An Overview of the Mars Exploration Rovers Flight Software

Glenn E. Reeves
MER Flight Software Architect
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109, USA
glenn.e.reeves@jpl.nasa.gov

Abstract - *The Mars Exploration Rovers (MER) Flight Software (FSW) is possibly the most complex software implementation to be deployed on another planet. The requirements dictated a software system that addressed four distinct mission phases (cruise, landing, egress, and surface) and the mission demanded a system with significant autonomy. The structure of the MER flight software is reflective of its object-oriented beginnings and the module functions are reflective of the requirements of the MER mission and spacecraft. This paper provides an overview of the function and structure of the MER flight software. The MER mission and spacecraft is briefly discussed to provide context for the flight software decomposition and several aspects of the software execution model are also discussed.*

Keywords: Mars, software, flight, rovers, autonomous.

1 Introduction

NASA's Mars Exploration Rovers (MER) project landed two rovers, Spirit and Opportunity, on Mars on January 4 and January 25, 2004, respectively. The software that controls these rovers was the product of a single development team and was developed over a period of 35 months beginning in July 2000.

The MER Flight Software inherits significant portions of structure, architecture, design, and source code from the Mars Pathfinder (MPF) software and Athena Rover software, and, to a lesser extent, from Deep Space 1 (DS-1), Mars 98 (MSP98), and Mars 01 (MSP01) missions. The architectural model was originally conceived for the MPF mission although the basis for many of the architecture features dates farther back to ground support equipment software for the Magellan and Cassini mission.

2 The MER Mission Requirements

The MER mission can be viewed as four distinct phases (or four distinct missions) separated by the physical environment and the physical changes in the vehicle configuration. These phases are Launch/Cruise, Entry/Descent/Landing, Deployment/Egress, and Surface operations. Figure 1 shows the multiple physical, configurations of the vehicle.

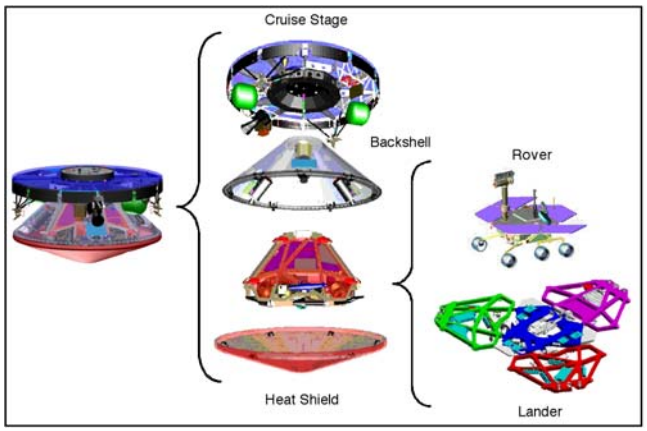


Figure 1 – MER Vehicle

The Launch/Cruise phase of the mission begins on the launch pad and terminates at the separation of the cruise stage of the vehicle. The Entry, Descent, and Landing phase starts while the vehicle is still approaching Mars, overlapping, the end of the cruise phase, and can be considered complete when the vehicle has landed successfully and has deployed itself into a power safe configuration. At the end of the EDL phase the rover is not yet ready to drive as it is in a compact mechanical configuration and still rests upon the "lander" structure. The Deployment/Egress phase requires the careful articulation of the mechanical configuration in order for the vehicle to become mobile. The phase ends with the successful egress from the lander onto the Martian surface. At this point the vehicle is ready to perform its primary, surface mission, which is to rove the planet surface, much as a geologist would, examining multiple sites using its suite of instruments.

2.1 Flight Software Responsibilities

The Flight Software is responsible for many aspects of the functionality of the spacecraft/rover. The infrastructure throughout all mission phases includes

control and management of the processor, bus and devices, command processing, sequence execution, engineering data collection, channelized telemetry, data compression, spacecraft time management, power switching control and solar array control. Celestial body position estimation (Earth, sun, Mars, etc.) is active during all phases too.

Flight software had to initiate and maintain communication with the Earth, including X-Band Direct-to-Earth (DTE) and Direct-from-Earth (DFE) throughout cruise, EDL and surface operations as well as UHF Communication to Earth during EDL and surface operations.

During cruise, propellant line thermal control was maintained by the flight software. Other cruise attitude control related capabilities included cruise attitude determination and control, axial and lateral trajectory change maneuvers, and star identification.

