Mix’n’Match Device I/O Transports for Evolving Closed-Loop FSW Testbed Fidelity

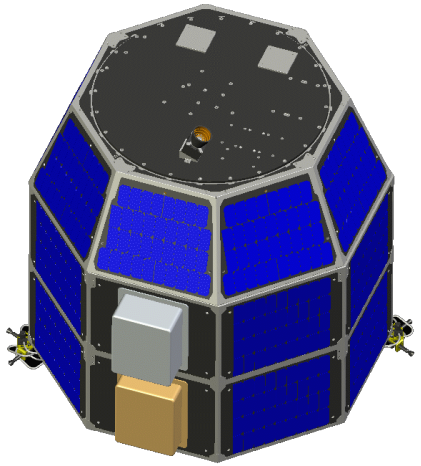
Douglas Forman
Millennium Engineering and Integration (MEI)
Craig Pires
NASA Ames Research Center (ARC)
Scott Christa
Aerospace Computing Inc (ACI)

Overview

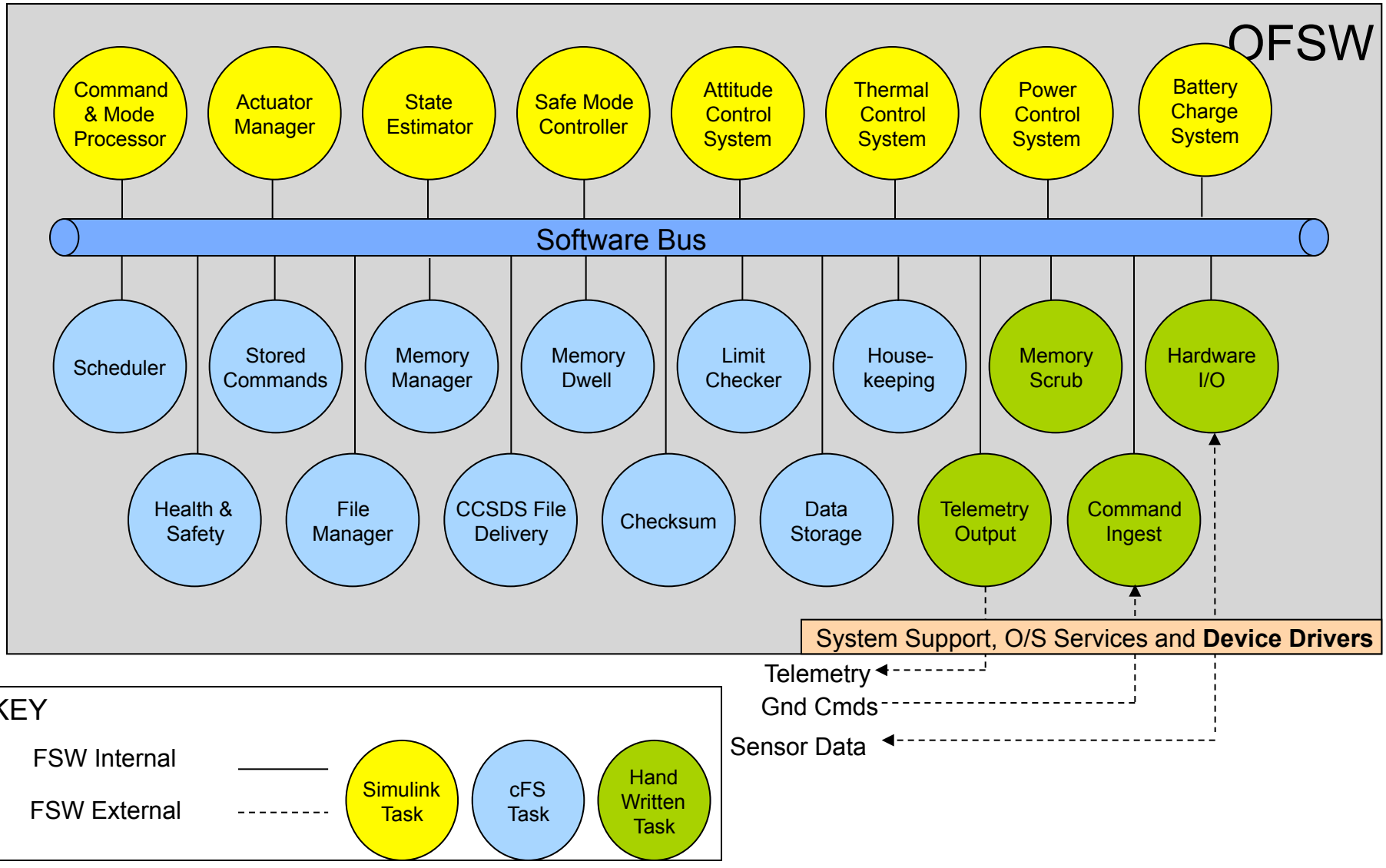
- Background
- Flight Software Architecture
- Closed Loop Testbed Systems
- Testbed S/W Implementation Goals
- Hardware I/O Module Implementation
  - With Non-portable Device Driver API
  - With Posix Device Driver API
- Posix I/O Driver Implementation

