



OpenSatKit Application Developer's Guide

OSK_C_FW 4.0

2/6/22

- **Objectives**

- Describe the OSK Application Framework design and how to develop apps with the framework
- Intended to augment the cFS Application Developer's Guide

- **Intended Audience**

- Software developers that want to develop cFS applications

- **Prerequisites**

- Basic understanding of the cFS architecture and the cFS Application Developer's Guide
- Working knowledge of the C programming language

- **Framework 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

- **Objectives**

- Describe the OSK Application Framework design and how to develop apps with the framework
- Intended to augment the cFS Application Developer's Guide

- **Intended Audience**

- Software developers that want to develop cFS applications

- **Prerequisites**

- Basic understanding of the cFS architecture and the cFS Application Developer's Guide
- Working knowledge of the C programming language

- **Framework 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

- **Configuration Management**

- This document is maintained in the OpenSatKit-Apps `osk_c_fw` repository

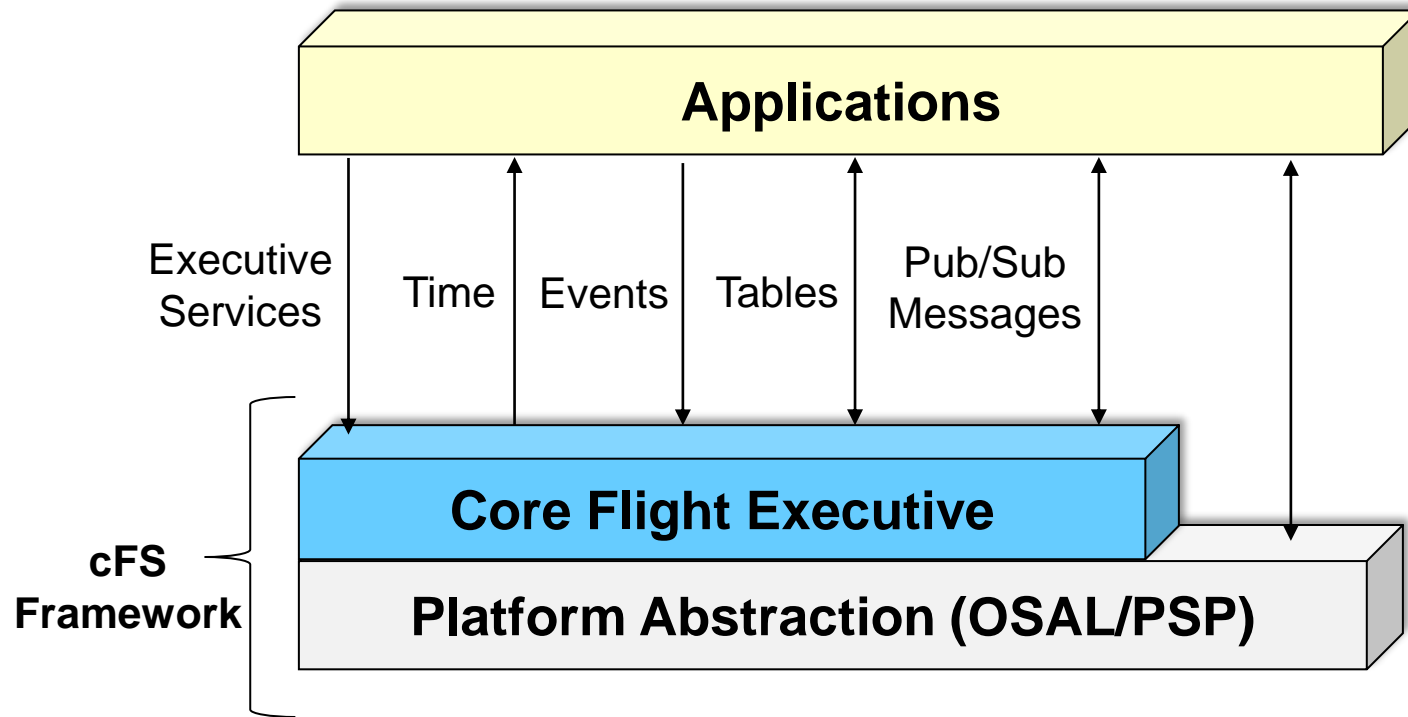
1. cFS Application Context
2. Application Framework Architecture
3. Building and Running Applications
4. Testing Considerations
5. Design Patterns
6. OSK_C_DEMO App Design
7. Refactoring NASA's File Manager App

This is a work in progress and not all sections are complete



cFS Application Context

- **Prior to describing the OSK Application Framework it is important to understand the context of applications that will be developed using the framework.**
- **A cFS application context is bounded by the following interfaces**
 1. cFE, OSAL, and PSP services and Application Programmer Interfaces (APIs)
 2. The message interface created by the 'standard' runtime environment application suite
 3. Hardware interfaces
- **This section discusses the first two interfaces**
- **Hardware interfaces are covered in the Design Pattern section**
 - The OSK application framework does not directly support hardware interfaces
 - Application design patterns are used to show how custom interface apps can be designed to work with other mission specific-apps to provide mission-specific functionality



Executive Services (ES)

- Manage the software system and create an application runtime environment

Time Services (TIME)

- Manage spacecraft time

Event Services (EVS)

- Provide a service for sending, filtering, and logging event messages

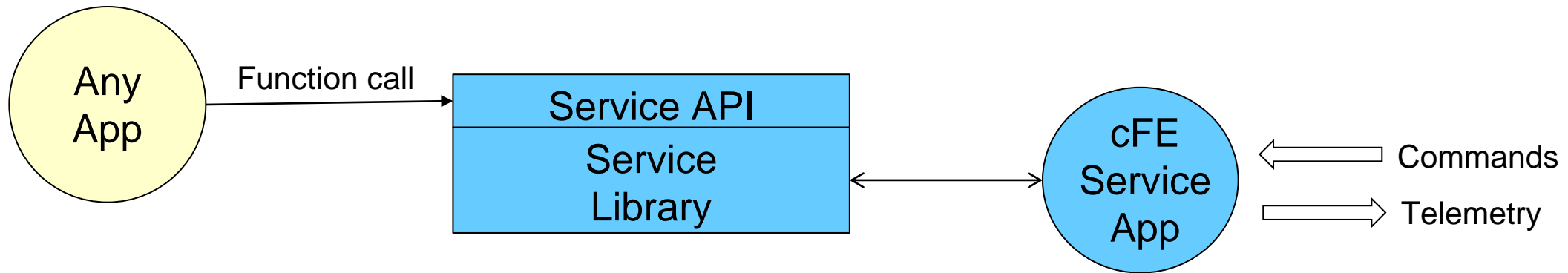
Software Bus (SB) Services

- Provide an application publish/subscribe messaging service

Table Services (TBL)

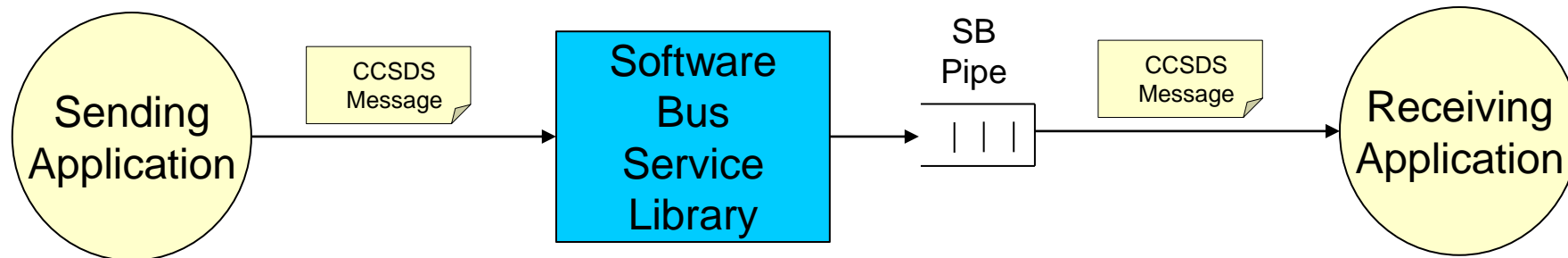
- Manage application table images

- **Applications are an architectural component that owns cFE and operating system resources via the cFE and OSAL Application Programmer Interfaces (APIs)**
- **cFE Services provide an Application Runtime Environment**
- **Resources are acquired during initialization and released when an application terminates**
 - Helps achieve the architectural goal for a loosely coupled system that is scalable, interoperable, testable (each app is unit tested), and maintainable
- **Concurrent execution model**
 - Each app has its own execution thread and apps can spawn child tasks
- **The cFE service and Platform Abstraction APIs provide a portable functional interface**
- **Write once run anywhere the cFS framework has been deployed**
 - Defer embedded software complexities due to cross compilation and target operating systems
 - Smartphone apps need to be rewritten for each platform
 - Provides seamless application transition from technology efforts to flight projects
- **Reload apps during operations without rebooting**



- **Each cFE service has**
 - A library that is used by applications
 - An application that provides a ground interface for operators to use to manage the service
- **Each cFE Service App periodically sends status telemetry in a “Housekeeping (HK) Packet”**
 - Obtaining additional information beyond HK with commands that
 - Send one-time telemetry packets
 - Write onboard service configuration data to files

 = Software Bus Message



- **Applications create *SB Pipe* (a *FIFO queue*) and subscribe to receive messages**
 - Typically performed during application initialization
- **If needed, apps can subscribe and unsubscribe to messages at any time for runtime reconfiguration**
- ***SB Pipes* used for application data and control flow**
 - Poll and pend for messages

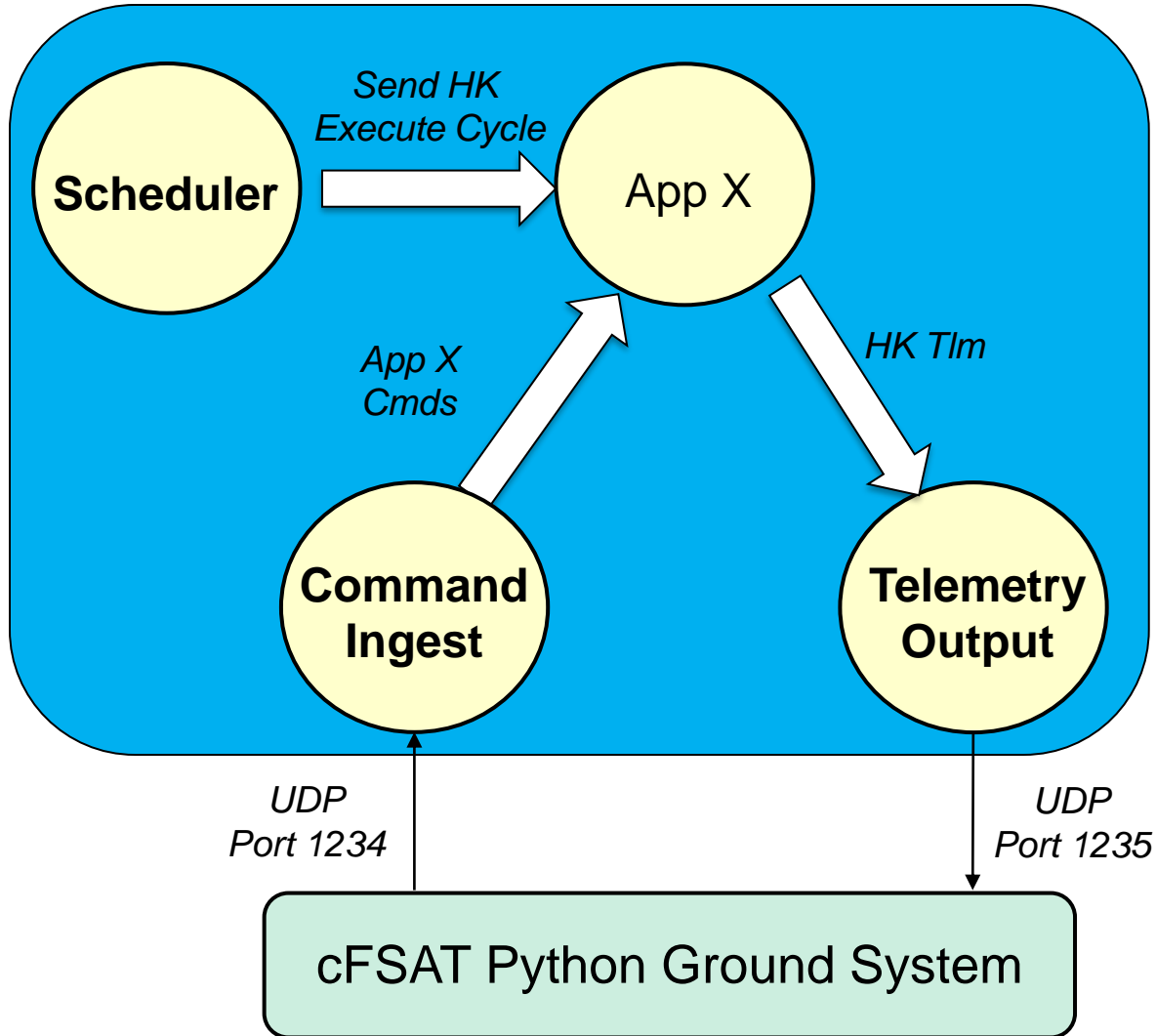
- **What is a library?**
 - A collection of utilities available for use by any app
 - Exist at the cFS application layer
- **Libraries are not registered with Executive Services and do not have a thread of execution so limited cFE API usage. For example,**
 - A library can't call CFE_EVS_Register() during initialization
 - The ES API does not provide a function for libraries analogous to CFE_ES_GetAppInfo()
- **Library functions execute within the context of the calling application**
 - CFE_EVS_SendEvent() will identify the calling app
 - Libraries can't register for cFE services during initialization and in general should not attempt to do so
- **No cFE API exists to retrieve library code segment addresses**
 - Prevents apps like Checksum from accessing library code space.
- **Libraries and be statically dynamically linked**
 - Dynamic linking requires support from the underlying operating system
- **Specified in the cfe-es-startup.scr and loaded during cFE initialization**



