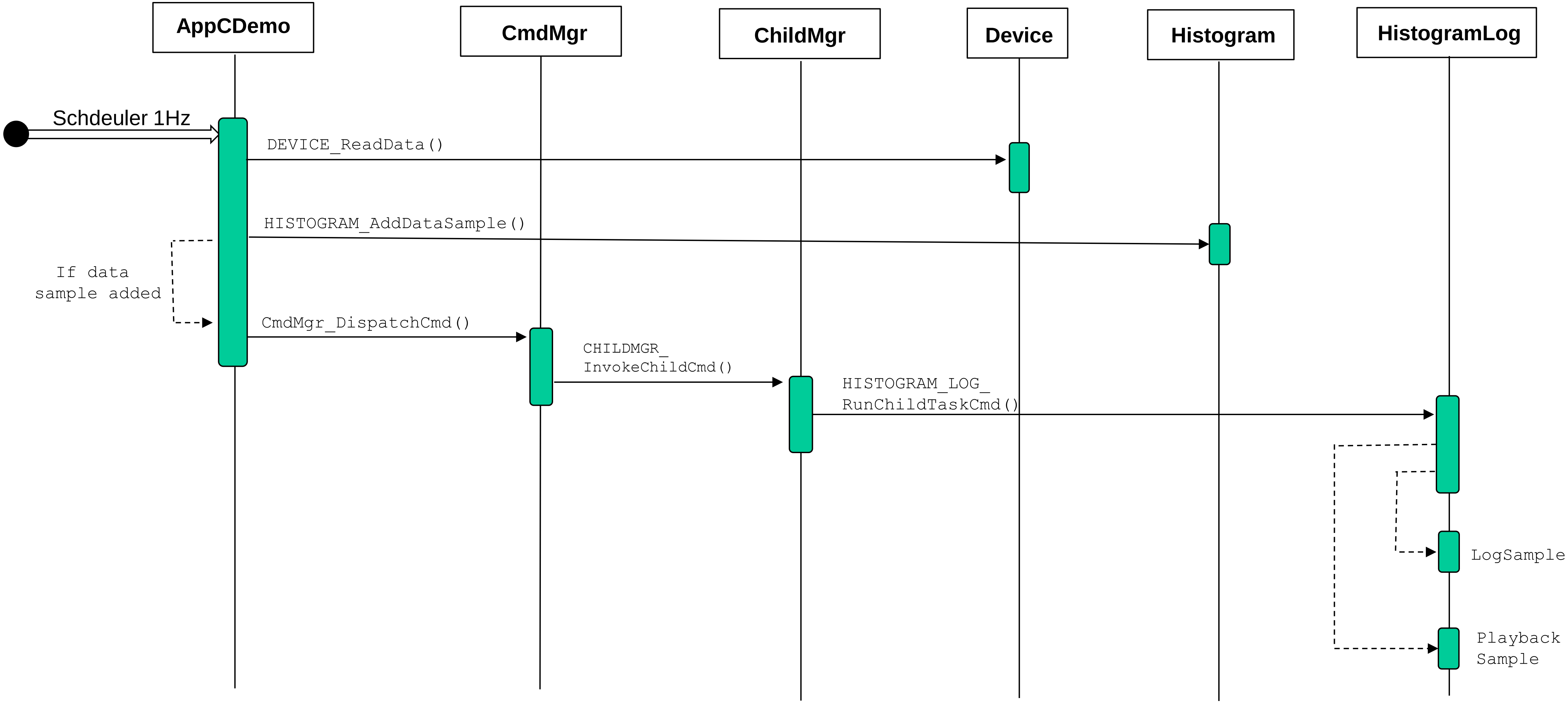
Demo App Main Control Flow



Demo App Object Diagram



1Hz Execution



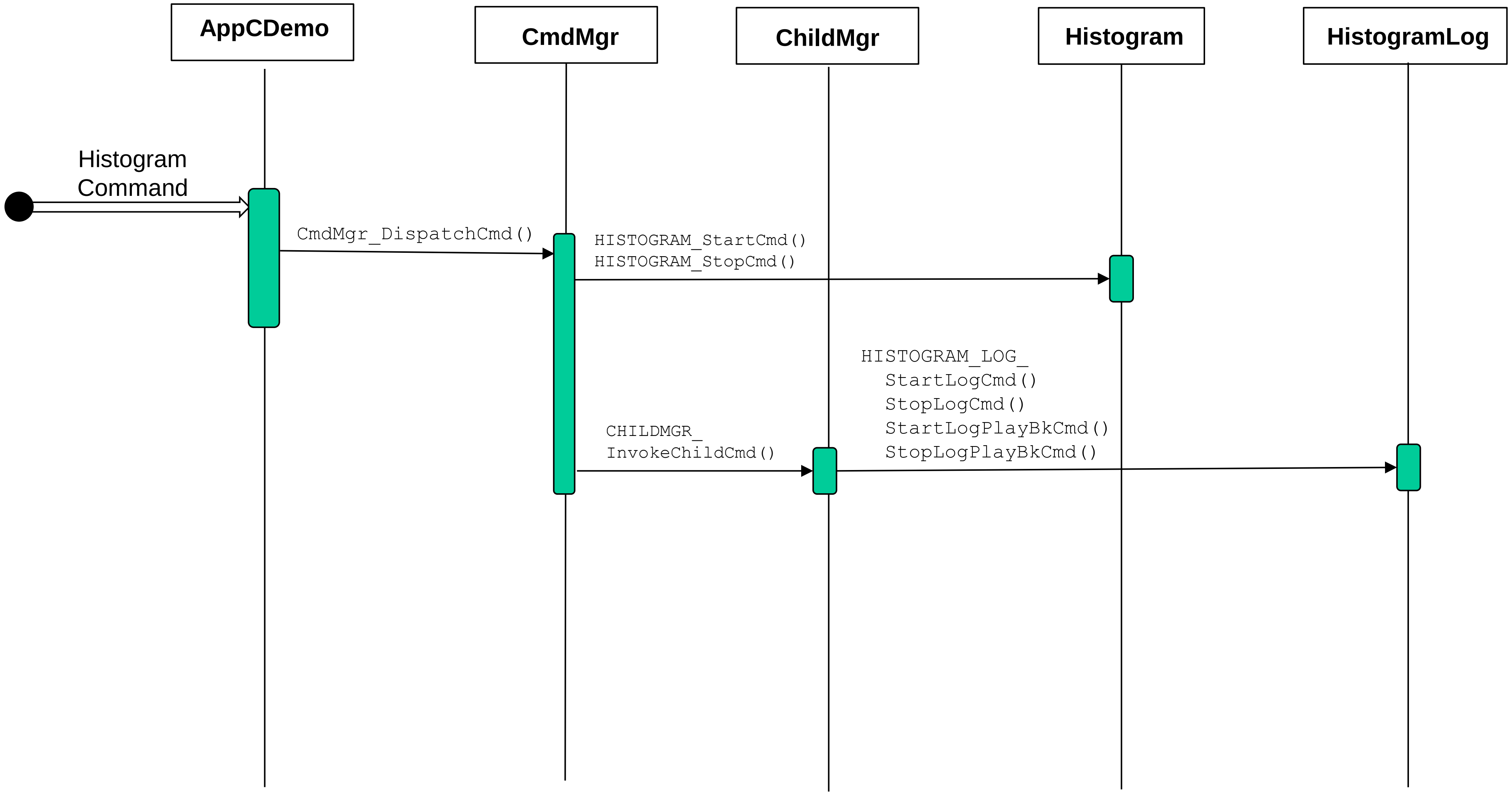
1Hz Execution Notes

- **Demo app uses a 1Hz scheduler app message to trigger its execution and process device data**
  - In a flight mission, this type of app's execution could be triggered by the device data
  - The *RUN_HISTOGRAM_LOG_CHILD_TASK _CC* command code is registered as an "alternate function" with CmdMgr which means the command counter will not be increment with each execution

- **Demo app's status message is sent at the app's execution frequency**
  - This allows every device data sample and histogram result to be viewed in realtime
  - In a flight mission, this may not be practical
  - Onboard data management and telemetry design is addressed in Basecamp's Systems Engineering course

- **Using a child task for histogram log activities is a little contrived since the histogram log processing is very short and doesn't need to execute a different priority than the main task**
  - The design shows how a periodic activity can be performed inn a child task using the CmdMgr and ChildMgr
  - Basecamp's File Manager and Raspberry Pi LED control (https://github.com/cfs-apps/rpi_led) apps illustrate two other child task designs

Fuctional Commands



Functional Command Notes

- **The HISTOGRAM_LOG commands are executed within the context of the child task**
  - They are registered with CmdMgr as “normal” commands (i.e. not alternative) so the valid/invalid command counters are incremented according to the command’s successful execution

