Preliminary Design of Robotic Control Software for Mars Sample Return - Capture, Containment, and Return System

Jacob T. Cassady[*]
NASA Langley Research Center, Hampton, Virginia, 23666

Ashok K. Prajapati[†], Elizabeth J. Geist[‡]
NASA Goddard Space Flight Center, Greenbelt, Maryland, 20771

Bradley C. Tse[§], Benjamin L. Osborne[¶]
Microtel LLC., Greenbelt, Maryland, 20771

Jeffrey A. Angielski[‖]
HII Mission Technology Corporation, Newport News, Virginia, 23607

Joseph B. Lattisaw[**], Francis B. Hallahan[††]
Embedded Flight Systems Inc., Laurel, Maryland, 10193

The Mars Sample Return (MSR) campaign aims to acquire and return to Earth a set of Mars samples for investigation in terrestrial laboratories. Mars 2020 has collected an adequate number of samples and deposited them in sealed sample tubes at a designated Martian depot. Sample tubes will be placed in cylindrical containers called Orbiting Samples (OS) by the Perseverance Rover, and later brought to Earth by the Earth Return Orbiter (ERO) and the MSR - Capture, Containment, and Return System (CCRS). Robot Software (RSW) is a set of software processes for commanding and monitoring the avionics that controls robotic mechanisms designed to sterilize and install OS into Earth Entry System (EES) for return to Earth. This paper describes a preliminary design of RSW including motion modes, architecture, and finite state machine. Additionally, software engineering procedures and testing of RSW is provided. A preliminary performance analysis is presented and the paper concludes with future work and a discussion of design decisions.

[*]Flight Software Engineer, Flight Software Systems Branch, Mail Stop 064, AIAA Member.
[†]Flight Software Engineer, Flight Software Systems Branch, Mail Stop 148.
[‡]Technical Management, Flight Software Systems Branch, Mail Stop 148.
[§]Flight Software Engineer, Flight Software Systems Branch, Mail Stop 148.
[¶]Flight Software Engineer, Flight Software Systems Branch, Mail Stop 148.
[‖]Flight Software Engineer, NASA's Exploration & In-space Services (NExIS), Mail Stop 480.
[**]Flight Software Engineer, Flight Software Systems Branch, Mail Stop 148.
[††]Flight Software Test Engineer, Flight Software Systems Branch, Mail Stop 148.

I. Introduction

The Mars Sample Return (MSR) campaign is being carried out in partnership by National Aeronautics and Space Administration (NASA) and European Space Agency (ESA). At the time of this publication, the MSR campaign architecture is undergoing further study. The MSR campaign architecture described here follows Mars Sample Return Mission Concept Status described in 2020 by Muirhead et al. of Jet Propulsion Laboratory (JPL) [1]. Figure 1 shows a proposed MSR Mission Timeline.

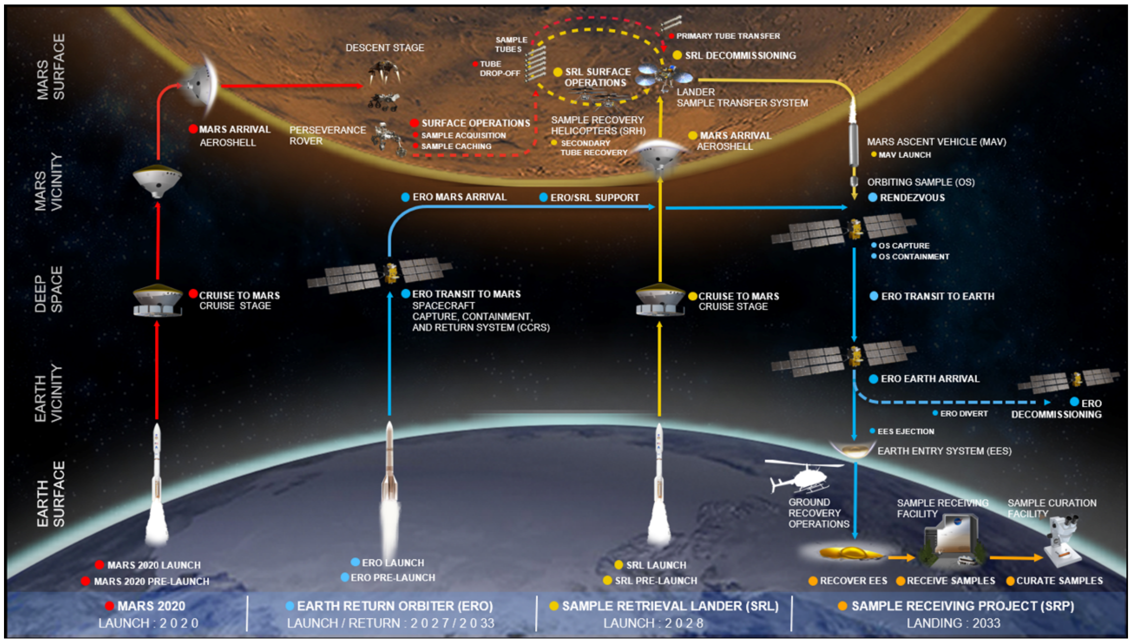


Fig. 1 MSR Mission Timeline

The first mission to launch in the MSR campaign was Mars 2020 Mission, led by JPL [2]. The goal of Mars 2020 was to land the Perseverance rover to acquire samples and cache them at drop-off locations. Mars 2020 has collected an adequate number of samples and deposited them in sealed sample tubes at a designated Martian depot.

The second and third missions of the MSR campaign are planned to happen in parallel. The Sample Retrieval Lander (SRL) Mission is led by JPL with support from ESA and NASA Marshall Space Flight Center (MSFC) [3]. The goal of SRL is to land on Mars, retrieve Mars samples from drop-off locations, and place the Mars samples into the Mars Ascent Vehicle (MAV) for rendezvous with the Earth Return Orbiter (ERO).

ERO is led by ESA with support from NASA Goddard Space Flight Center (GSFC) and JPL for design of Capture, Containment, and Return System (CCRS) payload [4]. ERO is designed to orbit Mars and provide support for sample retrieval activities including returning the samples inside Earth Entry System (EES). During rendezvous, CCRS is designed to capture, contain, and load the Mars Orbiting Samples (OS) into EES.

The final mission planned for the MSR campaign is the terrestrial Sample Receiving Project (SRP) Mission. SRP is being led by NASA and ESA. The goal is to recover the OS from the EES, and curate Martian samples at the Mars

Returned Sample Handling (MRSH) facility.

This paper presents a preliminary design of Robot Software (RSW): a component of the CCRS Flight Software (FSW) designed to command and control the avionics for a two degrees of freedom (DOF) robotic mechanism and End-Effector (EE) onboard the CCRS. In section II, an overview of the Concept of Operations (ConOps) and associated hardware is presented. Section III introduces the CCRS FSW architecture. Section IV describes the RSW design and implementation. Section V highlights NASA Procedural Requirements (NPRs) related to software engineering, and the RSW development team's implementation of processes to meet procedural and operational requirements. This paper ends with a preliminary analysis of performance in section VI and discussions on design choices and the current development state of RSW in section VII.

II. Concept of Operations

The ConOps described in this section is narrowly focused on information relevant to RSW operation and design decisions. Section II.A gives an overview of the Robotic Transfer Assembly System (RTAS) mechanisms and avionics. Section II.B describes the RTAS actions, called **motion primitives**, supported by RSW. Section II.C lists configurations of motion primitives, called **behaviors**, supported by RSW. The timeline of RTAS operations and the behaviors used to accomplish each operation are described in section II.D.

A. Robotic Transfer Assembly System

RTAS is a set of mechanisms and the associated avionics and software used to perform pick-and-place operations within the CCRS Assembly Enclosure (AE). This section introduces select components of RTAS, shown in Figure 2, related to RSW. Section II.A.1 details RTAS mechanisms. Section II.A.2 presents information on avionics designed to control RTAS mechanisms. The hardware and avionics overview ends in section II.A.3 with a description of the processor board selected to run CCRS FSW.

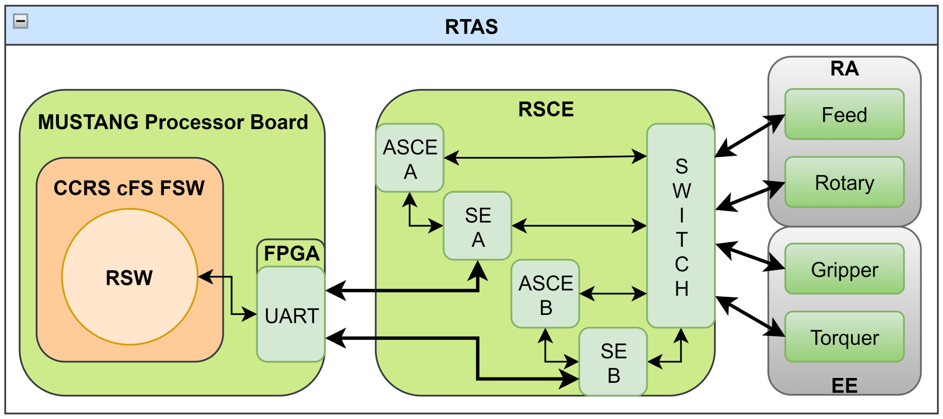


Fig. 2 RTAS Components

1. Robotic Transfer Assembly

The Robotic Transfer Assembly (RTA) is the set of mechanisms within RTAS including the Robotic Assembly (RA) and the EE. RA is a two DOF robotic mechanism allowing rotational and linear motion to position EE. RA enables linear translation using the Feed Stage, also known as the Gantry. Rotational translations of the RTA are achieved using the Rotary Stage. The EE is mounted to the end of the RA Feed Stage, and is designed to grip OS using the Gripper mechanism and provide axial torque during assembly operations using the Torquer mechanism. Figure 3 displays the RTA mechanisms.

