# cFS Basecamp Application Developer's Guide

Version 1.10

December 2023

# Document Audience and Prerequisites

- **Objectives**
  - Describe how to develop apps using cFS Basecamp's Application C Framework (APP_C_FW)
  - Basecamp app design patterns are object-based and deviate from some of the design patterns in the cFS Application Developer's Guide

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

- **Prerequisites**
  - An understanding of the material in Basecamp's cFS Overview and cFS Framework documents
  - Familiarity with the cFS Application Developer's Guide
  - C programming experience
  - Linux experience

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

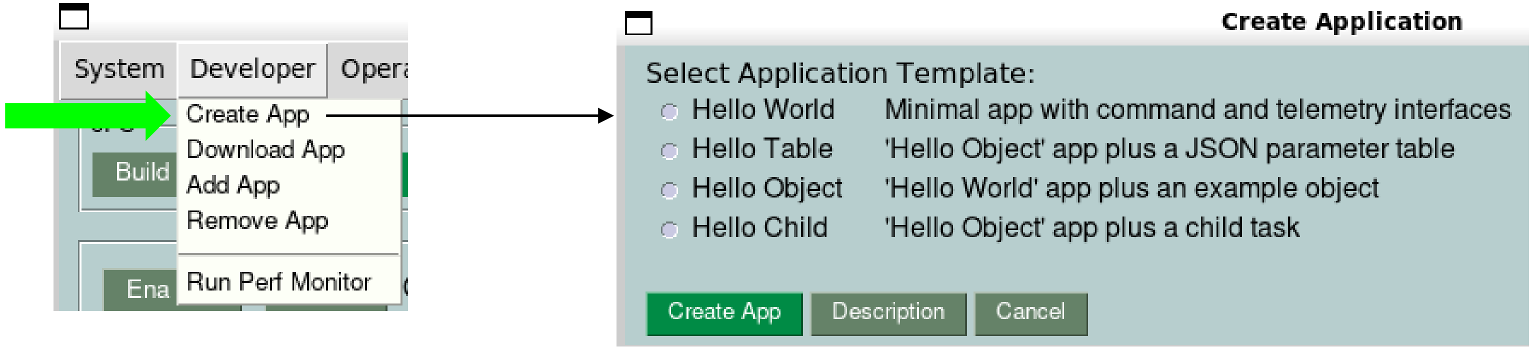
# Outline



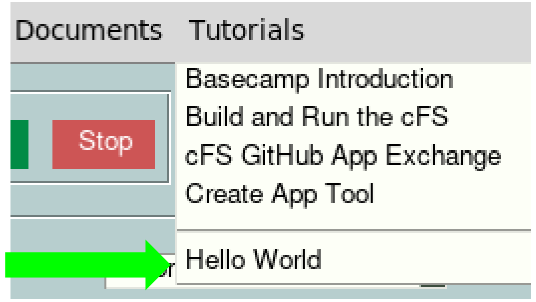
- **Outline approach**
  - This type of document is challenging because you often need to know multiple pieces of information in parallel, but not in depth, and then spiral through the topics going more in depth
  - The *“Hello App” coding templates* and *app_c_demo* sections are intended to help with this situation by introducing concepts without too much detail so the following sections can go into much more detail
  - The File Manager refactoring section is included to help readers that are familiar with the NASA app design approach understand the app_c_fw approach

# Hello App Designs

- **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 here are explained in greater detail later
- **Select "Create App" in the Developer dropdown menu to access the "Hello App" templates**

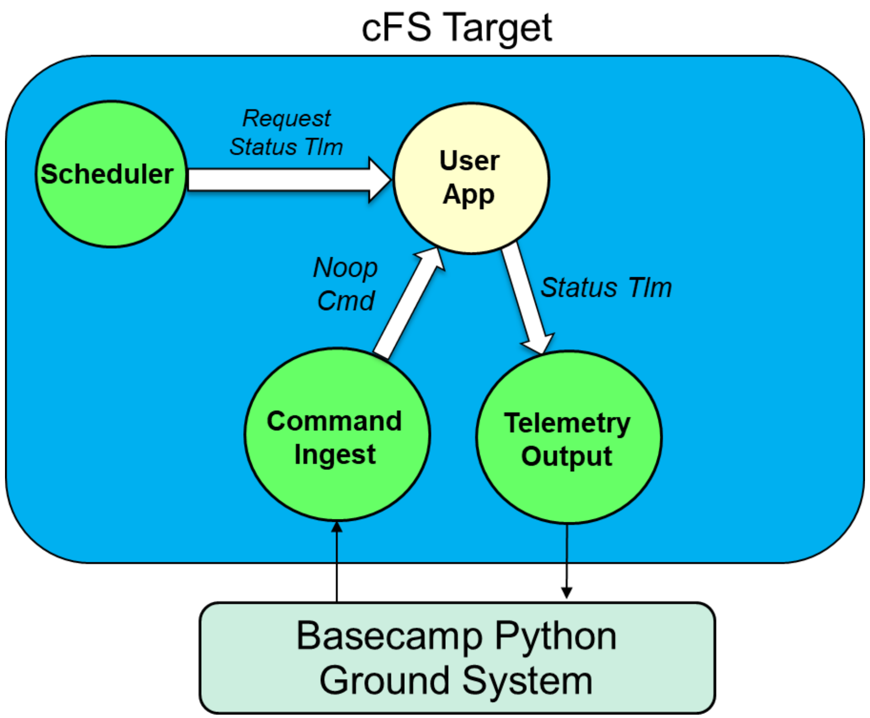



- **After an app is created and the Python GUI is restarted the coding tutorial will be listed in bottom section of the Tutorials dropdown menu**



# Application Runtime Context

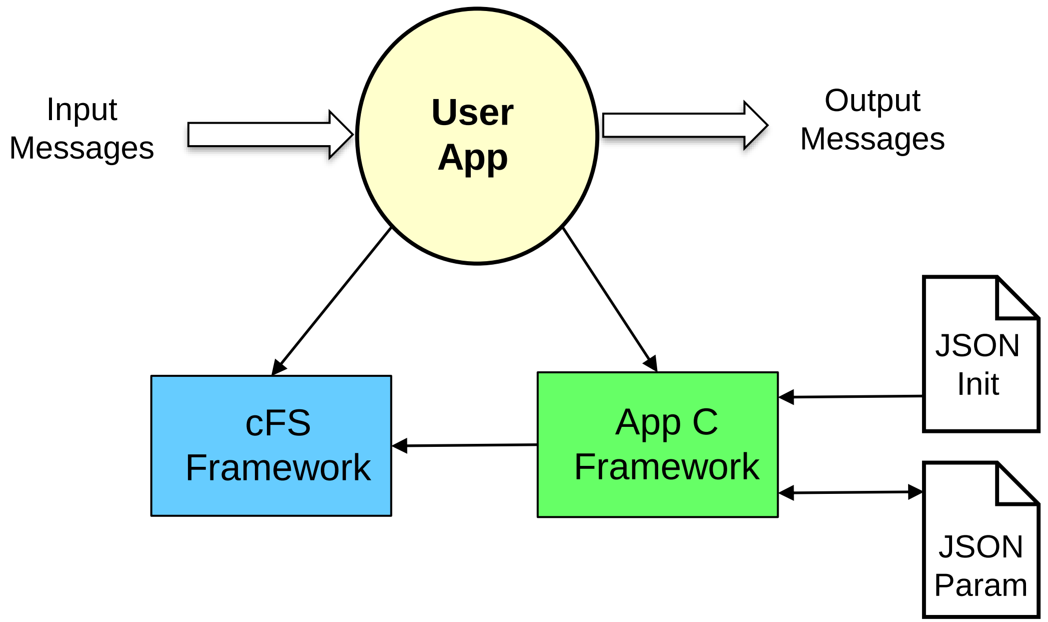
**The following diagram is from the cFS Framework document and it shows an app's context with respect to the ops service app suite**




- **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

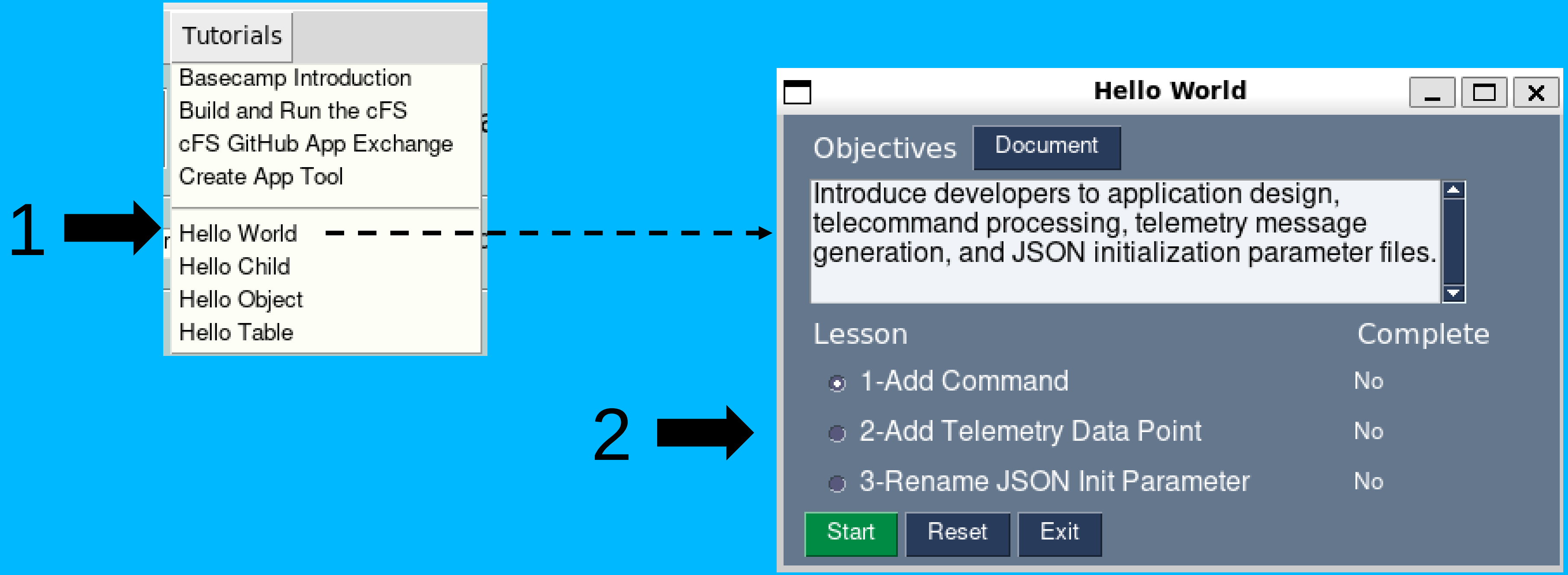





- **This is the cFS Basecamp application context**
  - Apps may have additional interfaces such as an app-specific library
  - When developing apps it's good practice to draw the app's detailed context to clearly define and understand it's interfaces
- **Input and output messages can either be command 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 using APP_C_FW services

# cFS Basecamp Exercises: Hello World

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



# Minimal App Functionality

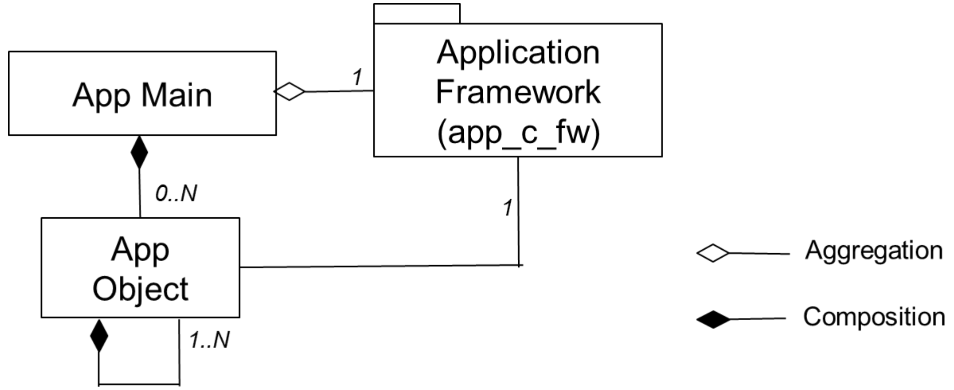
- **Basecamp's "Hello World" app template implements the minimal functionality required by an app**
  - Create a Software Bus "Pipe" and register to receive messages
  - Accept command messages and execute command-specific functions
  - Output status telemetry
- **The following functionality are NASA/Goddard design/code patterns that evolved based on 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" 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.
    - Allows command counters to 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 and augmentations**
- **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 to define runtime configurations**
  - cFS target management tools can modify these files that facilitates automated system integration
  - Read during an app's initialization
  - Many mission and platform configurations traditionally defined in C header files are defined in this initialization file
- **APP_C_FW Command Manager**
  - Apps register each object's command functions with the Command Manager
  - When a command message is received, Command Manager 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

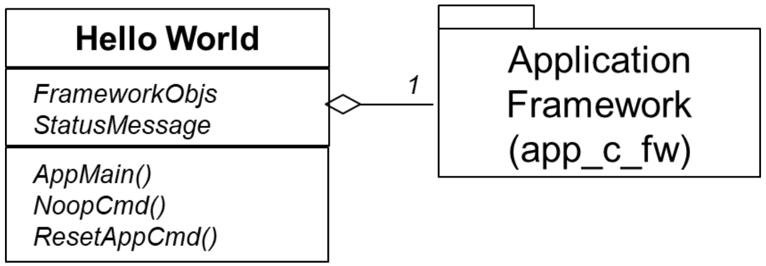
# Application Design Overview

- **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**



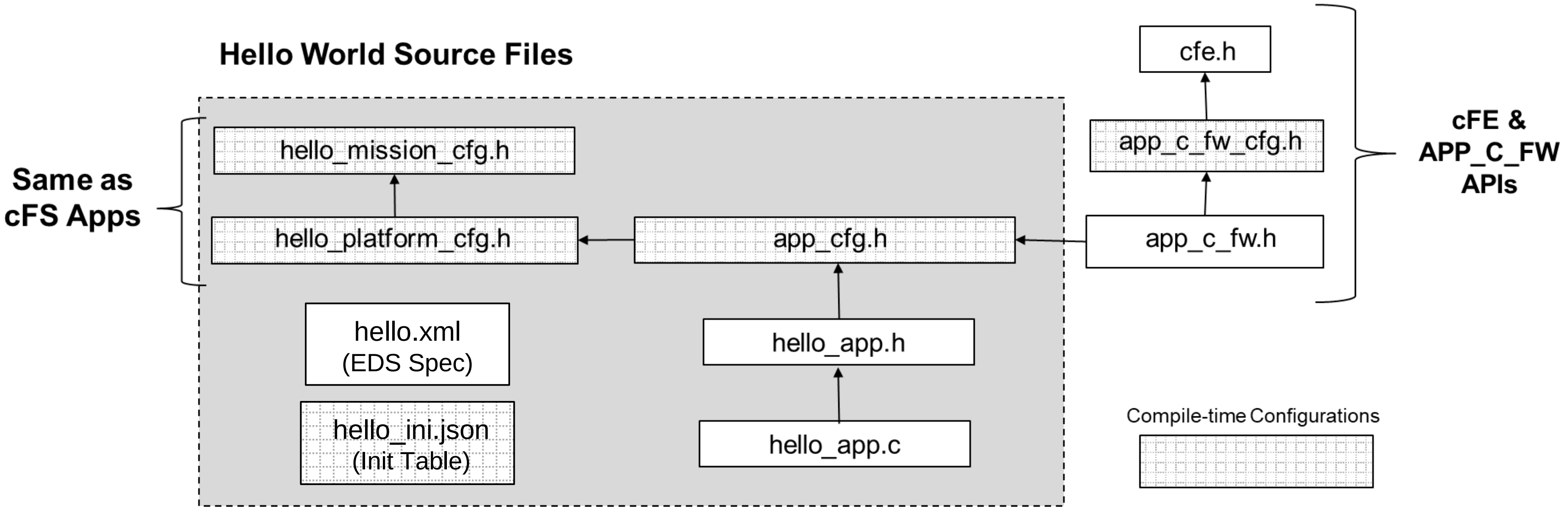

- The "App C Framework" is a package available to all apps
- Apps are composed zero or more objects

- **The Hello World app is the simplest app with no objects**



# 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 hello.ini. |
| **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 hello.ini. |
| **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 the 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 sharing the framework on a platform. |
| **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 root of the object hierarchy |
| **hello.xml** | Electronic Data Sheet (EDS) specification for primarily application message definitions |
| **hello_ini.json** | Configuration parameters that are read by the app when it initializes |

# Hello World App Main Loop Execution



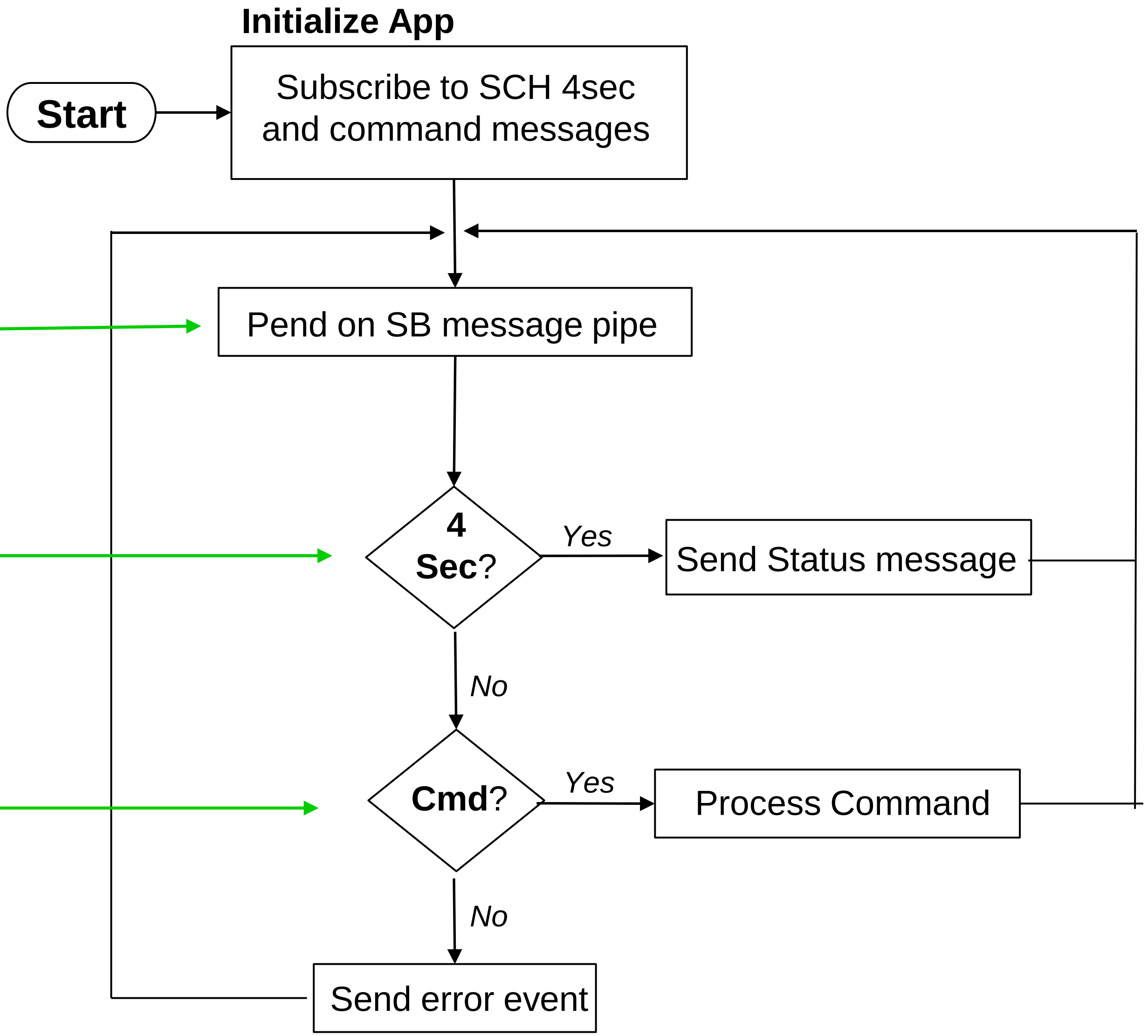
**Suspend execution until a message arrives on app's pipe**

