# The SPHERES Facility Description

Mark O. Hilstad, John P. Enright, and Arthur G. Richards
Swati Mohan

Massachusetts Institute of Technology
Space Systems Laboratory

*SPHERES Facility Description version 2.2*

2013-08-13

http://ssl.mit.edu/spheres/gsp/

## Abstract

To reduce mission cost and improve spacecraft performance, the National Aeronautics and Space Administration and the United States military are considering the use of distributed spacecraft architectures in several future missions. Precise relative control of separated spacecraft position and attitude is an enabling technology for many science and defense applications that require distributed measurements and autonomous docking. The SPHERES testbed provides a low-risk, representative dynamic environment for the interactive development and verification of formation flight, rendezvous, and docking control and autonomy algorithms.

## Table of Contents

# 1 SPHERES Overview

The SPHERES testbed is a risk mitigation tool, providing a risk-tolerant medium for the development and maturation of formation flight and docking algorithms. The testbed was originally manifested for launch to the International Space Station (ISS) on service flight 12A.1 in July 2003; however, due to the grounding of the Space Shuttle fleet after the Columbia disaster, the Space Systems Laboratory secured an alternate means of hardware delivery to the ISS. Three SPHERES satellites reached the International Space Station on 24-April-2006 aboard Progress P21. The first operating sessions occurred on Thursday, 18-May-2006 from 10:30am-1:30pm CST.

The testbed will be operated inside the station by United States astronauts. Six degree-of-freedom (DOF) operations on the ISS are complemented by a 3 DOF low-friction air table facility in the Space Systems Laboratory (SSL) at MIT. The SPHERES separated spacecraft testbed provides investigators with a long-term, replenishable, and upgradeable platform for the validation of high-risk metrology, control, and autonomy technologies.

The SPHERES testbed consists of three free-flyer vehicles (commonly referred to as "satellites" or "spheres"), five ultrasonic beacons, and a laptop control station. The satellites are self-contained, with onboard power, propulsion, communications, sensor, and computer subsystems. They operate semi-autonomously, requiring human interaction primarily for replenishment of consumables and to command the beginning of each test. An external view of the sphere design, showing thrusters, ultrasonic sensors, propellant tank, and pressure system regulator knob, is given in Figure 1.

The elements of the satellite hardware and software are organized by subsystems representative of those on real spacecraft. The avionics, communications, propulsion, control, and state estimation subsystems are directly relevant to the ability of the spheres to perform coordinated maneuvers. The following list summarizes key features of the testbed, from an end-user point of view:

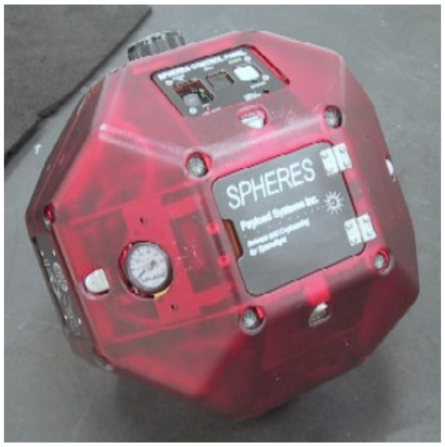

**Figure 1. A SPHERES "satellite".**

- The flight software is written in C, and runs on a Texas Instruments C6701 DSP at 167 MHz.
- Analog sensors are sampled and digitized by an FPGA at 12-bit resolution.
- The communications subsystem consists of two independent radio frequency channels. The sphere-to-sphere (STS) channel is used for communication between the spheres, and the sphere-to-laptop (STL) channel is used to send command and telemetry data between the spheres and the laptop control station.
- Actuation is provided by twelve cold-gas thrusters fed by a tank containing liquid $CO_2$ propellant. Thruster forces are fixed, but pulse modulation to a time resolution of one millisecond can be used to produce effectively variable forces.
- The position and attitude determination system has both inertial and external sensors. Gyroscopes and accelerometers are available for rapid updates to the state estimate over short time periods, and ultrasonic time of flight range measurements from wall-mounted beacons to the sphere surfaces are used to update the state estimate with respect to the laboratory reference frame.
- A periodic control interrupt can be used for implementation of fixed or variable frequency control laws. A suggested approach to the implementation of modular algorithms for use in the control interrupt is provided, and several useful modules are supplied with the GSP package.
- An event-driven background task is available to complement and augment traditional estimation and control processes. The combination of the task process with the estimation and control processes allows guest scientists significant freedom in algorithm design.

# 2 SPHERES Testing Locations

## 2.1 Laboratory

The laboratory is used to verify the expected operation of developed algorithms on the flight hardware. The hardware used in the laboratory is identical to the ISS flight hardware, and realistic imperfections, uncertainties, unmodeled effects, and hardware limitations are present; however, in the laboratory, the presence of gravity restricts the movements of the spheres.

The laboratory testbed operates on a 1.2 m × 1.2 m glass surface, mounted horizontally on a lab bench. The satellites are mounted to air carriages, which float on the glass surface by means of compressed gas. This arrangement allows planar translation and single-axis rotation. The ultrasound beacons used for range measurement are arranged in a configuration similar to that expected on the ISS. Imperfections and contaminants on the glass surface and a slight tilt of the tabletop relative to the gravity direction perturb the motion of the satellites on the surface.

It is also possible to arrange the laboratory testbed in a station configuration. For these tests, one or more spheres are suspended in the test volume. Although motion is limited, better sensor visibility and representative body-blockage effects make this technique useful for evaluating estimation algorithms in 3-D.

## 2.2 International Space Station

The micro-gravity environment of the ISS allows for maneuvers in 6 DOF. The useable test space will most likely be a 1.5 × 1.5 × 2 m (5 × 5 × 6 ft) volume. The most likely operating location for the testbed aboard the ISS is in the U.S. Node (the Unity module), where airflow rates are approximately 3 cm/s (0.1 ft/s). Based on the results of an experiment performed for us aboard the ISS, the perturbing effect of airflow onboard station is expected to be negligible. Additional environmental data will be included in this document when the operating environment and disturbances are more fully characterized. Accessibility to the testbed by the MIT SPHERES team and guest scientists is limited to occasional software updates. Telemetry and video footage will be available for evaluating algorithm performance.

# 3 Sphere Physical Properties

## 3.1 Body coordinate frame and external features

The sphere body coordinate frame is defined as follows, and is shown in Figure 2.

- The origin is located at the geometric center
- +x points in the direction of the expansion port
- +z points in the direction of the pressure system regulator knob
- +y completes a right-hand system

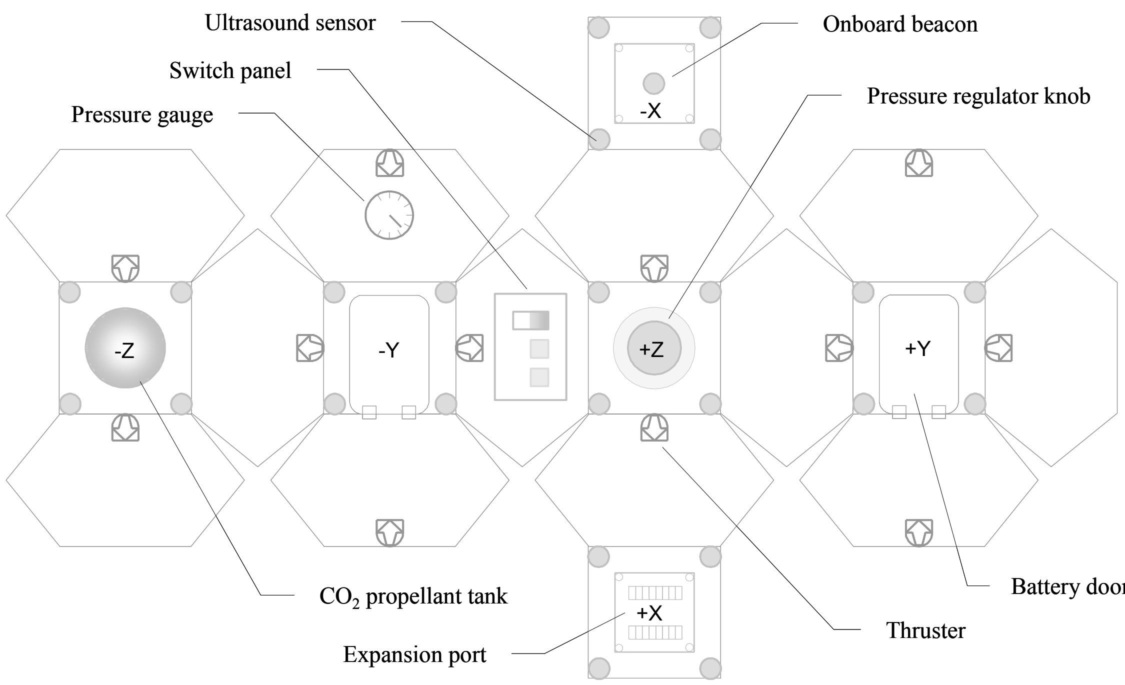

**Figure 2. An unwrapped view of a sphere, showing body frame coordinate system and physical features.**

Several features of the hardware that are visible in Figure 2, such as thrusters, sensors, and the pressure system, are discussed in detail later in this section.

## 3.2 Mass and inertia properties

Wet and dry mass and inertia properties were obtained using a CAD model and through testing in microgravity. The masses of individual parts and of the entire assembly were predicted by the CAD model and verified empirically. Wet mass values apply to a sphere with a full propellant tank, and dry mass values refer to a sphere with an empty tank. Estimates of the mass properties of the SPHERES satellites can be found in the SPHERES Specifications Sheet.

## 3.3 Propulsion system

Each satellite relies on a set of twelve on-off thrusters for management of both position and attitude. Each propellant tank contains 172 g of $CO_2$, stored in liquid form at 860 psig. A manual pressure regulator is used to decrease the thruster feed pressure to between 0 and 35 psig, and the propellant becomes fully gaseous before being exhausted through the thrusters. Detailed calibration data, such as force magnitudes for each thruster, can be found in the SPHERES Specifications Sheet.

### 3.3.1 Thrusters

The sphere thruster geometry enables the production of almost pure body-axis force or torque using only two thrusters. The twelve thrusters are arranged in six back-to-back pairs, enabling 6 DOF actuation. A diagram of the sphere thruster configuration is shown in Figure 3.

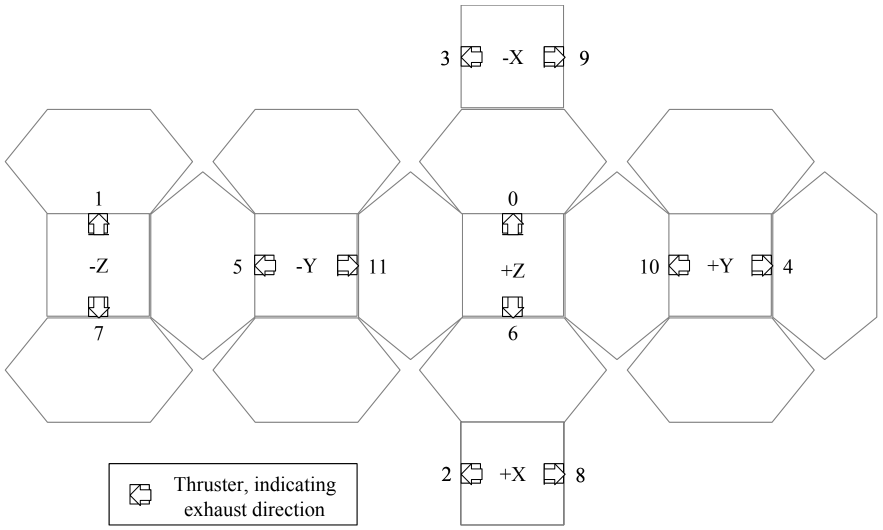

**Figure 3. Schematic view of the sphere thruster geometry.**

The thruster force and torque direction properties are listed in Table 1. For a given thruster number, these data indicate the nominal directions of the force and torque that will be produced by firing that thruster. For example, firing thruster number seven produces negative x-axis force and positive y-axis torque.

**Table 1. Thruster geometry, in the body coordinate frame.**

| Thr # | Thruster position [cm] | | | Nominal force direction | | | Nominal torque direction | | |
|---|---|---|---|---|---|---|---|---|---|
| | x | y | z | x | y | z | x | y | z |
| 0 | -5.16 | 0.0 | 9.65 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | -5.16 | 0.0 | -9.65 | 1 | 0 | 0 | 0 | -1 | 0 |
| 2 | 9.65 | -5.16 | 0.0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 3 | -9.65 | -5.16 | 0.0 | 0 | 1 | 0 | 0 | 0 | -1 |
| 4 | 0.0 | 9.65 | -5.16 | 0 | 0 | 1 | 1 | 0 | 0 |
| 5 | 0.0 | -9.65 | -5.16 | 0 | 0 | 1 | -1 | 0 | 0 |
| 6 | 5.16 | 0.0 | 9.65 | -1 | 0 | 0 | 0 | -1 | 0 |
| 7 | 5.16 | 0.0 | -9.65 | -1 | 0 | 0 | 0 | 1 | 0 |
| 8 | 9.65 | 5.16 | 0.0 | 0 | -1 | 0 | 0 | 0 | -1 |
| 9 | -9.65 | 5.16 | 0.0 | 0 | -1 | 0 | 0 | 0 | 1 |
| 10 | 0.0 | 9.65 | 5.16 | 0 | 0 | -1 | -1 | 0 | 0 |
| 11 | 0.0 | -9.65 | 5.16 | 0 | 0 | -1 | 1 | 0 | 0 |

The force and torque directions in Table 1 can be used to determine the combination of thrusters required to produce force along or torque about each body axis. The production of force or torque through a non-body axis can be achieved through a vector sum of the body-axis components.

