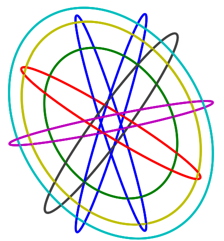
A Reconfigurable, Decentralized Framework for Formation Flying Control

Prototype Design Document

Joseph B. Mueller
November 30, 2005



Princeton Satellite Systems, Inc.
33 Witherspoon Street
Princeton, New Jersey 08542

Contents

1 Introduction **1**

2 Requirements **3**

3 System Overview **5**

3.1 Interfaces 5

3.2 Software Architecture 6

3.3 Software Module Design 8

3.4 Running a Simulation 11

3.5 Initializating the DFF Software 14

3.5.1 Command Lists 17

4 Software Modules **19**

4.1 Command Processing 21

4.2 Parameter Database 24

4.3 Relative Navigation 26

4.4 Coordinate Transformation 29

4.5 Team Management 31

4.6 Guidance Law 38

4.7 Control Law 49

4.8 Collision Monitoring 54

4.9 ISL Management 58

4.10 Delta-V Management 62

4.11 Attitude Management 65

A Data Structures **67**

A.1 Command Data 67

A.2 Team Data 67

A.3 State Data 68

A.4 Geometry Data 68

A.5 Eccentric Geometry Data 68

A.6 Window Data 69

A.7 Planning Parameters Data 69

A.8 Team Goals Data 69

A.9 Eccentric Team Goals Data 70

A.10 Cost Estimate Data 70

A.11 Constraints Data 70

A.12 Burn Data 71

A.13 Maneuver Data 71

A.14 Delta-V Command Data 71

A.15 Orientation Data 71

A.16 ISL Message Data 72

B Flowcharts **73**

CHAPTER 1. Introduction

The software described in this document is being developed in support of a Phase II SBIR contract with NASA Goddard Space Flight Center. The title of the project is "A Reconfigurable, Decentralized Framework for Formation Flying Control". The objective of this project is to develop software that will support autonomous guidance and control operations for a fleet of close-orbiting spacecraft in a wide range of orbits.

As with all SBIR projects, the work must possess a significant innovation. The innovative nature of this design is that it provides decentralized guidance and control for a spacecraft cluster that can be dynamically organized as a multiple-team hierarchy. Spacecraft can be grouped into teams of manageable size, and teams are connected in a hierarchical fashion so that control of the entire formation is maintained. This provides a flexible approach to implementing decentralized control. There are two main advantages to this approach. First, it is capable of supporting large clusters because the satellites can be divided into appropriately sized teams. Second, the team organization, along with the number of spacecraft in each team, can be changed over time. This will allow spacecraft to be added to or removed from the cluster, increasing the mission flexibility.

The software package is referred to as as the "Decentralized Formation Flying" system, or the DFF system for short. The complete DFF system is distributed uniformly across the fleet, such that each spacecraft is running the same software. To function properly, each spacecraft must have the ability to communicate with all other spacecraft by sending and receiving data over an inter-satellite link (ISL). In addition, the DFF system requires several interfaces with spacecraft subsystems, including other software modules and hardware components. Generic interfaces will be included with the DFF system; they will be implemented as interface plugins, isolated from the core DFF software. Each interface plugin will require modification to provide an actual interface with specific subsystems.

This document describes the prototype DFF system that has been developed in MATLAB. The development of the prototype design carries with it two objectives. First, it aims to establish feasibility. This has been accomplished in Phase I. Second, the prototype is meant to serve as a blueprint for subsequent development of the real-time software. In order to accomplish this objective, it is necessary to provide sufficient detail about how the prototype system works. However, in the course of this description, it is important to avoid the imposition of unnecessary or unrealistic constraints on the real-time software. Therefore, this document includes a detailed description of the DFF functionality that is platform-independent.

The MATLAB-based prototype is implemented in a manner that emulates the MANTA environment to the greatest extent possible. However, because MATLAB is single-threaded, it is inherently limited in its ability to emulate a multi-threaded system. The system is divided into discrete modules, with each module having its own block of persistent memory, and message-passing is imitated with a function call to the destination or source module. The prototype was designed in this fashion to aid in the process of transitioning the design from MATLAB to C++/MANTA. However,

it is not assumed that the MANTA-based design will necessarily match the MATLABbased design. Development in MANTA should be carried out with the objectives of producing efficient, fault-tolerant software that captures all of the functionality of the prototype.

Chapter 2 outlines the requirements for the DFF system. The requirements serve as the motivation for the prototype design. Chapter 3 provides an overview of the system, and how it has been implemented in MATLAB. The details of each software module are then provided in Chapter 4.

CHAPTER 2. Requirements

This section outlines the requirements of the DFF system.

The following quote is taken from the original 2002 NASA SBIR solicitation:

> Novel approaches to autonomous control of distributed spacecraft and the man-agement of large fleets of heterogeneous and/or homogeneous assets. Submissions should focus on one or several of the following technologies and system-level concepts:
>
> - **Formation self-organization**
> - **Reconfigurable control laws**
> - Robust and fault-tolerant control laws
> - **Algorithms for autonomous formation reconfiguration**
> - Nonlinear, robust estimation algorithms for relative navigation
> - **Integrated, multi-spacecraft formation guidance and control**
> - On-board, multi-spacecraft, closed-loop responsiveness to sensed events
> - **Low-cost approaches for formation navigation and control exploiting low-cost and existing technologies such as GPS Optimal (e.g., minimum fuel, minimum time) approaches for formation maintenance and maneuvering**
> - Unique concepts for dealing with relevant perturbations and disturbances such as J2, solar radiation pressure, etc.
> - New modeling techniques to support the technologies and concepts listed above

The items shown in bold are those specifically addressed by the DFF system. These represent the high-level requirements.

The high-level requirements for the DFF system may be summarized as follows:

Req. #1 Provide formation guidance and control capabilities to a multi-spacecraft cluster

Req. #2 Enable autonomous reconfiguration of the formation geometry

Req. #3 Enable autonomous initialization and organization of formations

Req. #4 Enable run-time modifications to be made to the software

Req. #5 Provide a dynamic and flexible command and telemetry interface

Req. #6 Be applicable to a wide range of formation flying missions

Requirement #6 is desirable from a commercialization standpoint, as it increases the potential market size for the software. In particular, the software should support both circular and eccentric orbits, and be capable of handling a variable number of spacecraft in the cluster. In order to provide guidance and control for both small and large clusters, a decentralized approach is applied within a multiple-team framework. This represents one of the key innovations in the project.

The combined purpose of Requirements #4 and #5 is to ensure that the system is flexible so that it can be adapted over time. The run-time modifications include the ability to upgrade and add to the functionality of the software. This requires a flexible command and telemetry interface, since the set of commands and telemetry will change as the software is modified.

The ability to perform run-time modifications will be supported by the MANTA software architecture. The DFF system will be implemented in MANTA as a set of independent, interactive tasks. Additional functionality may be added to the system at run-time by uploading new tasks.

Requirement #3 involves the ability to autonomously initialize and organize the cluster. This functionality enables a cluster of close-orbiting spacecraft to cooperatively form a multiple-team framework and collectively determine a safe set of relative trajectories.

Requirement #2 states that the software must be capable of autonomously reconfiguring the formation geometry. This refers to cases where the desired geometry of the cluster is specified in terms of high-level objectives, and the DFF system is responsible for defining and achieving a specific set of relative trajectories to meet those objectives.

Requirement #1 involves the core functionality of the DFF system. The software shall be capable of defining the desired relative trajectory of all spacecraft in the cluster (guidance), and be capable of planning and implementing maneuvers to achieve those trajectories (control).

The system has been designed to meet all of the above requirements.

CHAPTER 3.

System Overview

This section provides an overview of the DFF prototype design. The required interfaces are summarized first. The basic architecture of the DFF software is then presented, followed by a description of the general software module design. Next, a discussion of the simulation procedure is provided, and finally the details of the initialization method are described.

3.1 Interfaces

The DFF software is designed to provide three basic functions:

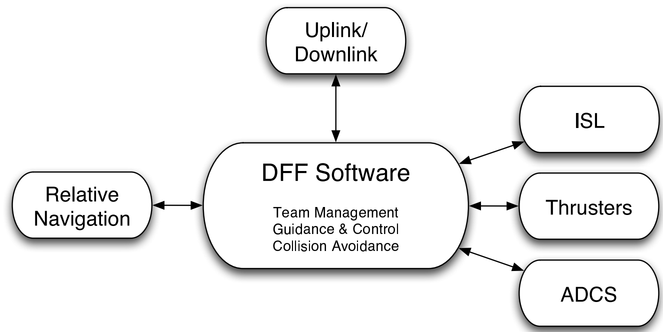
- Team Management
- Formation Guidance and Control
- Collision Avoidance

In order to provide these functions, the software must interface with with other subsystems and software packages on the spacecraft. The list includes:

- **Ground Communication System** – Enables commands to be received from the ground station, and telemetry to be sent to the ground station.
- **Relative Navigation System** – Provides an estimate of the absolute and relative state (position and velocity) of the local spacecraft in the cluster.
- **Inter-Spacecraft Link (ISL)** – Enables commands and data to be sent between spacecraft.
- **Thruster Subsystem** – One or more thrusters to enable orbital maneuvering.
- **Attitude Control System (ADCS)** – Controls the orientation of the spacecraft. Enables a target attitude to be reached, so that the thruster(s) may be oriented in the proper direction.

These interfaces are illustrated in Figure 3-1 on the following page.

Figure 3-1. DFF Interfaces



In the real DFF system, the interface to each subsystem will be implemented with a corresponding Interface Plugin module. These modules will be designed to handle the specific interface requirements associated with the specific hardware and software components. In order to support testing of the DFF system, a complete set of interface modules will be developed. The details are summarized below:

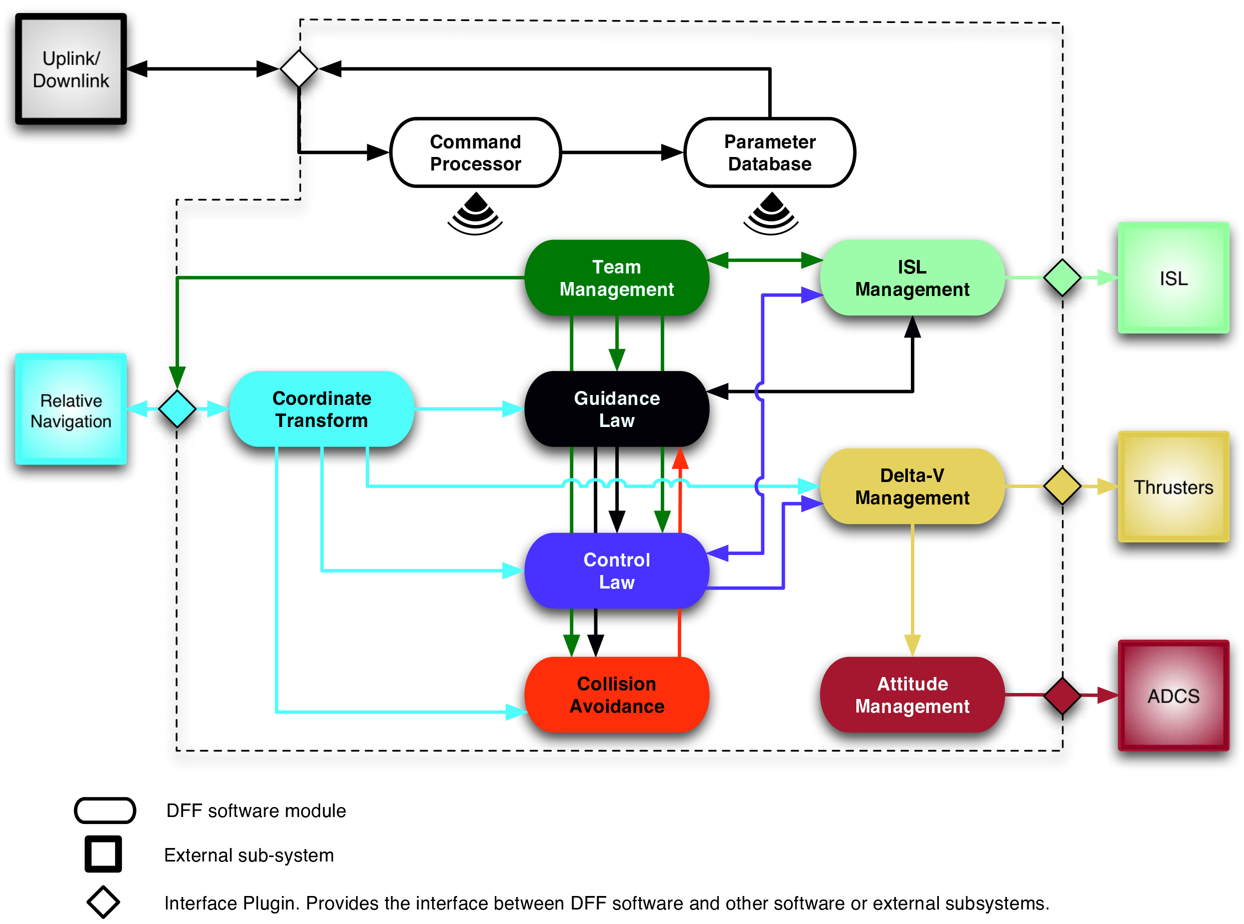
- **Ground Comm Interface Plugin** – At a minimum, this will read a pre-defined set of commands from a file, and supply them to the Command Processing module at specified times during the simulation. If time permits, an interface to the TelemetryCommander application will be developed. This will facilitate direct user-interaction with the DFF system as it runs.
- **Rel Nav Interface Plugin** – A specific interface will be developed for the GEONS navigation software.
- **ISL Interface Plugin** – This will interface directly with the ISL model in DSim. Binary data will be supplied to the ISL model on the local spacecraft. The model will limit the rate at which data is transmitted to respect the expected bandwidth of the ISL hardware. It will "transmit" the data by making it available to the DSim ISL model associated with the destination spacecraft(s). Transmission will only take place if the sending and receiving spacecraft are within range and the antennaes are within the field of view.
- **Thruster Interface Plugin** – This will interface directly with the thruster model in DSim. The spacecraft may have one or more thrusters, and one or more fuel tanks. Each thruster will have a nominal thrust capability, so that the desired delta-v is achieved through pulse-width modulation. The interface plugin will simply command the thruster to turn on or off at the specified times.
- **ADCS Interface Plugin** – This will interface directly with separate ADCS software modules that also run within MANTA. These modules are considered support software, and are not included in the DFF design. The ADCS modules were designed for previous contracts that involved attitude determination and control for a 3-axis stabilized spacecraft. The ADCS modules will interface directly with DSim models of attitude sensors and actuators.

In the prototype design, no interface plugins exist. The interface between the DFF modules and the external subsystems is handled directly by each module. For example, the Delta-V Management module interfaces directly with the thruster model. This was done for the sake of speed and simplicity in the prototype development phase. However, when the inputs and outputs of the modules are described in Chapter 4, the interface plugins are included in the descriptions. This is done so that the module descriptions provide a more accurate blueprint for how to develop the MANTA-based system.

3.2 Software Architecture

The DFF prototype consists of 10 software modules. A diagram of the architecture is shown in Figure 3-2. The dashed line surrounds the core DFF software. Each of the external software and hardware components require a separate interface. These interfaces are specific to the spacecraft design and are therefore implemented as interface plugins, separate from the DFF system. The arrows connecting the DFF modules indicate the flow of messages within the system. The Command Processing and Parameter Database modules send messages to all other modules.

Figure 3-2. Architecture of DFF Prototype



The primary functions of each module are summarized below.

- **Command Processing** – Receives commands from the ground station and forwards them to the appropriate module(s). Any commands that update the value of internal parameters are also sent to the Parameter Database.
- **Parameter Database** – Receives parameter updates from the Command Processing module. Serves as a central repository for parameters that may be requested at any time by other modules. Is used to initialize all other modules at startup.
- **Coordinate Transformation** – Transforms the state estimate from the Relative Navigation module into appropriate coordinate frames as required by the DFF algorithms.
- **Team Management** – Maintains the hierarchical team organization of the cluster. Provides autonomous team formation and autonomous reference rollover capabilities.
- **Guidance Law** – Determines the desired relative trajectory of all spacecraft based upon the desired geometry of the team or cluster.

- **Control Law** – Plans impulsive maneuvers to achieve the desired relative trajectory.
- **Collision Monitoring** – Monitors the probability of a collision (over a given time window) with other spacecraft in the cluster, and provides a preemptive collision avoidance capability.
- **ISL Management** – Interfaces with the ISL subsystem on the spacecraft. Enables internal DFF messages to be sent to and received from other spacecraft.
- **Delta-V Management** – Interfaces with the thruster(s) subsystem on the spacecraft. Receives delta-v commands from the Control Law and Collision Avoidance modules. Sends commands to fire thruster(s) at the appropriate times to achieve the desired delta-v. Computes the required attitude that the spacecraft must have for each thruster firing, if necessary.
- **Attitude Management** – Interfaces with the ADCS. Receives attitude commands from the Delta-V Management module. Commands the ADCS to slew to the desired quaternion prior to the thruster firing.

The Relative Navigation block provides the navigation solution to the rest of the DFF software. In the prototype design, this module simply obtains the state data from the simulation and provides it to the Coordinate Transformation module. Random noise will be added according to the standard deviations specified in this module's initialization function. In the MANTA design, a specific interface to the GEONS software will be developed.

The ADCS software is implemented as two additional software modules:

- `AttitudeManeuver.m`
- `AttitudeController.m`

These modules are considered supporting software, and are not part of the DFF system.

The details of each module are provided in Chapter 4.

3.3 Software Module Design

The software modules are implemented in MATLAB in an object-oriented fashion. The intent is to emulate the MANTA environment to the greatest extent possible. Each module is implemented as a single m-file. It has its own block of persistent memory, so that member variables may be stored and modified as the system runs. In addition, each module has member functions, that are visible only within the scope of the module function. Message-passing is imitated with a function call to the destination or source module.

The structure of each module is based upon the `DFFModuleTemplate.m` file. The complete contents of this file are shown in the code listing below. The function has three inputs – `action`, `d`, and `k`. The `action` is a string argument that tells the module what to do. In general, the action either corresponds to a member function (such as `Update` or `Initialize`) or the name of a message.

The second input, `d`, is used to pass data into the module. The type of data depends entirely on the action; some actions do not have corresponding data, in which case `d` is empty. For example, when the module is called with the `'update'` action, the Julian date is passed in as the input `d`. When the action corresponds to an incoming message, such as `'set teamData'`, the input `d` would be a team data structure.

The third argument, `k`, identifies the spacecraft. In general, the spacecraft ID may be any positive integer. However, in order to speed up and simplify the MATLAB implementation, the IDs for a simulation of N spacecraft are restricted to range from 1 to N. The reason for this restriction is based upon the manner in which the persistent memory is stored.