**Periodic *4 second* message from SCH app**

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

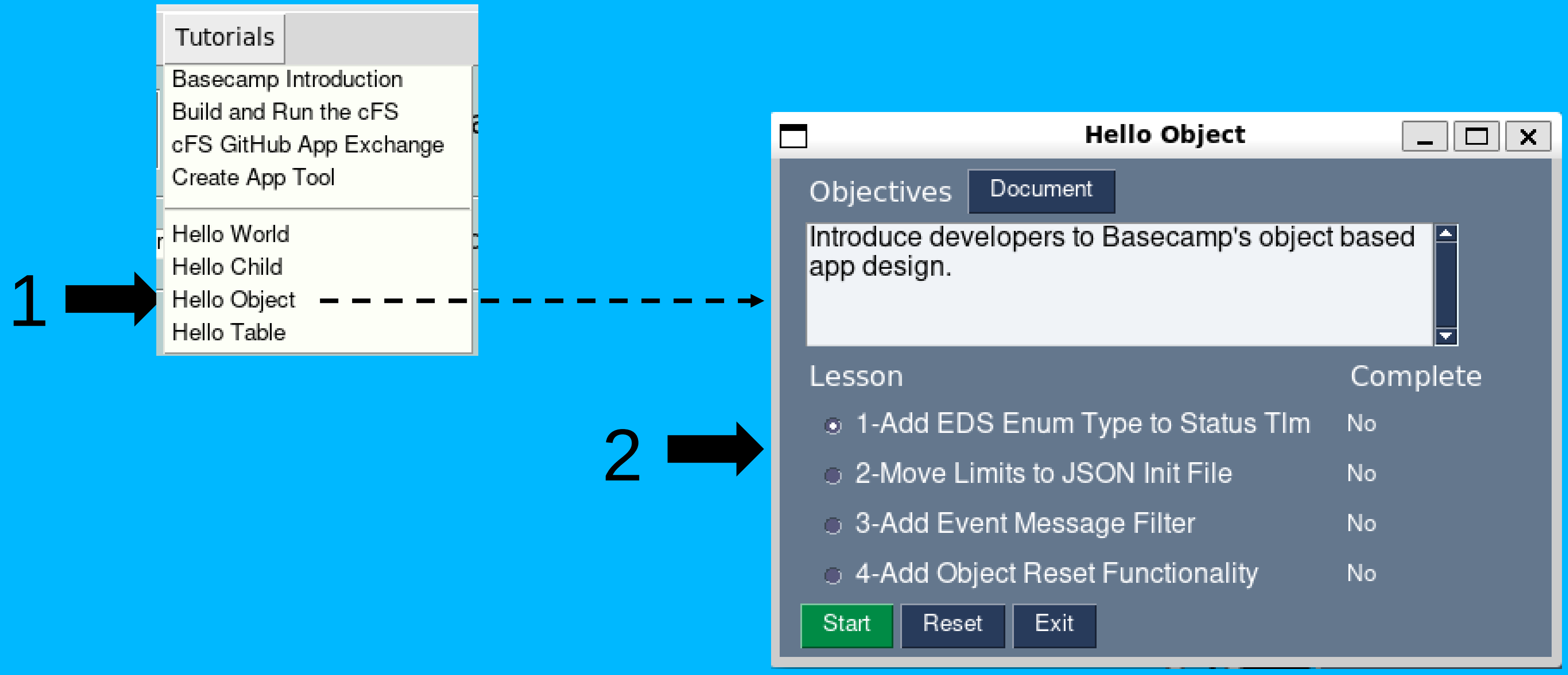
**Process commands**

- Commands can originate from ground or other onboard apps



# cFS Basecamp Exercises: Hello Object

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



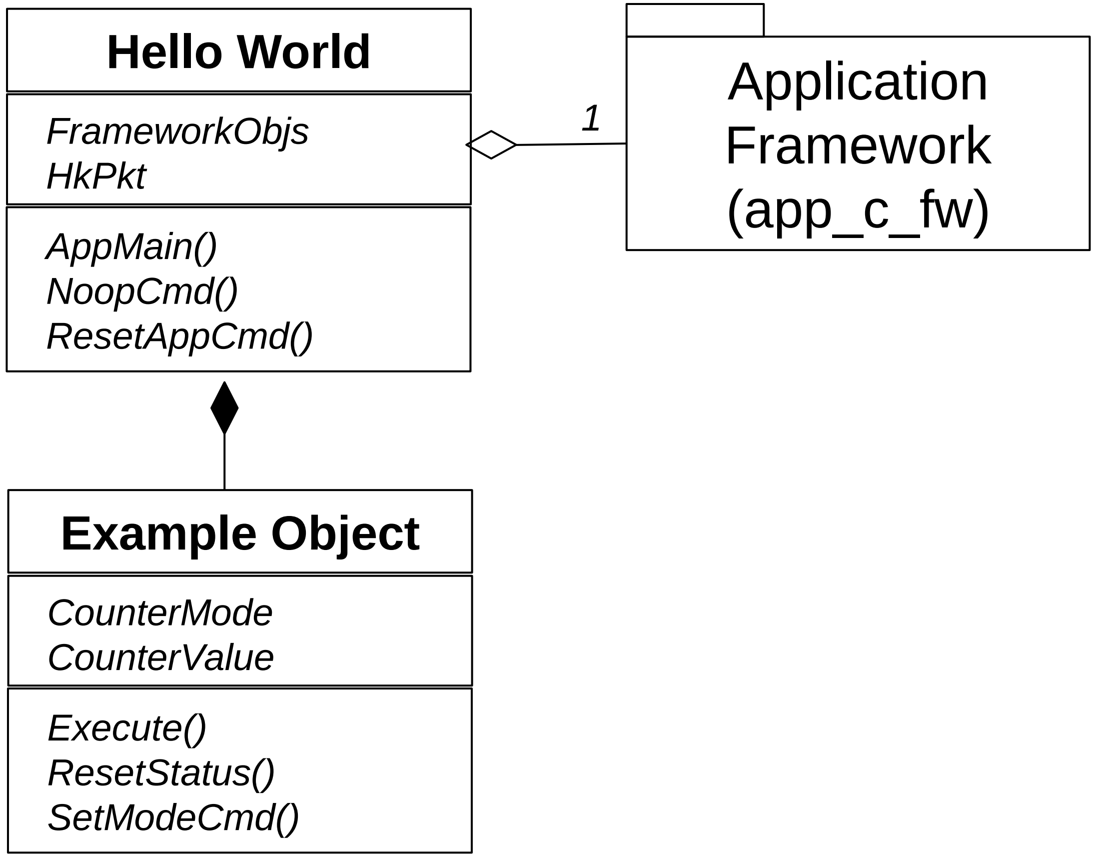
# Hello Object App Functionality

- **The Hello Object app adds an example object to the Hello World app**
  - The Hello World coding exercise additions are <u>not</u> 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

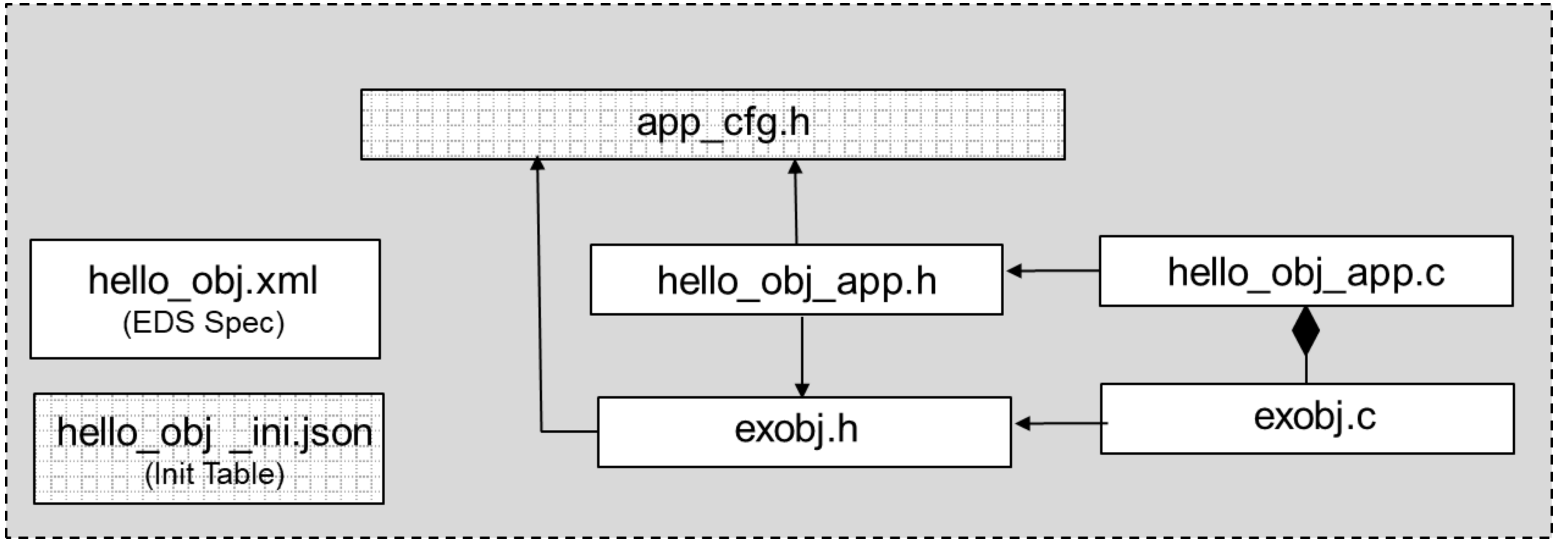
# Hello Object Design





# Hello Object Source Files

## App Source Files




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

```
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;
```

# Hello Object App Main Loop Execution

**Suspend execution until a message arrives on app's pipe**

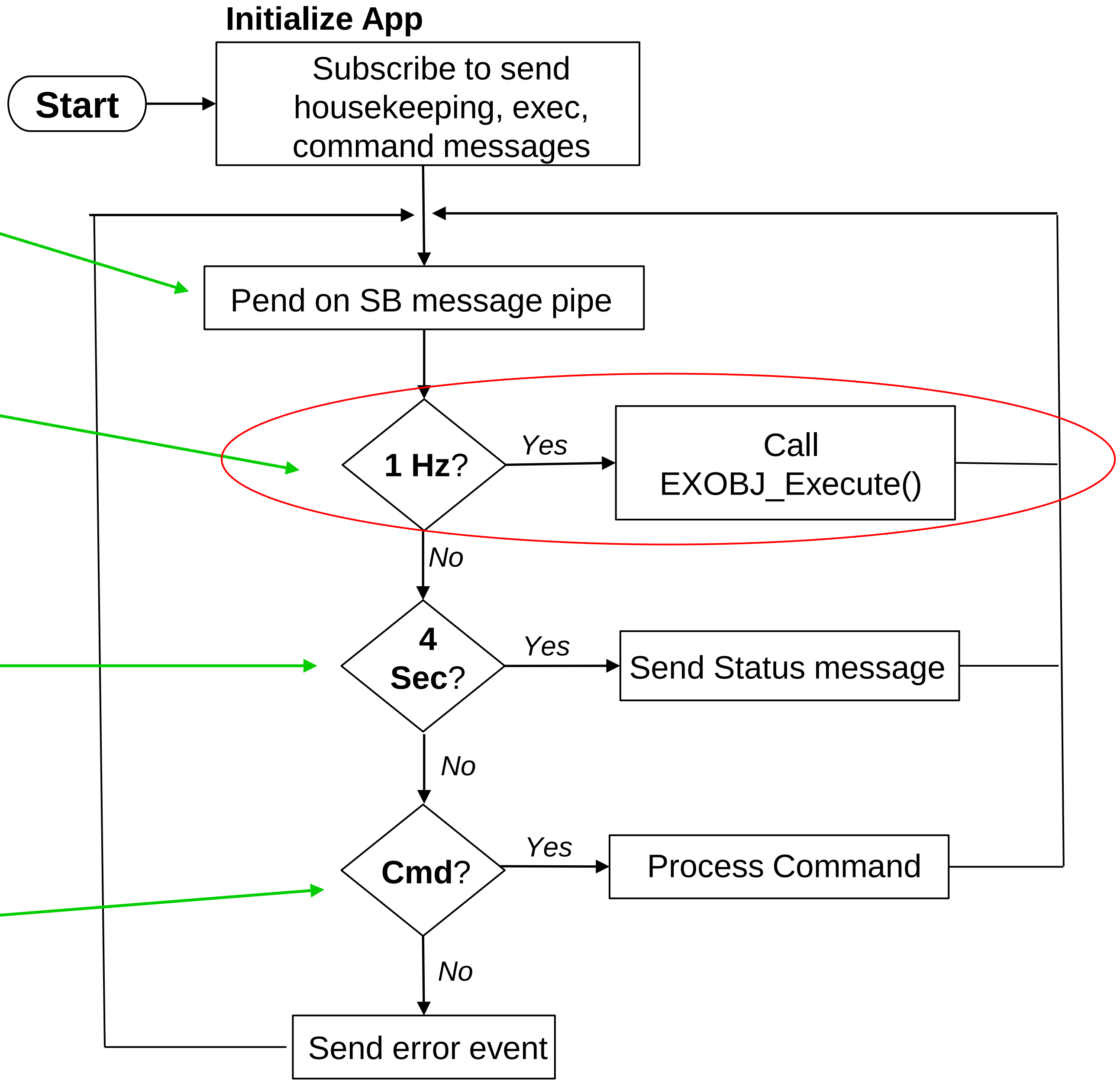
**Periodic *1Hz* message from SCH app (added to Hello World)**

**Periodic *4 second* message from SCH app**

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

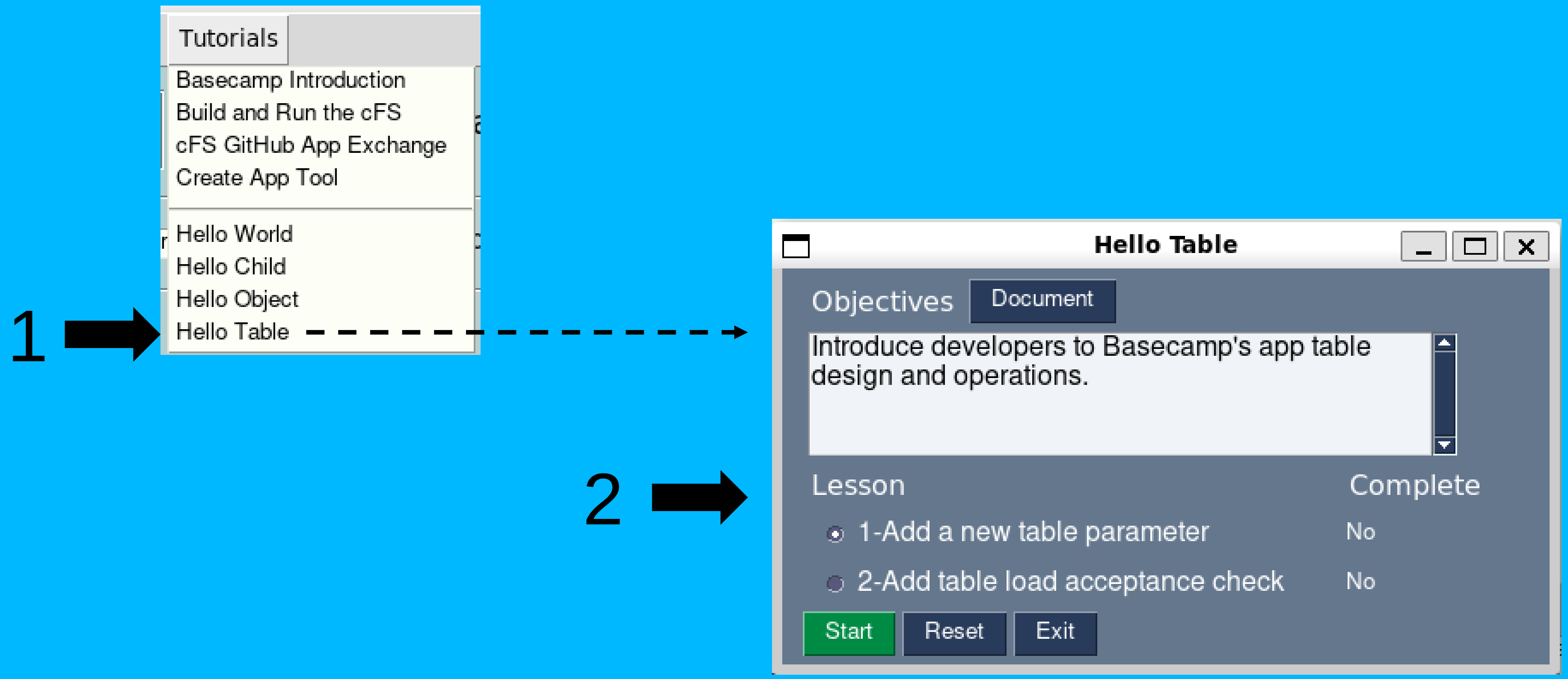
**Process commands**

- Commands can originate from ground or other onboard apps



# cFS Basecamp Exercises: Hello Table

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



# Table Introduction

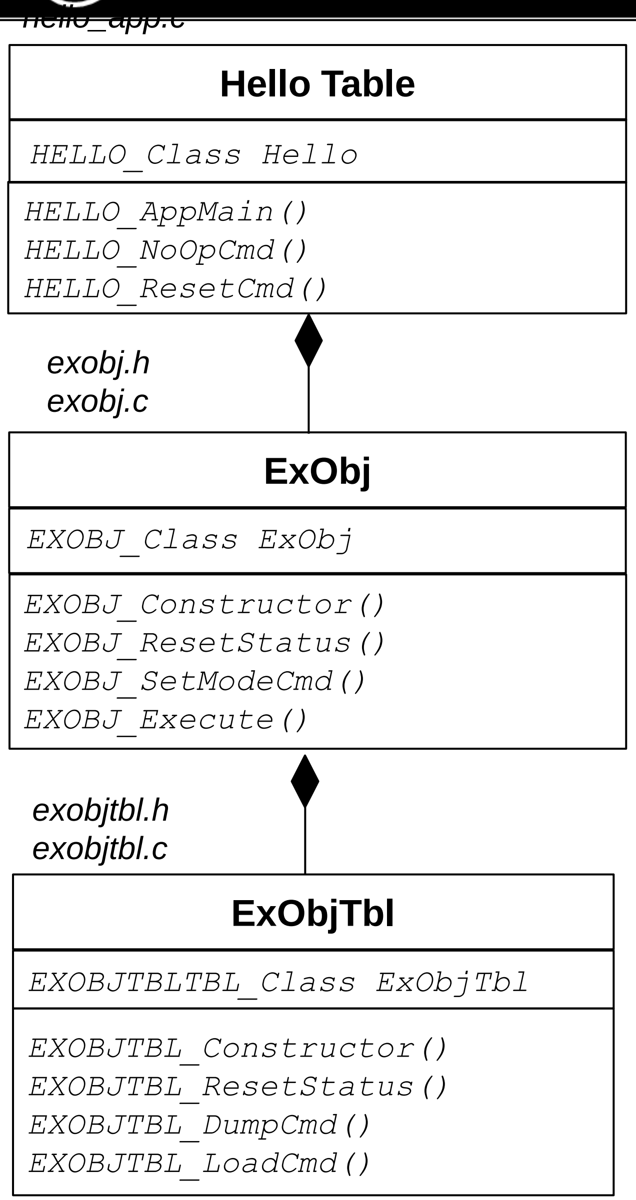
- **Basecamp's JSON tables serve the same purpose as cFS binary tables**
  - Tables are a collection of related parameters that could potentially change during runtime
- **If there are only a couple of parameters then a parameter command may suffice**
- **If there's a very low chance of a parameter changing during runtime and the app can be restarted then the app's init file may suffice**
- **The APP_C_FW table service uses the same JSON parser as app init parameter service**
  - Table parsing includes support for floating point parameters
- **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 <u>not</u> 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 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

