Mars Science Laboratory Frame Manager for Centralized Frame Tree Database and Target Pointing

Won S. Kim, Chris Leger, Stephen Peters, Joseph Carsten
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA, USA
Won.S.Kim@jpl.nasa.gov

Antonio Diaz-Calderon
Johns Hopkins University
Applied Physics Laboratory
Laurel, MD, U.S.A.
Antonio.Diaz-Calderon@jhuapl.edu

Abstract - *The FM (Frame Manager) flight software module is responsible for maintaining the frame tree database containing coordinate transforms between frames. The frame tree is a proper tree structure of directed links, consisting of surface and rover subtrees. Actual frame transforms are updated by their owner. FM updates site and saved frames for the surface tree. As the rover drives to a new area, a new site frame with an incremented site index can be created. Several clients including ARM and RSM (Remote Sensing Mast) update their related rover frames that they own. Through the onboard centralized FM frame tree database, client modules can query transforms between any two frames. Important applications include target image pointing for RSM-mounted cameras and frame-referenced arm moves. The use of frame tree eliminates cumbersome, error-prone calculations of coordinate entries for commands and thus simplifies flight operations significantly.*

Keywords: Frame tree, Coordinate transform, quaternion, flight software, target pointing.

1 Introduction

Based on operational experiences with MER [1] and Phoenix [2] flight missions, one of the highly desired enhancements identified was the centralized coordinate transform database maintained on board. Without the coordinate transform database, there are quite a few operations that require cumbersome, error-prone manual calculations on the ground. Examples include: 1) pointing a mast camera to an arm tool or an object and 2) moving the arm relative to an object frame. MSL is the first Mars surface mission system that has the onboard centralized coordinate transform database describing spatial relations between various tools, instruments, and subsystems.

To describe the spatial relation between two objects, we generally define the object frame (Cartesian coordinate system) fixed to each object and then use the coordinate transform between the two object frames. When there are a large number of spatial relations to describe from many objects (parts) in a large mechanical system, it is natural to maintain the spatial relations in a hierarchical tree data structure [3] by partitioning the system into subsystems, which are further partitioned into sub-subsystems and so on. The MSL FM (Frame Manager) FSW (flight software) module maintains the frame tree database containing coordinate frame transforms between tree nodes. Each node represents an object frame. Some client modules update the frame transforms for a group of frame tree nodes if they are the owner of the group. In general, client modules can query the coordinate frame transforms between any two nodes onboard.

The coordinate frame transform consists of the rotational and translational parts. Since the rotation has two distinct conventions, Section 2 clarifies the conventions to avoid confusion. Section 3 defines the frame transform employed in the MSL FM module. The FM frame tree database, synchronous access of the rover position and attitude data using double buffers, and rover motion counter are described in Sections 4, 5, and 6, respectively. Section 7 presents the FM state machine and an example a sequence diagram. Section 8 lists operational usage examples.

2 Two Conventions for Rotation

There are two conventions in describing a rotation: rotation of axes and rotation of the object relative to fixed axes. Let us consider a z-axis rotation by θ. In the "rotation of the object" convention, we rotate a vector (x, y, z) about the z-axis by θ in the fixed coordinate system and compute the new transformed vector (x', y', z') by

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \mathbf{R} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{1}$$

In the "rotation of axes" convention, we rotate the coordinate system while the vector is fixed relative to the original coordinate system. The transformed vector expressed in the rotated coordinate system results in an inverse rotation compared to the former method. Namely,

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \mathbf{C} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2}$$

$$\mathbf{C} = \mathbf{R}^{-1} = \mathbf{R}^{t} \tag{3}$$

where $\mathbf{R}^{-1}$ and $\mathbf{R}'$ denote the inverse and the transpose of the rotation matrix $\mathbf{R}$, respectively.