Listing 3.1. DFFModuleTemplate.m *DFFModuleTemplate*

```
1 function t = DFFModuleTemplate( action, d, k )
2
3 %-------------------------------------------------------------------------
4 %   Template for DFF software module. Describe the basic functionality here.
5 %-------------------------------------------------------------------------
6 %   Form:
7 %   t = DFFModuleTemplate( action, d, k )
8 %-------------------------------------------------------------------------
9 %
10 %   ------
11 %   Inputs
12 %   ------
13 %   action        (1,:)     Action to be performed
14 %   d             (1,1)     Data associated with incoming messages
15 %   k             (1,1)     Spacecraft ID
16 %
17 %   -------
18 %   Outputs
19 %   -------
20 %   t             (1,1)     Output for any accessors that you may add
21 %
22 %-------------------------------------------------------------------------
23
24 %-------------------------------------------------------------------------
25 %   Copyright (c) 2005 Princeton Satellite Systems, Inc.
26 %   All rights reserved.
27 %-------------------------------------------------------------------------
28
29 persistent s % This contains the memory for this function
30
31 if( nargin < 3 )
32    k = 1;
33    if( nargin < 2 )
34       d = [];
35    end
36 end
37
38 switch action
39
40    case 'initialize'
41       s{k} = Initialize(k);
42
43       % Commands  (some simple examples provided below)
44       %---------
45    case 'reset'
46       s{k} = Initialize(k);
47
48    case 'set_updatePeriodMT'
49       s{k}.updatePeriod = d;
50
51    case 'command_on'
52       s{k}.on = 1
53
54    case 'command_off'
55       s{k}.on = 0;
56
57    case 'set_data'
58       s{k}.data = d;
59
60       % Accessors for providing this module's data to other modules
61       %-------------------------------------------------------------
62    case 'get_updatePeriodMT'
63       t = s{k}.updatePeriod;
64
65    case 'get_data'
66       t = s{k}.data;
67
68       % Processing
69       %-----------
70    case 'update'
71       if( d - s{k}.timeLastUpdate >= s{k}.updatePeriod/86400.0 )
72          s{k} = Update( s{k}, d );
73          s{k}.timeLastUpdate = d;
74       end
75
76       % Test Cases
77       %-----------
78    case 'test'
79       DFFModuleTemplate('initialize',[],k);
```

```
80        s{k} = Update( s{k}, JD2000 );
81        t = DFFModuleTemplate('get_data',[],k);
82        disp(t);
83
84        % This will only be called if you send a command or try to access
85        % an accessor that doesn't exist
86        %------------------------------------------------------------
87     otherwise
88        MessageQueue('add',[mfilename,':SC',num2str(k)],...
89           sprintf('%s_is_not_a_valid_action',action)','error');
90        t = s;
91  end
92
93  %------------------------------------------------------------------------
94  %  FSW Processing. All your algorithms go here. Add additional
95  %  subfunctions or calls to other MATLAB functions as needed.
96  %------------------------------------------------------------------------
97  function d = Update( d, jD )
98
99  if( d.on )
100    d.data = foo( d.data );
101    DFFOtherModule( 'set_data', d.data, d.iD );
102 end
103
104 %------------------------------------------------------------------------
105 %  Member functions
106 %------------------------------------------------------------------------
107
108 % foo
109 function a = foo( b )
110
111 %------------------------------------------------------------------------
112 %  Initialize the member variables.
113 %------------------------------------------------------------------------
114 function d = Initialize( iD )
115
116 d.iD             = iD;
117 d.updatePeriod   = 10;
118 d.timeLastUpdate = -d.updatePeriod;
119
120 % flags
121 d.on   = 0;
122
123 % adjustable parameters
124 d.data = DFFParameterDatabase('get_data',[],iD);
```

DFFModuleTemplate

The persistent memory for the module is stored in the variable `s` (see line 29). This variable is initialized to be a cell array of similar data structures, where each cell corresponds to a different spacecraft. For example, if a cluster of 10 spacecraft are being controlled, `s` would be a cell of length 10. The fourth cell would correspond to the spacecraft whose ID is 4.

Every module in the DFF system has the following actions in common:

- `initialize`
- `update`
- `reset`
- `set updatePeriod##`
- `get updatePeriod##`

The first two actions are initiated by the executive function, `DFFControl`. The latter three actions are messages that may be sent from outside modules.

The "##" suffix in the last two messages is used to distinguish between different modules. In this case, the suffix is "MT" for "ModuleTemplate". A unique suffix is used for every module in the system.

The 'initialize' action is shown on line 40. The result is to call the Initialize member function, which begins on line 114. The purpose of this function is to initialize all member variables. Every module has at least the first three member variables shown: iD, updatePeriod, and timeLastUpdate. As an example, two additional member variables are initialized here: a flag (or boolean) called on and an adjustable parameter called data. Notice that the value for data is obtained from the Parameter Database module. The Parameter Database is initialized before any other module. It contains a database of parameters that are required by the other modules. This process is explained further in Section 3.5 on page 14.

The 'update' action appears on line 70. If the elapsed time since the last update surpasses the update period, then the Update member function is called. The Update function begins on line 97. This is where the main functionality of the module occurs. Note that The Julian date is passed in directly to this function. This is done in every module to increase the speed of the simulation. The alternative is for each module to obtain the Julian date from the software clock, via the function FSWClock.m, which is more akin to how the current time will be obtained in MANTA.

The 'reset' action appears on line 45. This causes the module to re-initialize. Clearly, it has the same effect as the 'initialize' action. The name "reset" is chosen simply for clarity, to distinguish this as a command that may be issued at run-time.

In general, a module can do two things: 1) modify member variables, and 2) send messages to other modules. These things may be done either during an update, or upon the receipt of a message. The resulting data structure is returned at the end of the update and is then stored in the cell array s. It is important to note that changes to the data structure are not stored until *after* the Update function returns – a consequence of the single-threaded nature of MATLAB.

Five of the actions shown in this template correspond to commands that may be sent from the ground or perhaps another module (lines 43-58). Two actions are accessors (lines 60-66). These are messages sent from some other module that is requesting data. In addition, a 'test' action is included on line 78. This is meant to be used for debugging and verification purposes.

The 'command on' action appears on line 51. It has the effect of setting the member variable on to 1. Alternatively, the variable is set to 0 with a 'command off' message. These messages represent commands that may be sent from the ground (via the Command Processing module), or from another module. A different type of command message 'set data', which appears on line 57. As opposed the first two command messages, this one has corresponding data. The messages called 'get updatePeriodMT' and 'get data' are both accessors. For example, another module may call the DFFTemplateModule function with the action 'get data' to obtain the value of the variable data.

3.4 Running a Simulation

The DFF software may be run in a multi-spacecraft simulation by calling the DFFSimulation function. The help header for this function is shown below.

Listing 3.2. DFFSimulation.m *DFFSimulation*

```
%-------------------------------------------------------------------------------
%   Initialize and run a multiple spacecraft simulation with the DFF software.
%-------------------------------------------------------------------------------
%   Form:
%   d = DFFSimulation( sim )
%-------------------------------------------------------------------------------
%
%   ------
%   Inputs
%   ------
%   sim                  (.)    Simulation data structure, containing simulation
%                                 parameters and spacecraft data
%   recordTargetState    (1)    Record the target state or not?    (optional)
%   showTeamOrg          (1)    Show the team organization or not? (optional)
%   showMsgQ             (1)    Show the Message Queue or not?     (optional)
%
```

```
%   -------
%   Outputs
%   -------
%   d            (.)    Output data structure, containing time histories of
%                          all states and controls
%
%--------------------------------------------------------------------------
```

DFFSimulation

The details of the simulation are governed by the contents of the simulation data structure. The elements of this data structure are described in Table 3-1.

Table 3-1. Simulation Data Structure

Field	Data	Description
`nSC`	`int`	Number of spacecraft
`nPMax`	`int`	Defines how frequently to store data for plotting. The state data is stored every `nPMax` time-steps during the simulation.
`ideal`	`int`	Indicates whether the simulation is ideal (1) or not (0). If ideal, the measurements and actuation are perfect.
`jDEpoch`	`double`	Julian date of the simulation start time.
`duration`	`double`	Duration in seconds.
`dT`	`double`	Integration time-step in seconds.
`prop`	`struct`	Contains the propagation settings. Described in Table 3-2.
`commandList`	char[]	Name of the command list file. This is an m-file that contains the complete set of time-tagged commands that are to be sent to the DFF system. Command lists are described in Section 3.5.1 on page 17.
`controlDataFile`	char[]	Name of mat-file that contains initialization data for the supporting ADCS software modules. The DFF modules do not use data from this file to initialize.
`spacecraftDataFile`	char[]	Name of the m-file that contains spacecraft-related data. This is described further in Section 3.5 on page 14.
`scNames`	char[][]	Names of the individual spacecraft. Used when the simulation results are loaded into the PlottingTool.
`state`	`matrix`[14,`nSC`]	State data. One column per spacecraft. Each column has 14 rows. States include: ECI position (3) and velocity (3), ECI-to-body quaternion (4), angular velocity (3), and fuel mass (1).
`stateName`	char[][]	Name of each state. Used when the simulation results are loaded into the PlottingTool.
`stateUnit`	char[][]	Units for each state. Used when the simulation results are loaded into the PlottingTool.

Table 3-2. Propagation Data Structure

Field	Data	Description
`hiFidelity`	`int`	Indicates whether to use a high-fidelity gravity model, or to simply compute the acceleration according to the inverse square law. The remaining fields of this data structure are required only if this field is true.
`gravityModel`	`struct`	Contains the gravity model data.
`numberOfTesseralHarmonics`	`int`	The number of zonal harmonics to include.
`numberOfZonalHarmonics`	`int`	The number of zonal harmonics to include.
`planetaryDisturbancesOn`	`int`	Indicates whether to model planetary disturbances.

The simulation data structure may be generated manually at the command line, or by means of a simulation setup file.

Several setup files have already been written in support of various demos. Any one of these may be used as a template. In order to run the autonomous team formation demo, you would type:

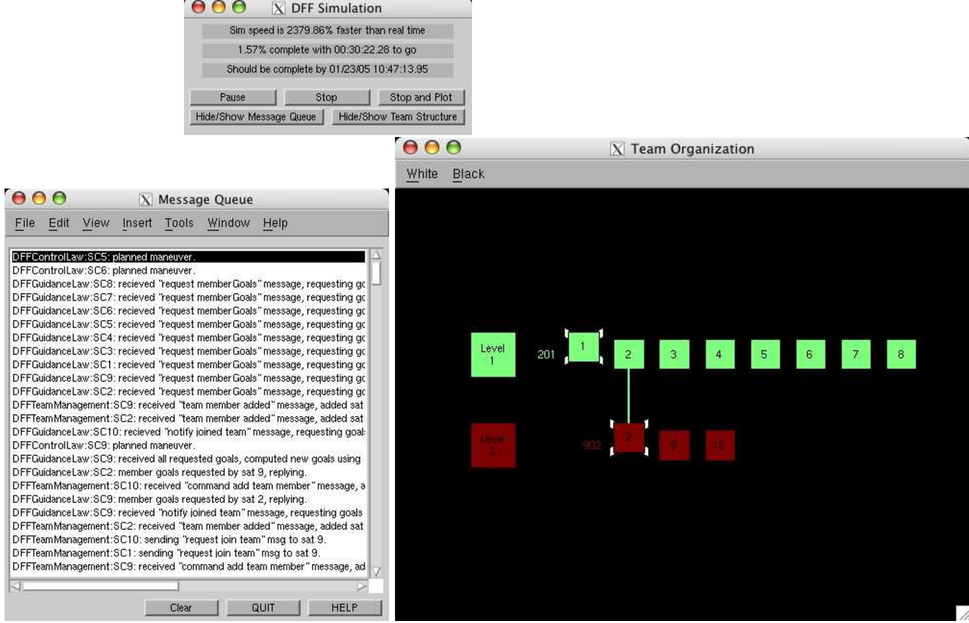
```
>> sim = AutoFormSimStruct;
>> d = DFFSimulation( sim );
```

The set of commands for this demo is generated with the file `AutoFormCommandList.m`. Command lists are discussed in Section 3.5.1 on page 17.

Within the simulation, the modules are initialized and updated using the executive function, `DFFControl`.

A small status window pops up once the simulation begins. It displays the simulation speed, the percent complete, and an estimate of how much time remains. The buttons allow you to pause and resume the simulation, stop it and quit, or stop it and plot the resulting data. In addition, you may elect to show or hide the Message Queue and the Team Organization displays at any time. Hiding these windows allows the simulation to run faster. Example screenshots of the Message Queue, status window, and Team Organization display are shown in Figure 3-3

Figure 3-3. DFF Simulation Windows



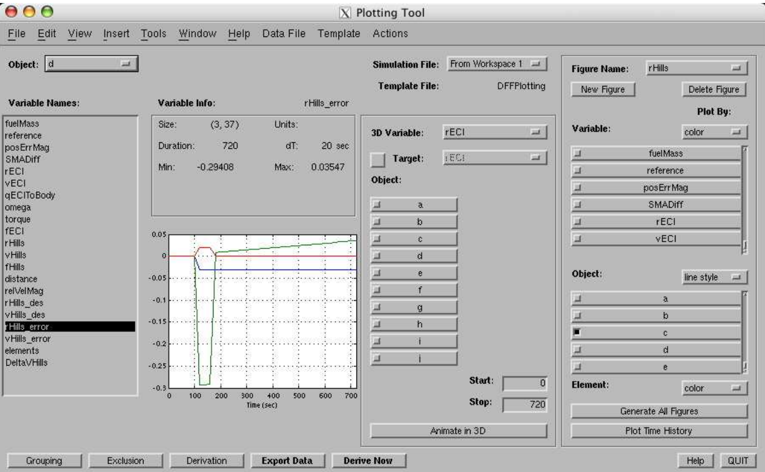
The Message Queue is used as a means of logging the activities of the software. Messages are displayed as the software runs to notify the user of any predefined actions or events of interest. Each message is generated with a single call to the `MessageQueue` function; the syntax is similar to that shown in the `otherwise` case of the module template (Listing 3.1 on page 9).

The Team Management display provides a graphic representation of the hierarchical team organization. This is particularly useful in scenarios where the team structure changes over time. As the simulation runs, the team data structure is repeatedly obtained from the Team Management module on each spacecraft. The resulting array of team data structures is then supplied to the `ShowTeams` function, which creates the graphic depiction.

Once a simulation is complete, or once the "Stop and Plot" button is pushed, the raw simulation data is loaded into the `PlottingTool`. The data set includes the mission elapsed time, the 14 states (as described in Table 3-1 on the facing page), the applied force and torque, the reference ID, and (if the `recordTgtState` input was true), the desired relative position and velocity in Hills-frame. A template is available for the DFF simulation data. Applying the

template named `DFFPlotting.mat` from the "Template" menu of the GUI will clean up the raw data and provide a more meaningful set of data to work with. It will group single position and velocity elements into vectors, transform inertial states into the relative frame, and derive other useful data, such as the distance to all other satellites, and the position error magnitude. A screenshot of the `PlottingTool` is shown in Figure 3-4.

Figure 3-4. PlottingTool with DFF Simulation Data



3.5 Initializating the DFF Software

The initialization procedure is an important element of the DFF design. One of the objectives with the software is to enable re-initialization of individual modules at run-time. This feature will enable individual modules to be killed and restarted at any time, without having to reboot the entire system. In addition, it will support the dynamic loading and initialization of new modules as the original system continues to run. Another objective is to make the software as generic as possible, so that it can be applied with minimal changes to a variety of formation flying missions.

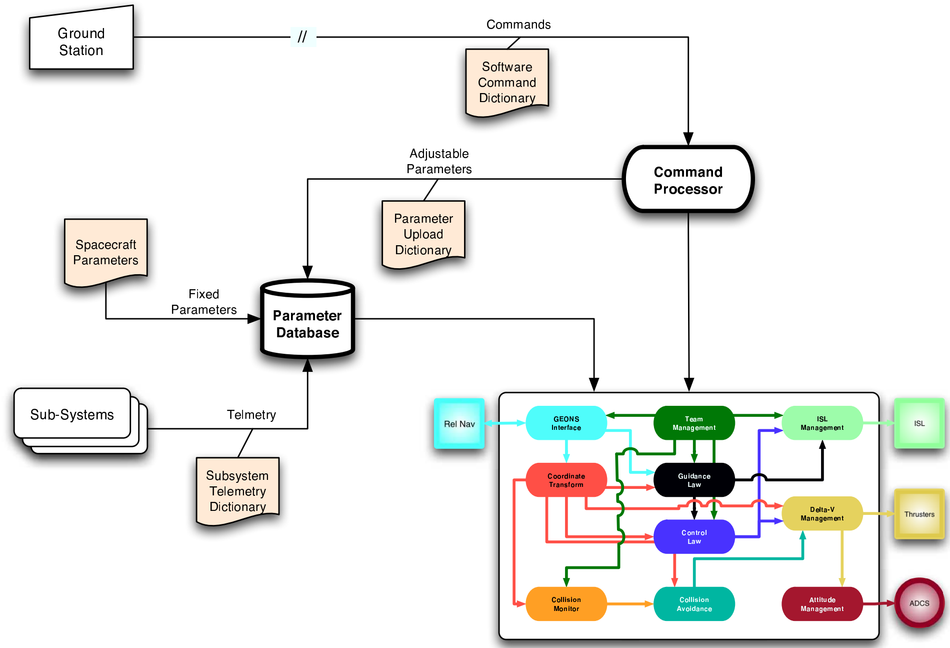
Consider the block diagram shown in Figure 3-5 on the facing page. The Command Processing and Parameter Database modules are the focus of this diagram. The point is to illustrate the flow of data through these modules and how they are initialized.

Every module in the DFF system requires a particular set of data for it to run. In general, the different types of data may be distinguished as follows:

- Fixed
- Variable
  - Parameters uploaded from the ground
  - Telemetry data obtained from subsystems

Examples of fixed data would include the spacecraft dry mass and inertia, the direction of fixed solar panels in the body frame, and the bandwidth of the ISL. Fixed data consists primarily of hardware measures that cannot change over the course of the mission. Variable data includes parameters that may be adjusted over time via ground commands, and time-varying data obtained from the spacecraft's subsystems. Examples of adjustable parameters would include the position error deadband and the geometric goals. The mass of fuel remaining in each fuel tank is an example

Figure 3-5. DFF Block Diagram with Initialization Scripts



of required subsystem telemetry. The following three initialization scripts are used to initialize the DFF system with these three different types of data.

- "Spacecraft Parameters"
- "Parameter Upload Dictionary"
- "Subsystem Telemetry Dictionary"

Figure 3-5 shows that these three scripts are supplied to the Parameter Database module. This module is initialized first. All other modules in the DFF system initialize themselves with data obtained from Parameter Database. In this manner, when a module is re-initialized at run-time, it uses the most current data from the Parameter Database.

A fourth initialization script that appears in the diagram is the "Software Command Dictionary" file, which is used by the Command Processing module. The job of this module is to receive commands from the ground station (via the comm system) and forward the commands in the form of messages to the appropriate modules. Two types of commands may be sent, according to the following naming conventions:

- *set* #
- *command* #

where the # represents the remainder of the command name string. The *set* messages include data, whereas the *command* messages do not. In the case of *set* messages, the # corresponds to a specific parameter, and it must be a single word without spaces. The following capitalization format is used for the parameters:

```
firstSecondLast
```

The Command Processing module checks the name of each command, and then prepares and sends a message to all modules who have registered for that command. All *set* messages are also forwarded to the Parameter Database so that

the included data may be stored. The Command Processing module is initialized with two files: a "Software Command Dictionary" for the data-less *command* messages, and a "Parameter Upload Dictionary" for the *set* messages. Each file contains a table of commands with the following columns:

- Command Name
- Destination Module
- Default Data
- Description

The description field is included for clarity purposes only; it is not used by the software. The default data column is blank in the "Software Command Dictionary", as its messages contain no data. In the "Parameter Upload Dictionary", the entries in this column are string representations of default data for each command. Also, commands may have more than one destination module, in which case multiple entries (multiple rows in the table) are required.

The Parameter Database receives data from three sources: 1) *set* messages forwarded from the Command Processing module (as discussed above), 2) telemetry data obtained from the subsystems, and 3) fixed spacecraft data. As previously mentioned, the module is therefore initialized with the following three files:

- "Parameter Upload Dictionary"
- "Subsystem Telemetry Dictionary"
- "Spacecraft Parameters"

The "Parameter Upload Dictionary" contains the list of all *set* commands. Embedded in the name of each command is the name of corresponding adjustable parameter. The "Subsystem Telemetry Dictionary" has the following columns:

- Data Name
- Subsystem Name
- Description

This information informs the Parameter Database as to what pieces of telemetry are to be obtained from which subsystems.