Almost all Entry, Descent, & Landing activities were performed by the flight software. This included turn to entry, cruise stage jettison, parachute deployment, altitude and velocity determination using radar data, retro-rocket initiation, horizontal velocity determination and damping, airbag deployment, lander separation, and landing event detection. Post landing critical activities included airbag retraction, petal deployment and solar array deployment.

After landing the FSW was responsible for controlling the mechanical actuations necessary to transform the rover from the stowed configuration to a configuration suitable for mobility. This include releasing the rover from the lander both physically and electrically, controlling the Rover Lift Mechanism (RLM) to lift the rover, and orchestrating the rotation of the front wheels to their mobility position. This multi-step process was done with ground operators in-the-loop but the FSW controlled the movement and monitored proper position and contact sensors to detect both nominal and off-nominal completion of each step.

During surface operations, the FSW was responsible for rover attitude and position determination. This capability supports bore sight pointing for the engineering and science cameras and for pointing (and tracking) Earth with the high gain antenna. The attitude and position knowledge is also a component of the mobility system as the FSW monitors tilt during motion to ensure safe orientations.

The rover mobility capability is almost entirely a responsibility of the FSW. The ground can command the vehicle to move using a suite of capabilities ranging from basic motion commands to a fully autonomous mode. In the autonomous mode the FSW takes stereo images that are processed into range and obstacle data, identifies hazards, and computes the safest path toward the goal. The FSW manages the repetitive cycle of capturing images with the front or rear fixed position cameras, determination of a safe path avoiding obstacles, and performs the necessary coordinated motor control for both steering and driving. The navigation software can also use cameras on the mast. By using the visual odometry capability, the vehicle can reach the commanded position even in the presence of slippage.

The FSW is responsible for managing the science instrument suite. This includes providing control of the instrument configurations and capturing the science data. The FSW does initial processing on the science data when commanded to do so. This includes both image compression and instrument specific data processing (Fast Fourier Transform (FFT) and phase correction for the mast spectrometer are examples).

The FSW manages the motion of the Instrument Deployment Device (IDD). The IDD is a 5 degree-of-freedom (DOF) mechanical manipulator that is mounted on the front of the rover's chassis. The IDD carries the suite of *in-situ* instruments. The primary function of the IDD is to place the instruments on targets (rocks, soil, magnets and calibration targets). The ground operator commands a desired position and selects an instrument and the FSW computes the forward and inverse kinematics, manages the necessary joint motion, and monitors the contact sensors to achieve the commanded placement.

The FSW uses a “data product” model to combine science (and engineering) data types with the metadata (vehicle attitude and position, time of day, relevant temperature and power telemetry) needed by the scientist or engineer on the ground. Data products are files stored in the onboard FLASH file system. Data products are broken into CCSDS formatted telemetry for transport and then reassembled by the ground data system. The FSW multiple data compression types for both science and engineering data. This includes: ICER, LOCO, and LZO.

On the surface the FSW also manages the coordination of activities. This is required for two significant reasons: 1) the design of the motor control hardware limited the simultaneity of activities, and 2) a mission design goal was to oversubscribe the system to maximize science return and optimize the onboard activities. The onboard software provides arbitration between potentially conflicting activities (communication using HGA and taking images for science) as well as precludes certain health and safety concerns such as trying to move/drive the vehicle while the IDD is not stowed. This coordination mechanism couples the nominal

commanded activities and the autonomous activities as well as the fault protection into a coordinated system.

2.2 Autonomous Activities

One premise behind the architectural and functional focus of the MER flight software is the recognition that many of the critical mission events must be done autonomously. The two-way light time precludes a ground in-the-loop solution. With this in mind, the development team had to decide the manner in which these "behaviors" would be implemented.

One implementation option was to use command sequences to embody all of the logic necessary for both the nominal and off-nominal (i.e. fault protection driven) scenarios. This would have required the creation of an extensive command suite allowing very precise, although primitive, control. The sequence execution mechanism would need to be significantly enhanced (beyond the MPF design) to include the necessary conditional and branching constructs and a robust rollback and reset recovery mechanism.

The second implementation option, and the one ultimately chosen, was to embed much of the required behavior in the flight software logic. Both the logic for the expected activities and their off-nominal variants were built in to the flight software. This approach largely removes the historical separation between the nominal activities of the system and the fault protection design. The approach simplified many aspects of the operational model (there was no need to build fault protection command sequences for off-nominal scenarios) and allowed for a reasonably simple sequence execution mechanism. The final architecture is one with the ground initiating the vehicle activities with “high level” commands and the FSW completing the activity even in the presence of faults.