Background

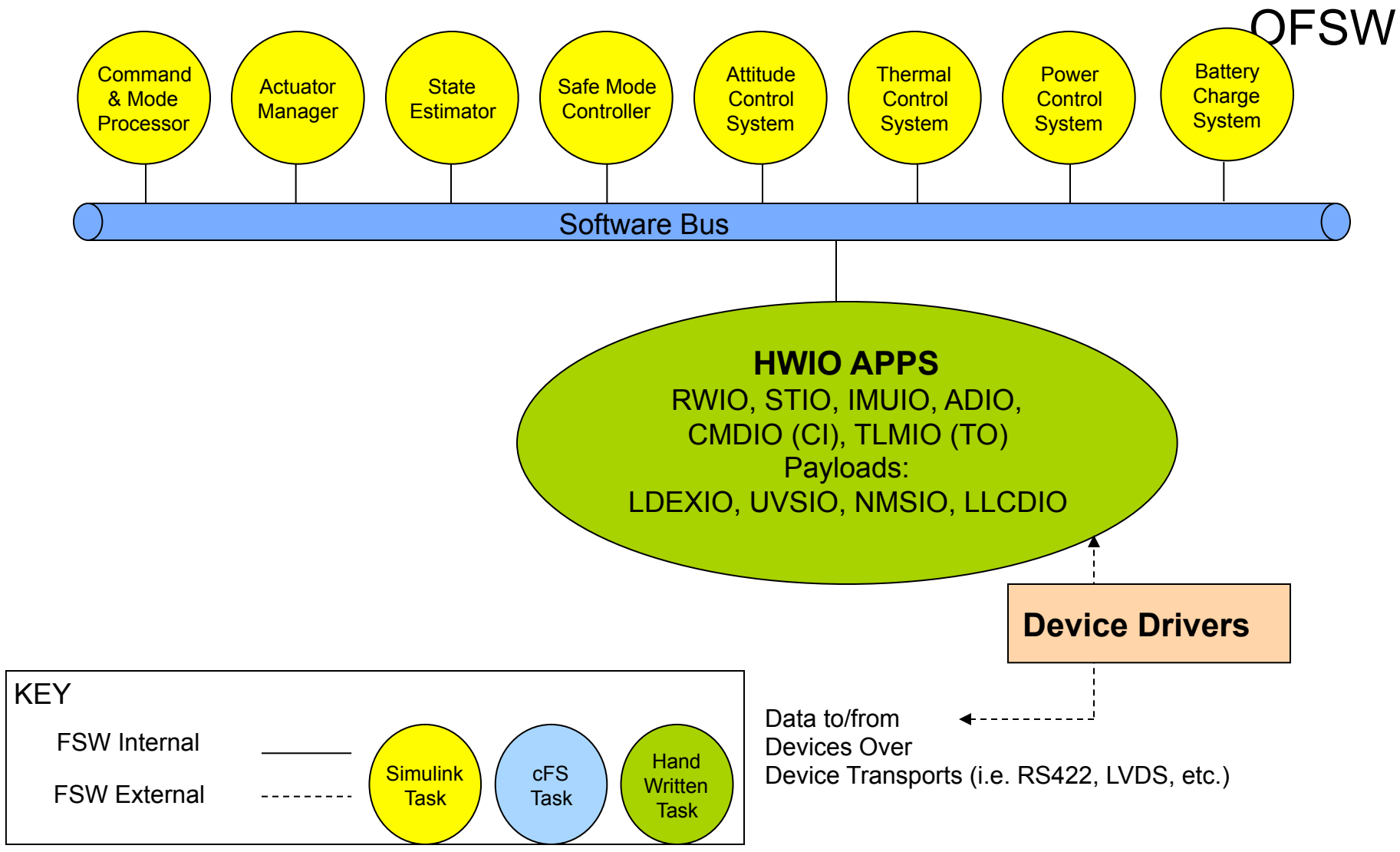
- Small Spacecraft Investigation
  - Modular CommonBus Spacecraft
- Hover Test Vehicle (HTV) Development
- Lunar Atmosphere and Dust Environment Experiment (LADEE)
  - Joint ARC/GSFC Mission
  - Lunar Orbiter, Launch 2013