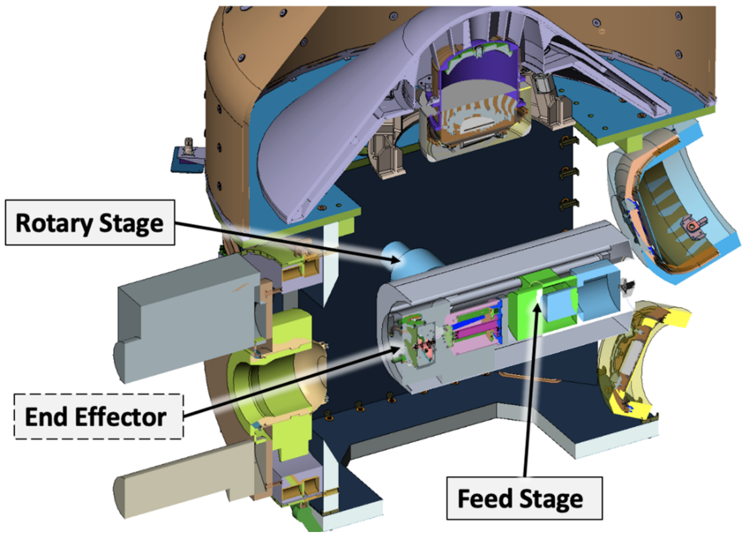


Fig. 3 RTA Mechanisms [Picture Credit: NASA/JPL]

2. Robot Servo Control Electronics

The Robot Servo Control Electronics (RSCE) is a redundant set of avionics configured to control the RTA mechanisms. RSCE includes redundant Sensor Electronics (SE), redundant Actuator Servo Control Electronics (ASCE), and a switch card as shown in Figure 2. SE is responsible for reading RTAS sensors and communicating with ASCE and RSW. The ASCE is a closed-loop actuator controller with heritage from the Mars Science Laboratory, the Robotic Fueling Mission, and the On-orbit Servicing, Assembly, and Manufacturing 1 (OSAM-1) missions.

3. Modular Unified Space Technology Avionics for Next Generation Processor Board

The FSW for MSR CCRS is designed to run on a Modular Unified Space Technology Avionics for Next Generation (MUSTANG) Processor Board [5]. MUSTANG was developed by GSFC. MUSTANG has a pedigree of successful space flight missions including the Magnetospheric Multiscale Mission and the Global Precipitation Measurement Mission.

MUSTANG is designed as a set of configurable cards. MUSTANG Processor Card includes a Frontgrade

Gaisler GR712RC Dual Core LEON3FT SPARC V8 Processor chip capable of 200 million instructions per second. Additionally, the MUSTANG Processor Card features a Microchip RTG4 Field Programmable Gate Array (FPGA) capable of producing hardware interrupt signals as well as supporting SpaceWire (SpW) [6] and Universal Asyncronous Receiver-Transmitter (UART) interfaces.

B. Motion Primitives

Motion primitives represent atomic RTAS motion actions. Motion primitives have configurable operational **arguments** and non-volatile **parameters**. Motion primitive arguments are configured using ground systems or in-flight scripting while parameters are stored in non-volatile memory.

RTAS supports four motion primitives including: Position Servo, Open-Loop Actuator, Active Limp, and Active Hold described in sections II.B.1, II.B.2, II.B.3, and II.B.4 respectively. The Position Servo and Open-Loop Actuator motion primitives are used in higher-level behaviors described in section II.C. The Active Limp and Active Hold motion primitives can be used in off-nominal situations where ground operator intervention is required.

1. Position Servo

The Position Servo Motion Primitive performs closed-loop position control of RTA mechanisms based on hall effect sensor data. The Position Servo Motion Primitive is used for single actuator control of RA Feed Stage and EE mechanisms. Arguments for Position Servo Motion Primitive include the target actuator, goal position, goal type (absolute or relative), speed, and functional motion type. Parameters include current limits, position ranges, maximum thresholds, and dead band configuration.

2. Open-Loop Actuator

The Open-Loop Actuator Motion Primitive is a critical redundancy if there were issues with closed-loop position control. The Open-Loop Actuator Motion Primitive is used to commutate an RTA actuator for a duration. Arguments for Open-Loop Actuator Motion Primitive include the target actuator, direction, speed, duration, and a flag denoting if a stall was required. Parameters include expected stall positions and tolerances, persistence for stall declaration, and max allowed duration.

3. Active Limp

The Active Limp Motion Primitive allows for back driving of RTA mechanisms. The Active Limp Motion Primitive is used to open the brakes of an actuator for a configurable duration. Arguments for Active Limp Motion Primitive include the target actuator, maximum allowed motion distance, and the duration. Parameters include max duration and max back-driving speed.

4. Active Hold

The Active Hold Motion Primitive performs closed-loop position control of RTA mechanisms to hold a position for a prescribed position using hall sensors for feedback. Arguments for Active Hold Motion Primitive include the target actuator and duration. Parameters include the max duration, max back-driving duration active hold, max back-driving speed, and current limits.

C. Behaviors

RTAS behaviors are created from configurations of motion primitives with additional logic for fault protection or autonomy. Similarly to motion primitives, Behaviors have configurable operational arguments and non-volatile parameters. Behavior arguments are configured using ground systems or in-flight scripting while parameters are stored in non-volatile memory. RTAS supports three behaviors including: Iterative Position Servo with Output Sensor Feedback, Servo-to-Load, and Open-Loop Actuator with Contact Switch Post-Motion Confirmation as described in sections II.C.1, II.C.2, and II.C.3 respectively.

1. Iterative Position Servo with Output Sensor Feedback

The Iterative Position Servo with Output Sensor Feedback Behavior enables control of the position of a mechanism using sensor feedback. Control of the target mechanism with sensor feedback is achieved by leveraging the measured difference between a current sensor reading and target sensor reading with the Position Servo Motion Primitive. The Iterative Position Servo with Output Sensor Feedback Behavior is used to position the RA rotary stage. Arguments include the goal position, max iterations, and speed. Parameters include limits and sensor signal period.

2. Servo-to-Load

The Servo-to-Load Behavior enables closed-loop position control of mechanisms that terminates when a Load Cell axial force reading exceeds a configurable threshold. The Servo-to-Load Behavior is performed by wrapping the Position Servo Motion Primitive and is used to move the RA Feed Stage until an axial force threshold is reached. Arguments include goal position, speed, success threshold, overshoot allowed, and a Boolean representing post-motion contact switch check. Parameters include limits and expected contact switch states after motion completes.

3. Open-Loop Actuator with Contact Switch Post-Motion Confirmation

The Open-Loop Actuator with Contact Switch Post-Motion Confirmation Behavior open-loop commutates a mechanism for a desired duration in a specified duration. The Open-Loop Actuator with Contact Switch Post-Motion Confirmation Behavior is performed by leveraging contact switch state checks and the Open-Loop Actuator Motion Primitve. The Open-Loop Actuator with Contact Switch Post-Motion Confirmation Behavior is used in a failure mode when an actuator's hall sensor fails or there are other issues with closed-loop position control. Arguments include target actuator, direction, duration, and a Boolean denoting if a stall is required. Parameters are the same as Open-Loop Actuator Motion Primitive with additional parameters for contact switches.

D. Operation Phases

RTAS operates soon after flight and again during Low Mars Orbit. This section details the timeline of operations, shown in Figure 4, including the sequence of behaviors used to accomplish each operation. During nominal operation phases, a Position Servo Motion Primitive is used to actuate RA Feed Stage, EE Gripper, and EE Torquer actuators. The Iterative Position Servo with Output Sensor Feedback Behavior is used to rotate RA Rotary Stage. Section II.D.1 describes commissioning of CCRS and RTAS during ERO's travel from Earth to Mars. Section II.D.2 details picking up the Integrated Lid Assembly (ILA). After OS is captured, RTAS places ILA on OS and uses it to pick up Lid+OS (LOS) as described in section II.D.3. Section II.D.4 describes behaviors of RTAS during Ultraviolet (UV) Illumination. The final phase, LOS Install, is described in section II.D.5 and ends with RTAS retracted and stowed.

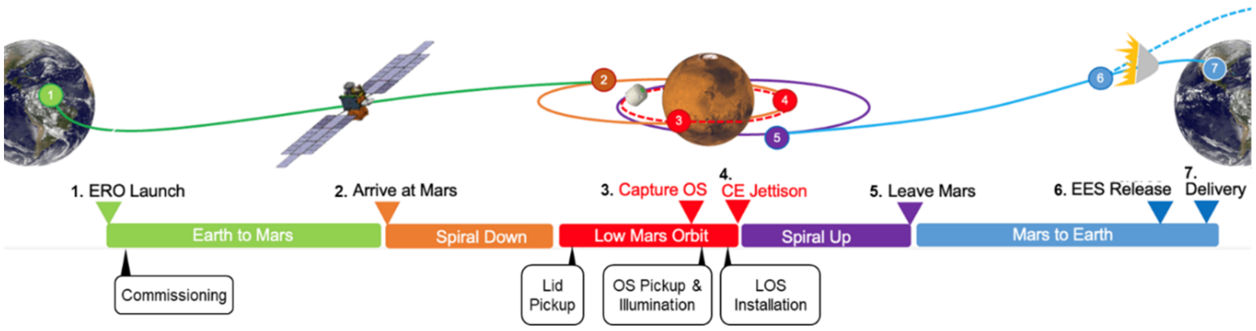


Fig. 4 RTAS Operation Timeline [Picture Credit: NASA/JPL]

1. Commissioning

Before launch, functional tests of RTAS and CCRS will occur on the launch pad. Commissioning of CCRS is expected to occur within the first 30 days after launch to ensure faster communications. During commissioning, CCRS is powered on. After functional checkouts, RSCE is powered on and RTAS is initialized. Launch locks on RTAS are released and a checkout is performed of mechanisms and actuator range of motions. To end commissioning, RSCE and CCRS are powered off.