The following list identifies some of the flight software autonomous behaviors (or supporting functions):

- Detection of launch vehicle separation, to initiate Earth communication
- J2000 pointing, including attempts at using multiple combinations of sensors and thruster branches, for emergency sun pointing during cruise and the critical turn to the entry attitude during EDL.
- Axial and lateral burn control, including retries should a reset occur.
- The entire EDL series of actions from the beginning of the approach phase (Landing minus 5 days through the successful completion of the rover solar array deployment (on the Martian surface)
- Coordinated mechanism and motor control for rover standup
- Coordinated driving that includes imaging, hazard detection and wheel movement. Fault detection and safety reactions are built into this behavior.
- Sky/sun search and identification for surface attitude determination.
- Communication including start and end times, hardware configuration, rates, and mode.

One other positive aspect of the decision to use behaviors rather than sequences was that it forced the maturity of the system design; there could be no procrastination or deferment because the design was coded in the flight software.

3 Flight Software Structure

The flight software is a collection of 93 modules. Table 1 lists the modules and briefly identifies their function. The design emphasizes interfaces, encapsulation, and modularity. As such, a modules’ existence could be for one of several reasons. Many of the modules are the result of an object-oriented analysis process bringing together objects with significant collaborations. This is true of many of the MPF heritage modules where the development team used the techniques in [1] to design the original architecture. An equal number, especially those that do not have MPF pedigree, reflect a functional decomposition of the overall system. The MER schedule largely precluded revisiting the broad object oriented analysis done originally. Some of the modules reflect a one-to-one mapping with the execution model viewpoint (i.e. they represent a task/thread). Lastly, a few of the modules exist solely to support the assignment of effort to the software development team; each of the modules had one primary developer.

There is a loose arrangement of modules into layers, though there is no enforcement. A module can theoretically communicate with any other module (only through its interfaces). In practice the interaction between modules is much more limited. The lowest layer is the Hardware Abstraction Layer and contains the simplest functions to access hardware devices. The highest layer is the behavior layer. The layers in between represent a conceptual separation between those functions necessary for all (or at least most) spacecraft and those that are increasingly mission unique or increasingly autonomous. Figure 2 identifies the layers.

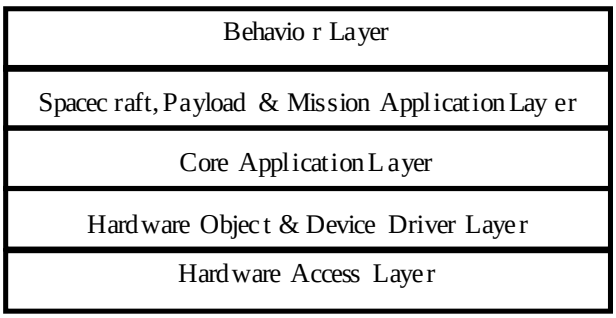


Figure 2 – FSW Layers

The Flight Software is coded primarily in ANSI C, with some targeted assembly code and some C++. The size of the system, in source lines of code (SLOC), is [300K] but this value does not include the operating system.

Although many aspects of the design are objected-oriented, the features of the language incorporating inheritance and polymorphism are not exploited. We have found that when implemented to their fullest extent in C++, these constructs result in detrimental code size and add the potential for non-deterministic behavior. These costs, especially in time and effort to achieve a robust implementation, outweigh the benefits.

4 Software Execution Model

The flight software executes on the single processor controlled by a real-time operating system. The MER FSW incorporates the VxWorks real-time operating system that supplies the basic outline of how the MER flight software operates. The flight software consists of multiple, pre-emptive, prioritized, tasks (threads) all of which run under vxWorks. There are 97 tasks in the MER flight software. The OS also provides basic facilities in addition to tasks including: time and timers, math libraries, I/O, files, logging, message queues, and semaphores (and many others). The operating system expects a real-time clock tick to drive the system clock and this is supplied as an interrupt from the MER specific hardware (versus from the computer board clock).

4.1.1 Tasks

The relationship between modules and tasks is not one-to-one. Several modules are libraries. There are a number of tasks that have common source code (sixteen sequence machine tasks) and some modules have multiple tasks (there are thirteen fault protection response tasks). In many cases, a task represents a service that has a functional responsibility, and a response-time requirement. For example, the *acs* task encapsulates all of the cruise attitude estimation and control function with a rate requirement of 8hz. It also processes commands and produces telemetry related to attitude estimation and control. Not all of the tasks have functional responsibility. For example, the *bcp* and *btp* tasks provide execution contexts for the processing of low priority commands and low priority telemetry for several devices and functions. Similarly, the *bc* task (for 1553 resident devices) and the *dat* tasks' sole purpose is to provide an active context for the hardware objects[1] to perform their data acquisition.