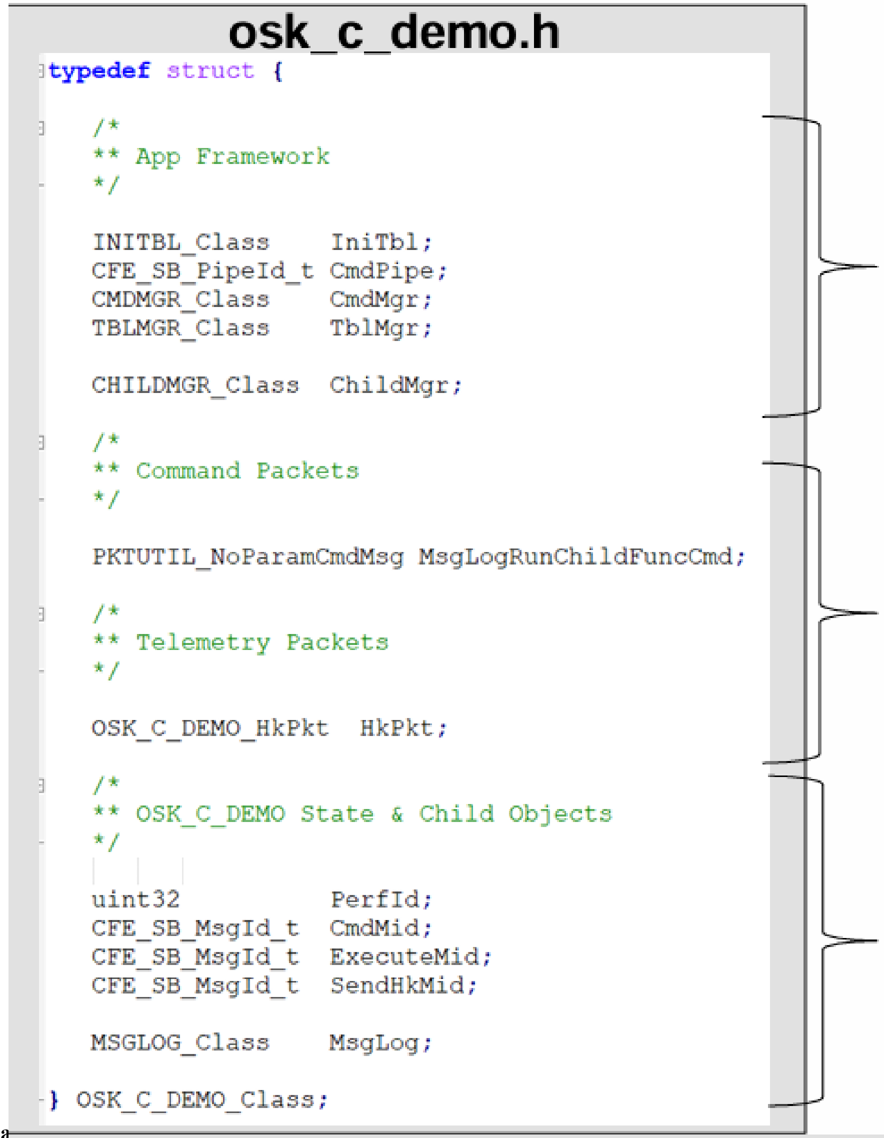
- **The simplicity of the sequence diagram shows how Basecamp’s framework and object-based designs encapsulate data & functions with well defined interfaces**

- **Demo app’s Reset command is not shown, but it should be noted that all of the functional object’s Reset() functions are called**
  - Each object resets its state data based on its current configuration
  - For example, if the histogram playback is not enabled then the playback count is reset to zero

STEM ware

Object Composition Model – Demo App

Basecamp



osk_c_demo.h

```
typedef struct {

   /*
   ** App Framework
   */

   INITBL_Class     IniTbl;
   CFE_SB_PipeId_t  CmdPipe;
   CMDMGR_Class     CmdMgr;
   TBLMGR_Class     TblMgr;

   CHILDMGR_Class   ChildMgr;

   /*
   ** Command Packets
   */

   PKTUTIL_NoParamCmdMsg MsgLogRunChildFuncCmd;

   /*
   ** Telemetry Packets
   */

   OSK_C_DEMO_HkPkt  HkPkt;

   /*
   ** OSK_C_DEMO State & Child Objects
   */

   uint32           PerfId;
   CFE_SB_MsgId_t   CmdMid;
   CFE_SB_MsgId_t   ExecuteMid;
   CFE_SB_MsgId_t   SendHkMid;

   MSGLOG_Class     MsgLog;

} OSK_C_DEMO_Class;
```

1. Instances of framework objects (components)

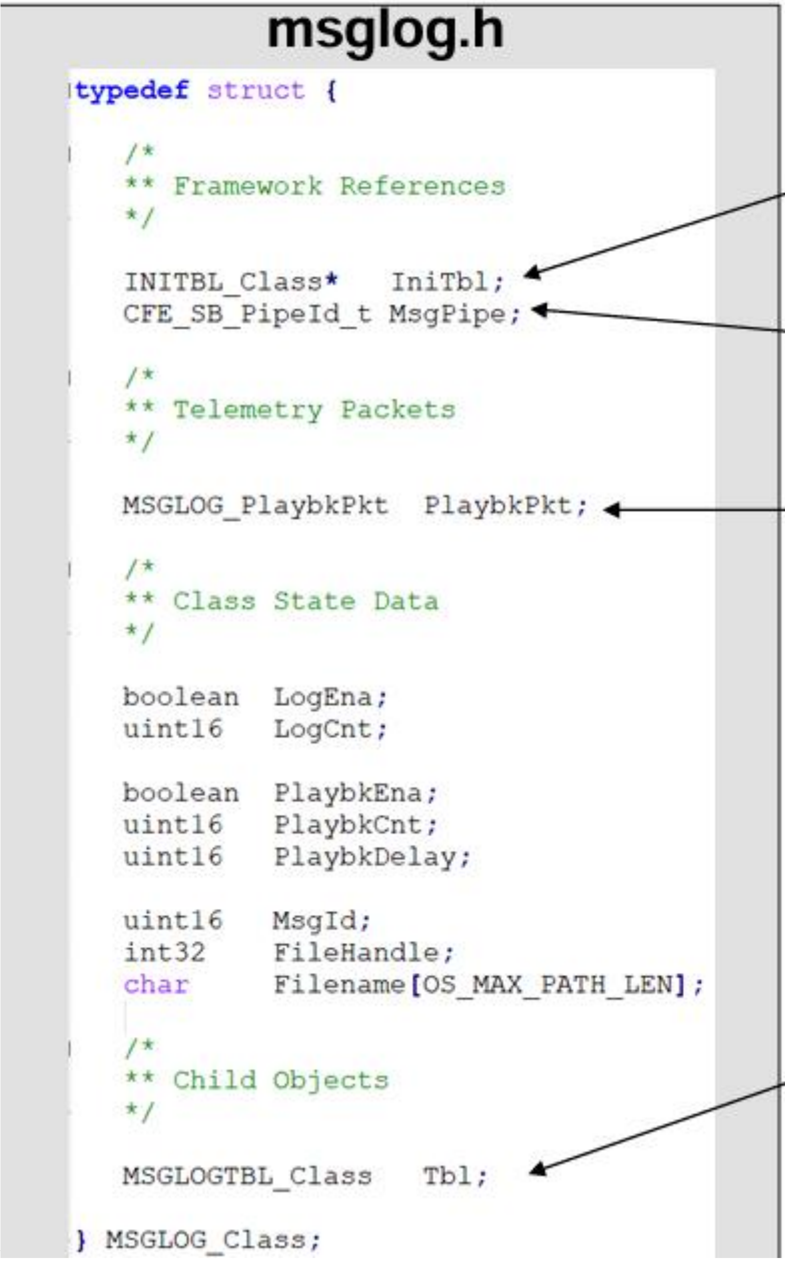
- Framework objects are not implemented as singletons, so a reference to an instance variable is always passes as the first parameter
- All framework objects are reentrant
- Only define instances for objects needed by the application. IniTbl, CmdPipe, and CmdMgr are common in most, if not all apps

2. Command & Telemetry Definitions

- Command packets sent by demo app. This is a special purpose child task command
- Telemetry packets generated by demo app

3. Object State data and Contained Objects

Object Composition Model – Message Log Header



msglog.h

```
typedef struct {

   /*
   ** Framework References
   */

   INITBL_Class*    IniTbl;
   CFE_SB_PipeId_t MsgPipe;

   /*
   ** Telemetry Packets
   */

   MSGLOG_PlaybkPkt  PlaybkPkt;

   /*
   ** Class State Data
   */

   boolean  LogEna;
   uint16   LogCnt;

   boolean  PlaybkEna;
   uint16   PlaybkCnt;
   uint16   PlaybkDelay;

   uint16   MsgId;
   int32    FileHandle;
   char     Filename[OS_MAX_PATH_LEN];

   /*
   ** Child Objects
   */

   MSGLOGTBL_Class   Tbl;

} MSGLOG_Class;
```