2. Lid Pickup

Prior to OS capture, RTAS is used to pick up ILA. Lid Pickup Phase begins by initializing RTAS and heating EE for sterilization. RA then rotates to align EE with lid pick up location shown in Figure 5. After rotating, RA linearly approaches ILA until making contact. Once nominal compression of contact switches is confirmed, the Lid Release Mechanism (LRM) is released. Next, RA fully grasps ILA and retracts linearly to clear the pose. RA then rotates to align with EES. After rotation, RA linearly approaches EES. To end Lid Pickup Phase, RTAS is shutdown.

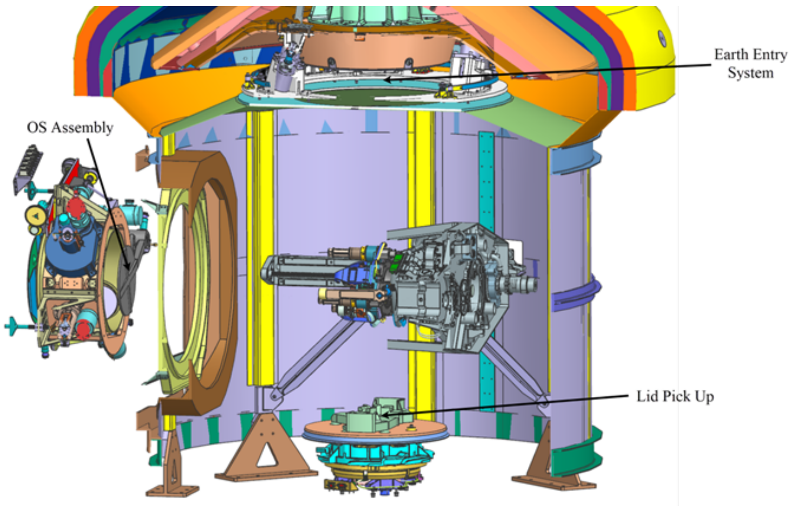


Fig. 5 AE [Picture Credit: NASA/JPL]

3. OS Pickup

During OS Pickup Phase, EE latches ILA onto OS. OS Pickup Phase begins by initializing RTAS and heating EE. Once initialized, RA retracts from EES and rotates to align ILA with OS assembly. RA then linearly approaches the OS. After RA approach, EE is relaxed to latch onto OS with ILA. After latching, RA is retracted to standoff and EE is used to fully grasp LOS. To end OS Pickup Phase, EE is shutdown.

4. Ultraviolet Illumination

UV Illumination is used to sterilize OS. UV Illumination Phase includes iterative partial retraction of RA, shutdown of RSCE, and then running the UV system for several hours, giving ground operations time to review related telemetry. UV Illumination Phase ends with complete retraction of RA and heating of EE for use in LOS Phase.

5. LOS Install

During LOS Install Phase, LOS is installed in EES. LOS Install Phase begins with rotating RA to align LOS with EES. RA then performs a linear approach to EES. To angularly align LOS to EES, EE relaxes grasp on LOS. In nominal operations, relaxing EE causes compression of EE contact switches.

After alignment, RTAS must secure LOS to EES. EE first opens Gripper fully. EE then engages Torquer to rotate LOS until Torquer stalls.

LOS Install Phase ends with retracting and stowing RTAS. EE first unwinds Torquer for full clearance. RA then retracts linearly to free space. EE returns Torquer to starting orientation. RA retracts fully to stow position. Finally RA rotates rotary stage to stow position before shutting down RTAS.

III. CCRS FSW Overview

FSW for MSR CCRS, including RSW, is built upon the open-source core Flight System (cFS) framework developed by GSFC [7] using C programming language. MSR CCRS FSW can be compiled for a Linux operating system for testing, and it can also be compiled for a VxWorks 7 [8] real-time operating system (RTOS).

core Flight Executive (cFE), a component of cFS, is a collection of services and associated framework for FSW development with extensive flight heritage and flight certification beginning in 2009 [9]. cFE has five core services: Executive Service (ES), Software Bus Service (SB), Event Service (EVS), Table Service (TBL), and Time Service (TIME) [10]. cFS applications leverage these services for communication, scheduling, logging, and more. cFS Applications are sets of functions and data structures recognized as a single-entity by the cFE [11]. cFS Applications communicate with each other using SB and a publish-subscribe messaging model.

CCRS FSW utilizes open-source cFS Applications developed by GSFC including Limit Checker (LC) Application, Stored Commands (SC) Application, Scheduler (SCH) Application, Housekeeping (HK) Application, Health and Safety (HS) and more. LC monitors telemetry and produces event messages when telemetry is found outside of pre-defined thresholds [12]. SC uses a timetable to produce sequences of stored commands [13]. SCH generates SB messages at pre-determined timing intervals [14]. HK builds and sends combined telemetry messages from multiple cFE Applications in order to minimize downlink telemetry bandwidth [15]. Lastly, HS is used to monitor applications and events [16].

New cFS Applications were developed for CCRS FSW including RSW, Thermal Control (TC), SpaceWire Router (SPWR), and SpaceWire Node (SPWN). TC is responsible for commanding heaters aboard CCRS. SPWR facilitates packet routing from SpW hardware to various cFS Applications. SPWN implements a SpW hardware node connection to a SpW endpoint.

CCRS FSW also leverages heritage cFS Applications from GSFC missions including the Flight System Test and Operations Language (FSTOL) Application. FSTOL enables execution of Systems Test and Operations Language (STOL) scripts during flight to execute logic. An exhaustive list of cFS Applications utilized or developed is shown in Appendix Table 5. Figure 6 provides a display of the interfaces of each cFS Application within CCRS FSW.

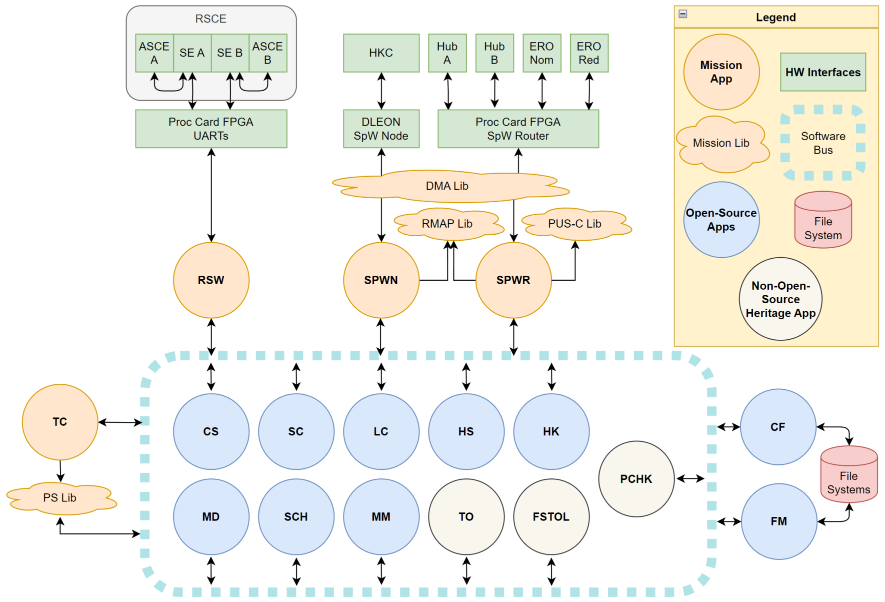


Fig. 6 CCRS FSW cFS Context

IV. RSW Preliminary Design

RSW is designed to facilitate RTAS motion primitives. The role of RSW in each motion primitive is to validate arguments, configure RSCE before motion, initiate motion, monitor motion and forward telemetry, and perform post-motion analysis. Section IV.A describes the external interfaces RSW has with FSW and avionics elements. Section IV.B introduces command types supported by RSW. Section IV.C overviews telemetry produced by RSW. Section IV.D details the implementation of RSW with a focus on the logic of tasks and Finite State Machine (FSM) design.

A. External Interfaces

RSW has external interfaces with SB and SE. Figure 7 depicts end-to-end interactions with RSW and avionics components. RSW is able to send commands to and receive telemetry from SE through the MUSTANG FPGA RS422 UART interface using the Consultative Committee for Space Data Systems (CCSDS) packet protocol. Command and Telemetry (C&T) between RSW and robot operators are communicated through ERO via SpW. ERO communicates with Ground Support Equipment (GSE) through windows of the Deep Space Network [17] .

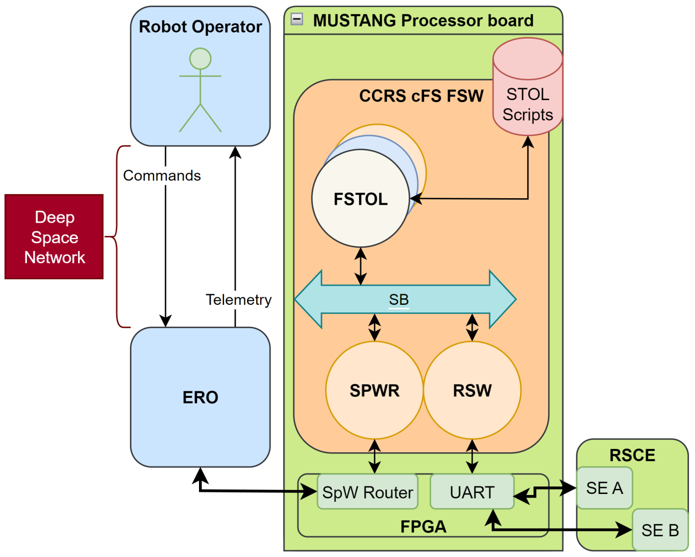


Fig. 7 RSW C&T Flow

RSW interacts with other cFS Applications as shown in Figure 8. FSTOL is used to implement RTAS behaviors through scripting of Motion Primitive Commands with additional logic. HK requests to RSW are produced by SCH at a frequency of 1 Hz. LC subscribes to RSW telemetry to monitor RSW faults at runtime.

