

MIDDLE EAST TECHNICAL UNIVERSITY

SMALL SPACE SYSTEMS LABORATORY

ON-BOARD COMPUTER TEAM

# NASA Core Flight System (cFS) Technical Documentation

June 7, 2025

# Contents

<b>1</b>	<b>Installation, First Run and System Requirements</b>	<b>2</b>
<b>2</b>	<b>cFS Architecture</b>	<b>3</b>
2.1	Start-Up Script . . . . .	3
<b>3</b>	<b>App Developing</b>	<b>4</b>
3.1	Creating the Hello World App . . . . .	4
3.2	Internal Communication Through Software Bus . . . . .	6
3.3	Events in cFS . . . . .	11
3.4	Tables in cFS . . . . .	14
<b>4</b>	<b>Built-In Applications</b>	<b>19</b>
4.1	Scheduler Application . . . . .	20
4.2	Data Storage . . . . .	23

# 1 Installation, First Run and System Requirements

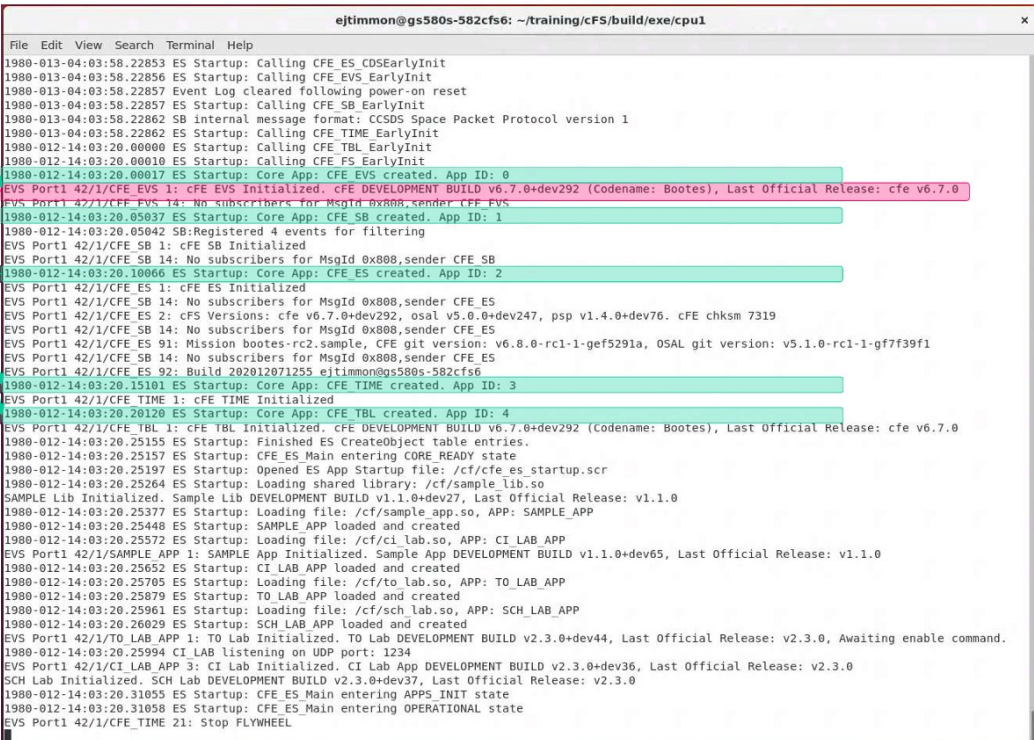
Throughout the explanation of the features and sample apps of cFS, an Ubuntu 22.04 LTS OS will be used. However, any Linux distribution or any OS that cFS supports can be used by reader. Note that there will be additional considerations if a Raspberry Pi is used, which will be explained in later chapters. For now, it is strongly suggested to use a Linux distribution on a computer with x86\_64 processor.

For the installation of cFS, please refer to the setup section of [this](#) document. Please show care to the copy commands.

Then, stay in the cFS folder and run the following commands for the initial run of the cFS software. What you ran will be explained in the following chapter. Note that you might need admin (sudo) privileges.

```
make prep
make
make install
cd build/exe/cpu1/
./core-cpu1
```

After running the commands, you should see a screen like in the Figure 1.



The image shows a terminal window titled "ejtimmon@gs580s-582cfs6: ~/training/cFS/build/exe/cpu1". The terminal displays a series of log messages from the cFS startup process. On the left side, there are three green arrows pointing to specific lines in the log, with labels: "cFE Version" pointing to "cfe v6.7.0+dev292", "cFE Services Started" pointing to "Core App: CFE ES created", and "cFE Services Started" pointing to "Core App: CFE TBL created". The log messages include timestamps, event IDs, and descriptions of the startup process, such as "Calling CFE ES CDS EarlyInit", "Event Log cleared following power-on reset", "SB internal message format: CCSDS Space Packet Protocol version 1", "Calling CFE TIME EarlyInit", "Calling CFE TBL EarlyInit", "Calling CFE FS EarlyInit", "Core App: CFE EVS created", "Core App: CFE SB created", "Core App: CFE ES created", "Core App: CFE TBL created", "Core App: CFE TIME created", "Finished ES CreateObject table entries", "CFS Main entering CORE\_READY state", "Opened ES App Startup file", "Loading shared library", "Sample Lib Initialized", "Sample App DEVELOPMENT BUILD v1.1.0+dev27", "Sample App loaded and created", "CI Lab APP loaded and created", "TO Lab APP loaded and created", "SCH Lab APP loaded and created", "CI Lab listening on UDP port: 1234", "CI Lab APP loaded and created", "SCH Lab DEVELOPMENT BUILD v2.3.0+dev37", "CFS Main entering APPS\_INIT state", "CFS Main entering OPERATIONAL state", and "Stop FLYWHEEL".

```
File Edit View Search Terminal Help
1980-013-04:03:58.22853 ES Startup: Calling CFE ES CDS EarlyInit
1980-013-04:03:58.22856 ES Startup: Calling CFE EVS EarlyInit
1980-013-04:03:58.22857 Event Log cleared following power-on reset
1980-013-04:03:58.22857 ES Startup: Calling CFE SB EarlyInit
1980-013-04:03:58.22862 SB internal message format: CCSDS Space Packet Protocol version 1
1980-013-04:03:58.22862 ES Startup: Calling CFE TIME EarlyInit
1980-012-14:03:20.00000 ES Startup: Calling CFE TBL EarlyInit
1980-012-14:03:20.00010 ES Startup: Calling CFE FS EarlyInit
1980-012-14:03:20.00017 ES Startup: Core App: CFE EVS created. App ID: 0
EVS Port1 42/1/CFE EVS 1: cFE EVS Initialized. cFE DEVELOPMENT BUILD v6.7.0+dev292 (Codename: Bootes), Last Official Release: cfe v6.7.0
EVS Port1 42/1/CFE EVS 14: No subscribers for MsgId 0x808, sender CFE EVS
1980-012-14:03:20.05037 ES Startup: Core App: CFE SB created. App ID: 1
1980-012-14:03:20.05042 SB:Registered 4 events for filtering
EVS Port1 42/1/CFE SB 1: cFE SB Initialized
EVS Port1 42/1/CFE SB 14: No subscribers for MsgId 0x808, sender CFE SB
1980-012-14:03:20.10066 ES Startup: Core App: CFE ES created. App ID: 2
EVS Port1 42/1/CFE ES 1: cFE ES Initialized
EVS Port1 42/1/CFE ES 14: No subscribers for MsgId 0x808, sender CFE ES
EVS Port1 42/1/CFE ES 2: cFS Versions: cfe v6.7.0+dev292, osal v5.0.0+dev247, psp v1.4.0+dev76. cfe chksm 7319
EVS Port1 42/1/CFE SB 14: No subscribers for MsgId 0x808, sender CFE ES
EVS Port1 42/1/CFE ES 91: Mission bootes-rc2.sample, cFE git version: v6.8.0-rc1.1-gf7f39f1, OSAL git version: v5.1.0-rc1.1-gf7f39f1
EVS Port1 42/1/CFE SB 14: No subscribers for MsgId 0x808, sender CFE ES
EVS Port1 42/1/CFE ES 92: Build 202012071255 ejtimmon@gs580s-582cfs6
1980-012-14:03:20.15101 ES Startup: Core App: CFE TIME created. App ID: 3
EVS Port1 42/1/CFE TIME 1: cFE TIME Initialized
1980-012-14:03:20.20120 ES Startup: Core App: CFE TBL created. App ID: 4
EVS Port1 42/1/CFE TBL 1: cFE TBL Initialized. cFE DEVELOPMENT BUILD v6.7.0+dev292 (Codename: Bootes), Last Official Release: cfe v6.7.0
1980-012-14:03:20.25155 ES Startup: Finished ES CreateObject table entries.
1980-012-14:03:20.25157 ES Startup: CFE ES Main entering CORE_READY state
1980-012-14:03:20.25197 ES Startup: Opened ES App Startup file: /cf/cfe_es_startup.scr
1980-012-14:03:20.25264 ES Startup: Loading shared library: /cf/sample_lib.so
SAMPLE Lib Initialized. Sample Lib DEVELOPMENT BUILD v1.1.0+dev27, Last Official Release: v1.1.0
1980-012-14:03:20.25377 ES Startup: Loading file: /cf/sample_app.so, APP: SAMPLE_APP
1980-012-14:03:20.25448 ES Startup: SAMPLE_APP loaded and created
1980-012-14:03:20.25572 ES Startup: Loading file: /cf/ci_lab.so, APP: CI LAB_APP
EVS Port1 42/1/SAMPLE APP 1: SAMPLE App Initialized. Sample App DEVELOPMENT BUILD v1.1.0+dev65, Last Official Release: v1.1.0
1980-012-14:03:20.25652 ES Startup: CI LAB_APP loaded and created
1980-012-14:03:20.25705 ES Startup: Loading file: /cf/to_lab.so, APP: TO LAB_APP
1980-012-14:03:20.25879 ES Startup: TO LAB_APP loaded and created
1980-012-14:03:20.25961 ES Startup: Loading file: /cf/sch_lab.so, APP: SCH LAB_APP
1980-012-14:03:20.26029 ES Startup: SCH LAB_APP loaded and created
EVS Port1 42/1/TO LAB APP 1: TO Lab Initialized. TO Lab DEVELOPMENT BUILD v2.3.0+dev44, Last Official Release: v2.3.0, Awaiting enable command.
1980-012-14:03:20.25994 CI LAB listening on UDP port: 1234
EVS Port1 42/1/CI LAB APP 3: CI Lab Initialized. CI Lab App DEVELOPMENT BUILD v2.3.0+dev36, Last Official Release: v2.3.0
SCH Lab Initialized. SCH Lab DEVELOPMENT BUILD v2.3.0+dev37, Last Official Release: v2.3.0
1980-012-14:03:20.31055 ES Startup: CFE ES Main entering APPS_INIT state
1980-012-14:03:20.31058 ES Startup: CFE ES Main entering OPERATIONAL state
EVS Port1 42/1/CFE_TIME 21: Stop FLYWHEEL
```