The "Spacecraft Parameters" file contains only two columns:

- Data Name
- Data Value

All three files are loaded into the Parameter Database at initialization. It uses the contents of the files to create a set of member variables that are named the same as the data names listed in the files. Three sets of member variables are generated: adjustable parameters, time-varying telemetry data, and fixed spacecraft parameters. The adjustable parameters and fixed spacecraft parameters are initialized with the values provided in the initialization files. The values for the telemetry data, however, are not included in the initialization file. These values are obtained directly from the subsystems.

3.5.1 Command Lists

In the real-time implementation of the DFF software, commands would be composed and sent to the system intermittently from a remote operator. This type of real-time interactive functionality will be supported in the MANTA-based system. For the prototype, however, there is no need to manually send commands one-at-a-time. Rather, we wish to compose predefined command lists that are designed to exercise specific features of the software.

As noted in Table 3-1 on page 12, a specific command list is loaded for each simulation. Several command list functions have already been written in support of various demos. A basic template is provided in `DefaultCommandList.m`. A portion of the template is given below.

Listing 3.3. `DefaultCommandList.m` *Command List*

```
function cmd = DefaultCommandList

% help header omitted here for brevity

% Initialize Command Structure array
%----------------------------------
cmd = Command_Structure(5);
k   = 0;

% Fill-in information for each command
%------------------------------------

k = k + 1;
cmd(k).timeTag = 1;
cmd(k).scID    = 1:6;
cmd(k).module  = 'DFFControlLaw';
cmd(k).command = 'command_control_on';
cmd(k).data    = [];
```

Command List

The function returns an array of command data structures. The format for a command data structure is defined in `Command_Structure.m`. The array is initialized first – in this case, a total of 5 commands are generated. In the next block of code, the fields in the first element of the array are defined. The `timeTag` field specifies the mission elapsed time at which the command is to be processed. The `scID` field indicates which spacecraft the command is sent to. the `module` field indicates the destination module. The `command` field specifies the name of the command. All commands must begin with either *set* or *command*. Finally, the `data` field contains any data that may be included with the command. In this case, the `'command control on'` message is sent to the Control Law on spacecraft ID's 1 through 6, at 1 second into the simulation.

The command list function is called by the executive `DFFControl` function during initialization. The commands are first sorted chronologically, then each command is distributed to the appropriate destination spacecraft by storing it in the corresponding elements of a cell array. This cell array of commands is then passed in to the Command Processing module when it is initialized. It therefore begins the simulation with the complete set of ground commands. Each time the module updates, it processes those commands whose time-tag is less than or equal to the current time, and then removes those commands from memory.

In the real DFF system, new commands will be received as the system runs. The same type of time-tagged processing is still necessary, though, because in many cases the ground station operator will want to send a batch of time-tagged commands while the satellite is within view. An additional support module should be developed for automated testing purposes. It would enable predefined sets of commands to be sent to the Command Processing module over time to avoid having to always send the commands manually.

CHAPTER 4. Software Modules

This section describes the software modules that compose the DFF prototype system.

Each section is organized into the following subsections:

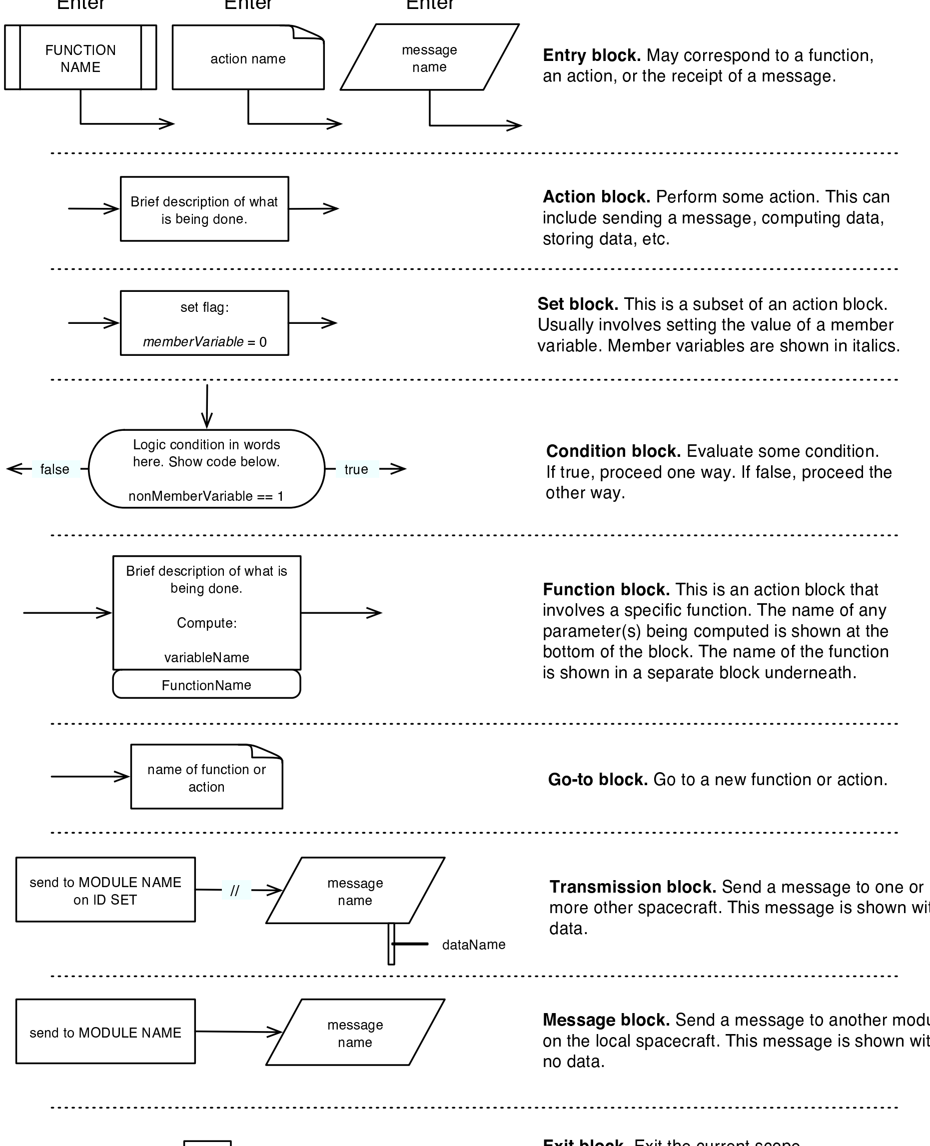
- **Scope** – Summarizes the main purpose and overall functionality of the module.
- **Messages** – Lists all input and output messages. Messages are divided into the following categories:
  - *Input Messages* – Incoming messages from other modules.
  - *Request Messages* – Messages sent from another module requesting data.
  - *Output Messages* – Outgong messages sent to other modules.
- **Required Functions** – Lists all functions that are called from within the module.
- **Member Variables** – Lists all member variables.
- **Functionality** – Describes the complete functionality of the module.

Note that although every module has a "`get updatePeriod##`" request message, these messages are not used in the prototype system. They are therefore not included in the list of request messages.

In addition, a subset of the modules must send messages to other spacecraft. The details of this process are described in Section 4.9 on page 58. All inter-spacecraft messages are sent through the ISL Management module via the "transmit" message. In the sections that follow, when an asterisk (*) is shown after the source or destination module, it indicates an inter-spacecraft message. The message first sent to the ISL Management module on the source spacecraft, is transmitted over the ISL, is then received at the ISL Management module on the destination spacecraft, and is finally sent to the destination module.

The modules that require inter-spacecraft communication exhibit distributed functionality. Identical copies of the same software operate on multiple processors, sharing information to coordinate their behavior. A series of flowcharts have been developed for each of these modules in order to more clearly describe the sequence of events and the flow of data in the distributed system. A complete set of more than 50 linked flowcharts is available in an OmniGraffle document entitled "DFFSystem.graffle". All of the diagrams are provided as an appendix to this document. A subset are included within the main body of this document to illustrate the primary aspects of some modules' functionality. The legend for these flowcharts is shown in Figure 4-1 on the following page.

Figure 4-1. Legend for Module Flowcharts



Although the flowcharts explicitly describe the prototype system, they were written with the intent of describing the system in a platform-independent fashion. Therefore, they should provide a useful blueprint from which to build the real-time system in MANTA.

The description of the *entry block* makes reference to an "action". This does not correspond to a particular function, and has nothing to do with the `action` input of each module. Rather, it simply corresponds to a piece of the module's code that is best represented with a separate flowchart.

4.1 Command Processing

`DFFCommandProcessing`

Scope

The purpose of the Command Processing module is to receive commands that are sent from the ground station operator and forward them to the appropriate modules. Much of this functionality is inherently built-in to MANTA. For example, each incoming command could simply be broadcast to all tasks, and it would be received by only those tasks with an input of the same name.

Messages

This module is designed to handle a dynamic set of inputs and outputs. A predefined set of command messages is defined in the "Software Command Dictionary" and "Parameter Upload Dictionary" initialization scripts. This file lists the names of all command messages, as well as the modules to which the commands should be forwarded. Two types of commands may be sent, according to the following naming conventions:

- *set* #
- *command* #

where the # represents the remainder of the command name string. The *set* messages include data, whereas the *command* messages do not. In the case of *set* messages, the # corresponds to a specific parameter, and it must be a single word without spaces. The following capitalization format is used for the parameters: `firstSecondLast`.

It is anticipated that, in the real-time system, much of the message-passing functionality in this module will be automatically handled by the built-in features of MANTA.

The set of input messages for *this* module is summarized in Table 4-1. All other messages are routed to other modules, and are therefore described in other sections of this chapter.

Table 4-1. Input Messages

Message Name	Data	Source Module	Description
resetCP	-	Ground Comm Interface Plugin	Causes the module to run the Initialization function.
set updatePeriodCP	`double`	Ground Comm Interface Plugin	Sets the update period (in seconds).
register	`double`	-	Add the supplied message name and correspodning module name to the list of outputs.
print	`double`	User	Causes the module to run the `PrintToFile` member function. Prints all outputs and destinations to a tab-delimited file.

There are no request messages sent to the Command Processing module.

All of the *set* and *command* messages that are received by this module are forwarded to other modules in the system (with the exception of `set updatePeriodCP`, which is meant only for this module). These are the only output messages sent by the Command Processing module.

Required Functions

The functions required by the Command Processing module are listed in Table 4-2.

Table 4-2. Required Functions

Function Name	Description
strmatch	Compare a string with a cell array of strings, find the index of the array that matches.
feval	Evaluate a function given its name as a string.
ReceiveCommands	Member function. Checks pre-loaded list of time-tagged commands, returns those commands whose time-tag is past the mission elapsed time, and removes them from memory.
AddOutput	Member function. Adds the specified message name and destination module to a cell array of outputs and destinations.
PrintToFile	Member function. Print all outputs and destinations to a tab-delimited file.

Member Variables

The member variables for the Command Processing module are listed in Table 4-3.

Table 4-3. Member Variables

Variable Name	Data	Description
iD	int	Unique spacecraft ID (positive)
updatePeriod	double	Update period (sec)
timeLastUpdate	double	Time at which the module updated last (JD)
commandList	command[]	Array of pre-defined command data structures
newCommands	command[]	Array for newly received command data structures
inputs	char[][]	Array of input names for commands to be forwarded (same as output names)
outputs	char[][]	Array of output names (based on "Software Command Dictionary")
destinations	char[][]	Array of destination modules

Functionality

As previously discussed, the primary role of this module is to receive commands from the ground and forward them to other modules. In this prototype design, commands are "sent" to the DFF system by initializing the Command Processing module with a pre-defined command list. The module also has the capability to receive commands as it runs. This would require either a user-interface for composing and sending commands during the simulation, or an additional support module designed to send the commands at the appropriate times.

Each time the module updates, it calls the ReceiveCommands function. This checks the pre-loaded list of time-tagged commands, returns those commands whose time-tag is past the mission elapsed time, and then removes them from memory. The output of this function, therefore, is an array of command data structures that are to be processed. In the real-time system, these data structures would be sent, as they are received, from an interface plugin that interfaces with the Ground Comm subsystem.

The code from the Update function is shown in Listing 4.1. The mission elapsed time, mET, is passed in instead of the Julian date, for convenience. The RecieveCommands function is called first. For each newly received command, the name of the command is compared with the list of known outputs. If a match is found, the message is sent (with any attached data) to all destination modules associated with this command. The MATLAB feval function is used to call the module function given its name as a string.

Listing 4.1. DFFCommandProcessing :: Update() *DFFCommandProcessing :: Update()*

```
function d = Update( d, mET )
```

```
% Receive commands
%-----------------
[receivedCommands, d.commandList] = ReceiveCommands( d.newCommands, d.commandList, mET );
d.newCommands = {};

% cycle through all received commands
%------------------------------------
nC = length(receivedCommands);
for i=1:nC,

   cmd = receivedCommands{i};

   % find the output index
   %----------------------
   p = strmatch( lower(cmd.command), lower(d.outputs), 'exact' );

   if( ~isempty(p) )
      module  = d.destinations{p};     % the list of destination modules
      data    = cmd.data;              % the corresponding data
      for j = 1:length(module)
         feval( module{j}, cmd.command, data, d.iD );  % route to all destination modules
         MessageQueue( 'add',['DFFCommandProcessing:SC',num2str(d.iD)], ...
            sprintf('Commmand␣"%s"␣sent␣to␣%s',cmd.command,module{j}));
      end
   else
      MessageQueue( 'add',['DFFCommandProcessing:SC',num2str(d.iD)], ...
         sprintf('%s␣is␣an␣unknown␣command',cmd.command),'error');
   end

end   % end cycle through received commands
```

DFFCommandProcessing :: Update()

The member variables `d.outputs` and `d.destinations` are generated in the initialization function, and are based solely upon the “Software Command Dictionary” and “Parameter Upload Dictionary” files. All command messages listed in the “Parameter Upload Dictionary” are *set* commands, and include data. Each of these commands is also forwarded to the Parameter Database for storage.

4.2 Parameter Database

`DFFParameterDatabase`

Scope

The purpose of the module is to provide a central repository for various types of data that may be accessed by the other modules. As discussed in Section 3.5 on page 14, the DFF system requires three different types of data:

- Fixed spacecraft parameters
- Parameters uploaded from the ground
- Telemetry data obtained from subsystems

This module loads three separate files to initialize these three differet sets of data. In this manner, the DFF software may be applied to a variety of different spacecraft configurations and mission scenarios, without having to change the software.

Messages

Like the Command Processing module, the Parameter Database is designed to handle a dynamic set of inputs and outputs. It receives a copy of all *set* commands from the ground, and obtains specified telemetry data from specified subsystems. The nominal list of *set* commands is provided in the "Parameter Upload Dictionary" file, and the nominal list of telemetry data and corresponding subsystems is provided in the "Telemetry Dictionary" file. Because the messages are summarized in these initialization files, they are not repeated here.

The set of input messages specifically for *this* module is summarized in Table 4-4. In addition, all *set* commands sent

Table 4-4. Input Messages

Message Name	Data	Source Module	Description
resetPD	-	Command Processing	Causes the module to run the Initialization function.
set updatePeriodPD	`double`	Command Processing	Sets the update period (in seconds).

to the DFF system are copied to this module.

The Parameter Database module has a dynamic set of request messages. Any parameter that has been stored may be obtained with a `get parameterName` message.

The output messages sent from the Parameter Database module are identical to the *set* messages that come in as inputs.

Required Functions

The functions required by the Parameter Database module are listed in Table 4-5 on the next page.

Member Variables

The nominal set of member variables for the Parameter Database module are listed in Table 4-6 on the facing page.

Table 4-5. Required Functions

Function Name	Description
`strmatch`	Compare a string with a cell array of strings, find the index of the array that matches.
`feval`	Evaluate a function given its name as a string.
`eval`	Evaluate an expression provided as a string. Used to store member variables given the parameter names.

Table 4-6. Member Variables

Variable Name	Data	Description
iD	`int`	Unique spacecraft ID (positive)
updatePeriod	`double`	Update period (sec)
timeLastUpdate	`double`	Time at which the module updated last (JD)
telemetryNames	`char[][]`	Array of telmetry parameter names
telemetrySubsystems	`char[][]`	Array of subsystem names corresponding to each telmetry parameter

Functionality

The Parameter Database is different from all other modules in that it does not require an update function. It behaves passively, only running when it receives a message from another module. Two different types of messages can be received:

- `set parameterName` – Sent from the Command Processing module. Causes the data in the message to be stored to the local copy of `parameterName`.
- `get parameterName` – Sent from any other module. Causes the local copy of `parameterName` to be returned.

When this module is initialized, the complete set of parameters contained in the 3 initialization scripts are stored as member variables. The name of each member variable matches the name supplied in the *set* and *get* messages.

When a *set* message is received, the member variable corresponding to the specified parameter is updated with data embedded in the message. If a member variable by this name does not yet exist, a new one is generated. In the MANTA framework, this may be accomplished by dynamically adding new inputs & outputs to the module. Because this involves dynamic memory allocation, some method of protection must be built-in to ensure the module does not exceed its memory capacity.

When a *get* message is received, one of two responses may be taken. If the requested parameter is a “fixed” or “uploadable” parameter, the currently stored copy is returned to the requesting module. However, if the requested parameter is telemetry data, then the corresponding subsystem is polled for the specified data. This requires, of course, that an interface to all such subsystems be maintained.

In the current implementation of the DFF prototype, the software requires telemetry data from only one subsystem: propulsion. The fuel mass is required by the Guidance Law, Control Law, Team Management and Delta-V Management modules. The Delta-V Management module also requires the tank pressure and thruster status.

In the real-time DFF system, it may be desirable for the Parameter Database to periodically poll all interfaced subsystems for the complete set of telemetry. Alternatively, each interface plugin might be instructed to periodically supply the telemetry to the Parameter Database at specified intervals. In either case, the telemetry data would be immediately available in the Parameter Databasee when requested by another module.

4.3 Relative Navigation

`DFFRelativeNavigation`

Scope

The sole purpose of the Relative Navigation module is to provide the absolute and relative states to the Coordinate Transformation module. The absolute state is defined as the position and velocity of the reference spacecraft in the ECI frame. The relative state is defined as the relative position and velocity in a translated ECI frame, centered at the reference spacecraft.

In the prototype design, this module simply obtains the true state data from the simulation and provides it to the Coordinate Transformation module. Random noise is added according to the standard deviations specified in the initialization function. In the MANTA design, this module will maintain an interface with the GEONS software. GEONS will use GPS measurements with an extended Kalman filter to estimate the absolute and relative states.

Messages

The set of input messages that may be sent to the Relative Navigation module is summarized in Table 4-7.

Table 4-7. Input Messages

Message Name	Data	Source Module	Description
resetRN	-	Command Processing	Causes the module to run the Initialization function.
set updatePeriodRN	`double`	Command Processing	Sets the update period (in seconds).
set referenceID	`int`	Team Management	Sets the reference ID.

The set of request messages that may be sent to the Relative Navigation module is summarized in Table 4-8.

Table 4-8. Request Messages

Message Name	Source Module	Description
get navigation data	Coordinate Transformation	Return the navigation data to the sender. Includes absolute position and velocity of reference; relative position and velocity of local spacecraft with respect to reference; and the measurement time.
get reci	Delta-V Management, ADCS	Return my ECI position vector to the sender.
get veci	Delta-V Management, ADCS	Return my ECI velocity vector to the sender.
get referenceID	-	Return the reference ID to the sender.

The only message sent from the Relative Navigation module is “get x”. This is sent to the State Sensor to obtain the ECI position and velocity of the specified spacecraft. The State Sensor is supporting software, not an actual DFF module. It is used to temporarily store the true state information from the simulation, so that it may be obtained from this and other modules in order to test the software.

Required Functions

The functions required by the Relative Navigation module are listed in Table 4-9 on the next page.