At a nominal feed pressure of 35 psig, each thruster delivers approximately 0.112 N of force. Variability from this value is small, but the force magnitude of each thruster is slightly offset from the nominal value. Temporal deviations due to the number or combination of thrusters open at a particular time are more difficult to characterize, and the effects of these variations are currently treated as disturbances.

The on-off thrusters used on the spheres exhibit nonlinear, discontinuous, bounded behavior. Each thruster consists of a solenoid valve and a nozzle. When a thruster is commanded on, a voltage spike-and-hold circuit activates and holds open the solenoid valve. The thruster output force increases rapidly (<1 ms rise time) from zero to the steady-state thrust, following an initial delay of approximately 5 to 7 ms due to solenoid actuation dynamics. The solenoid closes rapidly when the thruster is commanded off, causing the force to return to zero within a few milliseconds.

The thrusters produce ultrasonic noise when in use, which interferes with the global position measurement system. For this reason, the thrusters are automatically disabled whenever global measurements are in progress.

# 4 Position and Attitude Determination

The Position and Attitude Determination System (PADS) has inertial and global elements that may be combined to provide position and attitude information to the spheres in real-time. The spheres PADS sensors fall into two categories: inertial navigation sensors (rate gyroscopes and accelerometers) provide high-frequency measurements in the body coordinate frame; and global navigation sensors (ultrasonic rangefinders) provide low-frequency measurements of the sphere position and orientation with respect to the "global" (laboratory fixed) reference frame.

## 4.1 Inertial Sensors

The inertial sensor suite consists of three rate gyroscopes and three accelerometers. These sensors are described in detail in the following sections.

### 4.1.1 Rate gyroscopes

Three Systron Donner BEI Gyrochip II single-axis rate gyroscopes are used to measure body-axis angular rates. Analog to digital conversion results in measurable rates in the range of approximately ±80°/s. The gyroscopes are mounted in alignment with the body axes, at the positions listed in Table 2.

**Table 2. Rate gyroscope mounting locations in the sphere body frame.**

| Sensor | Location (body frame) [cm] | | |
|---|---|---|---|
| | x | y | z |
| x-axis gyro | | 3.10 | 6.39 |
| y-axis gyro | -5.49 | | -3.24 |
| z-axis gyro | -5.49 | 3.24 | |

Additional details regarding the performance of the gyroscopes, as integrated into the SPHERES hardware, are given in Table 3.

**Table 3. Rate gyroscope performance properties**

| Quantity | Value | Units |
|---|---|---|
| Measurement range | ± 83 | °/s |
| Measurement resolution | 0.0407 | °/s |
| Noise (0 – 100 Hz, 1 $\sigma$) | < 0.05 | °/(s Hz$^{1/2}$) |
| | < 0.71 | °/s RMS |
| Low pass filter* | 300 | Hz |

*Each rate gyroscope has a first-order single-pole RC filter at 300 Hz.

The frequency response of the gyroscope, without the additional filter at 300 Hz, is shown in Figure 4.

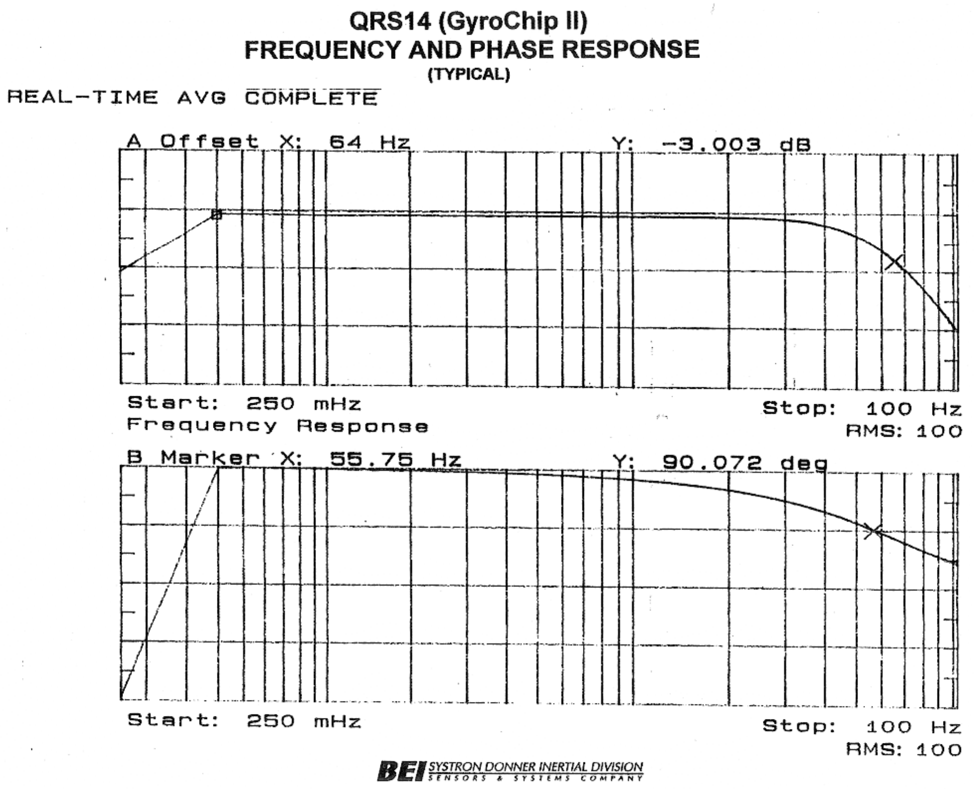

**Figure 4. Frequency response of the BEI QRS14 rate gyroscope used in the SPHERES satellites.[ 2]**

### 4.1.2 Accelerometers

Three Honeywell QA-750 single-axis accelerometers are used to measure linear acceleration. Analog to digital conversion results in a resolution of $1.23\times10^{-4}$ m/s$^2$ (12.5 $\mu$g) per count. The accelerometers are aligned parallel to, but displaced from, the body axes. The accelerometer mounting positions are listed in Table 4. The component of measured acceleration due to nonzero angular rates must be accounted for in the estimation algorithm.

**Table 4. Accelerometer mounting locations in the sphere body frame.**

| Sensor | Location (body frame) [cm] | | |
|---|---|---|---|
| | x | y | z |
| x-axis accelerometer | 5.19 | 2.17 | 3.27 |
| y-axis accelerometer | -2.66 | 3.35 | 3.30 |
| z-axis accelerometer | 3.28 | -4.37 | 3.35 |

Additional details regarding the performance of the SPHERES accelerometers, as integrated into the SPHERES hardware, are given in Table 5.

**Table 5. Accelerometer performance properties.**

| Quantity | Value | Units |
|---|---|---|

|  |  |  |
|---|---|---|
| Measurement range | ± 0.251 | m/s$^2$ |
| Measurement resolution | $1.23\times10^{-4}$ | m/s$^2$ |
| Bandwidth | < 200 | Hz |
| Noise (0 – 10 Hz) | $< 6.86\times10^{-5}$ | m/s$^2$ RMS |
| Noise (10 – 500 Hz) | $< 6.86\times10^{-4}$ | m/s$^2$ RMS |
| Low pass filter* | 300 | Hz |

*Each accelerometer has a first-order single-pole RC filter at 300 Hz.

The frequency response of the accelerometers, without the additional filter at 300 Hz, is shown in Figure 5.

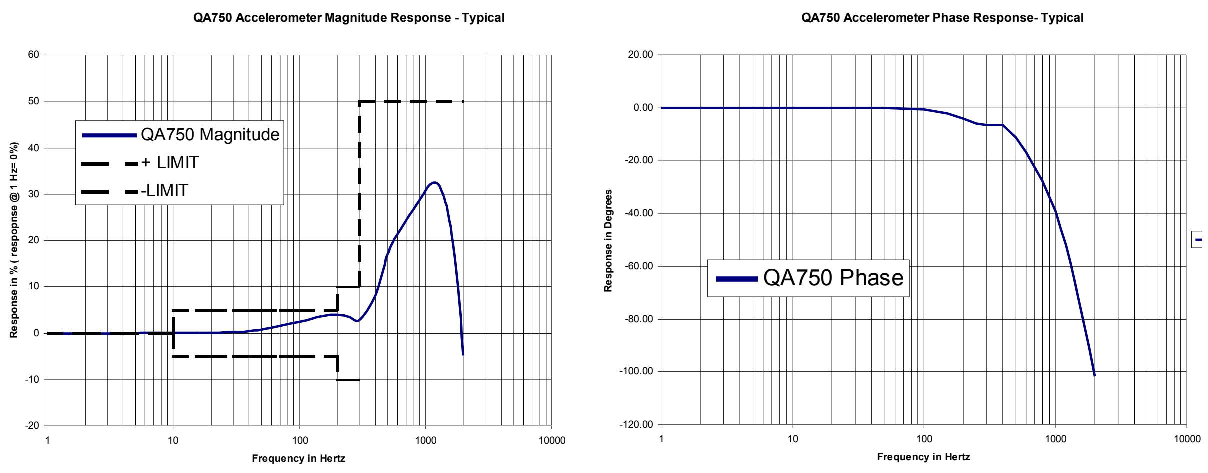

**Figure 5. Frequency response of the Honeywell QA-70 accelerometer used in the SPHERES satellites.[7]**

## 4.2 Global sensors

The PADS global metrology system allows each sphere to measure its position and attitude with respect to the global reference frame fixed to the laboratory or ISS. This system provides range measurements to points on each sphere from five external beacons mounted at known locations on the periphery of the test volume. The range measurements are calculated based on the times of flight of ultrasonic signals that are emitted from the beacons. The thrusters generate significant ultrasonic noise, so they must be turned off during global PADS measurements.

### 4.2.1 Global update process

The "global update" process is initiated when a sphere flashes an omni-directional infrared synchronization signal. This infrared signal is received by the other spheres and by the global beacons. In response to the infrared signal, the satellites turn off their thrusters, and each beacon waits a specified time and then transmits a set of ultrasonic pulses. The ultrasonic pulses are detected using threshold detection by the receivers that have a line of sight to that beacon. Times-of-flight are computed based on the difference in time between reception of the infrared and ultrasonic transmissions at each sphere receiver. These times-of-flight may be used along with knowledge of the beacon locations and the sphere geometry, to estimate the sphere position and attitude. The range measurements are shown in Figure 6 as lines between the beacon transmitters and the ultrasound receivers mounted on the sphere surface.

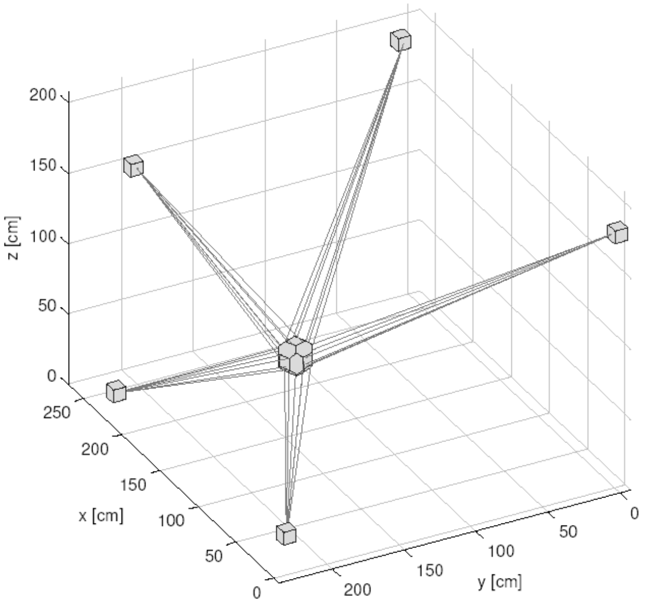

**Figure 6. The SPHERES global metrology system. Range measurements are portrayed as lines between external beacons and the sensors mounted on the sphere surface.**

In addition to the five externally mounted ultrasonic beacons, each satellite is equipped with a single body-mounted ultrasonic transmitter that may be used to determine direct inter-satellite range and bearing. A satellite receiving a signal from one of these onboard beacons can directly measure the relative distance and tip/tilt angles of the transmitting satellite.

Each external or onboard beacon waits a specified time after the infrared flash before transmitting ultrasound. The beacon timing is summarized in Table 6. The wait time of each onboard beacon can be specified from the flight software. Acceptable values of n in Table 6 are 0, 1, 2, 3, 4, 5, 6, or 7, but care must be taken to ensure that only one beacon using a particular beacon number is powered on at any given time, in order to avoid ultrasound interference.

**Table 6. Global update timing.**

| Beacon number | Beacon location | Wait time [ms] |
|---|---|---|
| 0 | external | 10 |
| 1 | external | 30 |
| 2 | external | 50 |
| 3 | external | 70 |
| 4 | external | 90 |
| 5 | sphere 1 | 10+20n* |
| 6 | sphere 2 | 10+20n * |
| 7 | sphere 3 | 10+20n * |

*Default off, with software selectable state.

By default, each sphere turns off its thrusters for 110 ms (or longer if the onboard beacons are in use) during each global update, in order to avoid corrupting the ultrasound signals sent by the external beacons. These periods of zero control authority should be considered during the algorithm design process.