cFS Training- Page 57

Figure 1: Expected Results of The First Run

## 2 cFS Architecture

Now, it is time to analyze what we ran in Chapter 1. From Figure 1, it can be seen that program started the core cFE services. Following that, it initiated some apps, which were *SAMPLE\_APP*, *TO\_LAB\_APP*, *SCH\_LAB\_APP*, and *CLLAB\_APP*. The main architecture of cFS can be seen in Figure 2.

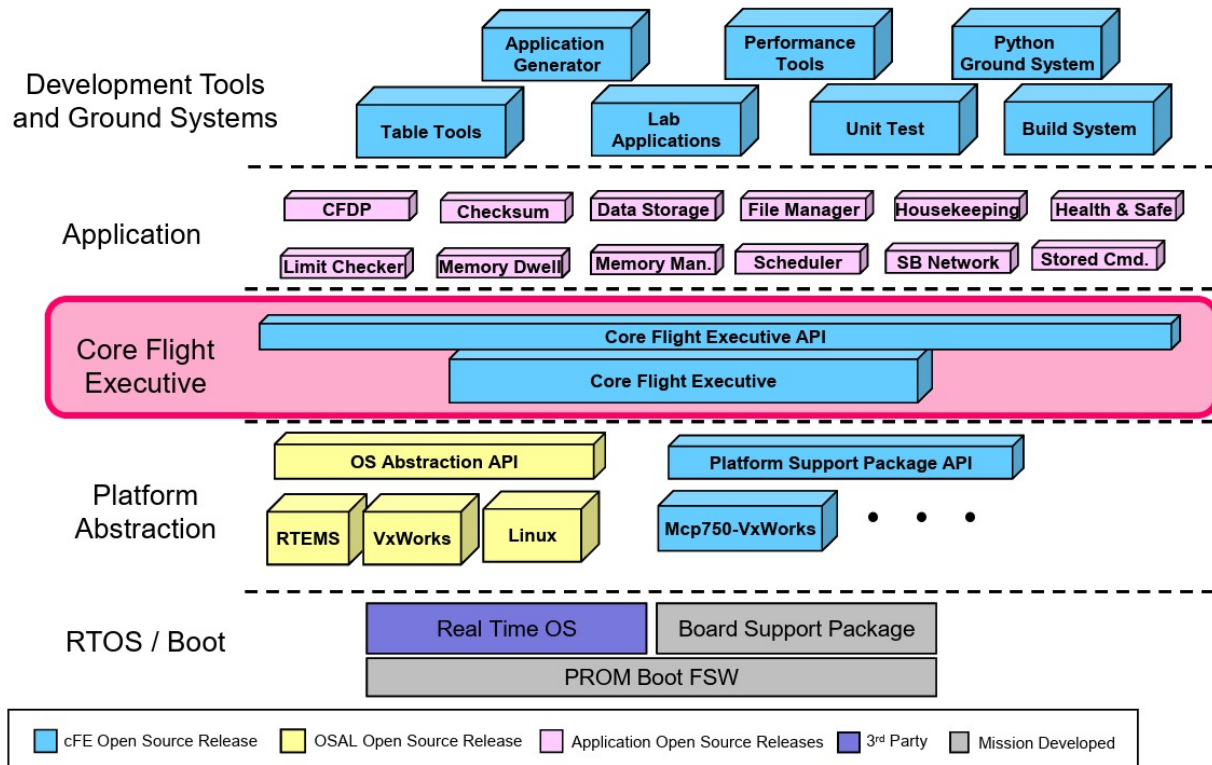


Figure 2: Main Architecture of Core Flight System

So, what happened is, when cFS started with the `./core-cpu1` command, it first initiated cFE (Core Flight Executive), which was marked in Figure 1. Then, cFE initiated some apps that was installed. This is where 3rd party developers come in to play. For dedicated tasks, developers can write their own apps, and they use those apps by integrating with cFE.

The cFE itself also has built-in apps where developers can use, such as event manager, time and table services and software bus. When we move up to application layer, it can also be seen that cFS has some other built-in apps such as memory manager, and data storage engine; other than the ones cFE have. These are developed by the NASA cFS team, and open to public. 3rd party developers can also use them integrated with their own apps, to have a fully working and stable end product.

### 2.1 Start-Up Script

When cFE initializes the apps and system in general, it uses a start-up script. This script will be analyzed further in later chapters, but it is important to know how it utilizes the apps.