Table 4-9. Required Functions

Function Name	Description
JD2SS1970	Convert Julian date time to "seconds since 1970".

Member Variables

The member variables for the Relative Navigation module are listed in Table 4-10.

Table 4-10. Member Variables

Variable Name	Data	Description
iD	int	Unique spacecraft ID (positive)
updatePeriod	double	Update period (sec)
timeLastUpdate	double	Time at which the module updated last (JD)
rECI	matrix[3,1]	ECI position vector of myself (km)
vECI	matrix[3,1]	ECI velocity vector of myself (km/s)
rECIRef	matrix[3,1]	ECI position vector of reference (km)
vECIRef	matrix[3,1]	ECI velocity vector of reference (km/s)
rECIRel	matrix[3,1]	Relative ECI position vector (km)
vECIRel	matrix[3,1]	Relative ECI velocity vector (km/s)
time	double	Time associated with relative navigation measurement (SS1970)
absPosNoise	double	Standard deviation of absolute position noise (km)
absVelNoise	double	Standard deviation of absolute velocity noise (km/s)
relPosNoise	double	Standard deviation of relative position noise (km)
relVelNoise	double	Standard deviation of relative velocity noise (km/s)

Functionality

The functionality of this module is best described by showing the MATLAB code from its update function. The first input `d` is the data structure that holds all member variables. This data structure is modified during the update and then returned. The current Julian date, `jD`, is passed in as the second input. The code is shown in Listing 4.2 on the next page.

The state data corresponding to this spacecraft is obtained first, and is stored in the member variables `d.rECI` and `d.vECI`. Next, the state data of the reference is obtained. The ID of the reference spacecraft is stored in `d.refID`, and is defined via the `'set referenceID'` message that is sent from the Team Management module. Next, the relative position and velocity is computed by subtracting the reference state from this spacecraft's state. Random noise is then added to all position and velocity vectors. Finally, the current time is stored in `d.time`, in units of "seconds since 1970". This is defined to be the time at which the measurements were taken.

The functionality of this module will be completely different in the real-time system. It will maintain an interface with the GEONS software, obtain the state estimates from it periodically, and make these estimates available to other software modules.

Listing 4.2. Relative Navigation :: Update() *Relative Navigation :: Update()*

```
function d = Update( d, jD )

% obtain truth values for abs pos & vel of local satellite in ECI frame
x      = StateSensor('get_x',[],d.iD);
d.rECI = x(1:3);
d.vECI = x(4:6);

% obtain truth values for abs pos & vel of team reference in ECI frame
if( d.refID ~= d.iD )
   x         = StateSensor('get_x',[],d.refID);
   d.rECIRef = x(1:3);
   d.vECIRef = x(4:6);
else
   d.rECIRef = d.rECI;
   d.vECIRef = d.vECI;
end

% compute relative pos & vel in ECI frame
d.rECIRel = d.rECI - d.rECIRef;
d.vECIRel = d.vECI - d.vECIRef;

% add noise
if( d.iD ~= d.refID )
   d.rECI    = d.rECI    + d.absPosNoise*randn(3,1);
   d.vECI    = d.vECI    + d.absVelNoise*randn(3,1);
   d.rECIRef = d.rECIRef + d.absPosNoise*randn(3,1);
   d.vECIRef = d.vECIRef + d.absVelNoise*randn(3,1);
   d.rECIRel = d.rECIRel + d.relPosNoise*randn(3,1);
   d.vECIRel = d.vECIRel + d.relVelNoise*randn(3,1);
end

% Time information
%-----------------
d.time = JD2SS1970(jD);
```

Relative Navigation :: Update()

4.4 Coordinate Transformation

`DFFCoordinateTransformation`

Scope

The purpose of the Coordinate Transformation module is to transform the absolute and relative state estimates into different coordinate frames required by other software modules.

Messages

The set of input messages that may be sent to the Coordinate Transformation module is summarized in Table 4-11.

Table 4-11. Input Messages

Message Name	Data	Source Module	Description
resetCT	-	Command Processing	Causes the module to run the Initialization function.
set updatePeriodCT	`double`	Command Processing	Sets the update period (in seconds).

The set of request messages that may be sent to the Coordinate Transformation module is summarized in Table 4-12.

Table 4-12. Request Messages

Message Name	Source Module	Description
get state	Guidance Law, Control Law, Delta-V Management, Collision Monitor	Return the transformed state data to the sender. Includes the orbital elements of the reference; orbital element differences between local spacecraft and the reference; relative Hill's-frame state of local spacecraft with respect to the reference; and the measurement time.

The set of messages sent from the Coordinate Transformation module to other modules is summarized in Table 4-13.

Table 4-13. Output Messages

Message Name	Data	Destination Module	Description
get navigation data	-	Relative Navigation	Obtain the navigation data. Includes absolute position and velocity of reference; relative position and velocity of local spacecraft with respect to reference; and the measurement time.

Required Functions

The functions required by the Coordinate Transformation module are listed in Table 4-14 on the next page.

Table 4-14. Required Functions

Function Name	Description
`AbsRelECI2Hills`	Transforms absolute and relative ECI states into a relative Hill's-frame state.
`ECI2MeanElements`	Transforms osculating ECI position and velocity to mean orbital elements.
`Alfriend2El`	Transforms standard orbital elements to the Alfriend set of elements.

Member Variables

The member variables for the Coordinate Transformation module are listed in Table 4-15.

Table 4-15. Member Variables

Variable Name	Data	Description
`iD`	`int`	Unique spacecraft ID (positive)
`updatePeriod`	`double`	Update period (sec)
`timeLastUpdate`	`double`	Time at which the module updated last (JD)
`state`	`state`	Data structure of absolute and relative state information

Functionality

The functionality of the Coordinate Transformation module is straightforward. Each time it updates, it carries out the following steps:

1. Obtain the current estimate of the absolute and relative states from the Relative Navigation module.
2. Compute the relative position and velocity in Hill's-frame coordinates, using `AbsRelECI2Hills`.
3. Compute the mean orbital elements and orbital element differences, using `ECI2MeanElements`.
4. Compute the Alfriend orbital element set, using `El2Alfriend`.
5. Store the results in a *state* data structure.

The state data is then supplied to other modules upon their request.

4.5 Team Management

DFFTeamManagement

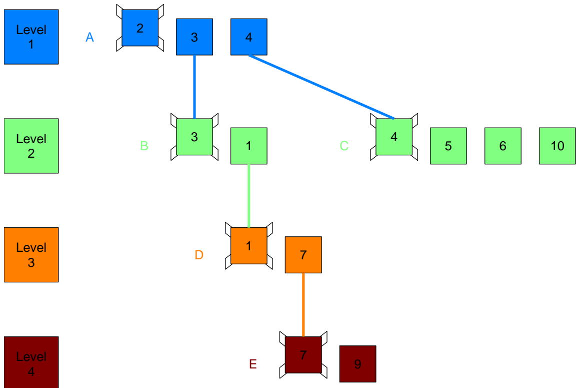
Scope

The job of the Team Management module is to maintain a hierarchical team organization for the cluster. In particular, this includes the following two functions:

1. Autonomous team formation
2. Autonomous reference rollover

Each team is composed of one reference spacecraft, which defines the origin of the relative frame, and one or more relatives. Within a given team the desired trajectories of all relative satellites are defined with respect to the reference, so that the reference has no desired trajectory within that team. In a hierarchy of teams, the reference of each team doubles as a relative on a higher level team. The exception is the reference of the highest level team, which serves as the cluster reference. An example illustration of a multiple-team hierarchy is shown in Figure 4-2. Teams are grouped together by proximity, with the team name shown just to the left. The reference of each team is the located to the left, slightly elevated above the other members, with an "X" behind it. Here, satellite #2 is the cluster reference. Although the teams are given lettered *names* here, in the DFF software they are distinguished by unique team IDs.

Figure 4-2. Example Team Organization



In general, a given satellite may be on an unlimited number of teams. However, it can be a relative on *only one* team at any instant. This rule, which is enforced by the Team Management module, prevents the possibility of assigning conflicting geometric goals to a satellite on multiple teams.

In addition to the reference, each team also has one captain. The captain is responsible for carrying out a limited number of tasks that require centralized coordination.

The "autonomous team formation" capability enables a cluster of spacecraft to self-initialize themselves into a multiple-team hierarchy. This is done in a completely distributed and decentralized manner, with no single spacecraft having any authority over another. The team organization can also be defined manually through commands from the ground.

Once a team organization is established, this module monitors the remaining fuel percentage on all team members. This is done to promote equal fuel usage among team members over time. Since the team reference does not control to a desired relative trajectory for that team, it expends less fuel than the relatives over time. We therefore wish to change the identity of the reference once the difference in remaining fuel between the current reference and one or more relatives becomes sufficiently large. This is termed an "autonomous reference rollover".

Messages

The set of input messages that may be sent to the Team Management module is summarized in Table 4-16.

Table 4-16. Input Messages

Message Name	Data	Source Module	Description
resetTM	-	Command Processing	Force the module to run the Initialization function.
command auto ref roll on	-	Command Processing	Enable autonomous reference rollover.
command auto ref roll off	-	Command Processing	Disable autonomous reference rollover.
command auto cap roll on	-	Command Processing	Enable autonomous captain rollover.
command auto cap roll off	-	Command Processing	Disable autonomous captain rollover.
command acquisition on	-	Command Processing	Enable autonomous team formation.
command acquisition off	-	Command Processing	Disable autonomous team formation.
command reference rollover	`int[]`	Command Processing, Team Management*	Initiate a reference rollover. Make the specified member ID the new reference of the specified team ID.
command remove team member	`int[]`	Command Processing, Team Management*	Remove the specified member ID from the specified team ID.
command add team member	`int[]`	Command Processing, Team Management*	Add the specified member ID to the specified team ID.
command add team member	`team`	Command Processing, Team Management*	Add my ID to the supplied team data structure, and store this new team in my existing array of teams.
set updatePeriodTM	`double`	Command Processing	Set the update period (in seconds).
set knownIDs	`int[]`	Command Processing	Set the array of known satellite IDs.
set joinRequestPeriod	`double`	Command Processing	Set the time period (in seconds) for sending “request join team” messages. Only used if autonomous acquisition is enabled.
set maxMembers	`int`	Command Processing	Set the maximum allowed number of members on any team.
set maxTeams	`int`	Command Processing	Set the maximum allowed number of teams for any member.
set teamData	`team[]`	Command Processing	Set the team data structure array. Defines the team organization.
set captainID	`int[]`	Command Processing	Set the captain ID for the specified team ID.
request join team	`int[]`	Team Management*	Request from another satellite to join my team.
notify team formation	`int`	Team Management*	Notification from another satellite that a new team has been formed. Data includes the captain ID of the new team. All subsequent “request join team” messages are sent to this ID.

Table 4-16. Input Messages, contd.

Message Name	Data	Source Module	Description
team member added	int[]	Team Management*	Notification from another team member that a new member has been added.
update team info	team[], int[]	Team Management*	Notification from another team member that other teams have been modified.
inform memberGoals arr	geometry	Team Management*	Notification of the geometric goals from another satellite in support of auonomous reference rollover in a multiple-team setting.

The set of request messages that may be sent to the Team Management module is summarized in Table 4-17.

Table 4-17. Request Messages

Message Name	Source Module	Description
get team data	Guidance Law	Return the team data structure array to the sender.
get local team data	Guidance Law	Return only the set of team data structures of which I am a member.
get relative status	Guidance Law	Return a true/false flag indicating whether I am a relative.
get reference status	Guidance Law, Control Law	Return a true/false flag indicating whether I am a reference.
get captain status	Guidance Law	Return a true/false flag indicating whether I am a captain.
get captain id	Guidance Law	Return the ID of the captain for the team on which I am a relative.
get captain id for team	Guidance Law	Return the ID of the captain for the specified team ID.
get relative ids for team	Guidance Law, Collision Monitor	Return the IDs of all relatives for the specified team ID.
get relative ids for reference	Guidance Law, Control Law	Return the IDs of all relatives for the specified reference ID.
get reference id for member	Guidance Law	Return the ID of the reference for the specified member ID.
get reference id for team	Guidance Law	Return the ID of the reference for the specified team ID.
get member ids for team	Guidance Law, ISL Management	Return the IDs of all members for the specified team ID.
get member ids for member	Guidance Law	Return the IDs of all members for the specified member ID.
get team ids for member	Guidance Law, ISL Management, Collision Monitor	Return the IDs of all teams which have the specified member ID.
get team id for captain	Guidance Law	Return the ID of the team which has the specified captain ID.

The set of output messages sent from the Team Management module to other modules is summarized in Table 4-18.

Table 4-18. Output Messages

Message Name	Data	Destination Module	Description
transmit	isl_message	ISL Management	Transmit the attached message over the ISL to another spacecraft.
command reference rollover	int[]	Team Management*	Initiate a reference rollover. Make the specified member ID the new reference of the specified team ID.
command remove team member	int[]	Team Management*	Remove the specified member ID from the specified team ID.
command add team member	int[]	Team Management*	Add the specified member ID to the specified team ID.

Table 4-18. Output Messages, contd.

Message Name	Data	Destination Module	Description
command add team member	`team`	Team Management*	Add my ID to the supplied team data structure, and store this new team in my existing array of teams.
request join team	`int[]`	Team Management*	Request from another satellite to join my team.
notify team formation	`int`	Team Management*	Notification from another satellite that a new team has been formed. Data includes the captain ID of the new team. All subsequent "request join team" messages are sent to this ID.
team member added	`int[]`	Team Management*	Notification from another team member that a new member has been added.
update team info	`team[]`, `int[]`	Team Management*	Notification from another team member that other teams have been modified.
request memberGoals arr	`int`	Guidance Law*	Request the geometric goals from the Guidance Law on the previous reference. Used in support of auonomous reference rollover in a multiple-team setting.
get memberGoals	-	Guidance Law	Obtain the geometric goals from the Guidance Law.
update memberGoals for new team	`geometry`	Guidance Law	Instruct the Guidance Law to update its geometric goals in response to an autonomous reference rollover.
reference change	`int[]`	Guidance Law	Notify the Guidance Law that the team reference has changed.
clear memberGoals	-	Guidance Law	Clear the geometric goals.
notify joined team	`int`	Guidance Law	Notify the Guidance Law that this satellite has joined a team.
clear goals	-	Control Law	Clear the geometric goals.
pause	-	Control Law	Command the Control Law to pause. A "resume" command will be sent later.
resume	-	Control Law	Command the Control Law to resume.
notify team member added	`int[]`	Control Law	Notify the Control Law that a team member has been added.
get fuelMass	`int[]`	Parameter Database	Obtain the remaining fuel mass of all team members.
set referenceID	`int`	Relative Navigation	Instruct the Relative Navigation module of the new reference ID.

Required Functions

The functions required by the Team Management module are listed in Table 4-19.

Table 4-19. Required Functions

Function Name	Description
`iscell`	ISCELL(C) returns 1 if C is a cell array and 0 otherwise.
`isfield`	ISFIELD(S,'name') returns 1 if 'name' is a field in the structure array S and 0 otherwise.
`isstruct`	ISSTRUCT(S) returns 1 if S is a structure and 0 otherwise.
`intersect`	INTERSECT(A,B) when A and B are vectors returns the values common to both A and B.
`setdiff`	SETDIFF(A,B) when A and B are vectors returns the values in A that are not in B.

Table 4-19. Required Functions, contd.

Function Name	Description
`unique`	UNIQUE(A) for the array A returns the same values as in A but with no repetitions.
`ISLMessage_Structure`	Initialize an ISL message data structure.
`Team_Structure`	Initialize a team data structure.
`FSWClock`	Access the flight software clock to obtain the current time.
`JD2SS1970`	Convert Julian date to seconds since 1970.
`TeamLevels`	Assign a hierarchical level to each team in the array.
`MessageQueue`	Display messages to a GUI while the software runs. Used for validation purposes only.
`CaptainStatus`	Member function. Determine whether the specified member ID is a captain.
`ReferenceStatus`	Member function. Determine whether the specified member ID is a reference.
`RelativeStatus`	Member function. Determine whether the specified member ID is a relative.
`CaptainIDForMember`	Member function. Find the captain ID for the specified member ID.
`CaptainIDForTeam`	Member function. Find the captain ID for the specified team ID.
`RelativeIDsForTeam`	Member function. Find the IDs of all relatives on the specified team ID.
`RelativeIDsForReference`	Member function. Find the IDs of all relatives that have the specified reference ID.
`ReferenceIDForMember`	Member function. Find the reference ID for the specified member ID.
`ReferenceIDForTeam`	Member function. Find the reference ID for the specified team ID.
`MemberIDsForTeam`	Member function. Find the IDs of all members on the specified team ID.
`MemberIDsForMember`	Member function. Find the IDs of all members that are on the same team as the specified member ID.
`ExtendedMemberIDsForMember`	Member function. Find the IDs of all members that are in the same hierarchy as the specified member ID.
`TeamIDsForMember`	Member function. Find the IDs of all teams that the specified member is on.
`NumberOfTeamsForMember`	Member function. Find the number of teams that the specified member is on.
`TeamIDForRelative`	Member function. Find the ID of the team which has the specified member ID as a relative.
`TeamIDForCaptain`	Member function. Find the ID of the team which has the specified captain ID.
`TeamIndex`	Member function. Find the index of the specified team ID.
`UniqueTeamID`	Member function. Generate a unique team ID.
`SetTeamData`	Member function. Update the team data structure array and implement all required functionality associated with the changes.
`CreateNewTeam`	Member function. Create and add a new team to the team data structure array and implement all required functionality associated with the creation of the new team.
`SendTeamUpdateNotification`	Member function. Notify other teams of changes to this team.
`ComputeJoinRequestTime`	Member function. Compute the next time to send a “request join team” message, and the ID to send to.
`RemoveTeamMember`	Member function. Remove the specified member from the specified team.

Many of the member functions directly support the retrieval of specific team-related information for a corresponding request message. For example, the member function `ReferenceIDForTeam` is called when a “`get reference id for team`” message is received.

Member Variables

The member variables for the Team Management module are listed in Table 4-20 on the following page.

Table 4-20. Member Variables, contd.

Variable Name	Data	Description

Table 4-20. Member Variables

Variable Name	Data	Description
`iD`	`int`	Unique spacecraft ID (positive)
`updatePeriod`	`double`	Update period (sec)
`timeLastUpdate`	`double`	Time at which the module updated last (JD)
`knownIDs`	`int[]`	
`maxTeams`	`int`	
`maxMembers`	`int`	
`team`	`team`	
`amCaptain`	`int`	Flag indicating whether I am a captain or not
`amReference`	`int`	Flag indicating whether I am a reference or not
`amRelative`	`int`	Flag indicating whether I am a relative or not
`autoRefRollOn`	`int`	Flag indicating whether autonomous reference rollover is enabled
`autoCapRollOn`	`int`	Flag indicating whether autonomous captain rollover is enabled
`rolloverRFPDiff`	`double`	Minimum allowable difference in remaining fuel percentage
`timeLastRefRollReq`	`double`	Time at which the module sent the last reference rollover request (SS1970)
`rolloverReqWaitTime`	`double`	Amount of time to wait for a rollover request to take effect before requesting again (sec)
`awaitingGoalsForAutoRefRoll`	`int`	Flag indicating whether I am awaiting goals for an autonomous reference rollover
`previousReferenceGoals`	`geometry`	Previous geometric goals of the current reference
`initialFuelMass`	`double`	Initial mass of fuel in all tanks (kg)
`acquisitionOn`	`int`	Flag indicating whether autonomous acquisition is enabled
`joinRequestPeriod`	`double`	Time period for sending "request join team" messages (sec)
`timeLastRequest`	`double`	Time at which the last "request join team" message was sent
`notifyTeamFormationRecd`	`int`	Flag indicating whether I have been notified of a new team formation
`newTeamCaptainID`	`int`	Spacecraft ID of the newly formed team
`addingMember`	`int`	Flag indicating whether I am in the process of adding a new team member
`addingMemberID`	`int`	Spacecraft ID of team member being added
`outgoing`	`isl_message[]`	Array of messages that are to be sent to the ISL Management module at the next update

Functionality

As discussed earlier, the Team Management module provides two main functions: autonomous team formation, and autonomous reference rollover. These functions are initiated in the `Update` function. In addition, the module also provides a variety of team-related information to other modules, as shown in Table 4-17 on page 33.