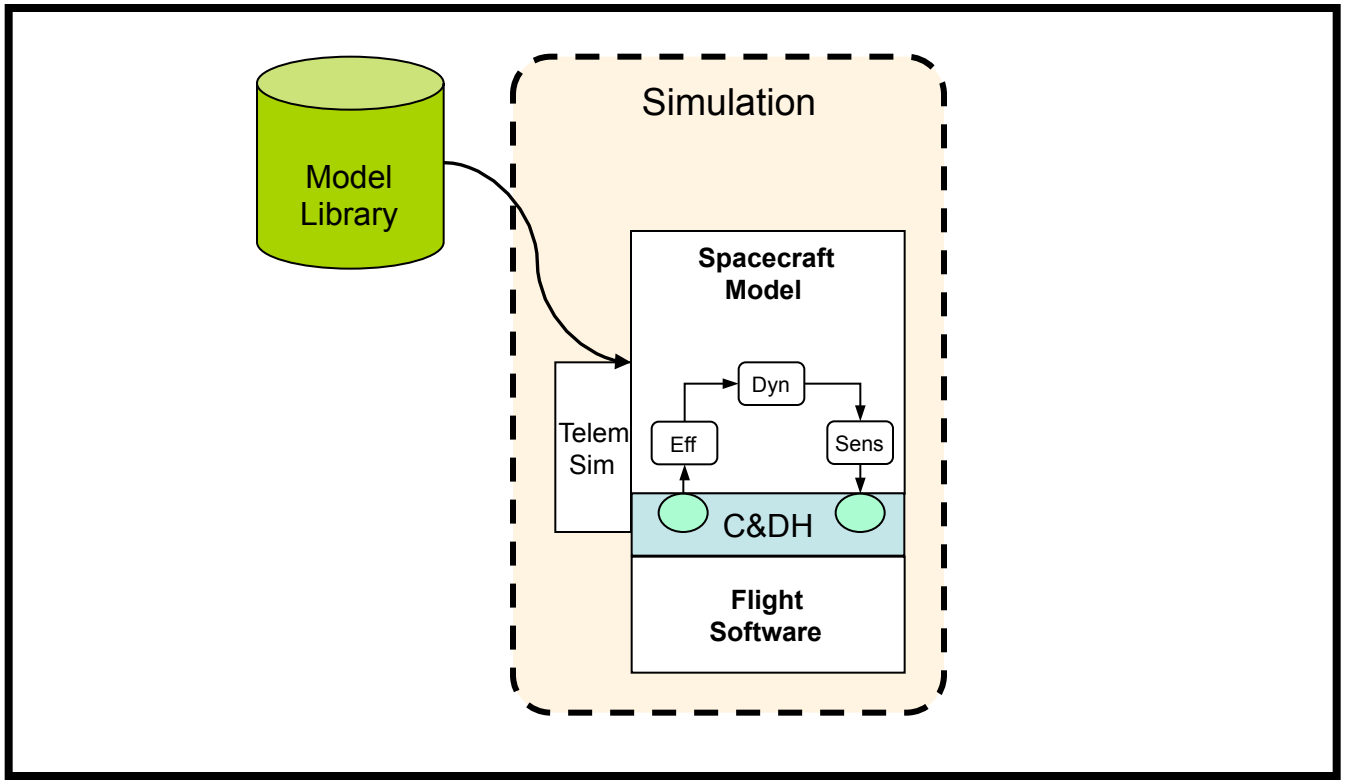
“Loosely-Coupled” FSW Architecture



HWIO Apps in FSW Architecture

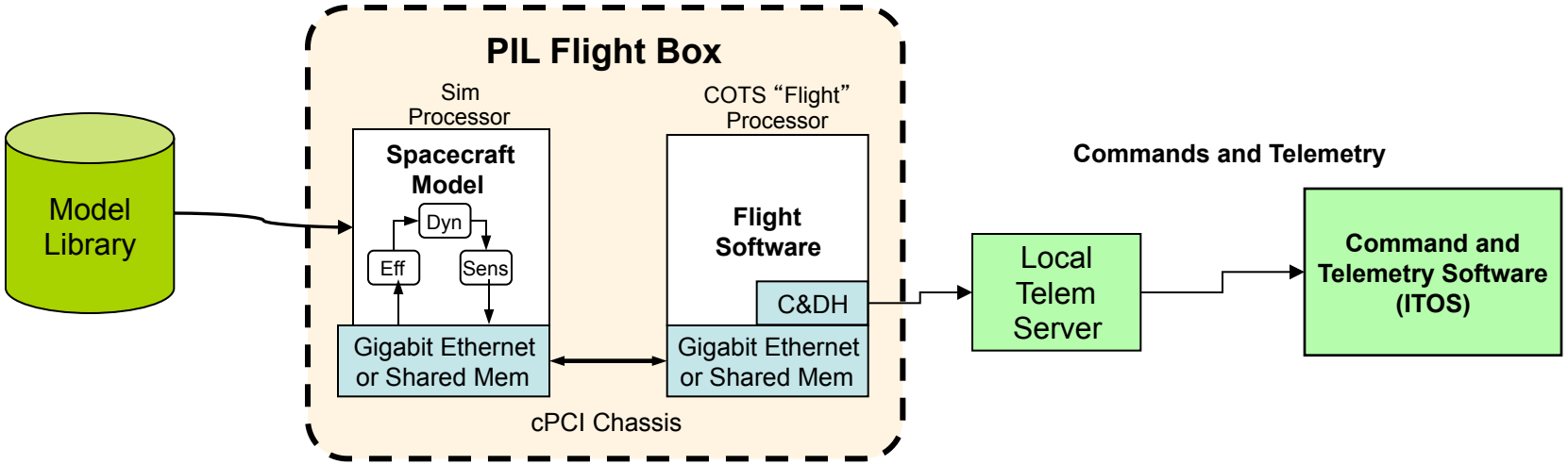


Workstation Simulation