## 4.2.2 Ultrasound sensor geometry

The spheres global ranging system uses pulses of 40 kHz ultrasound. Each sphere has 24 ultrasound sensors, arranged four per face on each of six faces as shown in Figure 7.

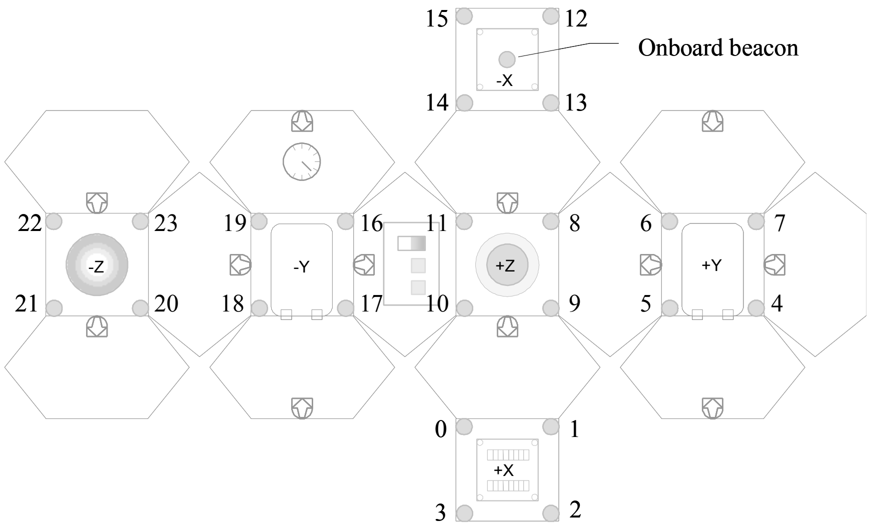

**Figure 7. Ultrasound sensor geometry and numbering scheme.**

The ultrasound sensor locations are listed in Table 7, organized by face. The faces are numbered 0 through 5, in the order +x, +y, +z, -x, -y, -z. The receiver numbering scheme presented in the table is used throughout the flight code to distinguish between the sensors.

**Table 7. Ultrasound sensor geometry and numbering scheme.**

| Face label | Receiver number | Location (body frame) [cm] x | y | z |
|---|---|---|---|---|
| +x | 0 | 10.23 | -3.92 | 3.94 |
| | 1 | 10.23 | 3.92 | 3.94 |
| | 2 | 10.23 | 3.92 | -3.94 |
| | 3 | 10.23 | -3.92 | -3.94 |
| +y | 4 | 3.94 | 10.23 | -3.92 |
| | 5 | 3.94 | 10.23 | 3.92 |
| | 6 | -3.94 | 10.23 | 3.92 |
| | 7 | -3.94 | 10.23 | -3.92 |
| +z | 8 | -3.92 | 3.94 | 10.26 |
| | 9 | 3.92 | 3.94 | 10.26 |
| | 10 | 3.92 | -3.94 | 10.26 |
| | 11 | -3.92 | -3.94 | 10.26 |
| -x | 12 | -10.23 | 3.92 | -3.94 |
| | 13 | -10.23 | 3.92 | 3.94 |
| | 14 | -10.23 | -3.92 | 3.94 |
| | 15 | -10.23 | -3.92 | -3.94 |
| -y | 16 | -3.94 | -10.23 | 3.92 |
| | 17 | 3.94 | -10.23 | 3.92 |
| | 18 | 3.94 | -10.23 | -3.92 |
| | 19 | -3.94 | -10.23 | -3.92 |

|  |  |  |  |  |
|---|---|---|---|---|
| -z | 20 | 3.92 | -3.94 | -10.23 |
|  | 21 | 3.92 | 3.94 | -10.23 |
|  | 22 | -3.92 | 3.94 | -10.23 |
|  | 23 | -3.92 | -3.94 | -10.23 |

These values are available through functions in the flight software.

In addition, each sphere is equipped with a single ultrasonic transmitter, for use in direct ranging between the spheres. This onboard beacon is centrally positioned on the -x face, on the body frame x-axis at a distance of -10.23 cm from the geometric center.

The ultrasound receivers used on the spheres are directional (60° full-cone), and are aligned with bore sight normal to their mounting surfaces. The directionality properties of the sensors are shown in Figure 8.

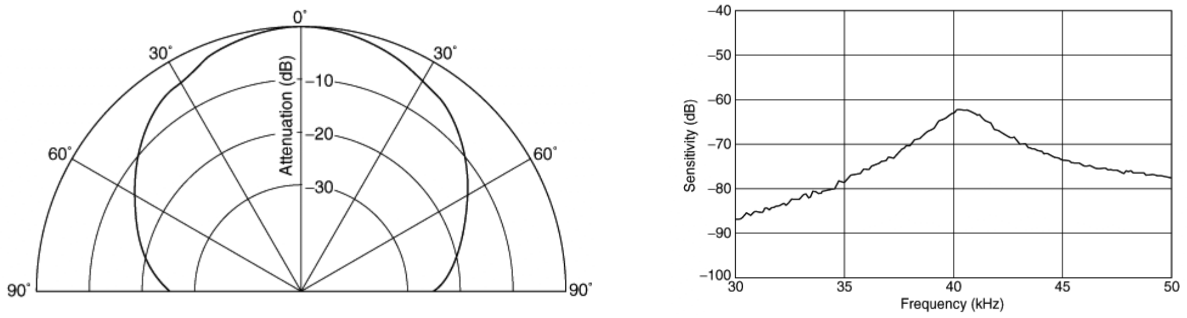

**Figure 8. Sensitivity properties for the Murata MA40S4R ultrasound receiver.[9]**

## 5 Software Overview

The GSP interface to the SPHERES flight software consists of two categories of interface functions: primary and secondary. In short, the Guest Scientist must provide primary functionality, and may use secondary functionality. There are several primary functions, all of which are all located in the file `gsp.c`. The primary functions are where the interface to the existing SPHERES code takes place, and the code internal to these functions may be freely modified by the guest scientist. Secondary functions are those that are available for use by the guest scientist, but may not be modified. The secondary functions are described in the Application Program Interface (API) section of this document, and the prototypes of the secondary functions may be found in the `*.h` files included with the GSP interface package.

The primary functions can be organized into three groups: initialization, periodic, and event-driven. Initialization functions are used to set program and test-specific values. Periodic functions may be used for fixed-frequency control algorithms, and to collect and process sensor data. Event-driven tasks provide the guest scientist with a means to implement algorithms that do not fit conveniently into the framework of the periodic processes. In addition, the task provides a means for performing long-term, non-real-time, or low-priority computation. The function interface (the arguments) to each of the primary functions is fixed, but the guest scientist can choose to leave empty any functions that are not required to implement the set of algorithms being tested.

Throughout this section, references will be made to specific primary and secondary interface functions. In general, primary functions will be described at length, while secondary functions will be described in passing. A complete list and description of secondary functions can be found in the Application Program Interface section of this document.

## 5.1 Programs, tests, and maneuvers

SPHERES operations are divided into three hierarchical levels: programs, tests, and maneuvers. Each program is associated with a particular executable file; therefore, each set of guest scientist source code files submitted will constitute a program. A program begins when an executable is uploaded into onboard memory, a satellite is powered on, or the CPU is reset. Each program consists of one or more tests.

Each test is a standalone experiment, and is accompanied by a description file on the laptop. Once a program is running, a test commences when explicitly commanded by the operator through the laptop control station. The test ends either when the software signals its completion or when aborted by the operator through the laptop interface or the hardware control panel. The execution of tests is controlled by the operator; tests may be run multiple times and in arbitrary order, and programs must be written to take this operational flexibility into account. When a test completes, the test conductor is notified through the laptop, and the satellites drift freely until the next test is commanded. Test completion is signaled through software by a call to the function `ctrlTestTerminate(…)`. Guest scientist code must call this function to explicitly end each test. The current test number and elapsed test time are available at any time through the functions `ctrlTestNumGet()` and `ctrlTestTimeGet()`, respectively. The test time counters on the separate SPHERES satellites are synchronized to within one millisecond (and reset to zero) whenever a new test command is received.

Each test may in turn consist of a linear or non-linear sequence of maneuvers. Maneuvers are a convenient bookkeeping convention, and the current maneuver number is automatically downloaded in the telemetry stream once per second in a state of health packet. The concept of the maneuver is intended to assist guest scientists with implementing complex sequencing within a single test. Maneuver numbers and elapsed maneuver times are available to the guest scientist through the functions `ctrlManeuverNumGet()` and `ctrlManeuverTimeGet()`, respectively. The function `ctrlManeuverTerminate()` terminates the current maneuver and increments the maneuver number automatically. Similarly, the function `ctrlManeuverNumSet(…)` may be used to terminate the current maneuver and proceed to a specified maneuver number. Guest scientist code may call one of these functions to explicitly end each maneuver.

Maneuvers can be used to separate a complex motion into a series of simpler movements. For example, a test may begin with each sphere translating from its deployment location to a specified initial position suitable for the test. In this maneuver, the desired positions could simply be fixed, and a simple PID control law on each sphere could be used to perform the translation. When all the spheres have arrived at their desired locations, a new maneuver begins wherein the spheres perform a coordinated formation rotation using a more complex decentralized control law and a distributed estimation scheme. Maneuvers may be defined by a specific trajectory, control law, estimation algorithm, pulse modulation scheme, or any other parameter.

Finally, it is requested that the final maneuver in each test null any residual velocity, in order to reduce drift after the test terminates. This saves time by allowing the operator to proceed to the next test without manually capturing and repositioning the satellites.

## 5.2 Summary of primary interface functions

Seven functions and one header file comprise the primary interface to the existing SPHERES flight software. These functions are listed in Table 8, along with short descriptions of their uses.

**Table 8. Typical uses for the primary interface functions.**

| Function name | Description |
|---|---|
| `gspPadsInertial(…)` | Perform state estimation based on inertial data. Called periodically. |
| `gspPadsGlobal(…)` | Record global data. Called at the end of each beacon's transmission period. |
| `gspControl(…)` | Apply control laws and set thruster on-times. Called periodically. |
| `gspTaskRun(…)` | Event-driven task for estimation, control, and communications. Called whenever a masked event occurs. |
| `gspIdentitySet(…)` | Set satellite identity. The first primary interface function called. |
| `gspInitProgram(…)` | Initialize communications and other subsystems. Must contain certain initialization functions for multi-sphere operations to work correctly. |
| `gspInitTask(…)` | Specify task trigger mask. |
| `gspInitTest(…)` | Perform test-specific configuration. Called prior to starting each test. |

Each of these functions has pre-defined arguments, but the function contents and any internal sub-functions may be freely designed and written by the guest scientist, with the exception of `gspIdentitySet(…)` and `gspInitProgram(…)`, which must contain certain function calls that set the sphere identity and other important properties. The inputs and outputs of each interface function are pre-defined, and each function is called based on one or more trigger events, as shown in Figure 9. Each satellite has a unique copy of the source file `gsp.c` that contains these functions, and a unique copy of the header file `gsp.h`, also available for modification as desired by the guest scientist.

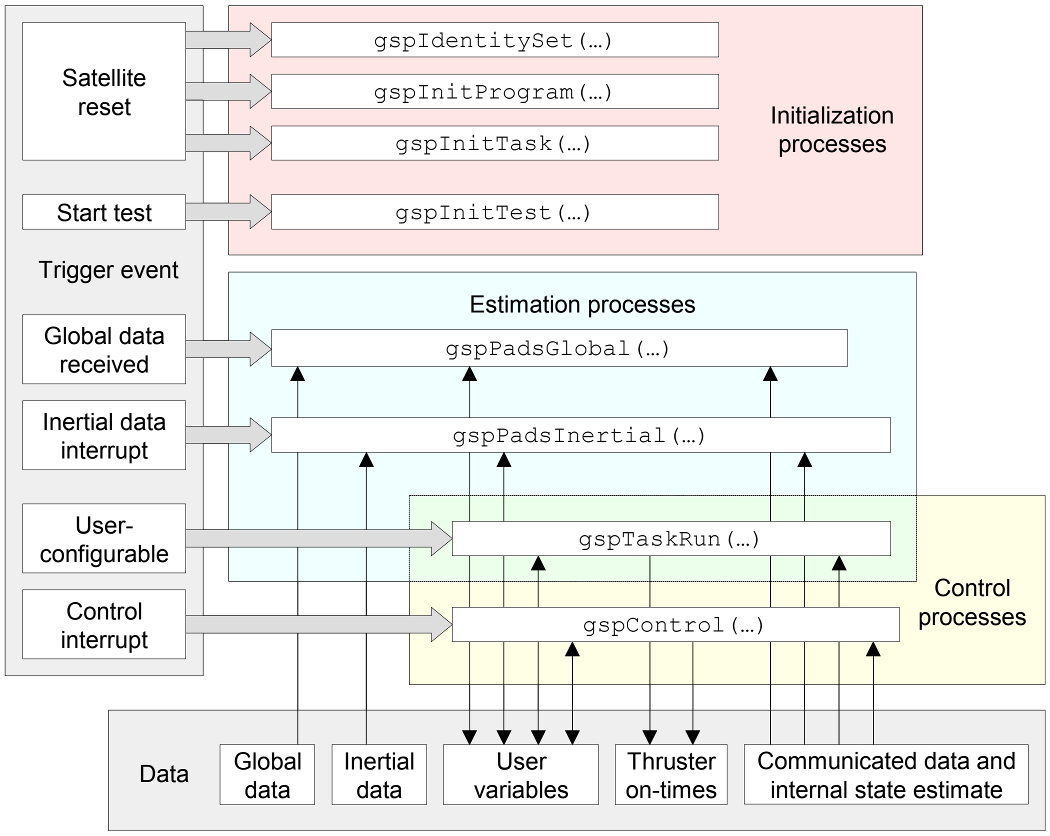

**Figure 9. Data flow and trigger events for user processes.**

## 5.2.1 Priority, pre-emption, and data integrity

The CPU supports several levels of process priority. In order from highest to lowest priority are hardware interrupts, software interrupts, and background tasks. The primary interface functions operate at several levels of priority, as shown in Table 9.