Reference to app's initbl instance

- This is needed because MsgLog uses some of the initialization parameters

MsgLog has its own SB pipe for reading packets to log

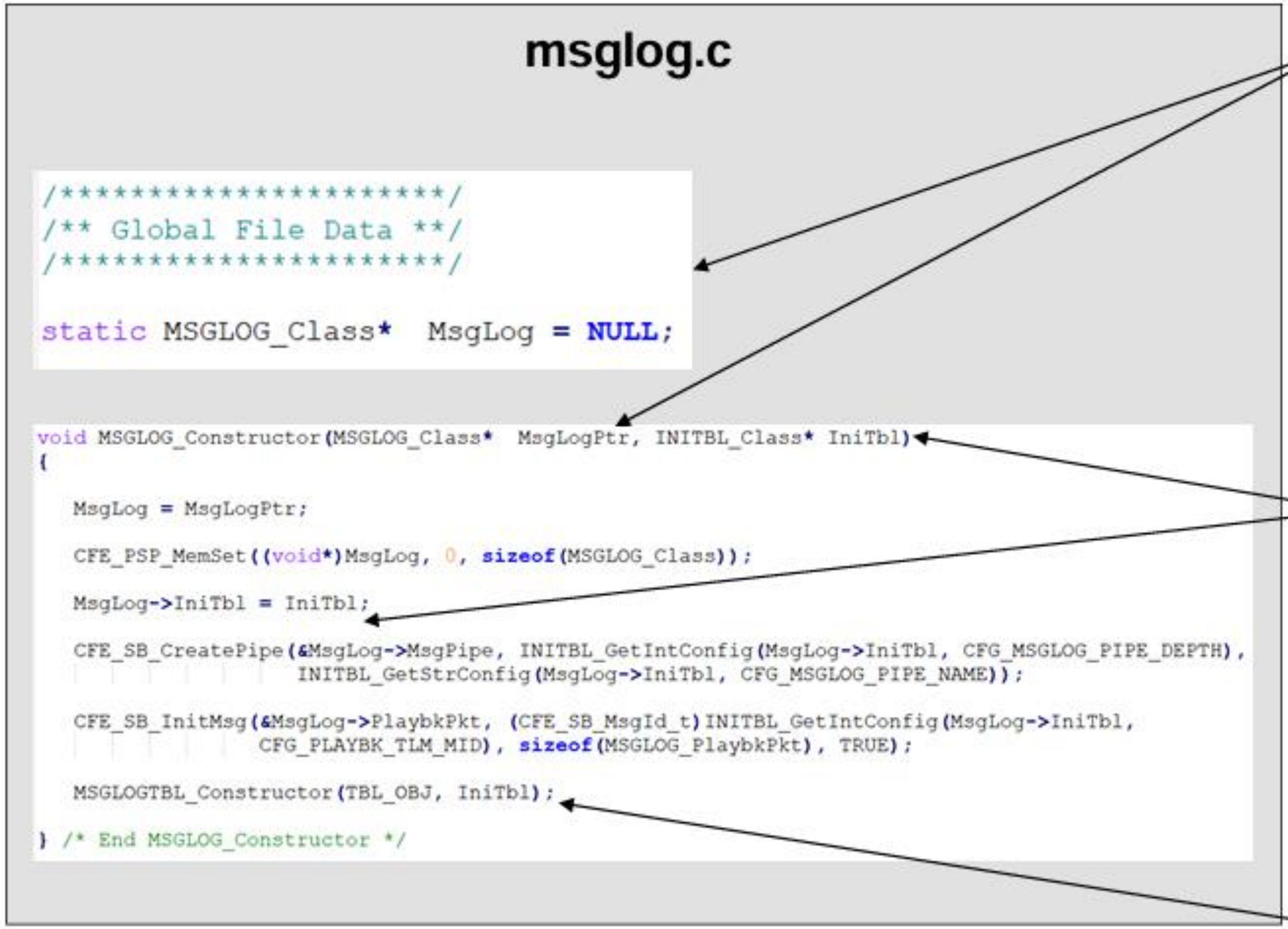
Message playback telemetry packet

MsgLog owns a MsgLogTbl

- All of the table parameters are used by MsgLog algorithms which why MsgLog owns the table

Coding

Object Composition Model – Message Log Source



msglog.c

```
/**********************/
/** Global File Data **/
/**********************/

static MSGLOG_Class*  MsgLog = NULL;
```

```
void MSGLOG_Constructor(MSGLOG_Class*  MsgLogPtr, INITBL_Class* IniTbl)
{

   MsgLog = MsgLogPtr;

   CFE_PSP_MemSet((void*)MsgLog, 0, sizeof(MSGLOG_Class));

   MsgLog->IniTbl = IniTbl;

   CFE_SB_CreatePipe(&MsgLog->MsgPipe, INITBL_GetIntConfig(MsgLog->IniTbl, CFG_MSGLOG_PIPE_DEPTH),
                           INITBL_GetStrConfig(MsgLog->IniTbl, CFG_MSGLOG_PIPE_NAME));

   CFE_SB_InitMsg(&MsgLog->PlaybkPkt, (CFE_SB_MsgId_t)INITBL_GetIntConfig(MsgLog->IniTbl,
                        CFG_PLAYBK_TLM_MID), sizeof(MSGLOG_PlaybkPkt), TRUE);

   MSGLOGTBL_Constructor(TBL_OBJ, IniTbl);

} /* End MSGLOG_Constructor */
```

Singleton coding idiom

- Parent sends a reference to object's instance data

Initialization Table

- Osk_c_demo owns the IniTbl and passes a reference to any object that needs IniTbl configurations
- This reference can be passed down the composite object hierarchy

Contained Objects constructed by owner

Section 6

Design Patterns

Introduction

- TBD – This section will include application design patterns
- The current slides are a collection of notes
- Functional and mechanistic apps/objects characterization. Layers within an object. An interface app’s child task perform mechanistic roles
- Reference Basecamp apps for examples
- Look into create system role categories & use NASA apps as examples rather than list NASA apps

Main Loop Control for Community Apps

Application	Main Loop Control	Control Notes
CF – CFDP	Pend Forever	Scheduler wakeup and HK request
CS – Checksum	Pend Forever	Scheduler wakeup and HK request
DS - Data Storage	Pend Forever	Subscribed message wakeup and HK request
F42 - 42 FSW Controller	Pend with timeout	Pends for sensor data packet from I42
FM – File Manager	Pend Forever	Ground Command, Scheduler HK request
HK - Housekeeping	Pend Forever	Scheduler combo pkt request and HK request
HS – Health & Safety	Pend with timeout	Scheduler HK request, no scheduler control
I42 – 42 Simulator I/F	Synched with 42	Flight equivalent depends upon H/W interfaces
KIT_CI – Command Ingest	Task Delay, Socket	
KIT_SCH – Scheduler	Synched with CFE_TIME	
KIT_TO – Telemetry Output	Pend with timeout	Subscribed message wakeup and HK request
LC – Limit Checker	Pend Forever	Scheduler wakeup and HK request
MD – Memory Dwell	Pend Forever	Scheduler wakeup and HK request
MM – Memory Manager	Pend Forever	Ground Command, Scheduler HK request
SC – Stored Command	Pend Forever	Scheduler wakeup and HK request
TFTP	Task Delay, Socket	Simulation environment (see CF for flight app)

Child Tasks

1. TIME service creating a child task
2. Task blocking waiting to receive data from SB
3. SCH Lab configured to periodically send the message to wake it up
   - You can also use a timer directly to run a loop at whatever frequency you want. As an example, see https://github.com/nasa/sch_lab for setting up a timer that calls SCH_LAB_LocalTimerCallback, giving a semaphore to run the processing loop.

- Reference Basecamp examples as well

- **Child tasks usage examples: I/O short execution at high priority, long term background tasks at low priority**

Telemetry

- Think about remote operations and autonomous onboard driven operations
- Command verification. Autonomous and manual. What can be verified when
- Use telemetry state rather than events
- Get notes from my cFE slides and system slides

The HK design pattern is not required and it happens to be common with the open source Command & data handling (C&DH) type apps. Many mission specific apps that run at a particular rate simply send a status telemetry packet at their execution rate. If this is too fast for telemetry then the telemetry output filter table can be used to reduce the telemetry rate.