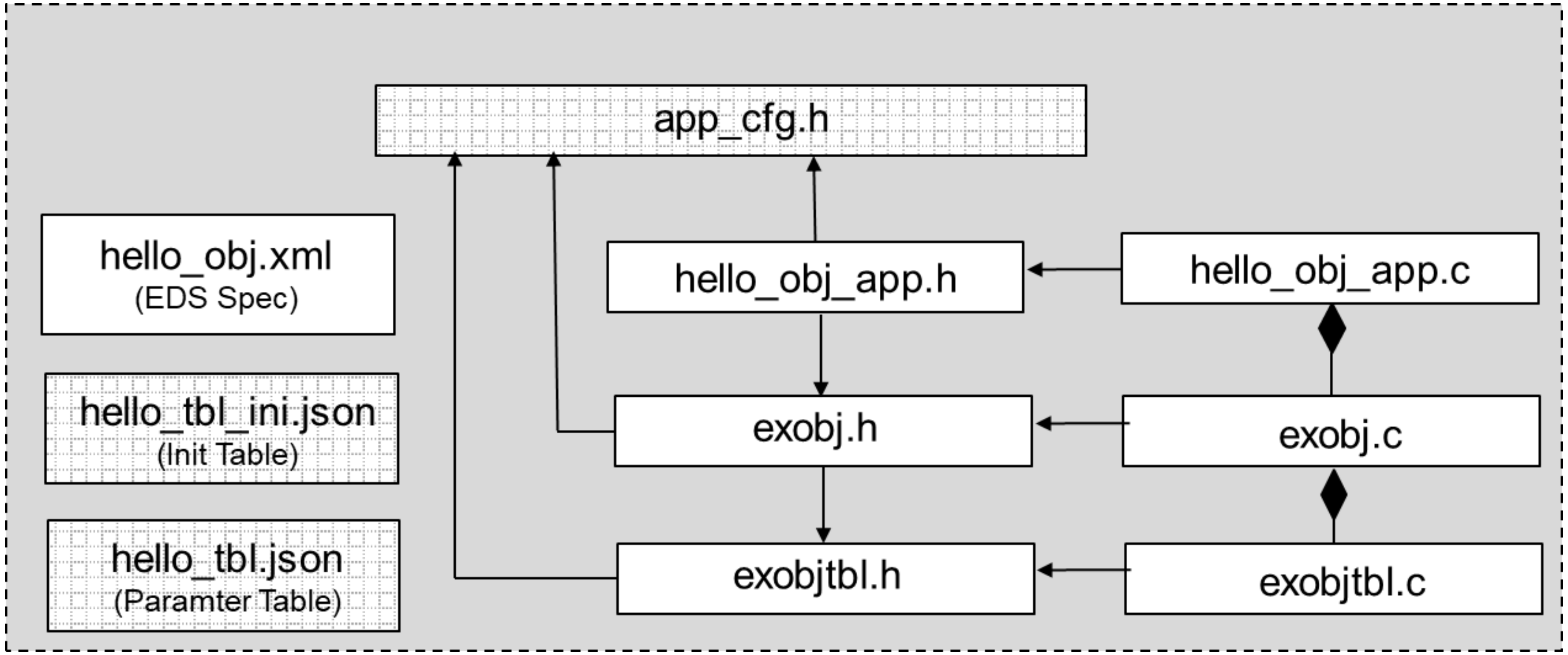
- **app_c_fw defines common command codes for the table load and dump commands**
  - All apps that have tables with use the same command codes just like the Noop and Reset commands

- **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_app.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;
```

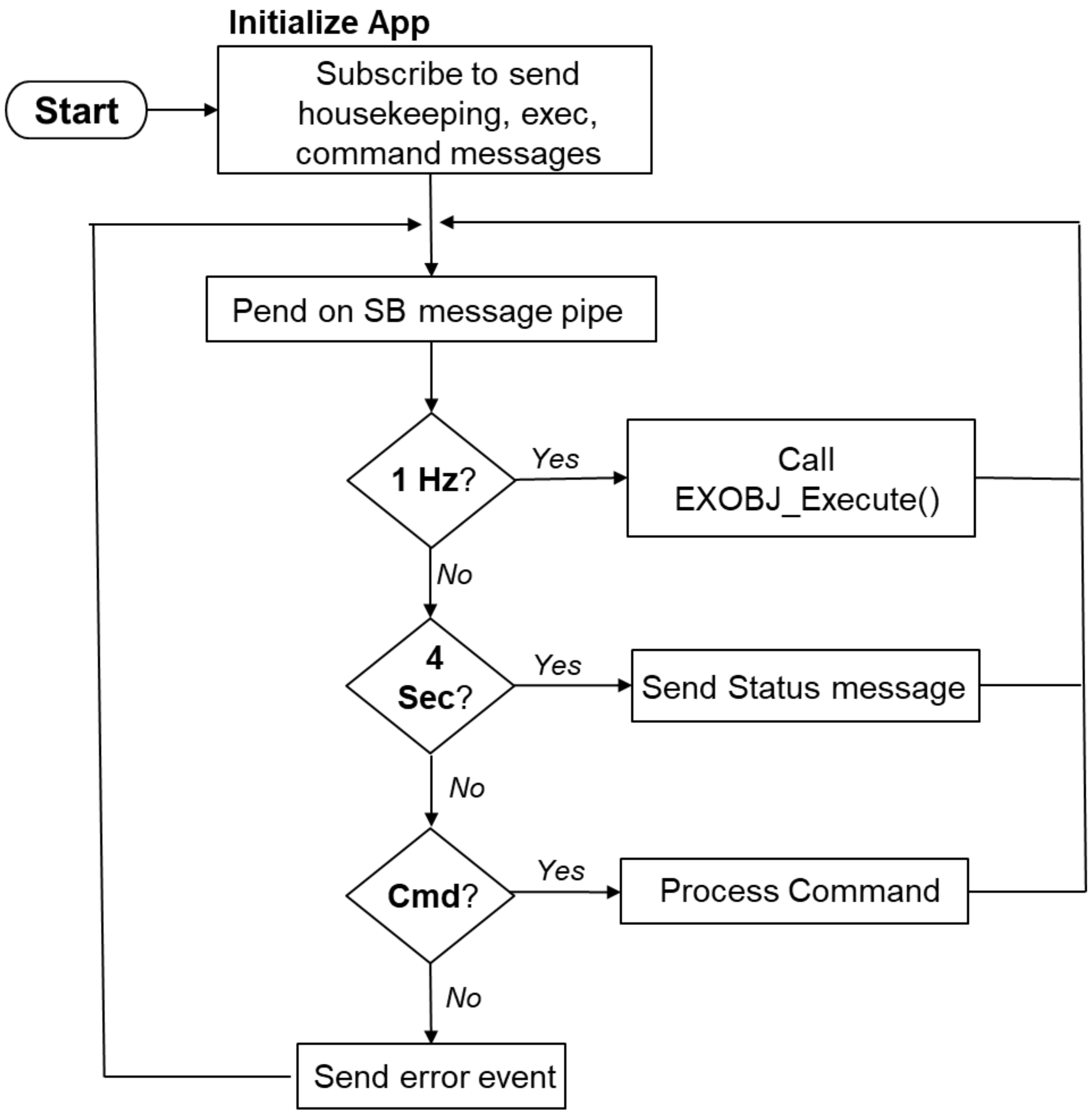
## *exobjtbl.h*

```
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



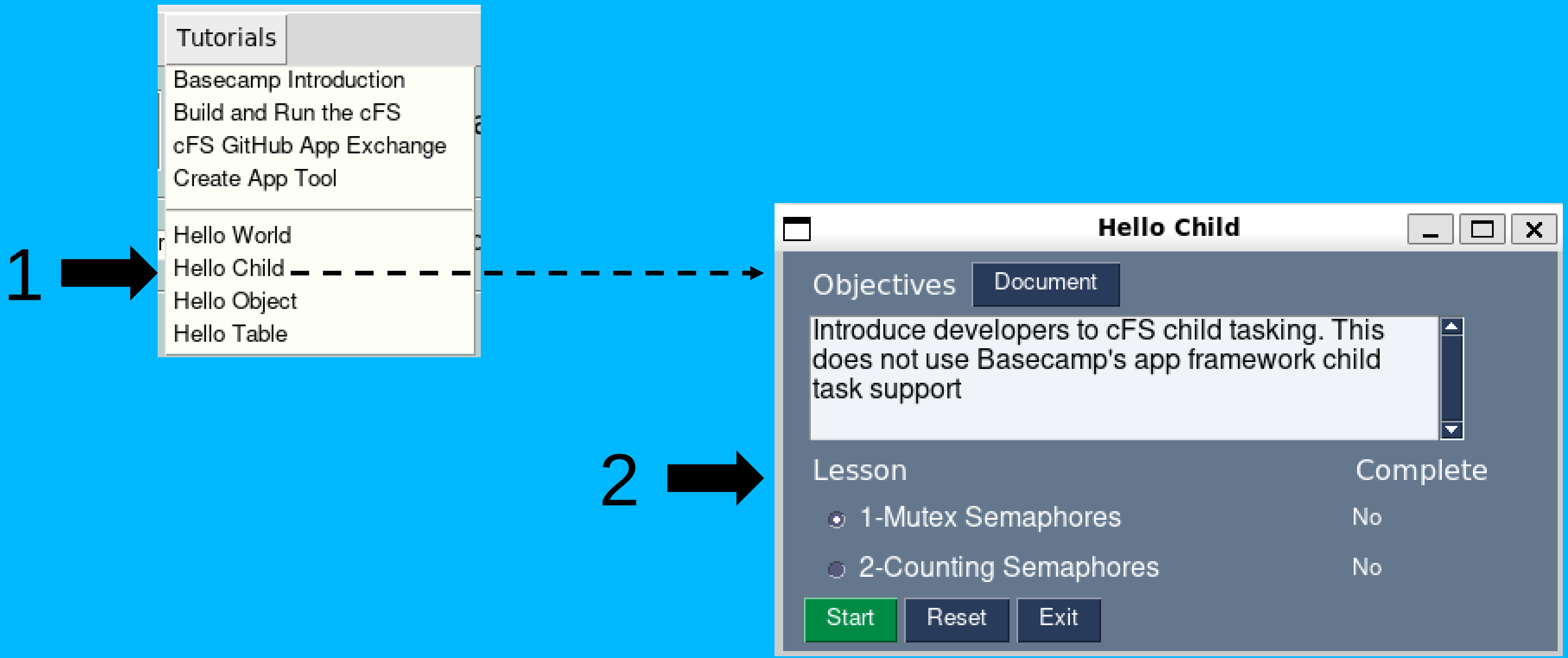
**Same logic as Hello Object**



# cFS Basecamp Exercises: Hello Child



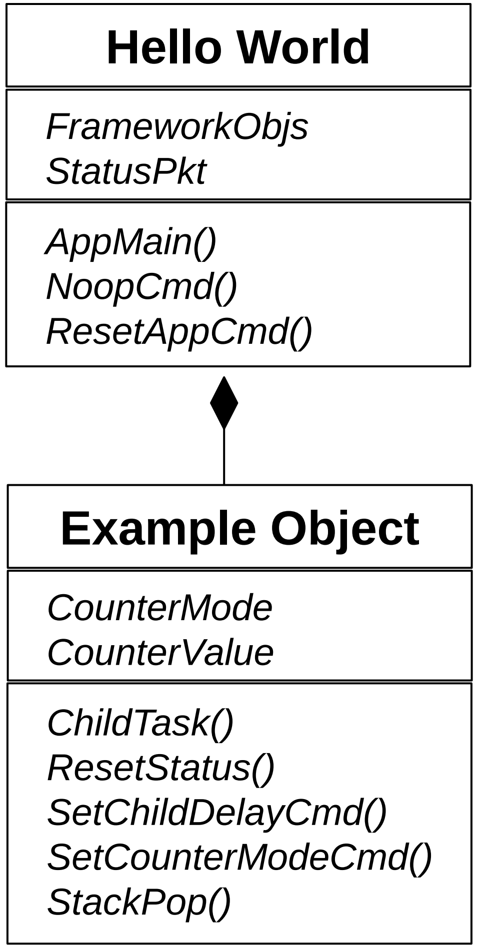
1. Read through this section for a basic understanding
2. From the Tutorial dropdown list select “Hello Child” and do the Lessons
   - Refer back to these slides as needed to deepen your understanding



# 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**
- **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**
  - Established for realtime systems that have strict timing requirements
  - System initialization timing is usually less stringent and dynamic resource management is minimized when the system is operational
  - See the child task ‘Use Cases’ and examples to help guide your decision
- **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

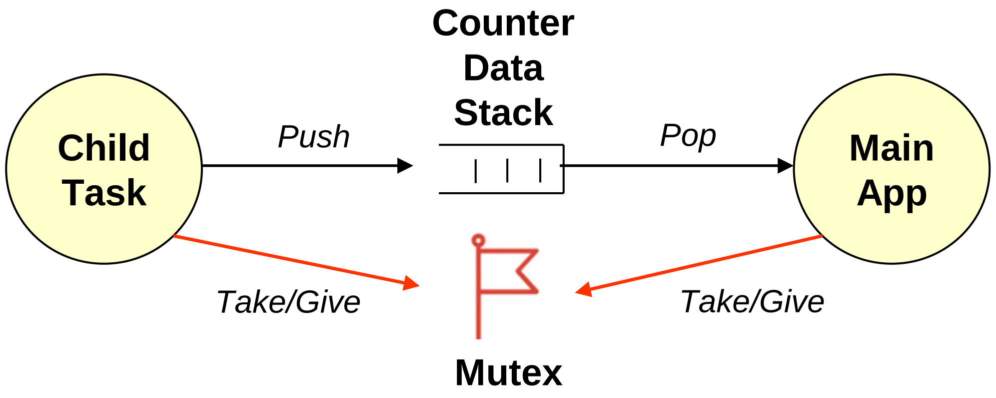
# Hello Child Design

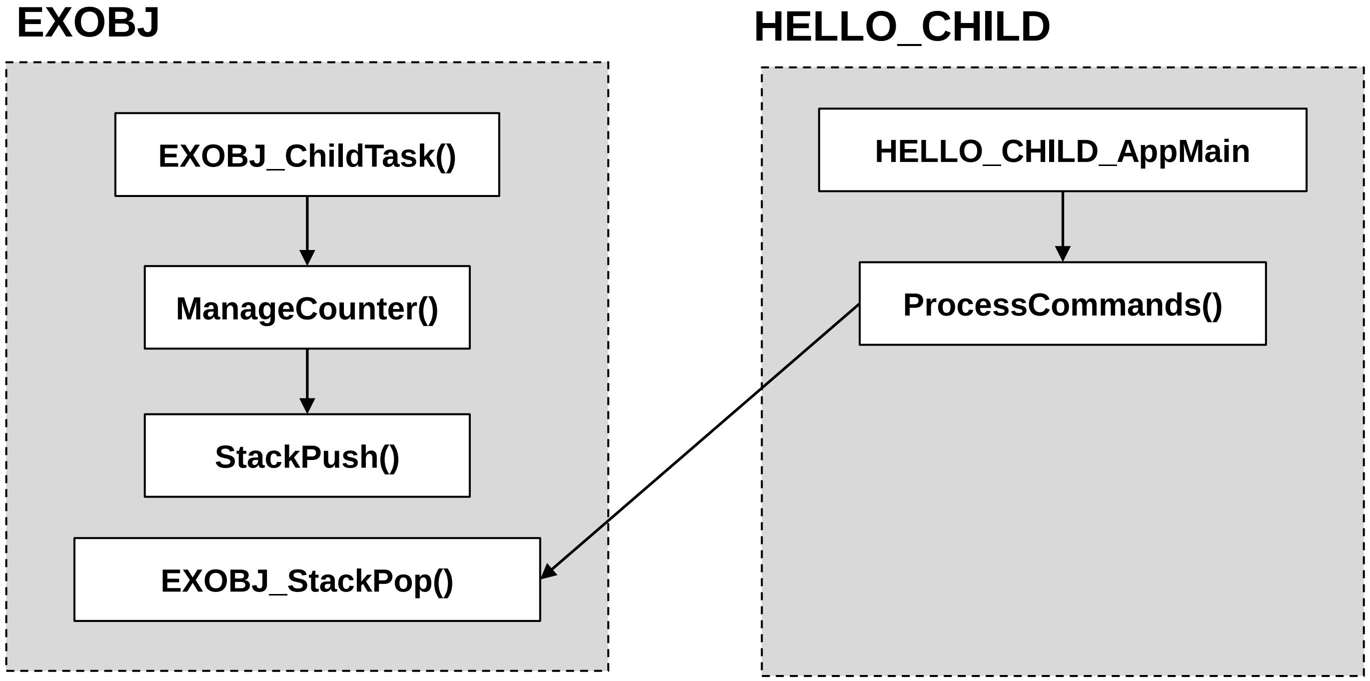



- **The Hello Child app has the same objects as the Hello Object app however the public interface has changed to accommodate running the Execute() function as a child task**
  - EXOBJ_ChildTask() replaces EXOBJ_Execute()

# 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 stack**
- **The cFS naming convention is to prefix global functions with the object's name**

# APP_C_DEMO Overview

# APP_C_DEMO Introduction

- **The APP_C_DEMO app features and design have been specified to provide a non-trivial app that**
  - 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 OSK_C_FW app framework
  - OSK_C_DEMO functions are designed to help teach app development concepts and may not be practical for a space mission
- **This section describes OSK_C_DEMO from an operational perspective so users can use OSK_C_DEMO to learn Base Camp's features**
- **OsK_C_DEMO's design is described in a later section and its design will be used to help developers understand developing apps with the OSK_C_FW**

# OSK_C_DEMO Functions

- **OSK_C_DEMO computes a histogram for a randomly generated integer**
- **The following commands control the app's functionality**
  - Start Histogram
  - Stop Histogram
  - Start Histogram Log
  - Stop Histogram Log
  - Start Histogram Log Playback
  - Stop Histogram Log Playback

# Status Telemetry

- TBD

# Initialization File

- "DEVICE_DATA_MODULO": 100,
- "HIST_LOG_FILE_PREFIX":    "/cf/hist_bin_",
- "HIST_LOG_FILE_EXTENSION": ".txt",
- "HIST_TBL_LOAD_FILE": "/cf/osk_c_hist_tbl.json",
- "HIST_TBL_DUMP_FILE": "/cf/osk_c_hist_tbl~.json"