**Table 9. Software process priority for primary interface functions.**

| Priority | Process description | Primary interface function | Trigger type |
|---|---|---|---|
| (highest) | Propulsion hardware management | -- | Periodic |
| | Inertial sensor sampling | -- | Periodic |
| | Control | gspControl(…) | Periodic |
| ↓ | Inertial data processing | gspPadsInertial(…) | Periodic |
| | Global data processing | gspPadsGlobal(…) | Event-driven |
| | Communications | -- | Event-driven |
| (lowest) | Task (control, estimation, etc) | gspTask(…) | Event-driven |

Because the flight software is a multi-process, multi-priority application, issues of data integrity arise when accessing shared memory from multiple processes. In particular, it is possible for a higher-priority function to interrupt a lower-priority function while the lower-priority function is engaged in a write operation to shared memory. If the higher-priority process reads that memory, it may read a corrupt combination of old and new data. Similarly, if the higher-priority process interrupts the lower-priority process during a low-priority read, the lower-priority process will read a corrupted combination of old and new data.

In order to guarantee data integrity, users wanting to directly share variables between two processes (e.g. the task and control interrupt) must use the `atomic_memcpy(…)` function when reading or writing multi-element (e.g. array) data from within the lower-priority process. Accessing the shared data from the higher-priority process does not require special treatment. All secondary interface functions automatically guarantee well-defined memory behavior.

Due to limitations with the DSP/BIOS operating system used on the SPHERES hardware, memory cannot be dynamically allocated from within an interrupt. This means that calls to memory handling functions such as `malloc(…)` and `dealloc(…)` must be used only in the `gspInitProgram()` function.

## 5.3 State vector

Table 10 shows the SPHERES convention for state vector elements. This convention is followed exclusively in the following standard utilities:

- Standard estimator
- Standard controllers
- Telemetry data reduction script and plotting utilities
- Background telemetry

Guest scientists may use their own state vector definitions, but should be aware that their resulting code will be incompatible with the standard utilities provided by the SPHERES team. To maintain compatibility with the standard utilities, it is recommended that custom state vectors consist of the elements given in Table 10 appended with custom state quantities.

**Table 10. State vector suggested elements and order.**

| Array position | Defined index | Element | Units |
|---|---|---|---|
| 0 | POS_X | Position, x-axis[1] | m |
| 1 | POS_Y | Position, y-axis | m |
| 2 | POS_Z | Position, z-axis | m |
| 3 | VEL_X | Velocity, x-axis[2] | m/s |
| 4 | VEL_Y | Velocity, y-axis | m/s |
| 5 | VEL_Z | Velocity, z-axis | m/s |
| 6 | QUAT_1 | Quaternion, vector component 1[3] | normalized |
| 7 | QUAT_2 | Quaternion, vector component 2 | normalized |
| 8 | QUAT_3 | Quaternion, vector component 3 | normalized |
| 9 | QUAT_4 | Quaternion, scalar component | normalized |
| 10 | RATE_X | Angular velocity, x-axis[4] | rad/s |
| 11 | RATE_Y | Angular velocity, y-axis | rad/s |
| 12 | RATE_Z | Angular velocity, z-axis | rad/s |

1 . The position is expressed with respect to the global frame.
2 . The velocity is expressed with respect to the global frame, in components of the global frame.
3 . The quaternion is expressed as the rotation from the global frame to the body frame.
4. The angular rate is expressed with respect to the global frame, in components of the body frame.

## 5.4 Naming conventions

The functions and variables in the SPHERES flight code follow (for the most part) the SERTS naming conventions, which can be found online at http://www.ee.umd.edu/serts/bib/unpublished/naming.pdf.

For purposes of convenience, the conventions most applicable to the ensuing discussion are repeated here.

Global variables begin with a software element identifier specifying the functional group to which the variable belongs; e.g., `prop_someVariable` could be a global variable that is used by the propulsion system. The definition of this variable would be found in the header file corresponding to its identifier, namely, `prop.h`. Similarly, exported (global) functions begin with an identifier; e.g., `padsGlobalPeriodSet(…)` is used to set the period of the global update sequence, and may be called by any process. The header for this function can be found in `pads.h`. Table 11 lists the pre-defined software element identifiers.

**Table 11. Software element identifiers.**

| Identifier | Subsystem |
|---|---|
| `comm` | Communications |
| `ctrl` | Control |
| `gsp` | Guest Scientist Program |
| `math` | Mathematics |
| `pads` | Estimation |
| `prop` | Propulsion |
| `sys` | System |

Note that the primary interface functions (see Table 8) begin with the `gsp` identifier. Functions not beginning with `gsp` may be called by guest scientist code, but may not be modified by the guest scientist.

Local variables are not preceded by an element identifier. In general, variables begin (after the identifier, if applicable) with a lowercase letter, and functions begin (after the identifier, if applicable) with an uppercase letter. When using static or global variables, be certain that you reset variables to initial values, if required, in `gspInitTest(…)` or elsewhere, as appropriate.

## 5.5 Function description conventions

Functions descriptions are presented with the function name and return type, a description, and a list of argument types. Each argument is numbered for easy reference. For example,

| **`int demoFunction(…)`** | | |
|---|---|---|
| An example used to demonstrate the function description conventions used in the SPHERES GSP interface document. As can be seen below, this function has two input arguments, the first of type `char*`, and the second of type `unsigned int`. From the function name, it can be seen that the return value is of type `int`. | | |
| *1* | *`char*`* | This might be passed the address of a character string, for example. |
| *2* | *`unsigned`* | This might be passed the length of that character string. |

Primary interface functions are surrounded by triple lines on all sides, and secondary functions are surrounded by double lines on the sides and single lines on top and bottom.

## 5.6 Individually customized software access

The GSP interface has been designed to provide a simple, flexible interface for SPHERES experiments. Every effort has been made to anticipate the needs of researchers in the fields of control, estimation, and autonomy. Researchers should contact the SPHERES team if they require specialized access to SPHERES hardware or software, beyond the interface described here. Feasibility of any interface customization is considered on an individual basis.

# 6 Estimation

The GSP interface provides the guest scientist with the ability to implement a custom estimation scheme to be used in the determination of absolute and relative position, velocity, attitude, and angular rate. The estimation interface consists of three processes: two measurement-activated processes and the configurable event-driven task. This section describes the measurement-activated processes `gspPadsInertial(…)` and `gspPadsGlobal(…)`; the task is described separately in Section 8.

## 6.1 Measurement-triggered functions

The inertial measurement-based process `gspPadsInertial(…)` occurs at a high fixed frequency, while the global measurement-based process `gspPadsGlobal(…)` occurs less frequently, whenever global metrology measurements are received during a global update. More precisely, `gspPadsGlobal(…)` is called at the end of each beacon's transmission window during global updates whether there are meaningful (non-zero) data or not.

Custom estimation algorithms may be implemented through the functions `gspPadsInertial(…)` and `gspPadsGlobal(…)`, which are called when new inertial and global data, respectively, are available. The arguments of these functions are pre-defined, but guest scientists are free to modify the function contents as desired, and to pass data between functions using global memory.

A limitation to the use of these functions is that all computation must be completed within much less than one millisecond. This is necessary because these processes are actually launched by a 1 kHz hardware interrupt, and concurrently running two instances of a single interrupt will cause undesired behavior. To avoid this problem, these two functions should be used only to copy data and/or to perform simple estimation tasks. Complicated or time-consuming tasks, such as running a Kalman filter, can be easily accommodated by launching a task process from within `gspPadsInertial(…)` or `gspPadsGlobal(…)`. An example of this approach is provided in the `gsp.c` template file included with the simulation.

Included estimation algorithms can be found in the sub-directory `standard\estimation\`.

### 6.1.1 Inertial

A high-priority sensor sampling routine reads and archives raw accelerometer and gyroscope measurements at a user-definable frequency. These archived measurements are then passed to the lower-priority data processing function `gspPadsInertial(…)`, which is also called at a user-definable frequency, to interpret the archived inertial data.

| **`gspPadsInertial(…)`** | | |
|---|---|---|
| The periodic estimation interrupt, called to interpret new inertial sensor data. A primary interface function. | | |
| *1* | *`IMU_sample*`* | (`accel`) The address of the raw accelerometer counts. |
| *2* | *`IMU_sample*`* | (`gyro`) The address of the raw rate gyroscope counts. |
| *3* | *`unsigned`* | (`num_samples`) The number of samples each of accelerometer and gyroscope counts. |

The rate at which inertial sensors are sampled and inertial data are processed can be changed with the command `padsInertialPeriodSet(…)`. Before setting the inertial sampling period, memory space must be set aside to store the samples. This is accomplished using `padsInertialAllocateBuffers(…)`.

The maximum allowed inertial sensor sample frequency is 1 kHz, corresponding to a 1 ms period. Calibration utilities that provide sphere-specific calibration and scaling factors will be provided with a future release.

### 6.1.2 Global

The function `gspPadsGlobal(…)` is called at 20 millisecond intervals during global updates, once at the end of each beacon transmission window. Because processing of global data is time-consuming, it is recommended that `gspPadsGlobal(…)` be used to archive the beacon data, and the task be used to perform the estimation using those data.

| **`gspPadsGlobal(…)`** | | |
|---|---|---|
| A primary interface function that is called whenever new global metrology measurements are available. Called at the end of each beacon's transmission window. | | |
| *1* | *`unsigned`* | (`beacon`) The beacon number. |
| *2* | *`beacon_measurement_matrix`* | (`measurements`) The range measurements from that beacon to each receiver. |

Global updates may be configured as either periodic or on-demand. Periodic global updates are configured using the function `padsGlobalPeriodSet(…)`. Because the thrusters create ultrasonic noise, they are automatically disabled during global updates. It is therefore important to balance the desire for frequent global updates with the resulting loss and irregularity of control authority. It is good practice to designate only one sphere to request global updates, as multiple spheres requesting global updates may result in unexpected behavior and excessive thruster off-time.

Instead of performing periodic global updates automatically, a routine may explicitly request updates at particular times. A call to the function `padsGlobalTriggerNow(…)` results in an immediate infrared flash, initiating a single instance of the global update process. If this function is called when periodic global updates are being used, the next periodic update will occur one update period after the explicitly requested flash, rather than one period after the last periodic flash.

Low-level functions ensure that only one global update occurs at a time, in order to prevent corruption of range data. Therefore, periodic or on-demand updates will be cancelled if they are scheduled to begin during an update already in process. Global updates can be performed at a maximum of 9 Hz when using only the five external beacons, and 5.8 Hz when using all 8 beacons. By default, the global update is performed at 2 Hz.

## 6.2 Internal state estimate

The internal (MIT) estimator algorithm determines the position, velocity, attitude quaternion, and angular rate of the sphere, at a low update rate. The internal estimator may not be disabled, but the guest scientist may choose to ignore the internal state estimate and use the results generated by a custom estimator instead. The state vector used by the standard estimator follows the state element convention described in Section 5.3. The current value of the internal state estimate may be retrieved at any time using the function `padsStateGet(…)`.

Guest scientists unconcerned with the details of the estimation process (e.g. interested only in control or autonomy experiments) may use estimation functions provided by the MIT SPHERES team.

The internal state estimate functionality has not yet been implemented.

## 6.3 Onboard direct-ranging beacons

The onboard beacons are disabled by default, in order to minimize thruster off-time. The onboard beacons may be enabled or disabled at any time using the function `padsBeaconNumberSet(…)`. Note that under the current implementation, enabling the beacon on any one sphere does not automatically increase the thruster quiet time on any of the satellites. The number of active beacons registered in the memory of each satellite must be explicitly updated using the function `padsInitializeFPGA(…)` locally on each satellite. This procedure may be simplified or automated in the future.

# 7 Control

Two separate processes are available for implementing control algorithms. A periodic interrupt process is available for performing repetitive, time-dependent operations such as following a curved trajectory and setting thruster on-times, and an event-driven background task is available for performing long-term, low-priority computation such as future trajectory planning. Both of these processes have fixed arguments, but guest scientists are free to use global memory to store or exchange additional data between any of the GSP processes. The task provides great freedom in algorithm design, while the control interrupt provides a simple structured interface. The task is described in Section 8.

For researchers primarily concerned with estimation, autonomy, or limited aspects of control problems, the MIT SPHERES team provides a standard set of modular control functions that can be called in the control interrupt to perform simple maneuvers. Development of these utilities is ongoing, and additional modules will be made available as they are developed.