It is located in *cfs/sample\_defs/cpu1\_cfe\_es\_startup.scr*. It is used to define the entry points of specific apps and libraries, alongside with some information of those apps. Information on those parameters are written in the file as comments. They will make more sense when we use this in an example.

## 3 App Developing

In this section, an app will be developed from scratch. The app will use most of the important cFS APIs, and its purpose will be to demonstrate some features.

During the development process, you will notice that authors will try to write the code as efficient as possible. That means that the aim is to get the desired result with as little code as possible. Thus, error handling and clean coding standards may not be used during the examples.

### 3.1 Creating the Hello World App

In cFS file system, apps are located in the *cfs/apps/* folder. In this folder, you can see different folders for the apps mentioned in Chapter 2, which were *SAMPLE\_APP*, *TO\_LAB\_APP*, *SCH\_LAB\_APP*, and *CL\_LAB\_APP*.

In the console, assuming you are in the */cfs* folder, use the following commands to create a new app folder named **hello\_app**. Please note that cFS is case sensitive, so it is important that reader uses the same naming methods as the author. However, the system author uses is not the only correct way to do it, reader can also use her/his own method, as long as it is consistent (not recommended for beginners).

```
cd apps
mkdir hello_app
cd hello_app
```

Then, create a header file named *hello\_app.h*. If you have limited knowledge on headers, or C in general, try to learn C before moving forward. This file should be as follows.

```
#ifndef _HELLO_APP_H_
#define _HELLO_APP_H_
#define HELLO_APP_PERF_ID 91 // Unique performance ID for logging
#include "cfe.h"

typedef struct
{
    uint32 RunStatus;
    uint32 counter;
} HELLO_AppData_t;

void HELLO_AppMain(void);

#endif /* _HELLO_APP_H_ */
```

Following that, create a *hello\_app.c* file. This file should be as follows.

```
#include "cfe.h"
#include "hello_app.h"

HELLO_AppData_t HELLO_AppData; // Global app data structure

void HELLO_AppMain(void){
    HELLO_AppData.RunStatus = 1;
    HELLO_AppData.counter = 0;

    while (CFE_ES_RunLoop(&HELLO_AppData.RunStatus)){
        OS_printf("Hello, World!, Counter: %d\n", HELLO_AppData.counter);
        HELLO_AppData.counter++;

        OS_TaskDelay(1000);
    }
}
```

The reader should easily understand what is going on with this code. If this is not the case for you, please try to refresh your C knowledge before moving forward.

Then, recall what is discussed in Chapter 2.1. Our app needs to be added to the start-up script. For that, following line can be added.

```
CFE_APP, hello_app, HELLO_AppMain, HELLO_APP, 50, 16384, 0x0, 0;
```

Also, in the same folder (*~/cfs/sample\_defs/*), there is a file named *targets.cmake*. In around line 89, one can see that how the original authors added the sample app to targeted apps list. To add our app (*hello\_app*), one can add the following line just below the one for the *sample\_app*. Note that there is no implementation of a library concept just yet, thus it doesn't need to be added here.

```
list(APPEND MISSION_GLOBAL_APPLIST hello_app)
```

Now, it is needed to let *cmake* know that this app will be used with the *cFS*. To do that, create a *CMakeLists.txt* file in the *hello\_app* folder. This file should be as follows.

```
cmake_minimum_required(VERSION 3.5)
project(CFS_HELLO_APP C)

add_cfe_app(hello_app hello_app.c)
```

Now, *cFS* is ready to run our app. First, go back to *~/cfs* folder, and run the following commands. Note that you need **cmake** installed. Refer to a web search for the installation.

```
make clean
make prep
make
make install
cd build/exe/cpu1/
./core-cpu1
```

At the end, author should read the Hello World text in console, alongside with a counter. Note that the text needs to be printed every second, with increasing counter. A sample result can be seen in Figure 3.

```

1980-012-14:03:20.50476 CFE_ES_ParseFileEntry: Loading shared library: /cf/sample.lib.so
SAMPLE Lib Initialized.Sample Lib Development Build equuleus-rc1+dev2 (Codename Equuleus), Last Official Release: Sample Lib v1.1.0)
1980-012-14:03:20.50516 CFE_ES_ParseFileEntry: Loading file: /cf/sample_app.so, APP: SAMPLE_APP
1980-012-14:03:20.50550 CFE_ES_ParseFileEntry: Loading file: /cf/ci_lab.so, APP: CI_LAB_APP
1980-012-14:03:20.50593 CFE_ES_ParseFileEntry: Loading file: /cf/to_lab.so, APP: TO_LAB_APP
1980-012-14:03:20.50626 CFE_ES_ParseFileEntry: Loading file: /cf/sch_lab.so, APP: SCH_LAB_APP
1980-012-14:03:20.50652 CFE_ES_ParseFileEntry: Loading file: /cf/hello_app.so, APP: HELLO_APP
EVS Port1 1980-012-14:03:20.55586 66/1/SAMPLE_APP 1: Sample App Initialized.Sample App Development Build equuleus-rc1+dev46 (Codename Equuleus), Last Official Release: Sa
1980-012-14:03:20.55602 CI_LAB listening on UDP port: 1234
EVS Port1 1980-012-14:03:20.55603 66/1/CI_LAB_APP 3: CI Lab Initialized.CI Lab App Development Build equuleus-rc1+dev61 (Codename Equuleus), Last Official Release: CI Lab
EVS Port1 1980-012-14:03:20.55648 66/1/TO_LAB_APP 1: TO Lab Initialized.TO Lab Development Build equuleus-rc1+dev62 (Codename Equuleus), Last Official Release: TO Lab v2.
Hello, world!, Counter: 0
SCH Lab Initialized.SCH Lab Development Build equuleus-rc1+dev29 (Codename Equuleus), Last Official Release: SCH Lab v2.3.0)
1980-012-14:03:20.60703 CFE_ES_Main: CFE_ES_Main entering APPS_INIT state
1980-012-14:03:20.60705 CFE_ES_Main: CFE_ES_Main entering OPERATIONAL state
Hello, world!, Counter: 1
EVS Port1 1980-012-14:03:21.00006 66/1/CFE_TIME 21: Stop FLYWHEEL
Hello, world!, Counter: 2
Hello, world!, Counter: 3
Hello, world!, Counter: 4
Hello, world!, Counter: 5
Hello, world!, Counter: 6
Hello, world!, Counter: 7
Hello, world!, Counter: 8
Hello, world!, Counter: 9
Hello, world!, Counter: 10
Hello, world!, Counter: 11
Hello, world!, Counter: 12
Hello, world!, Counter: 13
Hello, world!, Counter: 14

```

Figure 3: Expected Result For Hello World App

### 3.2 Internal Communication Through Software Bus

cFS also allows internal communication through a software bus. The proposed idea is a publish/-subscribe model. Apps are allowed to publish messages/data via a created channel, and any app can access this channel to read & publish more data. There is no limitation or a firewall. Apps can use these channels freely, as long as they reach the channel via an ID.

To work with an example, there will be two different apps. One named *sender\_app* and other named *engine\_app*. In this example, sender app will constantly send housekeeping data for battery systems (it is only for example, data can be anything). The engine app will only read this housekeeping data from the channel. Creating these apps are explained in Chapter 3.1. To follow up with the examples, go ahead and create these two apps.