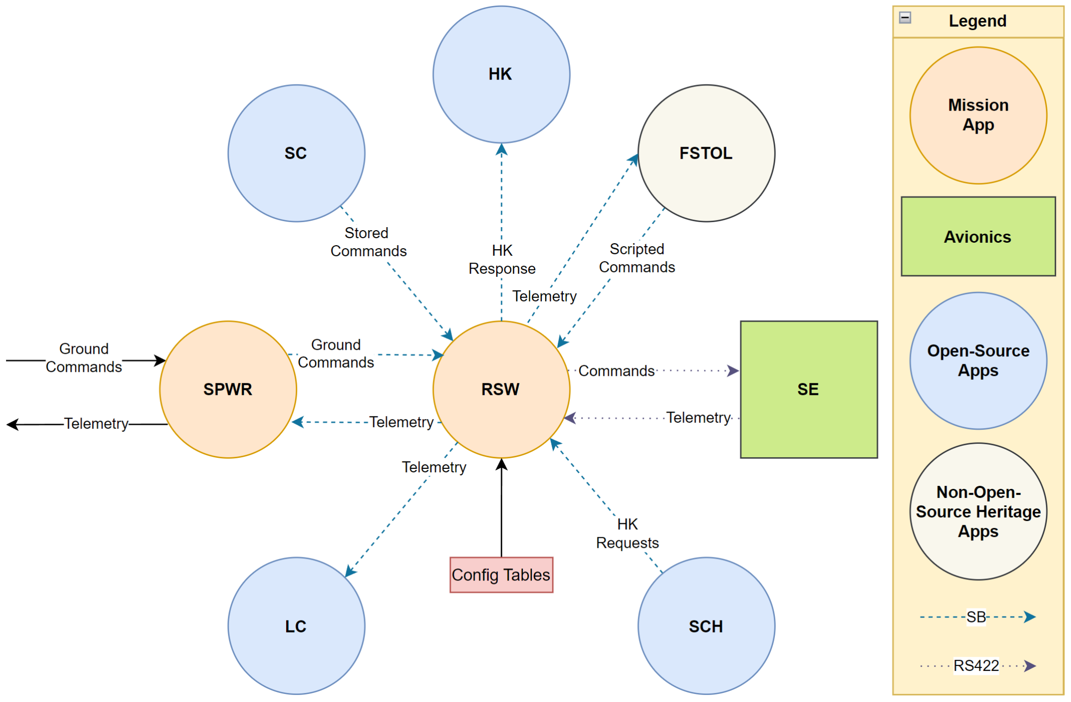


Fig. 8 RSW FSW Context

B. Commands

RSW supports four command types: High-Priority Commands, RSW Commands, RSCE Commands, and HK Requests. Commands are formatted following CCSDS packet protocol and are sent from ground systems or other cFE applications at runtime. High-Priority Commands include categories of stop commands as well as a clear fault state command. A list of High-Priority Commands can be found in Appendix Table 6.

RSW Commands are used to read and modify the state of RSW. RSW Commands include motion primitives, information requests, and a stop motion command. Motion primitive commands are sent by ground systems or in-flight FSTOL scripting of behaviors and include motion primitive arguments. Motion primitive parameters are sourced from on-board cFE tables. A list of RSW Commands can be found in Appendix Table 7.

RSCE Commands are meant to be passed directly to SE without advancing RSW FSM. RSCE Commands are first verified by RSW, and if and safe to perform, sent to SE. A list of RSCE Commands can be found across Appendix Tables 8 and 9.

C. Telemetry

RSW produces telemetry packets following CCSDS packet protocol and places them on SB to be telemetered to ground through SPWR. RSW produces two types of telemetry: telemetry forwarded from SE and telemetry designed to report status. Telemetry from RSW is produced each cycle with a maximum number of packets each cycle of 4: one HK request, three RSCE packets, and one FSM status packet.

Telemetry forwarded from SE include scheduled 10 Hz telemetry and command responses. Scheduled 10 Hz telemetry produced by SE include three data fields: a set of fault status bits, a set of 10 packets containing data from ASCE collected by SE at 100 Hz, and a set of recent motor speed, goal, and position data. Each of the 10, 100 hz packets, include potentiometer readings, motor currents, contact switch states, load cell readings, and hardware state information. Command responses from SE have two types: long command responses and short command responses. Long command responses include a larger allocation of bytes for data storage than short command responses. Both long command responses and short command responses include identification of the initial command related to the response. Including the command identification allows RSW to verify when confirming an acknowledgement.

Telemetry used to report status include HK telemetry and FSM information. HK telemetry is produced at a rate of 1 Hz. HK telemetry includes command validation success, failure, and execution counts that are incremented with each command. FSM information is telemetered once each cycle and includes current state and sequence count attributes.

D. Implementation

RSW is implemented with two cFE tasks, as shown in Figure 9. RSW has internal interfaces for communicating between tasks and accessing stored configuration tables. RSW FSM is described in section IV.D.1. Section IV.D.2

details Main Task containing core logic and FSM. Design of Worker Task, used for offloading computationally expensive operations, is presented in section IV.D.3.

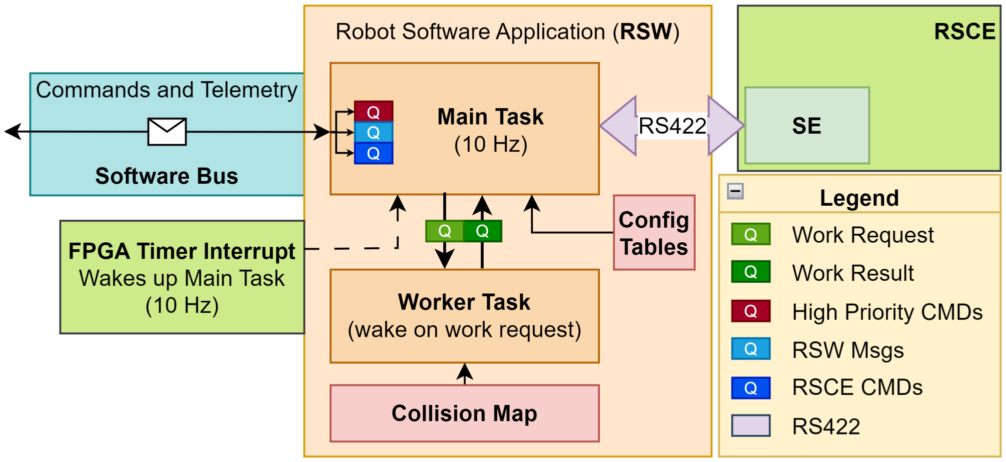


Fig. 9 RSW Application Design

1. Finite State Machine Design

The Main Task is implemented with a FSM with four attributes: current state, requested state, current state run count, and current state sequence number. Sequence number is used to facilitate behavior changes within a state. RSW FSM has six methods: *Initialize*, *StateExit*, *StateEntry*, *StateRun*, *Step*, and *RequestStateTransition*. Valid state transitions are shown in Figure 10. Table 1 includes a list of FSM states and associated descriptions.

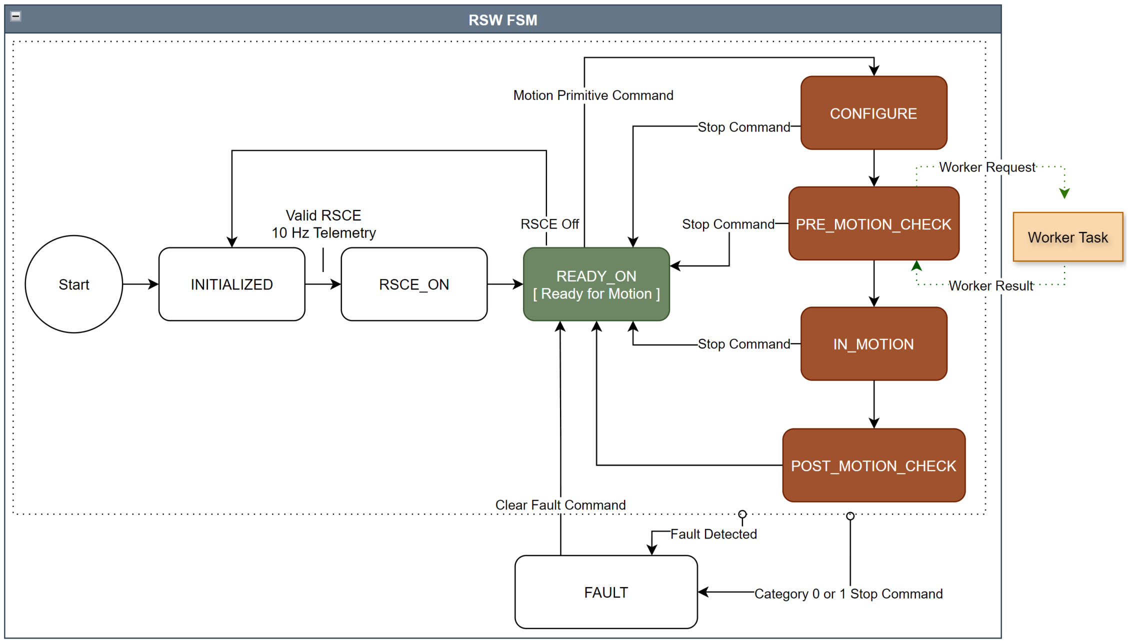


Fig. 10 RSW FSM

State	Description
UNKNOWN	Initial state on startup.
INITIALIZED	After start up procedures and before valid telemetry from RSCE.
RSCE_ON	Initial configuration of RSCE.
READY_ON	Systems are initially configured. Ready to receive motion primitive commands.
CONFIGURE	Configure RSCE for motion primitive.
PRE_MOTION_CHECK	Perform pre-motion check using Worker Task for motion primitive.
IN_MOTION	Execute motion. Monitor telemetry.
POST_MOTION_CHECK	Perform post motion anomaly checks.
FAULT	A fault has occurred. Pending external intervention.

Table 1 RSW FSM States

Upon startup, RSW begins by initializing current state to UNKNOWN and requested state to INITIALIZED. State run counts and sequence number are set to zero. Every cycle, Main Task calls *Step* method shown in Figure 11. *Step* begins by checking if current state and requested state are the same. If current state and requested state are not the same, RSW calls *StateExit* and then *StateEntry*. *StateExit* and *StateEntry* methods are used to cleanup attributes from a previous state and prepare for the next state. After *StateExit* and *StateEntry* methods, current state is then set to requested state. Once current state is set, or if current state and requested state were the same to begin with, *Step* method ends by calling *StateRun*.