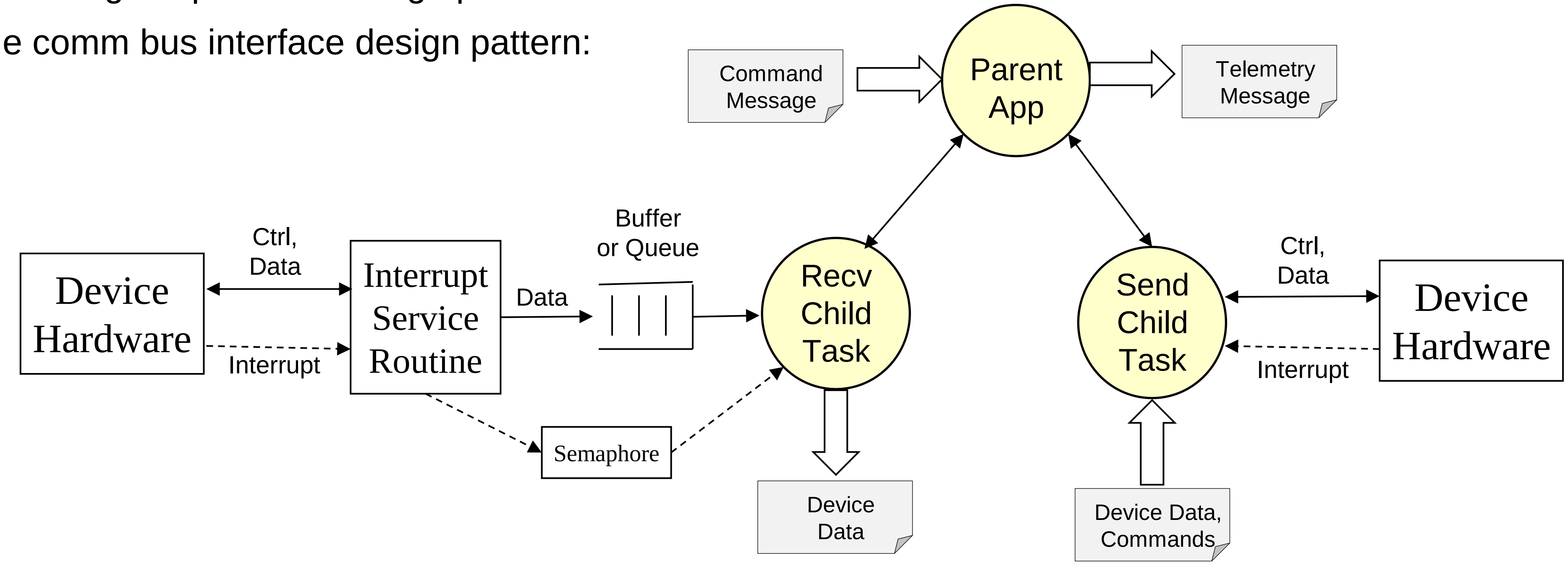
On time command driven packets

Diagnostic telemetry

Telemetry Monitor

Hardware Device Abstracted to Messages

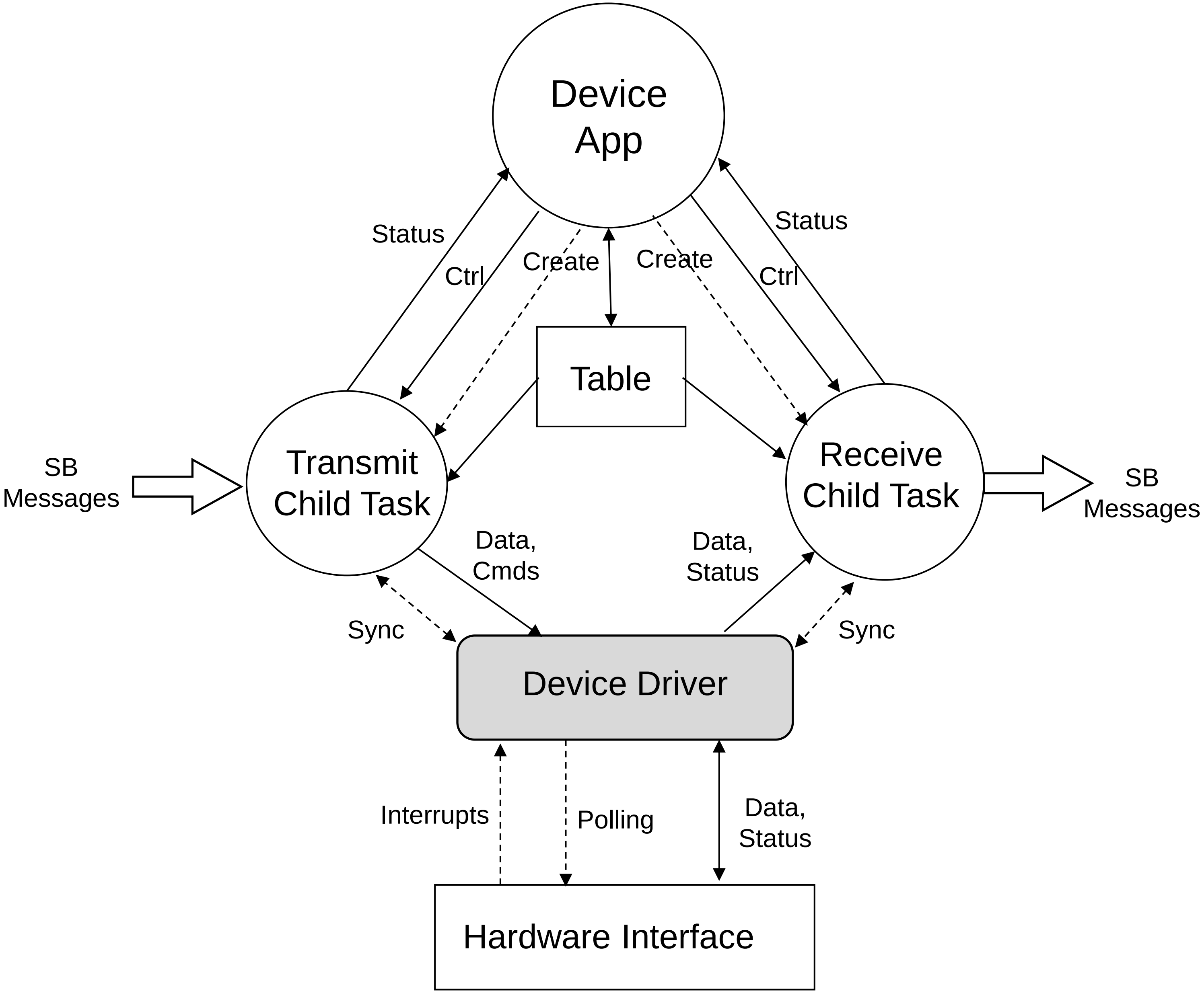
- **Device abstraction architectural role**
  - Read device data and publish on message bus
  - Receive messages and send to the device
- **Use a combination of software components to manage control/data**
  - Common design captured in design patterns
  - Example comm bus interface design pattern:



- **Not applicable to high data rate devices**
  - Require optimized point-to-point data transfer mechanisms including hardware acceleration

Comm I/O Application Design Pattern

TBD – Add semaphore
Create another design pattern
for dedicated hardware interface



Comm I/O Application Design Pattern

The diagram is accurate from a design perspective but it's a little misleading and the implementation is worth noting. The misleading part is that the shared table only contains what is used by both child tasks and there are other configuration tables that are not shared which are not shown in the diagram.

The child tasks do not call the CFE_TBL functions. In the main app's housekeeping cycle it performs table maintenance as follows:

```
OS_MutSemTake(global_data.TableMutex);

CFE_TBL_ReleaseAddress(handle)

CFE_TBL_Manage(handle)

CFE_TBL_GetAddress(global_data.TablePtr,handle)

OS_MutSemGive(global_data.TableMutex)
```

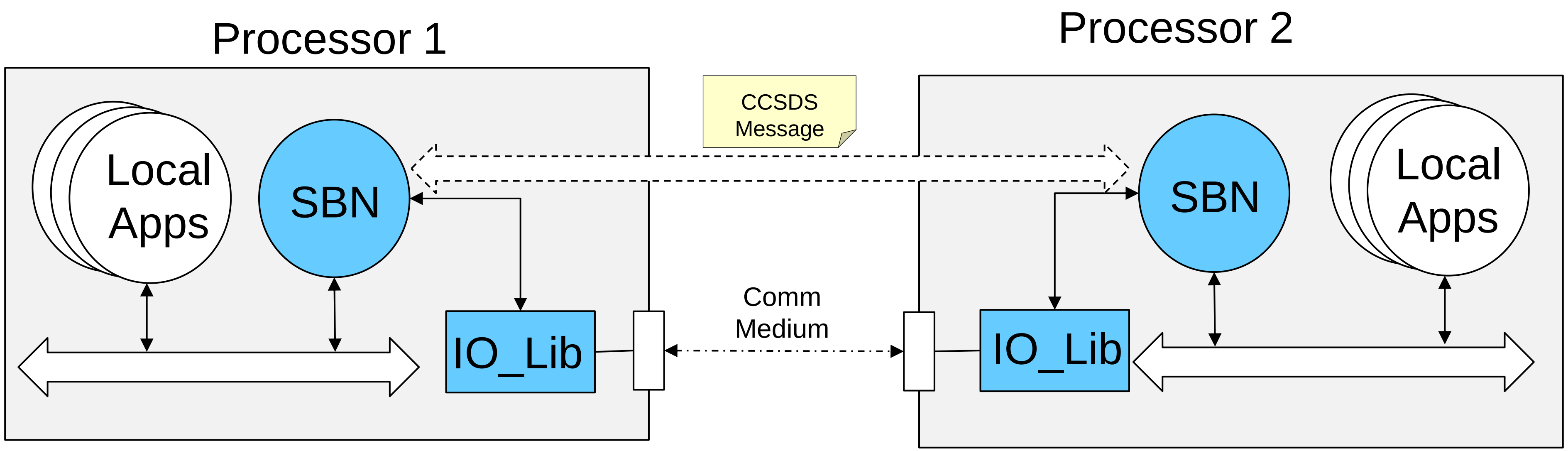
The child tasks use the global table pointer to access the table data

```
OS_MutSemTake(global_data.TableMutex);

...  global_data.TablePtr->...

OS_MutSemGive(global_data.TableMutex)
```

