Appendix G

SPHERES SOFTWARE DESIGN

This appendix presents the software design of the SPHERES satellites. The software developed by the SPHERES team consists of two main elements: the boot loader program and SPHERES Core. Figure G.1 illustrates the four pieces of software which operate in a satellite: the Texas Instruments boot firmware readies the DSP for operations; the Sundance custom firmware initializes the interfaces of the SMT375 board with the SPHERES avionics; the SPHERES boot loader allows to reprogram the operating system and loads the program into memory; the SPHERES Core Software (SCS) is the operating system which runs the satellites during normal operations.

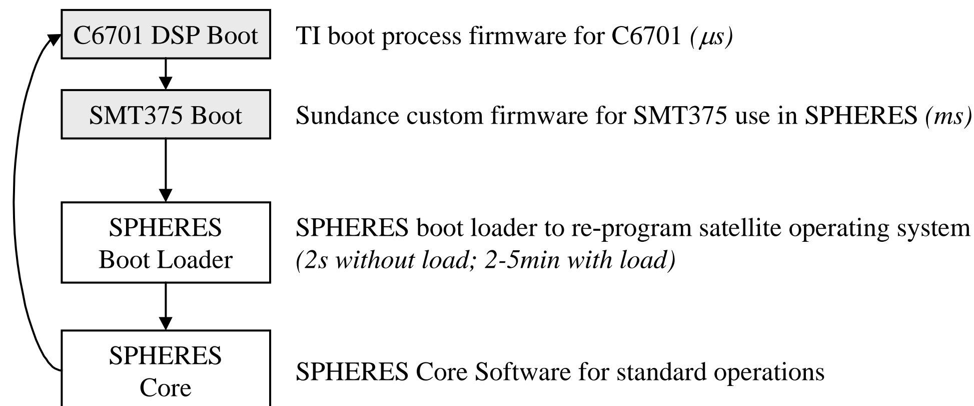

Figure G.1 Satellite software components

This appendix is divided into three main sections: the boot loader, the SPHERES Core Software operating system design, and the Standard Support Libraries.

G.1 Boot Loader

Figure G.2 shows the program development process for SPHERES. The SCS is programmed in assembly, standard ANSI C, or C++ using the Texas Instruments Code Composer Interface [TI, SPRU328B]. After compilation, the executable binary is translated into a text file and organized to enable storage in FLASH memory with `maketext` (this process is incorporated into the ground-based interface). This file is then transferred to the satellite via wireless communications. The boot loader reads the file from FLASH and loads it into RAM for operations. For ISS operations the file is packaged in a compressed file format for delivery to the ISS interface, which then decompresses the file and transmits it to the satellite in the same way that the ground-based interface does.

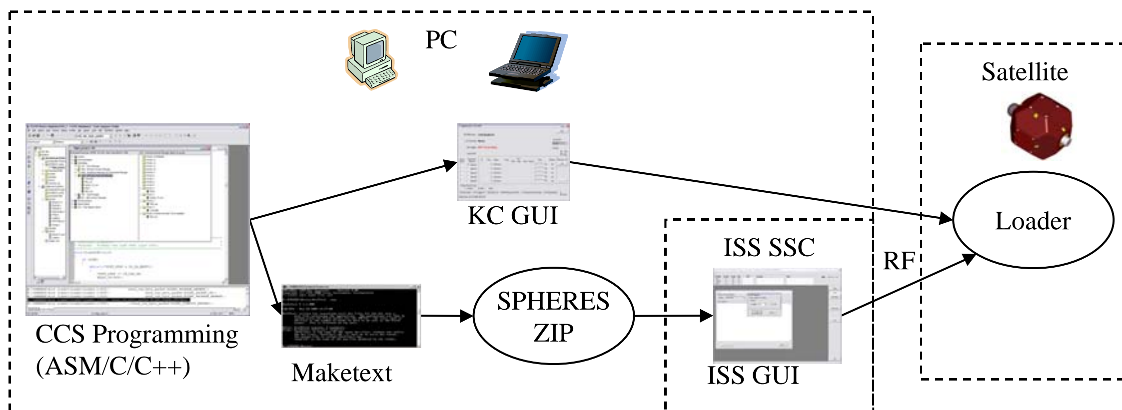

Figure G.2 SPHERES program development sequence

The TI Code Composer (CCS) interface assembles, compiles, and links the source code of the scientist and the DSP/BIOS kernel. The output from CCS is normally loaded directly into the RAM of a DSP system, therefore it requires some modifications for storage in FLASH. The function `maketext` utilizes two programs to enable transfer of the files to the satellites and their storage in FLASH. First, the TI provided program `hex6x.exe` is

used to convert the RAM ready output to hexadecimal format for storage in FLASH. This process involves separating the program into several sections and identifying the intended destination and size of those sections in RAM during operations. Next, the Sundance provided `hex2text.exe` is used to convert the binary hexadecimal into ASCII an text file. The text file is used in the wireless data transfer. The transferred file is converted to 32-bit words by the software on the satellites and stored in FLASH. When ready to run, the boot loader program on the satellites reads the sections written in FLASH and copies them to RAM.

G.1.1 Boot Loader Transfer Protocol

To balance the need for minimum transfer time together with the requirement for no data errors in the stored program, the boot loader uses packages of 20 packets to transfer the data. Each package must be acknowledged as received correctly or with a request to repeat the packet. The need to only acknowledge reception every 20 packets reduced the program upload time by both reducing the amount of data transferred and not requiring as many changes between transmission and reception in a unit.

Figure G.3 illustrates the boot loader data transfer protocol. The first step of both the PC and the satellite programs is to configure their DR200x transceivers (See Appendix H). Because programs are intended for specific satellites, the boot loader master program (on the PC) configures the DR200x to transmit only to a specific satellite; all other satellites will ignore the program. The satellite boot loader configures the DR200x for general operations. Table G.1 shows the boot-time configurations of the DR200x.

After initialization the master programs transmits a "start" packet to indicate to the satellite that data is available. It will transmit this packet at 1Hz until it receives an acknowledgement from the satellite. The satellite boot process is described below.

After the acknowledgement is received, the master program transmits the first packet individually, and awaits for a response. This packet is special because it includes the total pro-

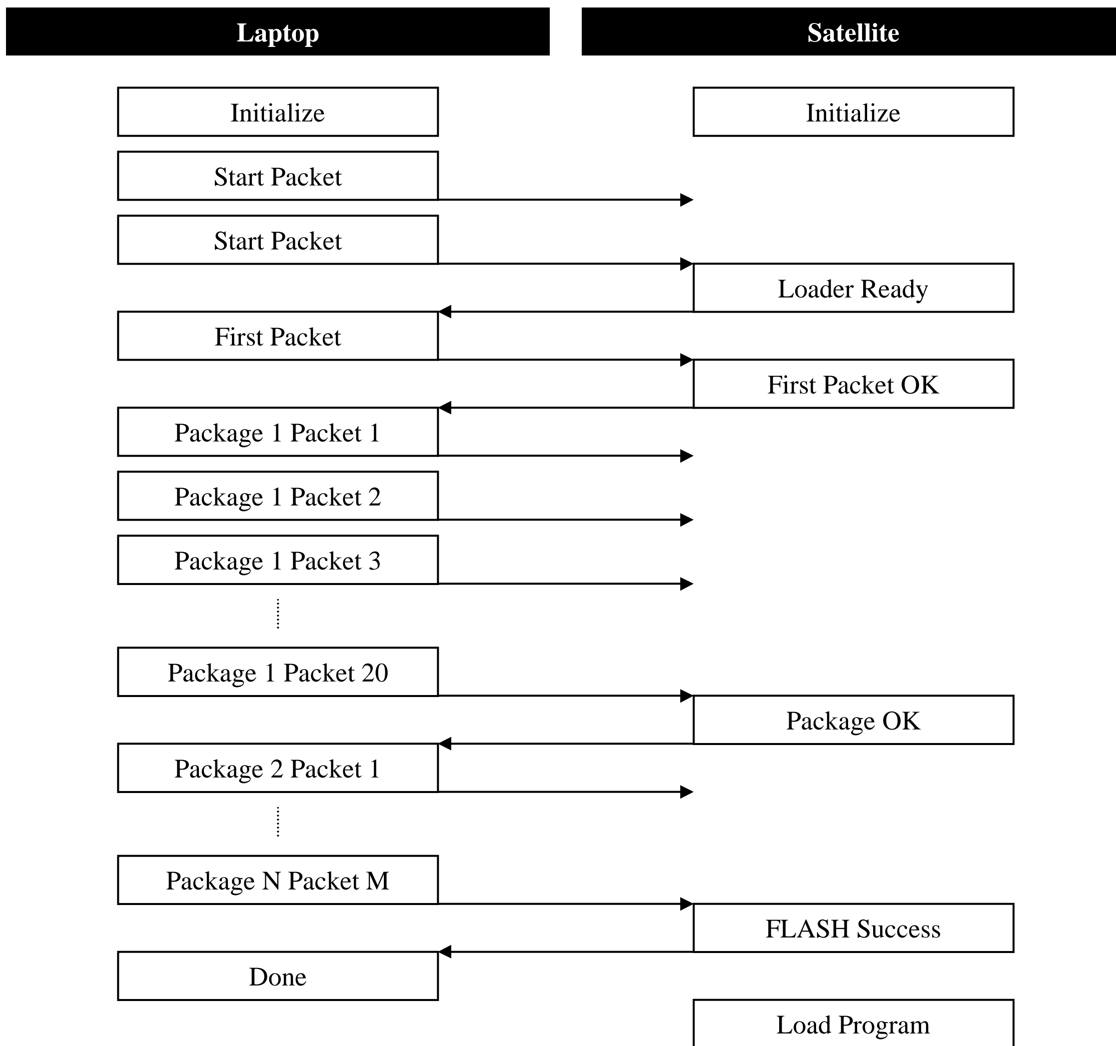

Figure G.3 Boot loader transfer protocol

TABLE G.1 DR200x configurations for boot load process

	PC	Satellite
TO	Satellite ID	0x00
FROM	0x30	Satellite ID
MODE	ASK	
RF data rate	56.6kpbs	
Packet size	6 then 75	6

gram size, that is, the amount of data to be transferred. The satellite will use this value to determine when it has received the full program, before overwriting the current program in FLASH. After the first packet has been acknowledged, the master program starts to transmit packages of 20 packets each. If a packet is not received correctly, the satellite requests a repeat, as illustrated in Figure G.4. This can be due to a data error (e.g. checksum error) or a timeout. Similarly, if the master does not sees a response after a timeout period, it transmits the packet again. Once all data has been received and confirmed with the checksums, the satellites unlocks the FLASH write function and overwrites the old program with the new one.