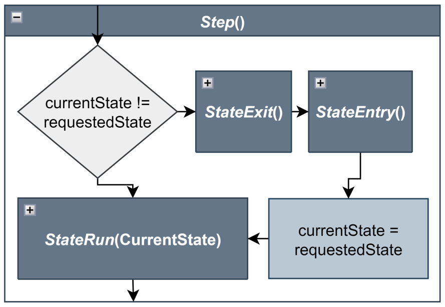


Fig. 11 FSM Step Design

StateRun switches behavior depending on current state and the active motion primitive. At the end of *StateRun*, current state run count attribute increments. *StateRun* also switches procedures on sequence count for states such as CONFIGURATION which include processes carried out over multiple cycles.

State transitions are validated before being set. Invalid state requests are rejected and produce event messages. Invalid state requests do not cause RSW to enter FAULT. State requests are allowed during command processing, by FSM itself when performing motion primitive procedures, or anytime there is a fault condition. Transitioning to FAULT is valid for all states.

Once RSW is in INTIALIZED, RSW waits for telemetry from SE before entering RSCE_ON. After performing initial configuration of RSCE, RSW enters READY_ON. When RSW is in READY_ON, RSW is ready to accept motion primitive commands.

When a motion primitive command is received while RSW is in a READY_ON, RSW enters CONFIGURE. During CONFIGURE, RSW sends a series of register write commands to SE to prepare for motion primitive execution. Once RSCE is configured, RSW enters PREMOTION_CHECK. During PREMOTION_CHECK, RSW sends a work request to Worker Task. If a nominal work result is received from Worker Task, RSW enters IN_MOTION. IN_MOTION begins by sending a start command to SE and ends when motion primitive stop conditions are met. When motion is complete, RSW performs a post-motion check for anomalies before re-entering READY_ON. If RSW receives a stop command during CONFIGURE, PRE_MOTION, IN_MOTION, and POST_MOTION_CHECK, RSW transitions directly back to READY_ON.

2. Main Task Design

The Main Task contains core logic for RSW including FSM. Main Task is woken up by a FPGA hardware timer at a rate of 10 Hz. Main Task interfaces with SB, SE, and Worker Task using SB pipes as message queues. Main Task also uses internal SB pipes as queues for triaging commands and splitting up processing over multiple cycles. Control flow for Main Task is shown in Figure 12.

Once RSW receives a wakeup signal from FPGA hardware timer, Main Task cycle begins by removing all CCSDS Commands from SB and categorizing commands in queues following command type, described in section IV.B. RSW iterates over command queues and interfaces starting with high priority command queue. If a high priority command is found, RSW immediately processes and forwards to SE.

After checking for high priority commands, Main Task checks for incoming packets from SE on RS422 interface. If SE telemetry packets are found, RSW parses the SE telemetry . If SE telemetry is determined to be 10 Hz Telemetry, the telemetry is processed to update FSM and data stores. If SE telemetry is determined to be an RSCE command response from a previous command to RSCE, RSW clears the RSCE command acknowledgement flag. After SE telemetry is parsed and processed, SE telemetry is forwarded to SB to be telemetered to ground through SPWR. In one cycle, RSW can process a maximum of 3 packets from RS422 interface. Setting a maximum number of packets restricts Main Task to constant runtime complexity. The maximum number of packets to process is determined by the maximum number of packets RSW could receive from each source in a nominal cycle.

Once all, or the maximum number of 3, RS422 packets are processed, Main Task checks for results from previous requests to Worker Task. An incoming worker result is only expected when FSM is in PRE_MOTION. Unexpected worker results cause FSM to enter FAULT. After checking for a response from Worker Task, Main Task processes a maximum of 3 RSW Commands. The maximum of RSW Commands per Main Task cycle driven by the two systems that produce RSW Commands, FSTOL and Ground Operations, which each have a maximum transmission rate of 1 Hz.

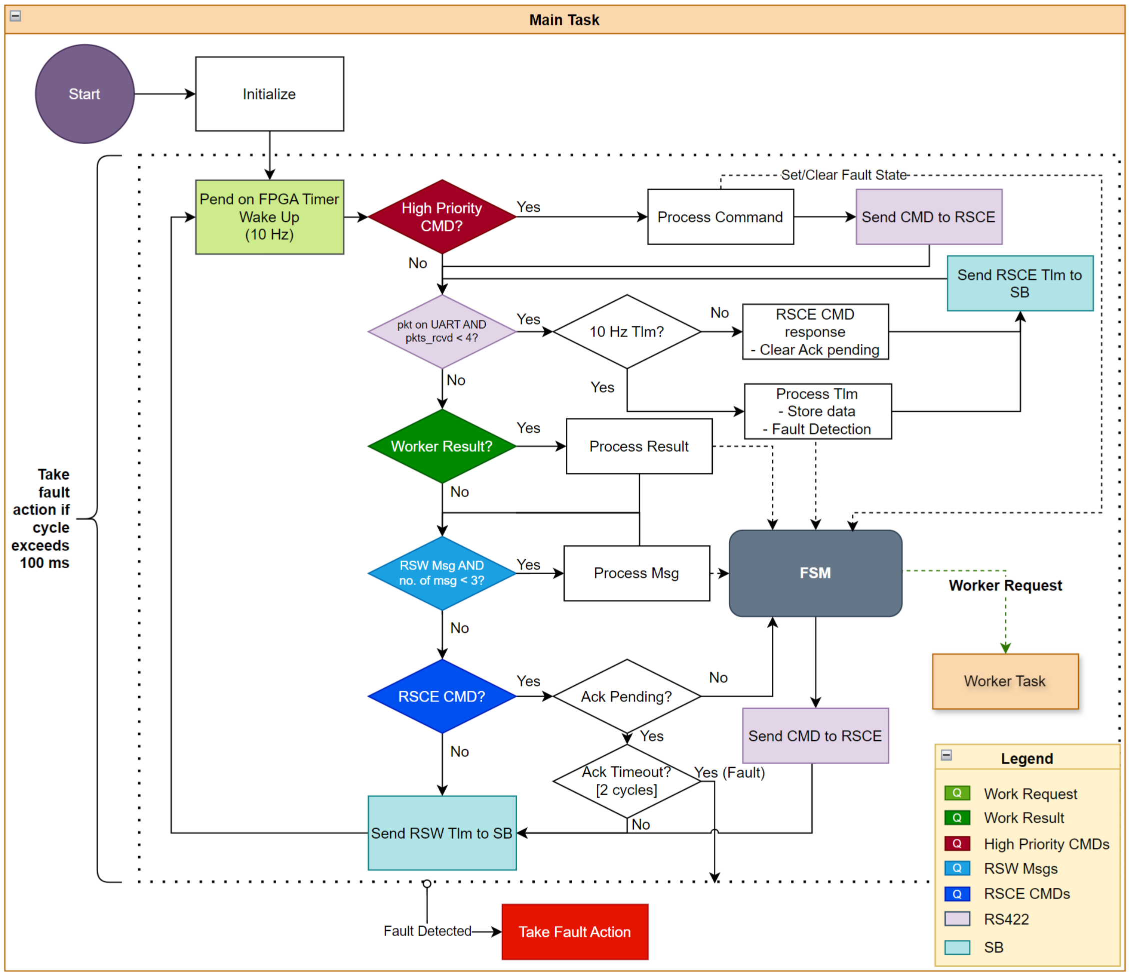


Fig. 12 RSW Main Task Design

Next, RSW checks for RSCE Commands. If a previous RSCE command acknowledgement is pending, Main Task waits until next cycle to send the RSCE command. If Main Task completes two cycles without receiving an RSCE command acknowledgement, FSM enters FAULT. Lastly, Main Task calls FSM *Step* and adds telemetry to SB. When finished, Main Task performs a non-blocking sleep until Main Task is woken up again by the hardware timer signal.

3. Worker Task Design

The Worker Task performs collision detection during PRE_MOTION. The collision detection operation evaluates a pending motion for off-nominal hardware collisions. Collision detection is off-loaded to a separate task due to possible large processing cost of performing the collision detection operation. Appendix Figure 13 displays the control flow for Worker Task. Worker Task waits on a work request from a SB pipe. After collision detection is completed, the result is sent back to Main Task through a different SB pipe.

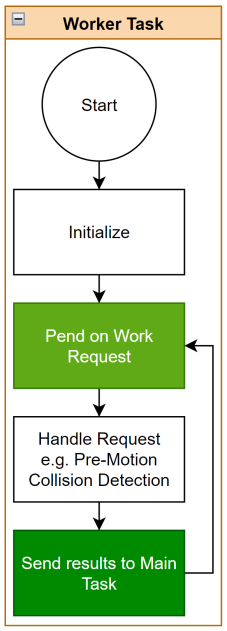

Worker Task
Start
Initialize
Pend on Work Request
Handle Request e.g. Pre-Motion Collision Detection
Send results to Main Task

Fig. 13 RSW Worker Task Design

V. Software Engineering and Testing

Software developed by NASA must follow NASA Procedural Requirements (NPR) 7150.2D which provides chapters of software engineering requirements [18]. Requirements in NPR 7150.2D are applicable based on the determined NASA software classification. RSW is classified as Class B, Non-Safety Critical. Class B software is described by 7150.2D as "non-human space-rated software systems or large-scale aeronautics vehicles." Select software engineering procedural requirements (SWE) from NPR 7150.2D are described in the following sections as well as processes to meet those NPRs. Section V.A relates to SWE-187 and Git version control. Section V.B relates to SWE-135 and usage of static analysis tools. Section V.C relates to SWE-062 and unit tests. Section V.D relates to SWE-066 and functional testing against operational requirements.

A. Configuration Management

SWE-187 states software shall be placed under configuration management before testing. Git [19] and GitLab [20] were utilized for configuration management. RSW source code exists as a submodule repository of a larger cFS FSW repository which includes all CCRS cFS Applications. Pre-commit git hooks are used to automate the usage of Clang Format [21] to format source code. GitLab workflows are used to automate build testing and unit testing.