Library Helper Apps

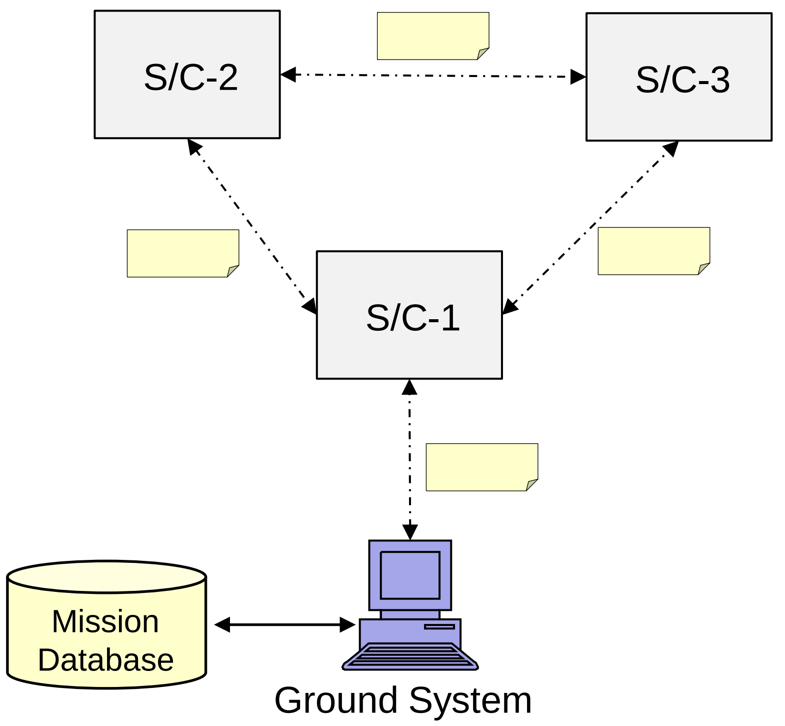
- **For libraries that require a ground interface, or some other more complex runtime environment, create a helper app to provide this support**
  - Conceptually the cFE’s service design uses this approach
  - From an implementation perspective, user libraries/apps must use cfe_es_startup.scr
- **PL_SIM**

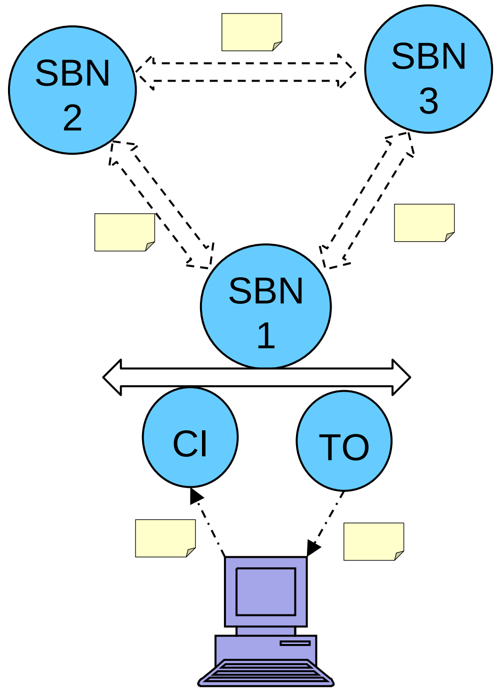
Bridging cFS Instances



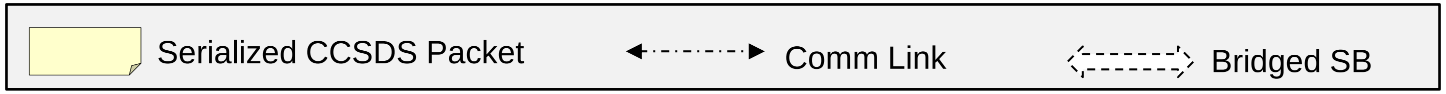
- **Software Bus Network (SBN, https://github.com/nasa/SBN)**
  - Provides a bridge over Ethernet using UDP or TCP
- **The current SBN design does not include an IO Lib as shown**
  - Command Ingest (https://github.com/nasa/CFS_IO_LIB) and Telemetry Output (https://github.com/nasa/CFS_IO_LIB) use IO_LIB (https://github.com/nasa/CFS_IO_LIB) that can be used as a reference design
- **Constellations using RF-based Inter-Spacecraft Links (ISL) will require a custom design**
- **Messages byte ordering must also be taken into account**
  - ToDo: Reference Systems Training Slides

Example Cluster





- Cluster of three spacecraft with S/C-1 provisioned for ground communications
- SBN used to virtualize the SB across ISLs
- Toolchains should manage message IDs/definitions and autogenerate FSW and ground code/artifacts to simplify system integration and deployment



Appendix A

Design Notation

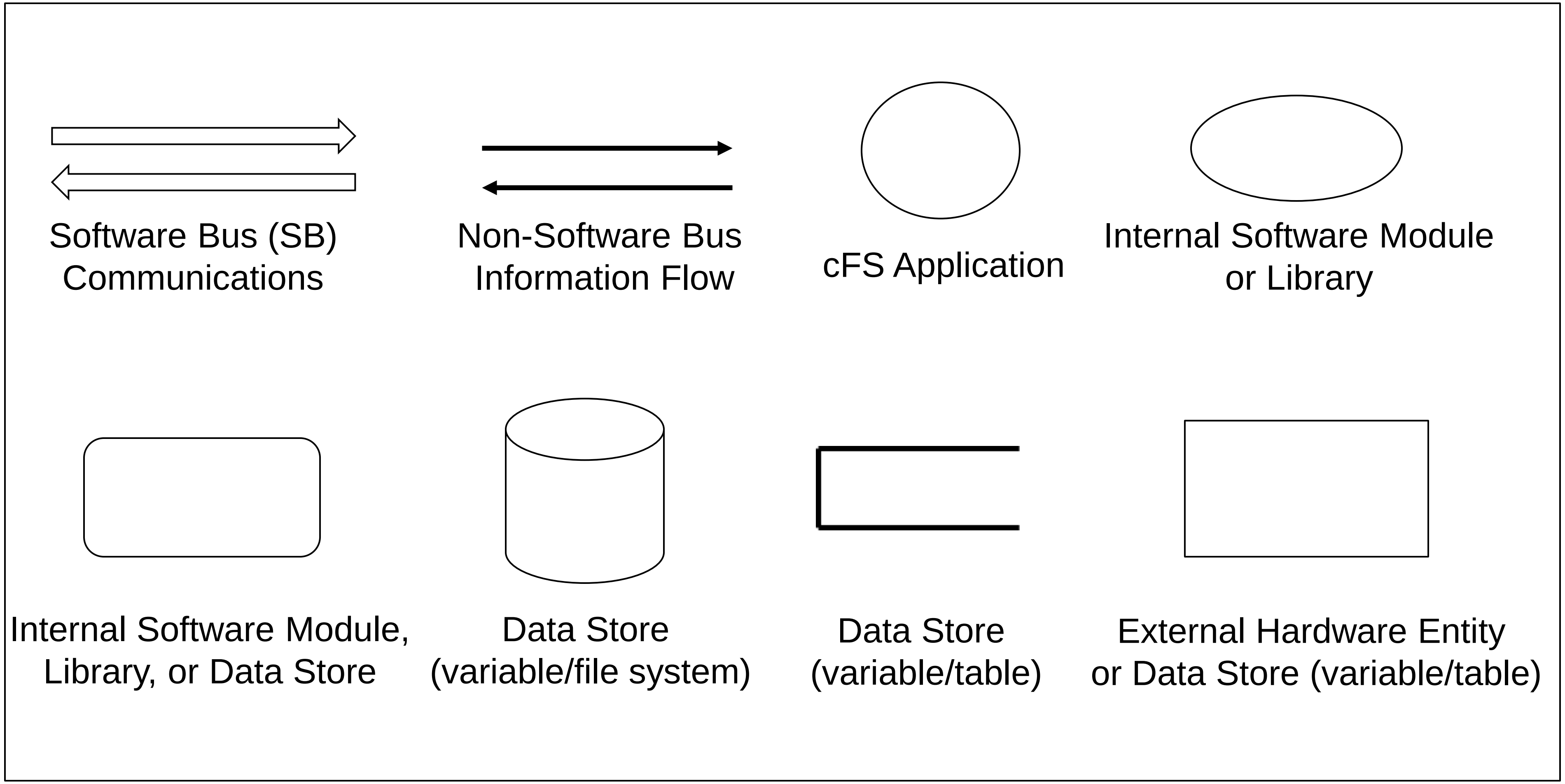
Object Diagrams

- **The cFS documentation does not adhere to a single notation such as the Unified Modeling Language (UML)**
- **The cFS has its roots in**
  - Data Flow Analysis (Hatley-Pirbhai Real-Time System Specification)
    - Context diagram
  - Flow charts
  - Unified Modelling Language Sequence Diagrams

Red text and arrows in the following slides are used to point out notation conventions and are not part of the design information

https://www.tutorialspoint.com/uml/uml_quick_guide.htm

Context Diagrams



Software Bus (SB) Communications

Non-Software Bus Information Flow

cFS Application

Internal Software Module or Library

Internal Software Module, Library, or Data Store

Data Store (variable/file system)

Data Store (variable/table)