- A common app suite that is typically present in a cFS distribution create a runtime environment that can be assumed by an application
- The cFE does not dictate this model but a minimal set of apps is required to make a cFS target** usable
 - A target needs to communicate (receive commands and send telemetry) with external systems
 - The cFS uses a message-based scheduler app to support the design of a synchronous system
- **Example runtime app suites**
 - OSK includes KIT_CI, KIT_TO, and KIT_SCH
 - NASA's cFS Bundle, <https://github.com/nasa/cFS>, includes 'lab' versions of these apps
- **The idea of app suites to provide functionality is common in the cFS**
 - OSK's *Mission FSW* provides a SimSat reference mission that describes in detail how groups of apps can collaborate to provide end-user functionality

** A cFS target is an instantiation of the cFE Framework on a platform with a set of library and apps. Not to be confused with a distribution. OSK is a distribution that contains multiple targets.

cFS Framework



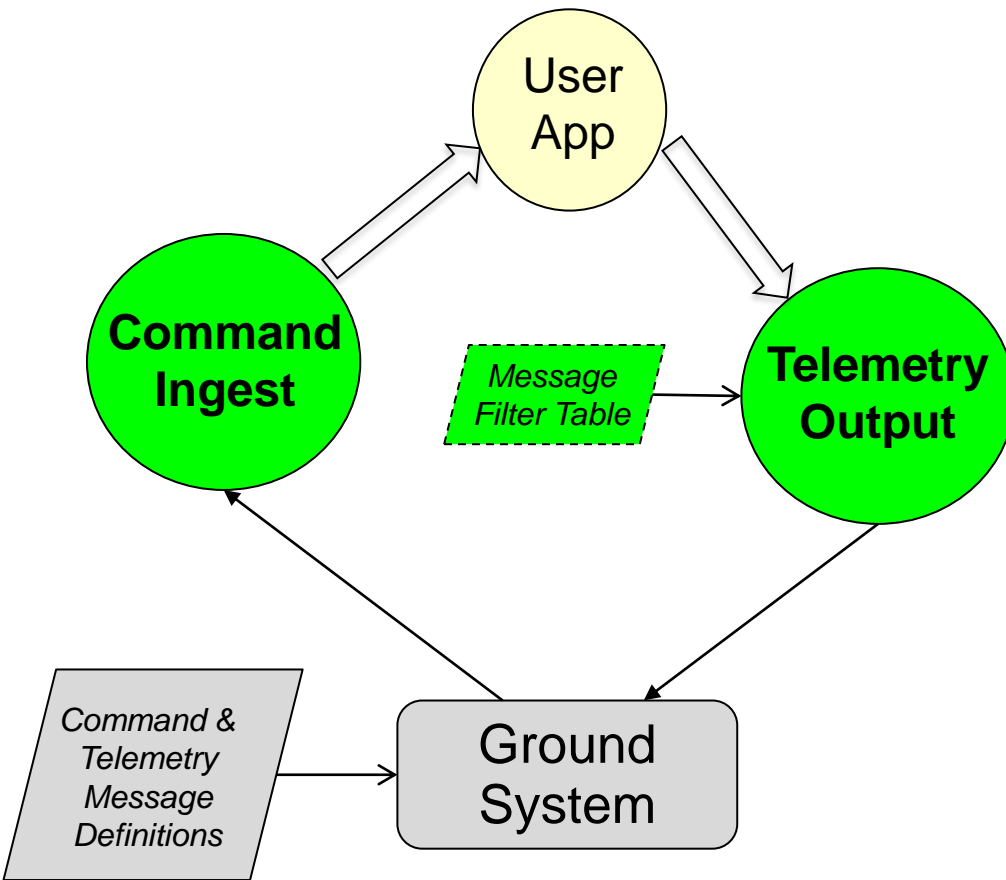
A core set of apps are required to provide a runtime environment

- Different app implementations can provide customized solutions for different platforms
- File management & transfer not shown

Scheduler (SCH) sends messages at fixed time intervals to signal apps to perform a particular function

Command Ingest (CI) receives commands from an external source and publishes them on the SB

Telemetry Out (TO) receives messages from the SB and sends them to an external destination



- **Command Ingest (CI) App**

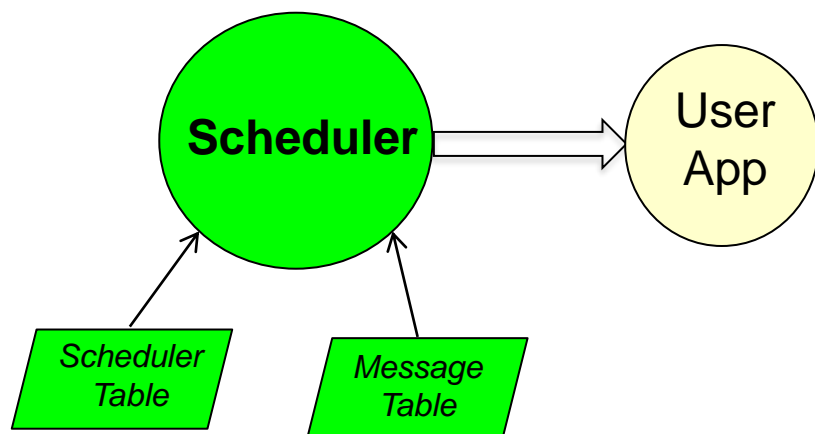
- Receives commands from an external source, typically the ground system, and sends them on the software bus

- **Telemetry Output (TO) App**

- Receives telemetry packets from a the software bus and sends them to an external source, typically the ground system
- Optional *Filter Table* that provides parameters to algorithms that select which messages should be output on the external communications link

- **Different versions of CI and TO used on different platforms**

- cFE delivered with 'lab' versions that use UDP for the external comm
- JSC released versions that use a configurable I/O library for a different external comm links
- OSK versions use UDP and a JSON filter table
- ITAR-restricted flight versions typically used inflight



- **Scheduler (SCH) App**

- Synchronizes execution with clock's 1Hz signal
- Sends software bus messages defined in the *Message Table* at time intervals defined in the *Scheduler Table*

- **Application Control Flow Options**

- Pend indefinitely on a *SB Pipe* with subscriptions to messages from the Scheduler
 - This is a common way to synchronize the execution of most of the apps on a single processor
 - Many apps send periodic "Housekeeping" status packets in response to a "Housekeeping Request" message from Scheduler
- Pend indefinitely on a message from another app
 - Often used when an application is part of a data processing pipeline
- Pend with a timeout
 - Used in situation with loose timing requirements and system synchronization is not required
 - The SB timeout mechanism uses the local oscillator so the wakeup time may drift relative to the 1Hz