## 7.1 Control interrupt interface

The control interrupt, consisting of the primary function `gspControl(…)` and some background housekeeping routines, is typically used to implement a periodic control law, with the result of setting thruster on and off times for the next control period. The period of the control interrupt may be set and queried using the functions `ctrlPeriodSet(…)` and `ctrlPeriodGet()`, respectively.

| **`gspControl(…)`** | | |
|---|---|---|
| The periodic control interrupt, intended for implementing fixed-frequency control laws and setting thruster on-times. A primary interface function. | | |
| *1* | *unsigned* | (`test_number`) The current test number. |
| *2* | *unsigned* | (`test_time`) The elapsed test time. |
| *3* | *unsigned* | (`maneuver_number`) The current maneuver number. |
| *4* | *unsigned* | (`maneuver_time`) The elapsed maneuver time. |

The MIT SPHERES team has developed a set of interface guidelines that facilitate rapid test development and the simple and effective reuse of existing code. Following these guidelines is not necessary, but doing so assists in operational organization, and allows the guest scientist to use a supplied set of modular algorithm blocks that satisfy common algorithmic needs. These interface guidelines are described in the following sections.

## 7.2 Implementation suggestions

The guidelines provided in this section are suggestions that are intended to assist guest scientists in the implementation of algorithms. These guidelines are motivated by algorithmic, operational, and simplicity considerations based on the experiences of the SPHERES team, but they are not development rules. Guest scientists are free to design the contents of `gspControl(…)` as desired.

The algorithmic processes that occur within a particular maneuver can often be broken down into the following categories:

- Command: generate a reference quantity, for example the current desired state on a particular trajectory.
- Control law: application of a control law to the state and reference quantities, leading to desired force and torque or $\Delta V$.
- Mixer: calculation and assignment of thruster on-times based on force and torque or $\Delta V$, thruster geometry, and other considerations.
- Termination: compare current conditions to some set of maneuver termination conditions, and signal maneuver completion when the termination conditions are met.

Some of these maneuver elements may change during a test, and others may not. Implementing each element in a separate algorithm module simplifies the development process, and enhances the ability to make incremental changes when problems arise. For example, using a well-tested mixer module during the debugging of a new control law module reduces the possibility that an algorithmic error in the mixer module is to blame for any unexpected behavior.

A suggestion for the organization of maneuvers is shown in Figure 10. When a test begins, the maneuver number is set to one by the underlying code. The periodic function `gspControl(…)` can be written with a switch statement such that each time it is called, the command, control law, mixer, and terminator corresponding to the appropriate maneuver number are called. When the terminator signals that the maneuver is complete, the maneuver number increments (automatically) and the next call to `gspControl(…)` calls the functions corresponding to the next maneuver number. This process of terminating and incrementing the maneuver number continues until the test is complete.

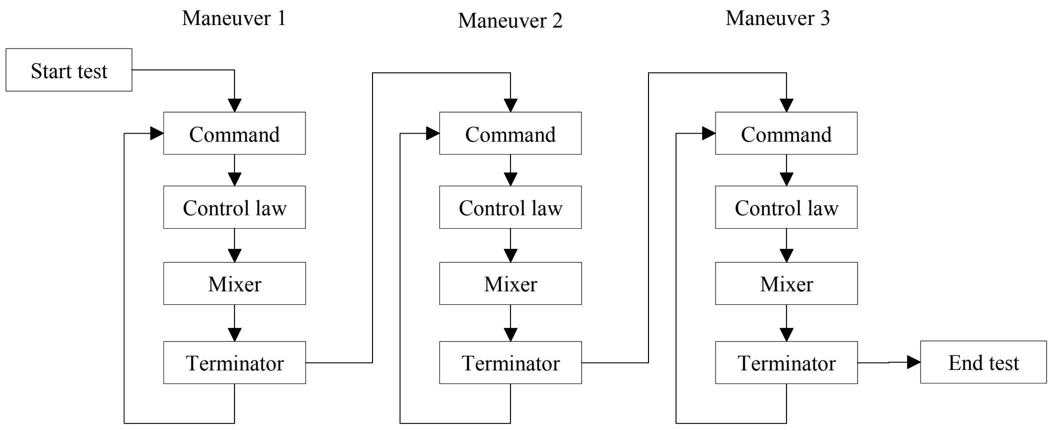

**Figure 10. Suggested process diagram for a sequence of maneuvers.**

A simple, single-maneuver example of how `gspControl(…)` might look is shown in Figure 11. In this example, the command function is `yourCommandFunction(…)` (i.e. a function provided by the guest scientist), the attitude and position control laws, are `ctrlAttitudeNLPDwie(…)` and `ctrlPositionPD(…)`, respectively, and the mixer is `ctrlMixSimple(…)`. The terminator function `ctrlTerminateTestTimed(…)` ends the test after a specified time has elapsed.

```
void gspControl(..., unsigned int maneuver_time, ...)
{
   // create state vector arrays
   static state_vector    actual, desired, error;
   static control_vector  control;
   static prop_time       thrusters;

   // assuming state estimate is updated elsewhere...
   memcpy(actual, my_state_estimate, STATE_LENGTH);

   // get the current desired state from your own algorithm
   yourCommandFunction(maneuver_time, desired);

   // determine the state error
   findStateError(error, actual, desired);

   // fill out control array (real gains would not be 1.0)
   ctrlAttitudeNLPDwie(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, error, control);
   ctrlPositionPD(1.0, 1.0, error, control);

   // determine when to turn thrusters on and off
   ctrlMixSimple(&thrusters, control, state,
                 10, ctrlPeriodGet(), FORCE_FRAME_INERTIAL);

   // command the thruster on and off times
   propSetThrusterTimes(&thrusters);

   // end the test 60 seconds into the maneuver
   ctrlTerminateTestTimed(maneuver_time, 60000, TEST_RESULT_NORMAL);
}
```

**Figure 11. Example contents for a simple `gspControl(…)`.**

The various secondary interface functions used in Figure 11 are described in Appendix A.

## 7.3 Thruster actuation

### 7.3.1 Commanding actuation

Typically, a control law generates continuous force, torque, or ΔV requests. These requests must be converted into individual thruster requests (forces, for example) based on the thruster geometry. Because the thrusters are on/off actuators, a pulse modulation algorithm must be employed to convert the forces into discrete on and off times for each thruster. Functions fulfilling this purpose are referred to as "mixers." The function `ctrlMixSimple(…)` is an example of a typical mixer utility. This function calls `propSetThrusterTimes(…)` to actually assign the thruster on and off times to the propulsion subsystem.

Standard mixers can be found in the directory `standard\mixers\`.

### 7.3.2 Calibration

Steady-state deviation in the force magnitude produced by each thruster from the nominal value has been measured in the laboratory, and calibration data are recorded in the flight software. The presence of the calibration values allows the guest scientist to use an idealized model for thruster forces, and the same idealized model for each sphere. The calibration data may be applied to the on-time values determined by the control algorithm before the pulse modulation algorithm is applied. The interface to calibration functions will be included in a future revision.

## 7.4 Standard control modules

For guest scientists who are not concerned with writing control algorithms, the SPHERES team provides standard control modules to implement simple control laws. Examples of these can be found in the sub-directory `standard\control\`.

# 8 Event-driven task

The purpose of the event-driven task is to provide guest scientists with a means to implement algorithms that do not fit conveniently into the framework of the periodic control and measurement-based estimation processes. In addition, the task provides a means for interpreting received communications data, and performing long-term, non-real-time, or low-priority computation. Task algorithms are implemented in the primary function `gspTaskRun(…)`.

| **`gspTaskRun(…)`** | | |
|---|---|---|
| An event-driven primary function. Runs whenever a masked event occurs. The trigger mask is set using `taskTriggerMaskSet(…)`. | | |
| *1* | *`unsigned`* | (`gsp_task_trigger`) The trigger type, an element of Table 12. |
| *2* | *`unsigned`* | (`extra_data`) An extra datum, the meaning of which depends on the type of trigger event. |

This function is called whenever an event occurs that matches one of the events specified in a trigger event mask. The trigger event mask may be specified at any time using the function `taskTriggerMaskSet(…)`, and each call to this function replaces the previous value of the trigger mask. Multiple triggers can be placed in the mask, resulting in calls to `gspTaskRun(…)` whenever one of the masked events occurs. Valid trigger events are listed in Table 12.

**Table 12. Valid trigger events for `gspTask(…)`.**

| Trigger event type | Occurs when… | Extra data |
|---|---|---|
| `CTRL_DONE_TRIG` | After each call to `gspControl(…)`. | Test number |
| `MESSAGE_TX_ACK_TRIG` | A packet acknowledgement is received. | Packet number |
| `DATA_TX_DONE_TRIG` | Data transmission thought to be complete. | Transaction ID |
| `DATA_RX_DONE_TRIG` | New data have been received. | Transfer tag |
| `PADS_GLOBAL_START_TRIG` | At the start of each global update period. | (Undefined) |
| `PADS_GLOBAL_BEACON_TRIG` | New range data are available. | Beacon number |
| `PADS_INERTIAL_TRIG` | New inertial data are available. | Number of samples |
| `TASK_TIME_TRIG` | The task timer reaches the specified time. | (Undefined) |
| `COMM_GSP_TRIG` | On debug messages (?). | Debug command |
| `TEST_START_TRIG` | At the start of each test. | Test number |
| `GSP_USER_TRIG` | `taskPostUserTrigger(…)` is called. | User-defined |

The suggested approach for handling multiple trigger types is to place a switch statement inside `gspTaskRun(…)` that determines from the first argument which type of event occurred, and acts accordingly. The meaning of the second argument to `gspTaskRun(…)` depends on the type of triggering event. For example, for an event caused by a PADS_GLOBAL_BEACON_TRIG, the second argument tells the beacon number.

The primary function `gspInitTask(…)` runs during task startup. This function is used only to set the task trigger mask with `taskTriggerMaskSet(…)`.

| **`gspInitTask()`** |
|---|
| An initialization routine for the GSP task function. Runs once, at startup. |

The functional flow of the task process can be visualized by the flow diagram in Figure 12. The task process begins only once, after being initialized through `gspInitTask(…)`. After initialization, it enters an infinite loop wherein events are compared against the mask as they occur. When an event occurs that is in the mask, `gspTaskRun(…)` executes. After completion of `gspTaskRun(…)`, the process waits for the next masked event to occur.

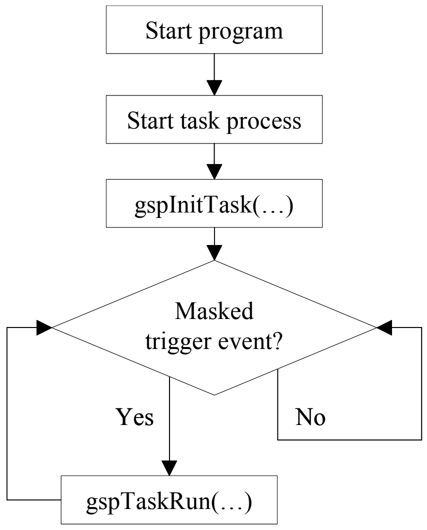

**Figure 12. Task process flow diagram.**

Because there is not a way to trigger on the end of a test, a separate function was written and inserted in the SpheresCore code directly before the test ends. This function, `gspEndTest(…)`, will run immediately before the test ends, no matter the reason for that test to end. This function is useful if special procedures are required, such as turning off external hardware, when a test completes in an off-nominal manner. Mainly this function exists for safety reasons as a catchall when a test ends.

| **`gspEndTest(…)`** | | |
|---|---|---|
| Runs once, when a test ends, regardless of how that test ended. `TEST_END_TRIG` must be defined in the `gsp.h` file for this function to be executed. Do not define `TEST_END_TRIG` if the `gspEndTest(…)` function is not required. | | |
| *1* | *`unsigned int`* | (`test_number`) The number of the current test. |
| *2* | *`unsigned int`* | (`ctrl_result`) Return number for how the test ended. |

Note that the user must define `TEST_END_TRIG` in the `gsp.h` file for this function to be executed. There are a couple `#ifdef` statements around relevant sections of code so that when the user is not using this function, the relevant sections will not add to the compiled size of SpheresCore and the test project. Nominally the user will likely not use this function.

# 9 Communications

## 9.1 Overview

Two separate channels are available for command and telemetry communications. The sphere-to-sphere (STS) channel is used to pass information between two or more spheres, and the sphere-to-laptop (STL) channel is used for the transfer of commands from and telemetry to the laptop control station. Relevant properties of the STS and STL communications channels are given in Table 13.

**Table 13. Communications system properties**

| Property | sphere-to-sphere (STS) | sphere-to-laptop (STL) |
|---|---|---|
| Radio frequency (MHz) | 916.5 | 868.35 |
| Packet data size (bytes) | 32 | 32 |
| Bandwidth (packets/s) | 70 | 70 |

It is important to ensure that the available communications capacity on each channel is not exceeded since data may be lost. The rate at which each buffer is emptied depends on the data rate of the channel that it serves.

Two types of communication occur on each channel: background and foreground. Background refers to the automatic, periodic transmission of standard information, such as state and basic housekeeping data. Foreground refers to data explicitly sent by the guest scientist. Later sections describe each of these in detail.

### 9.1.1 Time-division multiple access

The satellites and laptop (referred to in the context of communications as "stations") use a time-division multiple access (TDMA) protocol to enable transmission of information by all stations. During the software initialization process, a transmit window and frame length must be specified for each channel and station. The transmit window is the portion of the TDMA frame during which that particular station may transmit. The transmit windows on a given channel for different stations must not overlap, and the transmit windows for the two channels on a particular sphere should not overlap. For optimal performance, the window length should be approximately equal to the time required for an integral number of packet cycles (serial transmit followed by RF transmit), which is approximately 12 ms for a standard packet. The frame length may be chosen arbitrarily by the guest scientist, but it should be noted that a frame duration of no more than half the control period is necessary to guarantee that time for packet delivery is available within a given control period.