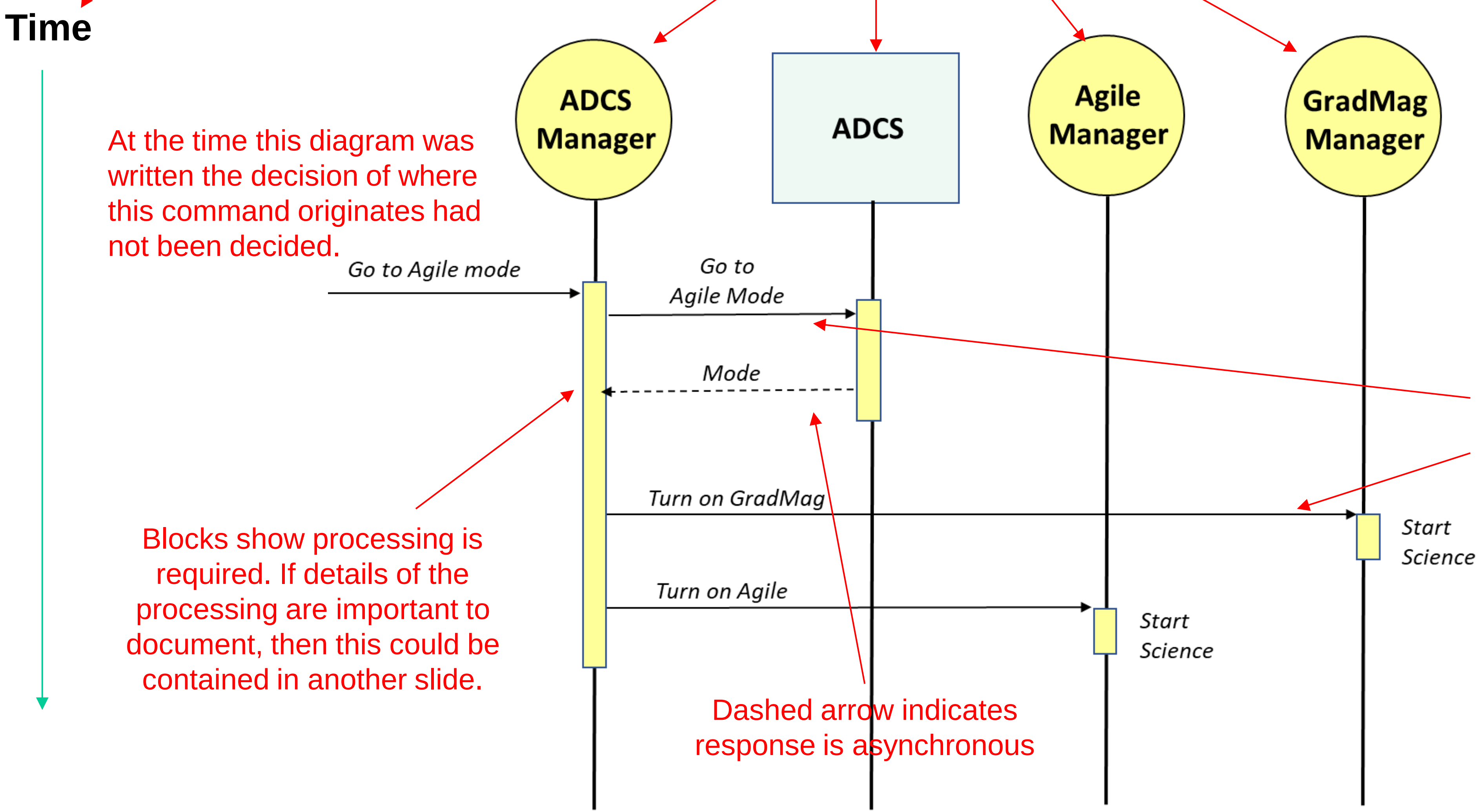
External Hardware Entity or Data Store (variable/table)

App-Level Sequence Diagram Example

Often not show but if audience is unfamiliar with sequence diagram it can be helpful

Applications, libraries and hardware devices shown across the top

Level of detail shown is based on the goal of what is trying to be communicated. This diagram is showing a mode transition sequence, so it is not important to show that an ADCS request may go through a library or device driver to command the ADCS device.



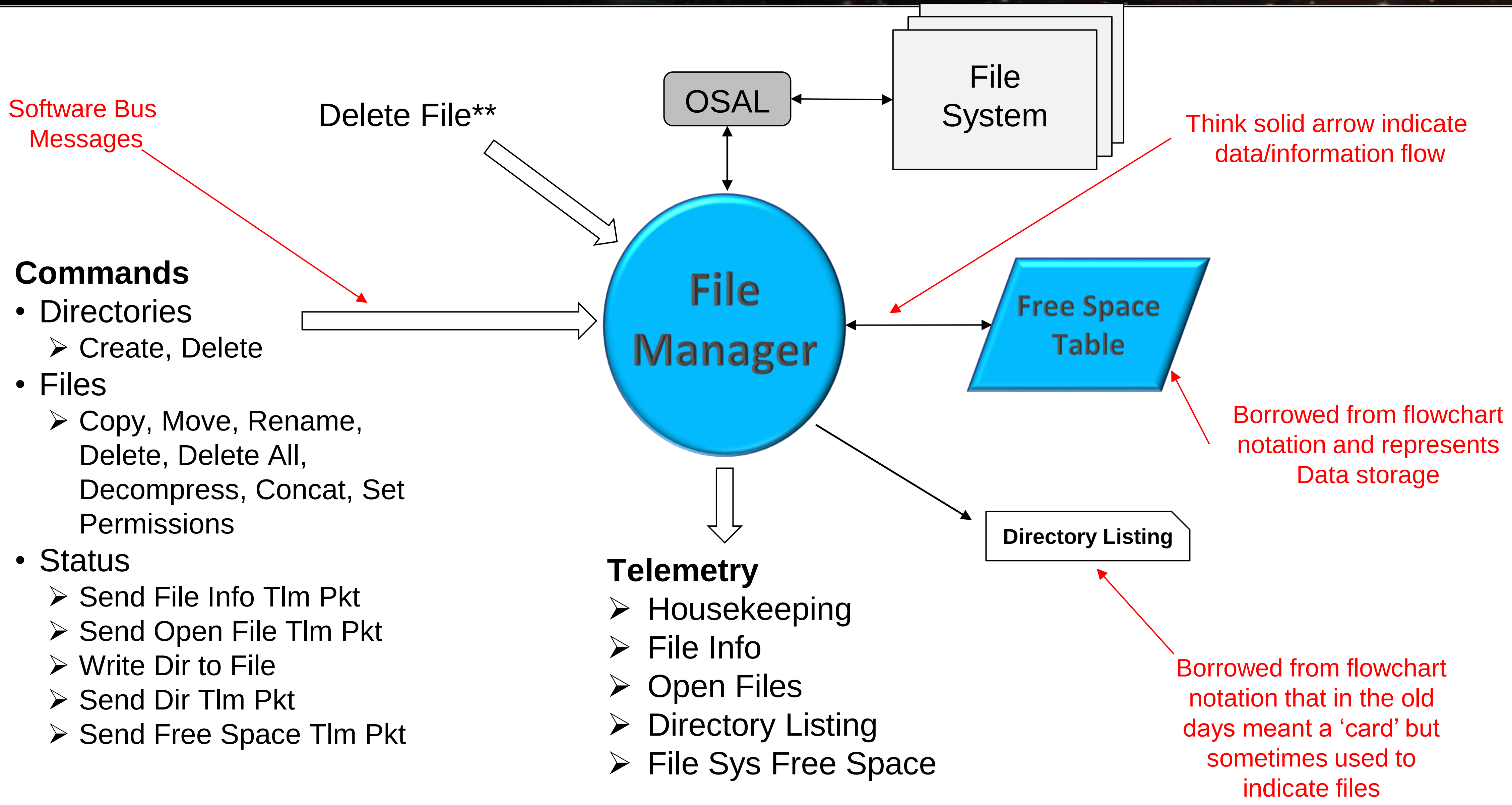
At the time this diagram was written the decision of where this command originates had not been decided.

Blocks show processing is required. If details of the processing are important to document, then this could be contained in another slide.

Dashed arrow indicates response is asynchronous

The thin line represents communication and how the communication occurs depends upon the item at the top of the subsystem line. App-to-app communication cocurs via software bus messages. An app-to-device communication typically occurs through a device driver.