Creating the engine app is relatively easier, since its task is to read data. However, sender app is a bit more tricky. To create this app, one needs to:

- Create a Payload Structure
- Import a Message Header Structure
- Create a Messaging Channel

They are not hard to do, but it is important to make everything correct in this step. Using a pre-built telemetry header (recommended) from cFE. the *sender\_app.h* file should be as follows. Understanding this code can be a bit challenging. Note the the mentioned channel ID is defined as *CFE\_MISSION\_SENDER\_APP\_HK\_TLM\_TOPICID*, and it can be any number as long as it is unique. Payload is defined such that it gives battery fill rate, battery health, a message counter and timestamp.



```

#ifdef _SENDER_APP_H_
#define _SENDER_APP_H_

#include "cfe.h"
#include "cfe_msg_hdr.h"
#include "cfe_core_api_base_msgids.h"

#define SENDER_APP_PERF_ID          92
#define CFE_MISSION_SENDER_APP_HK_TLM_TOPICID    0x84
#define SENDER_APP_HK_TLM_MID      CFE_PLATFORM_TLM_TOPICID_TO_MIDV(CFE_MISSION_SENDER_APP_HK_TLM_TOPICID)

typedef struct SENDER_APP_HkTlm_Payload
{
    uint8 batteryOccupancy;
    uint8 batteryHealth;
    uint32 counter;
    CFE_TIME_SysTime_t timestamp;
} SENDER_APP_HkTlm_Payload_t;

typedef struct
{
    CFE_MSG_TelemetryHeader_t TelemetryHeader; /**< \brief Telemetry header */
    SENDER_APP_HkTlm_Payload_t Payload;        /**< \brief Telemetry payload */
} SENDER_APP_HkTlm_t;

typedef struct
{
    uint32 RunStatus;
    uint32 counter;

    SENDER_APP_HkTlm_t HkTlm;
} SENDER_AppData_t;

void SENDER_AppMain(void);
int32 SENDER_init(void);
int32 SENDER_SendBatteryData(uint8 batteryOccupancy, uint8 batteryHealth);

#endif /* SENDER_APP_H_ */

```

Following that, *sender\_app.c* file should be as follows. Code is straight-forward, and it is expected from reader to understand easily.

```

#include "cfe.h"
#include "sender_app.h"

SENDER_AppData_t SENDER_AppData;

void SENDER_App_Main(void){
    CFE_Status_t status;

    SENDER_AppData.RunStatus = 1;
    SENDER_AppData.counter = 0;

    status = SENDER_init();

```



```

    if (status != CFE_SUCCESS)
    {
        SENDER_AppData.RunStatus = 0;
    }

    while(CFE_ES_RunLoop(&SENDER_AppData.RunStatus)){
        status = SENDER_SendBatteryData(55, 100);

        if(status != CFE_SUCCESS){
            OS_printf("SENDER: Error in sending battery data\n");
        }
        else{
            OS_printf("SENDER: Sent battery data by SENDER\n");
            SENDER_AppData.counter++;
        }

        OS_TaskDelay(2000);
    }
}

CFE_Status_t SENDER_init(void){
    CFE_Status_t status;

    status = CFE_EVS_Register(NULL, 0, 0);

    if(status != CFE_SUCCESS){
        OS_printf("SENDER: Error in registering for event services\n");
    }
    else{
        OS_printf("SENDER: Registered for event services\n");

        CFE_MSG_Init(CFE_MSG_PTR(SENDER_AppData.HkTlm.TelemetryHeader),
            CFE_SB_ValueToMsgId(SENDER_APP_HK_TLM_MID), sizeof(SENDER_AppData.HkTlm));
        OS_printf("SENDER: Initialized HK telemetry\n");
    }

    return status;
}

CFE_Status_t SENDER_SendBatteryData(uint8 batteryOccupancy, uint8 batteryHealth){
    CFE_Status_t status;

    SENDER_AppData.HkTlm.Payload.batteryOccupancy = batteryOccupancy;
    SENDER_AppData.HkTlm.Payload.batteryHealth = batteryHealth;
    SENDER_AppData.HkTlm.Payload.counter = SENDER_AppData.counter;
    SENDER_AppData.HkTlm.Payload.timestamp = CFE_TIME_GetTime();

    status = CFE_SB_TransmitMsg(CFE_MSG_PTR(SENDER_AppData.HkTlm.TelemetryHeader), true);

    return status;
}

```

Moving forward to engine app, please notice that the engine app must know the channel ID and payload structure to receive and process the message. This data is stored in *sender\_app.h* file. Now, it is of course possible to write this code in *engine\_app.h* file too, but let's stick to the write

once use everywhere principle and import this data to engine header file. The *engine\_app.h* file should be as follows.

```
#ifndef _ENGINE_APP_H_
#define _ENGINE_APP_H_

#include "cfe.h"
#include "cfe_msg_hdr.h"
#include "cfe_core_api_base_msgids.h"
#include "../sender_app/sender_app.h"

#define ENGINE_APP_PERF_ID          92
#define ENGINE_APP_PIPE_DEPTH      12

typedef struct
{
    uint32 RunStatus;
    uint32 counter;

    CFE_SB_PipeId_t CommandPipe;

    char    PipeName[CFE_MISSION_MAX_API_LEN];
    uint16 PipeDepth;
} ENGINE_AppData_t;

void ENGINE_AppMain(void);
int32 ENGINE_init(void);
int32 ENGINE_checkSenderAppTelemetry(void);

#endif /* ENGINE_APP_H_ */
```

Then, the *engine\_app.c* file should be as follows.

```
#include "cfe.h"
#include "engine_app.h"

ENGINE_AppData_t ENGINE_AppData;

void ENGINE_App_Main(void){
    CFE_Status_t status;

    status = ENGINE_init();

    if (status != CFE_SUCCESS)
    {
        ENGINE_AppData.RunStatus = 0;
    }

    ENGINE_AppData.RunStatus = 1;
    ENGINE_AppData.counter = 0;

    while(CFE_ES_RunLoop(&ENGINE_AppData.RunStatus)){
        ENGINE_checkSenderAppTelemetry();

        OS_TaskDelay(2000);
    }
}
```

```

    }
}

CFE_Status_t ENGINE_init(void){
    CFE_Status_t status;

    ENGINE_AppData.PipeDepth = ENGINE_APP_PIPE_DEPTH;
    strncpy(ENGINE_AppData.PipeName, "ENGINE_APP_HK_PIPE", sizeof(ENGINE_AppData.PipeName));
    ENGINE_AppData.PipeName[sizeof(ENGINE_AppData.PipeName) - 1] = 0;

    // create pipe and receive message

    status = CFE_EVS_Register(NULL, 0, 0);

    if(status != CFE_SUCCESS){
        OS_printf("ENGINE: Error in registering for event services\n");
    }
    else{
        OS_printf("ENGINE: Registered for event services\n");
    }

    status = CFE_SB_CreatePipe(&ENGINE_AppData.CommandPipe,
                              ENGINE_AppData.PipeDepth,
                              ENGINE_AppData.PipeName);

    if(status != CFE_SUCCESS){
        OS_printf("ENGINE: Error in creating pipe\n");
    }
    else{
        OS_printf("ENGINE: Created pipe\n");
    }

    // subscribe to sender app's telemetry

    status = CFE_SB_Subscribe(CFE_SB_ValueToMsgId(SENDER_APP_HK_TLM_MID), ENGINE_AppData.CommandPipe);

    if(status != CFE_SUCCESS){
        OS_printf("ENGINE: Error in subscribing to sender app's telemetry\n");
    }
    else{
        OS_printf("ENGINE: Subscribed to sender app's telemetry\n");
    }

    return status;
}

CFE_Status_t ENGINE_checkSenderAppTelemetry(void){
    CFE_Status_t status;
    CFE_SB_Buffer_t *SBBufPtr;

    // use CFE_SB_ReceiveBuffer

    status = CFE_SB_ReceiveBuffer(&SBBufPtr, ENGINE_AppData.CommandPipe, CFE_SB_PEND_FOREVER);

    if(status != CFE_SUCCESS){
        OS_printf("ENGINE: Error in receiving sender app's telemetry\n");
    }
}

```

```

}
else{
    // print battery occupancy and health
    OS_printf("-----\n");

    SENDER_APP_HkTlm_t *senderAppHkTlm = (SENDER_APP_HkTlm_t *)SBBufPtr;
    OS_printf("ENGINE: Battery occupancy: %d\n", senderAppHkTlm->Payload.batteryOccupancy);
    OS_printf("ENGINE: Battery health: %d\n", senderAppHkTlm->Payload.batteryHealth);
    OS_printf("ENGINE: Counter: %d\n", senderAppHkTlm->Payload.counter);
    OS_printf("ENGINE: Timestamp: %d\n", senderAppHkTlm->Payload.timestamp.Seconds);

    OS_printf("-----\n");
}

return status;
}

```

So, as a summary, both reader and writer apps need to know the payload structure and channel ID. Also, both apps need to register to the cFE event service (cFE\_ES) before sending or receiving any data.