The boot loading process uses three types of packets: command/reply packets, first packet, and general packets. All packets contain a standard five byte header which indicates the destination (to), origin (from), packet number (pkt), command (cmd), and data size (len). The command/reply packets (Figure G.5, Table G.2) have one single byte of data, the command. The data packets (first and general) follow the header with a package number (pkg) and a packet number (pkt). The first packet (Figure G.6, Table G.3) has a 32-bit integer which indicates the total size of the program (in 32-bit words), followed by 15 32-bit words. General packets (Figure G.7, Table G.4) contain 16 program words (64 bytes). Data packets end with a 32-bit CRC (4 bytes).

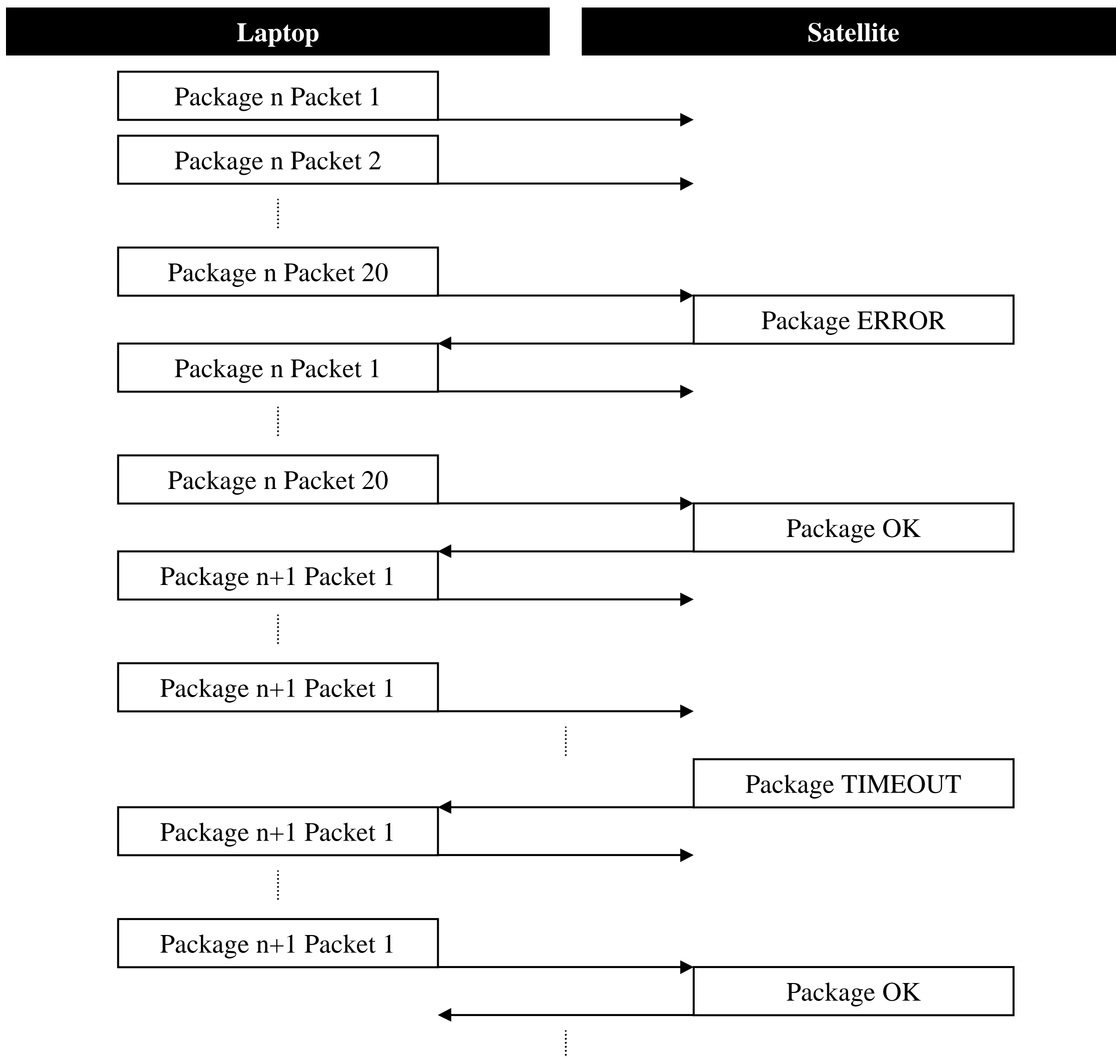

Figure G.4 Boot loader transfer protocol error handling

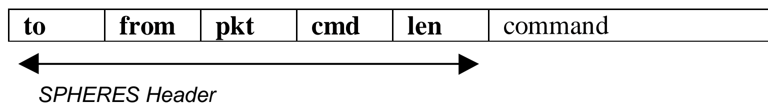

Figure G.5 Boot loader command/reply packets

TABLE G.2 Boot loader command/reply packet structure.

Byte	Width	Name	Function
1	1	to	The intended recipient.
2	1	from	This satellite's ID.
3	1	pkt	Packet counter byte, should change with every packet.
4	1	cmd	Command byte shared with SCS, therefore it **must** be 0x7F
5	1	len	The length of the data in the packet = 0x01

6	1	command	From satellite to laptop:
			`FL2EX_READY`
			`FL2EX_NO_PROGRAM`
			`FL2EX_INVALID_COMMAND`
			`FL2EX_PACKAGE_REPEAT`
			`FL2EX_FLASH_SUCCESS`
			`FL2EX_PACKET_OK`
			
			From laptop to satellite:
			`EX2FL_PROGRAM`
			`EX2FL_RUN`

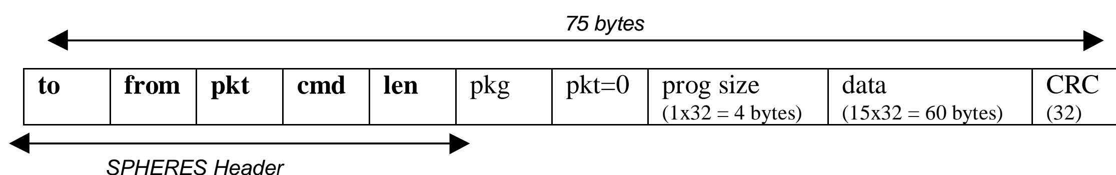

Figure G.6 Boot loader first data packet structure

TABLE G.3 Boot loader first packet structure

Byte	Width	Name	Function
1	1	to	The intended recipient
2	1	from	This satellite's ID
3	1	pkt	Packet counter byte
4	1	cmd	Command byte (0x7F)
5	1	len	The length of the data in the packet = 0x46

TABLE G.3 Boot loader first packet structure

Byte	Width	Name	Function
6	1	pkg	Package number
7	1	pkt	Packet number (1-20)
8-11	4	prog size	Total program size (in 32-bit words)
12-72	60	data	15 program words
72-75	4	crc	4 byte checksum

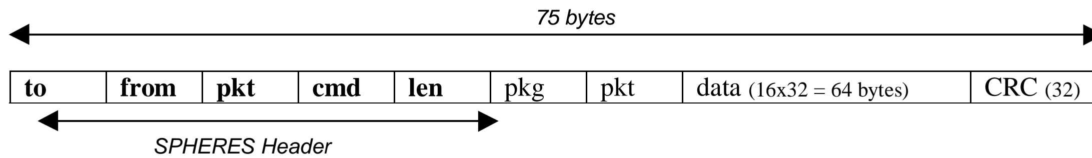

Figure G.7 Boot loader general data packets structure

TABLE G.4 Boot loader general packet structure.