App Context Diagram Example 1



Software Bus Messages

Delete File**

Think solid arrow indicate data/information flow

Commands

- Directories
  - ➢ Create, Delete
- Files
  - ➢ Copy, Move, Rename, Delete, Delete All, Decompress, Concat, Set Permissions
- Status
  - ➢ Send File Info Tlm Pkt
  - ➢ Send Open File Tlm Pkt
  - ➢ Write Dir to File
  - ➢ Send Dir Tlm Pkt
  - ➢ Send Free Space Tlm Pkt

Telemetry

- ➢ Housekeeping
- ➢ File Info
- ➢ Open Files
- ➢ Directory Listing
- ➢ File Sys Free Space

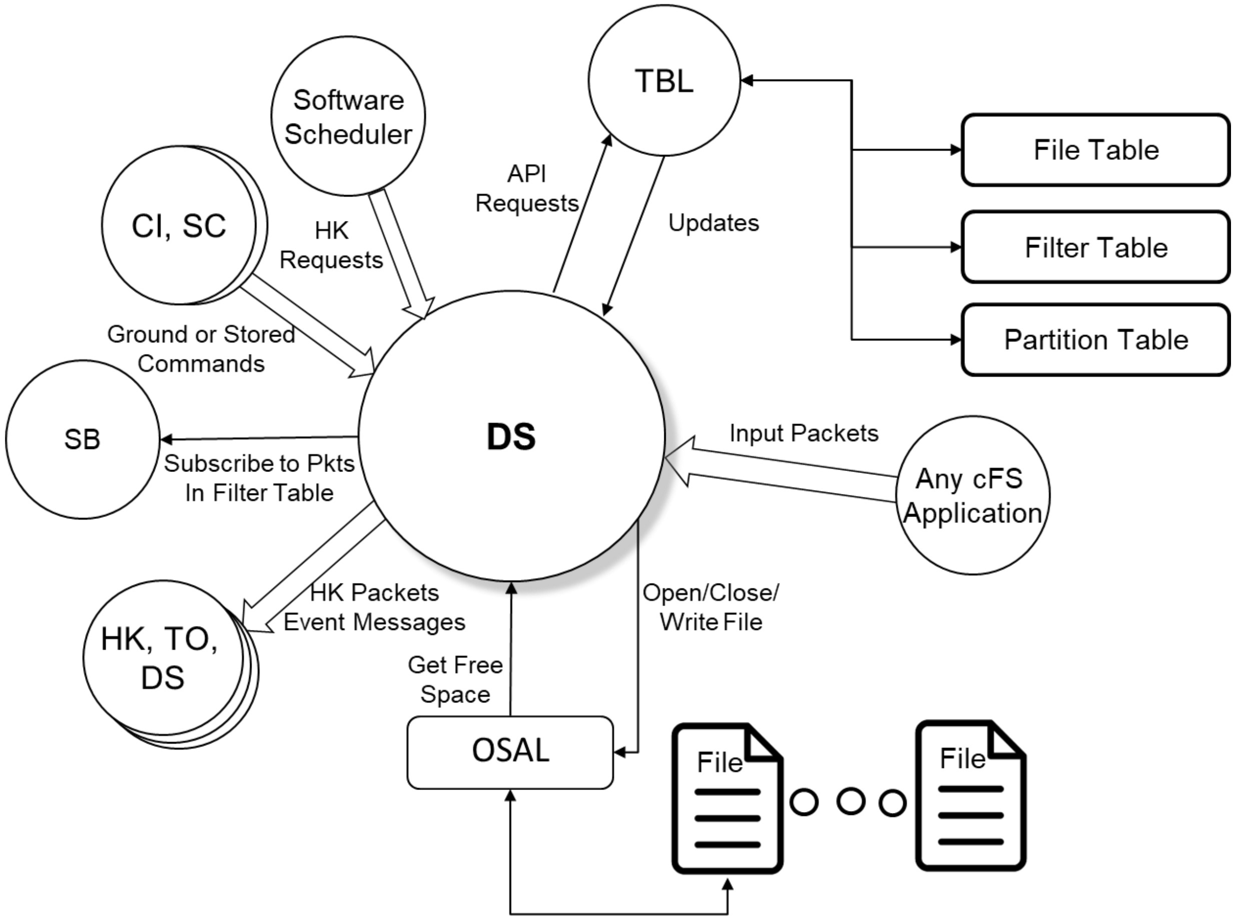
Borrowed from flowchart notation and represents Data storage

Borrowed from flowchart notation that in the old days meant a ‘card’ but sometimes used to indicate files

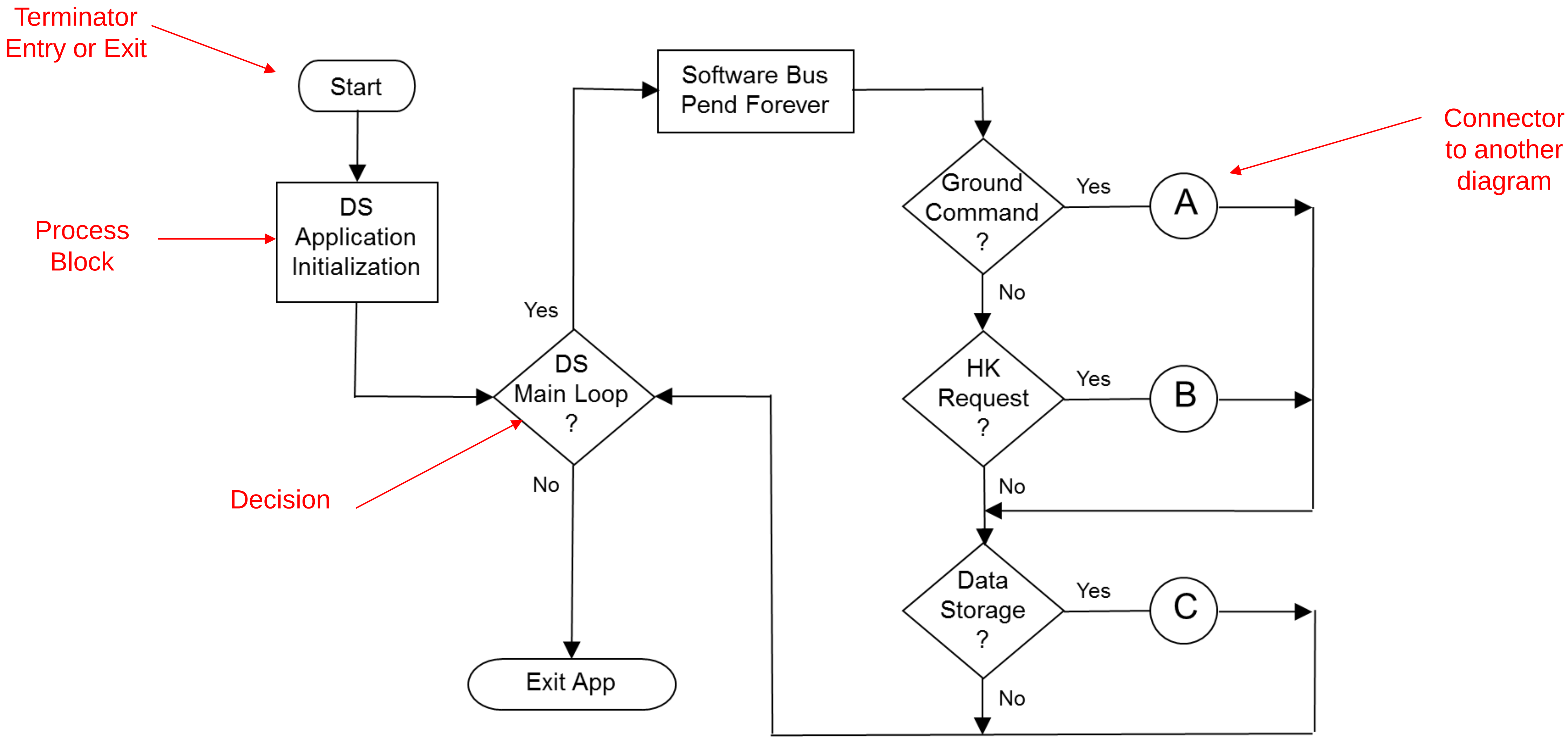
** Onboard command that doesn’t affect ground command counters

App Context Diagram Example 2

This is a more complete context diagram which is technically correct, but it can obscure the application-specific information that is most important. For example, HK request from the scheduler and outputting HK packet & event messages on the software bus are common design practice that may be omitted if people are comfortable with some assumptions. The important part of the diagram is showing interface boundaries to understand where control and data flow. Too much information is harder to maintain.