The sender app needs to initialize the message structure using **CFE\_MSG\_Init** function. Then, it can send message to the channel using **CFE\_SB\_TransitMsg** function.

The receiver app first needs to create a pipe to read messages using **CFE\_SB\_CreatePipe** function. Then, it needs to subscribe to the channel that it wants to retrieve data from using **CFE\_SB\_Subscribe** function. Note that one app can subscribe to multiple channels, using only one pipe. Of course, app can also use multiple pipes. Then, app can retrieve messages using **CFE\_SB\_ReceiveBuffer** function.

There are multiple ways to perform internal communication, and this is only an example. For parameters of these functions, it is recommended for reader to check the documentation of NASA on CFS and try out new things.

### 3.3 Events in cFS

In cFS, events are used for debugging and logging purposes. There are four event types defined: debug, informational, error and critical. When a cFS app sends an event, cFE either logs it or sends it via an output port, as in the Figure 4.

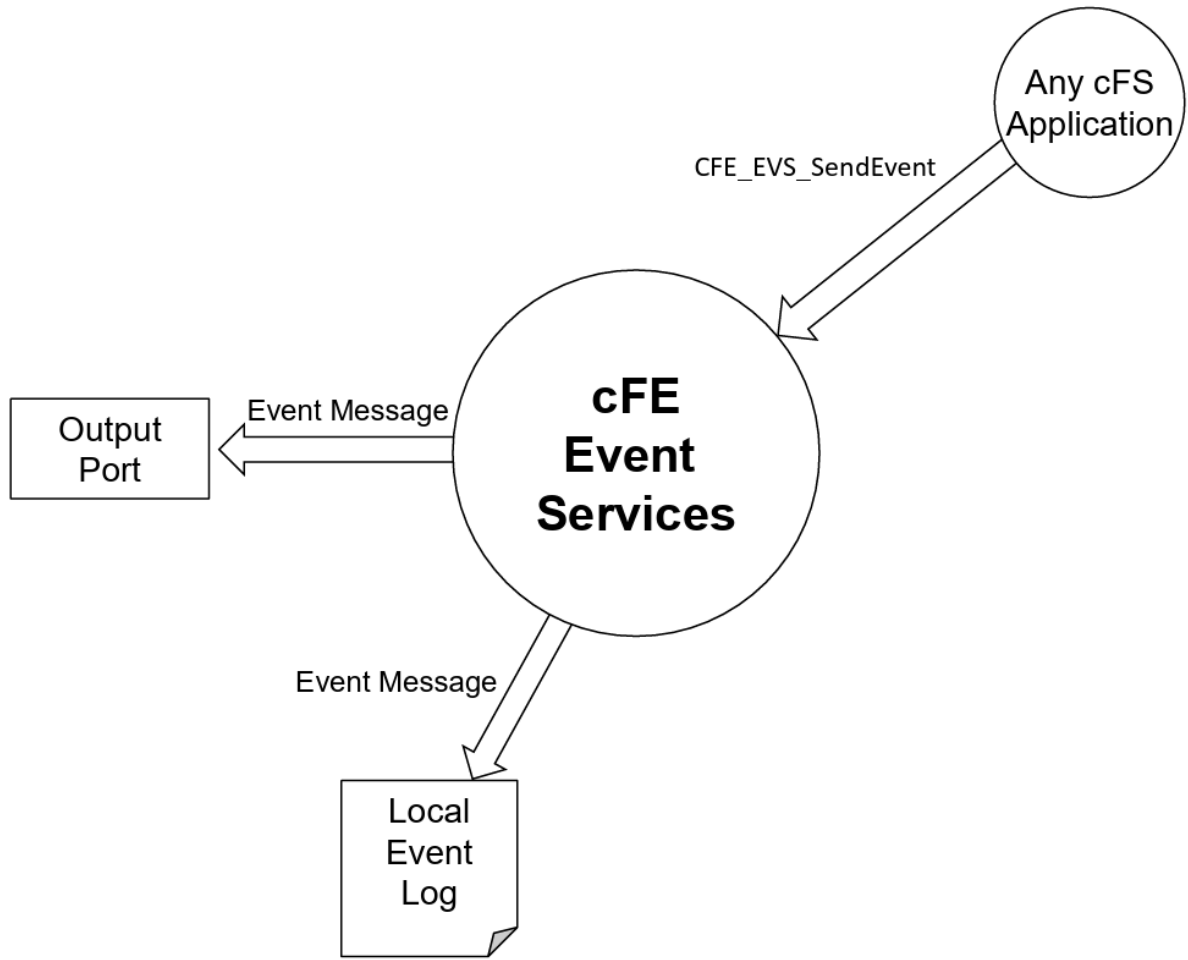


Figure 4: Events Flow Chart in cFS

Any app that is registered to cFE event service via **CFE\_EVS\_Register** can send events. Notice that events are normally not received by apps. For internal communication, use software bus and messages rather than events. Events are for logging purposes.

For an example, the sender app can be manipulated so it uses events for debugging. Notice that when it is implemented in Chapter 3.2, the classic print functions were used for debugging purposes. Now, create a *sender\_app\_eventids.h* file to define the IDs of logging types. Of course, it is possible to set ID=1 for all messages, but defining different IDs are more suitable. This file should be as follows.

```

#ifdef SENDER_APP_EVENTS_H
#define SENDER_APP_EVENTS_H

#define SENDER_APP_REGISTER_SUCCESS_EID 1
#define SENDER_APP_MESSAGE_ERROR_EID 2
#define SENDER_APP_MESSAGE_SUCCESS_EID 3

#endif /* SENDER_APP_EVENTS_H */

```

Then, the re-defined *sender\_app.c* file should be as follows.

```

#include "cfe.h"
#include "sender_app.h"
#include "sender_app_eventids.h"

SENDER_AppData_t SENDER_AppData;

void SENDER_App_Main(void){
    CFE_Status_t status;

    SENDER_AppData.RunStatus = 1;
    SENDER_AppData.counter = 0;

    status = SENDER_init();

    if (status != CFE_SUCCESS)
    {
        SENDER_AppData.RunStatus = 0;
    }

    while(CFE_ES_RunLoop(&SENDER_AppData.RunStatus)){
        status = SENDER_SendBatteryData(55, 100);

        if(status != CFE_SUCCESS){
            CFE_EVS_SendEvent(SENDER_APP_MESSAGE_ERROR_EID, CFE_EVS_EventType_ERROR,
                "SENDER: Error in sending battery data\n");
        }
        else{
            SENDER_AppData.counter++;
            CFE_EVS_SendEvent(SENDER_APP_MESSAGE_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
                "SENDER: Battery data sent successfully\n");
        }

        OS_TaskDelay(2000);
    }
}

CFE_Status_t SENDER_init(void){
    CFE_Status_t status;

    status = CFE_EVS_Register(NULL, 0, 0);

    if(status != CFE_SUCCESS){
        CFE_ES_WriteToSysLog("SENDER: Error in registering for event services\n");
    }
    else{
        CFE_EVS_SendEvent(SENDER_APP_REGISTER_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
            "SENDER:Successfully registered for event services\n");
        CFE_MSG_Init(CFE_MSG_PTR(SENDER_AppData.HkTlm.TelemetryHeader),
            CFE_SB_ValueToMsgId(SENDER_APP_HK_TLM_MID), sizeof(SENDER_AppData.HkTlm));
    }

    return status;
}

CFE_Status_t SENDER_SendBatteryData(uint8 batteryOccupancy, uint8 batteryHealth){
    CFE_Status_t status;