A simple function, `commTdmaStandardInit(…)`, is provided to set default values for the TDMA parameters.

### 9.1.2 Loss of communications

The laptop acts as a master time reference, and periodically broadcasts frame synchronization packets on the STL channel. Each time a sphere receives a synchronization packet, it resets a frame counter.

Failure to receive several of these synchronization packets in a row is interpreted as loss of communications due to departure from the test volume, and results in safing. The satellite response to a safing event is to cease all transmissions and disable the thrusters. This behavior is required to address ISS safety requirements.

## 9.2 Background Communications

Several types of important data may be regularly exchanged over the STL channel between the satellites and the laptop in processes transparent or semi-transparent to the guest scientist; these processes are termed background communications. Background telemetry provides a convenience service by automatically transmitting the current state estimate to the laptop and the other satellites. This simplifies data sharing and post-processing data reduction, since the telemetry data are in a standard form. By default, the background telemetry data contain the internal state estimate (note: the internal state estimate is not yet implemented), but this can be changed using `commBackgroundPointerSet(…)`, which tells the background telemetry process to read its state data from a user-specified state vector. The pointer can be reset to the default value with `commBackgroundPointerDefault()`. State data sent by one satellite through background communications can be retrieved by another satellite by calling `comBackgroundStateGet(…)`. Background telemetry is periodic, and the period at which the state estimate is sent can be changed at any time using the function `commBackgroundTelemetryPeriodSet(…)`. This function retrieves both the state estimate and the time at which the estimate was received from the specified satellite.

The internal state estimate is also transmitted over the STL channel with the state of health packet at a default rate of 1 Hz. No facility exists for extracting the state estimate from the state of health data during run-time.

## 9.3 Foreground Communications

Arbitrary data can be sent between spheres using a standard set of telemetry packing and queuing functions, though it is up to the guest scientist to provide code to interpret the data when they are received. Explicit transmission of data is termed foreground communication, and is achieved using the function `commSendPacket(…)` to send a single packet of data, or `datacommSendData(…)` to send data of arbitrary length. A set of packing functions is supplied to support conversion of various data types into an acceptable telemetry format. These packing functions will be detailed in a future release.

# 10 Initialization

## 10.1 Program initialization

During the initialization process after a program upload or reset, two initialization functions are called by the flight software. The function `gspIdentitySet(…)` is used only to set the logical (software) identity of the host satellite.

| **`gspIdentitySet(…)`** |
|---|
| A primary interface function used to set the logical SPHERE identity. This function should contain only a single call to `sysIdentitySet(…)`, and nothing else. |

The function `gspInitProgram(…)` must be used to set certain important parameters, as well perform any custom initialization.

| **`gspInitProgram(…)`** |
|---|
| A primary interface function used to initialize the program at startup. |

The functions called from `gspInitProgram(…)` specify the satellite logical identity number and other important properties of the satellite; this is therefore the only GSP-related function that must contain certain function calls. Specifically, the following functions must be called, in the order shown here:

- `commTdmaInit(COMM_CHANNEL_STL,…)`
- `commTdmaInit(COMM_CHANNEL_STS,…)`
- `padsInertialAllocateBuffers(…)`
- `padsInitializeFPGA(…)`
- `datacommTokenSetup(COMM_CHANNEL_STL,…)`
- `datacommTokenSetup(COMM_CHANNEL_STS,…)`

Note that the function `commTdmaStandardInit(…)` can be substituted for `commTdmaInit(…)`. The required initialization functions are demonstrated in the template version of `gsp.c`.

## 10.2 Test initialization

The primary interface function `gspInitTest(…)` is called whenever a new test is commanded through the laptop control station.

| **`gspInitTest(…)`** | | |
|---|---|---|
| A primary interface function that is called whenever a new test begins. | | |
| *1* | *`unsigned`* | (`test_number`) The new test number. |

This function should be used to initialize test-specific quantities, such as `padsInertialPeriodSet(…)` and `padsGlobalPeriodSet(…)`. Only one sphere should call `padsGlobalPeriodSet(…)`, in order to designate a PADS master. If multiple spheres call `padsGlobalPeriodSet(…)`, each one will periodically request global updates, possibly resulting in unexpected behavior.

# 11 SPHERES Simulation

The SPHERES simulation is designed to maximize the effectiveness of interactions between the Guest Scientist and the MIT SPHERES team. It is particularly valuable during early stages of algorithm development and implementation, as an aid in accelerating the learning curve for the GSP interface. The simulation environment is completely in MATLAB. Therefore, the simulation can be used for verifying the general desired behavior of a test, with multiple satellites. However, it cannot be used to verify compilation in C or integration with SPHERES C library functions. Detailed documentation on the SPHERES MATLAB simulation can be found at the following address: http://ssl.mit.edu/spheres/gsp/SpheresMatlabSimDocumentation.pdf.

**Note: the current version of the SPHERES MATLAB simulation is undergoing a major upgrade that will support a C-code interface to the GSP software. Updated documentation and software will be posted on the GSP website at http://ssl.mit.edu/gsp/.**

## 12 Expansion Port

Each sphere is equipped with an expansion port that may be used to interface with additional hardware. The SPHERES Expansion Port was upgraded to the SPHERES Expansion Port V2. Information about the SPHERES Expansion Port V2 can be found in the SPHERES Expansion Port V2 ICD.

## 13 Acknowledgements and References

The MIT SPHERES team would like to thank everyone who supplied feedback on the GSP interface design and on this document. In particular, Russell Carpenter of NASA GSFC and Arthur Richards of MIT provided valuable critiques of the interface at many stages in its development.

Data, text, and figures from several sources were used to prepare this document. Information came from the following sources:

1. BEI Systron Donner Inertial Division, "BEI GYROCHIP II micromachined angular rate sensor," http://www.systron.com/pdfs/GyroIIDS.pdf, 21 March 2002.
2. BEI Systron Donner Inertial Division.
3. Chen, Allen, "Propulsion system characterization for the SPHERES formation flight and docking testbed," Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2002.
4. Guerra, Edison, Payload Systems, Inc., personal contact.
5. Hilstad, Mark, "A Multi-Vehicle Testbed and Interface Framework for the Development and Verification of Separated Spacecraft Control Algorithms," Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2002.
6. Honeywell International Inc., Inertial Sensor Products Redmond, "Q-Flex QA-160/185 Accelerometer," http://www.inertialsensor.com/docs/qa-t160.pdf, 21 March 2002.
7. Honeywell International Inc., Inertial Sensor Products Redmond.
8. Miller, David W., et al., SPHERES critical design review, February 2002.
9. Murata Manufacturing Co., Ltd., MA40S4R online product specifications, Nov. 10, 2002, http://search.murata.co.jp/Ceramy/owa/CATALOG.showcatalog?sHinnmTmp=MA40S4R&sLang=2&sNhnm=MA40S4R&sHnTyp=NEW.
10. Saenz-Otero, Alvar, "The SPHERES satellite formation flight testbed: Design and initial control," Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, August 2000.
11. Sell, Steve, Payload Systems, Inc., personal communication.

## 14 Appendix A - Secondary interface API

This section contains descriptions of the secondary functions, those routines which may be called from Guest Scientist code, but containing source code which may not be directly modified, or even available. The secondary functions comprising the API are separated into several header files, grouped by subsystem or logical applicability. Most of these header files are included by default in the `gsp.c` template.

### 14.1 Header file comm.h

Prototypes for the following functions can be found in the file `comm.h`.

| **void commBackgroundPointerSet(…)** | | |
|---|---|---|
| Sets the background telemetry address to argument 1. All future background telemetry will be read from that address. | | |
| *1* | *state_vector** | The address of the state data to be sent through background telemetry. |

| **void commBackgroundPointerDefault()** |
|---|
| Sets the background telemetry address to its default value. All future background telemetry will be read from the internal state estimate. |

| **int commAssignChannel(…)** | | |
|---|---|---|
| Associates a logical channel identifier with a physical channel frequency. This function is experimental, and has not been fully tested. Returns zero for success, or one of the error codes `COMM_ILLEGAL_CHANNEL` or `COMM_ILLEGAL_FREQ` for failure. | | |
| *1* | *int* | Logical identifier: `COMM_CHANNEL_STS` or `COMM_CHANNEL_STL`. |
| *2* | *int* | Physical frequency: `COMM_FREQ_916MHZ` or `COMM_FREQ_868MHZ`. |

| **int commSendPacket(…)** | | |
|---|---|---|
| Places a single packet in a communications queue. Returns the sequential packet number if successful or `COMM_ERR_CANT_SEND` if the operation failed. | | |
| *1* | *int* | Channel: `COMM_CHANNEL_STL` or `COMM_CHANNEL_STS`. |
| *2* | *int* | To: `BROADCAST`, `GROUND`, `SPHERE1`, `SPHERE2`, `SPHERE3`, `SPHERE4`, or `SPHERE5`. |
| *3* | *int* | From: `GROUND`, `SPHERE1`, `SPHERE2`, `SPHERE3`, `SPHERE4`, or `SPHERE5`. |
| *4* | *int* | Command. Must be `COMM_CMD_GSP_PACKET`. |
| *5* | *default_rfm_payload* | A 32-byte memory space containing the packet data. |
| *6* | *unsigned* | Mode: `COMM_MODE_NEED_ACK` (for critical communications requiring a return receipt) or `COMM_MODE_NO_ACK` (for standard communications). |

| **int commTdmaInit(…)** | | |
|---|---|---|
| Configures the TDMA timing for this satellite. Arguments are specific to this particular satellite, for transmit and receive on the specified channel only. The window length (argument 3 minus argument 2) must be at least 20 ms, and the STL channel must reserve 40 ms at the end of the frame for the laptop. Note that the frame length (argument 4) is included for local information purposes; a subsequent frame will not actually begin until a frame synchronization packet is send by the laptop over the STL channel. This means that specifying an STS frame length shorter than the STL frame length will waste bandwidth. Returns 0 for success, or -1 for failure. | | |
| *1* | *int* | Channel: `COMM_CHANNEL_STL` or `COMM_CHANNEL_STS`. |
| *2* | *int* | Window start time [ms]. |
| *3* | *int* | Window end time [ms]. |
| *4* | *int* | Frame length [ms]. |

| **int commTdmaStandardInit(…)** | | |
|---|---|---|
| Initializes the TDMA timing using standard window assignments for a 200 ms TDMA frame. The standard TDMA frame assigns equal transmit time to each satellite. Returns 0 for success, or -1 for failure. | | |
| *1* | *int* | Channel: `COMM_CHANNEL_STL` or `COMM_CHANNEL_STS`. |

|   |   |   |
|---|---|---|
| *2* | *unsigned* | The logical identifier of the local satellite: SPHERE1, SPHERE2, SPHERE3, SPHERE4, or SPHERE5. |
| *3* | *unsigned* | Number of satellites: one through five, inclusive. Greater than five does not return an error, but does waste communications bandwidth. |

| **int commTdmaEnable(…)** |   |   |
|---|---|---|
| Enables transmission on the specified channel. Returns 0 for success, or -1 for failure. |   |   |
| *1* | *int* | Channel: COMM_CHANNEL_STL or COMM_CHANNEL_STS. |

| **int commTdmaDisable(…)** |   |   |
|---|---|---|
| Disables transmission on the specified channel. Returns 0 for success, or -1 for failure. |   |   |
| *1* | *int* | Channel: COMM_CHANNEL_STL or COMM_CHANNEL_STS. |

| **unsigned commTdmaIsEnabled(…)** |   |   |
|---|---|---|
| Returns TRUE if the specified channel is enabled, or FALSE if it is disabled. |   |   |
| *1* | *int* | Channel: COMM_CHANNEL_STL or COMM_CHANNEL_STS. |

| **int commBackgroundStateGet(…)** |   |   |
|---|---|---|
| Retrieves the most recently state telemetry for the specified satellite, as transferred through the background telemetry process. Returns 0 for success, or -1 for failure |   |   |
| *1* | *unsigned* | The logical identifier of the satellite for which state data are desired: SPHERE1, SPHERE2, SPHERE3, SPHERE4, or SPHERE5. |
| *2* | *unsigned** | The address into which the time stamp of the state data will be copied. |
| *3* | *state_vector** | The address of a state_vector into which the state data will be copied. |

| **int commBackgroundTelemetryPeriodSet(…)** |   |   |
|---|---|---|
| Sets the rate at which the background telemetry packets are sent to the communications queue. Returns 0 for success, or -1 for failure. |   |   |
| *1* | *unsigned* | Period: the time between background telemetry transmissions [ms]. Must be greater than 20. A value of SYS_FOREVER disables background telemetry. |

| **unsigned commBackgroundTelemetryPeriodGet()** |
|---|
| Returns the current background telemetry period, in milliseconds. A return value of SYS_FOREVER indicates that background telemetry is disabled. |

## 14.2 Header file comm_datacomm.h

Prototypes for the following functions can be found in the file comm_datacomm.h.

| **int datacommSendData(…)** |   |   |
|---|---|---|
| Sends data of arbitrary size through the STS or STL communication channel. The user must specify a tag, which is used by the recipient to determine how to parse the included data. More details to follow later…. This function is not fully tested. |   |   |
| *1* | *unsigned* | TBD |
| *2* | *unsigned char** | TBD |
| *3* | *unsigned* | TBD |
| *4* | *unsigned* | TBD |
| *5* | *unsigned* | TBD |

| | | |
|---|---|---|
| *6* | *unsigned* | TBD |
| *7* | *unsigned** | TBD |

| **`datacommTokenSetup(…)`** | | |
|---|---|---|
| Initializes flow control for data transfer, using a leaky bucket scheme. Suggested values for the arguments are given, based on the following definitions.<br>• N: number of satellites.<br>• C: TDMA frame time (default 200) [ms].<br>• K: Packet time (~12 for a standard packet) [ms].<br>• S: Fraction of TDMA cycle reserved for satellites.<br>• Q: Fraction of bandwidth reserved for datacomm transfers.<br>Suggested values are:<br>• 1 satellite: period = 25, tokens = 8.<br>• 2 satellites: period = 50, tokens = 8.<br>• 3 satellites: period = 75, tokens = 8. | | |
| *1* | *unsigned* | Channel: either `COMM_CHANNEL_STL` or `COMM_CHANNEL_STS`. |
| *2* | *unsigned* | Ideal period = N*K/(Q*S) |
| *3* | *unsigned* | Maximum number of stored tokens = C*S/(N*ideal period) |

## 14.3 Header file commands.h

This file does not contain any function prototypes. It is included primarily for the definitions of `COMM_CMD_GSP_PACKET` and `cmdpkt_payload_overlay`.