Application Run Loop Messaging Example

Suspend execution until a message arrives on app's pipe

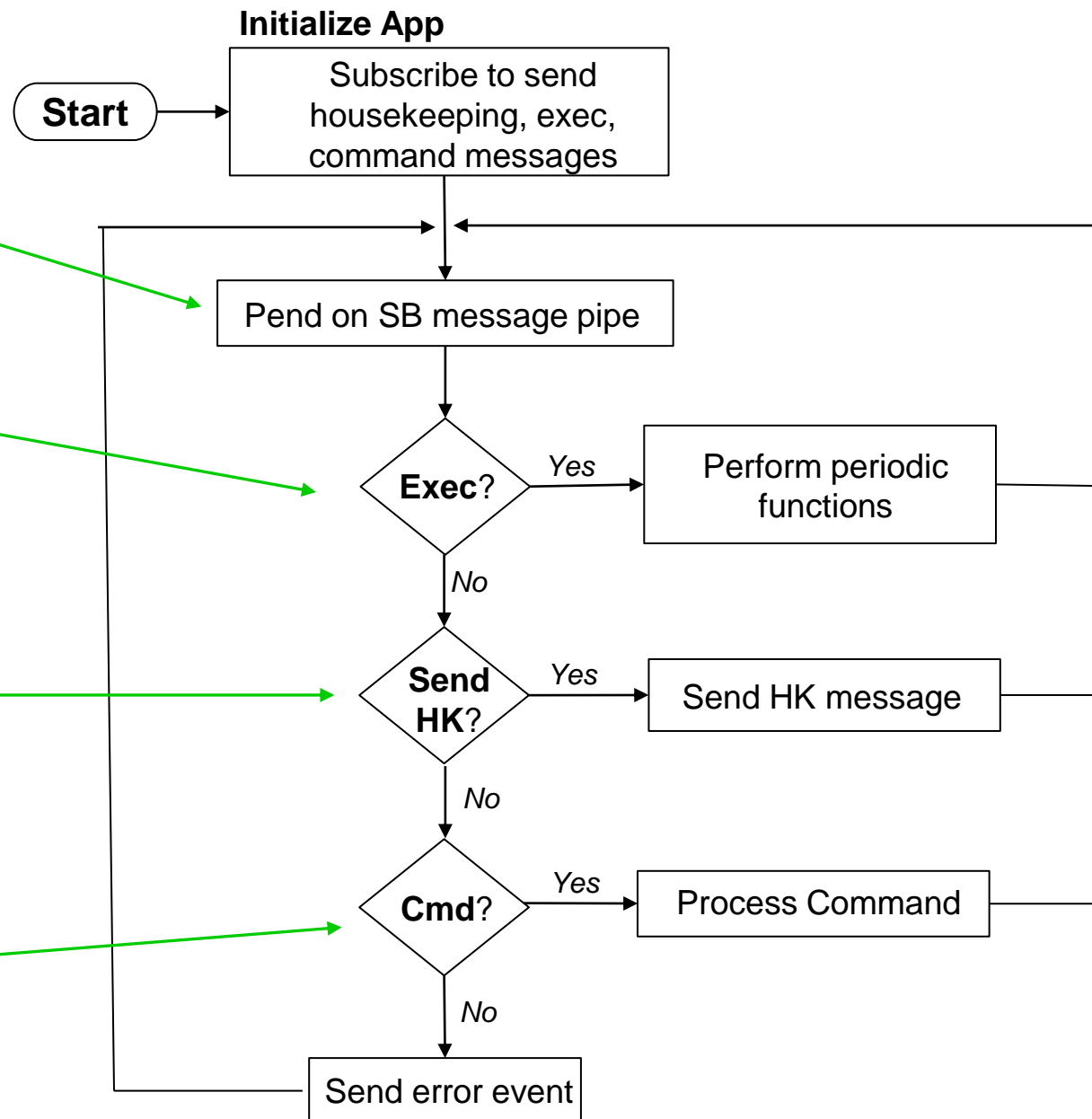
Periodic *execute* message from SCH app

Periodic *send housekeeping* message from SCH app

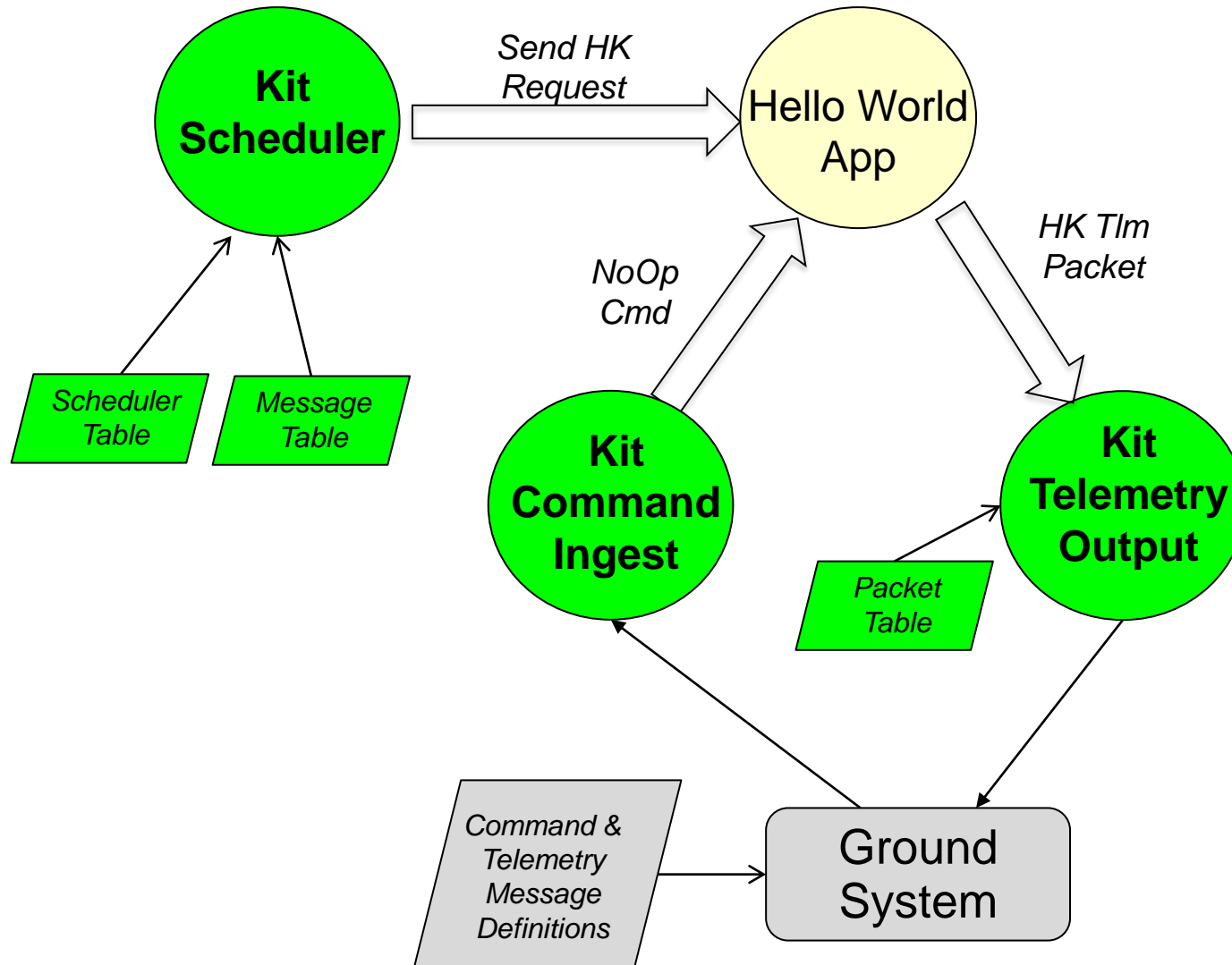
- Typically, on the order of seconds
- "Housekeeping cycle" convenient time to perform non-critical functions

Process commands

- Commands can originate from ground or other onboard apps



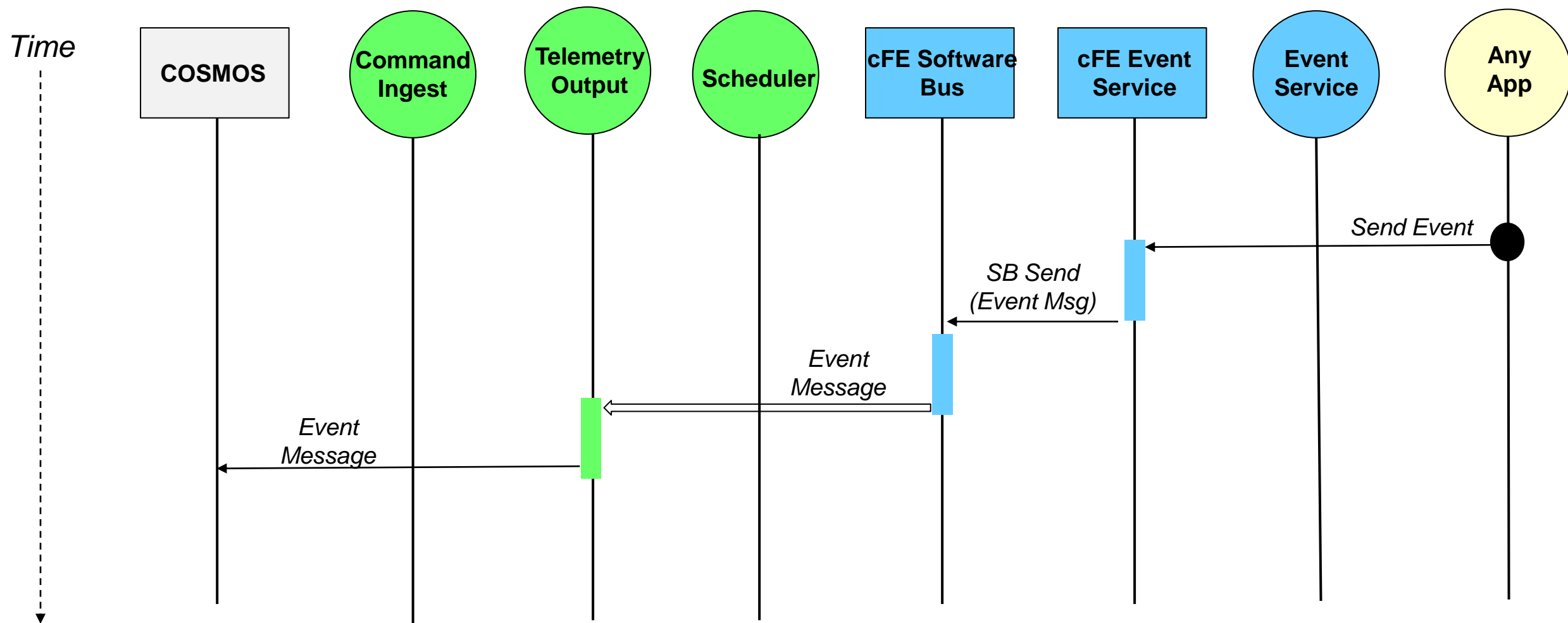
Hello World App Runtime Environment



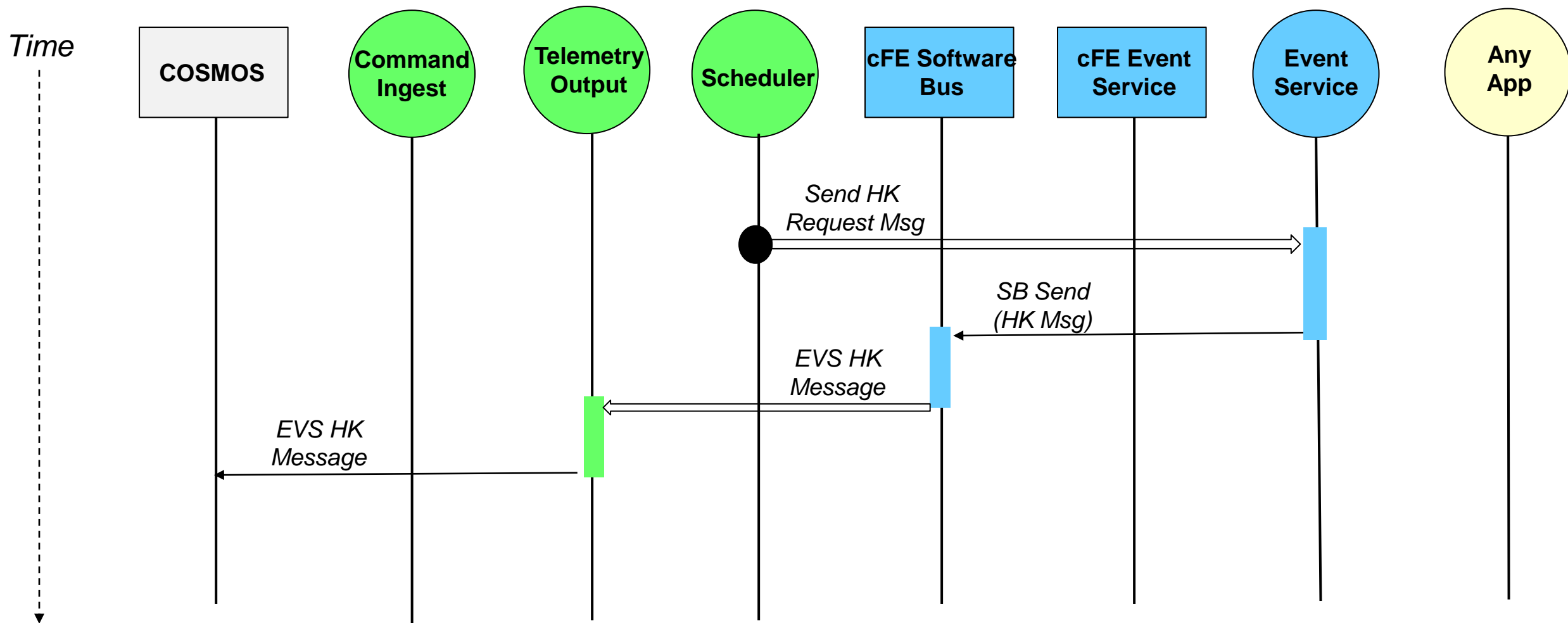
Context of "Hello World" app created in the next section

- Every 3 seconds Scheduler sends a "Send Housekeeping Telemetry Request"
 - HK telemetry includes valid and invalid command counters
- When user sends a "No Operation" command from the ground system Hello World responds with
 - An event message that contains the app's version number
 - Increments the command valid counter