Similarly, the quaternion rotation can be defined in two conventions. A unit quaternion describing a rotation by an angle θ about a unit vector $\mathbf{u} = (u_x\ u_y\ u_z)$ is given by

$$\begin{aligned}\mathbf{q} &= \mathbf{u}\sin(\frac{\theta}{2}) + \cos(\frac{\theta}{2}) \\ &= (u_x \sin(\frac{\theta}{2})\ u_y \sin(\frac{\theta}{2})\ u_z \sin(\frac{\theta}{2})\ \cos(\frac{\theta}{2})).\end{aligned} \tag{4}$$

The scalar part is written as the last element here. In the "rotation of the object" convention, a quaternion rotation of a vector in a fixed reference frame is given by

$$\mathbf{v}' = \mathbf{q}\,\mathbf{v}\,\mathbf{q}^{-1} = \mathbf{q}\,\mathbf{v}\,\mathbf{q}^{*} = \mathbf{R}\,\mathbf{v}\,, \tag{5}$$

where $\mathbf{q}^{-1}$ and $\mathbf{q}^{*}$ denote the inverse and the conjugate of a unit quaternion $\mathbf{q}$. On the other hand, in the "rotation of axes" convention, the reference frame is rotated by a quaternion rotation, and the vector transformed to the rotated frame is given by

$$\mathbf{v}' = \mathbf{q}^{-1}\,\mathbf{v}\,\mathbf{q} = \mathbf{q}^{*}\,\mathbf{v}\,\mathbf{q} = \mathbf{C}\,\mathbf{v}\,. \tag{6}$$

When an equivalent rotation matrix is requested for a quaternion, the "rotation of the object" convention yields **R,** while the "rotation of axes" convention yields **C.** An easy way to check the relation is to test with a quaternion rotation about the z-axis by 90º. In the "rotation of the object" convention, the conversion to the rotation matrix yields **R**.

$$\mathbf{q} = \begin{pmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \Leftrightarrow \mathbf{R} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$

On the other hand, in the "rotation of axes" convention, the same quaternion yields **C**.

$$\mathbf{q} = \begin{pmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \Leftrightarrow \mathbf{C} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

Although some quaternion library packages or extended tool boxes try to support both conventions of (5)-(8), the MSL math flight software supports the "rotation of axes" convention of (6) and (8) only.

We often assign a coordinate system (frame) to an object to describe the object position and orientation (attitude). After the rotation of an object frame which was initially aligned with the reference frame, the three column vectors of the rotation matrix represent the unit vectors of the rotated frame relative to the fixed reference frame.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \mathbf{R}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} | & | & | \\ u_x & u_y & u_z \\ | & | & | \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{9}$$

This relation can be easily understood by noting that **R** transforms (1, 0, 0) to u_x, (0, 1, 0) to u_y, and (0, 0, 1) to u_z. In the "rotation of axes" convention, three row vectors represent the unit vectors of the rotated frame relative to the fixed reference frame.

$$\mathbf{C} = \begin{bmatrix} - & u_x & - \\ - & u_y & - \\ - & u_z & - \end{bmatrix}. \tag{10}$$

Now we consider two consecutive frame rotations from A to B and then from B to C. In the "rotation of the object" convention, from (1)

$$\mathbf{v}_A = {}^A\mathbf{R}^B\ \mathbf{v}_B = {}^A\mathbf{R}^B\ {}^B\mathbf{R}^C\ \mathbf{v}_C = {}^A\mathbf{R}^C\ \mathbf{v}_C\,, \tag{11}$$

where $\mathbf{v}_A$, $\mathbf{v}_B$, and $\mathbf{v}_C$ are the twice-rotated vector expressed in frames A, B, and C, respectively, and the rotation matrix ${}^A\mathbf{R}^B$ describes the rotation of frame B relative to frame A. From (11), the two frame rotations can be described by two rotation matrices in left-to-right order

$${}^A\mathbf{R}^C = {}^A\mathbf{R}^B\ {}^B\mathbf{R}^C\,. \tag{12}$$

From (5), the two consecutive frame rotations can also be described by two quaternions in the same left-to-right order

$${}^A\mathbf{q}^C = {}^A\mathbf{q}^B\ {}^B\mathbf{q}^C \tag{13}$$

In the "rotation of axes" convention, the two frame rotations are described by two rotation matrices in right-to-left order

$${}^C\mathbf{C}^A = {}^C\mathbf{C}^B\ {}^B\mathbf{C}^A\,. \tag{14}$$

Similarly, from (6), the two frame rotations can also be described by two quaternions in right-to-left order.

$${}^C\mathbf{q}^A = {}^C\mathbf{q}^B\ {}^B\mathbf{q}^A\,. \tag{15}$$

3 Coordinate Frame Transform

So far, we only considered the rotational relation between frames. Figure 1 illustrates the coordinate frame transform from A to B including translational relation.