## 14.4 Header file control.h

Prototypes for the following functions can be found in the file `control.h`.

| **`unsigned ctrlPeriodSet(…)`** | | |
|---|---|---|
| Sets the period for the periodic control interrupt process. Returns the value of the period previously in use. | | |
| *1* | *unsigned* | The desired control period [ms]. A value of `SYS_FOREVER` disables the periodic control interrupt. |

| **`unsigned ctrlPeriodGet()`** |
|---|
| Returns the current control interrupt period, in milliseconds. A return value of `SYS_FOREVER` indicates that the periodic control interrupt is disabled. |

| **`ctrlTestTerminate(…)`** | | |
|---|---|---|
| Terminates the current test. Argument 1, sent to the laptop in the state of health packet, can be used as a return code to signal the state of success of the test. | | |
| *1* | *unsigned* | A return code that is sent to the laptop through the state of health packet. |

| **`ctrlManeuverTerminate()`** |
|---|
| Terminates the current maneuver and increments the maneuver number. Additional changes in the satellite behavior upon maneuver termination are strictly dependent on the Guest Scientist's implementation. |

| **`ctrlManeuverNumSet(…)`** | | |
|---|---|---|
| Terminates the current maneuver and sets the maneuver number to argument 1. Additional changes in the satellite behavior upon maneuver termination are strictly dependent on the Guest Scientist's implementation. | | |
| *1* | *unsigned* | Desired maneuver number. |

| **`unsigned ctrlManeuverNumGet()`** |
|---|
| Returns the current maneuver number. |

| **`int ctrlTestNumGet()`** |
|---|
| Returns the current test number. |

| **`unsigned ctrlTestTimeGet()`** |
|---|
| Returns the elapsed time since the beginning of the current test, in milliseconds. This timer is synchronized between the satellites to within one millisecond. |

| **`int ctrlManeuverTimeGet()`** |
|---|
| Returns the elapsed time for the current maneuver. |

| **`ctrlTestInfoGet(…)`** | | |
|---|---|---|
| Retrieves extended information about the test currently in progress. Useful for task and PADS-related functions. | | |
| *1* | *unsigned** | Test number. Equivalent to `ctrlTestNumGet()`. |
| *2* | *unsigned** | Elapsed test time. Equivalent to `ctrlTestTimeGet()`. |
| *3* | *unsigned** | Maneuver number. Equivalent to `ctrlManeuverNumGet()`. |
| *4* | *unsigned** | Elapsed maneuver time. Equivalent to `ctrlManeuverTimeGet()`. |

## 14.5 Header file gsp.h

The file `gsp.h` is available exclusively for customized use by the Guest Scientist. See the separate section describing this file for details.

## 14.6 Header file gsp_task.h

Prototypes for the following functions can be found in the file `gsp_task.h`. In addition to the following functions, the file contains definitions for task masks, as described in the section on the GSP task process.

| **`int taskTriggerMaskSet(…)`** | | |
|---|---|---|
| Sets the mask for the GSP task trigger. Multiple trigger values can be combined with the bitwise OR operator to create the mask, e.g. `PADS_GLOBAL_START_TRIG \| GSP_USER_TRIG`. Each time this function is called, the previous mask values are overwritten. Returns 0. | | |
| *1* | *unsigned* | The trigger mask for the GSP task process. |

| **`taskWaitForTime(…)`** | | |
|---|---|---|
| Triggers the task process after the number of milliseconds in argument 1 has transpired. | | |
| *1* | *unsigned* | Number of milliseconds before task trigger. |

| `taskPostUserTrigger(…)` | | |
|---|---|---|
| Explicitly posts a `GSP_USER_TRIG` event. The task must have been previously set up to trigger on this type of event, by using `taskTriggerMaskSet(…)`. | | |
| *1* | *unsigned* | Extra data to be passed to the task. |

## 14.7 Header file gsutil_pack_data.h

The functions in the file `gsutil_pack_data.h` will be described in a later release of this document.

## 14.8 Header file pads.h

Prototypes for the following functions can be found in the file `pads.h`.

| `int padsInertialAllocateBuffers(…)` | | |
|---|---|---|
| Allocates temporary storage for the rate gyroscope and accelerometer readings. This function must be called exactly once in every program that uses the inertial sensors, in the `gspInitProgram()` function. Since the kernel must store at least processing_period/sampling_period sets of samples, argument 1 should be set to the maximum ratio expected during a program's execution. Warning: this function MUST be called before `padsInertialPeriodSet(…)`. | | |
| *1* | *unsigned* | The number of samples for which storage space should be allocated. |

| `int padsInertialPeriodSet(…)` | | |
|---|---|---|
| Sets up the sampling behavior for the inertial sensors (rate gyroscopes and accelerometers). The sensors are sampled and the results recorded after the number of milliseconds specified by argument 1. The processing function `gspPadsInertial(…)` is called with the period specified by argument 2. This function can be called at any time, but calling it during a test discards all samples collected since the last processing call. Warning: the function `padsInertialAllocateBuffers(…)` must be called prior to using this function. | | |
| *1* | *unsigned* | Sampling period. |
| *2* | *unsigned* | Processing period. |

| `padsInitializeFPGA(…)` | | |
|---|---|---|
| Initializes the PADS avionics hardware. It must be called in the `gspInitProgram()` function, and may be called elsewhere as well. | | |
| *1* | *unsigned* | Highest beacon number in use. |

| `padsGlobalPeriodSet(…)` | | |
|---|---|---|
| Sets the global update period. The global period should be set to zero for all except one satellite. | | |
| *1* | *unsigned* | The number of milliseconds between periodic infrared flashes. A value of `SYS_FOREVER` disables periodic global updates. |

| `unsigned padsGlobalPeriodGet(…)` |
|---|
| Returns the number of milliseconds between the periodic infrared flashes used for global updates. A return value of `SYS_FOREVER` indicates that periodic global updates are disabled. |

| `padsGlobalTriggerNow()` |
|---|
| Explicitly initiates an infrared flash, triggering an immediate global update. |

| **`unsigned padsStateGet(…)`** | | |
|---|---|---|
| Retrieves the internal state estimate, and returns the time stamp of the estimate. | | |
| *1* | *`state_vector`* | A destination into which the state estimate will be copied. |

| **`padsBeaconNumberSet(…)`** | | |
|---|---|---|
| Sets the onboard beacon number to the value of argument 1. Warning: make sure that the beacon number does not conflict with that of a fixed beacon or another satellite. | | |
| *1* | *`unsigned`* | Beacon number for the onboard beacon. A value of 0 disables the beacon. |

| **`int padsBeaconLocationSet(…)`** | | |
|---|---|---|
| Sets the location and pointing direction of a fixed beacon, in the global frame. Returns 0 for success, or -1 for failure. | | |
| *1* | *`unsigned`* | The beacon number. |
| *2* | *`float[3]`* | The beacon position. |
| *3* | *`float[3]`* | The beacon pointing direction. |

| **`int padsBeaconLocationGet(…)`** | | |
|---|---|---|
| Retrieves the location and pointing direction of a fixed beacon, in the global frame. Returns 0 for success, or -1 for failure. | | |
| *1* | *`unsigned`* | The beacon number. |
| *2* | *`float[3]`* | The beacon position. |
| *3* | *`float[3]`* | The beacon pointing direction. |

| **`int padsTemperatureSet(…)`** | | |
|---|---|---|
| Sets the temperature used in calculating the speed of sound for use in global updates. Returns 0. | | |
| *1* | *`float`* | The temperature, in degrees Celsius. |

| **`float padsTemperatureGet()`** |
|---|
| Returns the currently saved temperature, in degrees Celsius. |

## 14.9 Header file pads_correct.h

Prototypes for the following functions can be found in the file `pads_correct.h`.

| **`padsCountsConvertGyros(…)`** | | |
|---|---|---|
| Converts raw rate gyroscope readings from counts to radians per second, using the currently stored conversion coefficients. | | |
| *1* | *`unsigned*`* | Pointer to the source reading in counts. |
| *2* | *`float*`* | Pointer to the destination reading in rad/s. |

| **`padsCountsConvertAccels(…)`** | | |
|---|---|---|
| Converts raw accelerometer readings from counts to meters per second squared, using the currently stored conversion coefficients. | | |
| *1* | *`unsigned*`* | Pointer to the source reading in counts. |
| *2* | *`float*`* | Pointer to the destination reading in $m/s^2$ |

| **`padsCountsCoefficientsSet(…)`** | | |
|---|---|---|
| Changes the bias and scale coefficients used for conversion of inertial measurements from counts to metric units. | | |
| *1* | *`float*`* | Rate gyro bias [count]. |

| | | |
|---|---|---|
| 2 | `float*` | Rate gyro scale [rad/s/count]. |
| 3 | `float*` | Accelerometer bias [count]. |
| 4 | `float*` | Accelerometer scale [m/s$^2$/count]. |

| **`padsCountsCoefficientsGet(…)`** | | |
|---|---|---|
| Retrieves the bias and scale coefficients used for conversion of inertial measurements from counts to metric units. The initial conditions for the coefficients are burned into the SPHERE flash memory. | | |
| 1 | `float*` | Rate gyro bias [count]. |
| 2 | `float*` | Rate gyro scale [rad/s/count]. |
| 3 | `float*` | Accelerometer bias [count]. |
| 4 | `float*` | Accelerometer scale [m/s$^2$/count]. |

## 14.10 Header file prop.h

Prototypes for the following functions can be found in the file `prop.h`.

| **`propSetThrusterTimes(…)`** | | |
|---|---|---|
| Specifies to the propulsion system when to open and close each thruster valve. The on and off times in argument 1 are interpreted as offsets from the current time. | | |
| 1 | `prop_time*` | A structure containing the on and off times for the thrusters [ms]. |

| **`unsigned short propGetThrusters()`** |
|---|
| Returns the current (commanded) on/off state of the thrusters. Each of the first twelve bits of the return value represents a thruster. A value of 1 is on; a value of 0 is off. |

## 14.11 Header file spheres_constants.h

The header file `spheres_constants.h` contains definitions for constants used in the SPHERES software. These constants should be used instead of defining new constants, whenever possible, to aid in readability and interoperability of the code.

## 14.12 Header file spheres_physical_parameters.h

Prototypes for the following functions can be found in the header file `spheres_physical_parameters.h`.

| **`sysPhysicalDefaultsSet()`** |
|---|
| Resets the physical parameters of the satellite (mass, inertia, etc) to their default values. |

| **`float sysMassGet()`** |
|---|
| Returns the mass of the satellite [kg]. |

| **`sysMassSet(…)`** | | |
|---|---|---|
| Sets the mass of the satellite. | | |
| 1 | `float` | The mass of the satellite [kg]. |

| **`sysInertiaGet(…)`** | | |
|---|---|---|
| Retrieves the inertia of the satellite. | | |
| 1 | `float[3][3]` | The inertia matrix of the satellite. |

| **sysInertiaSet(…)** | | |
|---|---|---|
| Sets the inertia of the satellite. | | |
| *1* | *float[3][3]* | The inertia matrix of the satellite. |

| **sysInertiaInverseGet(…)** | | |
|---|---|---|
| Retrieves the inverse of the inertia of the satellite. | | |
| *1* | *float[3][3]* | The inverse of the inertia matrix of the satellite. |

| **sysInertiaInverseSet(…)** | | |
|---|---|---|
| Sets the inverse of the inertia of the satellite. | | |
| *1* | *float[3][3]* | The inverse of the inertia matrix of the satellite. |

## 14.13 Header file spheres_types.h

The header file `spheres_types.h` defines several standard types used throughout the SPHERES code, including the data type used for the state estimate.

## 14.14 Header file std_includes.h

The header file `std_includes.h` includes several files that are specific to either flight or simulation builds. The functions in these files are unlikely to be directly useful to guest scientists.