App Send Event Sequence Diagram

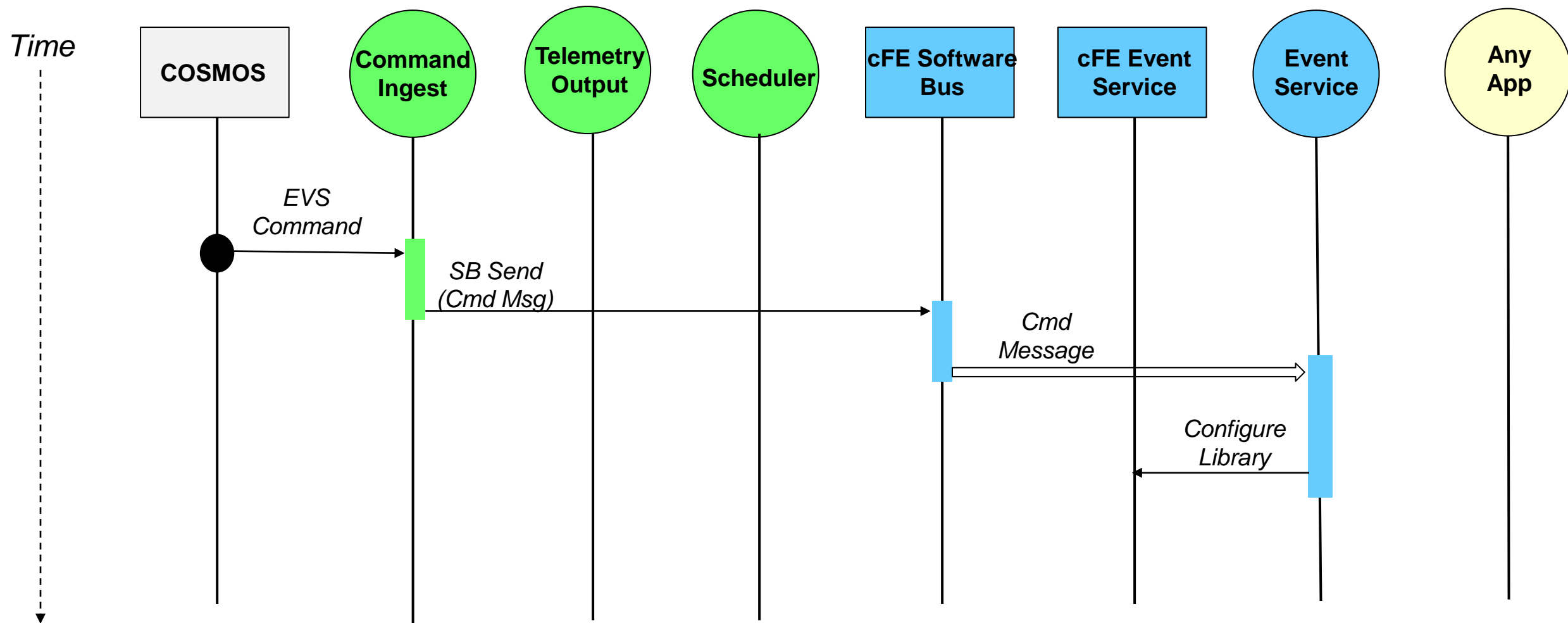


Event Service App Sends Housekeeping Telemetry



● = Initial event

Ops Sends EVS Configuration Command



● = Initial event

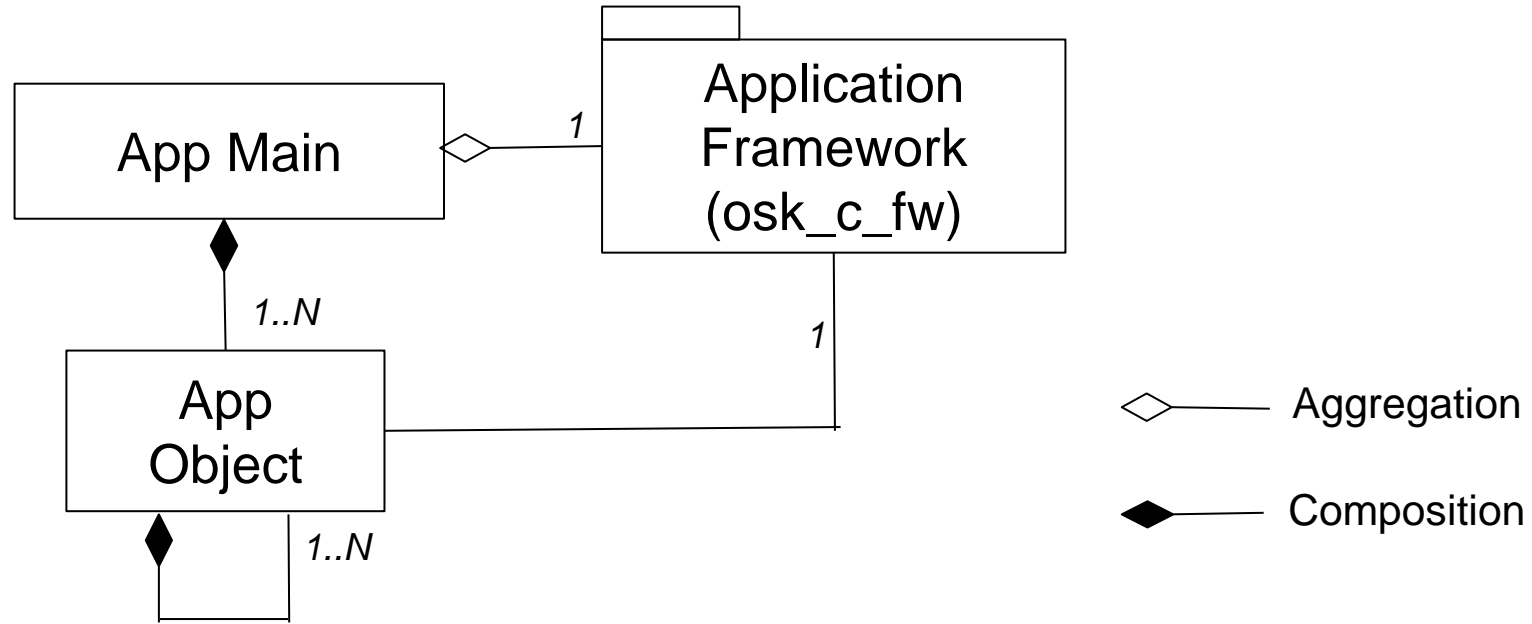


Application Framework Architecture

- **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
 - OSK contains a preliminary C++ framework called `osk_cpp_fw`
- **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>

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



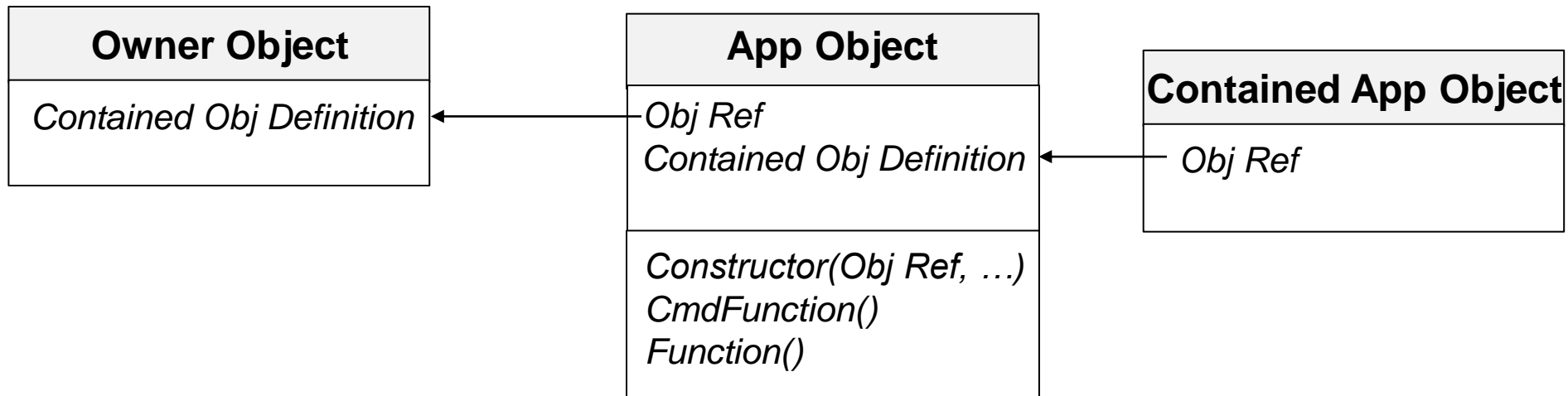
- **Aggregation** represents a relationship where the contained object (non-diamond connector) can exist independent of the owner
 - Conceptually one `osk_c_fw` exists for all applications
- **Composition** represents a relationship where the contained cannot exist independent of the owner
 - Application objects exist 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 Utility¹	fileutil	Utilities for verifying and manipulating files
Packet Utility¹	pktutil	Utilities for verifying and manipulating packets
CJSON	cjson	Adapter for interfacing to the FreeRTOS coreJSON library
JSON²	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 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 and methods that are intrinsic to the responsibilities of that concept
- **The figure below shows the object composition model**

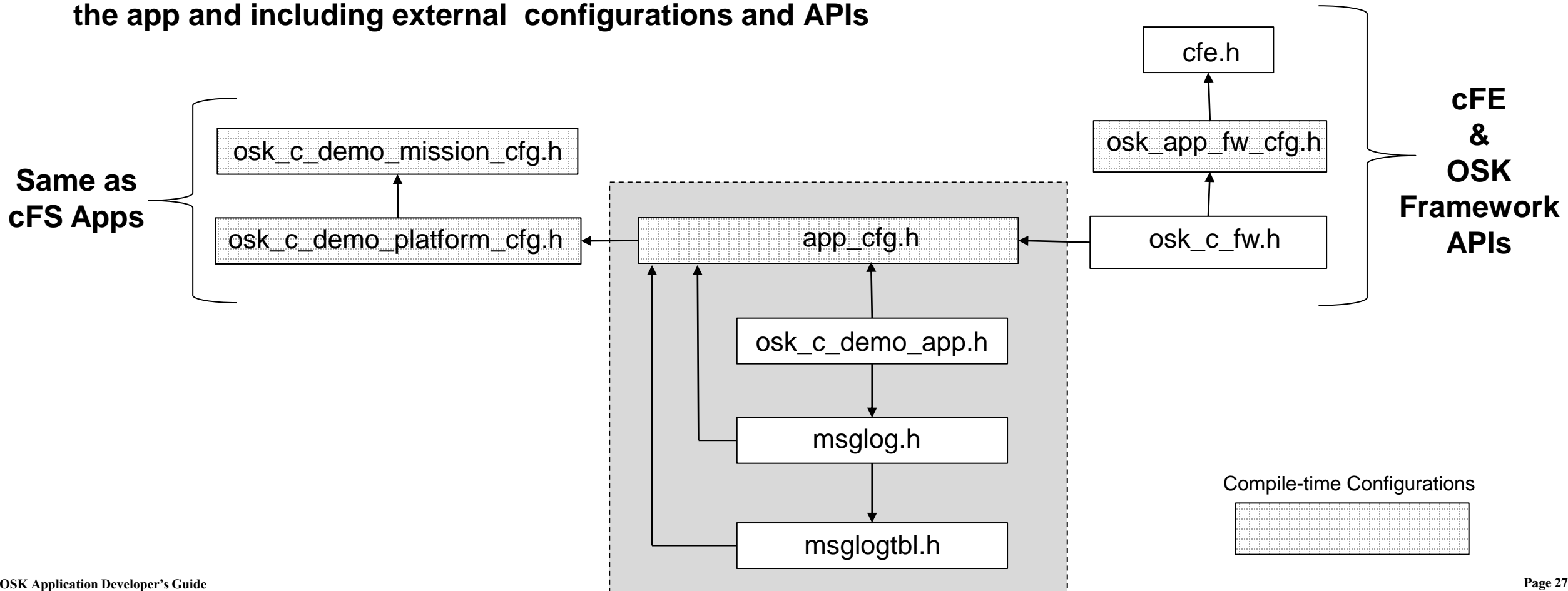


- **Owner objects** define the data for objects they contain and pass a reference to the contained object's constructor
- **Contained objects** store a reference to the owner's instance data
 - Only one instance of an object modeled after the App Object design pattern can exist in an app
 - Analogous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

- **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

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



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 OSK app configuration features
app_cfg.h	Every OSK app has a header with this name. Configurations have an application scope that define parameters that shouldn't need to change across deployments.
osk_c_fw.h	Defines the API for the OSK C Application Framework by including all of the framework component public header files
osk_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 OSK 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

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*   InitTbl;
    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

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

- **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

- **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,...)`

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 shouldn't need to change across deployments.
Initialization Table	Defines configuration parameters that be established 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.



OSK_C_FW Initialization Table

- **JSON file that defines application runtime configurations**
 - If a config parameter impacts a data structure, then it must be defined in a header file at the appropriate scope
- **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*) \
```

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

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

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.InitTbl, 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.



OSK_C_FW Command Manager

Command Manager Overview

CmdMgr
<i>Command Counters</i>
<i>Constructor()</i> <i>RegisterFunc()</i> <i>RegisterAltFunc()</i> <i>ResetStatus()</i> <i>DispatchFunc()</i>

- **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

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);
```




OSK_C_FW Table Manager

Table Manager Overview

TblMgr
<i>Load/Dump Status</i>
<i>Constructor()</i> <i>RegisterTbl ()</i> <i>RegisterTblWithDefs()</i> <i>LoadTblCmd()</i> <i>DumpTblCmd()</i> <i>ResetStatus()</i> <i>GetLastStatus()</i>

- 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 will 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

- **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

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));
```

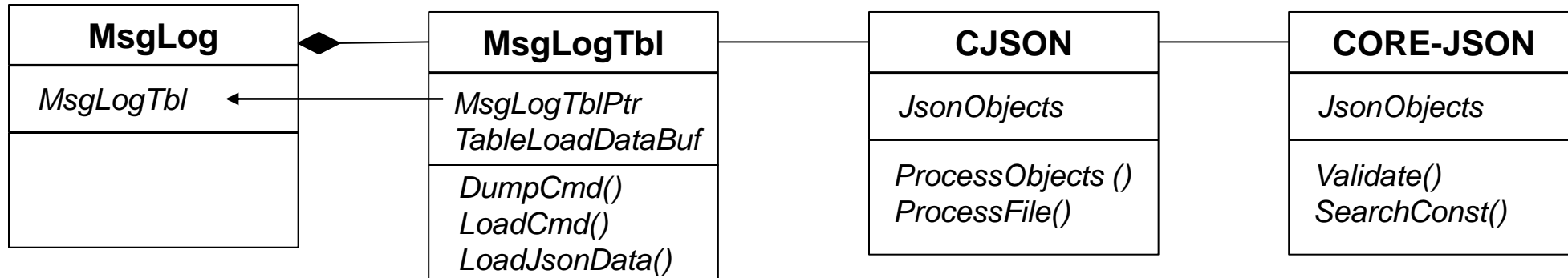
4. App Init: Register TblMgr's Load and Dump commands with CmdMgr