Each task is given a priority commensurate with its required response time, and there are very few changes to this priority during the mission. There are both cyclic (time event driven) and sporadic (event-driven) tasks in the system. Tasks execution is driven by the arrival of messages via the ipc system (see below) and do not poll. Interrupts produce a message that wakes the task that services the interrupt.

4.1.2 Inter-Process Communication

The principal mechanism for task communication is by the facilities provided in the *ipc* and *reply* modules. The inter-process communication is message based and implemented using VxWorks pipes. Messages are point-to-point – there is no broadcast and every task in the system has at least one message queue. When a client task sends a message to a server task, it calls a void-return method defined in the server module using a direct function call. This method will execute in the client's task context but results in putting a message on one of the server task's queues. Note that the server defines which queue is used and the format of the message itself. This approach keeps such information local to the server and changes here do not impact the client's code.

Each queue represents a series of messages in arrival time order. By assigning messages to queues, each queue can be treated as being at a particular priority, or specific queues can be ignored depending on the state of the task. Each queue is of fixed length and can hold a given maximum number of messages. Each message in a queue is made to consume the same amount of space. Interrupt handlers, timers, and commands also result in a message. The goal is for every interaction between tasks, between a service (like timers) and a task, and between an interrupt handler and a task be through an IPC message. Figure 3 graphically shows this model.

However, such is not always the most practical method of communication. Thus there are instances of shared memory interaction, either using semaphores, mutual-exclusion controls, or sometimes via no synchronization methods at all. One is the use of a double

[1] The hardware object pattern encapsulates all attributes and interfaces of the hardware device it represents. One public interface is the service_hw() method which extracts data from the hardware.

buffer, shared memory pattern in the data collection pipeline. Data consumer tasks access the shared memory via a public interface owned by the representative hardware object. The actual data extraction from the hardware interfaces occurs during the *dat* and *bc* task execution.

The ipc architecture includes a “reply” pattern that allows a client to request a callback via one of the client’s public interfaces. The original intention of this addition to the original MPF ipc system was to provide a synchronization mechanism to the event oriented command sequence mechanism and similar synchronization mechanism to be used by the spacecraft configuration management function. These two functions need to both wait for prior requests to complete but to also know the success or failure of the action.

Table 1 – MER FSW Modules

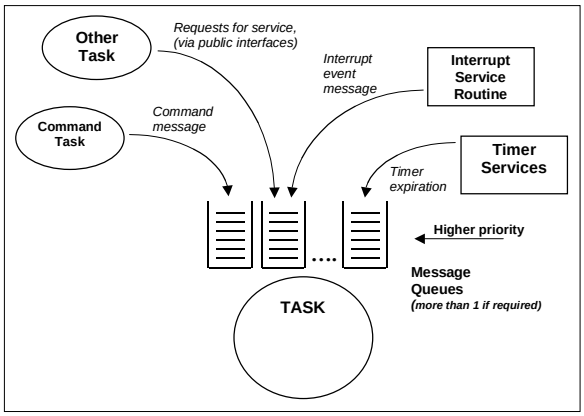
Module	Description
accm	Alarm clock and wakeup function
acl	Alarm Clock Hardware Object
acm	Activity Constraint Manager
acs	Attitude Control System
acsutils	ACS Math Utilities
adc	Analog Data Collection
aman	Auto Maneuver
arb	Resource Arbitration Control
bc	1553 Bus Control
bcp	Background Command Process
bsp	Boot and vxWorks initialization
btp	Background Telemetry Process
catbed	Catbed Heater Hardware Object
cbm	Communication Behavior
cmd,cmd_valid	Command translation and dispatch
comp	Compression Services
cpu	R6K SBC Services
crc	Critical Relay Control
csid	Cruise Stage Interface (HO)
dat	Non-1553 Data acquisition
ddi	Sequence variables
dimes	Descent trajectory estimation
drive	Basic driving services
dsa	Digital Sun Sensor (HO)
dwn	Downlink (XBAND and UHF)
edl	EDL Behavior Manager
eep	EEPROM Management
eha	Channel telemetry
eheap	Extended Memory/Heap Manager
evr	Event Record Telemetry
fbm	Fault Behavior Manager
files	File System Services
fme	File Metadata (Data Product)
fswld	Flight Software Load and Patch
globals	Constant global information
hal	Hardware Access Layer
health	FSW health determination
heap	Downlink heap services
hga	HGA Pointing
hst	Data history/ring buffer manager
idd	Instrument Deployment Device
idle	idle task, statistics collection
iit	IMU and Inertial Vector Propagation
img	Imaging services
imu	Inertial Measurement Unit (HO)
init	Initialization