Each request message asks for some piece of data that is directly obtained from the team data structure. Therefore, each message has a corresponding member function that is used to compute the requested data. Each of these member functions has two inputs. One input is the team data structure. The other input is either the data supplied with the request message (if any), or the ID of the local spacecraft. For example, the request message "`get relative ids for reference`" is sent with the reference ID as data. The `RelativeIDsForReference` function is called with the team data structure and the supplied reference ID to compute the array of relative IDs. The code for this function is shown below.

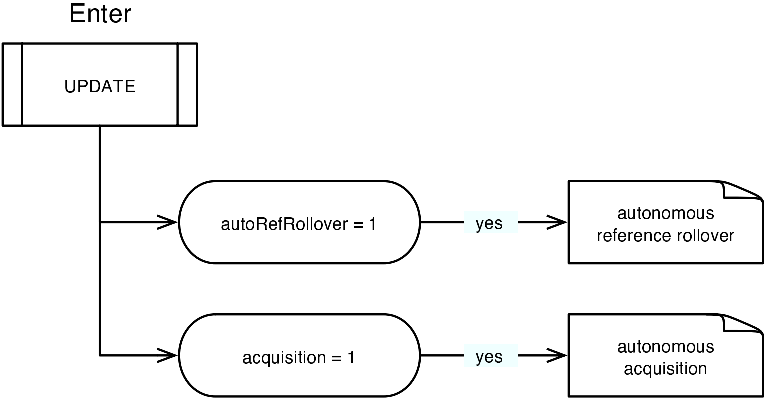
Listing 4.3. `RelativeIDsForReference` *Team Management :: RelativeIDsForReference()*

```
function [iDs,k] = RelativeIDsForReference( refID, teams )
iDs   = [];                          % default answer is empty, in case the supplied ID is not a reference for
      any listed teams
k     = [];
for i=1:length( teams )              % cycle through all teams that this s/c is a member of
   if( teams(i).refID == refID )     % if the supplied s/c ID is a reference for the ith team...
      iDs = [iDs, setdiff(teams(i).memID,teams(i).refID)];   % tack on the relative IDs for this team
      k   = [k, i];
   end
end
return
```

Team Management :: RelativeIDsForReference()

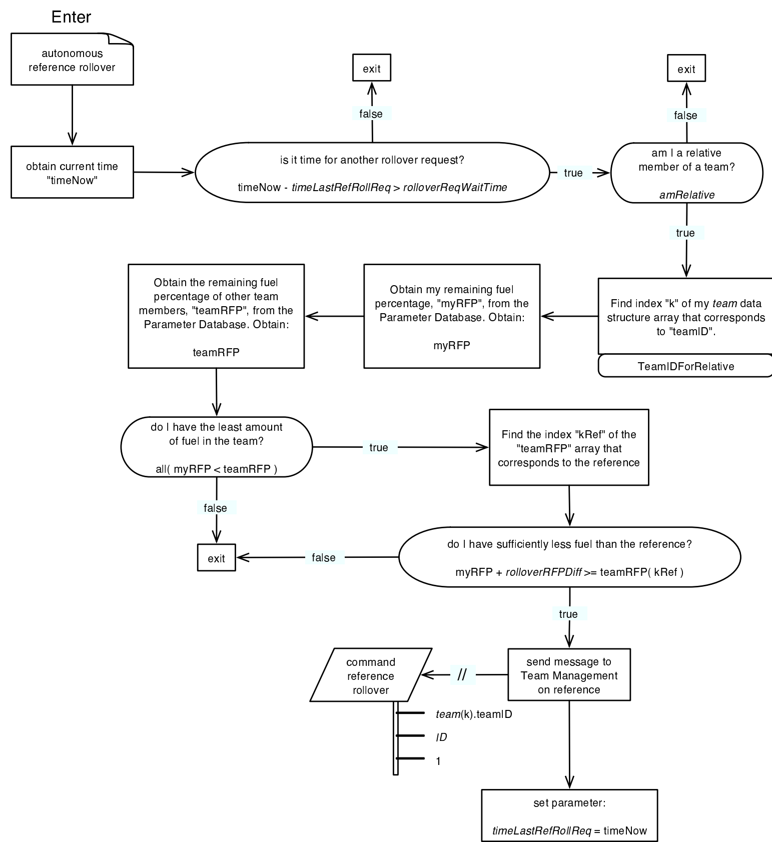
The autonomous team formation and reference rollover operations are initiated in the `Update` function. Because these actions are inherently distributed, however, their full functionality extends beyond the `Update` function and includes the actions that are taken in response to the receipt of different messages. The flowchart for the `Update` function is shown in Figure 4-3. See Figure 4-1 on page 20 for a legend of the various flowchart components.

Figure 4-3. Team Management Update



The `autoRefRollover` flag is set to true in response to a “command auto ref roll on” message, and is set to false with a “command auto ref roll off” message. If true, the module carries out the steps shown in Figure 4-4 on the following page.

Figure 4-4. Autonomous Reference Rollover



4.6 Guidance Law

DFFGuidanceLaw

Scope

The purpose of the Guidance Law module is to define the desired relative trajectory of the spaceraft. This information is supplied to the Control Law, which is responsible for achieving and maintaining the desired trajectory. The desired trajectory for each spacecraft is defined with respect to its reference, and is expressed as a set of geometric goals.

The Guidance Law provides several features:

- Distributed assignment of geometric goals for a team
- Distributed assignment of geometric goals for a cluster
- Automatic computation of team goals based on high-level objectives
- Acquisition of all geometric goals in a hierarchy through a recursive search mechanism
- Automatic determination of safe trajectories upon addition to a team or cluster
- Redefinition of geometric goals to support autonomous reference rollovers in a multiple-team hierarchy

All of these features are supported in both circular and eccentric orbits.

Messages

The set of input messages that may be sent to the Guidance Law module is summarized in Table 4-21.

Table 4-21. Input Messages

Message Name	Data	Source Module	Description
resetGL	-	Command Processing	Causes the module to run the Initialization function.
command zero drift	-	Command Processing	Command the Guidance Law to compute a new set of geometric goals that match the current trajectory, but that have zero relative drift.
command achieve formation	-	Command Processing, Guidance Law*	Command the previously specified formation (type, size, etc.) to be achieved.
set assignmentMethod	`int`	Command Processing	Sets the method to be used in the assignment process (privileged == 1, optimal == 2).
set costMetric	`int`	Command Processing	Sets the cost metric to be used for the privileged assignment method.
set formationType	`int`	Command Processing	Sets the formation type. Several types are supported, and are defined in the `GenerateTeamGoals` function.
set formationSize	`double`	Command Processing	Sets the length of the formation baseline.
set formationOrientation	`matrix`[3,1]	Command Processing	Sets the euler angles to rotate the specified geometry through.
set formationLocation	`double`	Command Processing	Sets the location in the orbit (true anomaly) where the specified orientation is to occur.
set timeWindow	`window`	Command Processing	Sets the time window data structure to use in maneuver planning.
set nSPO	`int`	Command Processing	Sets the number of samples per orbit to use for LP planning algorithms.
set assignmentTimeLimit	`double`	Command Processing	Sets the time limit (in seconds) for waiting for a cost estimate response during a distributed assignment process.
set assignmentAttemptLimit	`int`	Command Processing	Sets the limit on the number of attempts for conducting a distributed assignement process.
set minSepDistance	`double`	Command Processing	Sets the minimum along-track separation distance for autonomous goal selection.
set maxSepDistance	`double`	Command Processing	Sets the maximum along-track separation distance for autonomous goal selection.
set angularResolution	`double`	Command Processing	Sets the angular resolution to be used in the assignment algorithm.
set memberGoals	`geometry`	Command Processing, Guidance Law*	Sets the desired geometric goals for the local spacecraft.
set teamGoals	`team_goals`	Command Processing, Guidance Law*	Sets the desired geometric goal set for the team.
set clusterGoals	`team_goals`	Command Processing, Guidance Law*	Sets the desired geometric goal set for the entire cluster.
inform cluster state	`state`	Guidance Law*	Recursively provides the state information of the reference to all relatives.
inform cluster cost estimate	`cost`	Guidance Law*	Recursively provides the cost estimate of the relative (and all of its relatives) to the reference.
inform team cost estimate	`cost`	Guidance Law*	Provides the cost estimate of a relative to the captain.
distribute clusterGoals	`team_goals`	Guidance Law*	Recursively provides the assigned goals to all members of the cluster.

Table 4-21. Input Messages, contd.

Message Name	Data	Source Module	Description
notify joined team	`int`	Team Management	Causes the geometric goals to be computed automatically once a team has been joined. Used during the autonomous team formation process.
inform memberGoals	`geometry`	Guidance Law*	Recursively provides the geometric goals of all other spacecraft in the cluster. Sent in response to a "request memberGoals" message. Used during the autonomous team formation process.
update memberGoals for new team	`geometry`	Team Management	Causes the geometric goals to be recomputed if added to a new team. Used during the reference rollover process.
inform new reference goals	`geometry`	Guidance Law*	Provides the previous geometric goals of the new reference. Used during the reference rollover process.
reference change	`int[]`	Team Management	Instructs the Guidance Law that it has become a reference. Causes it to send an "inform new reference goals" message to all relative IDs. Used during the reference rollvoer process.
clear memberGoals	-	Team Management	Clear the geometric goals.

The set of request messages that may be sent to the Guidance Law module is summarized in Table 4-22.

Table 4-22. Request Messages

Message Name	Source Module	Description
get memberGoals	Team Management	Return the geometric goals to the sender.
request memberGoals	Guidance Law*	Forward this message to all relative IDs (if any). Once all have responded, reply with an "inform memberGoals" message to the Guidance Law on the requesting spacecraft ID.
request memberGoals arr	Team Management*	Reply with an "inform memberGoals arr" message to the Team Management module on the requesting spacecraft ID.

The set of output messages sent from the Guidance Law module to other modules is summarized in Table 4-23.

Table 4-23. Output Messages

Message Name	Data	Destination Module	Description
transmit	`isl_message`	ISL Management	Transmit the attached message over the ISL to another spacecraft.
get state	-	Coordinate Transformation	Obtain the state information.
get fuelMass	-	Parameter Database	Obtain the remaining fuel mass.
set goals	-	Control Law	Sets the geometric goals.
get captain id	-	Team Management	Obtain the ID of the captain for the team on which I am a relative.
get captain id for team	`int`	Team Management	Obtain the ID of the captain for the specified team ID.
get member ids for member	`int`	Team Management	Obtain the IDs of all members for the specified member ID.
get member ids for team	`int`	Team Management	Obtain the IDs of all members for the specified team ID.
get relative ids for reference	`int`	Team Management	Obtain the IDs of all relatives for the specified reference ID.
get relative ids for team	`int`	Team Management	Obtain the IDs of all relatives for the specified team ID.

Table 4-23. Output Messages, contd.

Message Name	Data	Destination Module	Description
get relative status	-	Team Management	Obtain a true/false flag indicating whether I am a relative.
get reference id for member	`int`	Team Management	Obtain the ID of the reference for the specified member ID.
get reference id for team	`int`	Team Management	Obtain the ID of the reference for the specified team ID.
get reference status	-	Team Management	Obtain a true/false flag indicating whether I am a reference.
get local team data	-	Team Management	Obtain the set of team data structures of which I am a member.
get team id for captain	`int`	Team Management	Obtain the ID of the team which has the specified captain ID.
get team ids for member	`int`	Team Management	Obtain the IDs of all teams which have the specified member ID.
inform team cost estimate	`cost`	Guidance Law*	Provides the cost estimate of a relative to the captain.
request memberGoals	`int[]`	Guidance Law*	Request the geometric goals of a spacecraft, including all of its relatives and/or superiors.
inform memberGoals	`geometry`	Guidance Law*	Response to a “request memberGoals” message.
inform memberGoals arr	`geometry`	Team Management*	Response to a “request memberGoals arr” message.
set clusterGoals	`team_goals`	Guidance Law*	Recursively forward the cluster goals to all team members.
inform cluster state	`state`	Guidance Law*	Recursively provide the state information of the reference to all relatives.
inform cluster cost estimate	`cost`	Guidance Law*	Recursively provide the cost estimate of the relative (and all of its relatives) to the reference.
distribute clusterGoals	`geometry[]`	Guidance Law*	Recursively provide the assigned goals to all members of the cluster.
inform new reference goals	`geometry[]`	Guidance Law*	Provide the previous geometric goals of the new reference. Used during the reference rollover process.
command achieve formation	-	Guidance Law*	Forward the command to achieve the previously specified formation (type, size, etc.) to the captain.
set teamGoals	`team_goals`	Guidance Law*	Provide the team goals to all relative team members.
set memberGoals	`geometry`	Guidance Law*	Provide the assigned geoemtric goals to all relative team members.

Required Functions

The functions required by the Guidance Law module are listed in Table 4-24.

Table 4-24. Required Functions

Function Name	Description
`iscell`	ISCELL(C) returns 1 if C is a cell array and 0 otherwise.
`isfield`	ISFIELD(S,'name') returns 1 if 'name' is a field in the structure array S and 0 otherwise.
`setdiff`	SETDIFF(A,B) when A and B are vectors returns the values in A that are not in B.
`unique`	UNIQUE(A) for the array A returns the same values as in A but with no repetitions.

Table 4-24. Required Functions, contd.

Function Name	Description
`EccGeometry_Structure`	Initialize an eccentric geometric goals data structure.
`Geometry_Structure`	Initialize a circular geometric goals data structure.
`ISLMessage_Structure`	Initialize an ISL message data structure.
`FFEccEqns`	Compute the Hills frame state an a different point of the trajectory given the initial state, initial and final true anomaly, and eccentricity
`AutoFormGeometry`	Define new geometric goals for a single satellite, such that any semi-major axis difference is eliminated, and the new trajectory maintains a minimum separation distance from all other team members trajectories in the team.
`EstimateCost`	Estimate the weighted cost to achieve all specified unique target states (for a circular reference orbit).
`FFEccEstimateCost`	Estimate the weighted cost to achieve all specified unique target states (for an eccentric reference orbit).
`FFEccGenerateTeamGoals`	Generate a Team Goals data structure given the formation type, size, location, and orientation.
`GenerateTeamGoals`	Generate a Team Goals data structure given the formation type and size.
`InitializeCostMatrix`	Given the team goals, initialize the cost matrix with the right size.
`OptimalAssignment`	Assign target states to satellites using the optimal assignment method.
`PopulateCostMatrix`	Fill in the single column of the cost matrix that corresponds to the given spacecraft ID.
`PrivilegedAssignment`	Assign target states to satellites using the priveleged assignment method.
`RestrictIDSet`	Given an initial set of relative spacecraft IDs, examine the constraints to determine which of these IDs should be included.
`SetupAssignmentProblem`	Compute the set of parameters from the team goals that define the assignment problem.
`SortTeamGoals`	Sort the team goals structure so that fixed states appear before variable states.
`AddGoals`	Add one set of geometric goals to the other.
`DeltaElem2Goals`	Transform a mean element difference set to a circular geometric goal set.
`FFEccDeltaElem2Hills`	Transform a mean element difference set to a Hills-frame state (for an eccentric reference orbit).
`FFEccHills2Goals`	Transform a Hills-frame state to a set of eccentric geometric goals
`GeometryCirc2Ecc`	Convert a cicular goals structure to an eccentric goals structure.
`GeometryEcc2Circ`	Convert an eccentric goals structure to a circular goals structure.
`JD2SS1970`	Convert Julian date to seconds since 1970.
`RotateState`	Transform a circular geometric goal set by rotating it through a prescribed angle.
`SubGoals`	Subtract two one set of geometric goals from the other.
`IsCircGeom`	Determine whether a geometric goals data structure is of the circular type or not.
`IsEccGeom`	Determine whether a geometric goals data structure is of the eccentric type or not.
`MessageQueue`	Display messages to a GUI while the software runs. Used for validation purposes only.
`Hills2DeltaElem`	Transform a Hills-frame state to mean element differences.
`OrbRate`	Compute the mean orbit rate from the semi-major axis.
`M2Nu`	Compute the true anomaly from the mean anomaly.
`BuildRequestIDs`	Member function. Build the set of IDs to forward “request memberGoals” messages to.
`FormatGoals`	Member function. Ensure goals have the proper format for the eccentricity.
`MapGoalsToSats`	Member function. Map geometric goals to satellites.
`PrepareMemberGoals`	Member function. Prepare member goals in response to recursive “request memberGoals” message.
`ReduceClusterGoals`	Member function. Reduce the cluster goals array by excluding the goal sets that correspond to member IDs that are on the same team of which I am a relative.

Member Variables

The member variables for the Guidance Law module are listed in Table 4-25.

Table 4-25. Member Variables

Variable Name	Data	Description
iD	int	Unique spacecraft ID (positive)
updatePeriod	double	Update period (sec)
timeLastUpdate	double	Time at which the module updated last (JD)
awaitingTeamCostEst	int	Flag indicating whether I am awaiting cost estimates for a team assignment
amCaptain	int	Flag indicating whether I am a captain
provideCostEstimate	int	Flag indicating if I am to provide a cost estimate to the captain
newTeamGoals	int	Flag indicating whether new team goals have been received
responded	int[]	Array of IDs that have responded with an "inform team cost estimate" message
zeroDrift	int	Flag indicating whether to achieve zero-drift
achieveFormation	int	Flag indicating whether to achieve the previously specified formation
joinedTeam	int	Flag indicating whether a new team has been joined
teamJoined	int	ID of team that has been joined
achievingFormation	int	Flag indicating whether a formation is currently being achieved
assignmentAttempts	int	Counter for the number of times a distributed assignment process has been attempted
assignmentTime	double	Time that the distributed assignment process was last initiated (SS1970)
newReferenceGoals	int	Flag indicating whether previous goals of the new reference have been received
nMemberGoalsRequested	int	Number of geometric goal sets requested from other members
nMemberGoalsResponded	int	Number of members that have responded with a geometric goals set
awaitingMemberGoals	int	Flag indicating whether I am awaiting geometric goals from other members
requestedOtherMemberGoals	int	Flag indicating whether I have requested geometric goals from other members
haveMemberGoals	int	Flag indicating whether geometric goals have been assigned
newMemberGoals	int	Flag indicating whether new geometric goals were just received
window	window	Time window data structure for maneuver planning
memberGoals	geometry	Geometric goals data structure defining the desired relative trajectory
teamGoals	team_goals	Team goals data structure defining the desired trajectories for a team
clusterGoals	team_goals	Team goals data structure defining the desired trajectories for the cluster
formationType	int	Desired formation type. Several types are supported, and are defined in the GenerateTeamGoals function.
formationSize	double	Desired formation size (km)
formationLocation	double	Desired location in the orbit where the specified formation geometry is to occur (rad)
formationOrientation	matrix[3,1]	Euler angles describing the desired orientation of the formation geometry at "formationLocation"

Table 4-25. Member Variables, contd.