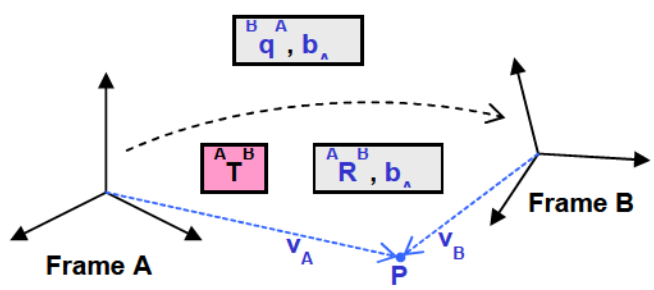

Figure 1. Coordinate transform from frame A to frame B.

In general, the spatial relation between two frames is best described by the "rotation of the object" convention by noting that the frame is attached to an object. For a position P fixed relative to frame B, let $\mathbf{v}_B$ be the position vector representing position P relative to frame B, and $\mathbf{v}_A$ be the position vector representing the same position P relative to the frame A. Then, $\mathbf{v}_B$ can be transformed to $\mathbf{v}_A$ by

$$\mathbf{v}_A = {}^A\mathbf{R}^B\ \mathbf{v}_B + \mathbf{b}_A, \tag{16}$$

where $\mathbf{b}_A$ is the position vector describing the position of frame B origin relative to frame A, and the rotation matrix ${}^A\mathbf{R}^B$ describes the rotation of frame B relative to frame A. In the 4x4 homogeneous transform representation,

$$\begin{bmatrix} \mathbf{v}_A \\ 1 \end{bmatrix} = \begin{bmatrix} {}^A\mathbf{R}^B & \mathbf{b}_A \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_B \\ 1 \end{bmatrix} = {}^A\mathbf{T}^B \begin{bmatrix} \mathbf{v}_B \\ 1 \end{bmatrix}. \tag{17}$$

However, the MSL quaternion library inherited from MER uses the "rotation of axes" convention. For instance, the rover attitude quaternion ${}^B\mathbf{q}^{LL}$ represents the rover (body) attitude relative to the LL (Local Level) frame in the right-to-left order [1]. So there is a need to support both conventions. To reduce confusion, the MSL Frame Manager (FM) supports both conventions in a very specific way. The quaternion is specified only in the right-to-left order, while the rotation matrix and the frame transform are specified only in the left-to-right order. The equivalent rotation matrix for a quaternion in right-to-left order is the transpose of the rotation matrix in left-to-right order.

$${}^B\mathbf{C}^A = ({}^A\mathbf{R}^B)^T. \tag{18}$$

For two consecutive transforms shown in Figure 2,

$$\begin{bmatrix} \mathbf{v}_A \\ 1 \end{bmatrix} = {}^A\mathbf{T}^B \begin{bmatrix} \mathbf{v}_B \\ 1 \end{bmatrix} = {}^A\mathbf{T}^B\ {}^B\mathbf{T}^C \begin{bmatrix} \mathbf{v}_C \\ 1 \end{bmatrix} = {}^A\mathbf{T}^C \begin{bmatrix} \mathbf{v}_C \\ 1 \end{bmatrix}, \tag{19}$$

where

$$\begin{aligned} {}^A\mathbf{T}^C &= \begin{bmatrix} {}^A\mathbf{R}^B & \mathbf{b}_A \\ 0 & 1 \end{bmatrix} \begin{bmatrix} {}^B\mathbf{R}^C & \mathbf{c}_B \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} {}^A\mathbf{R}^B\,{}^B\mathbf{R}^C & {}^A\mathbf{R}^B\mathbf{c}_B + \mathbf{b}_A \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} {}^A\mathbf{R}^C & \mathbf{c}_A \\ 0 & 1 \end{bmatrix}. \end{aligned} \tag{20}$$