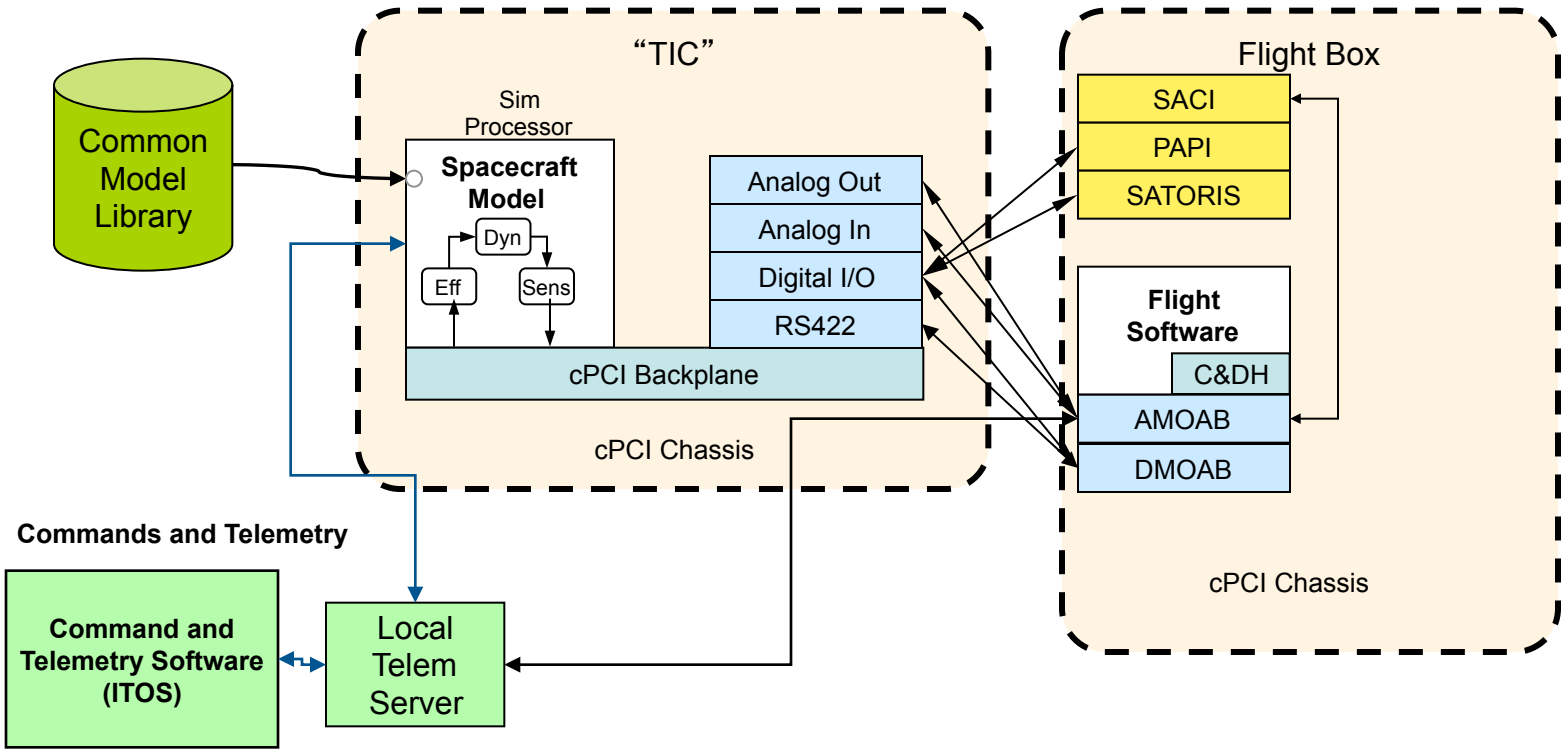
- Simulink Only (No Autocode)
- Developed early, maintained throughout development
- Algorithm Development
- Requirements Analysis