Byte	Width	Name	Function
1	1	to	The intended recipient
2	1	from	This satellite's ID
3	1	pkt	Packet counter byte
4	1	cmd	Command byte (0x7F)
5	1	len	The length of the data in the packet = 0x46
6	1	pkg	Package number
7	1	pkt	Packet number (1-20)
8-71	64	data	16 program words
72-75	4	crc	4 byte checksum

G.1.2 Master

The implementation of the master program which runs on the PC to upload the programs has four main states as shown in Figure G.8. After initializing the DR200x as described above, the program enters idle mode and waits for a reply from a satellite. The actions of the master program in each state are as follows:

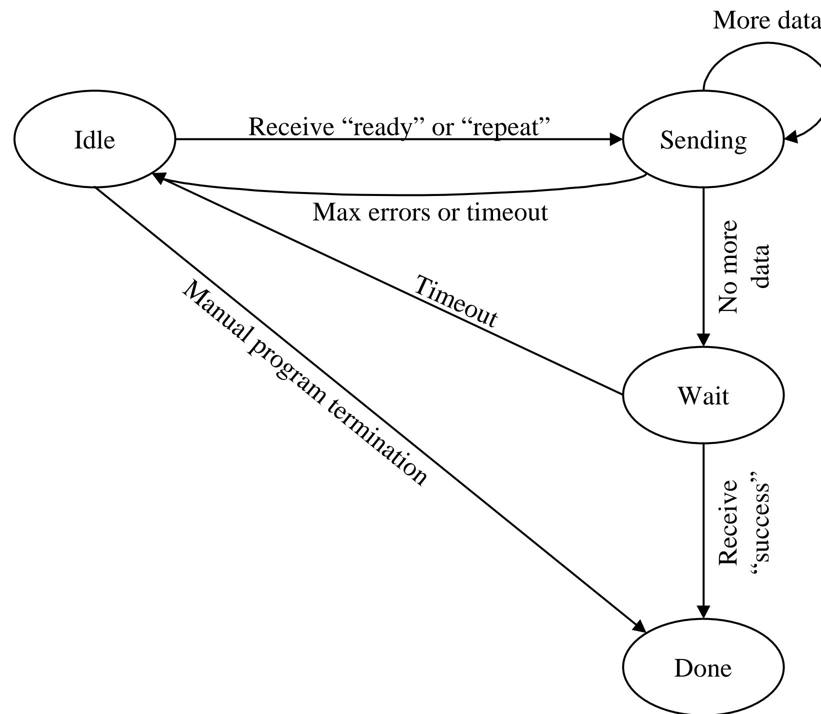

Figure G.8 Master program state diagram

- **Idle** - sends the `EX2FL_PROGRAM` command at 1Hz to the satellite until it gets a response or the program is terminated manually. If it receives the `FL2EX_READY` response it starts sending the program.
- **Sending** - sends packets until it is done or a pre-specified number of errors is reached (data errors or time outs). It transmit the first packet only, and awaits a response for that one packet the first time. Thereafter it transmits (or re-transmits) packages of 20 packets. When the last packet has been acknowledged it switches to the *wait* state. If the maximum number of error is reached it returns to *idle*.
- **Wait** - awaits a `FL2EX_SUCCESS` packet because all the program has been transferred. If the response is not received after a timeout it returns to *idle* to start again, since a failure in the burning the FLASH is assumed.
- **Done** - returns the DR200x configuration to normal operations (broadcast mode, 32-byte data) and exits.

G.1.3 Loader

The boot loader program on the satellites constitutes the only critical software element in the SPHERES project. Therefore, it has been programmed with minimal elements to reduce the sources of errors. Further, it uses some redundancy and a fallback mechanism to return the unit to operational conditions in case of a critical failure in the SCS. Figure G.9 presents the state diagram of the boot loader program. The states are explained below.

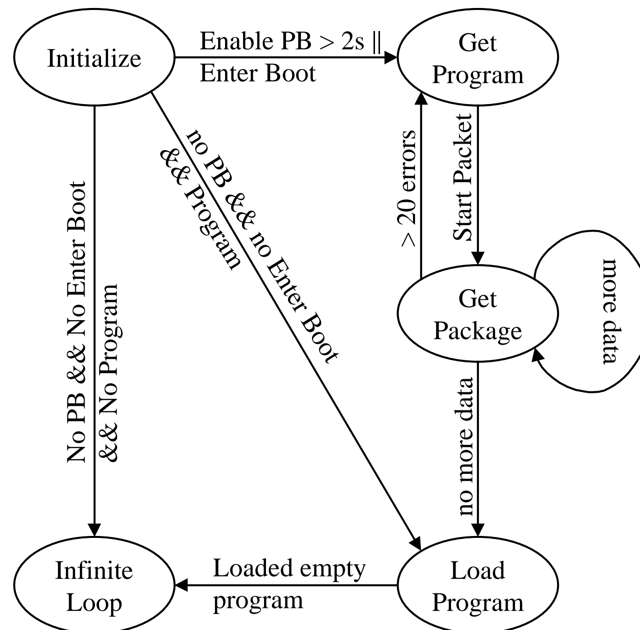

Figure G.9 Satellite boot loader state diagram

- **Initialize** - upon boot the loader initializes its hardware timers, reads the satellite identity and other information from a dedicated space in FLASH, initializes the DR200x modules and then checks to see if the *boot loader mode* should be entered or not. *Boot loader mode*, the *get program* state, waits for a program. The decision to enter boot loader mode depends on:
  - A register in FLASH being previously set by the boot loader itself or SCS.
  - The operator depressing the Enable push button in the SPHERES control panel for two seconds immediately after power on or upon reset.

If the boot loader mode is not entered, the boot loader checks if a program already exists in the FLASH memory. If it does, it will go into load program mode, otherwise it sends an `FL2EX_NO_PROGRAM` response and enters an infinite loop (slowly dimming Enable LED) to indicate no program exists in the FLASH.

- **Get Program** - in this state the satellite will flash the Enable LED slowly until it receives an `EX2FL_PROGRAM` command from the PC. It will then go to the *Get Package* state. If a satellite enters this state, it remains in this state until a program is received, an `EX2FL_RUN` command is received, or the satellite is reset.
- **Get Package** - executes the program transfer protocol described above (obtains program size from the first packet, acknowledges the packet, and then receives and acknowledges packages or 20 packets each). If there are more than 20 errors during the program upload (data error or timeouts) the loader returns to the *Get Program* state. If all the data is received, it stores the data in FLASH and then goes to the *Load Program* state.
- **Load Program** - this state reads the FLASH memory and copies each section of the program to its respective location in program RAM. After the program is placed in memory, it sets the program pointer to the location of `c_int00` (main) to execute the program. If the stored program length is zero or an error is detected with the program in FLASH, it will enter the infinite wait loop to indicate that no program exists.
- **Infinite Loop -** this state indicates that no program can be run with the satellite and therefore a new program must be loaded. The only way to exit this mode is to reset the satellite and enter boot loader mode.

The boot loader program has a single main thread which to perform its functions. Figure G.10 presents the general algorithm of the boot loader.

The redundancy and fallback mechanism include the use of hardware timers to control communications timing and to write data to the FLASH. The FLASH requires special timing sequences to start writing and between writing each sector, therefore the bootloader uses hardware timers to guarantee the timing. The timers do not create interrupts, they are polled to maintain a single-thread program. As discussed above, the boot loader initializes the DR200x modules. The reason is that the SCS programs can modify the DR200x configurations. The *load program* state will return the DR200x configurations to the standard SCS protocol, so that new programs are not jeopardized by configurations set in previous

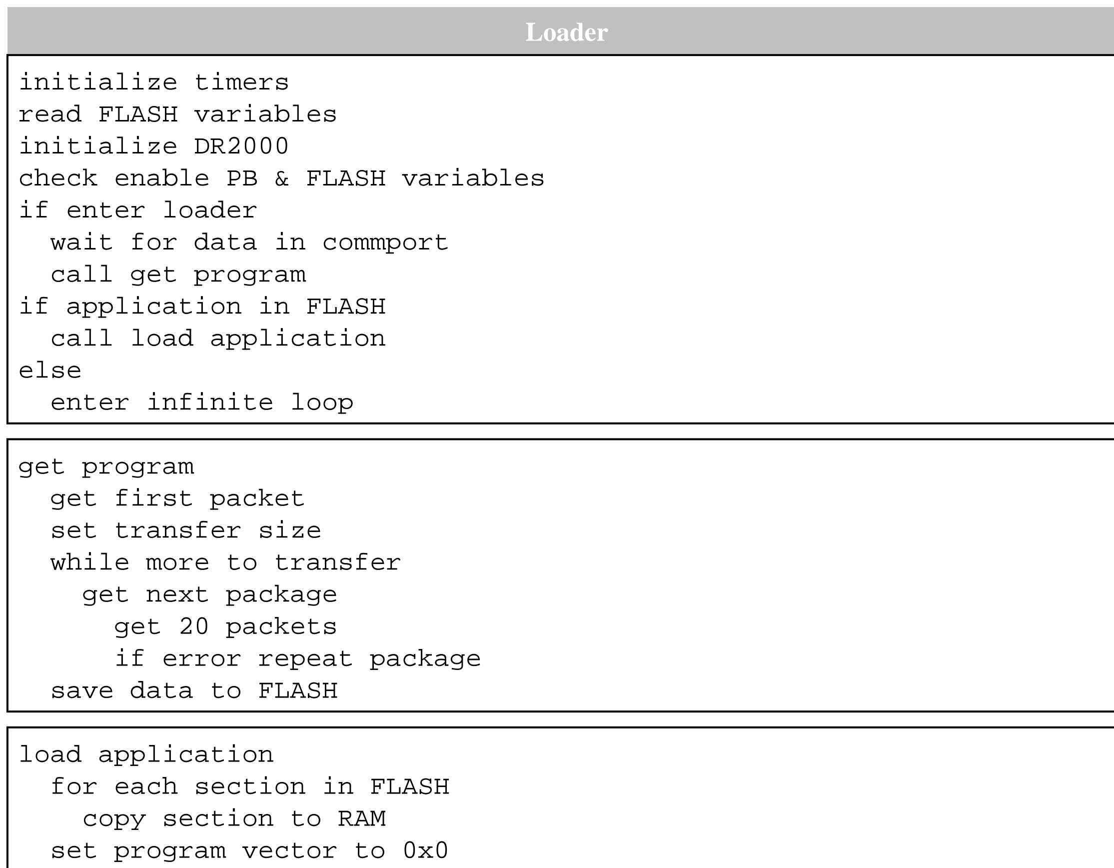

Figure G.10 Satellite boot loader general algorithm

ones. Lastly, the use of redundant sectors of FLASH to store critical values to identify the satellite and its configuration creates redundancy in the system. The use of these sections is explained further below.