Module	Description
init	Initialization
ipc	Inter-process communication
latch_valve	Latch Valve Hardware Object
lsid	Lander Stage Interface HO
math	Math Utilities
mcas	Motor Board Analog Scanner
mcl	Mission Clock Services
mem	Memory allocation services
mfsk	EDL tone telemetry service
mobm	Mobility Behavior Manager
mot	Motor control
mrf	Mirror RAM to Flash
nav	Surface Navigation
nvm	Non-Volatile Memory services
pas	Payload Analog Scanner (HO)
pdp	Prioritized Data Products
perf, prf	FSW Performance measure
pma	Pancam Mast Assembly
pty	Standard I/O Capture
pwr	Power device management
pyld	Science Instrument Control
pyro	Pyro event management
ras	Radar Altimeter System (HO)
rat	Rock Abrasion Tool
reply	Inter-process communication
reu	Remote Engineering Unit (HO)
rfr, rfs	Radio Subsystem Services
sam	Structures & Mechanisms
sapp	Surface Attitude & Position
scm	Spacecraft Config Manager
sdst	XBAND Transponder (HO)
seq	Sequence
ssa	Star Scanner Hardware Object
sspa	Solid State Amplifier (HO)
standup	Standup behavior
tffs	FLASH file system support
thermal	Thermal control
thruster	Thruster Hardware Object
tim	Timing Services
uhft	UHF Radio hardware object
upl	Uplink
util	Utility command service
vis	VME Interrupt Services
vsbm	Shutdown behavior manager
vxWorks	VxWorks OS
	* HO – Hardware Object

4.2 Rate Groups and Task Priorities

There are no rate groups per se. Instead, a task with rate requirement subscribes to a timer service (the *tim* module) that delivers messages at the correct rate. The message is received by the task subscribing to the service, and at every interval, that task's event loop gets the message and performs the work specified by its current state. The time event message is usually a higher-priority message (i.e. these messages are delivered to an ipc queue that is serviced first), so if there are other messages for the same task, the higher priority work is performed first. That is, one may look at commands and other events as executing between time event messages. It is possible to subscribe to more than one time event. The *tim* module provides event notifications for time periods synchronized to the spacecraft time as well as for relative durations (i.e. 10 seconds from now).

Each task has an execution priority, with higher-priority (lower priority number) going to those tasks with greater criticality. In the MER implementation several tasks wait for the arrival of the 8Hz time event. The priority of the task dictates the order that processing occurs.



4.3 Figure 3 - Messages, Queues, and Task Architecture

Many tasks do not have a rate requirement and are entirely event driven. This event can be the reception of a message from an interrupt handler, the arrival of a message indicating a command, or a request for action from another task. The priority of tasks in this category is dictated by the relative importance of their primary function. For example, the sequence engine tasks are higher priority than the image processing tasks (command execution is higher priority) but lower than the *cbm* task that controls the semi-autonomous communication (communicating is higher priority).

5 Conclusion

The MER flight software is a complex product that was developed on a very tight schedule. However, the system and software have performed very well throughout the entire mission. The design decomposition into modules reflects the system and mission requirements. The assignment of tasks under the real-time OS is reflective of the separation and priority of the different and varied system activities. The software enforces encapsulation by requiring module-to-module interaction only via public interfaces and the message based inter-process communication enforces this separation in the execution model.

6 Acknowledgments

The success of the MER mission is a testament to the dedication and tenaciousness of the combined team of management, system engineers, flight software developers, and test engineers who designed and built these spacecraft. The author wishes to thank the flight software team in particular for their professionalism, patience, and unswerving focus on building a quality product. The author would also like to thank everyone whose material is contained in the MER Flight Software Baseline Architecture and Design Document (FSBADD) from which a significant portion of this paper is based. In particular, the author wishes to thank Joe Snyder and Ed Kan for bringing that document to fruition and Kim Gostelow, Mary Lam, and Tracy Neilson who provided significant assistance in the preparation of this paper.

The flight software development described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

7 References

[1] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Weiner, Designing Object-Oriented Software, PTR Prentice-Hall, New Jersey, 1990

[2] Joseph F. Snyder, Edwin P. Kan, the MER FSW Team, "MER Flight Software Baseline Architecture and Design Document", Jet Propulsion Laboratory Document, December 21, 2001,

[3] Kim Gostelow, Mary Lam, "MSL FSW Architecture Description", Jet Propulsion Laboratory Document, February 2005,