Processor-in-the-Loop Simulation



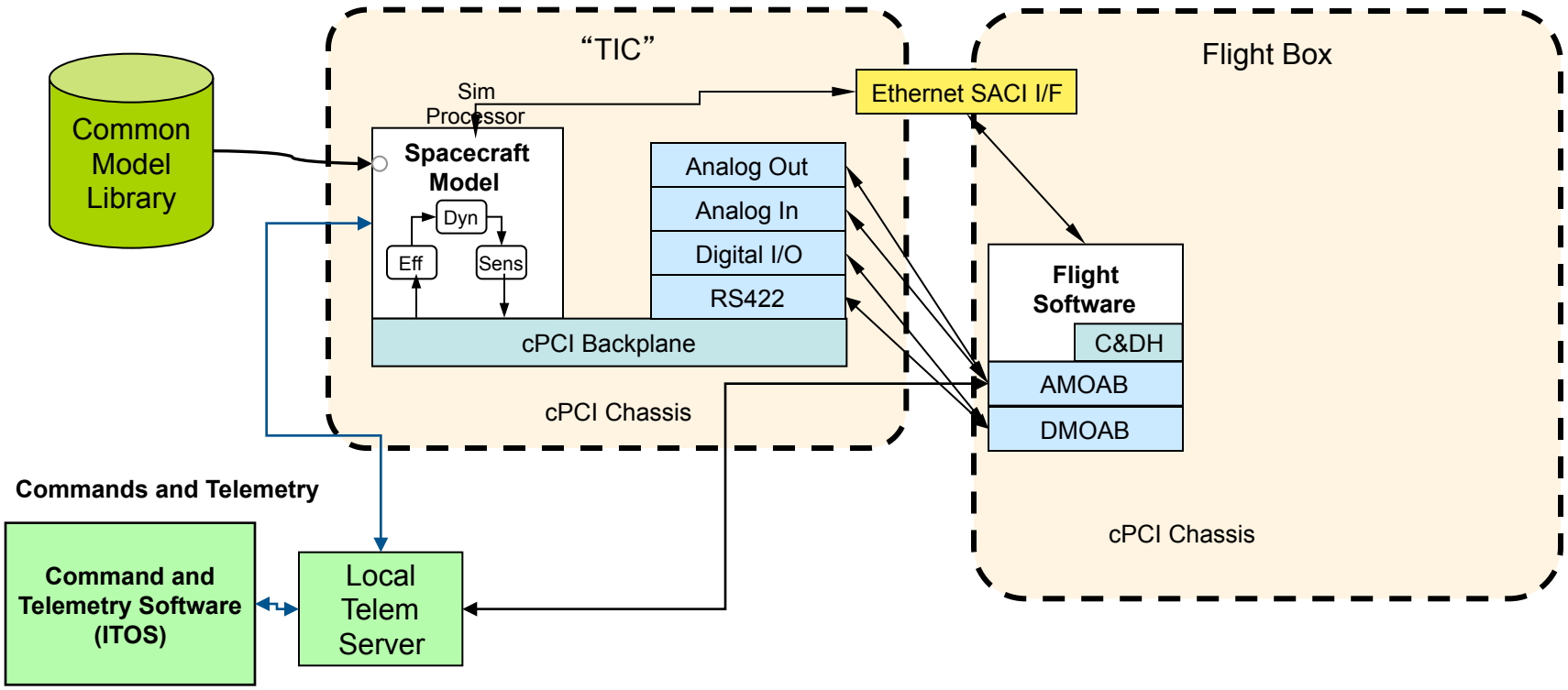
- Models autocoded and running on RT processors
- Inexpensive "flight-like" processor
- Tests autocoding process & integration with C&DH software
- Integration with Telemetry Software allows early development/testing of downlink
- Can be used for initial code size and resource utilization analysis

Hardware-in-the-Loop Simulation



- Flight code runs on Flight Avionics EDU
- Provides testing of FSW with Avionics I/O
- Definitive answers on resource utilization
- Highest fidelity simulations for verification/validation