```
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

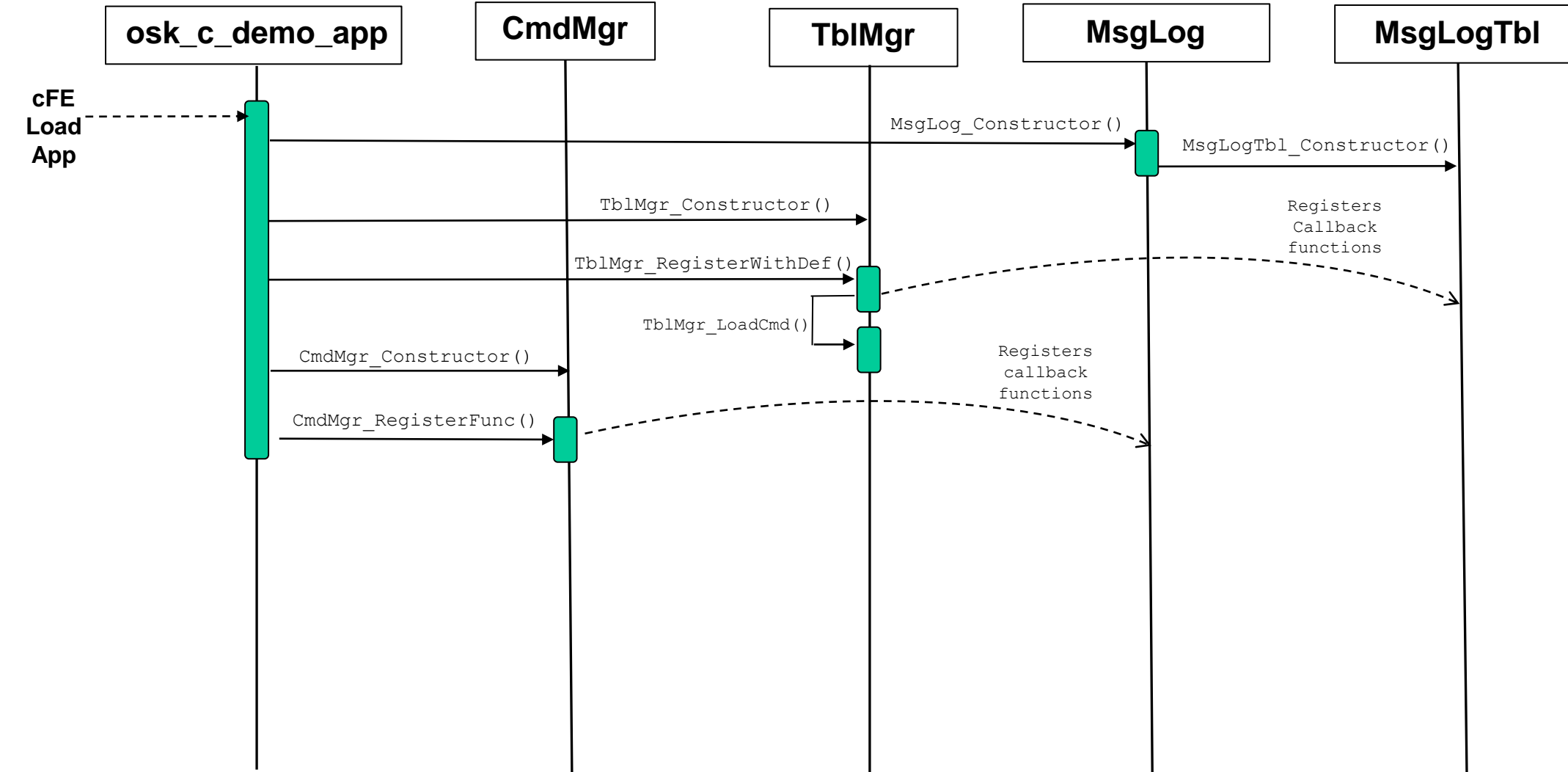
- The following slides use MsgLogTbl as an example to show to create a table object

Table Manager Object Design

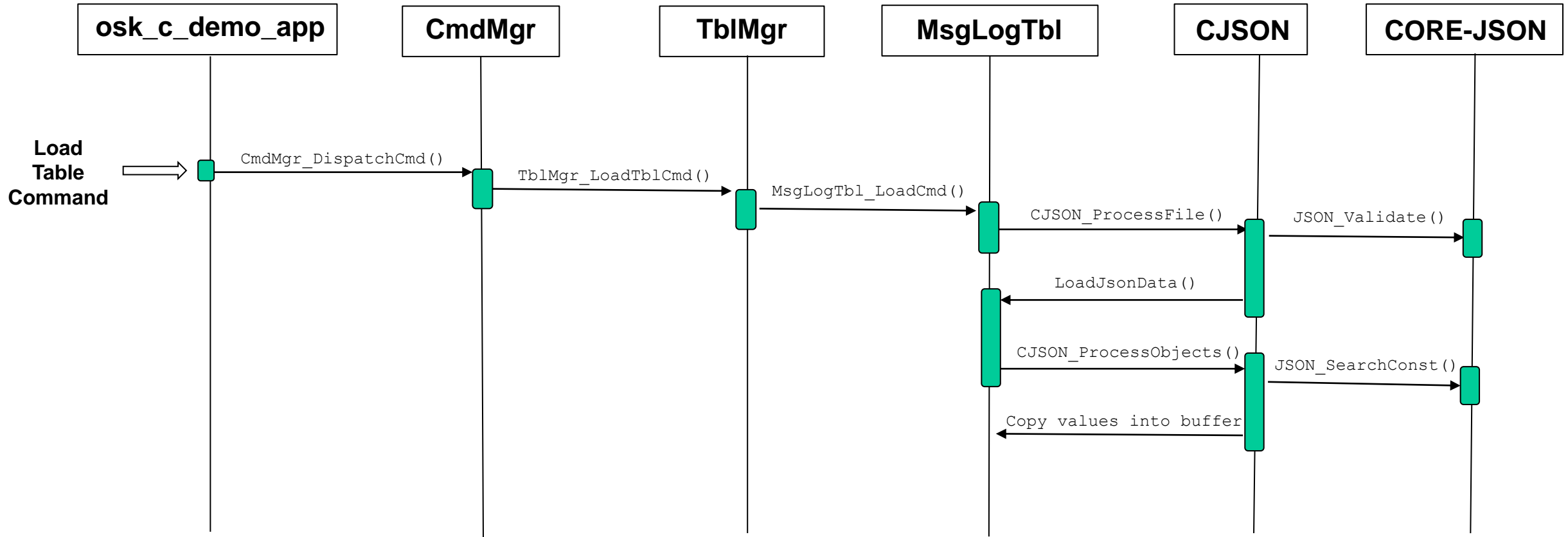


- **MsgLog** is the parent of **MsgLogTbl** so it contains an instance of **MsgLogTbl**
- **MsgLogTbl**
 - *MsgLogTblPtr* references *MsgLog*'s instance of *MsgLogTbl*
 - *TableLoadDataBuf* stores table load data and its contents are copied to *MsgLog*'s instance if the table load is successful
 - *LoadCmd()* 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 copy data into its 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.