Variable Name	Data	Description
fuelWeightExponent	double	Fuel weighting exponent (for promoting equal fuel usage)
assignmentMethod	int	Which assignment method to use (1==privileged, 2==optimal)
costMetric	int	Which cost metric to use with the privileged assignment method (1==minimum, 2==average)
minSepDistance	double	Minimum along-track separation distance for autonomous goal computation (km)
maxSepDistance	double	Maximum along-track separation distance for autonomous goal computation (km)
assignmentTimeLimit	double	Maximum time to wait for a cost estimate response during a distributed assignment process (sec)
assignmentAttemptLimit	int	Maximum number of times to attempt a distributed assignment process
angularResolution	double	Angular resolution to be used for the assignment algorithm (rad)
eTol	double	Eccentricity tolerance (treat as circular orbit below, eccentric above)
nSPO	int	Number of samples per orbit to use in LP maneuver planning
otherMemberGoals	geometry[]	Array of geometric goals for other members in the cluster
memberGoalsRequested	int	Flag indicating whether geometric goals were requested of me
memberGoalsRequestingID	int	ID of spacecraft that requested geometric goals of me
memberGoalsRequestedFARR	int	Flag indicating whether geometric goals were requested of me in support of an autonomous reference rollover
memberGoalsRequestingIDFARR	int	ID of spacecrfat that requested geometric goals of me in support of an autonomous reference rollover
newClusterGoals	int	Flag indicating whether new cluster goals have been received
clusterCostEstimates	cost[]	Array of cost estimate data structures used in the distributed assignment of cluster goals
clusterGoalsSentToIDs	int[]	Array of IDs that cluster goals have been sent to
clusterRespondedIDs	int[]	Array of IDs that have responded with a cost estimate for achieving cluster goals
clusterGoalsRelativeIDs	int[]	Array of relative IDs that must respond with cost estimates to achieve the cluster goals
awaitingRefClusterState	int	Flag indicating whether I am awaiting the cluster-based state of my reference
haveRefClusterState	int	Flag indicating whether I have received the cluster-based state of my reference
awaitingClusterCostEst	int	Flag indicating whether I am awaiting cost estimates for achieving cluster goals
refClusterState	state	Cluster-based state information of my reference
clusterGoalsAssigned	int	Flag indicating whether cluster goals have been assigned
clusterGoalsAssignment	struct	Data structure containing an array of cluster-based geometric goals, and the corresponding spacecraft IDs
priorGoals	geometry	Previous geometric goals
oldRefGoals	geometry	Geometric goals of the previous reference
updateGoalsForNewTeam	int	Flag indicating whether to update the geometric goals in response to joining a new team
initialFuelMass	double	Initial mass of fuel in all tanks (kg)
outgoing	isl_message[]	Array of messages that are to be sent to the ISL Management module at the next update

Functionality

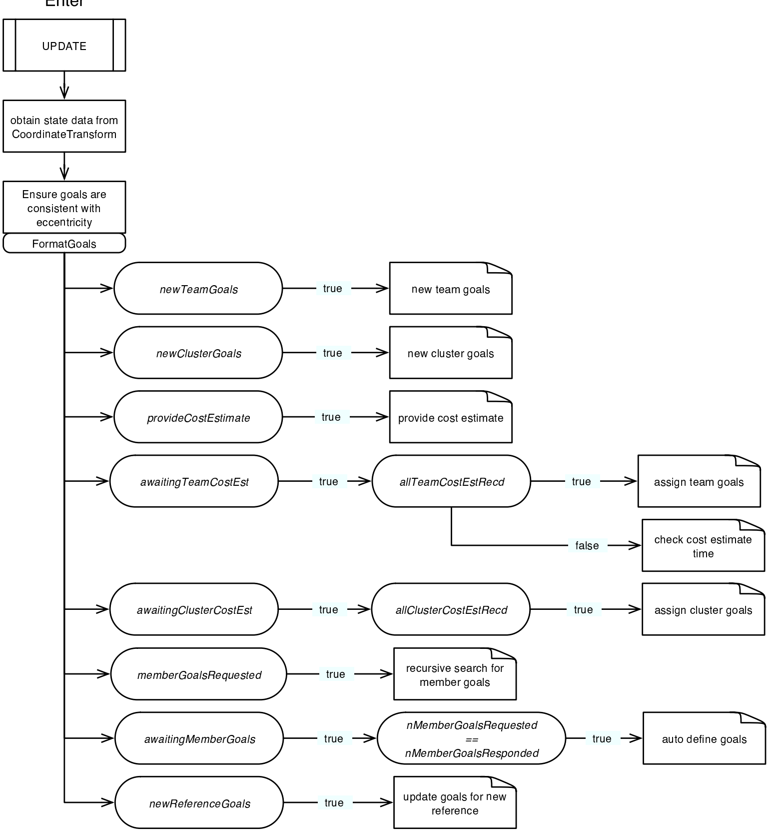
The purpose of the Guidance Law module is to define the desired relative trajectory for the spacecraft. The Guidance Law provides several features, including:

- Distributed assignment of geometric goals for a team
- Distributed assignment of geometric goals for a cluster
- Automatic computation of team goals based on high-level objectives
- Acquisition of all geometric goals in a hierarchy through a recursive search mechanism
- Automatic determination of safe trajectories upon addition to a team or cluster
- Redefinition of geometric goals to support autonomous reference rollovers in a multiple-team hierarchy

All of these features are supported in both circular and eccentric orbits.

This module, while designed to be as efficient and straightforward as possible, contains a considerable amount of functionality. Much of this functionality is distributed in nature, in that the complete process consists of a series of steps that are carried out on multiple spacecraft at different times. It is therefore best to describe the operations of the module in a graphical setting. The flowchart for the `Update` function is shown in Figure 4-5. See Figure 4-1 on page 20 for a legend of the various flowchart components. This simply illustrates the initial flow of logic that takes

Figure 4-5. Guidance Law Update



the module into the various aspects of functionality during each update period. For example, the `newTeamGoals`

flag is set to true if a new set of team goals are either commanded or computed. If this flag is true, the module enters the "new team goals" functional block, which is detailed in subsequent diagrams that are provided in the appendix.

Additional background material is provided in the following section.

Background

Formations are designed by combining several relative trajectories in such a way that a desired geometric configuration is achieved. In order to realize a formation, each spacecraft must achieve one of the relative trajectories. It is typically not important which spacecraft achieves which trajectory, as long as all desired trajectories are met so that the overall formation geometry is realized. Therefore, the main function of the formation guidance law is that of trajectory assignment. The objective is to achieve all desired trajectories in such a way that minimizes the total cost.

The guidance law designed for the DFF system operates at both the team and cluster level. The set of target states for an individual team or for the entire cluster may be defined by the operator according to three different levels of autonomy:

- **Low Autonomy** – The target state for each spacecraft is defined explicitly. It may be defined in Hill's frame coordinates, orbital element differences, or in terms of the geometric parameters. The control law immediately tracks the new desired trajectory.
- **Medium Autonomy** – A set of geometric goals is supplied, where each goal set corresponds to a desired trajectory. The guidance law works to find the optimal configuration, assigning each target to a different member such that the total cost is minimized.
- **High Autonomy** – A few high-level objectives are supplied, such as a pre-defined formation type, size, and orientation. The geometric goals for the team/cluster are computed online to meet the given objectives. Each target is then assigned to a different member such that the total cost is minimized.

The guidance law is not used in the "Low Autonomy" case, as the target states are assigned directly from the operator. At the medium and high autonomy levels, however, the targets are assigned on-orbit. The general procedure is as follows:

1. The geometric goals for the team are either supplied to or computed by the team captain.
2. The captain distributes the team goals to all relative team members.
3. Each recipient uses the control law to estimate the cost to achieve all desired trajectories.
4. The vector of costs from each relative member is returned to the captain.
5. The captain assembles all cost vectors into a single cost matrix.
6. The captain applies an assignment algorithm to the cost matrix to find a solution that minimizes the total cost.
7. The captain sends out the newly assigned geometric goals to each relative team member.

The guidance law is therefore both distributed and centralized. The cost estimation is distributed across all satellites in the team, but the assignment task is performed centrally, by the captain.

The total cost is defined as a weighted sum of the total delta-v's for each spacecraft to achieve its target state. The individual delta-v's are weighted according to their remaining fuel percentage, in order to promote equal fuel usage throughout the cluster. The cost for the ith satellite to reach the jth target state is:

$$c_{i,j} = f_i^{-x} \times \Delta V_{ij} \tag{4.6-1}$$

where f_i is the remaining fuel percentage of the ith spacecraft, and $x > 0$ is an adjustable parameter indicating the importance of fuel equalization. The general problem is to assign N target states to M satellites such that the total cost is minimized. This results in a $M \times N$ cost matrix. In the case where $N \neq M$, the cost matrix is made square by adding rows or columns whose elements are much larger than the maximum value of the original matrix.

Two different approaches to solving the assignment problem have been implemented within the Guidance Law. The first approach, termed the "optimal method", involves searching over all possible permutations to find the one with the minimum total cost. The total number of unique permutations is $N!$ for a square cost matrix of size N. This approach is therefore computationally cumbersome as N becomes large, i.e., ≥ 8. The advantage is that a globally optimal solution is guaranteed.

The second approach is called the "privileged method". This technique requires considerably less computation, but does not guarantee that a globally optimum solution is found. It consists of the following steps:

1. Determine the minimum projected cost of each satellite.
2. Determine which satellite has the highest minimum cost.
3. Assign that satellite to the target state corresponding to its minimum cost.
4. Repeat steps 1-2 for all remaining members and remaining target states.

Whichever method is used, the ground operator has the flexibility of restricting the target state distribution. Each target state may be restricted so that it is considered by only a specific subset of team members. This is accomplished in the cost estimation stage by forcing $c_{i,j}$ to be an extremely large number for cases where the jth target state is not allowed to be assigned to the i_{th} satellite. The assignment algorithm naturally avoids these high cost combinations.

The guidance law has also been designed to take advantage of the additional freedom available in circular reference orbits. Here, we allow two types of target states to be identified: *fixed* and *variable*. With the fixed target state, we specify both the desired state and the point in the orbit at which it is to occur. With the variable target state, we consider all possible states along the trajectory. When multiple target states are defined along the same trajectory, the original phase separation is maintained as a constraint. The details of this "variable state" method require a lengthy discussion and are beyond the scope of this paper.

In eccentric orbits, only fixed target states are possible. This is due to the restricted nature of the relative trajectories in eccentric orbits, where each state along a given trajectory must occur at a particular true anomaly. In the cost estimation stage of the guidance law, the fixed target state is defined by using the true anomaly associated with the longest allowable maneuver duration.

When assigning target states to a cluster composed of multiple teams, they are initially uploaded to the cluster reference. They are then transmitted to all relatives in the top-level team. If any of these relatives also serves as a reference for a lower-level team, it transmits the targets to its relatives, and so on in a recursive manner until the targets have been sent to all spacecraft in the cluster. The target states are all defined with respect to the cluster reference, so the relative state of each spacecraft must also be computed in the same frame. Therefore, along with the target state information, each reference also sends its relative state with respect to the cluster reference. This enables each satellite to compute its relative state in the cluster frame. Consider the ith spacecraft in the cluster, whose reference is spacecraft j. Let the C subscript denote the cluster frame, and T denote the team frame. The relative state of the ith spacecraft in the cluster frame is then:

$$x_{C,i} = x_{C,j} + x_{T,i} \tag{4.6-2}$$

where $x_{C,j}$ is transmitted from spacecraft j, and $x_{T,i}$ is readily available from the Coordinate Transformation module.

Once each satellite has the target states and its relative state in the cluster frame, it computes the set of costs to achieve all targets, and transmits the cost vector to back to its reference. Each reference in the hierarchy compiles the cost vectors from its relatives and then sends the data up to its reference, until finally the cluster reference has the complete set of cost data from all satellites in the cluster. The privileged assignment method is then used to quickly determine

the best possible assignment, and the targets are distributed accordingly throughout the teams. The final step is for each relative satellite to transform the target state to its team-based coordinate frame before reconfiguring to the new trajectory.

4.7 Control Law

`DFFControlLaw`

Scope

The purpose of the Control Law module is to achieve and maintain the desired relative trajectory. This target trajectory is provided by the Guidance Law as a set of geometric goals. Each time the module updates, it computes the error between the desired relative position (defined by the geometric goals) and the measured relative position. If the error in any axis exceeds the specified deadband, an impulsive maneuver is planned to reach the desired trajectory.

Messages

The set of input messages that may be sent to the Control Law module is summarized in Table 4-26.

Table 4-26. Input Messages

Message Name	Data	Source Module	Description
resetCL	-	Command Processing	Causes the module to run the Initialization function.
command verify mode on	-	Command Processing	Enables planned maneuvers to be implemented.
command verify mode off	-	Command Processing	Causes planned maneuvers to be sent as telemetry for verification.
command control off	-	Command Processing	Deactivates the control law.
command control on	-	Command Processing	Activates the control law.
command fine control off	-	Command Processing	Disables fine control.
command fine control on	-	Command Processing	Activates fine control.
command preemptive avoidance off	-	Command Processing	Deactivates preemptive collision avoidance.
command preemptive avoidance on	-	Command Processing	Activates preemptive collision avoidance.
set updatePeriodCL	`double`	Command Processing	Sets the update period (in seconds).
set nominalThrust	`double`	Command Processing	Sets the nominal thrust (in kiloNewtons).
set planHorizon	`double`	Command Processing	Sets the time horizon for maneuver planning (in seconds).
set minBurnDuration	`double`	Command Processing	Sets the minimum achievable burn duration (in seconds).
set maxDeltaV	`double`	Command Processing	Sets the maximum achievable delta-v (in km/s).
set eTol	`double`	Command Processing	Sets the eccentricity tolerance (in seconds).
set nSPOCoarse	`int`	Command Processing	Sets the number of samples per orbit for coarse LP maneuver planning.
set nSPOFine	`int`	Command Processing	Sets the number of samples per orbit for fine LP maneuver planning.
set coarseDeadband	`double`	Command Processing	Sets the deadband for coarse control (in meters).

Table 4-26. Input Messages, contd.

Message Name	Data	Source Module	Description
set fineDeadband	`double`	Command Processing	Sets the deadband for fine control (in seconds).
set timeWindow	`window`	Command Processing	Sets the time window data structure for maneuver planning.
set postBurnSettleTime	`double`	Command Processing	Sets the settle-time required after a burn (in seconds).
set goals	`geometry`	Command Processing, Guidance Law	Sets the geometric goals.
clear goals	-	Team Management	Clears the geoemetric goals.
maneuver approved	-	Collision Monitoring	Informs the Control Law that the maneuver was approved.
maneuver denied	-	Collision Monitoring	Informs the Control Law that the maneuver was denied.
reference active	-	Control Law*	Informs the Control Law that the reference is maneuvering.
reference idle	-	Control Law*	Informs the Control Law that the reference is no longer maneuvering.
notify team member added	-	Team Management	Informs the Control Law that a team member has been added, so that it may send a “reference active” message to it if necessary.
pause	-	Team Management	Causes the Control Law to temporarily forego its update function.
resume	-	Team Management	Causes the Control Law to resume its update function.

The set of request messages that may be sent to the Control Law module is summarized in Table 4-27.

Table 4-27. Request Messages

Message Name	Source Module	Description
get goals	-	Return the geometric goals to the sender.

The set of output messages sent from the Control Law module to other modules is summarized in Table 4-28.

Table 4-28. Output Messages

Message Name	Data	Destination Module	Description
transmit	`isl_message`	ISL Management	Transmit the attached message over the ISL to another spacecraft.
set maneuver	`maneuver`	Delta-V Management	Send the planned delta-v sequence to the Delta-V Management module.
cancel maneuver	`int`	Delta-V Management	Attempt to cancel the previously planned maneuver.
get state	-	Coordinate Transformation	Obtain the state information.
get fuelMass	-	Parameter Database	Obtain the remaining fuel mass.
get reference status	-	Team Management	Return a true/false flag indicating whether I am a reference.
get relative ids for reference	`int`	Team Management	Return the IDs of all relatives for the specified reference ID.

Table 4-28. Output Messages, contd.

Message Name	Data	Destination Module	Description
reference active	-	Control Law*	Inform the Control Law that the reference is maneuvering.
reference idle	-	Control Law*	Inform the Control Law that the reference is no longer maneuvering.

Required Functions

The functions required by the Control Law module are listed in Table 4-29.

Table 4-29. Required Functions

Function Name	Description
`IterativeImpulsiveManeuver`	description
`LinOrbLQG`	Generate an LQG controller for linearized relative orbit dynamics.
`Geometry_Structure`	Initialize a circular geometric goals data structure.
`ISLMessage_Structure`	Initialize an ISL message data structure.
`PlanningParameters_Structure`	Initialize an parameters data structure for maneuver planning.
`FFEccProp`	Compute the Hills frame state on another point of the trajectory given the integration constants.
`IdealActuator`	Command the desired force to the IdealActuator. Only used in validating the fine control.
`FSWClock`	Access the flight software clock to obtain the current time.
`FFEccGoals2Hills`	Compute the desired Hills frame state from the geometric goals and reference orbit data.
`GeometryCirc2Ecc`	Convert a cicular goals structure to an eccentric goals structure.
`GeometryEcc2Circ`	Convert an eccentric goals structure to a circular goals structure.
`Goals2Hills`	Computes the desired Hills frame state given the geometric goals and the reference orbital elements.
`JD2SS1970`	Convert Julian date to seconds since 1970.
`Nu2TimeDomain`	Convert a relative state from the nu-domain to the time-domain.
`SS19702JD`	Convert seconds since 1970 to Julian date.
`IsCircGeom`	Determine whether a geometric goals data structure is of the circular type or not.
`IsEccGeom`	Determine whether a geometric goals data structure is of the eccentric type or not.
`C2DZOH`	Perform a zero-order hold to convert continuous state-space matrices to discrete-time.
`MessageQueue`	Display messages to a GUI while the software runs. Used for validation purposes only.
`OrbRate`	Compute the mean orbit rate from the semi-major axis.
`M2Nu`	Compute the true anomaly from the mean anomaly.
`FormatGoals`	Member function. Ensure goals have the proper format for the eccentricity.

Member Variables

The member variables for the Control Law module are listed in Table 4-30.

Table 4-30. Member Variables

Variable Name	Data	Description
`iD`	`int`	Unique spacecraft ID (positive)
`updatePeriod`	`double`	Update period (sec)
`timeLastUpdate`	`double`	Time at which the module updated last (JD)

Table 4-30. Member Variables, contd.

Variable Name	Data	Description
cancellationLeadTime	double	Amount of lead-time required to cancel a maneuver before the first burn is applied (sec)
run	int	Flag indicating whether to run the update function or not (changed with pause/resume messages)
haveGoals	int	Flag indicating whether geometric goals have been supplied
newGoals	int	Flag indicating whether new geometric goals have just been received
maneuverPlanned	int	Flag indicating whether a maneuver has been planned
maneuverCompleted	int	Flag indicating whether the maneuver has been completed
referenceIdle	int	Flag indicating whether the reference is idle or active
controlEnabled	int	Flag indicating whether control is enabled
fineControlEnabled	int	Flag indicating whether fine control is enabled
cancelManeuver	int	Flag indicating whether a "cancel maneuver" command has been received
verify	int	Flag indicating whether the module is in verification mode
goals	geometry	Geometric goals data structure
xK	matrix[6,1]	Controller state vector for fine control
coarseDeadband	double	Size of the position error deadband for coarse control (m)
fineDeadband	double	Size of the position error deadband for fine control (m)
window	window	Time window data structure to use for maneuver planning
settleTime	double	Minimum time required after a thruster firing before the maneuver is considered complete (sec)
parameters	plan_param	Planning parameters data structure
dryMass	double	Spacecraft dry mass (kg)
fuelMass	double	Total mass of fuel in all tanks (kg)
maneuver	maneuver	Maneuver data structure

Functionality

The Control Law module has one primary purpose – to keep the spacecraft on its desired relative trajectory. The functionality of the Control Law is designed to serve this purpose.