PHIL Mix’ n’ Match Example



- Only DMOAB and AMOAB RS422 Available
  - No Power Boards in Flight Box
- Use Ethernet for other H/W interfaces
- Device Drivers Re-configured at runtime initialization
- Application Code unchanged

Realtime Testbed S/W Implementation Goals

- Flight Software Algorithms think they are flying
- Early/Frequent Integration and closed loop test of all Modules
  - Can be a challenge before H/W available
  - Need Mix’n’Match transport evolution in Test Beds
- Application code (including HWIO)
  - same in all targets (PIL, HIL or PHIL mix)
- Sim S/W infrastructure same as FSW
  - Simulink, cFE

Example Non-Portable FSW HWIO App

```
ST_Main() {
    while(AppIsRunning) {
#if (TARGET == ONE_PIL)
        read_shared_mem(&st_in, timeout);
#elseif (TARGET == TWO_PIL)
        read_socket(&st_in, timeout);
#else /* HIL and flight */
        /* VxWorks Msg Queue updated in ISR */
        read_msg_queue(&st_in, timeout);
#endif
         parse_st_data(&st_swbus_msg, &st_in);
        sb_send_msg(st_swbus_msg);
    }
}
```

Posix I/O API Features

- Portable Operating System Interface for Unix Input/Output Application Programmer Interface
  - Apps open() named devices to get File Descriptor
    - open("/dev/reactionWheel",…) returns File Descriptor (fd)
  - Apps access device through File Descriptor
    - read(fd,…), write(fd,…), ioctl(fd,…), close(fd,…)
  - Portable method for wake-up on interrupts
    - select(fd, …)
  - Support for serial and block devices
  - Supported in vxWorks 6.x

Posix I/O API Advantages for FSW and Sim

- Application Code can remain **exactly** the same whether emulated or real devices are used.
  - Only the driver implementation change
  - Device Emulation easy to implement with actual files, UDP packets, Shared Memory, etc.
  - Allows inexpensive simulation targets that exercise high percentage of code.
  - Allows same application binaries used in all targets
    - Requires same OS and Processor Class (i.e vxWorks, PPC)
- Allows Mix’n’Match evolution of devices
  - Drivers can be re-loaded at runtime
- Protocol code can reside in HWIO app
  - Ex. CCSDS, SLP code tested on all targets

FSW HWIO App with Posix I/O API

```
fd = open(“/st”, O_READ_ONLY, 0);
fdSet = addToFdSet(fd);
…
while (appRunning() ) {
  select(fdSet, …, timeoutVal);
…
if (FD_ISSET(fd)) {
        read(fd, msgBuff, sizeof(msgBuff));
        parseDeviceMsg(msgBuff, &swbusMsg);
        publishMsg(swbusMsg,sizeof(swbusMsg));
    }
}
```

Simulation HWIO App with Posix I/O API

```
fd = open("/st", O_READ_ONLY, 0);

while (simRunning) {
  rcvMsg(&swbusMsg, sizeof(swbusMsg));
  createDevMsg(&swbusMsg, &msgBuff);
  write(fd, &msgBuff, sizeof(msgBuff));
}
```

Posix Device Driver API

Summary:

- open(name, options, flags)
  - options such as non-blocking reads, sync writes
  - returns unique device file descriptor (fd)
- read(fd, buff, size) – returns bytes read
- write(fd, buff, size) – returns bytes written
- ioctl(fd, option, &arg) –
  - Get/set special device control options
  - Examples: downlink rate, device heartbeat counter, startAddress, endAddress, deviceID
- close(fd)
- select(fdSet, …, optionalTimeoutValue)

Posix Driver Installation

```
int mydevOpen(DEV_STRUCT* pDev, …);
int mydevClose(DEV_STRUCT* pDev, …);
int mydevWrite(DEV_STRUCT* pDev, …);
int mydevRead(DEV_STRUCT* pDev, …);
int mydevIoctl(DEV_STRUCT* pDev, …);

installPosixDriver(mydevOpen, mydevClose,
   mydevRead, mydevWrite, mydevIoctl);
```