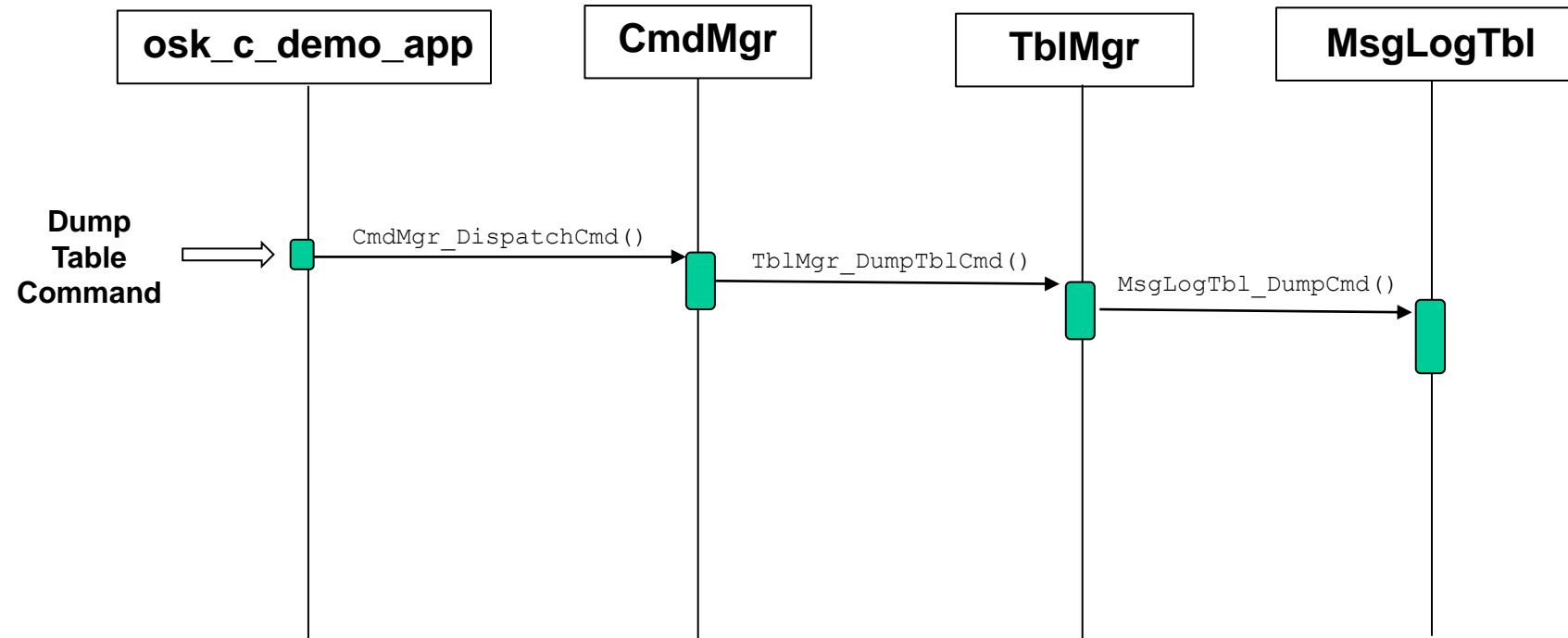
Table Initialization Sequence Diagram



Load Table Sequence Diagram



Dump Table Sequence Diagram



```
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.



OSK_C_FW

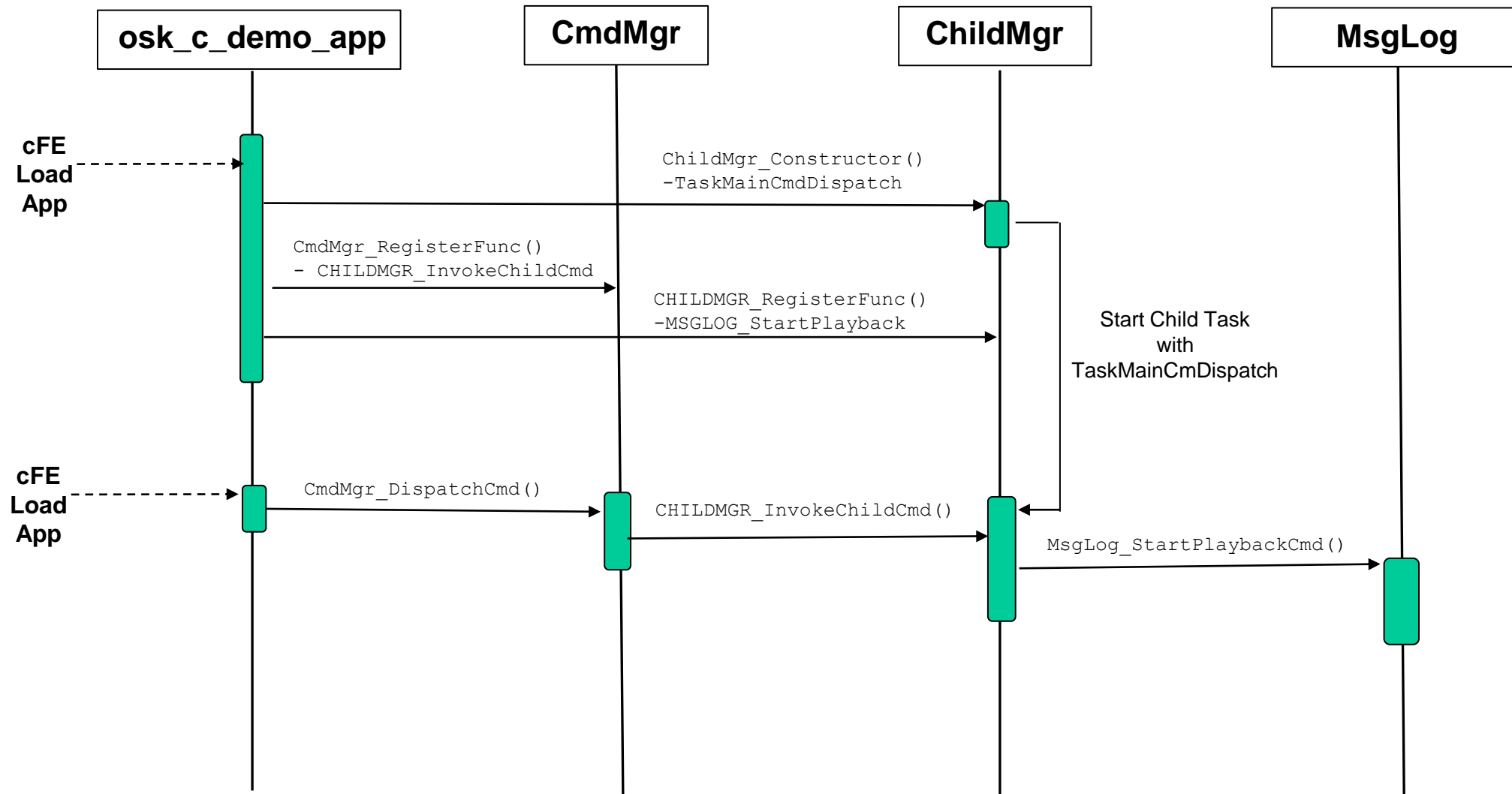
Child Task Manager

- **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 not 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

ChildMgr
<i>CmdQueue</i> <i>Task Info</i> <i>Cmd & Task Status</i>
<i>Constructor()</i> <i>RegisterFunc()</i> <i>ResetStatus()</i> <i>InvokeChildCmd()</i> <i>PauseTask()</i> <i>TaskMainCallback()</i> <i>TaskMainDispatch()</i>

- **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 loop that pends on the Command Queue semaphore

MsgLog Start Playback Sequence Diagram





OSK_C_FW

State Reporter



ToDo



- **TBD – Coming Soon**



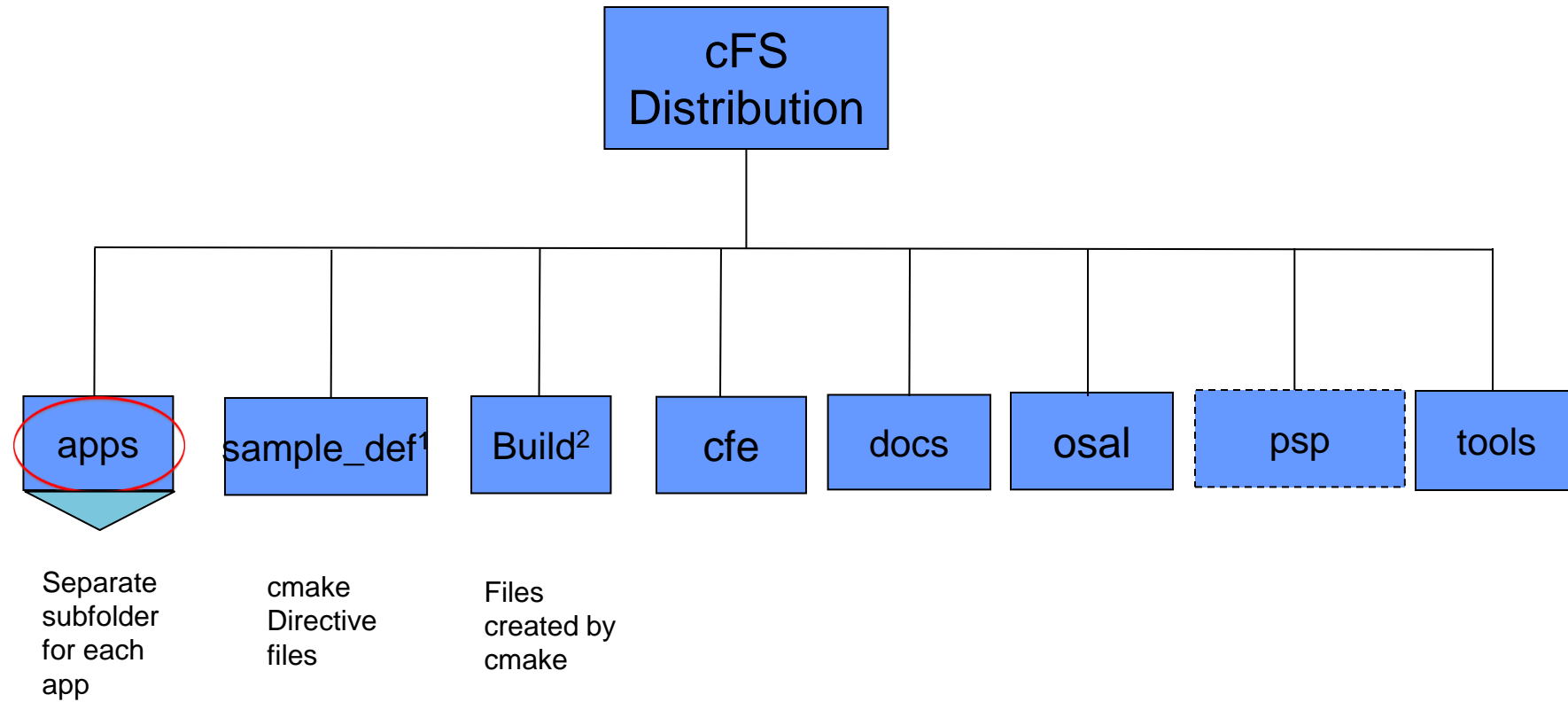
OSK_C_FW

Utilities

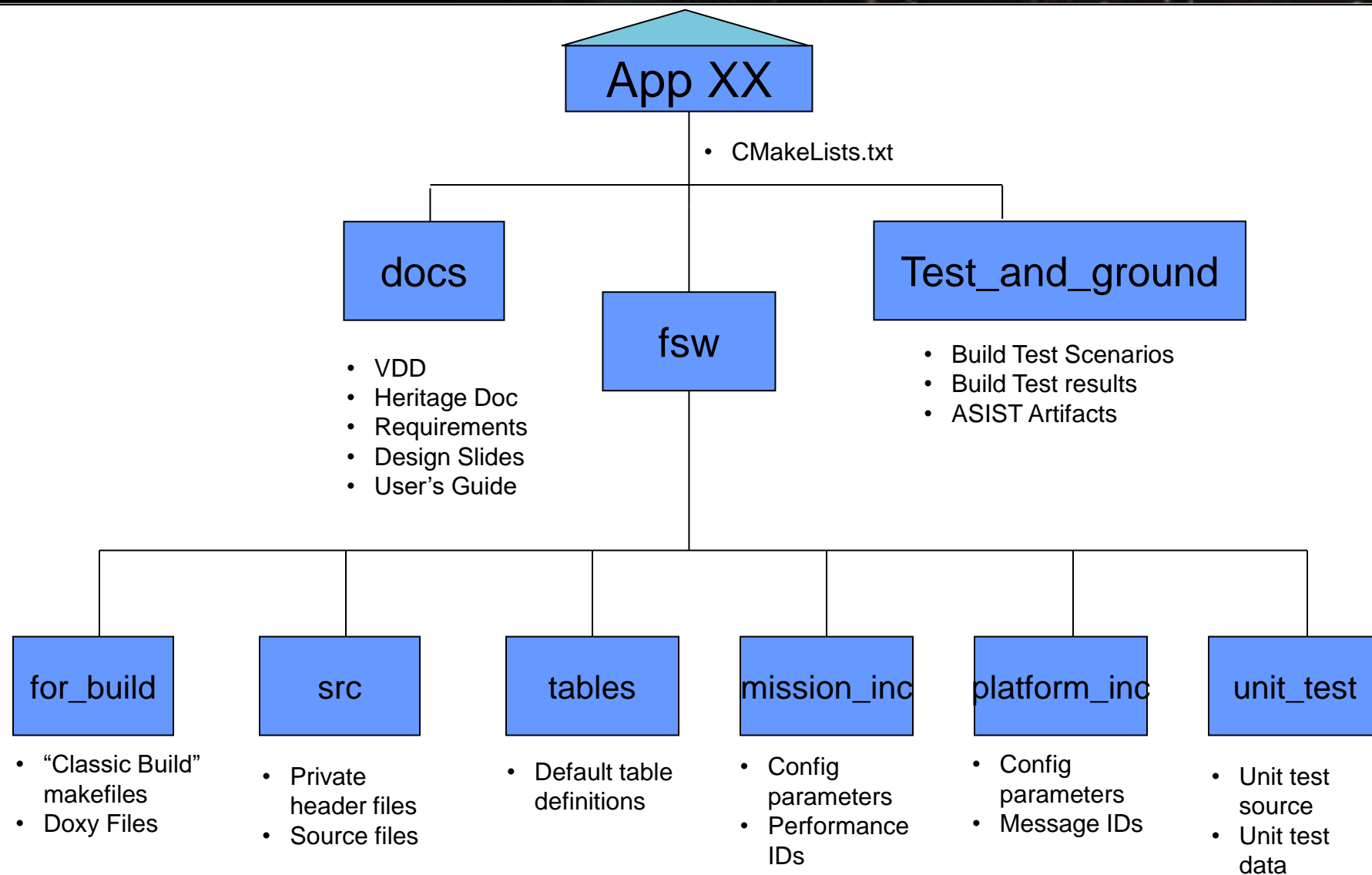
- **osk_c_fw utilities are collections of functions that operate on the function parameters**
 - In OO parlance they are like class functions as opposed to instance functions
 - There is no object instance with state information
- **In v3.1 osk_c_fw contains two utilities: FileUtil (fileutil.h) and PktUtil (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



Building and Running Applications



1. Initially copied from `.../cfe/cmake/sample_def`. Missions typically rename this directory
2. Files created by cmake





- **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



Testing Considerations



Introduction



- **TBD – This section will cover unit and functional testing**



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)

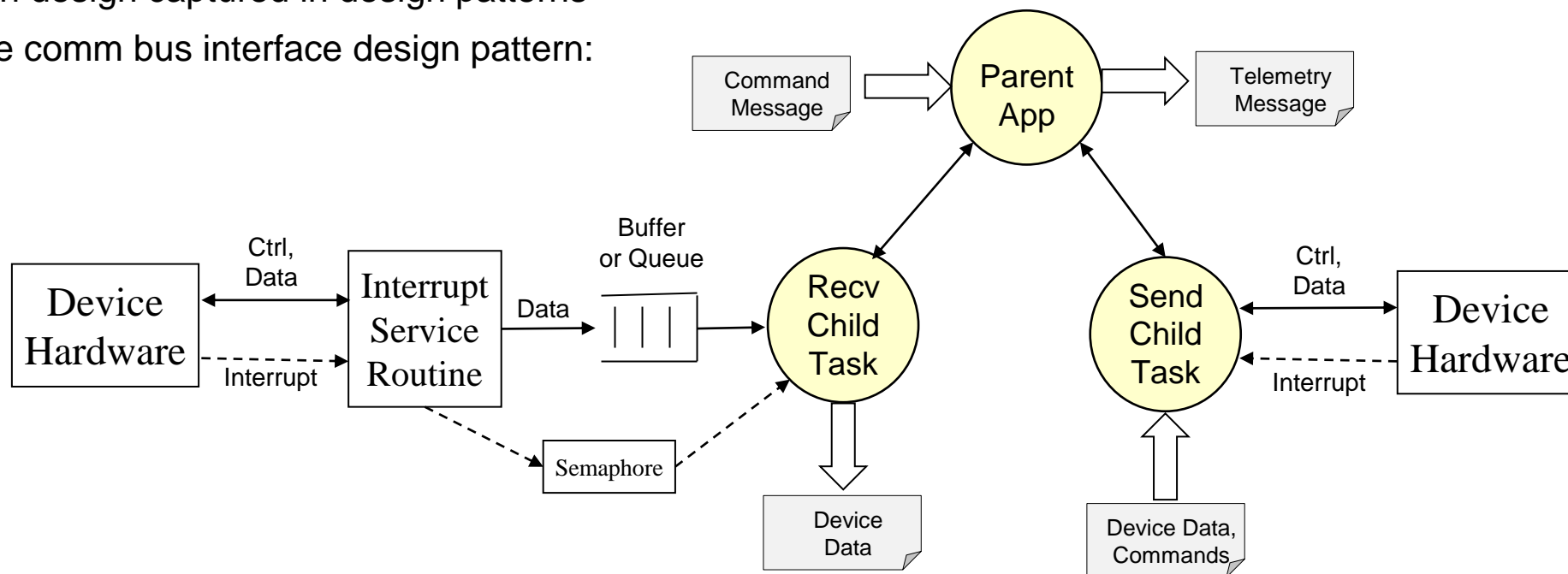




Long Processing Child Tasks

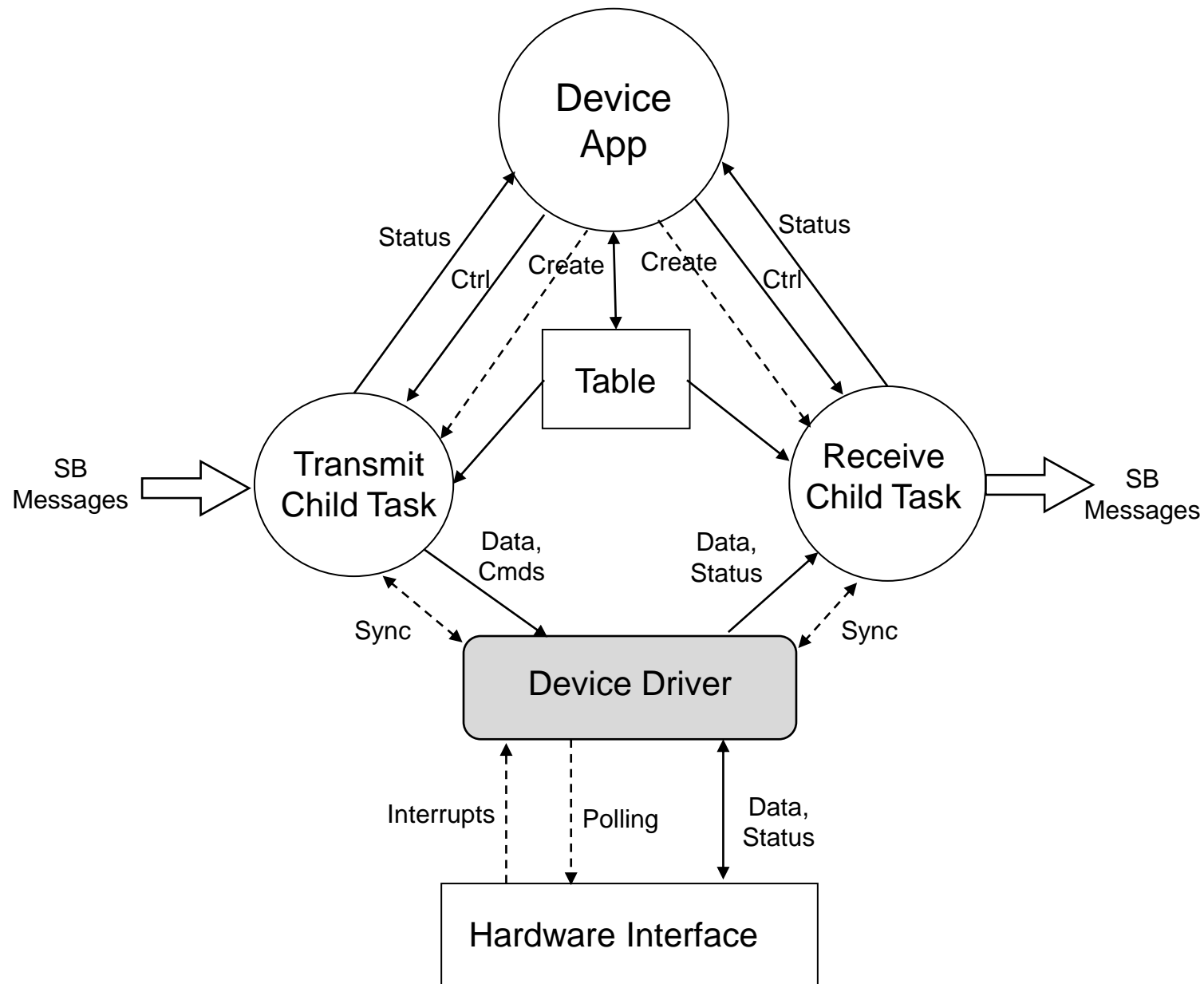


- **Device abstraction architectural role**
 - Read device data and publish on message bus
 - Receive messages and send to the device
- **Use a combination of software components to manage control/data**
 - Common design captured in design patterns
 - Example comm bus interface design pattern:



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

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





Comm I/O Application Design Pattern



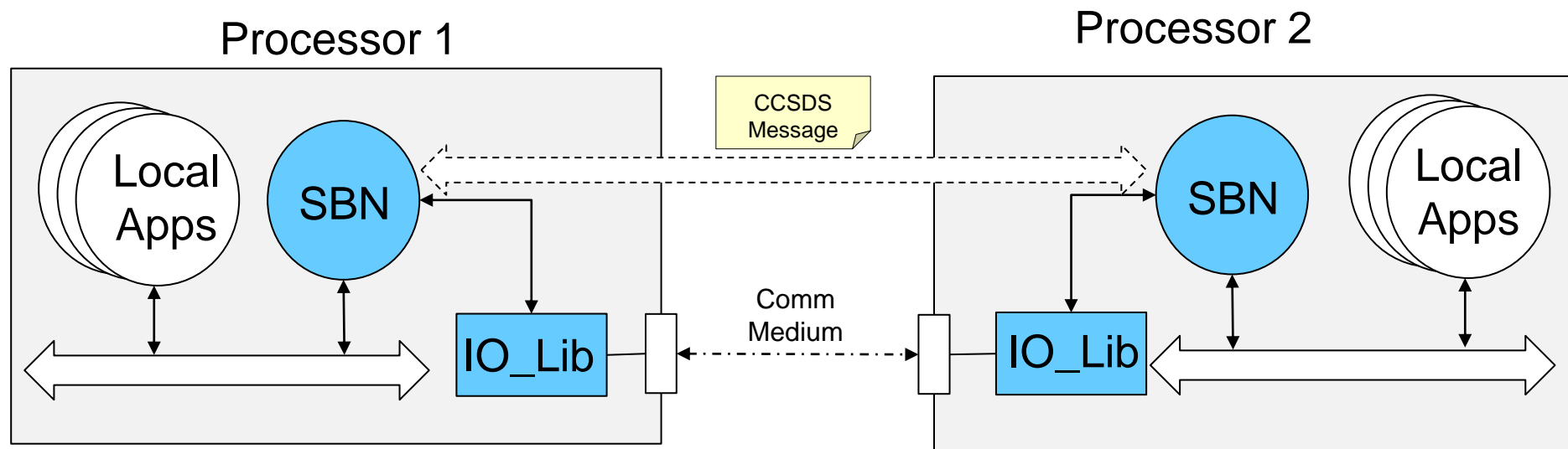
The diagram is accurate from a design perspective but it's a little misleading and the implementation is worth noting. The misleading part is that the shared table only contains what is used by both child tasks and there are other configuration tables that are not shared which are not shown in the diagram.

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)
```