# Histogram Table File

```
"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
     }
  ]
```

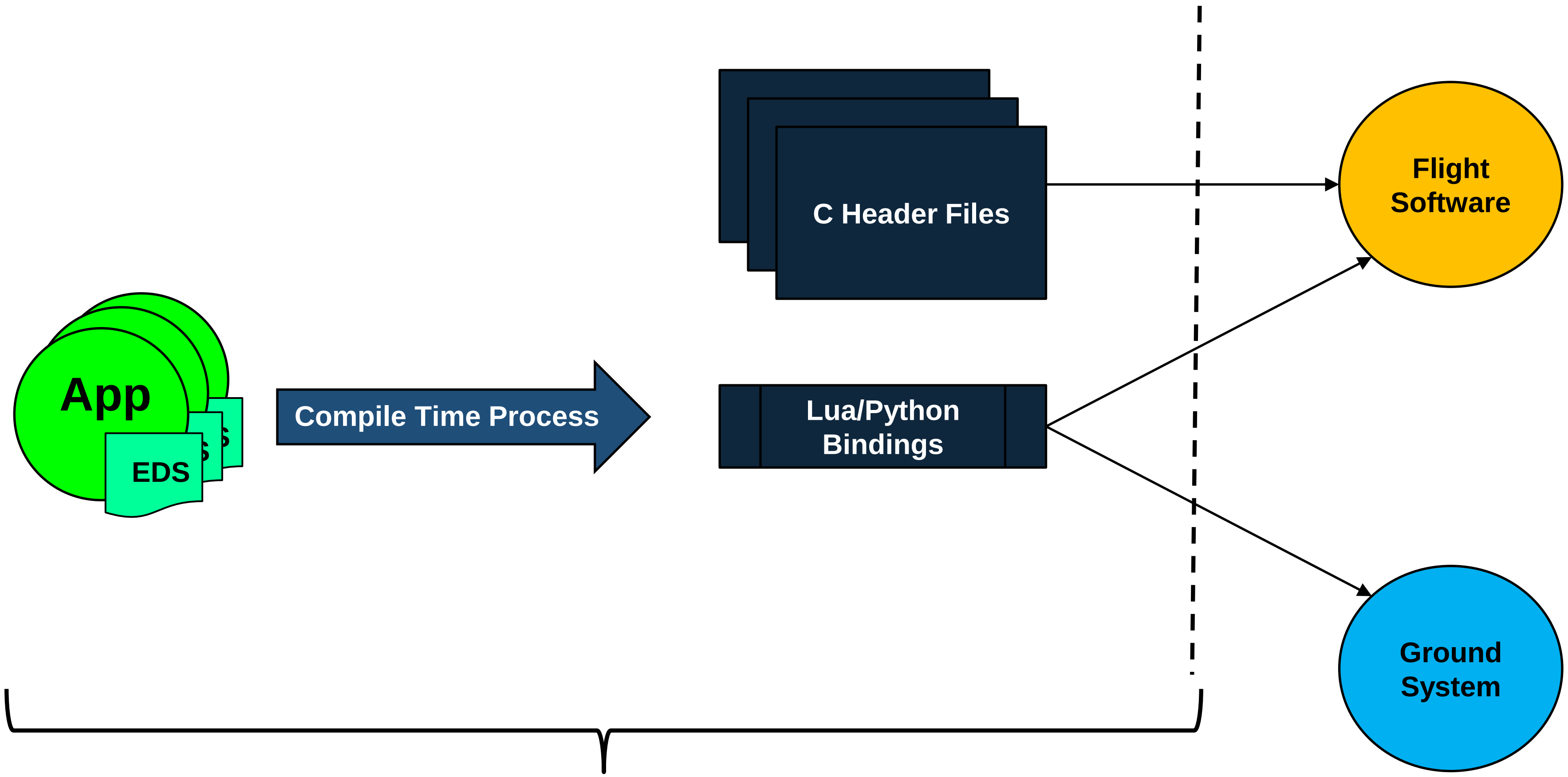
# Electronic Data Sheets

# Overview



- **cfsat_defs**
  - Topicids.xml
  - Config/xml
- **EDS has an app level scope**
  - 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
  - This does not align with the OSK object-based model
    - Naming conventions are not completely followed
    - Global type definition inclusion increases object coupling and reduces information hiding
  - #include “<app>_eds_cc.h“ in app’ main c file

# Electronic Data Sheet Workflow




Single definition of data in EDS propagates to rest of system.

- EDS overview and global definitions
- OSK App EDS file organization & conventions
- Topic ID tool
- EDS conventions and tips for developing your code
- Since commands use function codes within a single command message the naming convetions differs from the telemetry messages.

# osk_def Directory

- **Targets.cmake**
  - Identifies the target architectures and configurations
  - Identifies the apps to be built
  - Identifies files that will be copied from sample_def to platform specific directories

- **Copied file examples**
  - cpu1_cfe_es_startup.scr
  - cpu1_msgids.h
  - Cpu1_osconfig.h

Describe topicids tool

# Application Framework Architecture

# Basecamp Application Framework Introduction



- **Motivation**
  - Since the cFS is a message-based system many apps have a common control and data flow structure
  - A common object-based framework (written in C) 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 publish and exchanged
  - Coupling and cohesion are not easy to measure and often reveal themselves during maintenance. When you make a change observe how th change is manifested. Is it localized?  How many components are impacted? Are details encapsulated behind an API?
    - See the File Manager refactor analysis section for how APP_C_FW-based design can improve these attributes
- ***app_c_demo*** **is used as a concrete example to help users use this document**
  - It is part of Basecamp's default app suite so users can immediately start to interact with it
  - It's a non-trivial app that performs onboard data processing functions and its design were intentionally chosen to help users understand an ap that will most likely be a part of their mission
- **This document relies on consistent versioning and compatibility between the following Basecamp components that each have their own git repos**
  - cfe-eds-framework: Defines the core Flight Executive (cFE) Electronic Data Sheet (EDS) specs
  - app_c_fw: OpenSatKit application framework library
  - app_c_demo: Example app that shows best practices for using osk_c_fw and creating apps that can be published and shar

# Introduction



- **The OSK C Application Framework is light-weight object-based framework for writing cFS applications in C**
  - The framework library is named osk_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
- **OSK_C_DEMO is a fully functioning cFS app that is delivered as part of OSK's Research & Development (R&D) Sandbox target**
  - Uses many of osk_c_fw's features and serves as the end-goal for the app development tutorial
  - This guide uses it as a reference app implementation to illustrate how osk_c_fw is used

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

# Object-based Design 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_".
- **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 <u>not</u> 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)

# Object-based Design 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)**

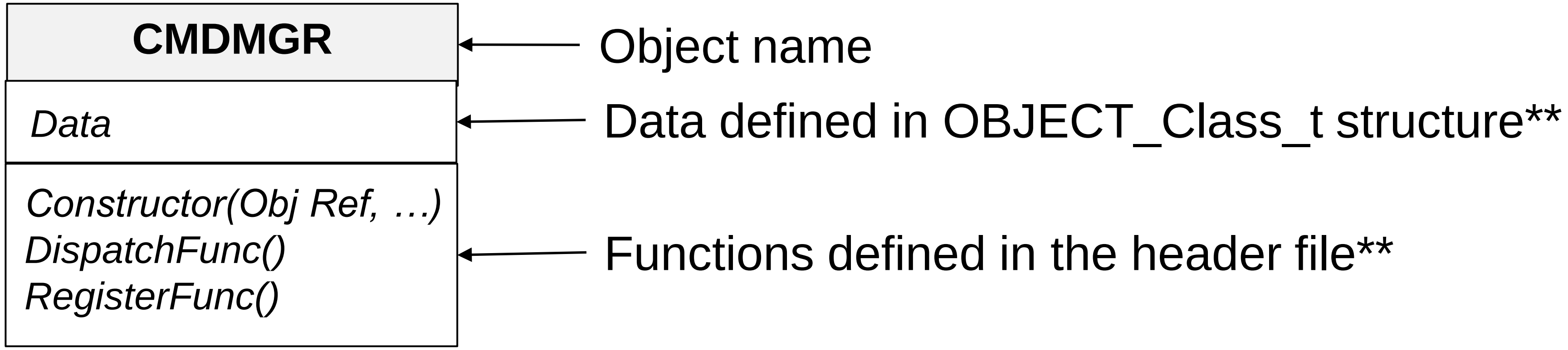
- **The source file may also include local definitions for constants, typedefs and functions**
  - Local names should be meaningful and may follow a local naming convention, but they should not be prefixed with the object's global name prefix. This makes it easy for someone reading the code to immediately 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.

# Object-based Design Conventions (3 of 3)

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




** 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.

OpenSTEMware

# Configuration Strategy

Basecamp

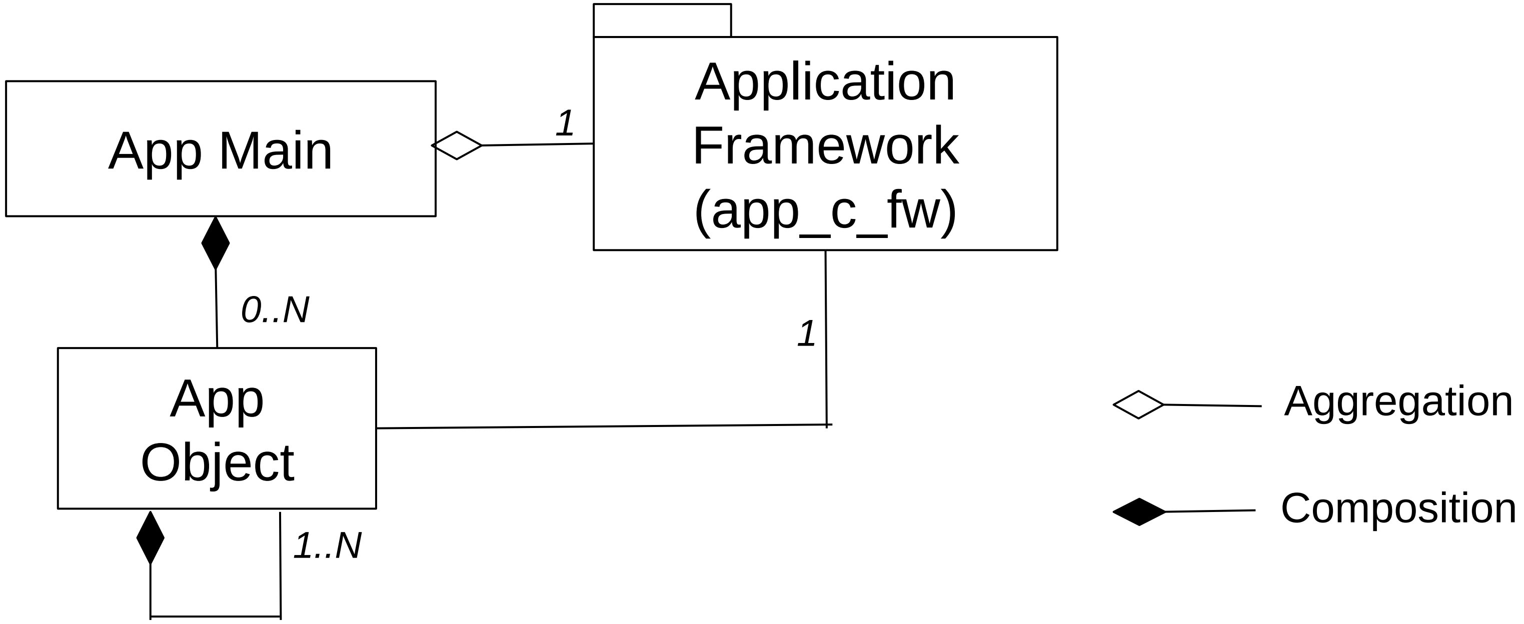
- **Build time**
  - Application -
  - Deployment – Mission tuning
- **Runtime**
  - Initialization
  - Runtime

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

# OSK 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
  - Conceptually one app_c_fw exists for all applications
- **Composition** represents a relationship where the contained cannot exist independent of the owner
  - Application objects exists to provide behavior and functionality and they only exist within the context of the application
- These are conceptual definitions, from an implementation perspective an application is the hierarchical aggregation of objects

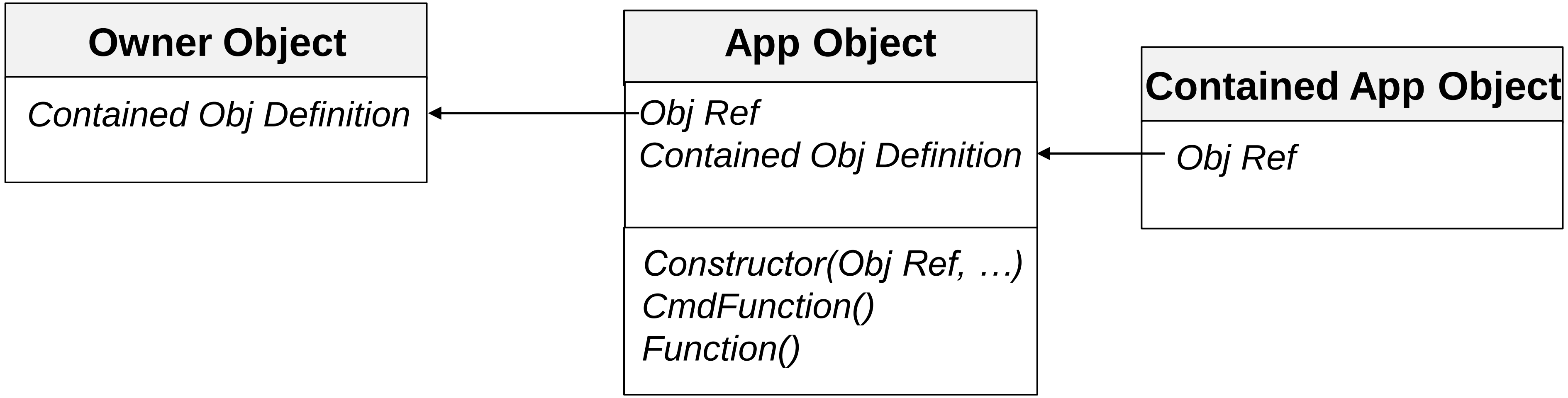
# Application Framework Components

| 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 |
| **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. |
| **File Utility[1]** | fileutil | Utilities for verifying and manipulating files |
| **Packet Utility[1]** | pktutil | Utilities for verifying and manipulating packets |
| **CJSON** | cjson | Adapter for interfacing to the FreeRTOS coreJSON library |
| **JSON[2]** | json | Adapter for interfacing to the JSMN JSON library |

1. Collection of functions that don't have class data (i.e., stateless)
2. This will be deprecated once all of the JSON tables are converted to use cjson

# App Object Design (1 of 2)

- ***App Objects*** **implement the required behavior and functions for an app**
- **Objects should be designed to represent a single concept represented by its name**
  - Contain properties (data) and methods(functions) that are intrinsic to the scope and responsibilities of that concept
- **The figure below shows the object composition model**

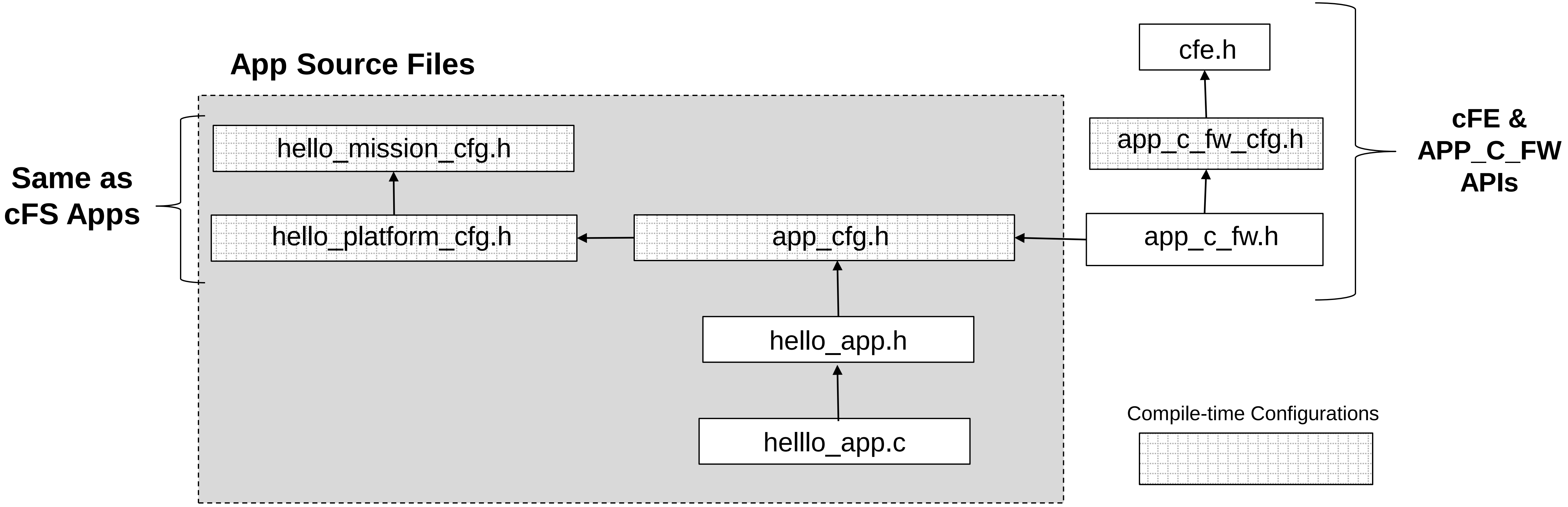



- **Owner objects define the data for objects they contain and pass a reference to the contained object's constructor**
- **Contained objects store a reference to the owner's instance data**
  - Only one instance of an object modeled after the App Object design pattern can exist in an app
  - Analogous to the OO Singleton design pattern without any wrapper protection

# App Object Design (2 of 2)

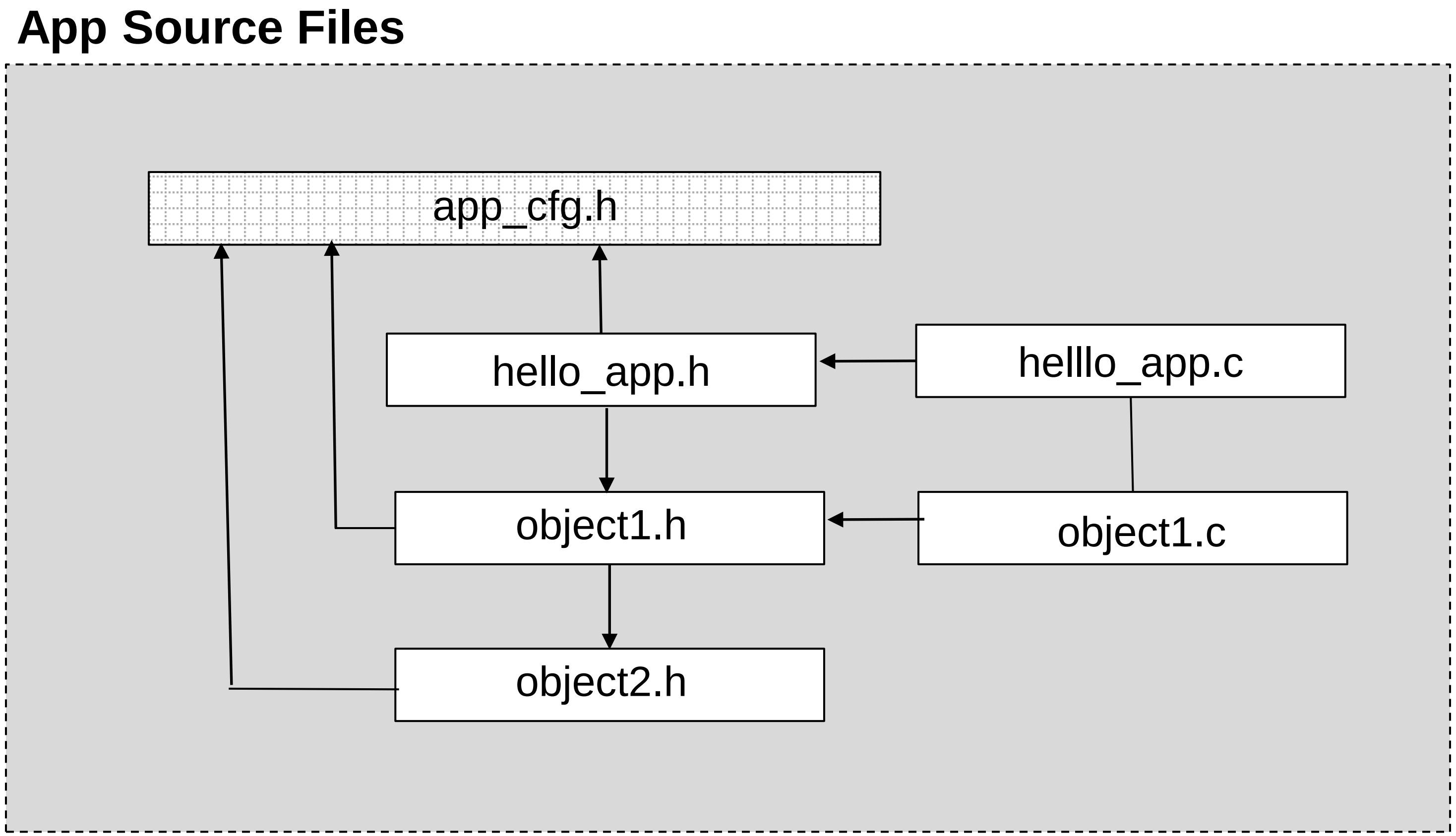
- **Public App Object functions (or methods) fall into two categories**
  - Command functions are executed when the parent app receives a command message on the software bus that contains the function’s command function code
    - Command functions are registered by the main app during initialization
  - All other functions are called by the main app or by other app 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
  - The other public app object functions define the app object’s public interface within an app
- **App Objects can create Software Bus interfaces as needed**
- **Relative event message ID numbering is used within each App Object**
  - Ranges of IDs are managed at the application level
- **Table objects are a specialization of an App Object that 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, but developers should not feel constrained by the model if it doesn’t fit a particular situation