The FM internal computation is done using quaternions instead of rotation matrices. The quaternion corresponding to ${}^A\mathbf{R}^C$ is computed by (15). By applying (18), we can show that the translation $\mathbf{c}_A$ is obtained by: 1) conjugate the quaternion ${}^B\mathbf{q}^A$, 2) transform vector $\mathbf{c}_B$ by using the conjugated quaternion rotation, and 3) add the vector $\mathbf{b}_A$.

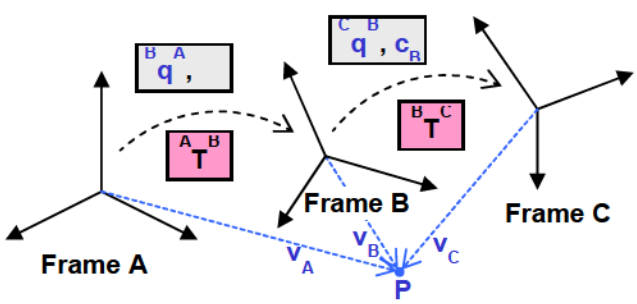

Figure 2. Two consecutive transforms from A to B to C.

4 Frame Tree Database

As mentioned in Introduction, a hierarchical tree data structure is the natural choice for the centralized coordinate frame database. A tree is a collection of nodes connected by directed edges. Each node has exactly one parent except for the root node and has zero or more children.

FM (Frame Manager) maintains the frame tree database with each node containing the coordinate frame transform from the parent to the self as well as the tree node relational data such as the parent node, leftmost child, right sibling, number of children, and depth level. The frame tree consists of the surface and rover frame trees. In Figure 3, all the site frames as well as any associated saved frames belong to the surface frame tree, while all the rover frames from LL (local level) and below as well as celestial body frames belong to the rover frame tree. Site 1 is the root node. When a new site frame is added as the new current site to the surface frame tree, the entire rover frame tree (LL node and below) is moved to the new site node. The frame node is specified by frame_id and frame_index. Site frames are indexed. For instance, Site-N frame is specified by frame_id = FM_SITE and frame_index = N. On the other hand, frames in the rover tree are not indexed, and 0 is normally used for non-indexed frames. Actual coordinate frame transforms are updated by their owner as indicated in Figure 3. For example, FM updates site and saved frames. The ARM module updates the frames associated with the arm operations such as turret-mounted tools/instruments frames. The DRIVE module updates six wheel frames. The RSM (Remote Sensing Mast) module updates various frames associated with RSM and RSM-mounted cameras. The IVP (Inertial Vector Propagation) module updates celestial body frames of sun, Earth, Phobos, and Deimos.

Internally, the FM coordinate frame transform data structure consists of a quaternion, a position vector, and a valid flag. The valid flag indicates whether the current transform values are valid. Externally, however, FM provides public interface functions so that clients can assign the frame transform ${}^A\mathbf{T}^B$ (from frame A to frame B; left-to-right order) data structure either by a quaternion ${}^B\mathbf{q}^A$ (right-to-left order) and a position $\mathbf{b}_A$ (frame B origin relative to frame A) or by a rotation matrix ${}^A\mathbf{R}^B$ (left-to-right order) and a position $\mathbf{b}_A$.

If a client module is the owner of the frame tree nodes, it can update the frame transforms of the nodes by using FM asynchronous function calls that can update either a single node or up to 20 nodes at a time. Upon completion of the frame tree update request, FM returns the reply with the status. Asynchronous function calls require the client to wait until the completion reply is received. Client modules can also get a single frame transform between any two nodes or multiple sets of frame transforms by using FM asynchronous function calls. FM returns the resulting frame transform(s). Specific to the FM nomenclature is the definition of input and output frames so that the resulting frame transform is from the output frame to the input frame based on the following relation.