G.1.3.1 FLASH Variables

Critical values of the satellite are maintained in redundant FLASH sectors. There are two copies of these variables (referred to as locations ’0’ and ’1’), such that if one copy is corrupted, the second copy can be used. The variables stored in these special sectors are:

```
unsigned FlagAppInFlash;      // existence of a program
unsigned AppLength;           // length (words) of program
unsigned EnterBoot;           // command to enter boot
unsigned ID;                  // satellite ID
unsigned BattTime;            // time satellite has been on
```

```
unsigned TankTime;              // estimate tank usage
float BeaconPos[6][3];          // global metrology setup
float BeaconDir[6][3];
float AccelScale[3];            // accelerometer scale factors
float AccelBias[3];             // accelerometer bias
float GyroScale[3];             // gyroscope scale factors
float GyroBias[3];              // gyroscope bias
unsigned count;                 // count of FLASH storage
unsigned checksum;              // checksum of FLASH memory
unsigned ver;                   // boot loader version
unsigned Padding[7];            // unused
```

The most important variables to the boot process are:

- `FlagAppInFlash` - indicates if a program has been stored in FLASH memory. A value of `0x12ABCDEF` indicates a program is present. Any other value indicates a program is not present (or is corrupt).
- `EnterBoot` - if set to 0x1 it will enter the boot loader mode and wait for a program even if one already exists and the enable push button has not been depressed during boot time. This allows the SCS to command the satellite to enter boot loader mode through the wireless communications systems, without the need for the operator to use the control panel. But, since the boot loader only changes this byte back to zero after a program has been loaded, it also effectively erases any current program in the satellite.
- `ID` - the satellite ID, which will always be matched to the packets to ensure the data is for this satellite.
- `Count` - a count of how many times these special FLASH variables have been saved. The count is used to identify the latest information between the two storage locations, since the boot loader always guarantees to the SCS that the latest information will be in one location '0'. Therefore, if location '1' contains the latest information, it is copied over to location '0'.
- `Checksum` - a checksum of all the previous values stored in the sector. If the latest version of the sector is corrupted, the boot loader checks the other sector. If the other sector is correct, it is copied to the corrupted sector, and the boot loading process continues. If both sectors are corrupt, the boot loader creates a new sector with the default values shown in Table G.5

TABLE G.5 Boot loader FLASH variables default values

Variable	Value
FlagAppInFlash	0;
AppLength	0;
EnterBoot	1;
ID	0x39;
BattTime	0;
TankTime	0;
BeaconPos[6][3]	{0};
BeaconDir[6][3]	{0};
AccelScale[3]	{0.0};
AccelBias[3]	{0.0};
GyroScale[3]	{0.0};
GyroBias[3]	{0.0};
count	0;
checksum	<new checksum>;
ver	0x31;

G.2 SPHERES Core

The SPHERES Core Software (SCS) layer acts as a buffer between the user-provided experiment code, the DSP/BIOS operating system, and the satellite hardware. The Texas Instruments DSP/BIOS [TI, SPRU423B] real-time operating system is used as the operating system on the SPHERES satellites. This kernel provides multi-processing capability, inter-process communication, and a number of input/output management tools. Through multiple execution threads created using DSP/BIOS, SCS controls the scientist provided functions which implement their specific algorithms. In this manner, the SCS creates a generic real-time operating system for the development of metrology, control, and autonomy algorithms. The core software encapsulates the hardware interfaces of the satellites by providing software interfaces to them. The SCS also preforms several housekeeping and data management tasks. The main functions of the SCS are:

- **Control.** The SCS implements a digital real-time controller and executes the code provided by scientists at a specified rate. The control module implements the test-management functions which allow a program to have multiple tests.
- **Propulsion.** SCS interfaces between the user code and the digital outputs to the propulsion hardware. The basic interface implements standard on/off pulses commanded through an array of on/off times.
- **Communications.** The SCS manages the TDMA communications protocol, handles incoming communications, and prepares standard and custom packets for transmission. The SCS also provides a module which allows scientists to transfer data of variable lengths (longer than a standard packet).
- **Metrology.** The SCS implements several threads to capture data and run metrology algorithms. High-priority threads collect the data. Middle priority threads can process data at high frequencies. Two low priority threads are implemented to run extended procedures in the background.
- **Housekeeping.** The SCS performs a number of routine tasks automatically in the background: it monitors the tank usage, reports health status to the control station, and downloads telemetry information.

The organization of the SCS modules with respect to the guest scientists modules, the DSP/BIOS kernel, and the SPHERES hardware is depicted graphically in Figure G.11.

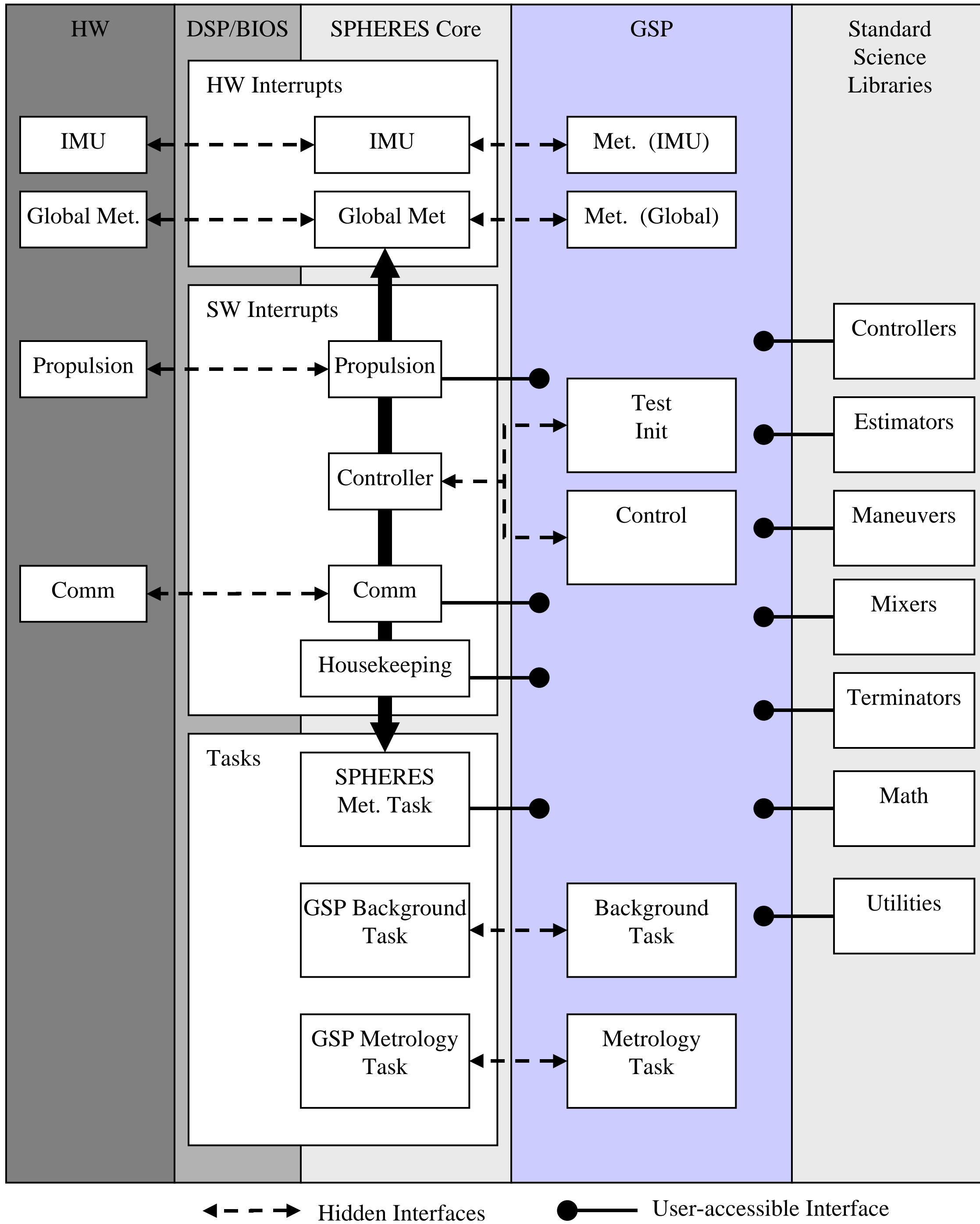

Figure G.11 SPHERES Core Software overview

SCS makes use of five types of threads available in the DSP/BIOS kernel (listed from highest to lowest priority): hardware interrupts (HWI), hardware clock interrupts (CLK), software interrupts (SWI), periodic software interrupts (PRD), and semaphore driven tasks (TSK). Figure G.12 presents the threads implemented by the SCS to create the SPHERES generic operating system. The following threads are used:

- **Hardware interrupts (HWI)**
  - **IR Rcv** - reception of a global metrology infrared signal.
  - **PADS Int** - interrupt by the metrology FPGA indicating data availability.
  - **COMM Rx** - reception of data through one of the commports
  - **CLK** - an interrupt created by DSP/BIOS to drive the periodic hardware clock interrupts
- **Periodic Hardware Interrupts (CLK)**
  - **Propulsion** - creates the propulsion solenoid signals at 1kHz
  - **Comm TDMA Mgr** - manages the TDMA transmission windows for the satellite
- **Software Interrupts (SWI)**
  - **PADS** - triggered from the PADS Int HWI, it collects the data, performs some data processing, and stores the results for other processes
  - **COMM Tx** - transmits data out through the commports one full packet at a time during the TDMA transmission window
  - **PRD** - a thread created by DSP/BIOS to manage periodic SWI's
  - **Control Dispatch** - manages the timing of the control software interrupt at 1kHz, allowing simple changes in the rate of the controller interrupt
  - **Fast Housekeeping** - keeps track of the state of health of the satellite and triggers the watchdog to prevent a hardware reset
  - **Telemetry** - downloads state information periodically
  - **Controller** - implements the test management routines and runs the periodic control algorithm specified by scientists
  - **KNL** - the kernel process created by DSP/BIOS
- **Background Tasks (TSK)**
  - **Comm Mgr** - provides the interface for all incoming and outgoing data
  - **PADS Global TX** - used to trigger an infrared pulse for global metrology

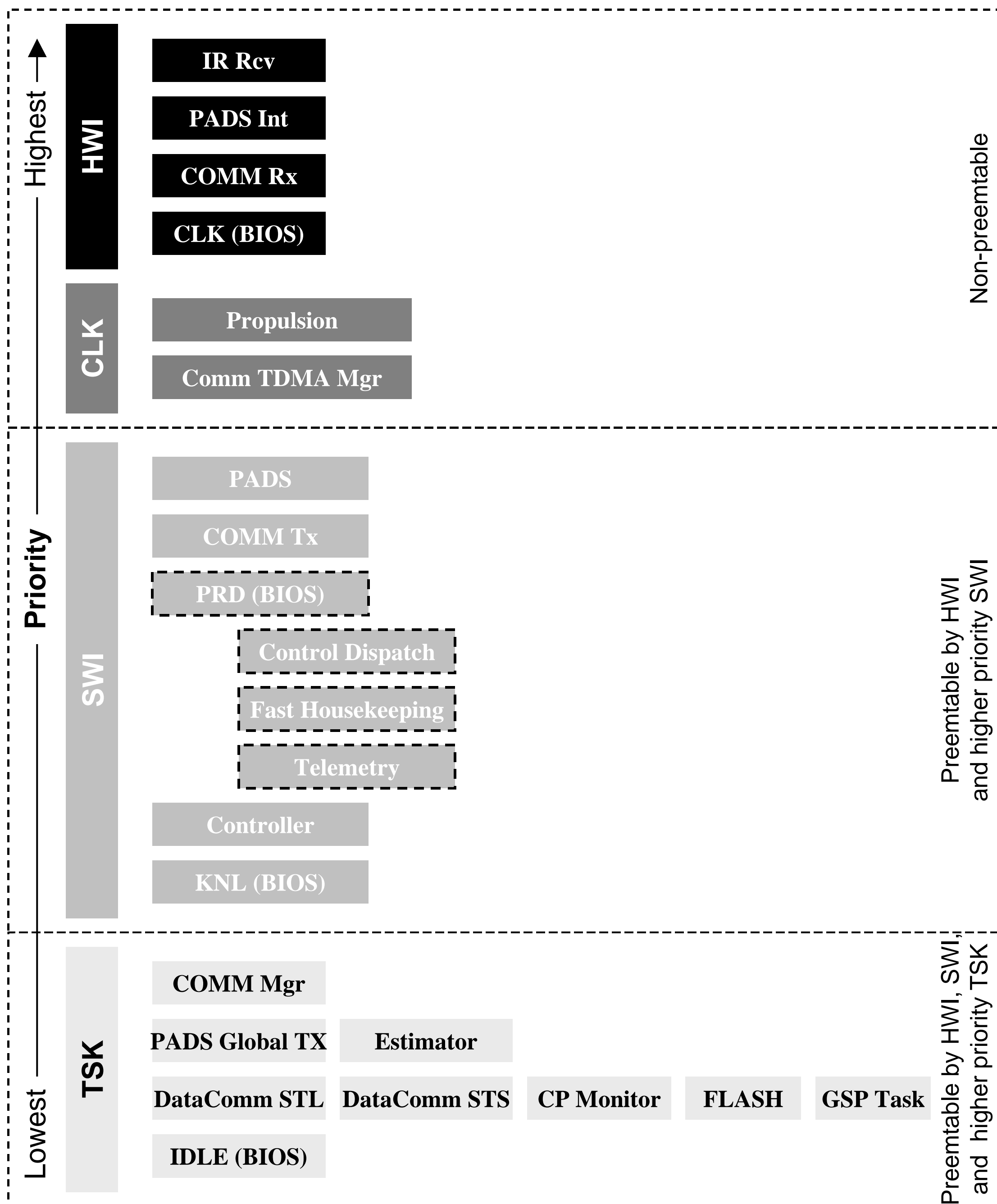

Figure G.12 SCS threads

- **Estimator** - runs the SPHERES standard estimator
- **DataComm STL/STS** - manages large data transfers by dividing it up into standard SPHERES packets on the transmitting satellite and rebuilding the data in the receiving satellite

- **CP Monitor** - monitors the state of the commport interrupts upon boot
- **FLASH** - writes to the FLASH memory in the background, since the flash requires several milliseconds between writing each sector
- **GSP Task** - allows guest scientists to run extended tasks in the background
- **IDLE** - the idle process created by DSP/BIOS

The SCS utilizes several global timers to maintain a system time, to trigger timed threads, and to control timed events. Figure G.13 presents the major timers used in the SCS. Their description follows.

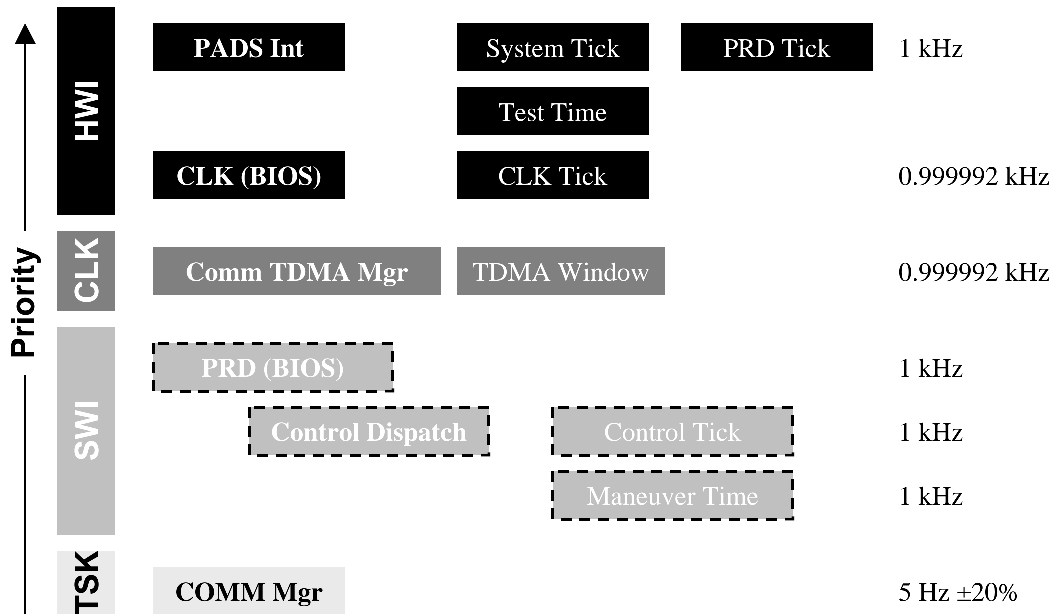

Figure G.13 SCS real-time clocks

- **PADS Int -** From the Metrology FPGA 1kHz hardware interrupt
  - **System Tick** - Maintains the global system time in milliseconds. Reset to zero when the satellite is reset (hardware or software reset).
  - **Control Tick** - The control tick is used by the control dispatcher thread to post new controller SWI's. Because the control tick is based on the PRD and the PRD on the PADS Int, the controller is always synchronized with the system tick.

  - **Test Time & Maneuver Time** - The test time and maneuver time reset whenever a new test or maneuver starts, respectively. They are synchronized with the system time.

- **CLK** - From the DSP hardware timer based on the SMT375 clock of 166.67MHz, which results in a period of 1.0078ms which gives a frequency of 0.999998kHz

  - **Propulsion & Comm TDMA Window** - The propulsion and communications TDMA manager functions utilize the SMT375 clock, rather than the PADS clock. Therefore, these functions are slight off-sync from the global system time. Because they operate through relative times (active = off_time - on_time in both cases) the errors are always minimal.

- **COMM Mgr** - The communications manager task processes the packets received from the control laptop, which include the command to start a new TDMA frame. Because the laptop can have errors as large as 20ms, this process operates at 5 Hz $\pm 20\%$. This requires the satellites to store all the data until a valid window appears, and breaks any correlation between the data reception time and the time it was created, therefore packets are identified with the system time when they are created.