B. Static Analysis

SWE-135 asserts development teams shall use static analysis tools. CodeSonar[22] was used for static analysis. Legacy configuration of CodeSonar was leveraged from GSFC Flight Software Systems Branch following coding standards such as MISRA as well as JPL specific standards. Static analysis using CodeSonar is performed periodically throughout development lifecycle and all errors and warnings are addressed.

C. Unit Testing

SWE-062 asserts software shall be unit tested. Additionally, SWE-135 states code coverage metrics shall be gathered. CCRS FSW leveraged UtAssert [23] unit test framework developed at GSFC. Unit tests are automated using workflows inside GitLab and run with each FSW repository push. All unit tests must pass with 100% coverage of functions, branches, and lines before a pull request is ready for review.

D. Functional Testing

SWE-066 asserts software shall be tested against operational requirements. RSW has operational subsystem requirements developed to meet the needs of higher-level system requirements. Section V.D.1 describes a ground data system developed by GSFC to test operational subsystem requirements. Section V.D.2 provides an overview of a simulator to test RSW interactions with SE interface. Section V.D.3 gives insight into software used to monitor runtime performance.

1. Integrated Test and Operations System

The Integrated Test and Operations System (ITOS) is used as a ground data system to test operational subsystem requirements. ITOS was developed by GSFC with mission heritage beginning in 1990 with the Small Explorers project [24]. ITOS is a comprehensive C&T solution that supports flight operations, integration and testing, and development.

ITOS provides an application programming interface (API) for defining C&T structures as well as an API to develop mission specific graphical user interfaces for sending commands and viewing telemetry. RSW C&T is integrated into ITOS. ITOS display pages were created for viewing telemetry and plots were developed for monitoring FSM transitions.

ITOS also supports scripting commands using the Systems Test and Operation Language (STOL). FSTOL scripts could be tested in ITOS as STOL scripts first. STOL and FSTOL scripts are used to run higher level behaviors using motion primitive commands. Integration tests are scripted using STOL and run with each pull request to ensure system integrity.

2. Sensor Electronics Interface Simulator

The Sensor Electronics Interface Simulator (SEIS) is a lightweight simulation meant to interact with RSW as a substitute for RSCE used in-flight. SEIS is written in the C++ programming language following the interface control document governing the communication between RSW and RSCE. Communication between RSW and SEIS is done using User Datagram Protocol (UDP) [25] instead of RS422. SEIS includes an internal state machine matching the modes of RSCE. SEIS sends motion telemetry packets at 10Hz containing fault status, 10 samples of 100Hz high-rate data, and 1 sample of low-rate 10Hz data. Additionally, SEIS is designed to ingest commands from RSW and produce command responses back. Figure 14 shows the interfaces of RSW when running with on desktop Linux with the SEIS.

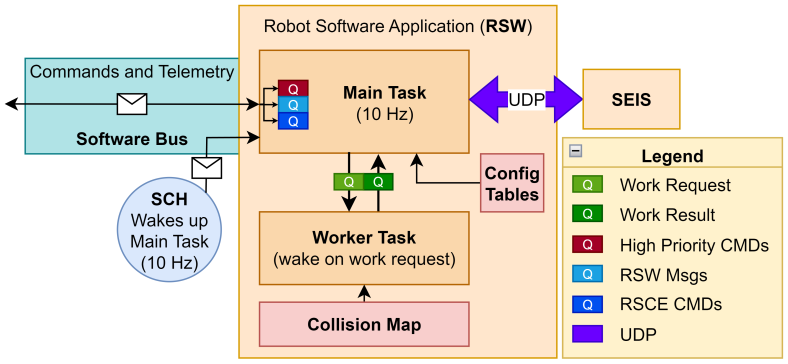


Fig. 14 RSW Linux Configuration

SEIS is integrated into ITOS. SEIS is launched when ITOS is run in a desktop configuration to enable comprehensive testing of RSW. Part of ITOS integration included adding support to SEIS for receiving commands directly from ITOS. ITOS commands supported by SEIS include Start, Stop, and No-Operation. A set of verbose modes were implemented into SEIS to allow for different levels of logging during debugging.

3. Performance Monitoring

Performance monitoring of RSW is accomplished with cFE's performance collection API which provides precise timing information. Modifications to cFE performance monitor API were made to interface with a Saleae logic analyzer and Saleae Logic 2 software [26] when CCRS FSW is compiled for VxWorks. Collections were analyzed using Software Timing Analyzer (STA) software provided with the open-source distribution of cFE [27]. Performance collections were made periodically throughout development to maintain benchmarks.

STA can produce tables of time and "% CPU Util" to provide insight into hardware timing results by considering any active signal to represent ongoing computation. "% CPU Util" can be misleading if there is system processing that is not coordinated perfectly with hardware timing signals. Instead, this paper will define a more precise definition of "% CPU Util" as $SYSTEM_{approx-util}$, shown in Equation 1, where $S_i[t_k]$ is a hardware timing signal for cFS Application i at time t_k. k is used to index time samples. In STA, Δt is set to a default of $500ms$.

$$SYSTEM_{approx-util} = \frac{100}{N}\sum_{k=1}^{N}\sum_{i=1}^{n} S_i[t_k] \tag{1}$$

STA also produces metrics for each signal including minimum, maximum, and average width of each signal. Width is intended to capture runtime of each process. Additionally, STA provides minimum, maximum, and average period of each signal. Period is intended to represent time between the start of each process.

VI. Analysis

Analysis of image size and performance of RSW's preliminary design were performed to verify compliance with operational requirements. For analysis presented here, CCRS FSW was compiled and run on a, commercial off the shelf (COTS), Gaisler GR712 Development Board [28] which has the same processor chip as MUSTANG Processor Card selected for flight. Section VI.A presents image size analysis of RSW with comparisons to other cFS applications. Section VI.B analyzes process runtime of RSW when at idle and when executing a motion primitive.

A. Image Size Analysis

Given memory limitations of MUSTANG and GR712, image size of RSW was required to be less than 1 MB. Appendix Table 10 includes all cFS Applications in CCRS FSW and their respective image sizes when compiled for GR712. Image size of RSW compiled for GR712 is 95.176 KB.

RSW is the second largest cFS Application in CCRS FSW. When compiled for GR712, RSW is more than three times the size of other mission apps developed specifically for CCRS: SPWR, SPWN, and TC. When compared to the average image size of open-source cFS Applications included in CCRS FSW, 38.685 KB, RSW is 2.46 times larger. RSW is 9.37 times smaller than the largest cFS Application in CCRS FSW, FSTOL.

When compiled for VxWorks, the image of RSW is a single object file. Compiling RSW for Linux gives insight into which source files are the largest as object files are created for every source file. The RSW image size when compiled for Linux is 125.3 KB. Out of the 125.3 KB total, definitions and processing functions for RSCE Commands occupy 52.4 KB or 41.82% of the image. Object files for Main Task and FSM implementations occupied 28.5 KB or 22.74% of the image.

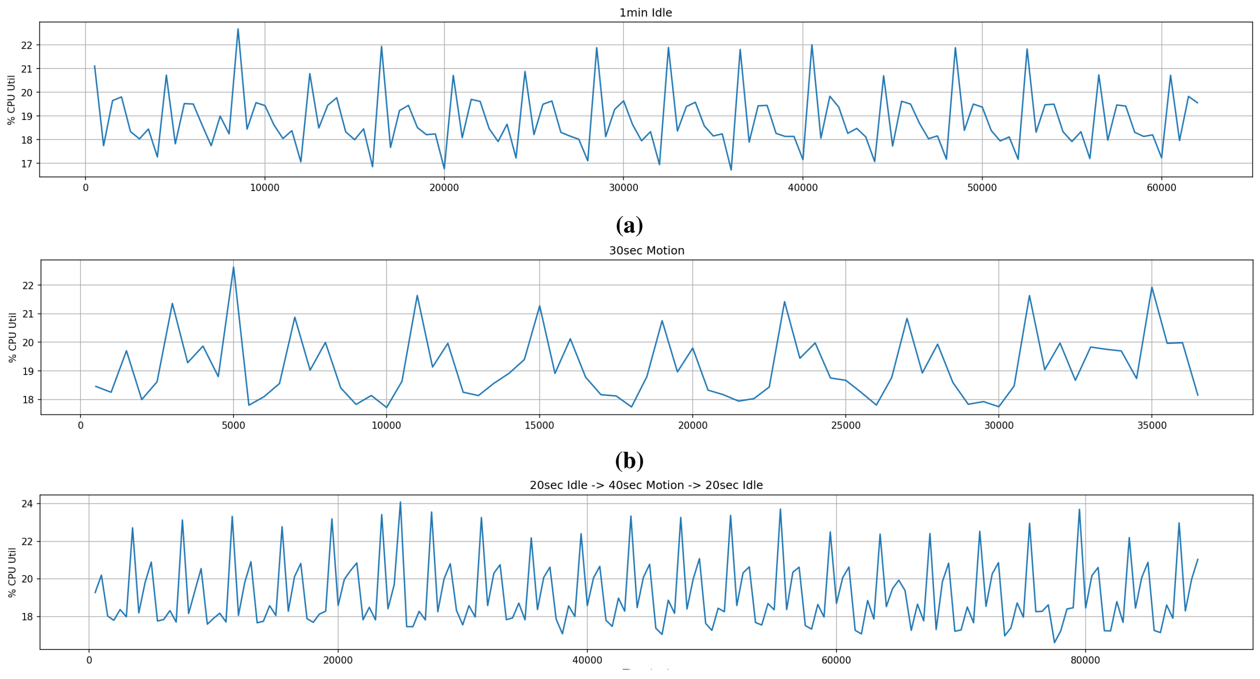
B. Performance Analysis

To test the preliminary design of RSW, three tests were conducted to gain insight into the behavior of RSW when executing a motion primitive command. For every test, CCRS FSW was running on the COTS board and an SE emulator was connected to RSW using an RS422 UART interface. The SE emulator produced CCSDS Telemetry packets, was receptive to CCSDS Commands, and was connected to a motor for hardware-in-the-loop testing. cFE performance entry and exit logs were made to track the runtime and interval of Main Task. Worker Task was implemented with baseline functionality to test communication between cFE Tasks. Table 2 includes brief descriptions of each test.