```

```

SENDER_AppData.HkTlm.Payload.batteryOccupancy = batteryOccupancy;
SENDER_AppData.HkTlm.Payload.batteryHealth = batteryHealth;
SENDER_AppData.HkTlm.Payload.counter = SENDER_AppData.counter;
SENDER_AppData.HkTlm.Payload.timestamp = CFE_TIME_GetTime();

status = CFE_SB_TransmitMsg(CFE_MSG_PTR(SENDER_AppData.HkTlm.TelemetryHeader), true);

return status;
}

```

Reader can see that instead of using printf functions, the app establishes the event service to send logs to command line. In later chapters, saving these logs to a file will be explained.

### 3.4 Tables in cFS

In cFS, tables are data storage units, a.k.a. databases. They store parameters that may change the behavior of flight software. These parameters can be anything from control gains to operational modes. The reason that cFS recommends storing values in tables rather than defining constants is that tables can easily be manipulated from ground station. However, code manipulations are relatively harder since all of the compiled code needs to be uploaded to S/C (or code needs to be compiled on-board, which is still a hard process). Tables are binary files, thus when something changes the ground station only needs to upload the table. Tables are defined to be app-specific. Which means that it is not common for one table to be shared by two apps.

Now, let's start with initiating the table structure. Starting with *sender\_app\_tbl.h* file. This file should be as follows. We will define a table structure, where the table will store two integers.

```

#include "cfe.h"
#include "sender_app.h"

#define SENDER_APP_TABLE_FILE "/cf/sender_app_tbl.tbl"
typedef struct
{
    uint32 Int1;
    uint32 Int2;
} SENDER_APP_ExampleTable_t;

```

Following that, *sender\_app\_tbl.c* file should be as follows. Notice that we need to introduce these variables in our code. For now, 55 and 66 are chosen. Also notice that the name of binary file **sender\_app\_tbl.tbl** is given as parameter.

```

#include "cfe_tbl_filedef.h"
#include "sender_app_table.h"

SENDER_APP_ExampleTable_t ExampleTable = { 55, 66 };

CFE_TBL_FILEDEF(ExampleTable, SENDER_APP.ExampleTable, "Table Utility Test Table", sender_app_tbl.tbl)

```

Then, *sender\_app.h* can be defined as follows. Notice that the only changes from last example is the definition of **TblHandle** definition, alongside with table reading function definitions.



```

#ifdef _SENDER_APP_H_
#define _SENDER_APP_H_

#include "cfe.h"
#include "cfe_msg_hdr.h"
#include "cfe_core_api_base_msgids.h"

#include "sender_app_eventids.h"

#define SENDER_APP_PERF_ID          92
#define CFE_MISSION_SENDER_APP_HK_TLM_TOPICID    0x84
#define SENDER_APP_HK_TLM_MID      CFE_PLATFORM_TLM_TOPICID_TO_MIDV(CFE_MISSION_SENDER_APP_HK_TLM_TOPICID)

typedef struct SENDER_APP_HkTlm_Payload
{
    uint8 batteryOccupancy;
    uint8 batteryHealth;
    uint32 counter;
    CFE_TIME_SysTime_t timestamp;
} SENDER_APP_HkTlm_Payload_t;

typedef struct
{
    CFE_MSG_TelemetryHeader_t TelemetryHeader; /**< \brief Telemetry header */
    SENDER_APP_HkTlm_Payload_t Payload;        /**< \brief Telemetry payload */
} SENDER_APP_HkTlm_t;

typedef struct
{
    uint32 RunStatus;
    uint32 counter;

    SENDER_APP_HkTlm_t HkTlm;
    CFE_TBL_Handle_t TblHandle;
} SENDER_AppData_t;

void SENDER_AppMain(void);
int32 SENDER_init(void);
int32 SENDER_SendBatteryData(uint8 batteryOccupancy, uint8 batteryHealth);
int32 SENDER_ReadTableContent(void);

#endif /* SENDER_APP_H_ */

```

Lastly, the *sender\_app.c* file must be defined as follows.

```

#include "cfe.h"
#include "sender_app.h"
#include "sender_app_eventids.h"
#include "sender_app_table.h"

SENDER_AppData_t SENDER_AppData;
SENDER_APP_ExampleTable_t *SENDER_TablePtr;

void SENDER_App_Main(void){

```

```

CFE_Status_t status;

SENDER_AppData.RunStatus = 1;
SENDER_AppData.counter = 0;

status = SENDER_init();

if (status != CFE_SUCCESS)
{
    SENDER_AppData.RunStatus = 0;
}

while(CFE_ES_RunLoop(&SENDER_AppData.RunStatus)){
    status = SENDER_SendBatteryData(55, 100);

    if(status != CFE_SUCCESS){
        CFE_EVS_SendEvent(SENDER_APP_MESSAGE_ERROR_EID, CFE_EVS_EventType_ERROR,
            "SENDER: Error in sending battery data\n");
    }
    else{
        SENDER_AppData.counter++;
        CFE_EVS_SendEvent(SENDER_APP_MESSAGE_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
            "SENDER: Battery data sent successfully\n");

        SENDER_ReadTableContent();
    }

    OS_TaskDelay(2000);
}

CFE_Status_t SENDER_init(void){
    CFE_Status_t status;

    status = CFE_EVS_Register(NULL, 0, 0);

    if(status != CFE_SUCCESS){
        CFE_ES_WriteToSysLog("SENDER: Error in registering for event services\n");
    }
    else{
        CFE_EVS_SendEvent(SENDER_APP_REGISTER_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
            "SENDER: Successfully registered for event services\n");
        CFE_MSG_Init(CFE_MSG_PTR(SENDER_AppData.HkTlm.TelemetryHeader),
            CFE_SB_ValueToMsgId(SENDER_APP_HK_TLM_MID), sizeof(SENDER_AppData.HkTlm));
    }

    status = CFE_TBL_Register(&SENDER_AppData.TblHandle, "ExampleTable", sizeof(SENDER_APP_ExampleTable_t)
    if (status != CFE_SUCCESS) {
        CFE_EVS_SendEvent(SENDER_APP_TABLE_ERROR_EID, CFE_EVS_EventType_ERROR,
            "SENDER: Error in registering table\n");
        return status;
    }else {
        CFE_EVS_SendEvent(SENDER_APP_TABLE_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
            "SENDER: Successfully registered table\n");
    }
}