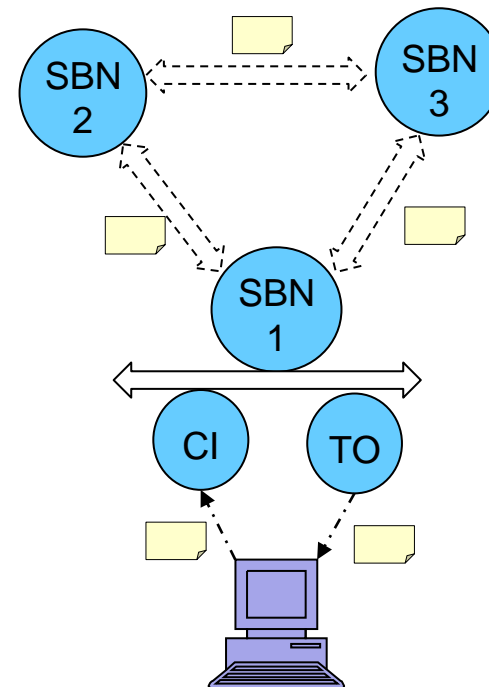
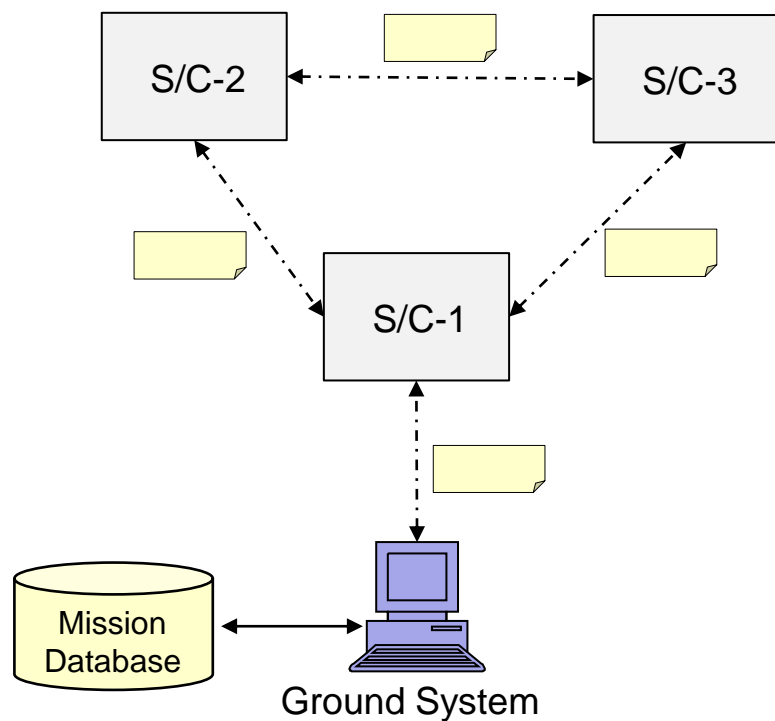
The child tasks use the global table pointer to access the table data

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



- **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





OSK_C_DEMO Application

- **Upon 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

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**

Message Log in Progress

OSK_C_DEMO DEMO_OPS_SCREEN

OSK C Demo

Commands

No Op	Reset	Load Tbl	Dump Tbl
Start Log	Stop Log	Start PlayBk	Stop PlayBk

Housekeeping Status

Cmd Cnt

2

Cmd Err

0

Child Cmd Cnt

2

Child Cmd Err

0

Log Ena

TRUE

Log Count

1

Playbk Ena

FALSE

Filename

/cf/msg_0801.txt

Display

Message Log File Playback

Entry

0

Pri Header

Flight Event Messages

Created new log file /cf/msg_0801.txt with a maximum of 5 entries

Log File Playback in Progress

OSK_C_DEMO DEMO_OPS_SCREEN

OSK C Demo

Commands

No Op	Reset	Load Tbl	Dump Tbl
Start Log	Stop Log	Start PlayBk	Stop PlayBk

Housekeeping Status

Cmd Cnt

3

Cmd Err

0

Child Cmd Cnt

3

Child Cmd Err

0

Log Ena

FALSE

Log Count

5

Playbk Ena

TRUE

Filename

/cf/msg_0801.txt

Display

Message Log File Playback

Entry

1

Pri Header

0801 C026 0095

Flight Event Messages

Playback file /cf/msg_0801.txt started with a 5 cycle delay between updates

- 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

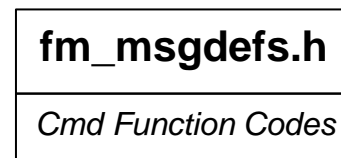
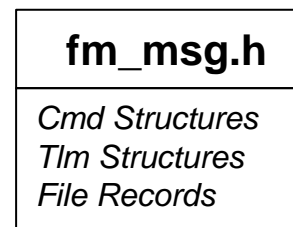
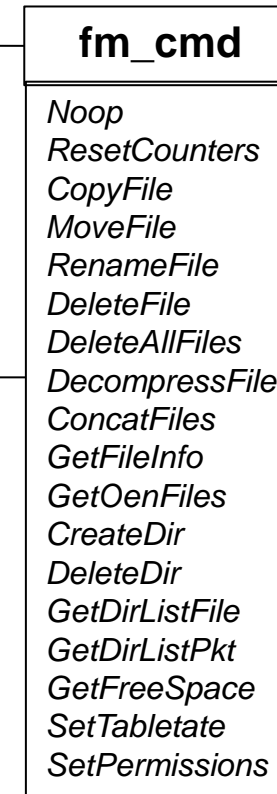
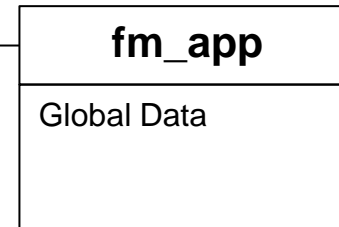
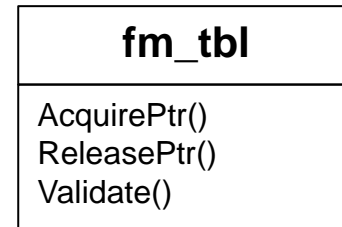
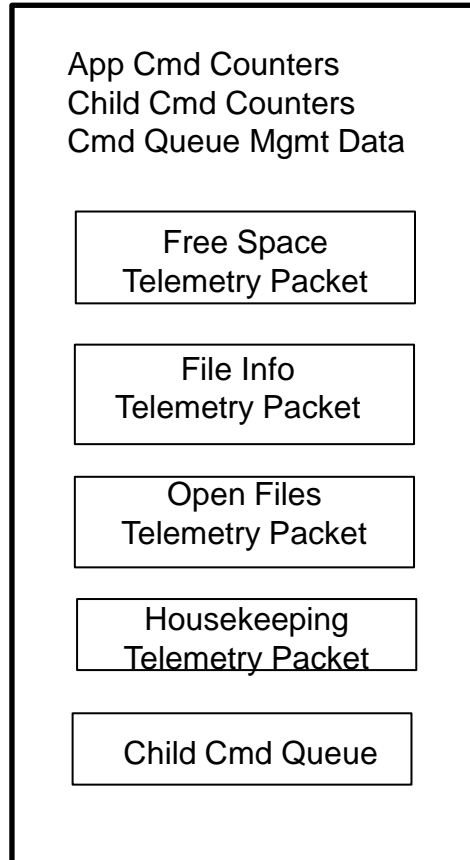