# Minimal Header File Inclusion Tree



# Minimal Header File Inclusion Tree





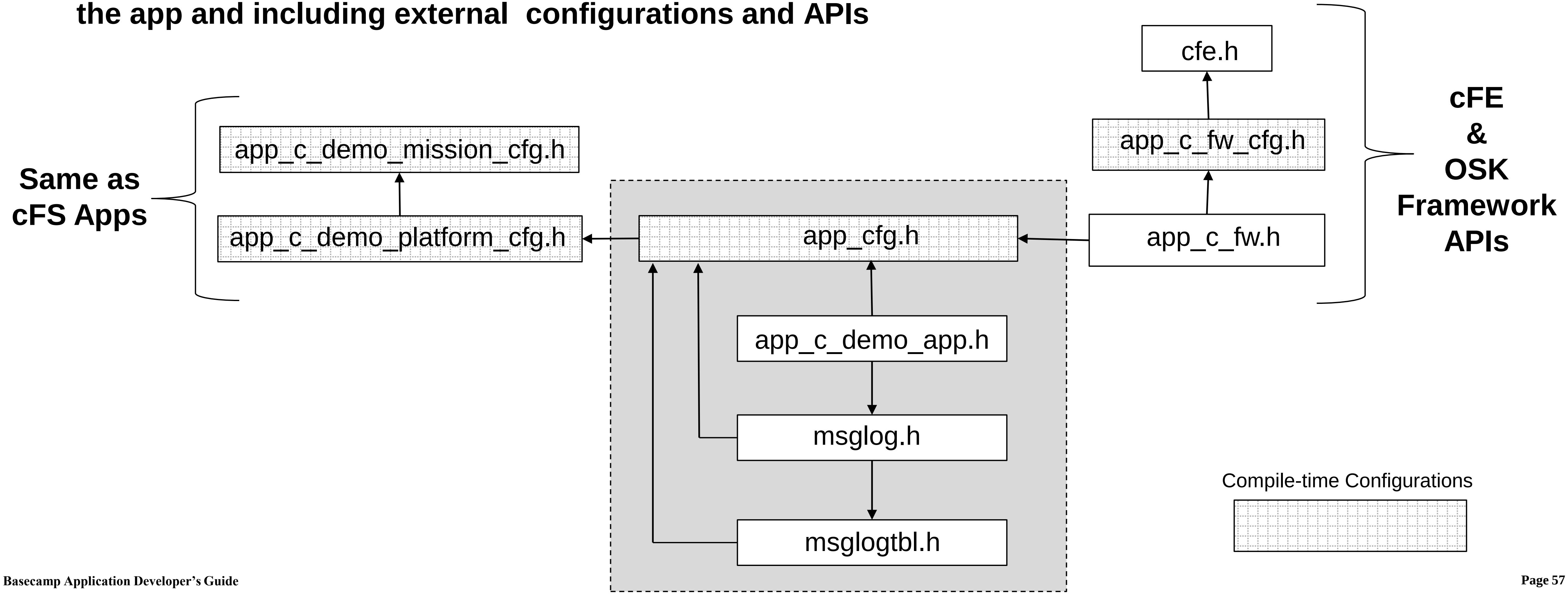
# Object Composition Model – Header File Overview

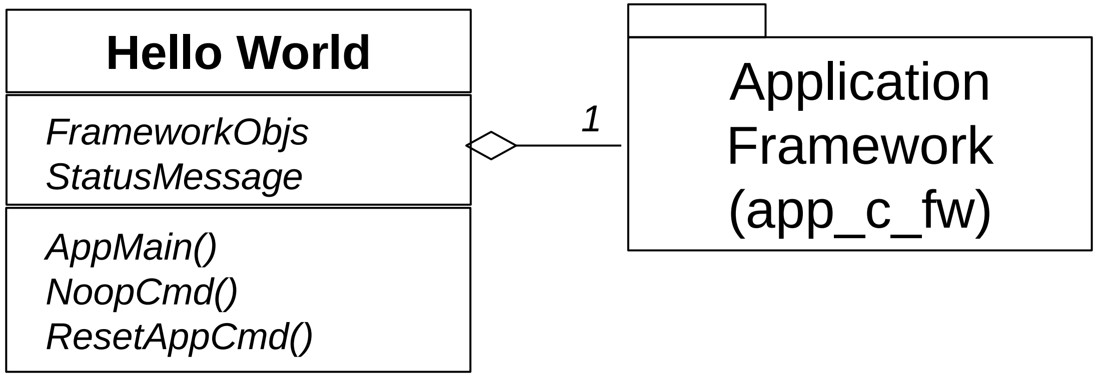


| Header File | Purpose |
| --- | --- |
| **osk_c_demo_mission_cfg.h** | Analogous to cFS app mission config header in scope |
| **osk_c_demo_platform_cfg.h** | Analogous to cFS app platform config header in scope, but very few if any parameters should be defined in this header due to other Basecamp app configuration features |
| **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. |
| **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 sharing the framework on a platform. |
| **cfe.h** | Defines the cFE API and included by the framework so Basecamp definitions can build on cFE definitions. |
| **osk_c_demo_app.h** | Demo app's "class structure" that's serves as the root of the object hierarchy |
| **msglog.h** | Example App Object named message Log. osk_c_demo is its owner and msglogtbl is its contained object |
| **msglogtbl.h** | Adapter for interfacing to the FreeRTOS core-JSON library |

# Object Composition Model – Header Files Inclusion Tree

- **The osk_c_demo app will be used to show a concrete example of the app object composition model**
  - osk_c_demo is covered in detail in a later section and at this step detailed knowledge is not required
- **osk_c_demo's header inclusion tree shows the app's structure and dependencies**
- **Every app has an app_cfg.h file that serves as the single point for configuring structural aspects of the app and including external  configurations and APIs**




Hello World
FrameworkObjs
StatusMessage
AppMain()
NoopCmd()
ResetAppCmd()
1
Application
Framework
(app_c_fw)

# Object Composition Model – Demo App



## 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

# 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**

# 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

# 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,…)*

UNDER CONSTRUCTION Need to capture all OSK's coding idioms and styles

# Configuration Parameter Summary

| Configuration | Configuration Scope |
| --- | --- |
| **osk_c_fw_cfg.h** | Defines platform-scoped configuration parameters for the OSK framework. The defaults should accommodate most deployments. The configurations must meet the needs of all apps sharing the framework on a platform. |
| **xxx_mission_cfg.h** | Defines mission-scoped application configurations. These configurations apply to every app deployment on different platforms within a single mission. |
| **xxx_platform_cfg.h** | Defines platform-scoped application configurations. Analogous to cFS app platform config header in scope, but very few if any parameters should be defined in this header due to app_cfg.h and IniTbl configuration options |
| **app_cfg.h** | Every OSK app has a header with this name. Configurations have an application scope that define compile-time parameters that typically don't change across deployments. |
| **Initialization Table** | Defines configuration parameters that can be defined at runtime.  For example, command pipe name, command pipe depth, and command message identifier. |
| **Table & Commands** | 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 osk_c_demo description. |

# App

# Initialization Table

# Initialization Table Introduction (1 of 2)



- **Initialization tables are JSON files that define application runtime configurations**
  - If a configuration parameter impacts a data structure, then it must be defined in a header file at the appropriate scope
- **Some 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
- **Some challenges with using JSON files read during initialization include**
  - JSON doesn't support comments
    - Later slides describe some conventions that help overcome this challenge
  - When two apps need the same parameter such as a message ID then it must be defined twice
    - Basecamp uses a tool to eliminate this issue
    - Each message ID is defined once and the tool populates the initialization tables

# Initialization Table Introduction (2 of 2)

- **File is read in during application initialization**
  - JSON table filename is defined in app's xxx_platform_cfg.h
- **"config" JSON object contains the key-value pair definitions**
- **Keys are defined in app's app_cfg.h**
- **Currently supports integer and strings types**
- **Easy coding steps to define and use an initialization table**
  - Implementation details abstracted and hidden from the user

## osk_c_demo_ini.json

```
{
   "title": "OSK C Demo initialization file",
   "description": [ "Define runtime configurations"]
   "config": {

      "APP_CFE_NAME": "OSK_C_DEMO",
      "APP_PERF_ID":  127,

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

      "CMD_MID":        8048,
      "EXECUTE_MID":    6593,
      "SEND_HK_MID":    6594,
      "HK_TLM_MID":     3952,
      "PLAYBK_TLM_MID": 3953,

      "CMD_PIPE_DEPTH": 5,
      "CMD_PIPE_NAME":  "OSK_C_DEMO_CMD",

      "MSGLOG_PIPE_DEPTH": 5,
      "MSGLOG_PIPE_NAME":  "OSK_C_DEMO_PKT",

      "TBL_LOAD_FILE": "/cf/osk_c_demo_tbl.json",
      "TBL_DUMP_FILE": "/cf/osk_c_demo~.json"

   }
}
```

# Define App Initialization Parameters (1 of 2)



## 1a. Define configurations in app_cfg.h

```
#define CFG_MSGLOG_PIPE_DEPTH   MSGLOG_PIPE_DEPTH
#define CFG_MSGLOG_PIPE_NAME    MSGLOG_PIPE_NAME

#define CFG_TBL_LOAD_FILE       TBL_LOAD_FILE
#define CFG_TBL_DUMP_FILE       TBL_DUMP_FILE
```

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) \
   XX(CHILD_NAME,char*) \
   XX(CHILD_PERF_ID,uint32) \
   XX(CHILD_STACK_SIZE,uint32) \
   XX(CHILD_PRIORITY,uint32) \
   XX(CMD_MID,uint32) \
   XX(EXECUTE_MID,uint32) \
   XX(SEND_HK_MID,uint32) \
   XX(HK_TLM_MID,uint32) \
   XX(PLAYBK_TLM_MID,uint32) \
   XX(CMD_PIPE_NAME,char*) \
   XX(CMD_PIPE_DEPTH,uint32) \
   XX(MSGLOG_PIPE_DEPTH,uint32) \
   XX(MSGLOG_PIPE_NAME,char*) \
   XX(TBL_LOAD_FILE,char*) \
   XX(TBL_DUMP_FILE,char*) \

DECLARE_ENUM(Config,APP_CONFIG)
```

Add the XXX definition to APP_CONFIG macro and declare the type: uint32 or char*

# Define App Initialization Parameters (2 of 2)



## 1b. 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. Define IniTbl object in the app's main class

```
typedef struct {

   /*
   ** App Framework
   */

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

**IniTbl Definition**

## 1d. Add the JSON filename to the appropriate "FILELIST" in targets.cmake

# Use the App Initialization Table



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

```
INITBL_Constructor(&OskCDemo.IniTbl, OSK_C_DEMO_INI_FILENAME, &IniCfgEnum)
```

## 2b – Retrieve parameter values using CFG_XXX macro and INITBL's Integer or String get functions

```
CFE_SB_CreatePipe(&OskCDemo.CmdPipe, INITBL_GetIntConfig(INITBL_OBJ, CFG_CMD_PIPE_DEPTH),
                  INITBL_GetStrConfig(INITBL_OBJ, CFG_CMD_PIPE_NAME));
```

**Notes**

- 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 osk_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.

# App

# Commands

- Standard commands: noop, reset (describe how different than NASA), load, dump tables
- Every app should have a noop
- 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
- Add a telemetry design section
- Get notes from my cFE slides and system slides

# Command Manager Overview

| CmdMgr |
| --- |
| *Command Counters* |
| *Constructor()*<br>*RegisterFunc()*<br>*RegisterAltFunc()*<br>*ResetStatus()*<br>*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 concise

# Using 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);
```

# App

# Telemetry

- **. App defines a 'send/request HK packet message ID' and subscribes to receive the message. Typical on app's command pipe**
- **2. Add message to scheduler's message table and add a scheduler table entry to send the message. HK packet at some interval.**
- **3. Process the packet in the app's main loop. File manager fm_app.c is a good example; FM_ProcessPkt(). Since FM only runs in response to commands, it pends indefinitely on its command pipe, other apps may poll their command pipe.**
- 
- **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.**
- 

# App

# Events

- **Describe event message strategies**
- 

# App

# Tables

# Table Manager Overview



| TblMgr |
|---|
| *Load/Dump Status* |
| *Constructor()*<br>*RegisterTbl ()*<br>*RegisterTblWithDefs()*<br>*LoadTblCmd()*<br>*DumpTblCmd()*<br>*ResetStatus()*<br>*GetLastStatus()* |

- **Provides a table registration service and manages table loads and dumps**
- **Tables are defined in JSON text files**
- **Tables are parsed using an open-source JSON library**
  - In v3.1 FreeRTOS core-JSON parser was added
  - Prior to v3.1 JSMN was used
- **osk_c_fw uses adapter objects to interface with the parser**
  - json.h interfaces with JSMN
  - cjson interfaces with core-JSON
- **osk_c_demo is the first app to use cjson and the other apps with be transitioned in future releases**
- **A table object must be defined for each table**
  - The table object provides table-specific load/dump functionality
  - It defines a local table data buffer for loads

# OSK Application JSON Tables

- **Objectives**
  - Provide a text-based table service
  - Create a consistent application JSON table management operational interface
  - Facilitate consistent application designs that abstract complexities, minimize application developer learning curves and simplify maintenance
- **Rationale**
  - cFE binary tables require an added layer of ground processing for translating between binary tables and human readable/writable text
- **OSK C application framework (osk_c_fw) JSON file management**
  - Utilities for parsing JSON files
  - Functional API for retrieving JSON-defined values
  - Design is independent of table concept/design
- **Application object design pattern**
  - Defines an object-based design for using the framework utilities to manage loading and dumping JSON table files

# Using Table Manager



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

```
TBLMGR_Class     TblMgr;
```

2. **App Init: Construct the TblMgr object**

```
TBLMGR_Constructor(TBLMGR_OBJ);
```

3. **App Init: Register app's tables with TblMgr (these are table object's callback functions)**

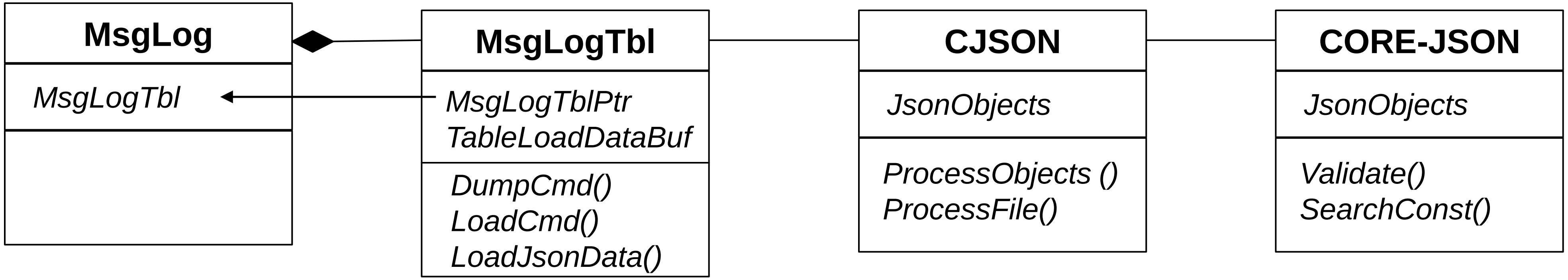
```
TBLMGR_RegisterTblWithDef(TBLMGR_OBJ, MSGLOGTBL_LoadCmd, MSGLOGTBL_DumpCmd,
                          INITBL_GetStrConfig(INITBL_OBJ, CFG_TBL_LOAD_FILE));
```

4. **App Init: Register TblMgr's Load and Dump commands with CmdMgr**