In Test A, RSW was left to idle with no commands injected for 60 seconds. Test B was focused on the behavior of RSW when performing an Open Loop Actuator Motion Primitive Command with a duration argument of 30 seconds. Test C captured 20 seconds at 20 idle before sending RSW an Open Loop Actuator Motion Primitive Command with a duration argument of 40 seconds followed by 20 seconds of idle. STA was used to extract tables of time and $SYSTEM_{approx-util}$ from Saelae Logic 2 data. Figure 15 include plots of $SYSTEM_{approx-util}$ for each test.

Test	Description
A	1 minute of idle
B	30 seconds of motion
C	20 seconds of idle followed by 40 seconds of motion followed by 20 seconds of idle

Table 2 RSW Test Descriptions



(a)

(b)

(c)

Fig. 15 $SYSTEM_{approx-util}$ **for Test A (a), Test B (b), and Test C (c)**

STA was also used to analyze Main Task runtime, denoted by τ_{RSW}, and period, denoted by T_{RSW}. Equation 2 was developed to approximate the average runtime of Main Task, denoted by $\text{RSW}_{\text{approx-util, avg}}$. Additionally, equation 3 was created to approximate the worst-case performance of Main Task, denoted by $RSW_{\text{approx-util, max}}$. Table 3 shows $\text{RSW}_{\text{approx-util, avg}}$ and $RSW_{\text{approx-util, max}}$ calculated for Tests A, B, and C.

$$\text{RSW}_{\text{approx-util, avg}} = \frac{\tau_{\text{RSW, avg}}}{T_{\text{RSW, avg}}} \times 100\% \tag{2}$$

$$\text{RSW}_{\text{approx-util, max}} = \frac{\tau_{\text{RSW, max}}}{T_{\text{RSW, min}}} \times 100\% \tag{3}$$

Test	$\text{RSW}_{\text{approx-util, avg}}$	$\text{RSW}_{\text{approx-util, max}}$
A	3.309%	6.98%
B	3.749%	9.92%
C	3.378%	23.452%

Table 3 $RSW_{approx-util}$% **Average and Worst Cases**

VII. Discussion

The preliminary design of RSW—RTAS command and control software—was presented. Section VII.A discusses RSW analysis results. Section VII.B describes a reimplementation of RSW from C++ to C and a supporting trade study. This paper ends with section VII.C introducing work left to improve design and analysis of RSW.

A. Analysis Results

Although RSW image size is relatively large when compared to the average cFS application, RSW image size is within requirements by a factor of more than 10. The relatively large size of RSW can be explained by the amount of RSCE Commands and the required data structures and associated processing functions needed. RSCE Commands, which provide a pass through to SE, are critical components of redundancy.

Performance analysis of RSW shows $\text{RSW}_{\text{approx-util, avg}}$ is less than 4% in all recorded tests. RSW shows significant differences between $\text{RSW}_{\text{approx-util, max}}$ and $\text{RSW}_{\text{approx-util, avg}}$, especially in Test C. Visual inspection of Figure 15 shows steady system performance across cycles; although, there is an increase in $SYSTEM_{approx-util}$ when a motion primitive command is received by CCRS FSW secondary to increased RSW uptime.

B. Redesign from C++

Initially, RSW's implementation leveraged a heritage C++ code base from OSAM-1. The C++ implementation of RSW had a 1.314 MB image size after optimizations. Further analysis of compiled images showed 214.81 KB could be saved by not including C++ standard library. Given size requirements of 1 MB and need for additional space for new features, a trade study, shown in Table 4, was performed to evaluate the development cost of pivoting to an implementation in the C programming language or modifying the existing C++ implementation.

Chosen metrics for the C vs. C++ trade study were: complexity, development effort, flight heritage, language knowledge, memory usage, performance, safety, and tooling. Weights for each metric were chosen subjectively by the team on a scale of 1 to 10. After team discussion, values were assigned for each metric-language pairing with 10 being the best score.

Metric	Weight	C	C++
Complexity	4	9	7
Development Effort	8	2	8
Flight Heritage	8	10	6
Language Knowledge	4	8	8
Memory Usage	9	10	3
Performance	10	9	10
Safety	10	8	10
Tooling	8	10	10
Total	610	504	479
Percentage		82.62%	78.52%

Table 4 C vs. C++ Trade Study

Complexity relates to the inherent complexity of a programming language and consequent complexity of programs written in it. A language with simpler syntax and fewer features might be easier to understand and debug but also require more code for complex tasks. Complexity also affects the maintainability of the code. C programming language's simplicity makes it more predictable and easier to reason about. On the other hand, C++ is inherently more complex due to the extensive feature set.

Development Effort relates to the amount of time and resources needed to develop or modify RSW. A higher development effort score means less development effort. C received a low score because it meant starting from nothing. C++ received a higher score because we had an existing C++ implementation of RSW. Development effort was weighted high due to limited resources.

Flight Heritage measures the programming language's track record in space flight applications. Languages that have been successfully used in space flight applications are preferred. C received a higher score because cFS Applications have traditionally been written in C. C++ also received a high score because RSW was leveraging an existing OSAM-1 code base.

Language Knowledge was used to evaluate the development team's knowledge of each language. Since the development team was equally knowledgeable on C and C++, both languages received the same rating.

Memory Usage relates to the size of RSW when compiled on target hardware. Smaller executables can also runner faster due to better cache utilization, and they consume less storage space, which is a crucial factor in space-constrained environments. C received a high score because other cFS applications written in C do not have the same image size problems. C++ received a low score because even without using the standard library, the RSW image size exceeded requirements and there were remaining features to be implemented.

Performance relates to CPU usages. Performance is often evaluated using benchmarks that measure the time required to execute standard tasks. Both C and C++ were given high scores; although, C++ was given a higher score due to optimizations of C++ libraries over C.

Safety relates to the ability of a programming language to prevent, detect, and handle errors. C received an 8 because it is weakly typed. C++ has stronger type checking and features like Resource Acquisition Is Initialization (RAII) that can prevent common bugs that might be harder to catch in C.

Tooling compares tools available for the programming languages such as static code analyzers and testing frameworks. Both C and C++ received perfect scores in the trade study because both C and C++ have static analysis tools that meet NPR 7150.2D standards. Also, C and C++ can both be integrated into the UtAssert unit test framework.

The trade study resulted in a reimplementation of RSW from C++ to C being advantageous.

C. Future Work

Following a September 2023 report by an Independent Review Board [29], NASA Office of Inspector General released an audit of the Mars Sample Return Program which concluded "before the MSR Program is approved to proceed from formulation into development, viable alterantives to the Program's mission architecture considered – including mission launch and sample return alternatives" [30]. As a consequence, development on CCRS has been stopped. The design described here was the current state of RSW when work stopped. This section provides insight into work to go to transition RSW from a preliminary design to a critical design.

First off, performance monitoring of RSW currently only includes monitoring the runtime of Main Task. Adding performance monitoring at a lower level, such as in each function, would provide more insight into problem areas of RSW. Analysis shows RSW has increased runtime when facilitating a Motion Primitive Command. One likely cause is sending and/or receiving packets on the UART interface; still, it is unclear without lower level performance monitoring what part of Motion Primitive Command execution is causing the most processing load.

RSW is designed to support Motion Primitive Commands and uses FSTOL to support RTAS behaviors. Given large margins of image space remaining for RSW and benefits to the high frequency of RSW at 10 Hz compared to FSTOL at 1 Hz, modifications to RSW were being traded to incorporate a FSM to facilitate RSW Behavior Commands without FSTOL. One implementation possibility to achieve RSW Behavior Command support is modifying FSM to be hierarchical.

Lastly, there are open questions on the efficacy of having Worker Task. At the time of work stoppage the collision detection algorithm was under trade, but consensus was leaning towards a static mapping of collision zones in the form of a lookup table. If collisions were predicted using a static data structure, there is no need to offload the processing to a separate Worker Task. There may be benefits though to utilizing Worker Task to offload communication with SE to a separate task. Additional performance monitoring needs to be implemented first to inform Worker Task design decisions.

Appendix

Application	Acronym	Description
Limit Checker	LC	Monitors telemetry and produces event messages when telemetry is found outside of pre-defined thresholds [12]
Stored Commands	SC	Produces sequences of stored commands using a time table [13]
Scheduler	SCH	Generates SB messages at pre-determined timing intervals [14]
Housekeeping	HK	Builds and sends combined telemetry messages from cFE Applications in order to minimize downlink telemetry bandwidth [15]
Heath and Safety	HS	Application monitoring, event monitoring, and hardware watchdog services [16]
CFDP	CF	CCSDS File Delivery Protocol (CFDP) CCSDS 727.0-B-5 compliant services. [31]
Checksum	CS	Ensures integrity of onboard memory using Cyclic Redundancy Checks [32]
File Manager	FM	Onboard file system management services [33]
Memory Manager	MM	Used for loading and dumping system memory [34]
Memory Dwell	MD	monitors memory addresses accessed by the CPU [35]
Flight-STOL	FSTOL	Runs STOL scripts onboard to execute logic and make decisions
Processor Card Housekeeping	PCHK	Provides read, write and read-mod-write access to FPGA and DLEON memory spaces
Telemetry Output	TO	Provides telemetry filtering capabilities for SPWR
Robot Software	RSW	Controls and monitors RTAS
SpaceWire Router	SPWR	Packet routing from SpW hardware to SB
SpaceWire Node	SPWN	Provides SpW hardware node connections to SpW endpoints
Thermal Controller	TC	Responsible for commanding all operational heaters aboard CCRS

Table 5 CCRS FSW cFS Applications

Command	Description
Category 0 Stop	Uncontrolled stop with immediate power removal. Puts RSW in fault state.
Category 1 Stop	Controlled stop. Puts RSW in fault state.
Clear Fault State	Clears RSW's fault state.