These threads and timers support the modules of the SCS. The next sections describe the modules of the SCS in detail:

- System
- Control
- Propulsion
- Communications
- Metrology
- Housekeeping
- Guest Scientist Program Interface

G.2.1 System

The system module includes the system initialization routines and maintains the global satellite time and the satellite's physical parameters.

The system is initialized using the standard C function `void main()`. It is a thread which executes **once** after a reset to initialize the satellite and exits after completion. The DSP/BIOS kernel starts the real-time environment threads after the main function completes. The function performs the following actions:

- Initializes the hardware timers, global bus, DR200x, metrology FPGA, and satellite state estimate.
- Loads the FLASH variables to identify the satellite.
- Sets physical parameters.
- Runs the GSP initialization routines.
- Sets up hardware interrupts.
- Loads the DSP/BIOS kernel.
- The module is comprised of the following files:

The system module implements the local variables used to maintain the time of the satellite and tests. These times are kept as follows:

- **System time** - Set to zero during program initialization. Updated in the PADS Int HWI
- **Test time** - Set to `SYS_FOREVER` when a test is not running. It is increment whenever it is *not* `SYS_FOREVER` (the control module starts the clock by asking the system module to set it to zero) through the PADS Int HWI.

The system module also maintains the *logical* identity of the satellite. The logical identity tells the satellite what role it plays within a distributed system. This allows a scientist to decouple the physical serial number (i.e., the hardware ID used in the communications system) from the logical role played by the unit during a test. In this manner, any satellite can be used to perform any role.

Every program is identified by a unique number. This number is controlled by the SPHERES team so that the ISS GUI can identify the program currently loaded in a satellite and indicate the name of the program to the astronaut. The ground-based GUI shows the integer identifier so that scientists also know which program is loaded. Neither the SCS nor the interfaces control the sequence of the program ID, which means that the SPHERES team members must manually ensure the numbers uniquely identify a program readied for tests aboard the ISS.

The system module also provides a high-level interface to enable and disable interrupts following the guidelines of the DSP/BIOS kernel. It includes a function to perform atomic memory copy by handling the interrupts automatically.

The module also provides an interface to the physical properties of the satellite, including:

- satellite dry mass
- full tank propellant mass
- estimated total wet mass given current propellant consumption
- satellite inertia matrix and its inverse
- satellite center of mass (dry and wet)
- model of the thrusters: strength, direction, and location

Source Files

- init_sphere.c
- fpga.c
- main.c
- system.c
- spheres_physical_parameters.c

Internal Header Files

- init_sphere.h
- system_internal.h

Public Header Files

- fpga.h

- spheres_physical_parameters.h
- SMT335Async.h
- spheres_constants.h
- spheres_types.h
- system.h

G.2.2 Control

The control module uses two threads to implement a periodic routine which allows substantial calculations at a user-selectable rate. Rather than using a hardware interrupt based on the hardware times, which is commonly done in embedded control systems, SCS utilizes two software interrupts. Using HWI would give the controller a high priority and prevent any other threads from executing while the control algorithm executes, unless special steps are taken to enable certain levels of preemption within the hardware interrupts. By using the software interrupts provided by DSP/BIOS, the SCS can easily configure the priority of the control interrupts and enable preemption by processes with more strict real-time requirements or higher rates. The use of SWI also makes a hardware timer available for other functions (it is used to control the timing of writes to FLASH memory).

The two threads which implement the control module are presented in Figure G.14. Their functions are:

- **Control dispatch** - This thread executes at a constant 1kHz regardless of the state of the satellite. Its purpose is to provide a simple interface to change the rate of the controller with period increments of 1ms independently of the controller itself. It also forms part of the synchronization routines so that multiple satellites start the tests at the same time.

  When a test start command is received by the communications module, it indicates to the software dispatcher that a test will start. The dispatcher then waits one second (1000 cycles) before starting the test, which gives the communications module enough time to acknowledge the command.

  The thread maintains a local timer to control the period at which the controller software interrupt is posted. Once the dispatcher posts the controller SWI, the controller will execute as soon as no other higher priority tasks are pending, usually within micro seconds. Because the dispatch thread executes continuously, the controller interrupt will be posted regardless of the state of the satellite. It is the controller SWI that determines what action to take, not the dispatcher.

  The dispatcher also maintains the maneuver time when a test is running.

- **Controller** - This thread executes upon being posted by the dispatcher. It is a state machine which implements the test management functions of the SCS. Figure G.15 illustrates the state machine used by the controller thread. It has the following states:

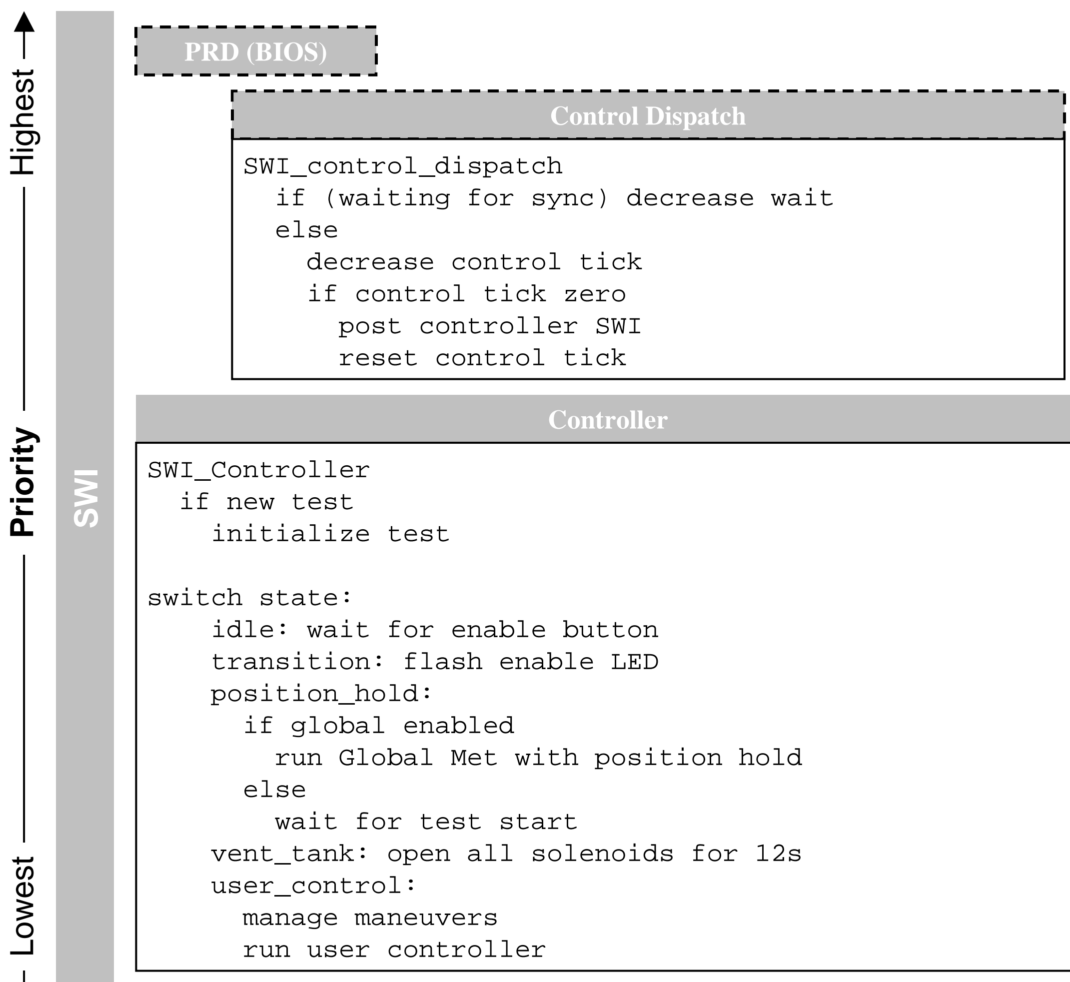

Figure G.14 SCS controller module threads and general algorithm

- **Idle** - When idle, the controller performs no actions. It waits for a new test to start or the Enable PB to either enable the satellite (depressed for one but no more than two seconds) or command a tank vent (depressed for more than five seconds). The idle mode is indicated in the SPHERES control panel by the Enable LED being off.

  To start a test the operator must first enable the satellite by using the Enable PB so the state goes to the Position Hold / Ready state. Scientists can bypass the need to enable a satellite for tests in ground-based facilities where starting a test is time-critical (e.g., the RGA). This feature is also useful during operations at the MIT SSL where dozens of tests are conducted in series to debug a program. The SPHERES team will ensure that enabling a satellite is not bypassed for tests aboard the ISS.