```
CMDMGR_RegisterFunc(CMDMGR_OBJ, OSK_C_DEMO_TBL_LOAD_CMD_FC, TBLMGR_OBJ,
                    TBLMGR_LoadTblCmd, TBLMGR_LOAD_TBL_CMD_DATA_LEN);
CMDMGR_RegisterFunc(CMDMGR_OBJ, OSK_C_DEMO_TBL_DUMP_CMD_FC, TBLMGR_OBJ,
                    TBLMGR_DumpTblCmd, TBLMGR_DUMP_TBL_CMD_DATA_LEN);
```

5. **Implement the table app object**
   - The following slides use MsgLogTbl as an example to show to create a table object

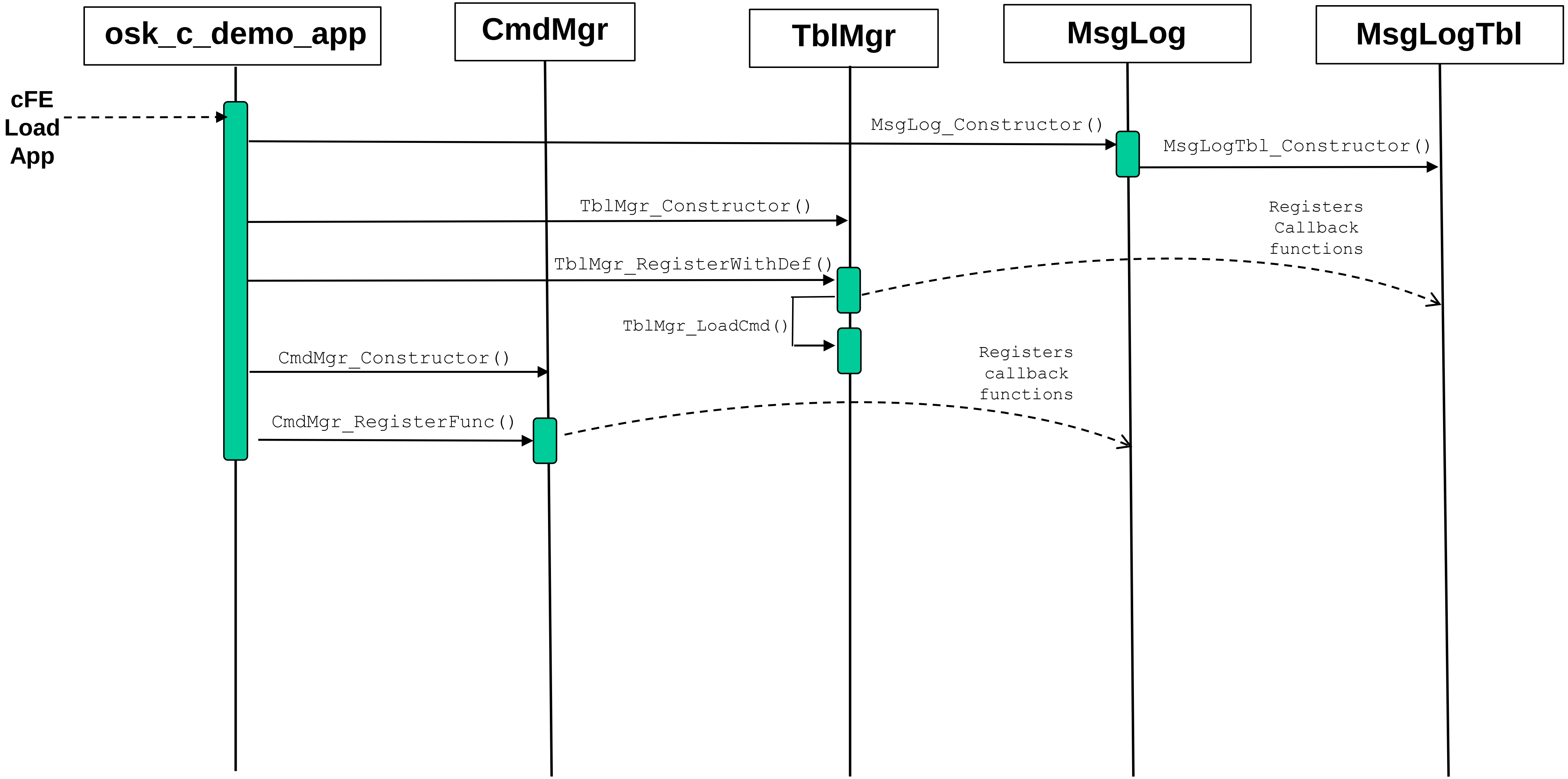
# Table Manager Object Design



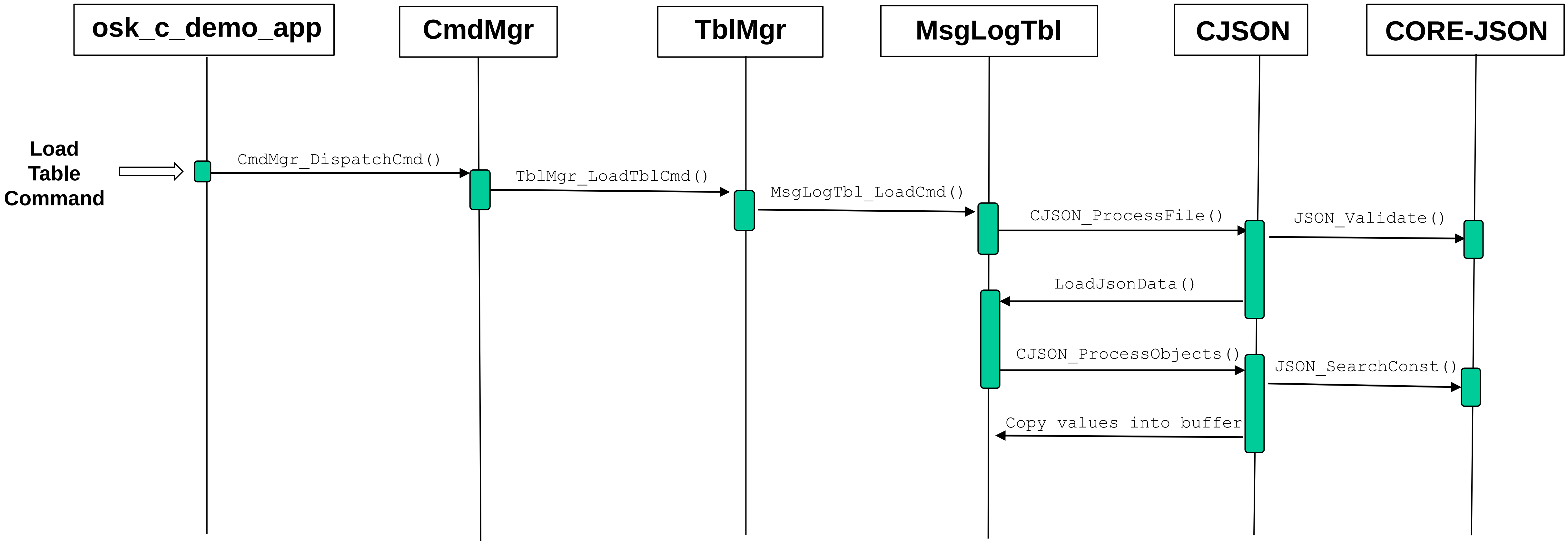

- ***MsgLog* is the parent of *MsgLogTbl* so it contains an instance of *MsgLogTbl***
- ***MsgLogTbl***
  - *MsgLogTblPtr* references *MsgLog's* instance of *MsgLogTbl*
  - *TableLoadDataBuf*  stores table load data and its contents are copied to *MsgLog's* instance if the table load is successful
  - *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*
  - *ProcessObjects()* loops through the *MsgLogTbl's CJSON_Obj array* to populate *MsgLogTbl's TableLoadDataBuf* with the JSON defined values
  - *ProcessFile()* validates the JSON file and calls the user supplied callback function to coy data into it's table load buffer. *LoadJsonData()* is the callback for *MsgLogTbl.*
- ***CORE-JSON is an open-source parser provided by the FreeRTOS project***
  - *Validate()* validates a JSON structure passed in a character buffer
  - *SearchCOnst()* searches for a key uses a dot notation for nested JSON objects. See core-json.h for details.

# Table Initialization Sequence Diagram

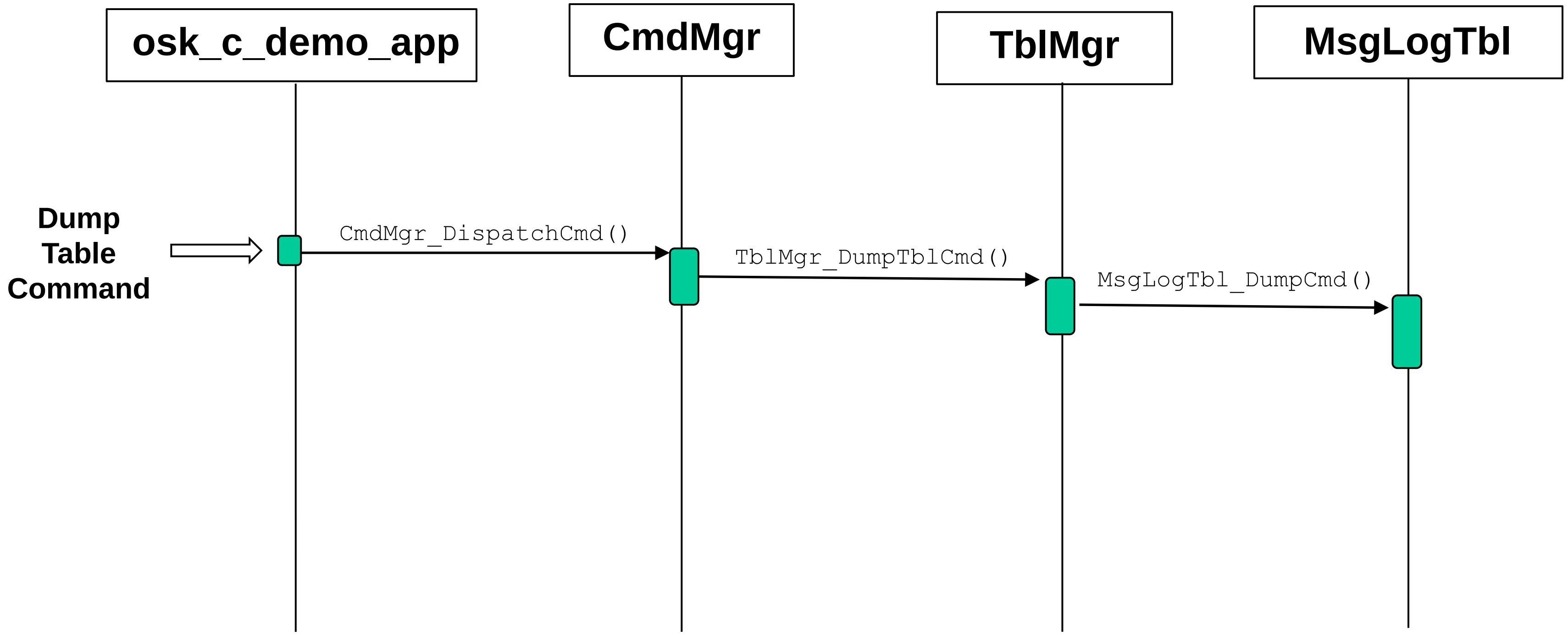




# Load Table Sequence Diagram



# Dump Table Sequence Diagram



# MsgLogTbl Highlights

## osk_c_demo_tbl.json

```
{
   "app-name": "OSK_C_DEMO",
   "tbl-name": "Message Log",
   "description": "Define parameters for demo message logger",
   "file": {
      "path-base-name": "/cf/msg_",
      "extension":      ".txt",
      "entry-cnt":      5
   },
   "playbk-delay": 3
}
```

## msglogtbl.c's JSON object definitions maps C structure to JSON objects

```
static CJSON_Obj JsonTblObjs[] = {

   /* Table Data Address        Table Data Length           Updated, Data Type,   core-json query string, length of query string */

   { TblData.File.PathBaseName, OS_MAX_PATH_LEN,            FALSE,   JSONString, { "file.path-base-name", strlen("file.path-base-name")} },
   { TblData.File.Extension,    MSGLOGTBL_FILE_EXT_MAX_LEN, FALSE,   JSONString, { "file.extension",      strlen("file.extension")}      },
   { &TblData.File.EntryCnt,    3,                          FALSE,   JSONNumber, { "file.entry-cnt",      strlen("file.entry-cnt")}      },
   { &TblData.PlaybkDelay,      2,                          FALSE,   JSONNumber, { "playbk-delay",        strlen("playbk-delay")}        }

};
```

## MSGLOGTBL_LoadCmd(), the table load callback function, calls

```
CJSON_ProcessFile(Filename, MsgLogTbl->JsonBuf, MSGLOGTBL_JSON_FILE_MAX_CHAR, LoadJsonData)
```

## LoadJsonData(), the CJSON process file callback, calls