```

```

    return status;
}

CFE_Status_t SENDER_SendBatteryData(uint8 batteryOccupancy, uint8 batteryHealth){
    CFE_Status_t status;

    SENDER_AppData.HkTlm.Payload.batteryOccupancy = batteryOccupancy;
    SENDER_AppData.HkTlm.Payload.batteryHealth = batteryHealth;
    SENDER_AppData.HkTlm.Payload.counter = SENDER_AppData.counter;
    SENDER_AppData.HkTlm.Payload.timestamp = CFE_TIME_GetTime();

    status = CFE_SB_TransmitMsg(CFE_MSG_PTR(SENDER_AppData.HkTlm.TelemetryHeader), true);

    return status;
}

CFE_Status_t SENDER_ReadTableContent(void){
    CFE_Status_t status;

    status = CFE_TBL_Load(SENDER_AppData.TblHandle, CFE_TBL_SRC_FILE, SENDER_APP_TABLE_FILE);

    if (status != CFE_SUCCESS) {
        CFE_EVS_SendEvent(SENDER_APP_TABLE_ERROR_EID, CFE_EVS_EventType_ERROR,
            "SENDER: Error in loading table\n");
        return status;
    }else {
        CFE_EVS_SendEvent(SENDER_APP_TABLE_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
            "SENDER: Successfully loaded table\n");

        // read table content and print it
        status = CFE_TBL_Manage(SENDER_AppData.TblHandle);

        if (status != CFE_SUCCESS) {
            CFE_EVS_SendEvent(SENDER_APP_TABLE_ERROR_EID, CFE_EVS_EventType_ERROR,
                "SENDER: Error in managing table\n");
            return status;
        }else {
            CFE_EVS_SendEvent(SENDER_APP_TABLE_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
                "SENDER: Successfully managed table\n");
        }

        status = CFE_TBL_GetAddress((void*)&SENDER_TablePtr, SENDER_AppData.TblHandle);

        if (status != CFE_TBL_INFO_UPDATED) {
            CFE_EVS_SendEvent(SENDER_APP_TABLE_ERROR_EID, CFE_EVS_EventType_ERROR,
                "SENDER: Error in getting table address\n, status: %d\n", status);
            return status;
        }else {
            CFE_EVS_SendEvent(SENDER_APP_TABLE_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
                "SENDER: Successfully got table address\n");

            CFE_EVS_SendEvent(SENDER_APP_TABLE_CONTENT_EID, CFE_EVS_EventType_INFORMATION,
                "SENDER: Table content: %d, %d\n", SENDER_TablePtr->Int1, SENDER_TablePtr->Int2);
        }

        status = CFE_TBL_ReleaseAddress(SENDER_AppData.TblHandle);
    }
}

```

```

    if (status != CFE_SUCCESS) {
        CFE_EVS_SendEvent(SENDER_APP_TABLE_ERROR_EID, CFE_EVS_EventType_ERROR,
            "SENDER: Error in releasing table address\n, status: %d\n", status);
        return status;
    }else {
        CFE_EVS_SendEvent(SENDER_APP_TABLE_SUCCESS_EID, CFE_EVS_EventType_INFORMATION,
            "SENDER: Successfully released table address\n");
    }
}

return status;
}

```

Now, as last thing, add the following line to CMakeLists.txt file of sender app.

```
add_cfe_tables(sender_app sender_app_table.c)
```

Now, let's explain what this code does. It simply initializes the table with initial values, 55 and 66 for our case. It then saves this data to a file, we named it *sender\_app\_tbl.tbl*. After running the make commands, reader can find this file at *cfs/build/exe/cpu1/cf*. The tables of the apps can be found here. After running the cFS code, table content will be printed to screen.

Also, recall what we have mentioned in the beginning of the sub-chapter. These tables needs to be updated from ground station, obviously. Now, the tables are read from *cfs/build/exe/cpu1/cf*. If one changes the initial variables *sender\_app\_tbl.c*, one can build *sender\_app\_tbl.tbl* files at that folder. After that, the tbl file can be moved. So, for example, a tbl file needs to be built in ground station. Then, this file can be sent to S/C via telemetry.

An example for this can be seen in Figure 5. The initial values were 55 and 66, then a pre-built table file with values 42 and 43 were uploaded to *cfs/build/exe/cpu1/cf*.

```

Hello, world!, Counter: 18
EVS Port1 1980-012-14:03:37.71080 66/1/SENDER_APP 3: SENDER: Battery data sent successfully

EVS Port1 1980-012-14:03:37.71087 66/1/CFE_TBL 35: Successfully loaded 'SENDER_APP.ExampleTable' from '/cf/sender_app_tbl.tbl'
EVS Port1 1980-012-14:03:37.71089 66/1/SENDER_APP 5: SENDER: Successfully loaded table

EVS Port1 1980-012-14:03:37.71090 66/1/SENDER_APP 5: SENDER: Successfully managed table

EVS Port1 1980-012-14:03:37.71092 66/1/SENDER_APP 5: SENDER: Successfully got table address

EVS Port1 1980-012-14:03:37.71093 66/1/SENDER_APP 7: SENDER: Table content: 55, 66

EVS Port1 1980-012-14:03:37.71093 66/1/SENDER_APP 5: SENDER: Successfully released table address

EVS Port1 1980-012-14:03:37.79000 66/1/CFE_SB 17: Msg Limit Err,MsgId 0x884,pipe ENGINE_APP_HK_PIPE,sender CI_LAB_APP
SAMPLE_APP_Main: Working
Hello, world!, Counter: 19
EVS Port1 1980-012-14:03:38.74000 66/1/CFE_SB 17: Msg Limit Err,MsgId 0x884,pipe ENGINE_APP_HK_PIPE,sender CI_LAB_APP
SAMPLE_APP_Main: Working
-----
ENGINE: Battery occupancy: 55
ENGINE: Battery health: 100
ENGINE: Counter: 6
ENGINE: Timestamp: 1001011
-----
Hello, world!, Counter: 20
EVS Port1 1980-012-14:03:39.71104 66/1/CFE_SB 17: Msg Limit Err,MsgId 0x884,pipe ENGINE_APP_HK_PIPE,sender SENDER_APP
EVS Port1 1980-012-14:03:39.71109 66/1/SENDER_APP 3: SENDER: Battery data sent successfully

EVS Port1 1980-012-14:03:39.71118 66/1/CFE_TBL 35: Successfully loaded 'SENDER_APP.ExampleTable' from '/cf/sender_app_tbl.tbl'
EVS Port1 1980-012-14:03:39.71119 66/1/SENDER_APP 5: SENDER: Successfully loaded table

EVS Port1 1980-012-14:03:39.71121 66/1/SENDER_APP 5: SENDER: Successfully managed table

EVS Port1 1980-012-14:03:39.71122 66/1/SENDER_APP 5: SENDER: Successfully got table address

EVS Port1 1980-012-14:03:39.71124 66/1/SENDER_APP 7: SENDER: Table content: 42, 43

EVS Port1 1980-012-14:03:39.71125 66/1/SENDER_APP 5: SENDER: Successfully released table address

SAMPLE_APP_Main: Working
Hello, world!, Counter: 21
SAMPLE_APP_Main: Working

```

Figure 5: Expected Table Results

## 4 Built-In Applications

Now, since that the basics of cFS are completed, the built-in applications of cFS can be used & manipulated. There are important applications, which are listed in Figure 6.

Application	Function
<a href="#">CFDP</a>	Transfers/receives file data to/from the ground
<a href="#">Checksum</a>	Performs data integrity checking of memory, tables and files
<a href="#">Command Ingest Lab</a>	Accepts CCSDS telecommand packets over a UDP/IP port
<a href="#">Data Storage</a>	Records housekeeping, engineering and science data onboard for downlink
<a href="#">File Manager</a>	Interfaces to the ground for managing files
<a href="#">Housekeeping</a>	Collects and re-packages telemetry from other applications.
<a href="#">Health and Safety</a>	Ensures critical tasks check-in, services watchdog, detects CPU hogging, calculates CPU utilization
<a href="#">Limit Checker</a>	Provides the capability to monitor values and take action when exceed threshold
<a href="#">Memory Dwell</a>	Allows ground to telemeter the contents of memory locations. Useful for debugging
<a href="#">Memory Manager</a>	Provides the ability to load and dump memory
<a href="#">Software Bus Network</a>	Passes Software Bus messages over various “plug-in” network protocols
<a href="#">Scheduler</a>	Schedules onboard activities (e.g. HK requests)
<a href="#">Scheduler Lab</a>	Simple activity scheduler with a one second resolution
<a href="#">Stored Command</a>	Onboard Commands Sequencer (absolute and relative)
<a href="#">Stored Command Absolute</a>	Allows concurrent processing of up to 5 (configurable) absolute time sequences
<a href="#">Telemetry Output Lab</a>	Sends CCSDS telemetry packets over a UDP/IP port

Figure 6: Built In App List by cFS

Apps will be explained one by one. There is no order in explanation, it was selected with authors judgment.