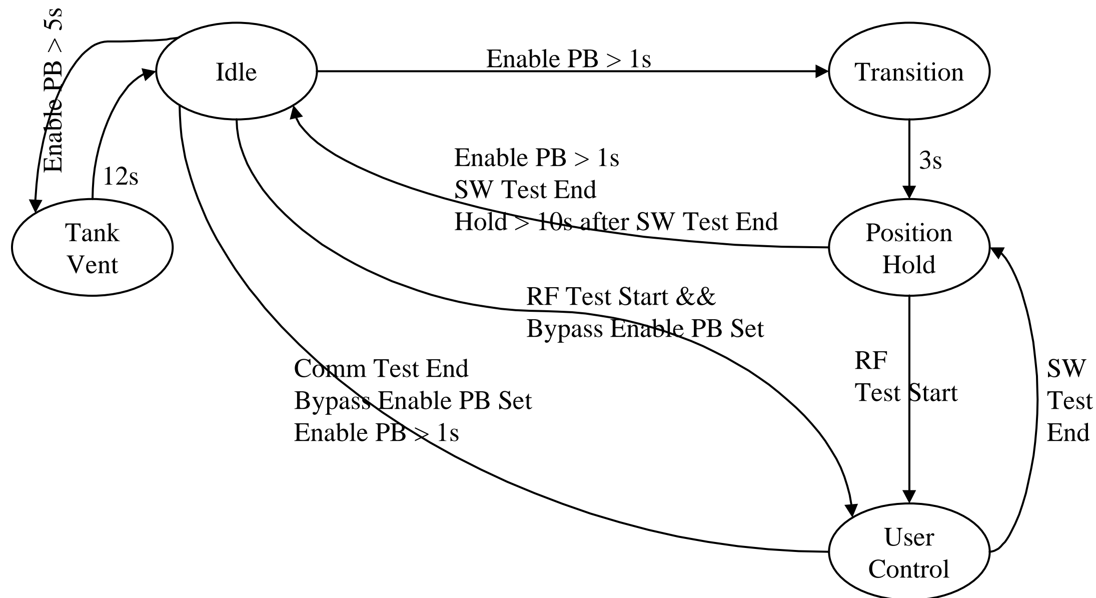

Figure G.15 SCS controller state diagram

- **Transition** - If the Enable PB is used to enable a satellite for tests, the controller enters a transition state to allow the operator to deploy (let go of) the satellite prior to entering the Position Hold mode. The transition state is fixed at three seconds, and changes automatically to the Position Hold state after the pause. The transition mode is indicated in the SPHERES control panel by a flashing Enable LED.
- **Position Hold / Ready** - This state can operate in two ways, depending on whether the global metrology system and the MIT estimator are available or not.

  If the global metrology system and the estimator are available, then the satellite will measure its position when released by the operator (after the three second transition) and maintain that position until a test is started. This helps operators locate multiple satellites without having to worry about drift.

  When the global system and/or estimator are not available, the satellite performs no actions.

  If the Enable PB is not explicitly bypassed in the program, the satellite must be in Position Hold / Ready mode before a test is started.
- **User Control** - Once a new test is commanded and the unit is enabled (or bypass enable is selected) the controller first runs the test initialization routines. These include management of local variables, including the change of state to user control, as well as executing the GSP test initial-

ization functions. Because these functions are executed within the controller SWI, they must run in the time allotted to one control period.

After the test is initialized, the controller performs maneuver management functions to maintain the maneuver number and time, it then executes the GSP controller functions.

The controller changes state again after a test has ended. If the GSP algorithm indicates a successful test and the global metrology and MIT estimator are available, the satellite will enter Position Hold mode for 10 seconds to cancel any residual velocity from the test. After the 10 second position hold, the satellite returns to Idle.

If the test terminates successfully but there is no global metrology and/or the MIT estimator is not available, the satellite returns to Idle.

If the test is interrupted using the Enable PB or via a wireless command, the satellite returns to Idle immediately.

- **Tank Vent** - This state opens all the solenoid valves so that a tank is completely empty before its removal. The state terminates automatically after 12 seconds of firing the thrusters and always returns to Idle.

Source Files

- control.c

Internal Header Files

- control_internal.h

Public Header Files

- control.h

G.2.3 Propulsion

The propulsion module creates a software interface to control the thruster solenoid valves with one millisecond increments. The module utilizes one main thread, CLK Propulsion, to control the state of the solenoids. Because the thrusters create white noise which triggers the ultrasound receivers of the metrology system, the thrusters cannot be active during a global metrology cycle. The global metrology process is initiated by an infrared pulse. Therefore, the IR Rcv HWI affects the propulsion module directly because upon IR reception the thrusters are turned off until the Metrology module indicates that the global metrology cycle has finished. Figure G.16 presents the two processes used in the propulsion module.

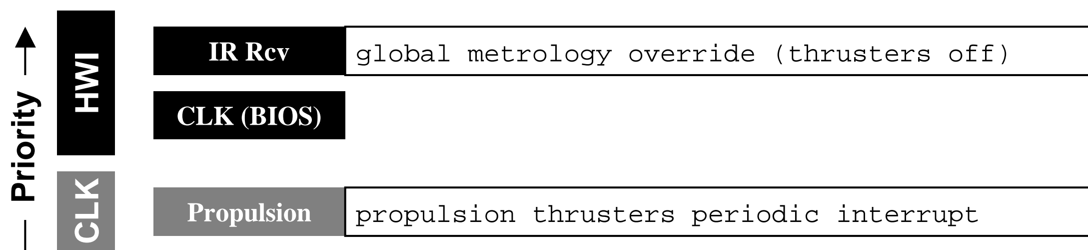

Figure G.16 SCS propulsion module threads

The propulsion CLK interrupt interfaces with the rest of the modules using an array of on/off times for each thruster. These times are relative to the first time the array is read by the propulsion interrupt, they are not relative to the satellite, test, or maneuver times. Therefore, when any other module (usually the controller) commands a set of on/off times to the propulsion module, it is equivalent of sending the command to an external module which is not synchronized with the rest of the system. The timing of the propulsion module is illustrated in Figure G.17.

By interfacing the propulsion module though an array of on and off times, the scientist can simulate several types of discrete control actuators. In its simplest form, as illustrated in Figure G.17, it commands on/off pulses which start at the same time and end at different

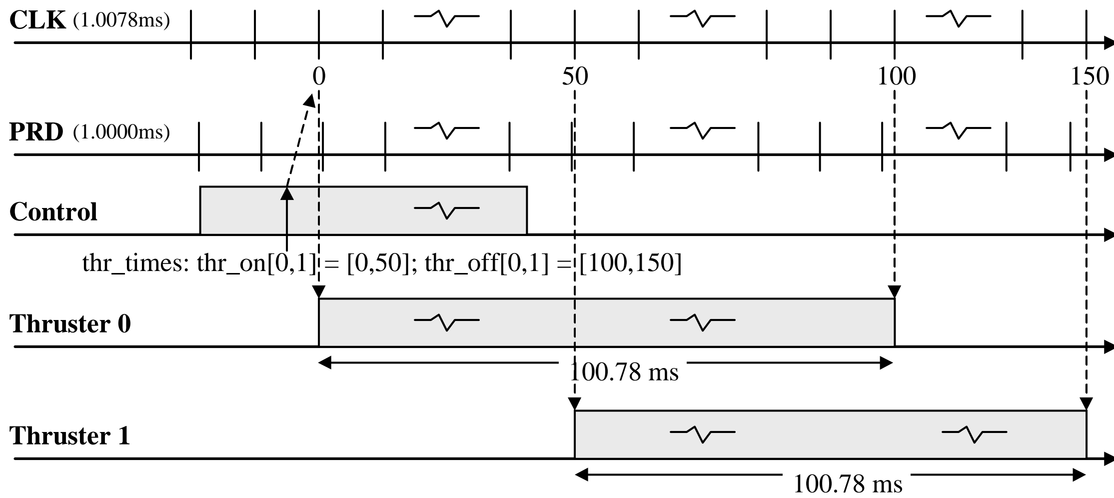

Figure G.17 Propulsion module timing diagram

times. But the scientist can also create algorithms to center the pulses, have them at the end of a period, or mix them. This allows the implementation of several types of modulation, including pulse width and frequency modulation. These possibilities are pictured in Figure G.18

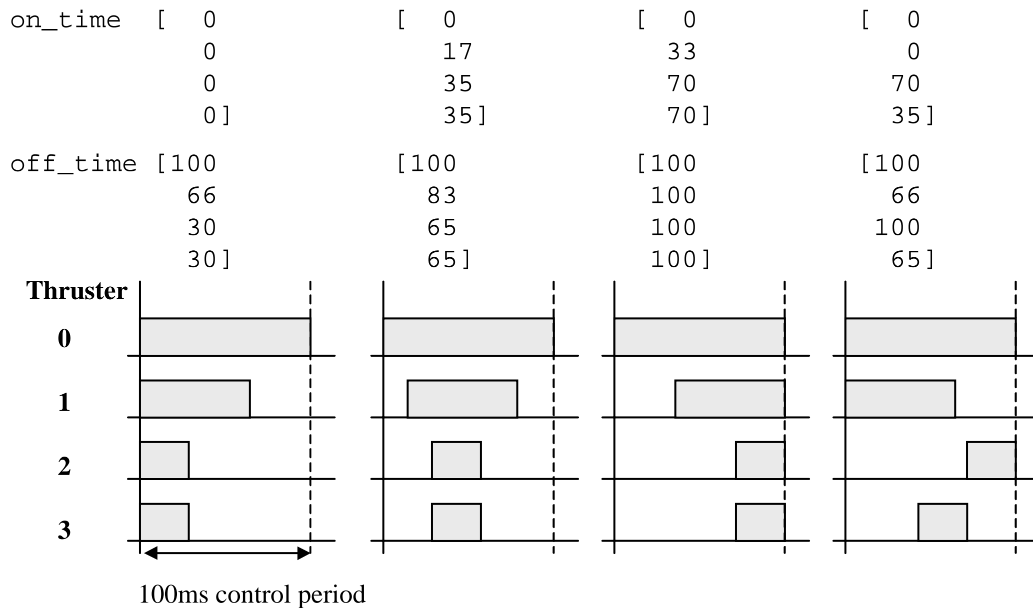

Figure G.18 Propulsion modulation options

Source Files

- prop.c

Public Header Files

- prop.h

G.2.4 Communications

The communications module implements the TDMA protocol and provides both high priority and low priority queues for data transmission. The module uses several synchronization and data management tools provided by DSP/BIOS to manage the data securely in a multi-thread environment. Appendix H presents details on the TDMA protocol implementation and the SPHERES data packets. This section describes the data transfer and processing between threads of the SCS communications module.