File Manager Refactor Overview

- **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

- **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**

FM Global Data



- **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;

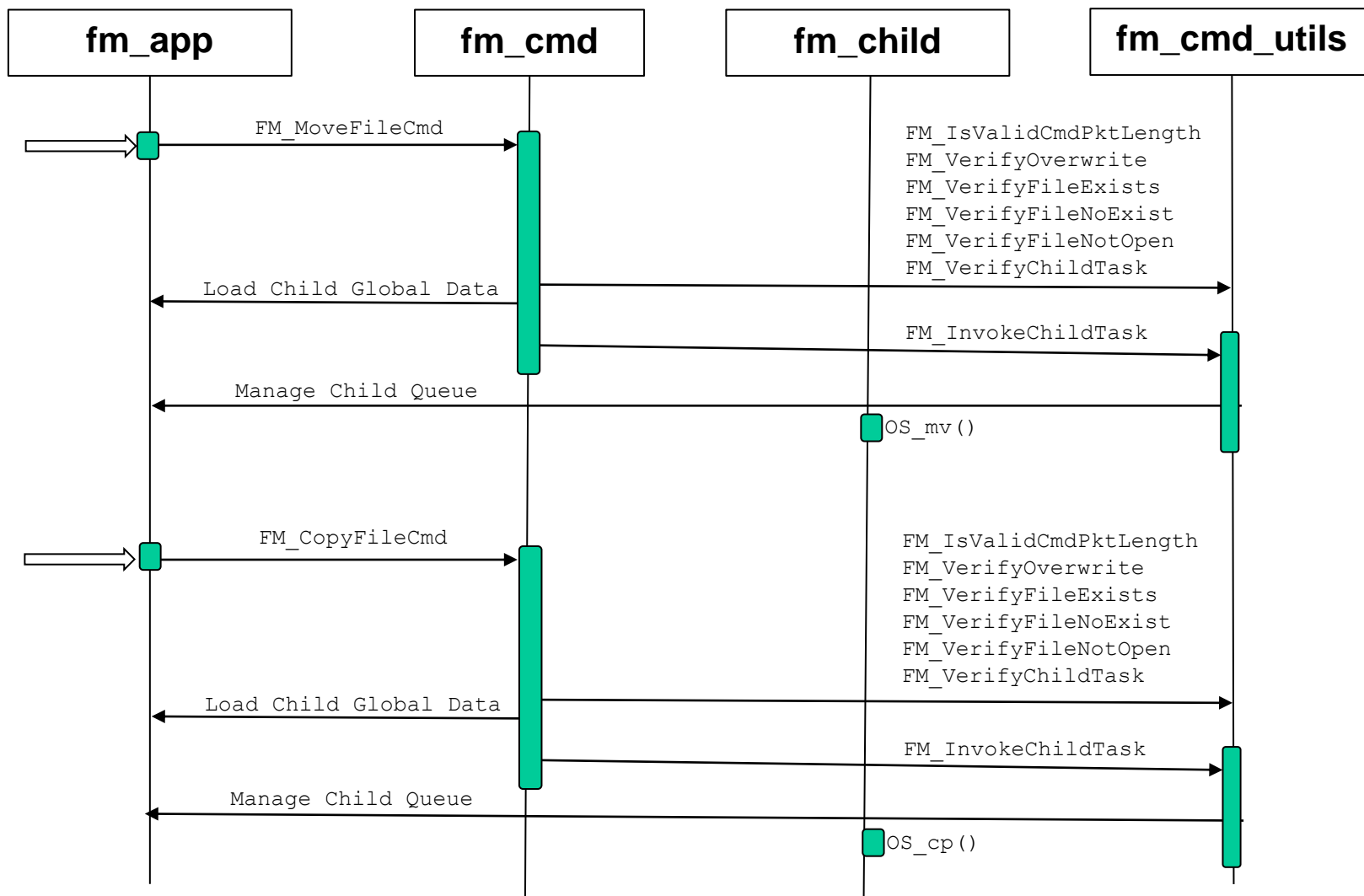
    /* Verify command packet length */
    CommandResult = FM_IsValidCmdPktLength(MessagePtr, sizeof(FM_CopyFileCmd_t), FM_COPY_PKT_ERR_EID, CmdText);
    /* Verify that overwrite argument is valid */
    if (CommandResult == TRUE) {
        CommandResult = FM_VerifyOverwrite(CmdPtr->Overwrite, FM_COPY_OVR_ERR_EID, 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();
    }
}
```



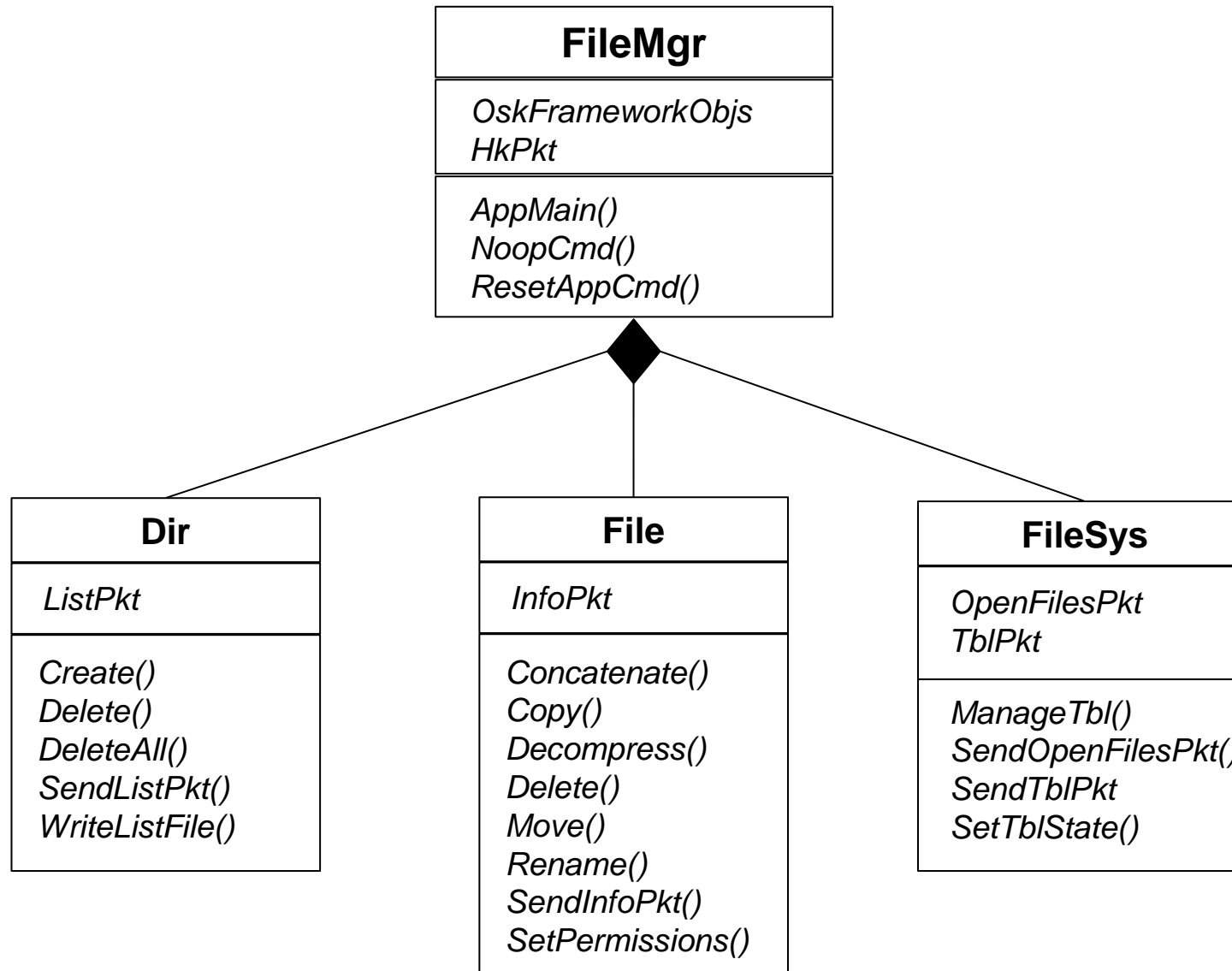
File Move



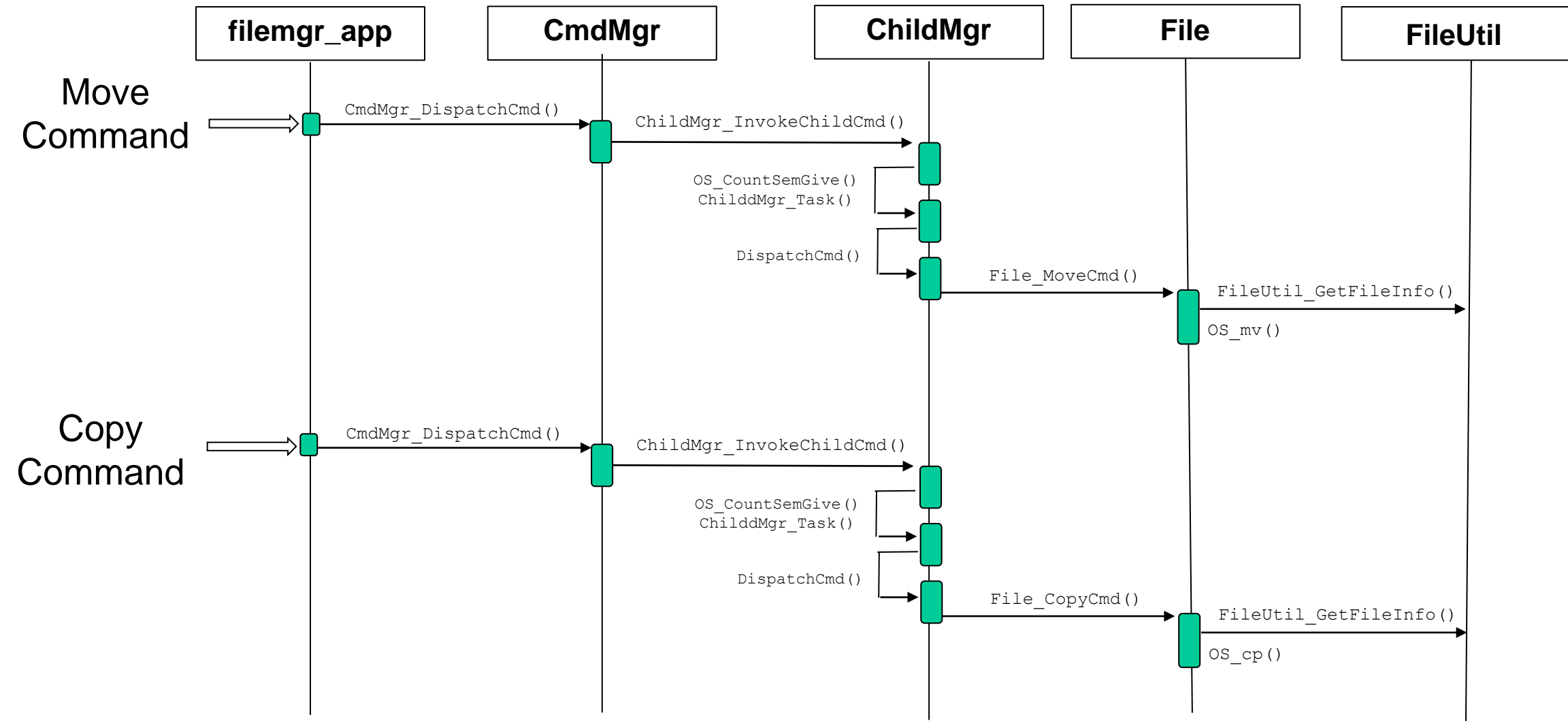
```
boolean FM_MoveFileCmd(CFE_SB_MsgPtr_t MessagePtr){    FM_MoveFileCmd_t  *CmdPtr = (FM_MoveFileCmd_t *) MessagePtr;
FM_ChildQueueEntry_t *CmdArgs;    char *CmdText = "Move File";    boolean CommandResult;    /* Verify command packet
length */    CommandResult = FM_IsValidCmdPktLength(MessagePtr, 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() */
```



Copy Command



Refactored File Manager Sequence Diagram



- **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