Summary

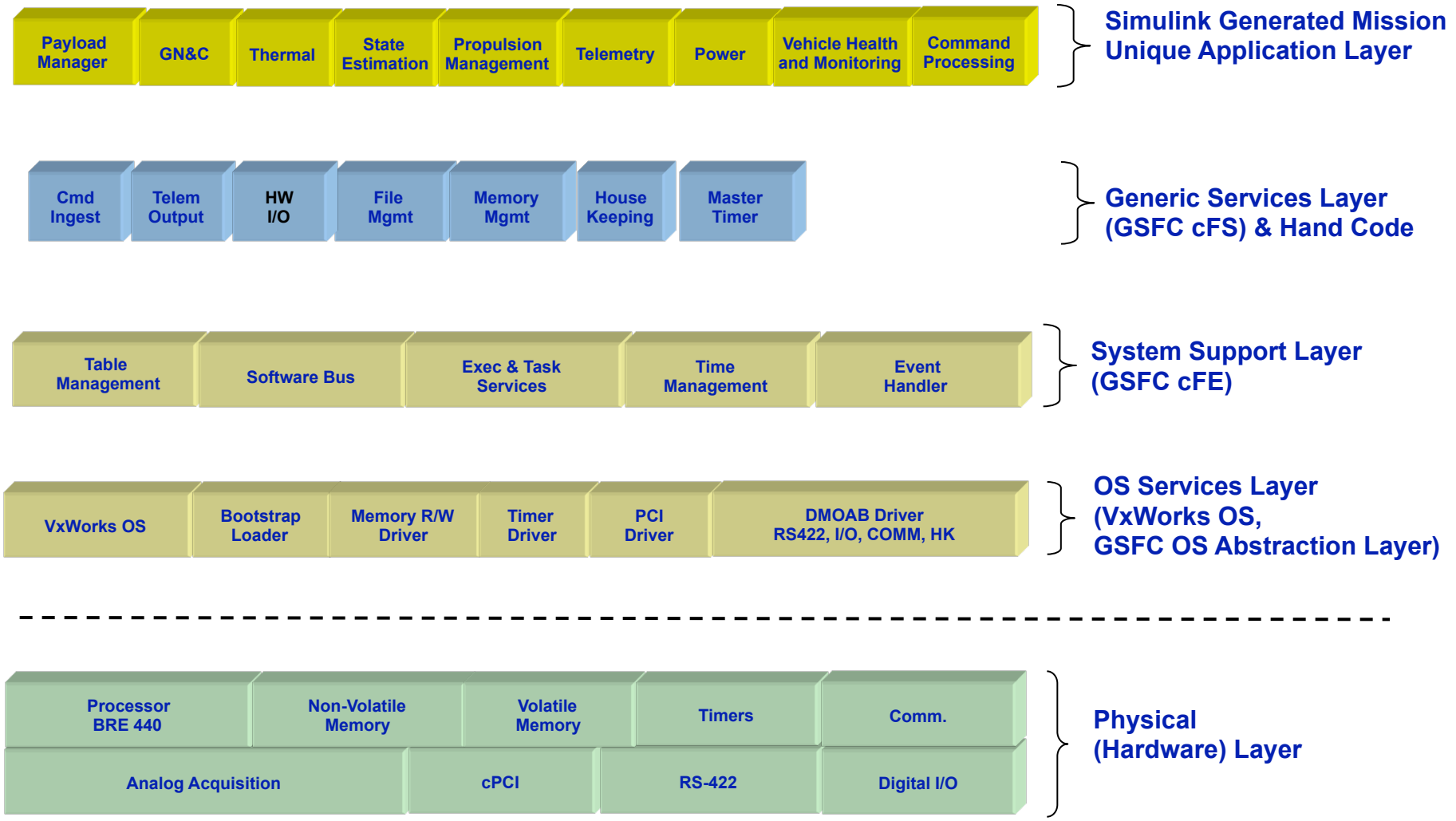
- Rapid Development Infrastructure
  - Model based (Simulink) autocode
  - Loosely-Coupled Architecture (cFE)
  - Early and Frequent Integration
- Early Integration in PHIL Testbeds
  - Evolving fidelity as H/W is available
  - Early Integration of Device Protocol Software
- Posix I/O Approach
  - Facilitates Mix’n’Match Evolution of transports
  - Meets LADEE Testbed Implementation Goals