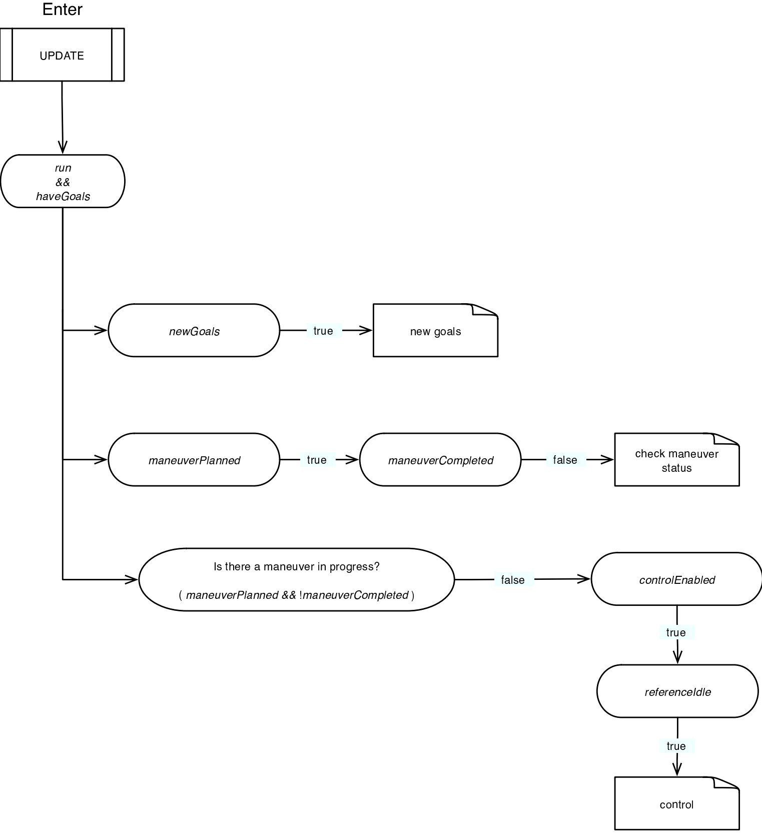
The flowchart for the `Update` function is shown in Figure 4-6 on the next page. See Figure 4-1 on page 20 for a legend of the various flowchart components. This diagram shows the logical steps that take place each update period to cause the module to enter the various functional blocks. The steps taken in each of these functional blocks are summarized below, and are also illustrated in subsequent flowcharts in the appendix.

When the Control Law updates, it first checks whether new goals have been received. If they have, the current state is obtained from the Coordinate Transformation module, so that the eccentricity may be checked. If the eccentricity exceeds the prescribed tolerance for circular orbits, then the geometric goals are defined using eccentric framework. Otherwise, they are defined using the circular framework. The format of the goals is updated automatically using the `FormatGoals` function.

The module then checks internal flags to determine whether a maneuver is in progress. If one is in progress, it then checks whether it has been instructed to cancel a maneuver. If a cancellation request has been made, and the first burn of the maneuver has not yet begun, then a burn cancellation command is issued to the Delta-V Manager. The maneuver ID is sent as the attached data to this message. Next, the module compares the current time with the maneuver end time (`maneuver.tF`) to determine if the current maneuver is now complete. If this condition is true, and if the local spacecraft is a reference for another team, it sends a "reference idle" message to the relatives over the ISL.

The next stage of the update function is for maneuver planning. If control is enabled (`controlEnabled` true), there is no maneuver in progress (`maneuverInProgress` false), and the reference spacecraft is idle (`referenceIdle`),

Figure 4-6. Control Law Update



then it proceeds to the next step. The state estimate is obtained from the Coordinate Transform module, and the fuel mass is obtained from the Parameter Database. The desired relative state is computed from the geometric goals, and then compared with the current relative state to compute the position error in Hill's frame. If any axis of the position error exceeds the prescribed deadband for that axis, then a maneuver is planned. The `IterativeImpulsiveManeuver` function is used to plan a time-weighted fuel-optimal maneuver such that the largest delta-v in the sequence does not exceed the prescribed maximum. This function is a high-level wrapper for both circular and eccentric-based optimal maneuver planning functions. If a solution is found, and if the maneuver is to be achieved (`verify` false), it is then sent to the Delta-V Management module in a "set maneuver" message.

If the Control Law has been instructed to observe *preemptive collision avoidance*, it first sends the maneuver data to the local Collision Monitoring module in a "inform local maneuver" message. If the maneuver is approved for safety, the Control Law will later receive a "maneuver approved" message, at which time it will then send the "set maneuver" message to the Delta-V Management module to be processed.

4.8 Collision Monitoring

`DFFCollisionMonitoring`

Scope

The Collision Monitoring module is tasked with both the monitoring of failed spacecraft for potential collisions and the determination of whether reconfiguration trajectories are collision safe. This is implemented as two stages of functionality within a single module's update function.

Messages

The set of input messages that may be sent to the Collision Monitoring module is summarized in Table 4-31.

Table 4-31. Input Messages

Message Name	Data	Source Module	Description
command start monitoring	`int`	Command Processing	Start monitoring the specified spacecraft.
command stop monitoring	`int`	Command Processing	Remove the specified spacecraft from the monitoring list.
reconfig IDs	`int array`	Guidance Law	The IDs of the spacecraft maneuvering during the reconfiguration.
inform local maneuver	`maneuver`	Control Law	The maneuver of the local spacecraft.
reconfig denied	`int`	Collision Monitor	Another Collision Monitor has denied the reconfiguration, stop processing the maneuvers.
reconfig approved	`int`	Collision Monitor	Another Collision Monitor has approved the reconfiguration, add it to the approved list.
distribute maneuver	`maneuver`	Collision Monitor	The maneuver of a reconfiguring spacecraft.

The set of output messages sent from the Collision Monitoring module to other modules is summarized in Table 4-32.

Table 4-32. Output Messages

Message Name	Data	Destination Module	Description
distribute maneuver	`maneuver`	Collision Monitor	Transmit the local maneuver to the other reconfiguring spacecraft.
reconfig denied	`double`	Collision Monitor	Inform the other spacecraft that a collision was detected and the reconfiguration is denied.
reconfig approved	`int`	Collision Monitor	Inform the other spacecraft that there are no potential collisions with this spacecraft and the reconfiguration is approved.
maneuver approved	`int`	Control Law	Inform the Control Law that the maneuver has been approved.
maneuver denied	`int`	Control Law	Inform the Control Law that the maneuver has been denied.
get state	-	Coordinate Transformation	Obtain the state information.
get relative state wrt self	-	Coordinate Transformation	Obtain the relative's state with respect to the local spacecraft.
get member ids for member	`int`	Team Management	Return the IDs of all members for the specified member ID.

Required Functions

The functions required by the Collision Monitoring module are listed in Table 4-33.

Table 4-33. Required Functions

Function Name	Description
MessageQueue	Display messages to a GUI while the software runs. Used for validation purposes only.
PredictCollision	Runs the collision monitoring algorithm by propagating forward for a fixed time assuming no maneuvers.
CollisionSurvey	Runs the collision monitoring algorithm for n spacecraft relative to the self spacecraft when the spacecraft are maneuvering.
CollisionMonAlg	Collision monitoring algorithm which discretizes the orbit according to input time vector and handles maneuvers through an acceleration matrix.
CollProbSet	Calculates the probability of collision given a relative uncertainty ellipsoid.
DistantPtToEll	Finds the distance (and the corresponding point) from a distant point to the closest point on an ellipsoid.
Laguerre	Finds polynomial roots using Laguerre's method.
GVEErrorDynamics	Compute continuous-time dynamics for Gauss' variational equations
C2DZOH	Create a discrete time system from a continuous system assuming a zero-order-hold at the input.
WrapPhase	Wrap a phase angle (or vector of angles) to keep it between -pi and +pi
UnwrapPhase	Unwrap a vector of angular values to change continuously.
FFEccLinOrb	Compute the continous A,B matrices for linearized relative motion in an eccentric reference orbit.
Nu2M	Converts true anomaly to mean anomaly.
M2Nu	Computes the true anomaly from the mean anomaly.
M2NuAbs	Computes the true anomaly from the mean anomaly without wrapping btwn -pi / pi
OrbRate	Computes the orbital rate.
DeltaEl2Alfriend	Compute Alfriend differential elements from standard differential elements.
El2Alfriend	Convert the standard orbital element set into an Alfriend orbital element set
DeltaElem2Hills	Computes the Hills frame state from orbital element differences and reference orbital elements.
Mag	Compute the magnitude of a vector or matrix.
GenerateTimeVector	Generate a time vector evenly spaced over true anomaly
VerifyCollStruct	Ensure consistency in collision data structure fields.
ManeuverStruct2AccelVector	Compute a 3xN acceleration vector from a "maneuver" data structure.

Member Variables

The member variables for the Collision Monitoring module are listed in Table 4-34.

Table 4-34. Member Variables

Variable Name	Data	Description
iD	int	Unique spacecraft ID (positive)
updatePeriod	double	Update period (sec)
timeLastUpdate	double	Time at which the module updated last (JD)
monitorIDs	int	List of spacecraft IDs for continuous collision monitoring
reconfigIDs	int	List of spacecraft which are maneuvering in the current reconfiguration
checkedIDs	int	List of reconfigIDs which have already been checked

Table 4-34. Member Variables, contd.

Variable Name	Data	Description
`listApproved`	`int`	List of reconfigIDs which have reported approval of the reconfiguration
`processManeuvers`	`int`	Flag indicating that a local reconfiguration survey is in progress
`mvrSent`	`int`	Flag if local maneuver has been sent to other reconfiguring spacecraft
`allChecked`	`int`	Flag indicating that all reconfigIDs have been checked by the local Collision Monitor
`collDetected`	`int`	Flag indicating that a collision has been detected
`approved`	`int`	Flag indicating that reconfiguration has been approved locally and informing messages have been transmitted
`warningLevel`	`double`	Probability level above which a collision is considered to be detected
`lenSC`	`int`	Spacecraft longest dimension (meters)
`initBounds`	`int`	Typical one-sigma bounds on relative measurement (km and km/s)
`scalev`	`int`	Desired scale factor for ellipsoids, in sigma
`predictTime`	`int`	Prediction time period for continuous monitoring (sec)
`nSamples`	`int`	Number of samples per orbit for continuous monitoring
`Pmin`	`int`	Minimum probability corresponding to `scalev`, calculated
`Ssc`	`int`	Spacecraft hard-body bounding sphere corresponding to `lenSC`, calculated
`S0`	`int`	Initial ellipsoid corresponding to `scalev` and `initBounds`, calculated

Functionality

As discussed in the Scope the Collision Monitor module has two distinct modes of functionality. These are performed sequentially in the Update() function and each maintains its own list of monitored IDs.

The first mode, termed simply *collsion monitoring* involves invoking continuous monitoring in the case of a spacecraft failure in the cluster. Each functioning spacecraft is responsible for monitoring collisions between itself and any failed spacecraft in true decentralized fashion. Commands to start monitoring a spacecraft are received from the ground. The collision monitoring algorithm is then called onboard at each update period for each failed spacraft ID in the list. It scans some amount of time into the future, such as one or two orbits, and determined the maximum probability of a collision during that period. The prototype nominally assumes that no maneuvers are taking place during this period.

The second mode, termed *preemptive avoidance* or equivalently called a *collision survey*, is enacted on a team-basis during a reconfiguration. The reconfiguration IDs are shared with the Collision Monitor by the Guidance Law once a formation has been commanded to be achieved. Once each spacecraft has determined its delta-v sequence, they are communicated throughout the team and each maneuvering spacecraft will check for possible collisions between itself and the rest of the team. Non-maneuvering spacecraft do not perform any computations during the survey. Since this is performed on a discrete basis, only for reconfigurations, it is similar to a single function call to the collision monitoring algorithm, or n function calls for n satellite pairs when you consider the whole team.

This mode has the following steps:

- Receive reconfig IDs and begin waiting for maneuver data from self and others
- If self is maneuvering, survey self and all non-maneuvering spacecraft
- Check each maneuvering spacecraft as data arrives
- Transmit 'reconfig denied' and 'maneuver denied' messages for detected collisions

- Once all spacecraft are surveyed send 'reconfig approved'
- Once all maneuvering spacecraft report approved, send 'maneuver approved'

These steps require a number of logic flags as seen in the member variables list.

4.9 ISL Management

`DFFISLManagement`

Scope

The purpose of the ISL Management module is to facilitate inter-spacecraft communication via the inter-spacecraft link (ISL). Messages that must be sent from one spacecraft to another are sent to this module, transmitted over the ISL, retrieved by this module on the other spacecraft, and finally sent to the destination module.

Messages

The set of input messages that may be sent to the ISL Management module is summarized in Table 4-35.

Table 4-35. Input Messages

Message Name	Data	Source Module	Description
resetIM	-	Command Processing	Causes the module to run the Initialization function.
set updatePeriodIM	`double`	Command Processing	Sets the update period (in seconds).
set maxISLAttempts	`int`	Command Processing	Sets the limit for consecutive attempts to send a message over the ISL.
transmit	`isl_message`	Team Management, Guiance Law, Control Law	Provides a message data structure (from another module) to be sent over the ISL.
receive	`isl_message`	ISL Interface Plugin	Provides a message data structure (from the ISL subsystem) to be converted to a message and sent to another module.

There are no request messages sent to the ISL Management module.

The set of output messages sent from the ISL Management module includes all of the messages that are sent from other spacecraft via the ISL. In each case, a "receive" message is first obtained from the ISL Interface Plugin. The "receive" message contains an `isl_message` data structure, which is unpacked and sent to the destination module on the local spacecraft.

Required Functions

The functions required by the ISL Management module are listed in Table 4-36.

Table 4-36. Required Functions

Function Name	Description
`isfield`	ISFIELD(S,'name') returns 1 if 'name' is a field in the structure array S and 0 otherwise.
`intersect`	INTERSECT(A,B) when A and B are vectors returns the values common to both A and B.
`DataSize`	Compute the size (in bytes) of a piece of data. Used to determine the amount of data to be sent over the ISL.
`ISLMessage_Structure`	Initialize an ISL message data structure.
`FSWClock`	Access the flight software clock to obtain the current time.
`StateSensor`	Access the true simulation state. Used in "TransmitOverISL" function to determine capability of transmitting the data to the destination spacecraft.

Table 4-36. Required Functions, contd.

Function Name	Description
JD2SS1970	Convert Julian date to seconds since 1970.
QTForm	Transforms a vector opposite the direction of the quaternion.
MessageQueue	Display messages to a GUI while the software runs. Used for validation purposes only.
Dot	Compute the dot product of two vectors.
Mag	Compute the magnitude of a vector or matrix.
RouteMessage	Member function. Attempt to route a message through a different spacecraft if the destination cannot be reached.
TransmitOverISL	Member function. Model the transmission of data over the ISL.
FormatMessage	Member function. Ensure the supplied message structure has the proper and complete format.

Member Variables

The member variables for the ISL Management module are listed in Table 4-37.

Table 4-37. Member Variables

Variable Name	Data	Description
iD	int	Unique spacecraft ID (positive)
updatePeriod	double	Update period (sec)
timeLastUpdate	double	Time at which the module updated last (JD)
memoryOn	int	Flag indicating whether to record messages sent over the ISL. This is used for software testing and validation purposes only.
maxISLAttempts	int	Maximum number of times to attempt transmitting to one spacecraft before routing it through another spacecraft instead
counter	int	Counter for generating unique ID tags for messages
counterLimit	int	Limit for message counter before resetting
bandwidth	double	Bandwidth capability of the ISL (kb/sec)
antennae	struct	Data structure of antennae information
messageQueueIn	isl_message[]	Array of ISL message structures received from the ISL, to be sent to local modules.
messageQueueOut	isl_message[]	Array of ISL message structures received from local modules, to be sent over ISL.

Functionality

The ISL Management module serves as an interface between the other DFF software module and the ISL. Messages that must be sent to another spacecraft attached as data to a "transmit" message and sent to this module. The message data is then transmitted over the ISL. The act of transmission is simulated with a simple built-in model. When an inter-spacecraft message is received at the ISL, the message is attached as data and sent in a "receive" message to the ISL Management module. The message data is then unpacked and sent to the specified destination module.

The same approach will be taken in the MANTA-based DFF system. Figure 4-7 on the following page illustrates the steps of the procedure. First, some module on spacecraft A prepares a message to be sent to spacecraft B. Next, it converts the message to binary and attaches it to a new message called "transmit over isl", which is sent to the ISL Management module. The binary data is then submitted to the ISL hardware, and is transmitted to the destinaion spacecraft. At this point, the data is retrieved from the ISL hardware on the destination spacecraft by the ISL Management module. Finally, the binary data is converted back into a MANTA message and is sent to the appropriate module.

Figure 4-7. Proposed ISL Management Approach in MANTA

Spacecraft A

Module X

transmit

ISL Management

ISL

Spacecraft B

ISL

receive

ISL Management

On Spacecraft A:

1. "Module X" prepares a MANTA message destined for Spacecraft B and convert it to binary data.
2. The data is sent in a **"transmit"** message to the ISL Management module.
3. The data is sent through the ISL Interface Plugin to the ISL hardware
4. The data is transmitted over the ISL to Spacecraft B

On Spacecraft B:

5. The ISL Interface Plugin retrieves the data from the ISL hardware and sends it in a **"receive"** message to the ISL Management module
6. The ISL Management module converts the binary data to a message ands sends it out

There is one significant difference between the prototype version of this module and how the MANTA version will be designed. In the prototype, the ISL transmission model is built-in to the module. This is done for the sake of speed and simplicity. In the real DFF system, the software will be completely stand-alone, and therefore the ISL transmission model must be implemented externally (ie, in DSim).

This module also provides an extra layer of communication robustness by attempting to transmit a message multiple times if the first attempt is not successful. This functionality requires that the ISL hardware be capable of immediately informing the module whether the transmission attempt was sucessful or not. If the maximum number of attempts are reached (defined by the uploadable parameter, `maxAttempts`), it then attempts to route the data through another spacecraft. This can help if the destination spacecraft is either out of range or out of view. The module attempts to route data only through the local spacecraft's team members. The current approach is simply to cycle through all team members. It may prove more efficient to identify one or more spacecraft that are likely to be in range and in the field of view of both the sender and the receiver. This would require maintaining a local copy of the relative state of all team members.

In the prototype design, each "transmit" message that is sent to the ISL Module contains a `message` data structure, which is defined by the `ISLMessage_Structure` function. Upon the receipt of a "transmit" message, the data is stored in a `message` struct array called `messageQueueOut`. When the module updates, the messages in this queue are sent on a FIFO basis via the "TransmitOverISL" function. The amount of data being transmitted each time is monitored, so that, when the maximum bandwidth is reached, transmissions cease until the next update period.

When data is received at the local ISL, it is placed in a "receive" message and sent to the ISL Management module. Upon the receipt of a "receive" message, the data is stored in a second `message` struct array called `messageQueueIn`. When the module updates, the messages in this queue are processed on a FIFO basis. First, the destination ID is checked. If it matches this spacecraft's ID, the message is sent to the appropriate module. Otherwise, the message is

being routed, so it is added to the `messageQueueOut` array to be sent during the next update.

4.10 Delta-V Management

`DFFDeltaVManagement`

Scope

The purpose of the Delta-V Management module is to process the impulsive delta-v sequences planned by the Control Law and Collision Monitoring modules. It prepares and sends commands to fire the thruster(s) at the appropriate times to achieve the desired delta-v. If necessary, it also computes the required attitude that the spacecraft must have for each thruster firing, and notifies the Atttitude Management module. This is necessary in spacecraft configurations that do not have omin-directional thrust capability, so that the entire spacecraft must slew to a particular attitude to orient the thruster(s) in the proper direction for a burn.

Messages

The set of input messages that may be sent to the Delta-V Management module is summarized in Table 4-38.