```
CJSON_LoadObjArray(JsonTblObjs, MsgLogTbl->JsonObjCnt, MsgLogTbl->JsonBuf, MsgLogTbl->JsonFileLen)
```

# 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**

# App

# Child Tasks

# 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're being executed
- **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

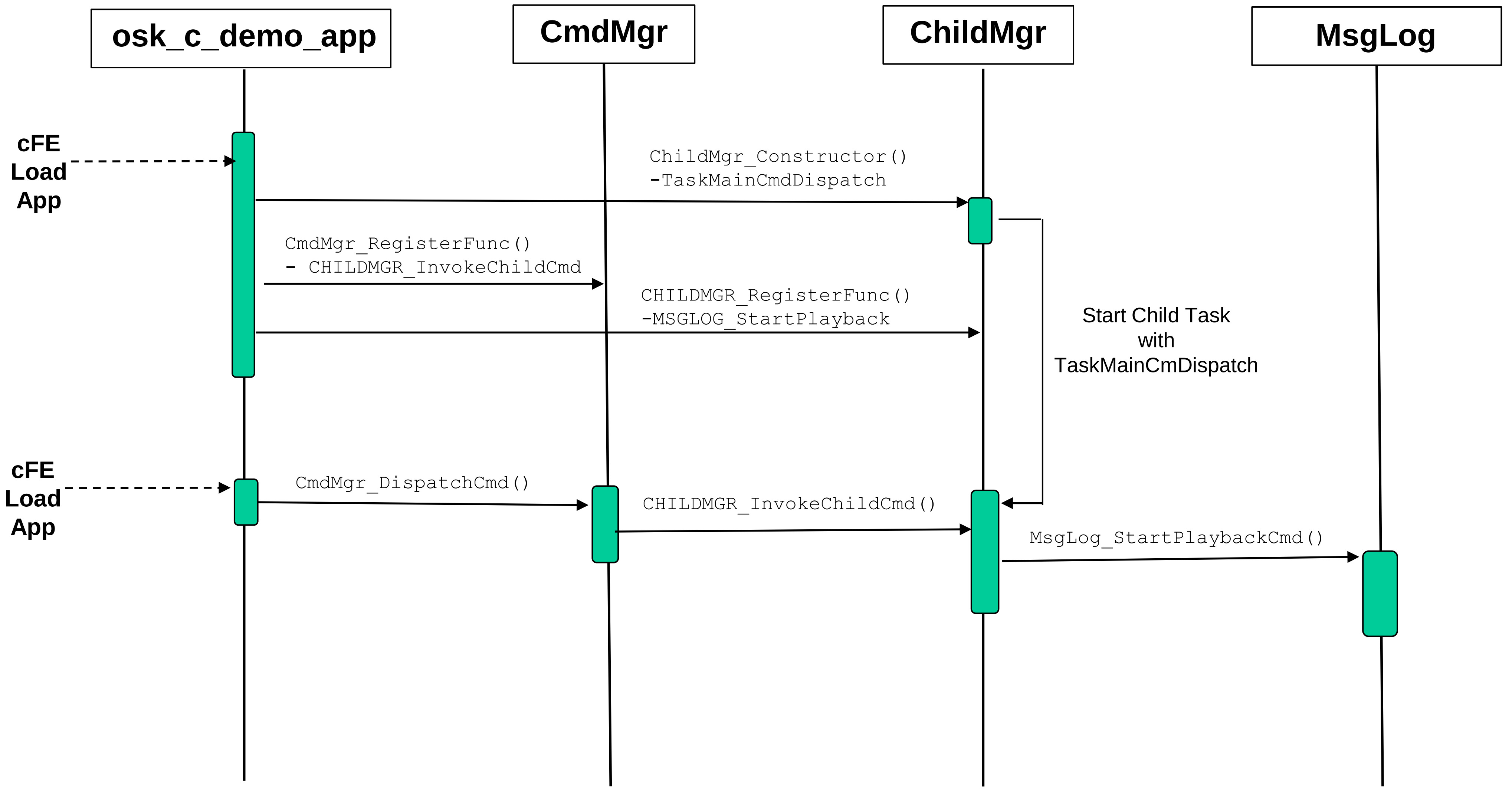
# Child Manager Functions



| **ChildMgr** |
| --- |
| *CmdQueue*<br>*Task Info*<br>*Cmd & Task Status* |
| *Constructor()*<br>*RegisterFuncl ()*<br>*ResetStatus()*<br>*InvokeChildCmd()*<br>*PauseTask()*<br>*TaskMainCallback()*<br>*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

# MsgLog Start Playback Sequence Diagram



- **TBD**
  - TBD
  - Add ChildMgr framework to app
  - Add child task init table parameters
  - Constructor child task
  - NASA app examples

# App

# Utilities

- **TBD – Coming Soon**

# Utilities Overview

- **osk_c_fw utilities are collections of functions that operate on the function parameters**
  - In OO parlance they are like class functions as opposed to instance functions
  - There is no object instance with state information
- **In v3.1 osk_c_fw contains two utilities: FileUtil (fileutil.h) and PktUtil (pktutil.h)**
  - cjson (the backend for table processing) could also be considered a utility, it has state information
- **The header files serve as the API**
- **FileUtil highlights**
  - Get file information to determine whether it exists, is a directory, and is closed/open
  - File verification functions for filenames, files for reading, and directories for writing
- **PktUtil highlights**
  - Packet filtering functions that were created from NASA's Data Storage app

Open STEM ware

# Application Design Patterns

# Introduction

- **TBD – This section will include application design patterns**
- **The current slides are a collection of notes**

# Main Loop Control for Community Apps

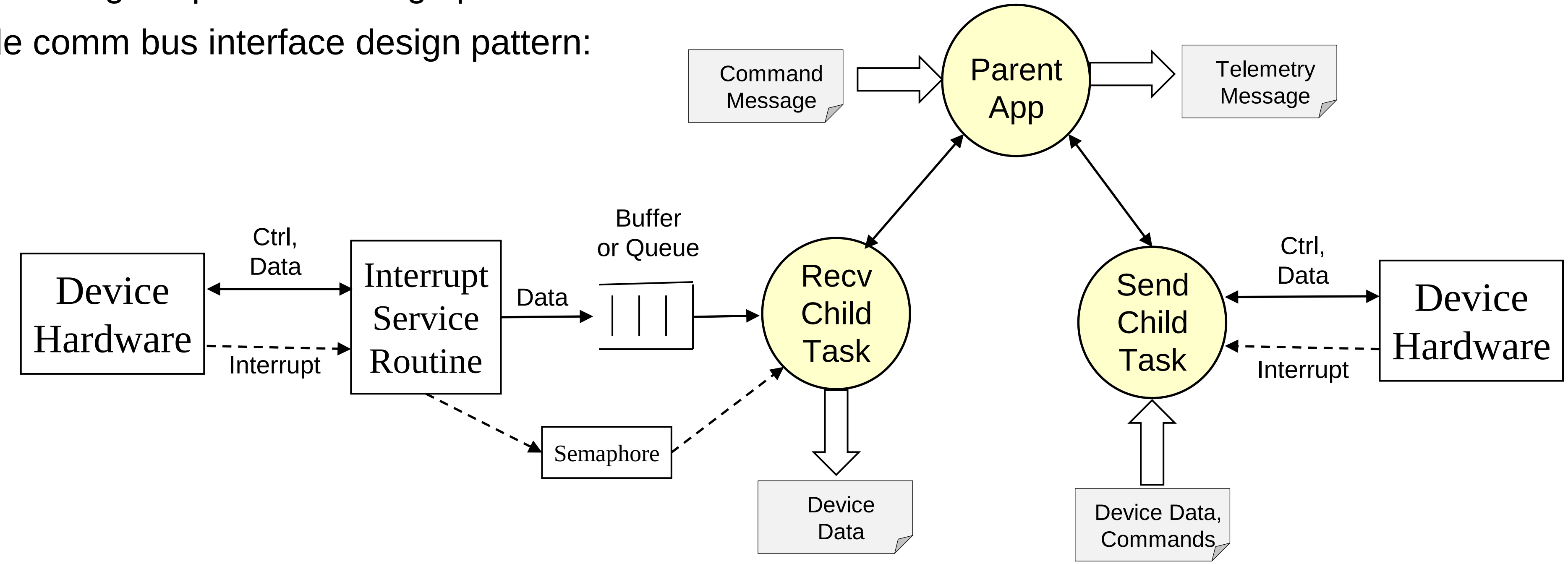
| Application | Main Loop Control | Control Notes |
|---|---|---|
| CF – CFDP | Pend Forever | Scheduler wakeup and HK request |
| CS – Checksum | Pend Forever | Scheduler wakeup and HK request |
| DS - Data Storage | Pend Forever | Subscribed message wakeup and HK request |
| F42 - 42 FSW Controller | 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) |

# Long Processing Child Tasks

# Hardware Device Abstracted to Messages



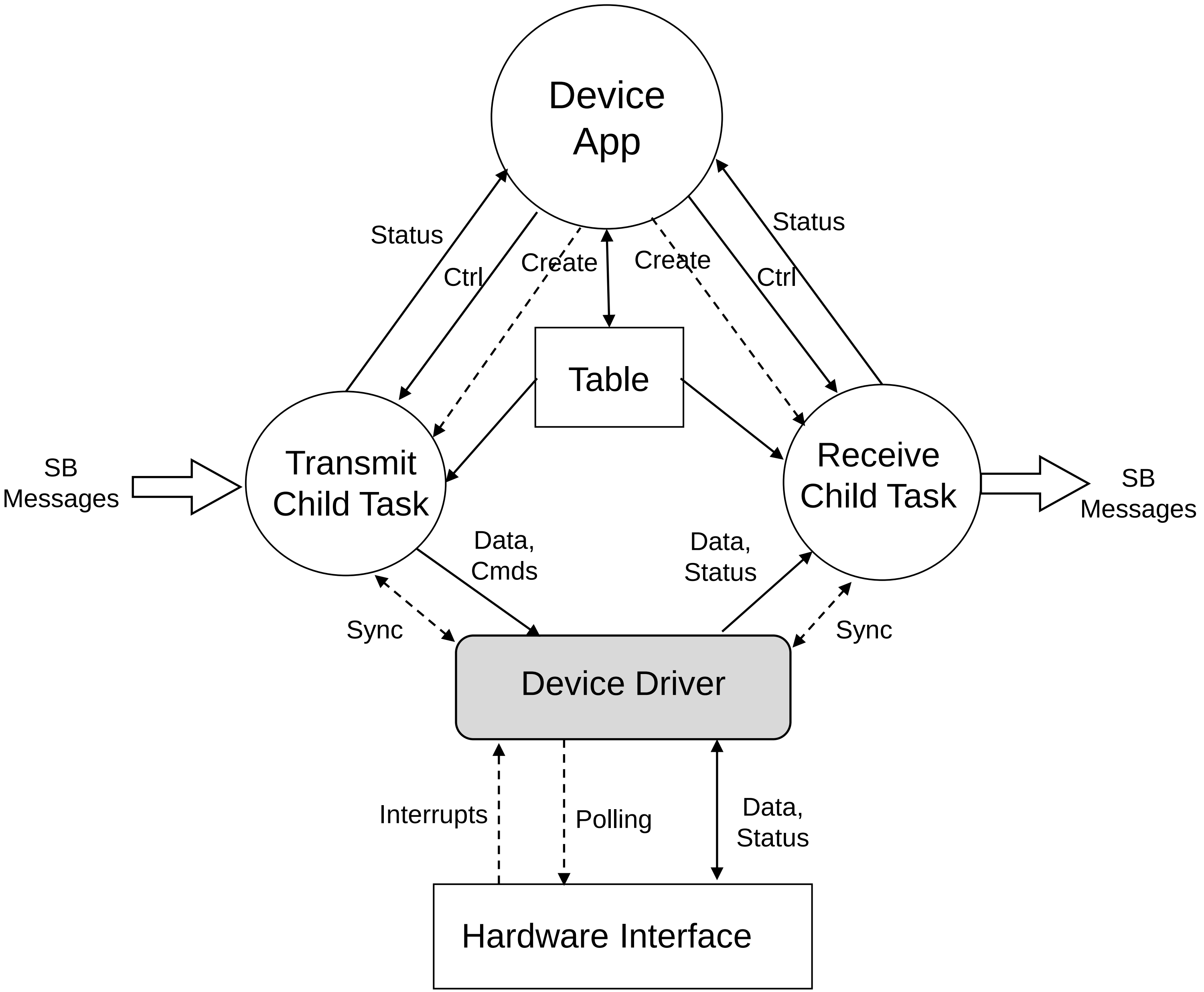
- **Device abstraction architectural role**
  - Read device data and publish on message bus
  - Receive messages and send to the device
- **Use a combination of software components to manage control/data**
  - Common design captured in design patterns
  - Example comm bus interface design pattern:




- **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**

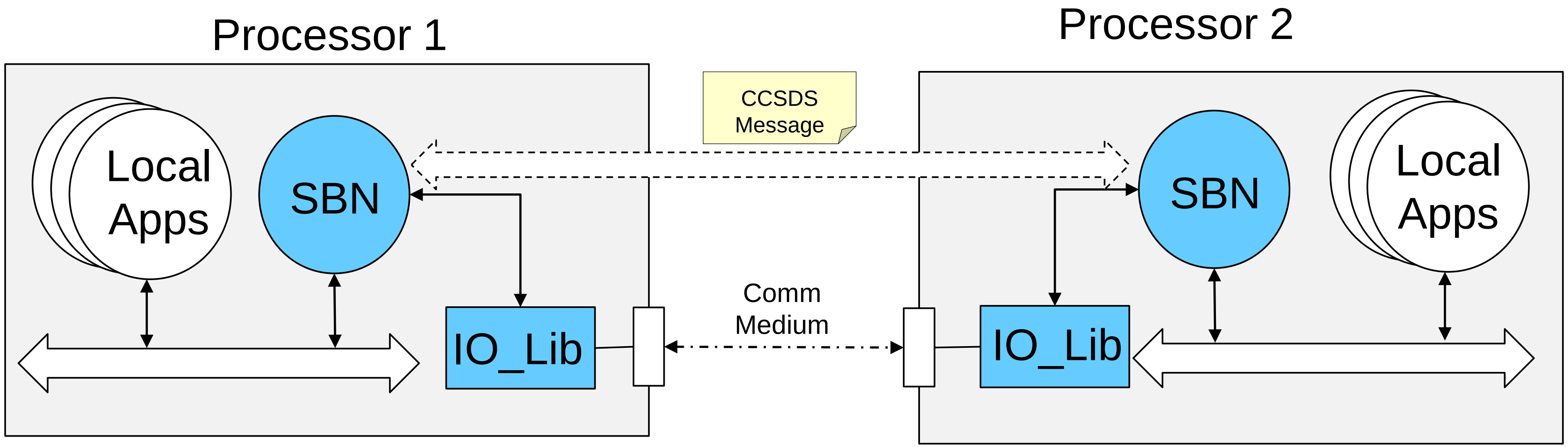
```
OS_MutSemTake(global_data.TableMutex);
...  global_data.TablePtr->...
OS_MutSemGive(global_data.TableMutex)
```

# 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**

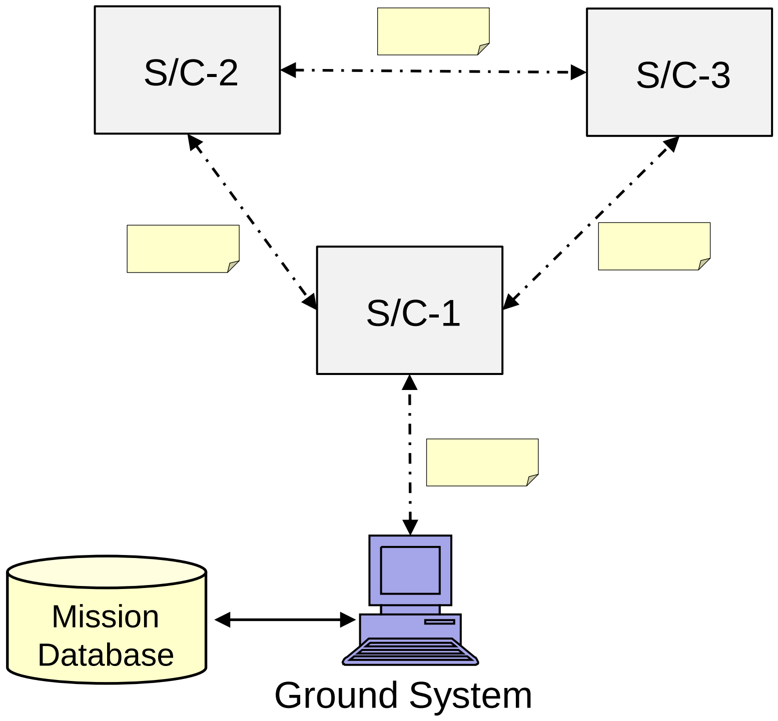
# Bridging cFS Instances

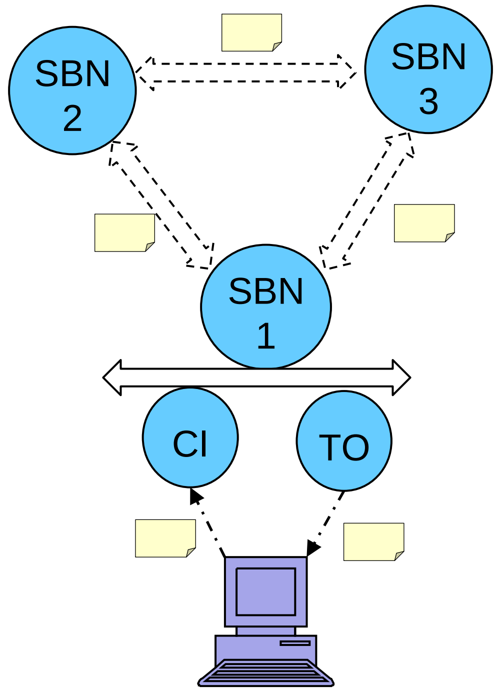





- **Software Bus Network (SBN, [https://github.com/nasa/SBN](https://github.com/nasa/SBN))**
  - Provides a bridge over Ethernet using UDP or TCP
- **The current SBN design does not include an IO Lib as shown**
  - Command Ingest (https://github.com/nasa/CFS_IO_LIB) and Telemetry Output (https://github.com/nasa/CFS_IO_LIB) 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**

Serialized CCSDS Packet　Comm Link　Bridged SB

# OSK_C_DEMO Design

# OSK_C_DEMO Introduction



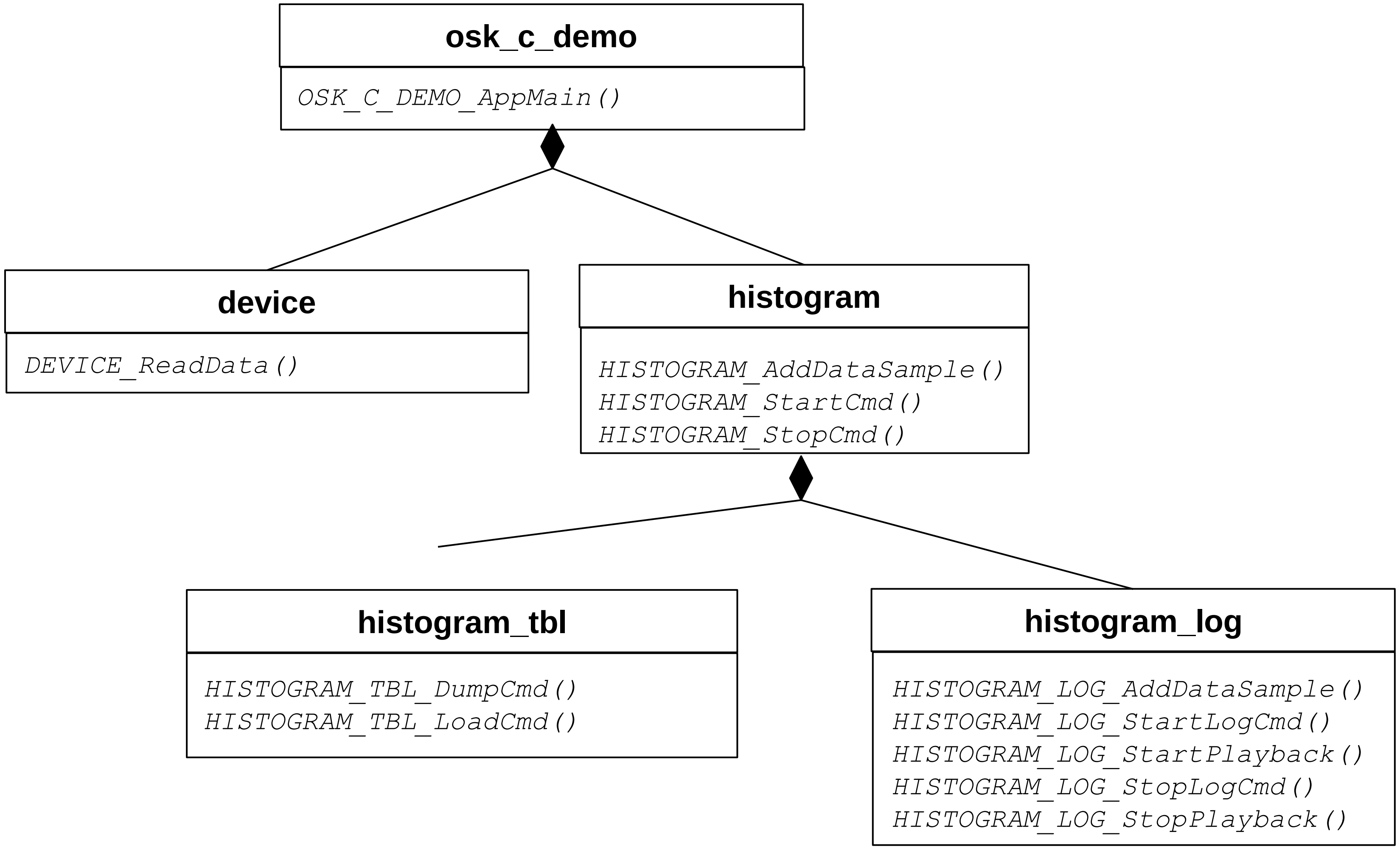
- **OSK_C_DEMO provides a non-trivial example app whose design is based on the OpenSatKit (OSK) C application framework OSK_C_FW.**
- **designed using demonstrates many of the OSK_C_FUpon command start logging the primary header of the command-specified message ID**
  - The header is written as hexadecimal text
  - Logging stops when a table-defined number of entries have been written or when the user issues a command to stop logging
- **Upon command playback in telemetry the contents of the message log file**
  - One header is contained in each playback telemetry message
  - A table-defined value specifies the delay between telemetry messages
  - The playback loops through the message log file until a stop playback or start new log command is received

# Design Goals

- Like a payload management app (popular custom mission app) without the need for simulation
- Complement command driven FM app design
- Utilize osk_c_fw child’s option not used by FileManager
- Different telemetry design then FM
- Options for demo status, break into diag
- Explain why want command counters & reset status so the FW provides value
- EDS versus fw object based design

# Demo App Design





# OSK_C_DEMO Table



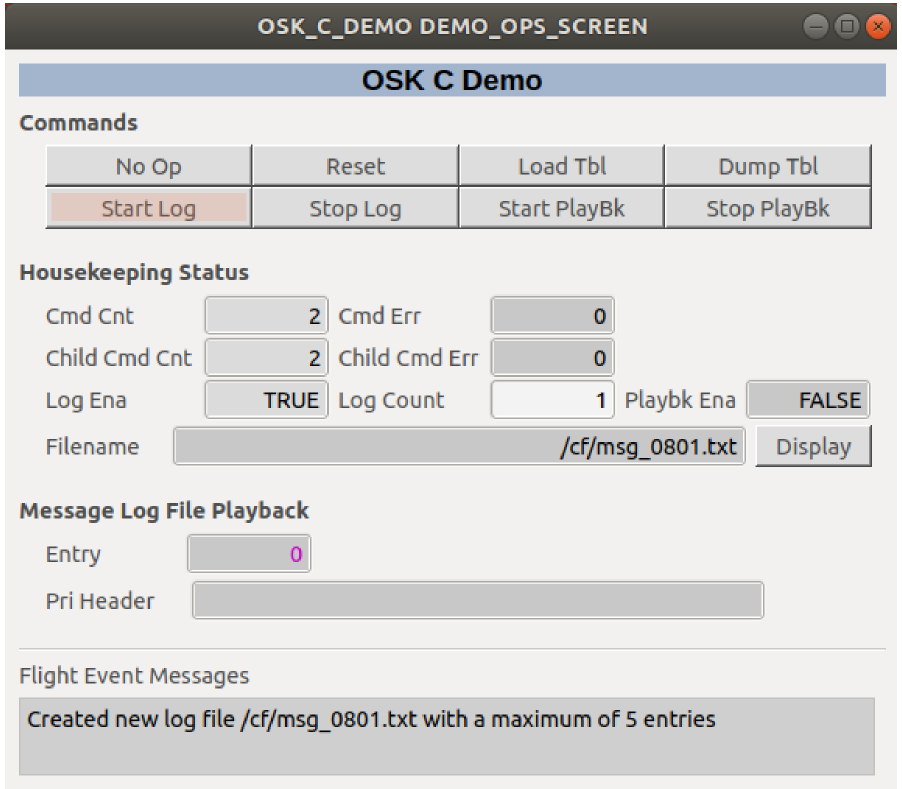
## osk_c_demo.json