$$\begin{bmatrix} \mathbf{v}_{out} \\ 1 \end{bmatrix} = {}^{out}\mathbf{T}^{in} \begin{bmatrix} \mathbf{v}_{in} \\ 1 \end{bmatrix} \tag{21}$$

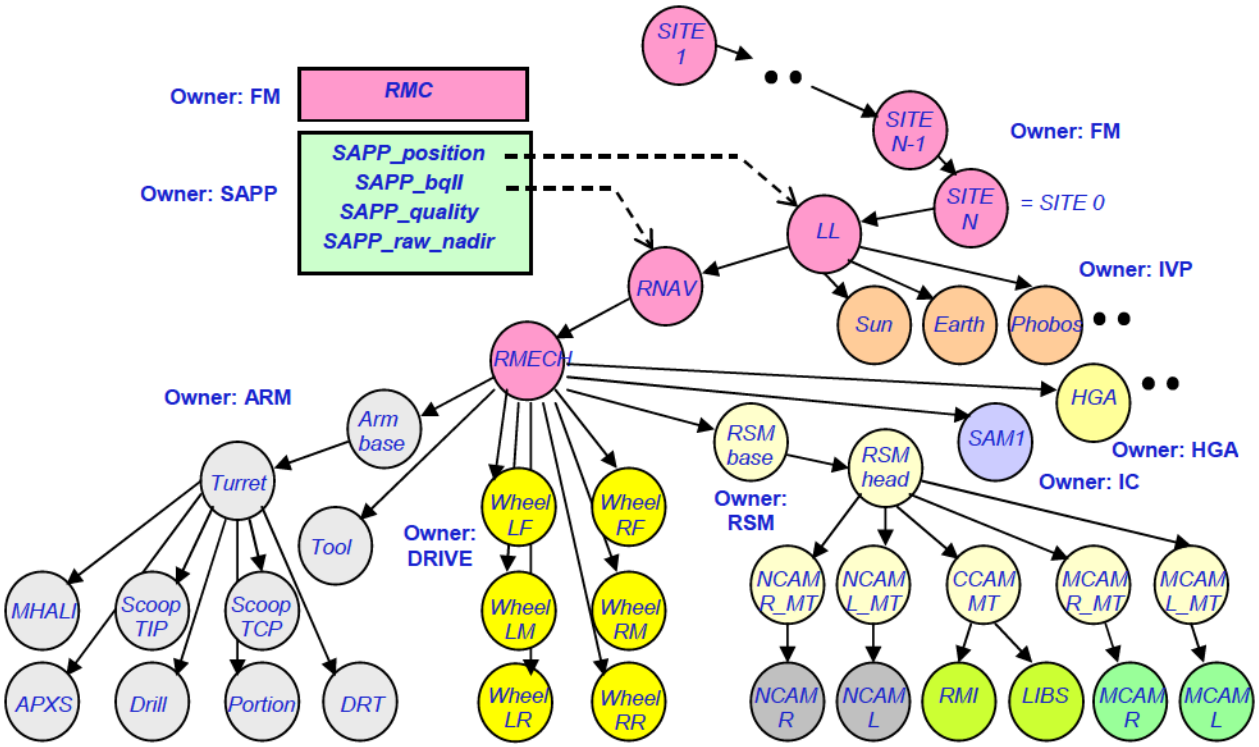

Figure 3. A partial display of the frame tree database containing surface and rover frames trees.

Figure 4 illustrates an example of how the coordinate transform from frame B to frame E is computed. First, find the common ancestor and then use the following relation.

$$
\begin{aligned}
{}^{B}\mathbf{T}^{E} &= ({}^{P}\mathbf{T}^{B})^{-1}\ {}^{P}\mathbf{T}^{E} \\
&= ({}^{P}\mathbf{T}^{A}\ {}^{A}\mathbf{T}^{B})^{-1}\ ({}^{P}\mathbf{T}^{C}\ {}^{C}\mathbf{T}^{D}\ {}^{D}\mathbf{T}^{E})
\end{aligned}
\tag{22}
$$