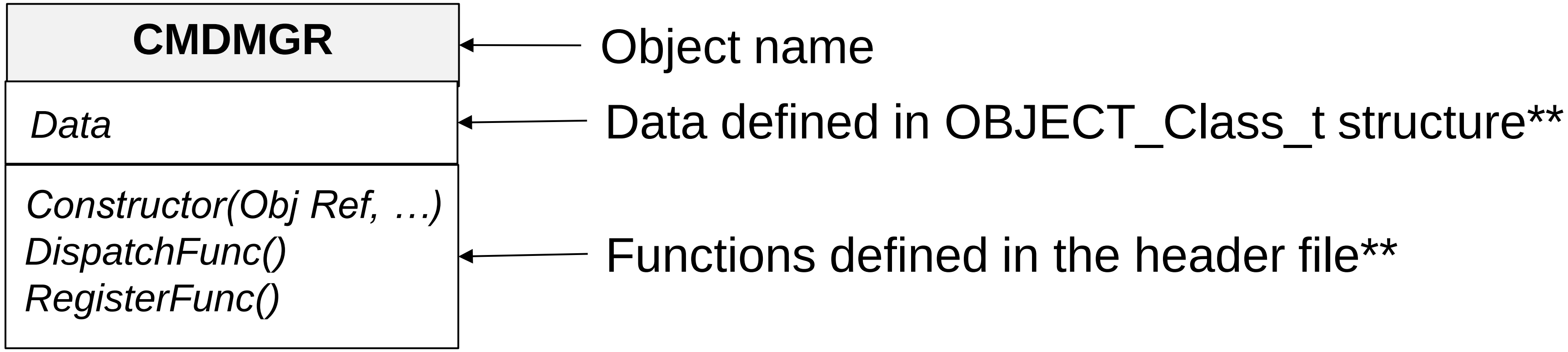


App Main Activity Example



Object Diagrams

- **The following Unified Modeling Language object notation is used in this document**


CMDMGR
Data
Constructor(Obj Ref, …)
DispatchFunc()
RegisterFunc()
Object name
Data defined in OBJECT_Class_t structure**
Functions defined in the header file**

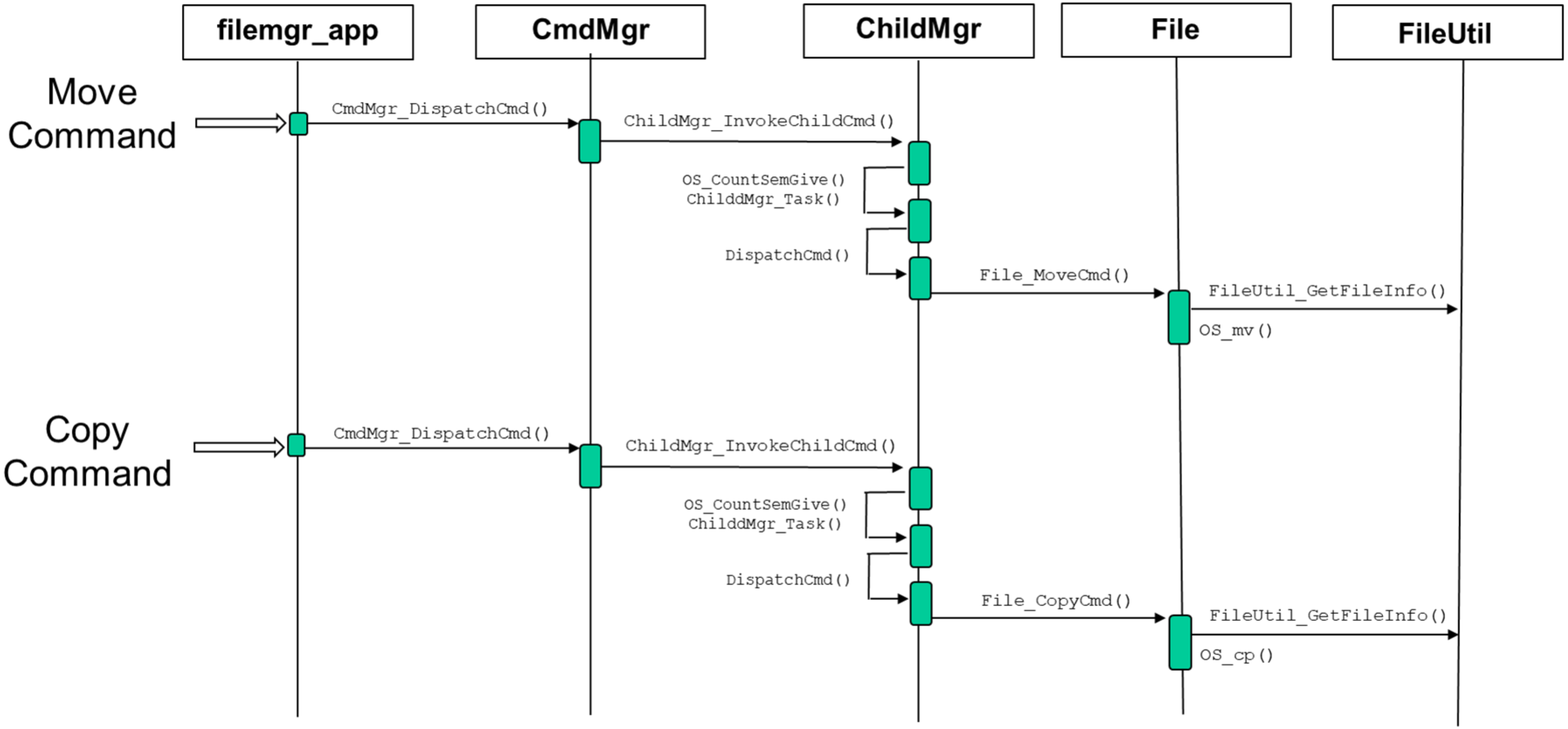
** If data or functions are not relevant to the context in which the object diagram is being used then it should not be shown in order to enhance readability.

Object Diagrams

Object Diagrams

Intra-App Sequence Diagram Diagram

Intra-app sequence diagrams are typically not used by the cFS app but are used by OSK apps documents. The top elements represent objects and the communication between objects is via calls to an object’s public methods



Appendix B

Coding Conventions

Coding Conventions

- **There are a couple of coding conventions that help make osk_c_fw-based apps consistent and easier to maintain**
  - Even if these conventions are not followed, establishing your own and being consistent helps increase productivity and reduce errors
- **Each object declares a type with the name XXX_Class where XXX is the filename and the object name**
  - Definitions within a class use consistent groupings and order as shown in osk_c_demo.h
- **Object variable names should be the same name as the class type but without '_Class'**
  - Names within a class should not repeat the class's name or information conveyed by the name so the concatenation of the nested names reads well: *OSK_C_DEMO.MsgLog.PlaybkEna*
- **"Convenience macros" can be used to reference framework objects that need to be passed as the first parameter to osk_c_fw components**
  - For example, use *"#define INITBL_OBJ (&(OskCDemo.IniTbl))"* in function call to *INITBL_GetIntConfig(INITBL_OBJ,…)*

Need to capture all OSK's coding idioms and styles

Object Implementation Conventions (1 of 2)

- **Each object is defined using two files: The .h file defines the object's specification (i.e., interface) and the .c file defines the object's methods both public and private**
  - The base filename is the object's name although sometimes due underscores, abbreviations or acronyms they are not exact. Regardless of whether they're exact the object name should be consistent.
  - All global identifiers (macros, types, and functions) are prefixed with the capitalized object name followed by an underscore to minimize the chances of a global name clash. Type definitions end in "_t" which is consistent with the cFS.
  - The osk_c_fw library Command Manager object will be used as a concrete example, and it can be referenced to illustrate a complete example. Command Manager files are cmdmgr.h and cmdmgr.c and the global object prefix is "CMDMGR_".
  - TODO: be aware of common object names that could cause a global name space collision.
- **The header file (i.e., cmdmgr.h) uses the following conventions**
  - Preprocessor header file "guards" are used to protect against the multiple definition if the header is included more than once. The naming convention is to use the base filename with leading and trailing underscores:

```
#ifndef _cmdmgr_
#define _cmdmgr_
    Header file contents
#endif /* _cmdmgr_ */
```

- **To enhance readability Basecamp header files always follow the same order**
  - Constants (macros), typedefs, exported (global) function prototypes
- **What should be in a header file**
  - Only constants, typedefs and function definitions that need to be global
  - Every object defines a typedef for a class structure using the OBJECT_Class_t convention (i.e., CMDMGR_Class_t)
- **What should not be in a header file**
  - Variables should not be declared
  - For reusable apps/libraries, configuration parameters that may be changed in future instantiations (covered later)

1. https://deviq.com/principles/solid

Object Implementation Conventions(2 of 2)

- **The source file or body file (object-oriented terminology) at a minimum implements all the object’s global functions (aka methods)**

- **The source file may also include local definitions for constants, typedefs and functions**
  - Local names should be meaningful and may follow a local naming convention, but they should not be prefixed with the object’s global name prefix. This makes it easy for someone reading the code to immediately understand the scope of a particular name.
  - Data and functions global to the source file are defined as static to limit their scope and not clutter the global namespace

- **To enhance readability Basecamp source files always follow the same order**
  - Macro constants, typedefs, global file data, local (static) function prototypes, global function implementation and local function implementation

- **File prologue and function comments also play an important role in code readability and maintenance**
  - Design related information is typically captures in a list of Notes
  - File prologue notes should provide important/relevant object-level design information. What’s is its role? Is there important rationale that should be provided for understanding why the object’s interface is defined like it is? This
  - Function prologue notes should provide implementation level rationale.

app_cfg.h Contents

- **Application version**
  - Defines app’s major and minor versions
  - If a change is made to any app source file during a deployment, then OSK_C_DEMO_PLATFORM_REV in osk_c_demo_platform_cfg.h should be updated
- **Initialization table configuration definitions**
  - Define the C macro and JSON object names for each
- **Command Function Codes**
  - Define all of the app’s command function codes
  - This follows the design pattern of a single app command message with the function code being used to distinguish between commands
- **Event Message Identifiers**
  - Define the base event ID for each App Object
- **App Object configurations**
  - These should be compile-time configurations, runtime configurations should be defined in the IniTbl
  - Defining these configurations in app_cfg.h breaks the OO encapsulation, but it allows app_cfg.h to serve as the app’s single point of configuration