Backup

Example Legacy FSW HWIO App

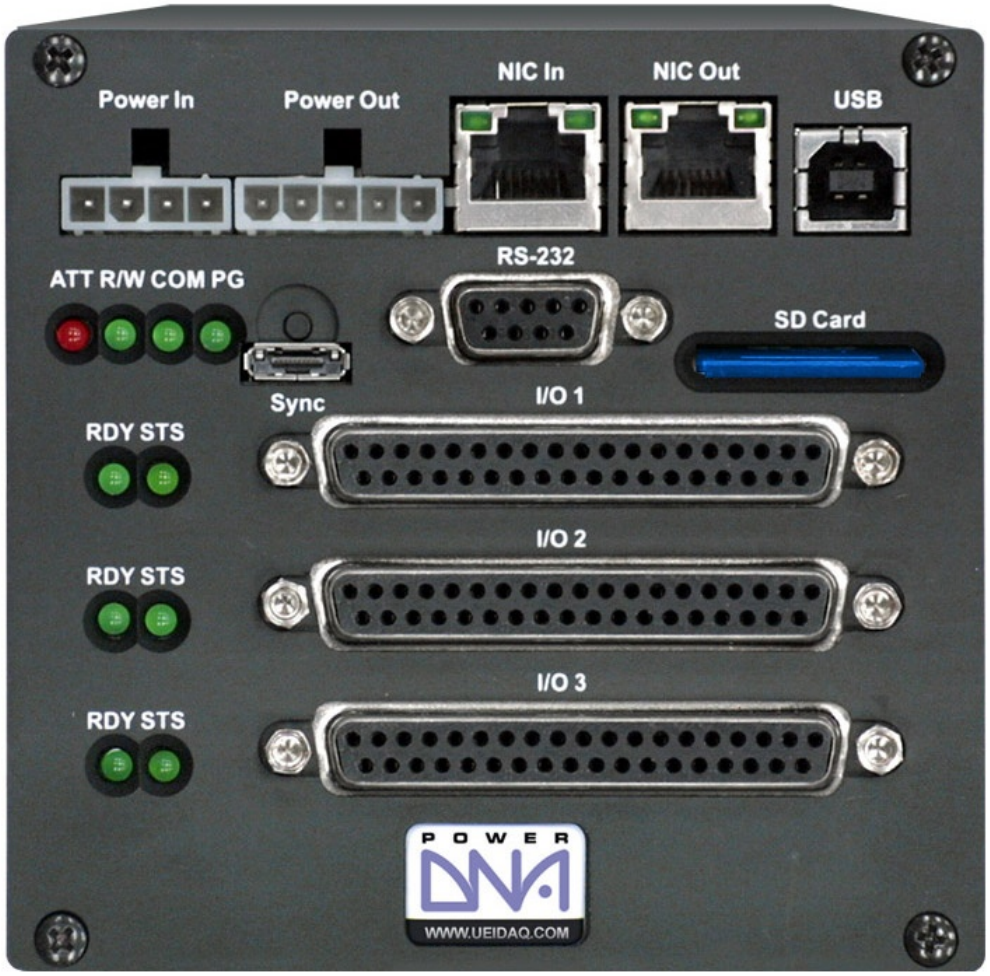
```
ST_Main() {
   while(AppIsRunning) {
        read_st(&st_in, timeout);
         parse_imu(&st_swbus_msg, &st_in);
         sb_send_msg(st_swbus_msg);
   }
}
/* Makefile targets link in the proper read_st
   for ONE_PIL, TWO_PIL or HIL */
```

Layered Architecture Approach



Hover Test

UEI Cube


Power In
Power Out
NIC In
NIC Out
USB
ATT R/W COM PG
RS-232
SD Card
Sync
I/O 1
RDY STS
I/O 2
RDY STS
I/O 3
RDY STS
POWER DNA
WWW.UEIDAQ.COM

Implementing Posix Wrapper around UEI API

- Concept
  - UEI Cube sends/receives data over gigabit ethernet to using API
  - UEI Cube internal sends data to i/o cards
- UEI API Modes
  - Blocking (not realtime)
  - Message Based (fixed size)
  - **Polling at High Frequency (background)**

Implementing Posix Driver for VxWorks

- Create a Table of Named Devices

  UEI_UART_DEV ueiUartDev[] = {
  … {“/dev/st”….}};
- Create Function to Initialize UEI API in polling mode
- Implement ueiUartOpen(), ueiUartClose(), ueiUartRead(), ueiUartWrite()
  - Tie together UEI API polling task ueiUartIomUpdate() with device table for read/ write application calls using ring buffers and semaphores.

Implementing Posix Driver for VxWorks

- Implement ioctol()
  - Implement selNodeAdd() to take advantage of built-in select() wakeup mechanism
- Register as a posix driver with VxWorks
  - iosDrvInstall(0, 0, (FUNCPTR)ueiUartOpen, (FUNCPTR)ueiUartClose, (FUNCPTR) ueiUartRead, (FUNCPTR)ueiUartWrite, (FUNCPTR)ueiUartIoctl);

Implementing Posix Driver for VxWorks

- ueiUartIomUpdate() implemented to utilize UEI API in high frequency polling task
  - Sends data to UEI Cube gotten from the write () ring buffer
  - Receives data from UEI Cube and makes available to read() ring buffer
    - Uses selWakeupAll() to un-block any select() calls waiting for activity on this file descriptor.