## 14.15 Header file system.h

Prototypes for the following functions can be found in the file `system.h`. See also the file `spheres_physical_parameters.h` for additional system functions.

| **sysIdentitySet(…)** | | |
|---|---|---|
| Sets the logical identity of the local satellite. Note that this does not change the hardware (communications) address of the satellite. This function *must* be called in the primary interface function `gspIdentitySet(…)`. | | |
| *1* | *unsigned* | Logical identity of the satellite: `SPHERE1`, `SPHERE2`, `SPHERE3`, `SPHERE4`, or `SPHERE5`. |

| **unsigned sysIdentityGet()** |
|---|
| Returns the logical identity of this satellite. One of `SPHERE1`, `SPHERE2`, `SPHERE3`, `SPHERE4`, or `SPHERE5`. |

| **unsigned sysSphereTimeGet()** |
|---|
| Returns the elapsed time since the last hardware reset, in milliseconds. Unlikely to be useful, but available nonetheless. Likely to be more useful is `ctrlTestTimeGet()`. |

## 14.16 Header file util_memory.h

Prototypes for the following functions can be found in the file `util_memory.h`.

| **`atomic_memcpy(…)`** | | |
|---|---|---|
| Copies the number of bytes specified in argument 3 from the address specified by argument 2 to the address specified by argument 1. Identical in function to the standard C routine memcpy(…), but modified so that it performs the copy procedure atomically with respect to all onboard software processes. For this reason, this function should always be used instead of memcpy(…). | | |
| *1* | *`void*`* | Address of destination memory space. |
| *2* | *`void*`* | Address of source memory space. |
| *3* | *`unsigned`* | Number of bytes to copy. |

## 14.17 Header file util_serial_printf.h

Prototypes for the following functions can be found in the file `util_serial_printf.h`. Note that printing data to the screen is inherently slow, so using any of the following functions can significantly slow the progress of the simulation.

| **`ser_print(…)`** | | |
|---|---|---|
| Prints a null-terminated character string to the simulation client message window. Suggested use is to first format the string using `sprintf(…)`. | | |
| *1* | *`char*`* | A string to be displayed in the simulation client message window. |

| **`ser_nprint(…)`** | | |
|---|---|---|
| Prints a specified number of characters to the simulation client message window. | | |
| *1* | *`char*`* | The address of the character buffer. |
| *2* | *`int`* | The number of characters to print. |

| **`print_int(…)`** | | |
|---|---|---|
| Prints a single integer, formatted in hexadecimal, to the simulation client message window. | | |
| *1* | *`unsigned`* | The unsigned integer to be displayed. |

# 15 Appendix B – Standard Utilities

This section describes the source files included with the GSP package. Included in these files are the command, control, estimation, mixer, and terminator functions mentioned previously.

## 15.1 Commands

No command functions are included with this release of the GSP package. A function is provided, however, to determine the state error given the current state and the current desired state. This function, `find_state_error(…)`, can be found in the files `find_state_error.h` and `find_state_error.c`.

| **`find_state_error(…)`** | | |
|---|---|---|
| Determines the state error, given the current state and desired state. | | |
| *1* | *`float*`* | Resultant state error, of type `state_vector` or `state_vector_extended`. |
| *3* | *`float*`* | Current state estimate, of type `state_vector` or `state_vector_extended` |
| *4* | *`float*`* | Current desired state, of type `state_vector` or `state_vector_extended`. |

## 15.2 Control

The standard control algorithms provided with the GSP package control either position and velocity or quaternion and rate. In order to have full position and attitude control, one position and one attitude control function must be used. These functions can be found in the directory `standard\control`.

### 15.2.1 Position control

Detailed descriptions and source code for the algorithms implemented in the following functions can be found in the files `ctrl_attitude.h` and `ctrl_attitude.c`.

| **`ctrlPositionPD(…)`** | | |
|---|---|---|
| Applies a proportional/derivative control law to the position and velocity errors, to produce force commands in the global frame. | | |
| *1* | *`float`* | Position gain. |
| *2* | *`float`* | Velocity gain. |
| *3* | *`float*`* | State error, of type `state_vector` or `state_vector_extended`. |
| *4* | *`float*`* | Resultant command, of type `control_vector`. |

| **`ctrlPositionPDgains(…)`** | | |
|---|---|---|
| Applies a proportional/integral/derivative control law to the position and velocity errors, to produce force commands in the global frame. | | |
| *1* | *`float`* | Position gain, x direction. |
| *2* | *`float`* | Velocity gain, x direction. |
| *3* | *`float`* | Position gain, y direction. |
| *4* | *`float`* | Velocity gain, y direction. |
| *5* | *`float`* | Position gain, z direction. |
| *6* | *`float`* | Velocity gain, z direction. |
| *7* | *`float*`* | State error, of type `state_vector` or `state_vector_extended`. |
| *8* | *`float*`* | Resultant command, of type `control_vector`. |

| **`ctrlPositionPID(…)`** | | |
|---|---|---|
| Applies a proportional/derivative control law to the position and velocity errors, to produce force commands in the global frame. | | |
| *1* | *`float`* | Position gain. |
| *2* | *`float`* | Integral gain. |
| *3* | *`float`* | Velocity gain. |
| *4* | *`float`* | Thruster firing period, in seconds. |
| *5* | *`float*`* | State error, of type `state_vector` or `state_vector_extended`. |
| *6* | *`float*`* | Resultant command, of type `control_vector`. |

| **`ctrlPositionPIDgains(…)`** | | |
|---|---|---|
| Applies a proportional/integral/derivative control law to the position and velocity errors, to produce force commands in the global frame. | | |
| *1* | *`float`* | Position gain, x direction. |
| *2* | *`float`* | Integral gain, x direction. |
| *3* | *`float`* | Velocity gain, x direction. |
| *4* | *`float`* | Position gain, y direction. |
| *5* | *`float`* | Integral gain, y direction. |
| *6* | *`float`* | Velocity gain, y direction. |

| | | |
|---|---|---|
| *7* | *float* | Position gain, z direction. |
| *8* | *float* | Integral gain, z direction. |
| *9* | *float* | Velocity gain, z direction. |
| *10* | *float* | Thruster firing period, in seconds. |
| *11* | *float** | State error, of type `state_vector` or `state_vector_extended`. |
| *12* | *float** | Resultant command, of type `control_vector`. |

### 15.2.2 Attitude control

References, detailed descriptions, and source code for the attitude control algorithms implemented in the following functions can be found in the files `ctrl_attitude.h` and `ctrl_attitude.c`.

| **ctrlAttitudeNLPDsidi(…)** | | |
|---|---|---|
| Applies a proportional/derivative control law to the position and velocity errors, to produce force commands in the global frame. Note: algorithm seems to work only for small error angles. | | |
| *1* | *float* | Angle gain, body x axis. |
| *2* | *float* | Rate gain, body x axis. |
| *3* | *float* | Angle gain, body y axis. |
| *4* | *float* | Rate gain, body y axis. |
| *5* | *float* | Angle gain, body z axis. |
| *6* | *float* | Rate gain, body z axis. |
| *7* | *float** | State error, of type `state_vector` or `state_vector_extended`. |
| *8* | *float** | Resultant command, of type `control_vector`. |

| **ctrlAttitudeNLPDwie(…)** | | |
|---|---|---|
| Applies a proportional/derivative control law to the position and velocity errors, to produce force commands in the global frame. | | |
| *1* | *float* | Angle gain, body x axis. |
| *2* | *float* | Rate gain, body x axis. |
| *3* | *float* | Angle gain, body y axis. |
| *4* | *float* | Rate gain, body y axis. |
| *5* | *float* | Angle gain, body z axis. |
| *6* | *float* | Rate gain, body z axis. |
| *7* | *float** | State error, of type `state_vector` or `state_vector_extended`. |
| *8* | *float** | Resultant command, of type `control_vector`. |

| **ctrlAttitudeNLPIDwie(…)** | | |
|---|---|---|
| Applies a proportional/integral/derivative control law to the position and velocity errors, to produce force commands in the global frame. | | |
| *1* | *float* | Angle gain, body x axis. |
| *2* | *float* | Integral gain, body x axis. |
| *3* | *float* | Rate gain, body x axis. |
| *4* | *float* | Angle gain, body y axis. |
| *5* | *float* | Integral gain, body y axis. |
| *6* | *float* | Rate gain, body y axis. |
| *7* | *float* | Angle gain, body z axis. |
| *8* | *float* | Integral gain, body z axis. |
| *9* | *float* | Rate gain, body z axis. |
| *10* | *float* | Thruster firing period, in seconds. |

| | | |
|---|---|---|
| *11* | *float** | State error, of type `state_vector` or `state_vector_extended`. |
| *12* | *float** | Resultant command, of type `control_vector`. |

## 15.3 Estimation

No estimation algorithms are included in this release of the GSP package.

## 15.4 Mixers

Detailed descriptions and source code for the following mixer functions can be found in the files `ctrl_mix.h` and `ctrl_mix.c`.

| **ctrlMixSimple(…)** | | |
|---|---|---|
| Determines and sets thruster on and off times based on force and torque commands and pulse width modulation parameters. Force may be specifed | | |
| *1* | *float** | Force and torque command, of type `control_vector`. |
| *2* | *float** | State estimate, of type `state_vector` or `state_vector_extended`. |
| *6* | *unsigned* | Frame in which forces are measured: `FORCE_FRAME_INERTIAL` or `FORCE_FRAME_BODY`. |
| *3* | *unsigned* | Minimum allowable pulse width [ms]. |
| *4* | *unsigned* | Maximum allowable pulse width [ms]. |
| *5* | *unsigned* | Delay multiplier. Number of control periods to delay actuation. |

## 15.5 Terminators

Detailed descriptions and source code for the following terminator functions can be found in `ctrl_terminate.h` and `ctrl_terminate.c`.

| **int ctrlTerminateManeuverTimed()** | | |
|---|---|---|
| Ends the current maneuver when the clock in argument 1 reaches or surpasses the time specified in argument 2. Returns 1 if the maneuver was terminated or 0 if it was not. | | |
| *1* | *unsigned* | The reference clock, in milliseconds. |
| *2* | *unsigned* | The desired termination time, in milliseconds. |

| **int ctrlTerminateTestTimed(…)** | | |
|---|---|---|
| Ends the current test when the clock in argument 1 reaches or surpasses the time specified in argument 2. Returns 1 if the test was terminated or 0 if it was not. | | |
| *1* | *unsigned* | The reference clock, in milliseconds. |
| *2* | *unsigned* | The desired termination time, in milliseconds. |
| *3* | *unsigned* | The test result return code, as explained for `ctrlTestTerminate(…)`. |

# 16 Appendix C –Change Log

The following is a summary of the changes made to each release version of the GSP package.

### 2009-04-23 (v2.1)

Updated references to simulation to include note about upcoming upgrades. Added a link to the expansion port ICD.

## 2009-04-23 (v2.0)

This release is primarily to remove the previous C simulation and replace it with working Matlab simulation. The C simulation was not fully functional and could not continue to be supported. Thus, the Matlab very has been tested to be functional and will now on be the supported version of the simulation by the SPHERES team.

## 2003-10-29 (v1.2)

This release contains several major and minor bug fixes, in addition to additional functionality and increased simulation fidelity.

API changes:

- Added a new primary interface function `gspIdentitySet(…)`, which contains only a single call to the function `sysIdentitySet(…)`. Removed the call to `sysIdentitySet(…)` in `gspInitProgram(…)`. Note: all existing copies of `gsp.c` must be modified by adding `gspIdentitySet(…)`.
- The defined quantities `TASK_WAIT_FOREVER` and `PADS_GLOBAL_DISABLE` have been deprecated, and will be removed in a future release. Use `SYS_FOREVER` instead.

Bug fixes:

- Fixed initialization of thruster torque; thruster 0 previously worked properly, but thrusters 1-11 were incorrectly initialized, leading to random torque.
- Fixed `ctrlManeuverTerminate(…)`; it previously caused a random maneuver number, rather than the next maneuver number, to begin.
- Fixed range data units, now consistently reported in [m]; speed of sound was previously initialized in [cm/s], resulting in range data in [cm].
- Fixed test termination behavior. Ending a test with `ctrlTestTerminate(…)` now stops both the simulation and the test, rather than just the test.
- Fixed memory initialization: powering off and on a sphere client closes and reopens the executable, guaranteeing that the memory state is initialized correctly.
- Fixed onboard beacon: `padsBeaconNumberSet(…)` no longer crashes the simulation.
- Fixed `padsGlobalTriggerNow(…)`; it was not triggering global updates.
- TDMA slot conflicts are reported by the server.
- STL and STS communications percentages on the server display are more accurate.
- Selecting and deselecting "Check for updates on start-up" multiple times no longer prints garbage data to the messages window.
- Removed extraneous zeros added to SPHERE5 telemetry by `spheres_data_convert.m`.

Simulation:

- Modified server/client communications behavior to support changing the software ID of a satellite during a test.
- Opening the GSP interface document from within the simulation now loads the most recent version from the MIT SSL web site.
- Added menu items to access online bug status and HOWTOs from the simulation server.

Documentation:

- Added description of new primary interface function `gspIdentitySet(…)`. Merged document and simulation version numbers.

## 2003-09-26 (document v1.1, simulation v1.0)

This is the first complete release of the GSP package.

Simulation:

- The initial public release of the simulation.

Document:

- This is a major update to the document, to accompany the initial public release of the SPHERES GSP simulation. Due to a CPU and operating system upgrade, the capabilities of the testbed, from a software implementation perspective, have increased dramatically since the release of v1.0 of the interface document. The large number of changes in this document reflects these new capabilities. Because this is the first release of the document that is accompanied by source code and the simulation, and because the changes to the interface are so pervasive, individual changes from v1.0 are not listed here.

## 2002-11-20 (document v1.0)

The initial public release of the GSP interface document.