```
{
   "app-name": "OSK_C_DEMO",
   "tbl-name": "Message Log",
   "description": "Define parameters for demo message logger",
   "file": {
      "path-base-name": "/cf/msg_",
      "extension":      ".txt",
      "entry-cnt":      5
   },
   "playbk-delay": 5
}
```

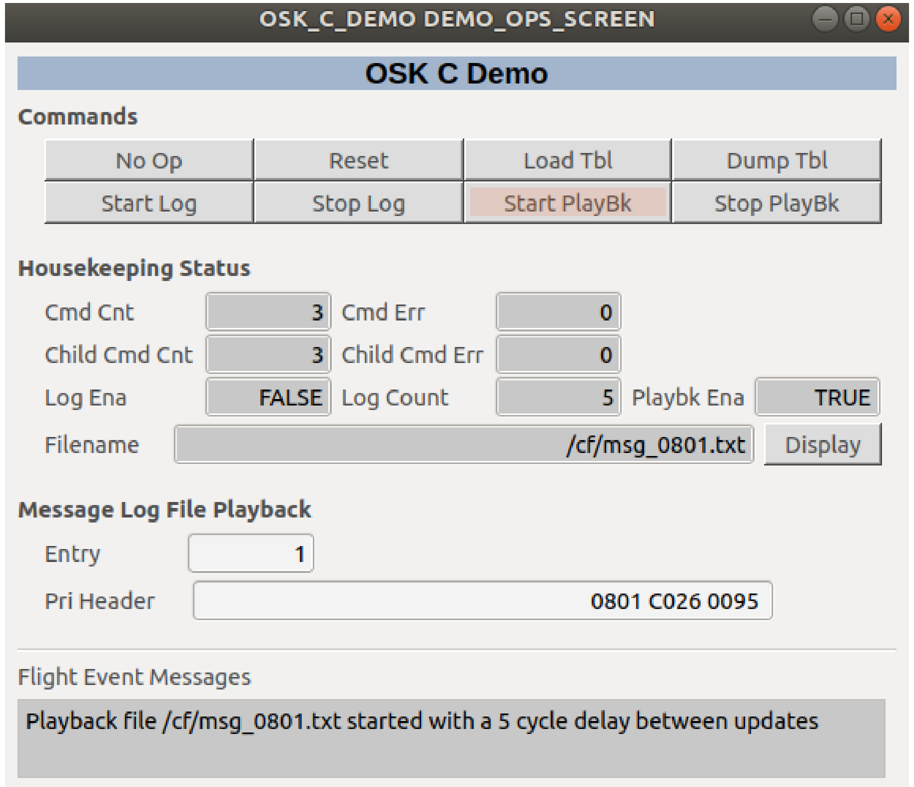
- **Message log file name created by concatenating *“path-base-filename”*, command-specified message ID, and *“extension”***
  - e.g. Sending the OSK_C_DEMO start log command ith a parameter of 0x0801 (cFE EVS housekeeping telemetry message)  results in a log filename of “msg_0801.txt”
- ***“entry-cnt”* defines maximum number of message log file entries**
- ***“playbk-delay”* defines number of OSK_C_DEMO execution cycles between playback telemetry messages**

# OSK_C_DEMO Ops Screen

## Message Log in Progress




## Log File Playback in Progress




- **cFE event service housekeeping message (ID = 0x0801) logged**
- **A child task performs logging and playback**
- ***"Display"* button transfers log file to ground and displays it in a text window**

# Refactoring NASA's File Manager App

# File Manager Refactor Overview

- **This section presents the results of refactoring NASA's File Manager (FM) app to use osk_c_fw**
- **Motivations for performing this exercise**
  - The initial effort started when OSK's cFE was updated and the latest NASA FM was not compatible with the latest cFE, so I performed local FM updates. As I performed the updates, I starting seeing how the app could benefit from the osk_c_fw that I had been using for OSK apps.
  - In general, I've been looking at all cFS community apps with an eye for how to make them more amenable to an app store concept. At the time of the refactor, FM had 32 compile-time configuration parameters! Configuration parameters add to an app's ease of adoption, so I wanted to assess what needs to be a configuration parameter and when does it need to be defined, compile-time or runtime?
  - Using an app like FM that has long successful history would help valid the osk_c_fw architecture if the refactoring is successful.
- **osk_c_fw may be too much of a 'baby step' for the app store concept**
  - This refactor keeps apps in the C programming language domain which may not be a big enough step forward
  - I hope it is still helpful to the community because it does show benefits of an object-based approach in C
- **Comments on the original NASA FM design are not intended to be critical, but instructional**
  - The NASA app design has a long history rooted in extremely constrained flight environments that evolved from procedural programming design practices
  - Refactoring a piece of software has the benefit of seeing the complete picture so patterns and optimizations can be discovered regardless of the technology being used

# File Manager Refactor Approach

- **This section does not document every aspects of the refactor**
  - Keep this section relatively short
  - The source code can be analyzed once the basic design structures are described
- **The original FM's design is shown with a brief description of how data and functionality was decomposed and allocated to different files**
- **The file copy and move commands are analyzed in detail to show how the original vs the refactored code implement the functions**
- **Some general observations are made with a summary of results**

# File Manager Design

## FM Global Data

App Cmd Counters
Child Cmd Counters
Cmd Queue Mgmt Data

Free Space Telemetry Packet

File Info Telemetry Packet

Open Files Telemetry Packet

Housekeeping Telemetry Packet

Child Cmd Queue

### fm_tbl

AcquirePtr()
ReleasePtr()
Validate()

### fm_app

Global Data

### fm_cmd

*Noop*
*ResetCounters*
*CopyFile*
*MoveFile*
*RenameFile*
*DeleteFile*
*DeleteAllFiles*
*DecompressFile*
*ConcatFiles*
*GetFileInfo*
*GetOenFiles*
*CreateDir*
*DeleteDir*
*GetDirListFile*
*GetDirListPkt*
*GetFreeSpace*
*SetTabletate*
*SetPermissions*

### fm_cmd_utils

*IsValidCmdPktLength*
*VerifyOverwirte*
*GetOpenFilesData*
*GetFilenameState*
*VerifyNameValid*
*VerifyFileClosed*
*VerifyFileExists*
*VerifyFileNoExist*
*VerifyFileNotOpen*
*VerifyDirExists*
*VerifyDirNoExist*
*VerifyChildTask*
*InvokeChildTask*
*AppendPathSep*

### fm_child

*Task, Process, Loop*
*CopyFile*
*MoveFile*
*RenameFile*
*DeleteFile*
*DeleteAllFiles*
*DecompressFile*
*ConcatFile*
*FileInfo*
*CreateDir*
*DeleteDir*
*DirListFile*
*DirListPkt*
*SetPermissions*
*DirListFile*
*SizeTimeMode*
*SleepStat*

### fm_msg.h

*Cmd Structures*
*Tlm Structures*
*File Records*

### fm_defs.h

*Status*

### fm_events.h

*Event Msg IDs*

### fm_msgdefs.h

*Cmd Function Codes*

- **The original cFS application designs are procedural**
- **Functions and data defined separate files and dictate program structure**

File
- Copy
- Move
- Rename
- Delete
- Delete Internal
- Delete All
- Decompress
- Concatenate
- File Info
- List open files
- Set permissions

Directory
- Create
- Remove
- Delete
- Send Listing
- Write Listing

Freespace Table
- Get Free Space
- Set Entry state

# File Copy



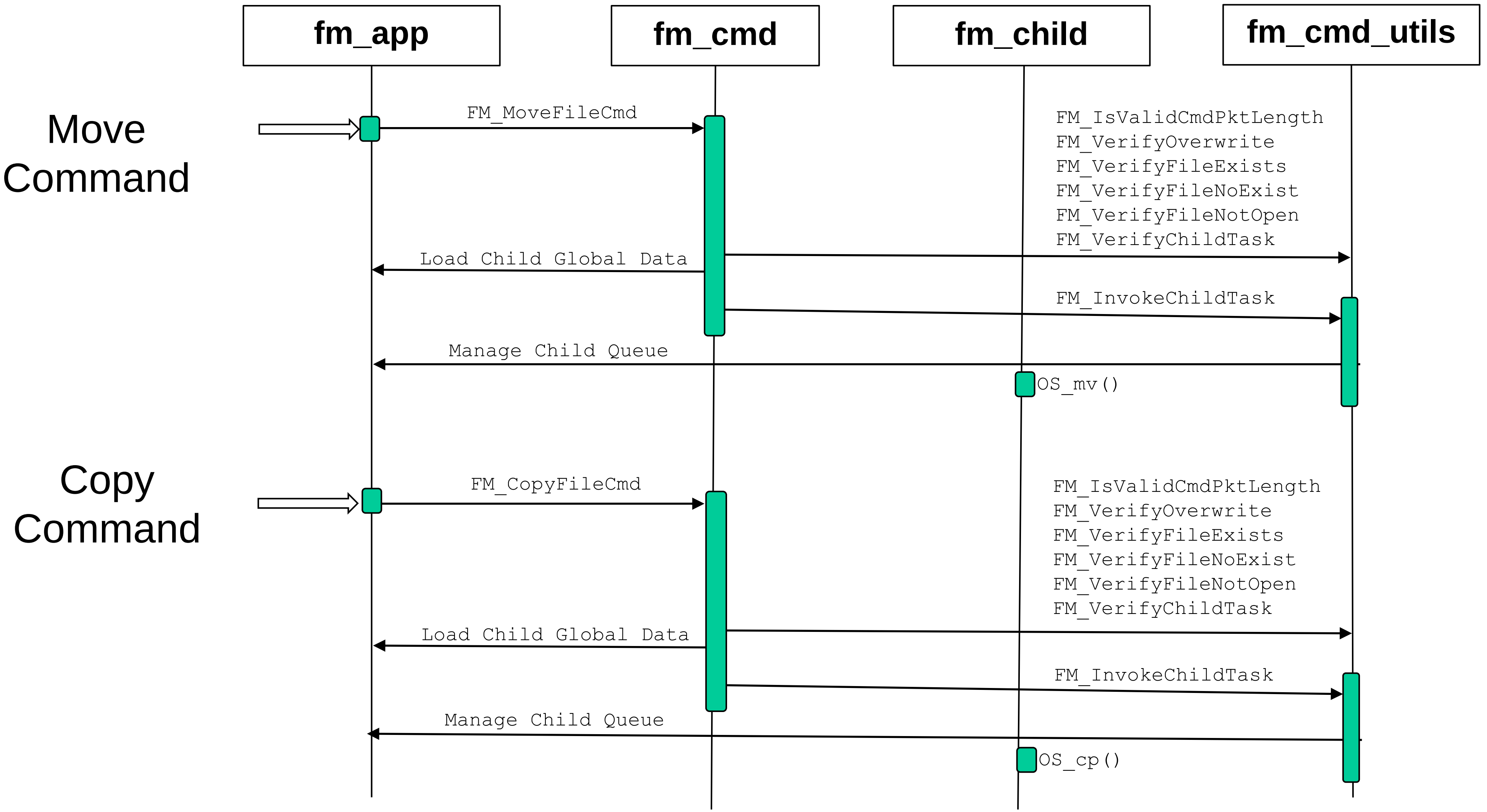
```
boolean FM_CopyFileCmd(CFE_SB_MsgPtr_t MessagePtr){
   FM_CopyFileCmd_t *CmdPtr = (FM_CopyFileCmd_t *) MessagePtr;
   FM_ChildQueueEntry_t *CmdArgs;
   char *CmdText = "Copy File";
   boolean CommandResult;

   /* Verify command packet length */
   CommandResult = FM_IsValidCmdPktLength(MessagePtr, sizeof(FM_CopyFileCmd_t), FM_COPY_PKT_ERR_EID, CmdText);
   /* Verify that overwrite argument is valid */
   if (CommandResult == TRUE) {
      CommandResult = FM_VerifyOverwrite(CmdPtr->Overwrite, FM_COPY_OVR_ERR_EID, CmdText);
   }
   /* Verify that source file exists and is not a directory */
   if (CommandResult == TRUE)  {
      CommandResult = FM_VerifyFileExists(CmdPtr->Source, sizeof(CmdPtr->Source), FM_COPY_SRC_ERR_EID, CmdText);
   }
   /* Verify target filename per the overwrite argument */
   if (CommandResult == TRUE) {
      if (CmdPtr->Overwrite == 0) {
          CommandResult = FM_VerifyFileNoExist(CmdPtr->Target, sizeof(CmdPtr->Target), FM_COPY_TGT_ERR_EID, CmdText);
      }
      else {
          CommandResult = FM_VerifyFileNotOpen(CmdPtr->Target, sizeof(CmdPtr->Target),FM_COPY_TGT_ERR_EID, CmdText);
        }
   }
   /* Check for lower priority child task availability */
   if (CommandResult == TRUE) {
      CommandResult = FM_VerifyChildTask(FM_COPY_CHILD_ERR_EID, CmdText);
   }
   /* Prepare command for child task execution */
   if (CommandResult == TRUE) {
      CmdArgs = &FM_GlobalData.ChildQueue[FM_GlobalData.ChildWriteIndex];
      /* Set handshake queue command args */
      CmdArgs->CommandCode = FM_COPY_CC;
      strcpy(CmdArgs->Source1, CmdPtr->Source);
      strcpy(CmdArgs->Target,  CmdPtr->Target);
      /* Invoke lower priority child task */
      FM_InvokeChildTask();
```

# File Move

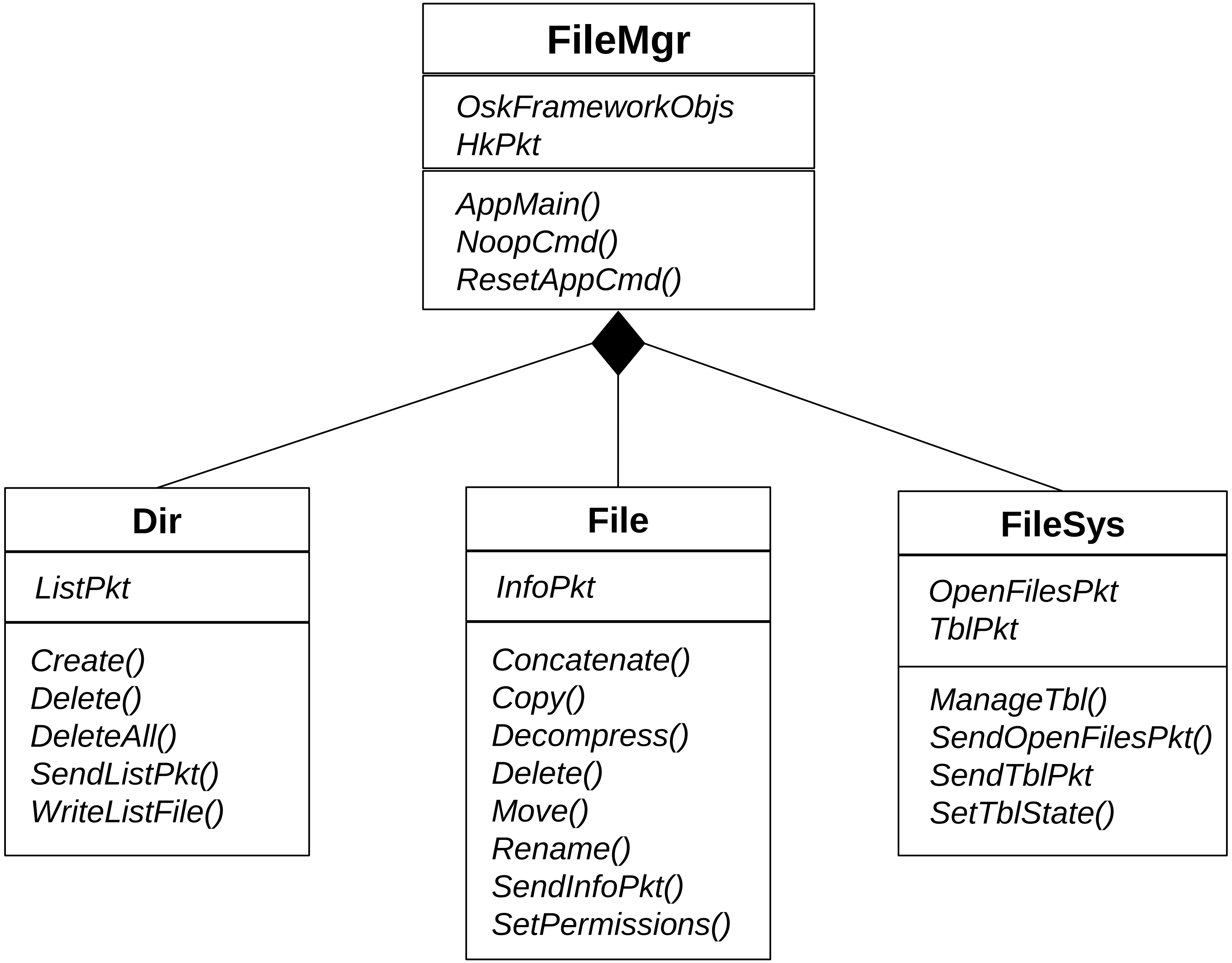


```
boolean FM_MoveFileCmd(CFE_SB_MsgPtr_t MessagePtr){    FM_MoveFileCmd_t  *CmdPtr = (FM_MoveFileCmd_t *) MessagePtr;
FM_ChildQueueEntry_t *CmdArgs;    char *CmdText = "Move File";    boolean CommandResult;    /* Verify command packet
length */    CommandResult = FM_IsValidCmdPktLength(MessagePtr, sizeof(FM_MoveFileCmd_t),
FM_MOVE_PKT_ERR_EID, CmdText);    /* Verify that overwrite argument is valid */    if (CommandResult == TRUE)    {
CommandResult = FM_VerifyOverwrite(CmdPtr->Overwrite,                                            FM_MOVE_OVR_ERR_EID,
CmdText);    }    /* Verify that source file exists and not a directory */    if (CommandResult == TRUE)    {
CommandResult = FM_VerifyFileExists(CmdPtr->Source, sizeof(CmdPtr->Source),
FM_MOVE_SRC_ERR_EID, CmdText);    }    /* Verify target filename per the overwrite argument */    if (CommandResult ==
TRUE)    {        if (CmdPtr->Overwrite == 0)        {            CommandResult = FM_VerifyFileNoExist(CmdPtr->Target,
sizeof(CmdPtr->Target),                                                     FM_MOVE_TGT_ERR_EID, CmdText);        }
else        {            CommandResult = FM_VerifyFileNotOpen(CmdPtr->Target, sizeof(CmdPtr->Target),
FM_MOVE_TGT_ERR_EID, CmdText);        }    }    /* Check for lower priority child task availability */    if
(CommandResult == TRUE)    {        CommandResult = FM_VerifyChildTask(FM_MOVE_CHILD_ERR_EID, CmdText);    }    /* Prepare
command for child task execution */    if (CommandResult == TRUE)    {        CmdArgs =
&FM_GlobalData.ChildQueue[FM_GlobalData.ChildWriteIndex];        /* Set handshake queue command args */        CmdArgs-
>CommandCode = FM_MOVE_CC;        strcpy(CmdArgs->Source1, CmdPtr->Source);        strcpy(CmdArgs->Target,  CmdPtr-
>Target);        /* Invoke lower priority child task */        FM_InvokeChildTask();    }    return(CommandResult);} /*
End of FM_MoveFileCmd() */
```

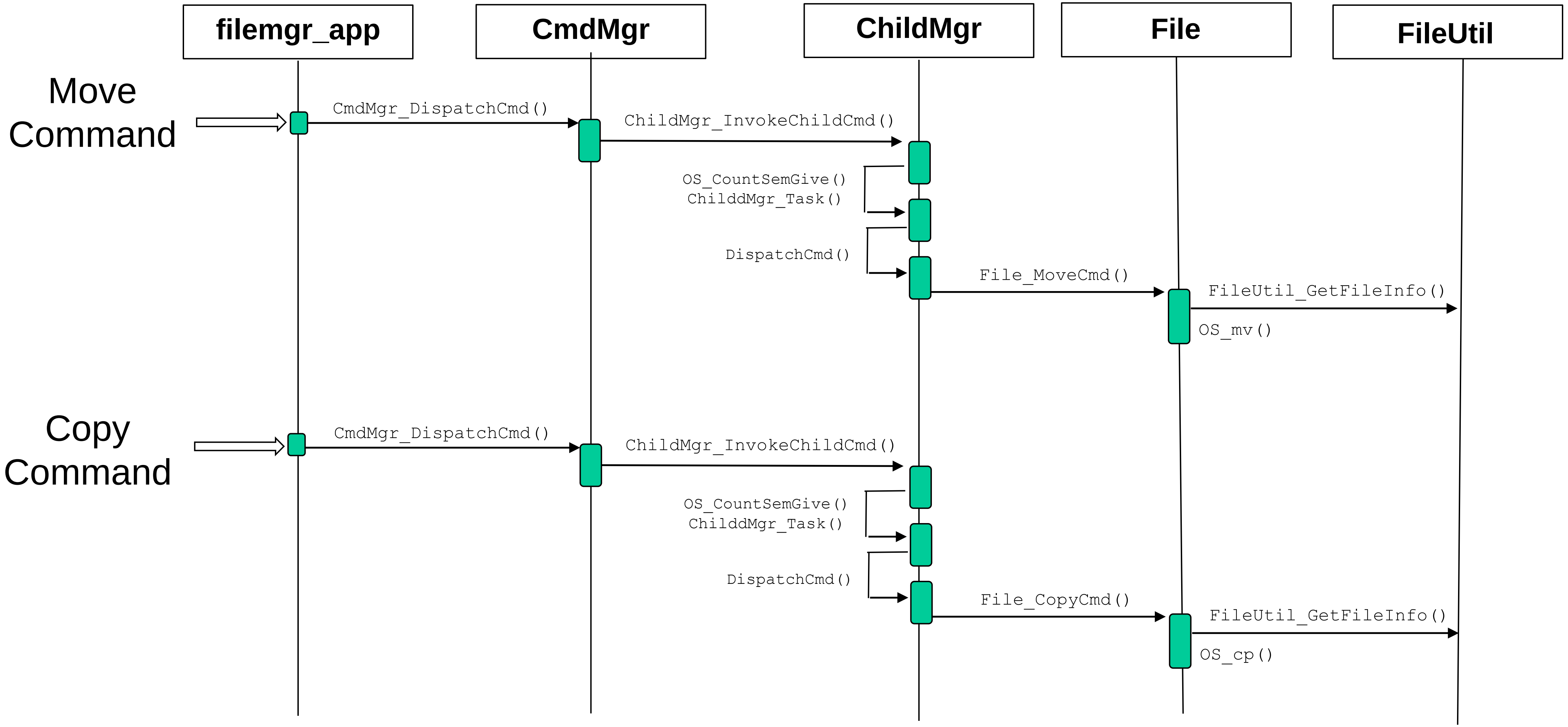
# Original File Manager Sequence Diagram



# Refactored File Manager Design



# Refactored File Manager Sequence Diagram



# Refactor Observations and Results

- **TBD**

- **Objects suitable for multiple apps migrate to the framework or to a shared library**
  - FileUtils
- **OO 'smells'**

| App | C Src Files | LOC | Platform Compile- Cfg | Init Runtime-Cfg | Notes |
|---|---|---|---|---|---|
| FM | 20 | 3038 | 32 | n/a | |
| FileMgr | 12 | 1681 | 6 | 25 | App name and table name repeated because binary table requires them during compilation |

- **Telemetry Design**
  - HK vs
- **Ops versus design nomenclature**
  - Design command names vs EDS operational names

# Testing

# Introduction

- **TBD – This section will cover unit, functional, and continuous integration**

# Appendix A
# Design Notation

# 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