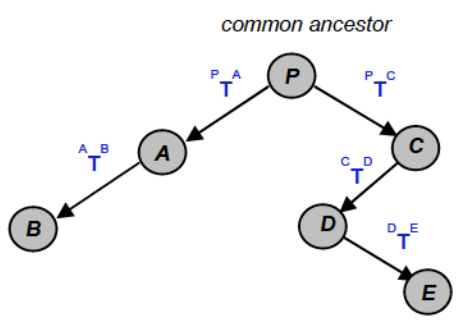

Figure 4. Computing frame transform from B to E

When the rover is stationary over multiple sols to perform science involving robotic arm operations, it is desirable to eliminate small variations occurring in rover attitude updates caused by IMU drift and sun identification error. If the rover did not move, the science target defined relative to the site frame in the previous sol should not change in the next sol. Figure 5 illustrates how FM achieves this goal by saving frames in a specific way. A new saved frame branch consisting of RNAV_SAVED, LL_SAVED, and SITE_SAVED is created off the current site frame. The transforms for the saved frames are assigned such that SITE_SAVED and SITE frames are initially aligned. As long as RNAV_linked flag remains TRUE, RNAV_SAVED frame is updated synchronously with the LL and RNAV update, keeping RNAV_SAVED and RNAV aligned. In this case, ${}^{SITE_SAVED}\mathbf{T}^{RNAV}$ does not change even though ${}^{SITE}\mathbf{T}^{RNAV}$ changes upon rover attitude updates. Thus, this method compensates for the rover attitude estimate noise. For this reason, robotic arm operations nominally use the SITE_SAVED frame instead of SITE frame. As soon as the rover drives, RNAV_linked is set to FALSE.

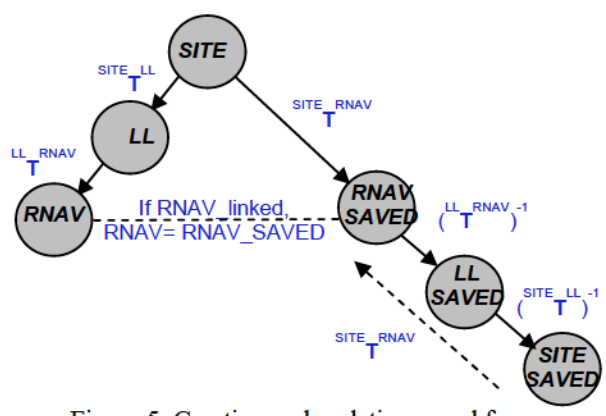

Figure 5. Creating and updating saved frames

5 SAPP Rover Position/Attitude Data

Synchronous calls are far more convenient to use than asynchronous calls, since clients do not need to wait for FM replies. To provide synchronous functions for shared

access of the SAPP (Surface Attitude, Positioning, and Pointing) data, FM holds two sets of ping-pong double buffers with a counter. SAPP rover position, attitude, and attitude quality use one set of double buffers (Figure 6), while SAPP raw nadir vector uses another set. SAPP writes the SAPP data to the FM ping/pong buffers up to 8 Hz, and DRIVE and other client modules read the data by synchronous calls. Since SAPP is the only writer, it is a single-writer/multiple-readers case [4]. In this case, it is possible to support synchronous access by using double buffers without using more system-intrusive disabling interrupts or semaphores.

The single writer does the following.

1. Read the counter.
2. Increment the counter.
3. Get the write data buffer location based on the LSB of the counter.
4. Write the data into the data buffer.
5. Write the counter back.

Multiple readers do the following.

1. Read the counter.
2. Get the read data buffer location based on the LSB of the counter.
3. Read the data from the data buffer.
4. Read the counter again.
5. If old_count ≠ new_count, repeat.
6. Else done.

Note that readers never write anything, not requiring locks. The counter read and write are atomic. The counter's LSB indicates which buffer is for READ and which buffer is for WRITE. The counter is used for data integrity. The data read is valid if the count did not change after the READ.