## 4.1 Scheduler Application

For this test, it is recommended to make a fresh start. Feel free to clone a new cFS project, and work on it. Now, go ahead and create an app called **power\_app**. Aim here is to simulate a power systems app, specifically power housekeeping data.

Recall that this section is a sub-project for scheduler app. So, let’s talk about the use case. In housekeeping applications, apps do not just call a function that prints telemetry with a frequency. In general, it is never recommended for an app to call functions in an interval. Instead, apps define a set of commands, and create a software bus pipeline just like in Chapter 3.2. Then, other apps send these commands to this pipeline when they need this specific app to get something done.

For the set-up of this section, the power app will receive a housekeeping request from the scheduler app. Also, the scheduler app will send these requests with a pre-defined frequency. Now, we just said it is not recommended to call functions from apps in interval, but the case with scheduler is different. The app is designed such that it will do something in specific frequencies.

Now, we will not go over on how to create a pipeline and subscribe to it, it is already mentioned. However, as a difference, the config files will be organized in two files. The *config.h* file

should be as follows.

```
#ifndef POWER_APP_CONFIG_H
#define POWER_APP_CONFIG_H

#include "power_msgids.h"

#define HAVE_POWER_APP

#define POWER_APP_PIPE_DEPTH 12
#define POWER_APP_PERF_ID 91

#define POWER_APP_CR_PIPE_ERR_EID 1
#define POWER_APP_SUB_HK_ERR_EID 2
#define POWER_APP_SUB_ERR_EID 3
#define POWER_APP_PIPE_ERR_EID 4

#endif /* POWER_APP_CONFIG_H */
```

Also, *power\_msgids.h* needs to be as follows. Notice that this is a new file structure, and it is more organized.

```
#ifndef POWER_MSGIDS_H
#define POWER_MSGIDS_H

#include "cfe.h"
#include "cfe_msg.h"
#include "cfe_core_api_base_msgids.h"

#define CFE_MISSION_POWER_APP_HK_TLM_TOPICID 0x84
#define CFE_MISSION_POWER_APP_SEND_HK_TOPICID 0x84
#define CFE_MISSION_POWER_APP_CMD_TOPICID 0x85

#define POWER_APP_HK_TLM_MID CFE_PLATFORM_TLM_TOPICID_TO_MIDV(CFE_MISSION_POWER_APP_HK_TLM_TOPICID)
#define POWER_APP_SEND_HK_MID CFE_PLATFORM_CMD_TOPICID_TO_MIDV(CFE_MISSION_POWER_APP_SEND_HK_TOPICID)
#define POWER_APP_CMD_MID CFE_PLATFORM_CMD_TOPICID_TO_MIDV(CFE_MISSION_POWER_APP_CMD_TOPICID)

#endif /* POWER_MSGIDS_H */
```

Notice that you need to include *config.h* file in *power\_app.h* file. The *power\_app.c* file should be as follows. The header file of this app is left as an exercise for the reader :).

```
#include "cfe.h"
#include "power_app.h"

POWER_AppData_t POWER_AppData;
CFE_SB_Buffer_t *SBBufPtr;

void POWER_AppMain(void){
    CFE_Status_t status;

    status = POWER_APP_Init();

    if (status != CFE_SUCCESS){
        POWER_AppData.RunStatus = CFE_ES_RunStatus_APP_ERROR;
```



```

}

while(CFE_ES_RunLoop(&POWER_AppData.RunStatus)){

    status = CFE_SB_ReceiveBuffer(&SBBufPtr, POWER_AppData.CommandPipe, CFE_SB_PEND_FOREVER);

    if (status == CFE_SUCCESS){
        OS_printf("POWER BUFFER RECEIVED\n");
    }else{
        CFE_EVS_SendEvent(POWER_APP_PIPE_ERR_EID, CFE_EVS_EventType_ERROR, "POWER APP: SB ReceiveBufferError");

        POWER_AppData.RunStatus = CFE_ES_RunStatus_APP_ERROR;
    }

    OS_TaskDelay(1000);
}

CFE_ES_ExitApp(POWER_AppData.RunStatus);
}

CFE_Status_t POWER_APP_Init(void)
{
    CFE_Status_t status;

    /* Zero out the global data structure */
    memset(&POWER_AppData, 0, sizeof(POWER_AppData));

    POWER_AppData.RunStatus = CFE_ES_RunStatus_APP_RUN;

    /*
    ** Initialize app configuration data
    */
    POWER_AppData.PipeDepth = POWER_APP_PIPE_DEPTH;

    strncpy(POWER_AppData.PipeName, "POWER_APP_CMD_PIPE", sizeof(POWER_AppData.PipeName));
    POWER_AppData.PipeName[sizeof(POWER_AppData.PipeName) - 1] = 0;

    /*
    ** Register the events
    */
    status = CFE_EVS_Register(NULL, 0, CFE_EVS_EventFilter_BINARY);
    if (status != CFE_SUCCESS)
    {
        CFE_ES_WriteToSysLog("POWER APP: Error Registering Events, RC = 0x%08lX\n", (unsigned long)status);
    }
    else
    {
        /*
        ** Initialize housekeeping packet (clear user data area).
        */
        CFE_MSG_Init(CFE_MSG_PTR(POWER_AppData.HkTlm.TelemetryHeader), CFE_SB_ValueToMsgId(POWER_APP_HK_TLM),
                    sizeof(POWER_AppData.HkTlm));

        /*
        ** Create Software Bus message pipe.
        */

```

```

    status = CFE_SB_CreatePipe(&POWER_AppData.CommandPipe, POWER_AppData.PipeDepth, POWER_AppData.PipeDepth);
    if (status != CFE_SUCCESS)
    {
        CFE_EVS_SendEvent(POWER_APP_CR_PIPE_ERR_EID, CFE_EVS_EventType_ERROR,
            "POWER APP: Error creating SB Command Pipe, RC = 0x%08lX", (unsigned long)status);
    }
}

if (status == CFE_SUCCESS)
{
    /*
    ** Subscribe to Housekeeping request commands
    */
    status = CFE_SB_Subscribe(CFE_SB_ValueToMsgId(POWER_APP_SEND_HK_MID), POWER_AppData.CommandPipe);
    if (status != CFE_SUCCESS)
    {
        CFE_EVS_SendEvent(POWER_APP_SUB_HK_ERR_EID, CFE_EVS_EventType_ERROR,
            "POWER APP: Error Subscribing to HK request, RC = 0x%08lX", (unsigned long)status);
    }
}

return status;
}

```

Now, we need to manipulate **sch\_lab.app** to send commands to housekeeping pipe of power\_app. sch\_lab is already installed in cFS distributions. Go to main folder and open *CMakeLists.txt* file. In Line 8, add power\_app to apps list.

Following that, go to *fsw/tables/sch\_lab\_table.c*. In ifdef sections, add following code.

```

#ifdef HAVE_POWER_APP
#include "power_msgids.h"
#endif

```

Then, in section below, in table definition, add following code.

```

#ifdef HAVE_POWER_APP
    {CFE_SB_MSGID_WRAP_VALUE(POWER_APP_SEND_HK_MID), 99, 0},
#endif

```

When run this code, reader will see in console that power app receives a buffer every second. The number 99 is the tick number, in this case.

## 4.2 Data Storage