Table 4-38. Input Messages

Message Name	Data	Source Module	Description
resetDM	-	Command Processing	Causes the module to run the Initialization function.
command alignment verification on	-	Command Processing	Enables the alignment verification function.
command alignment verification off	-	Command Processing	Disables the alignment verification function.
command star tracker constraint on	-	Command Processing	Commands the star-tracker pointing constraint to be checked.
command star tracker constraint off	-	Command Processing	Commands the star-tracker pointing constraint to be disregarded.
set updatePeriodDM	`double`	Command Processing	Sets the update period (in seconds).
set alignmentTolerance	`double`	Command Processing	Sets the alignment tolerance (in radians).
set maxSlewTime	`double`	Command Processing	Sets the approximate maximum slew time (in seconds).
set maneuver	`maneuver`	Control Law	Sets the maneuver data (a time-tagged burn sequence) that was planned by the Control Law.
cancel maneuver	`int`	Control Law	Attempt to cancel the previously planned maneuver.

There are no request messages sent to the Delta-V Management module.

The set of output messages sent from the Delta-V Management module to other modules is summarized in Table 4-39.

Table 4-39. Output Messages

Message Name	Data	Destination Module	Description
get state	-	Coordinate Transformation	Obtain the state information.
get fuelMass	-	Parameter Database	Obtain the remaining fuel mass.
get thrusterStatus	`int`	Parameter Database	Obtain the status of the specified thruster.
get tankPressure	`int`	Parameter Database	Obtain the pressure of the specified tank.
get reci	-	Relative Navigation	Obtain the ECI position vector.
get veci	-	Relative Navigation	Obtain the ECI velocity vector.

Table 4-39. Output Messages, contd.

Message Name	Data	Destination Module	Description
add target orientation	orientation	Attitude Management	Add the orientation data structure to the queue.
cancel slews	int	Attitude Management	Cancel the previously commanded slews.
get qecitobody	-	ADCS Interface Plugin	Obtain the ECI-to-body quaternion.
set hills force	matrix[3,1]	Ideal Actuator	Set the force to be applied in Hills-frame (kN).
set hills force time window	matrix[2,1]	Ideal Actuator	Set the time window during which the force is to be applied (kN).
set pulsewidth	double	Thruster Interface Plugin	Obtain the ECI-to-body quaternion.

Required Functions

The functions required by the Delta-V Management module are listed in Table 4-40.

Table 4-40. Required Functions

Function Name	Description
DeltaVCommand_Structure	Initialize a delta-v command data structure.
Orientation_Structure	Initialize an orientation data structure.
IdealActuator	Command the desired force to the IdealActuator. Only used in validating the software.
FSWClock	Access the flight software clock to obtain the current time.
SS19702JD	Convert seconds since 1970 to Julian date.
QForm	Transforms a vector in the direction of the quaternion.
QHills	Generate the quaternion that transforms from the ECI to the Hills frame.
QMult	Multiply two quaternions.
QPose	Transpose a quaternion.
MessageQueue	Display messages to a GUI while the software runs. Used for validation purposes only.
Dot	Compute the dot product of two vectors.
ThrusterAlignment	Computes body vectors to align with velocity and nadir for a thruster firing.
RVOrbGen	Generate keplerian elements from an initial ECI position and velocity.

Member Variables

The member variables for the Delta-V Management module are listed in Table 4-41.

Table 4-41. Member Variables

Variable Name	Data	Description
iD	int	Unique spacecraft ID (positive)
updatePeriod	double	Update period (sec)
timeLastUpdate	double	Time at which the module updated last (JD)
preBurnVerification	int	Flag indicating whether alignment verification is enabled
starTrackerConstraint	int	Flag indicating whether the star-track pointing constraint should be applied
alignmentTolerance	double	Allowable angular error in alignment for a thruster-firing (rad)
uStarTrackerBody	matrix[3,1]	Boresight vector of star-tracker in body frame

Table 4-41. Member Variables, contd.

Variable Name	Data	Description
minAngularSep	double	Minimum allowable angular separation between star-tracker boresight and bright bodies (rad)
maxSlewTime	double	Approximate time it takes the spacecraft to slew 180 deg (sec)
dryMass	double	Spacecraft dry mass (kg)
nTanks	int	Number of fuel tanks
tank	struct[]	Data structure array of fuel tank information
commandInQueue	burn[]	Array of burn commands (from the maneuver data structure) to be processed
commandOutQueue	dv_command[]	Array of thruster commands to be sent out
ideal	int	Flag indicating whether ideal actuation is to be used. This is used for software validation purposes only.
idealForce	matrix[3,1]	Force to be applied in the case of ideal actuation (N). This is used for software validation purposes only.
leadTime	double	Amount of lead-time to use in commanding the thrusters (sec)

Functionality

The functionality of the Delta-V Management module is straightforward. Each time a "set maneuver" message is received from the Control Law, the included burn data is stored in an array, `commandInQueue`. Each time the module updates, it checks this queue. If it is not empty, it processes each entry of the queue, converting the data into a `dv_command` structure, which is described in Section A.14 on page 71. It contains the time at which to command the burn, and the required burn duration (pulsewidth). Each structure is stored in a new array, `commandOutQueue`. For each burn, an `orientation` structure is also computed and sent to the Attitude Management module in a "add target orientation" message.

Later in the update function, the module checks the `commandOutQueue`. The queue is processed in a FIFO manner. If it is not empty, the first item in the queue is extracted. First, the current time is checked against the specified command time. A user-defined lead time is also included to account for hardware delays. If the timing condition is satisfied, it then checks the `preBurnVerification` flag, which is a switch that may be toggled by the user. If pre-burn verification is true, the module first checks the measured attitude with the desired attitude, and ensures that it is within the prescribed tolerance before commanding the burn. If this verification is not activated, or if it is and the pointing error is small enough, then the burn is commanded by sending the pulsewidth to the thruster model.

4.11 Attitude Management

`DFFAttitudeManagement`

Scope

The purpose of the Attitude Management module is to command the ADCS software to achieve the specified orientation at the specified time. This functionality is required for spacecraft configurations that do not have omni-directional thrust capability. In these cases, the entire spacecraft must be slewed to a particular attitude so that the thruster is aligned in the proper direction for a burn.

Messages

The set of input messages that may be sent to the Attitude Management module is summarized in Table 4-42.

Table 4-42. Input Messages

Message Name	Data	Source Module	Description
resetAM	-	Command Processing	Causes the module to run the Initialization function.
command coordination on	-	Command Processing	Enables the distributed attitude coordination function.
command coordination off	-	Command Processing	Disables the distributed attitude coordination function.
set updatePeriodAM	`double`	Command Processing	Sets the update period (in seconds).
set maxSlewTime	`double`	Command Processing	Sets the approximate maximum slew time (in seconds).
add target orientation	`orientation`	Delta-V Management	Causes the supplied orientation data structure to be added to the queue.
cancel slews	`int`	Delta-V Management	Cancels the previously commanded slews.

There are no request messages sent to the Attitude Management module.

All output messages of the Attitude Management module are sent to the ADCS Interface Plugin.

Required Functions

The functions required by the Attitude Management module are listed in Table 4-43.

Table 4-43. Required Functions

Function Name	Description
`ACSPointingConversion`	Computes a body vector and an LVLH vector (to be aligned).
`FSWClock`	Access the flight software clock to obtain the current time.
`MessageQueue`	Display messages to a GUI while the software runs. Used for validation purposes only.

Member Variables

The member variables for the Attitude Management module are listed in Table 4-44 on the following page.

Table 4-44. Member Variables

Variable Name	Data	Description
iD	int	Unique spacecraft ID (positive)
updatePeriod	double	Update period
timeLastUpdate	double	Time at which the module updated last (JD)
solarArrayNormal	matrix[3,1]	Normal vector of the solar arrays in the body frame
maxSlewTime	double	Approximate time it takes the spacecraft to slew 180 deg (sec)
coordination	int	Flag indicating whether attitude coordination is enabled
deltaVSlewInProgress	int	Flag indicating whether a slew is in progress
returnToNominalJD	double	Time at which to return to the nominal pointing mode (JD)
slewCommandQueue	orientation[]	Queue of slew commands to be processed

Functionality

The functionality of the Attitude Management module is straightforward. Each time an “add target orientation” message is received, the associated data is stored at the end of a queue. The attached data is an `orientation` data structure, which is described in Section A.15 on page 71. It includes the desired Hills-to-body quaternion, the time at which this orientation should be achieved, and the length of time that it should be maintained. When the module updates, the queue is checked. If it is not empty, the first item in the queue is extracted. If this item corresponds to a new slew command that has not already begun, then the time is checked. The current time is checked against the specified orientation time. A user-defined lead time is also included to account for time required to slew. If the timing condition is satisfied, then the slew command is issued.

Two types of spacecraft configurations are supported: spin-stabilized and 3-axis stabilized. The software is initialized with a flag that indicates the spacecraft configuration. For 3-axis stabilized spacecraft, the desired ECI-to-body quaternion (`qEBDes`) is commanded to the ADCS IP. This quaternion is computed anew each update period, because it changes slowly as the spacecraft traverses its orbit. It is found by performing multiplying the measured ECI-to-Hills quaternion by the desired Hills-to-body quaternion. This is a quaternion multiplication operation. The measured ECI-to-Hills quaternion is computed directly from the reference orbit ECI position and velocity. A code fragment is provided below:

Listing 4.4. `DFFAttitudeManagement.m` *Computing the Target ECI-To-Body Quaternion*

```
1       % compute a new target quaternion each update
2       %------------------------------------------
3       state     = DFFCoordinateTransformation('get_state',[],d.iD);
4       [r,v]     = El2RV( state.el );
5       qEH       = QHills(r,v);             % current ECI to Hills quaternion
6       qEBDes    = QMult( qEH, slewCommand.qHB );
```

Computing the Target ECI-To-Body Quaternion

For the spin-stabilized spacecraft, the desired right ascension and declination are commanded to the ADCS IP. These are computed directly from `qEBDes` using the *Q2RADec* function.

When a slew command is first issued, the module computes sets the `deltaVSlewInProgress` flag to true, and then records the time at which the slew command should end. It stores this time in the variable `returnToNominalJD`. At a later point in the update function, the module checks the `deltaVSlewInProgress` flag to determine if a delta-v slew is currently underway. If true, it compares the current time to the `returnToNominalJD` time to determine when the delta-v slew ends. Upon the completion, it commands the ADCS to return to sun-pointing mode and resets the `deltaVSlewInProgress` flag to false.

CHAPTER A. Data Structures

This section describes all of the data structures used in the DFF system.

A.1 Command Data

The `command` data structure is initialized with the `Command_Structure` function. It is used in conjunction with the Command Processing module. A command list, which contains an array of `command` structures, is supplied to the Command Processing module at the beginning of the simulation. The module then processes a command once the mission-elapsed-time (MET) surpasses the command's time-tag.

Table 1-1. Command Data Structure

Field Name	Data	Description
`timeTag`	`double`	Time at which the command is to be processed (MET)
`scID`	`int`	Unique spacecraft ID
`module`	`char`[]	Name of the destination module
`command`	`char`[]	Name of the command message
`data`	*var*	Included data

A.2 Team Data

The `team` data structure is initialized with the `Team_Structure` function. It contains all of the data that completely defines a team, including the team ID, the member IDs, the reference and captain IDs, and the level.

Table 1-2. Team Data Structure

Field Name	Data	Description
`teamID`	`int`	Unique team ID
`nMembers`	`int`	Number of members in the team
`memID`	`int`[nMembers]	Array of member IDs
`refID`	`int`	ID of member serving as the reference
`captainID`	`int`	ID of member serving as the captain

Table 1-2. Team Data Structure, contd.

FieldName	Data	Description
`level`	`int`	Hierarchical level of the team (low number means high in the hierarchy)

A.3 State Data

The `state` data structure is initialized with the `State_Structure` function. It contains a few different sets of absolute and relative state information.

Table 1-3. State Data Structure

Field Name	Data	Description
`el`	`matrix`[1,6]	Standard set of orbital elements
`elA`	`matrix`[1,6]	Alfriend set of orbital elements
`dEl`	`matrix`[1,6]	Orbital element differences in Alfriend format
`xH`	`matrix`[6,1]	Relative position and velocity in Hills frame
`tM`	`double`	Measurement time (SS1970)

A.4 Geometry Data

The `geometry` data structure is initialized with the `Geometry_Structure` function. It contains the set of parameters that define the geometric goals for a repeating relative trajectory in a circular orbit.

Table 1-4. Geometry Data Structure

Field Name	Data	Description
`y0`	`double`	Along-track offset of center of relative motion (km)
`aE`	`double`	Semi-major axis of relative ellipse (km)
`beta`	`double`	Phase angle on relative ellipse at equator crossing, measured from -x axis to +y axis (rad)
`zInc`	`double`	Maximum cross-track amplitude due to inclination difference (km)
`zLan`	`double`	Maximum cross-track amplitude due to right ascension difference (km)

A.5 Eccentric Geometry Data

The `ecc_geometry` data structure is initialized with the `EccGeometry_Structure` function. It contains the set of parameters that define the geometric goals for a repeating relative trajectory in an eccentric orbit.

Table 1-5. Eccentric Geometry Data Structure

Field Name	Data	Description
`y0`	`double`	Along-track offset of center of relative motion (km)
`xMax`	`double`	Maximum radial separation (km)
`nu_xMax`	`double`	True anomaly where maximum radial separation occurs (rad)
`zMax`	`double`	Maximum cross-track separation (km)
`nu_zMax`	`double`	True anomaly where maximum cross-track separation occurs (rad)

A.6 Window Data

The window data structure is initialized with the Window_Structure function. It defines the details of the time window to be used in maneuver planning.

Table 1-6. Window Data Structure

Field Name	Data	Description
startTime	double	Earliest allowable start time (SS1970)
nOrbMin	int	Minimum number of orbits that the maneuver may last
nOrbMax	int	Maximum number of orbits that the maneuver may last
nManeuvers	int	Total number of maneuvers to plan, between min and max durations
timeWeightExp	double	Exponent for weighting the cost with maneuver time

A.7 Planning Parameters Data

The plan_param data structure is initialized with the PlanningParameters_Structure function. It defines the set of adjustable parameters that are used in the maneuver planning process.

Table 1-7. Planning Parameters Data Structure

Field Name	Data	Description
fNom	double	Nominal thrust capability (kN)
horizon	double	Minimum time required between planning and first burn (sec)
dTMin	double	Minimum achievable burn time (sec)
maxDeltaV	double	Maximum achievable delta-v for a single burn (km/s)
nSPOCoarse	int	Number of samples per orbit to use in coarse LP planning
nSPOFine	int	Number of samples per orbit to use in fine LP planning
eTol	double	Eccentricity tolerance

A.8 Team Goals Data

The team_goals data structure is initialized with the TeamGoals_Structure function. It contains the information that defines the team goals for a circular orbit.

Table 1-8. Team Goals Data Structure

Field Name	Data	Description
nU	int	Number of unique states
teamID	int	Team ID
geometry	geometry[nU]	Array of circular geometry structures
constraints	constraints[nU	Array of constraints structures
dPhi	double	Angular resolution (rad)

A.9 Eccentric Team Goals Data

The ecc_team_goals data structure is initialized with the EccTeamGoals_Structure function. It contains the information that defines the team goals for an eccentric orbit. It is similar to the team_goals structure, but it contains eccentric geometric goals rather than circular geometric goals.

Table 1-9. Eccentric Team Goals Data Structure

Field Name	Data	Description
nU	int	Number of unique states
teamID	int	Team ID
geometry	ecc_geometry[nU	Array of eccentric geometry structures
constraints	constraints[nU	Array of constraints structures
dPhi	double	Angular resolution (rad)

A.10 Cost Estimate Data

The cost data structure is initialized with the CostEstimate_Structure function. It contains the estimated cost to achieve a specified set of target states. This structure is used by the Guidance Law during the distributed assignment process.

Table 1-10. Cost Estimate Data Structure

Field Name	Data	Description
nU	int	Number of unique target states that costs were computed for
memID	int	Unique spacecraft ID
targetIndex	int[nU]	Array of indeces corresponding to unique target states
costLength	int[nU]	Length of the cost vector for each unique target state
cost	matrix[1,nU]	Cost vectors for all unique target states

A.11 Constraints Data

The constraints data structure is initialized with the Constraints_Structure function. These structures are contained within the team_goals and ecc_team_goals data structure. Each constraints structure corresponds to an associated geometry structure. The purpose of this structure is to further define the associated geometric goals, and to place restrictions on how those goals are assigned.

Table 1-11. Constraints Data Structure

Field Name	Data	Description
variable	int	Flag indicating whether the associated geomtric goals are fixed (0) or variable (1)
nRestrict	int	Number of members to restrict assignments to
restrictID	int[nRestrict]	Array of member IDs to restrict assignments to
nDuplicates	int	Number of duplicate states
phase	double[nDuplic	Phase offset of each duplicated state (rad)

A.12 Burn Data

The `burn` data structure is initialized with the `BurnData_Structure` function. It contains the information needed to define the time, direction and magnitude of a "burn", or thruster-firing. An array of `burn` structures is included in the `maneuver` data structure.

Table 1-12. Burn Data Structure

Field Name	Data	Description
`t`	`double`	Burn start time (SS1970)
`dT`	`double`	Burn duration (sec)
`dV`	`double`	Delta-v magnitude (km/s)
`uX`	`double`	Unit x-direction in Hills frame
`uY`	`double`	Unit y-direction in Hills frame
`uZ`	`double`	Unit z-direction in Hills frame
`iD`	`int`	Burn ID (matches `maneuver` ID)

A.13 Maneuver Data

The `maneuver` data structure is initialized with the `Maneuver_Structure` function. It

Table 1-13. Maneuver Data Structure

Field Name	Data	Description
`achieve`	`int`	Flag indicating whether to achieve the maneuver or not
`nBurns`	`int`	Number of impulsive burns in the maneuver
`burnData`	`burn[nBurns]`	Array of `burn` data structures
`t0`	`double`	Maneuver start time (SS1970)
`tF`	`int`	Maneuver completion time (SS1970)
`iD`	`int`	Unique maneuver command ID

A.14 Delta-V Command Data

The `dv_command` data structure is initialized with the `DeltaVCommand_Structure` function. It is used in the Delta-V Management module to prepare and send thrust commands to the propulsion system.

Table 1-14. Delta-V Command Data Structure

Field Name	Data	Description
`tank`	`int`	Index of tank to be used
`pulsewidth`	`double`	Burn duration (sec)
`jD`	`double`	Julian date at which to send the command
`iD`	`int`	Delta-V command ID (matches the `maneuver` ID)

A.15 Orientation Data

The `orientation` data structure is initialized with the `Orientation_Structure` function. It contains the data required to define a desired orientation for the spacecraft during thruster firings. These structures are generated in the

Delta-V Management module and sent to the Attitude Management module.

Table 1-15. Orientation Data Structure

Field Name	Data	Description
qHB	matrix[4,1]	Desired Hills-to-body frame quaternion for aligning the thruster
jD	double	Julian date at which to send the command
dT	double	Duration to remain at target orientation
iD	int	Orientation command ID (matches the maneuver ID)

A.16 ISL Message Data

The isl_message data structure is initialized with the ISLMessage_Structure function. It is used in conjunction with the ISL Management module. Messages that are to be sent to another spacecraft are first packaged in a isl_message structure and sent with a "transmit" message to the ISL Management module.

Table 1-16. ISL Message Data Structure

Field Name	Data	Description
sourceID	int	ID of the source spacecraft
destID	int	ID of the destination spacecraft
destModule	char[]	Name of the destination module
action	char[]	Name of the message to be sent
data	*var*	Included data
timeTag	double	Time at which the message is sent (SS1970)
routed	int	Flag indicating whether the message is being routed
attempts	int	Counter to keep track of the number of times transmission was attempted
historySourceI	int[]	History of spacecraft IDs that this message was routed through
historyTimeTag	double[]	History of times at which this message was transmitted (SS1970)

CHAPTER B. Flowcharts

This remainder of the appendix provides the flowchart diagrams that have been developed for the Control Law, Guidance Law, Team Management and Collision Avoidance modules.