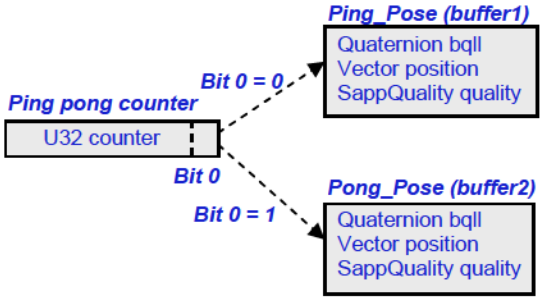

Figure 6. Synchronous access of shared SAPP data using ping-pong double buffers

FM is also responsible for updating LL and RNAV nodes of the frame tree with the SAPP data (Figure 3). Since updating the LL and RNAV nodes does not need to be as often as the SAPP data, they are updated only when FM receives an access request from clients by checking the dirty bit that indicates whether the update is needed.

6 Rover Motion Counter

FM also provides synchronous functions for the shared access of the RMC (rover motion counter) data. RMC is used to tag data products. It is a set of ten indexes to uniquely identify the rover operational state in terms of the site, drive, pose, and various mechanism states (Figure 7). Unlike the SAPP data access, the RMC data access is a multiple-writers/multiple-readers case. In this case, the double-buffer technique does not work, and thus the int_lock (disable interrupt) mechanism is used to provide synchronous functions for RMC data access. In order not to worry about undesirable side effects, no function calls are used within the interrupt locked section. Reading the 20-byte RMC data is simple: just read them within the interrupt locked section. Writing RMC data is, however, more involved, since every RMC update requires NPM (Non-volatile Parameter Memory) save. The NPM save rate is limited up to 8 RMC records per second. Overflow records are dropped silently.

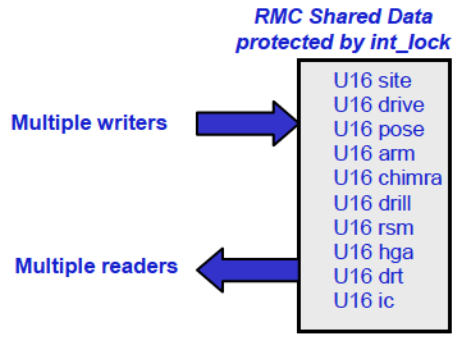

Figure 7. Synchronous access of shared RMC data protected by int_lock. U16 is unsigned short integer.

7 Hierarchical State Machine

Figure 8 shows the HSM (hierarchical state machine) of the FM module. During the FM module initialization, FM first resets the FM state variables and shared data to default values and then load the frame tree database, FM states, RMC indexes, and parameters from the NPM. Thereafter, FM HSM is started. As the FM state machine enters the INACTIVE state as the initial state, FM checks the integrity of the frame tree such as acyclicity and connectivity. If it fails, an FM_FAULT_INTEGRITY fault occurs. Until the fault is cleared, FM simply returns a FALURE reply to any incoming FM commands or requests.

Upon receiving the ACTIVATE request in the INACTIVE state, FM state machine transitions to the READY state. When commands or requests are received in the READY state, FM state machine disables the command and request message queues and then transitions to the BUSY state. After completing the command or request, FM state machine re-enables the message queues and transitions back to the READY state if no tree topology change is involved. If the command or request results in tree topology change, however, FM state machine transitions to the AWAITING_NPM state to save the frame tree database to NPM. After saving the frame tree to NPM, the FM state machine transitions back to the READY state.