Table 6 High-Priority Commands

Command	Description
NOOP	No-Operation Command
Stop	Send a stop command to RSCE if in motion. Transition RSW to Ready-On state if possible
Reset Counter	Reset counters within RSW HK Telemetry
Verify Docking Post Compression	Verifies the docking posts were compressed.
Verify EE Docked	Verify the end effectors were docked.
Position Servo	Position Servo Motion Primitive Command
Open-Loop Actuator	Open-Loop Actuator Motion Primitive Command
Active Limp	Active Limp Motion Primitive Command
Active Hold	Active Hold Motion Primitive Command

Table 7 RSW Commands

Command	Description
Noop	Maintains communication with RSCE but does not require a response.
Stop	Disables power to the motor phases and the friction brake.
Select Actuator	Select Active Actuator.
Select ASCE	Select ASCE to Operate.
Configure PIDs	Set controller Gains
Configure Controller	Set current scheduling, stall detection, and other low-level controller parameters
Configure Actuator	Set constants for a given actuator.
Configure Motion Parameters	Set speed, tolerance, motion type, and target position.
Configure Goal Parameters	Set parameters used to signify completion of motion.
Configure Position Hold	Set parameters such as feedforward torque.
Configure Fault Management	Set limits for fault management.
Clear Stop	After a stop command, a clear stop command was needed to clear the ASCE registers necessary for motion to be restarted.
Configure Load Sensor	Set load sensor calibration tables.
Configure Resolver	Set resolver calibration parameters needed by RSCE.
Configure Feed Stage Halls Sensor	Set halls position sensor parameters.

Table 8 RSCE Commands (Part 1 of 2)

Command	Description
Configure Thermal Model	Set thermal model.
Configure Potentiometers	Set potentiometer calibration parameters.
Load Sensor Tare	Define tare values per channel.
Load Channel Sensor Mask	Define load channels to use or ignore.
Potentiometer Mask	Define potentiometer to use or ignore.
Feed Stage Halls Sensor Mask	Define halls positions sensors to use or ignore.
Contact Switch Mask	Define torque mechanism or gripper limit switches to use or ignore.
Sensor Enable	Turn excitation to a sensor or sensor channel on.
Sensor Disable	Turn excitation to a sensor or sensor channel off.
Execute Joint Motion	Execute joint motion.
Active Limp	Open break and allow back driving.
Active Limp	Open break and hold position actively.
Peek	Read a register.
Poke	Write to a register.
Register Dump	Return RSCE register content from a specified range of addresses.

Table 9 RSCE Commands (Part 2 of 2)

Application	COTS Image Size
LC	34.508
SC	40.532
SCH	20.280
HK	12.720
HS	34.460
CF	81.816
CS	63.476
FM	41.396
MM	32.148
MD	25.512
FSTOL	892.160
PCHK	13.260
TO	16.232
RSW	95.176
SPWR	31.896
SPWN	29.844
TC	5.192

Table 10 COTS cFS Image Sizes in kilobytes

Test	Minimum	Maximum	Average	Standard Deviation
A	2.120	6.850	3.309	1.065
B	2.190	9.498	3.749	1.213
C	2.125	22.495	3.378	1.347

Table 11 RSW Width Times in milliseconds

Test	Minimum	Maximum	Average	Standard Deviation
A	98.084	101.714	99.999	0.445
B	95.752	103.602	100.001	0.835
C	95.918	104.034	100.000	0.726

Table 12 RSW Interval Times in milliseconds

Acknowledgments

The development of MSR CCRS was funded by Mars Sample Return Program. The authors would like to thank the multi-center and multi-disciplinary MSR CCRS team members who contributed to the development of RSW. Daniel Chavez-Clemente, Paulo J. Younse, and Tom Green from JPL supported RSW algorithm development and integration with RTAS. Pathways Intern Johnathan Middleton contributed to command and telemetry definitions in RSW ground and flight software. Pietro Sparacino, Josh Nichols, and Thomas Schluszas helped develop and integrate RSW's interface with RSCE. William Posey led the CCRS FSW Test Team responsible for Software Assurance.

RSW development team received tremendous support from MSR CCRS Payload Software team, led by Keegan Moore, responsible for implementation and integration of all other MSR CCRS FSW cFS apps. Rich Landau integrated FSTOL and helped with RSW testing using FSTOL. Jaiden Ferraccioli modified cFE performance monitor to interface with a Saleae logic analyzer. Sarah Garcia-Beech gave valuable insights on UtAssert and architecture decisions. Alan Cudmore supported VxWorks integration and C++ vs. C trade study. Additional valuable contributions to RSW development were made by Sean Malone and Isabel Dailey.

References

[1] Muirhead, B. K., Nicholas, A., and Umland, J., "Mars Sample Return Mission Concept Status," *2020 IEEE Aerospace Conference*, 2020, pp. 1–8. https://doi.org/10.1109/AERO47225.2020.9172609.

[2] Farley, K. A., Williford, K. H., Stack, K. M., Bhartia, R., Chen, A., de la Torre, M., Hand, K., Goreva, Y., Herd, C. D., Hueso, R., et al., "Mars 2020 mission overview," *Space Science Reviews*, Vol. 216, 2020, pp. 1–41.

[3] Muirhead, B. K., and Karp, A., "Mars sample return lander mission concepts," *2019 IEEE Aerospace Conference*, IEEE, 2019, pp. 1–9.

[4] Sarli, B., Gough, K., Hagedorn, A., Bowman, E., Rondey, J., Yew, C., Neuman, M., Green, T., Lin, J., Cataldo, G., et al., "NASA Capture, Containment, and Return System: Bringing Mars Samples to Earth," *34th International Symposium on Space Technology and Science*, 2023.

[5] Green, C., Haghani, N., Hernandez-Pellerano, A., Gheen, B., Lanham, A., and Fraction, J., "MUSTANG: A workhorse for NASA spaceflight avionics," *2023 IEEE 9th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, IEEE, 2023, pp. 94–103.

[6] Parkes, S., and Armbruster, P., "SpaceWire: A spacecraft onboard network for real-time communications," 2005, pp. 6–10. https://doi.org/10.1109/RTC.2005.1547397.

[7] Wilmot, J., "A core flight software system," *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Association for Computing Machinery, New York, NY, USA, 2005, pp. 13–14. https://doi.org/10.1145/1084834.1084842.

[8] Wind River Systems, "VxWorks," , 2024. URL https://www.windriver.com/products/vxworks, version 7.

[9] Goddard Space Flight Center, "core Flight System," , 2024. URL https://cfs.gsfc.nasa.gov.

[10] Wilmot, J., "A core plug and play architecture for reusable flight software systems," *2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)*, 2006, pp. 5 pp.–447. https://doi.org/10.1109/SMC-IT.2006.7.

[11] NASA, "cFE Application Developers Guide," , 2024. URL https://github.com/nasa/cFE/blob/main/docs/cFE%20Application%20Developers%20Guide.md.

[12] NASA, "Limit Checker Application," , 2024. URL https://github.com/nasa/LC/.

[13] NASA, "Stored Commands Application," , 2024. URL https://github.com/nasa/SC/.

[14] NASA, "Scheduler Application," , 2024. URL https://github.com/nasa/SCH/.

[15] NASA, "Housekeeping Application," , 2024. URL https://github.com/nasa/HK/.

[16] NASA, "Health and Safety Application," , 2024. URL https://github.com/nasa/HS/.

[17] Mudgway, D. J., *Uplink-downlink: a history of the nasa deep space network, 1957-1997*, Vol. 4227, National Aeronautics and Space Administration, 2001.

[18] *NASA Software Engineering Requirements NPR 7150.2D*, National Aeronautics and Space Administration, 2022. URL https://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=7150&s=2D, accessed: 2024-02-15.

[19] Software Freedom Conservancy, "Git," , 2024. URL https://git-scm.com/, version 2.43.2.

[20] GitLab Inc., "GitLab," , 2024. URL https://about.gitlab.com/.

[21] LLVM Project, "ClangFormat," , 2024. URL https://clang.llvm.org/docs/ClangFormat.html.

[22] CodeSecure, "CodeSonar," , 2024. URL https://codesecure.com/our-products/codesonar/.

[23] Goddard Space Flight Center, "UtAssert," , 2024. URL https://opensource.gsfc.nasa.gov/projects/ut-assert/index.php.

[24] Goddard Space Flight Center, “Integrated Test and Operations System,” , 2024. URL https://itos.gsfc.nasa.gov.

[25] Postel, J., “User Datagram Protocol,” Tech. Rep. RFC 768, Internet Engineering Task Force, Aug. 1980. URL https://www.rfc-editor.org/rfc/rfc768.txt.

[26] Saleae, Inc., “Logic 2,” , 2024. URL https://www.saleae.com/.

[27] Goddard Space Flight Center, “core Flight Executive Open Source Project,” , 2024. URL https://opensource.gsfc.nasa.gov/projects/cfe/index.php.

[28] Gaisler, “GR712 Development Board,” , 2024. URL https://www.gaisler.com/index.php/products/boards/gr712rc-board.

[29] NASA, “MSR Independent Review Board - 2 Final Report,” , 2023. URL https://www.nasa.gov/wp-content/uploads/2023/09/mars-sample-return-independent-review-board-report.pdf.

[30] NASA, “Audit of the Mars Sample Return Program,” , 2024. URL https://oig.nasa.gov/wp-content/uploads/2024/03/ig-24-008.pdf.

[31] NASA, “CFDP Application,” , 2024. URL https://github.com/nasa/CF/.

[32] NASA, “Checksum Application,” , 2024. URL https://github.com/nasa/CS/.

[33] NASA, “File Manager Application,” , 2024. URL https://github.com/nasa/FM/.

[34] NASA, “Memory Manager Application,” , 2024. URL https://github.com/nasa/MM/.

[35] NASA, “Memory Dwell Application,” , 2024. URL https://github.com/nasa/MD/.