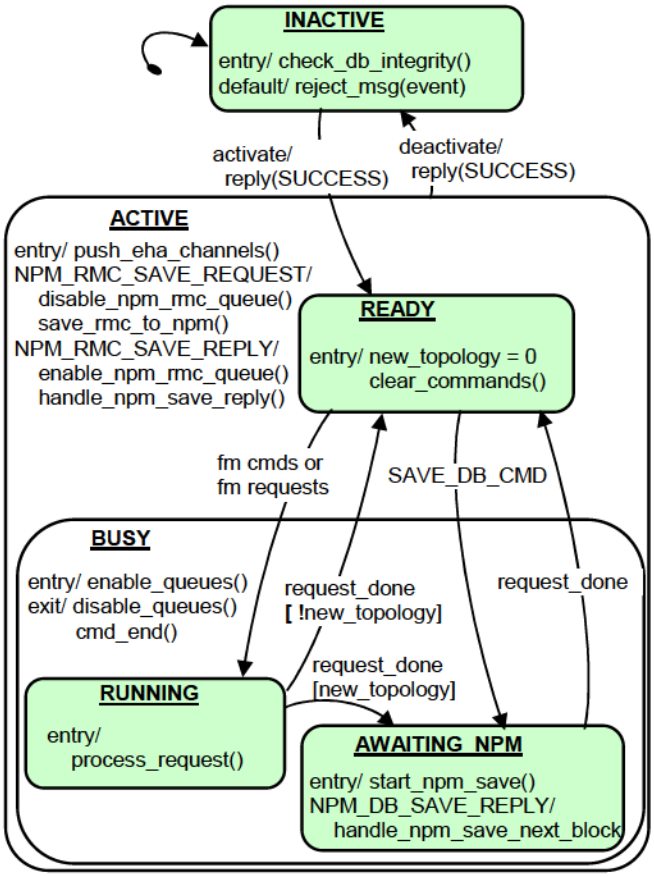

Figure 8. FM hierarchical state machine

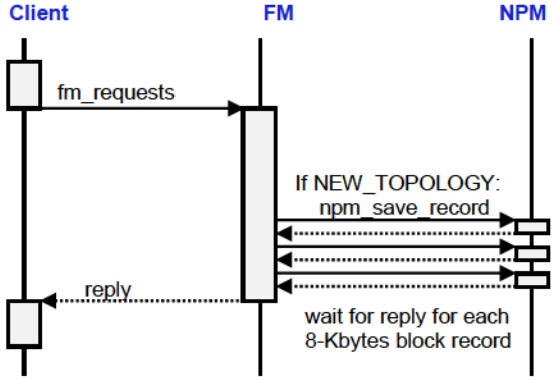

Figure 9. A sequence diagram for requests that require NPM save after the frame tree update

An example of a sequence diagram is shown in Figure 9 for a command execution that actually changes the topology of the frame tree. This requires saving the updated frame tree in NPM. Since the frame tree database size is about 24 Kbytes, three 8-Kbytes NPM record blocks are needed to store the entire database. The frame tree is recorded sequentially with one block at a time, by waiting for the NPM reply before requesting for the next NPM save. The command and request message queues are disabled until the entire database is saved to NPM.

8 Flight Operation Examples

Several surface operations greatly benefit from the onboard centralized frame tree database by eliminating complex, error-prone manual calculations on the ground. A few operational examples are listed here.

- Point RSM-mounted cameras to a target specified relative to any frame of the frame tree for imaging
  - Point MCAM to a robotic arm tool
  - Point NAVCAM to a saved frame
  - Acquire images of the sun
- Move or point the arm tool to a target specified relative to any frame of the frame tree
  - Move PORTIONER (turret mounted tool) above the SAM-1 inlet cover for sample drop-off
  - Move MAHLI (turret-mounted camera) relative to a SITE_SAVED frame
- Drive the rover to a goal specified relative to any site or saved frame

9 Conclusions

The MSL Frame Manager successfully implemented the centralized frame tree database onboard for the first time for a Mars surface mission. The frame tree enables the use of frames for target pointing of mast-mounted cameras and frame-reference arm moves, enhancing surface operations significantly.

Acknowledgements

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] K. Ali, C. Vanelli, J. Biesiadecki, M. Maimone, Y. Cheng, M. San Martin, J. Alexander, "Attitude and Position Estimation on the Mars Exploration Rovers," IEEE Conf. on Systems, Man and Cybernetics, Oct. 2005.

[2] R. Bonitz, L. Shiraishi, M. Robinson, J. Carsten, R. Volpe, A. Trebi-Ollennu, R. E. Arvidson, P. C. Chu, J. J. Wilson, K. R. Davis, "The Phoenix Mars Lander Robotic Arm," Proc. of 2009 IEEE Aerospace Conf., Mar. 2009.

[3] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1997.

[4] H. Huang, P. Pillai, K. G. Shin, "Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems", USENIX Annual Technical Conference, Jun. 2002.