Figure G.19 shows the threads used by the communications module. The module separates the reception and transmission tasks in high priority interrupts, but joins them in the background communications management task which provides the actual interfaces to the communications module.

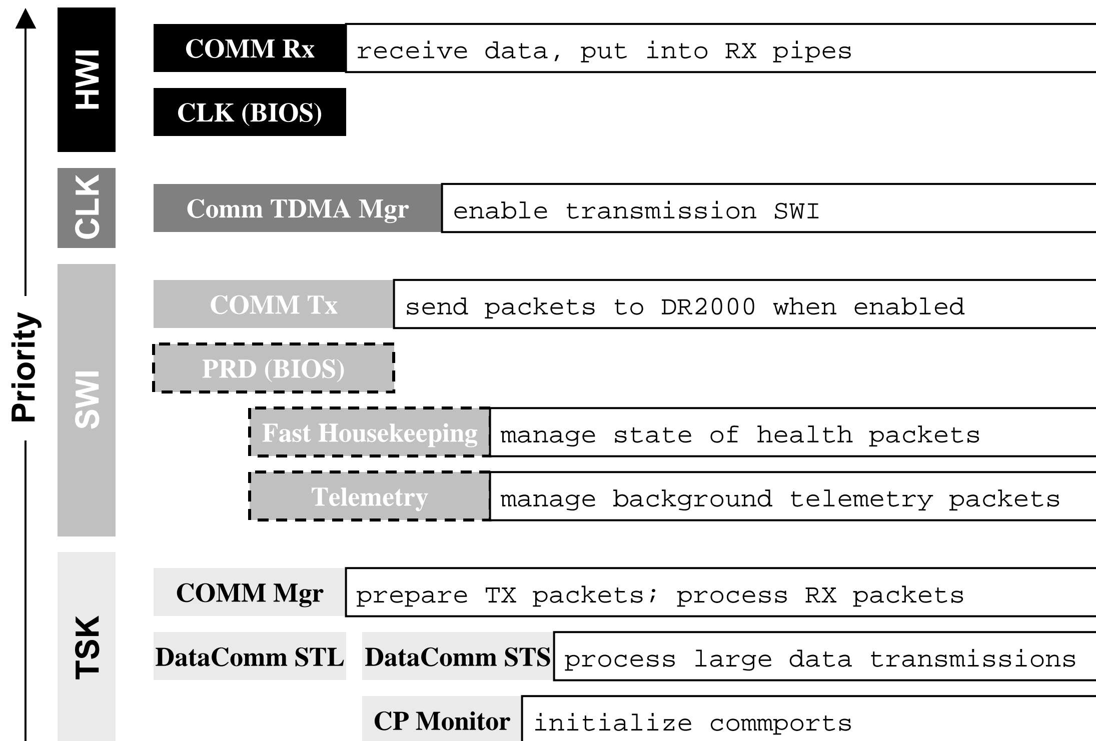

Figure G.19 SCS communications module threads

Data Reception

Figure G.20 presents the data reception processes. When data is received by the commports, a hardware interrupt is posted. The reception hardware interrupt collects the data and stores it in a temporary buffer. This procedure *does* process the data to identify complete packages, and only places complete packages in the pipes of lower priority processes. The reception interrupt also identifies commands to start a new test, since it resets the synchronization of the control periods between multiple units (the wait in the control dispatcher). It does not perform any other data processing; interpreting all other commands and data is done by other threads.

The reception HWI places complete packets in a *pipe* construct. Pipes are data management tool provided by DSP/BIOS which implement queues and call data processing functions automatically. The DSP/BIOS PIP module, used by the SCS, accounts for the multi-threaded nature of the system. When a packet is placed in the pipe, a semaphore is posted to the communications management task to indicate that new data is waiting.

The communications management task works in the background of all real-time processes to handle commands from the laptop and identify packets from other satellites. The communications module of the SCS handles the following types of packets:

- **Telemetry** - the state information of other satellites is placed in local variables
- **State of health** - the state of health of other satellites, including their role, is stored in local variables
- **General commands** - commands from the ground station, described below
- **Beacon initialization** - the configuration of the global metrology system is uploaded by the ground station
- **Large data transfer** - packets that form a custom data transmission of the guest scientist

All other types of packets are ignored by the SCS.

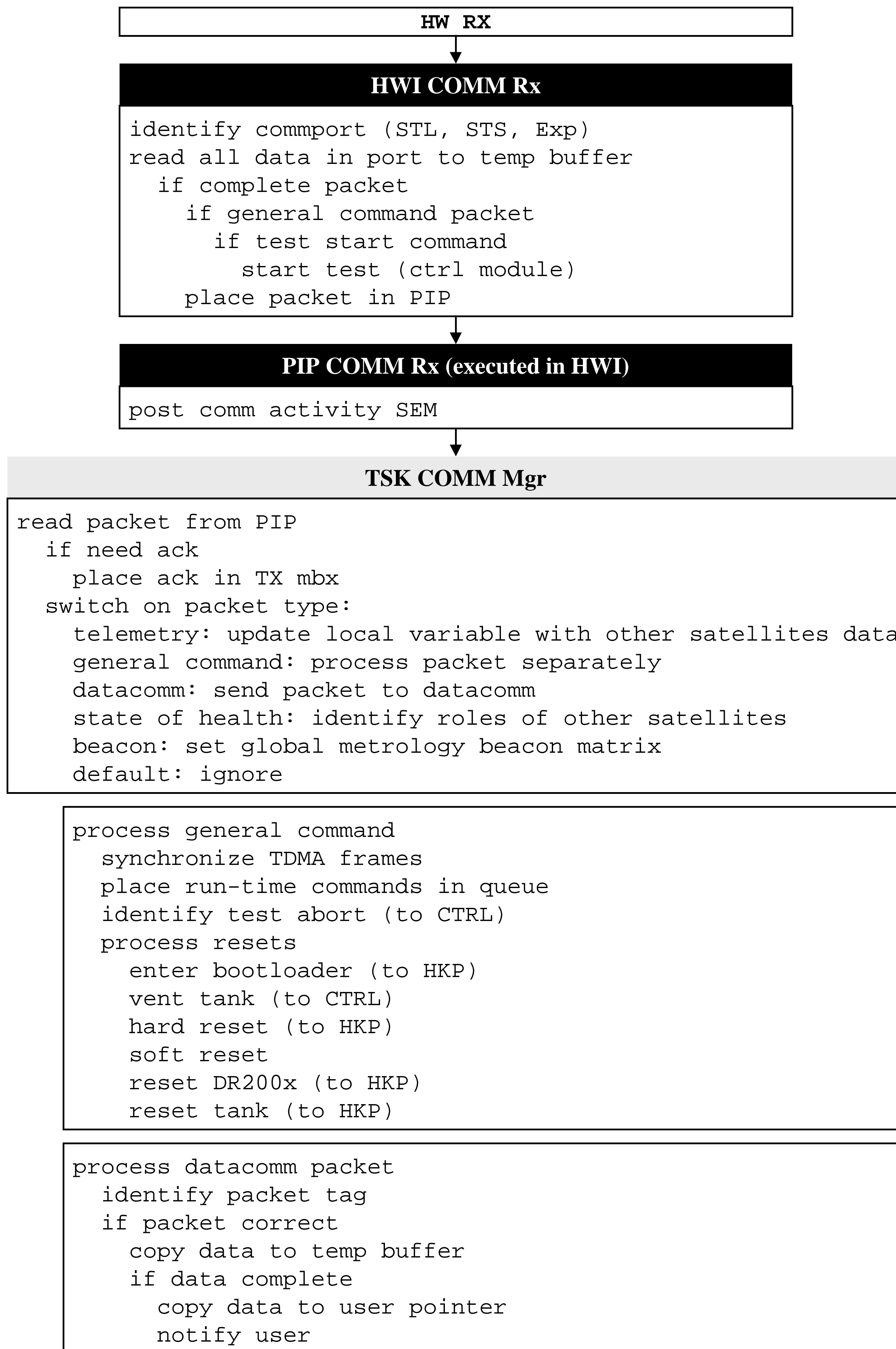

Figure G.20 Communications data reception process

Scientists can transfer data between satellites by using the large data transfer tools of the communications module, even if their data is smaller than a standard SPHERES packet. These tools automatically format the data for transmission and re-incorporate the data upon reception by the intended satellite. Scientists must use this tool, since unknown packet types are ignored by the communications module and are not available to the scientist.

General command packets from the ground station are processed in this task, except for a test start, which is handled in the hardware interrupt to synchronize the satellites. The commands are actually executed by other modules (housekeeping, control), but are triggered by this process.

Data Transmission

Figure G.21 presents the processes used for data transmission. The transmission of data must only occur during the TDMA window assigned to the satellite to prevent contention in the wireless network. A periodic hardware interrupt, the Comm TDMA Mgr CLK process, times the length of the TDMA windows. The start time of a TDMA cycle is commanded by the reception of a general command packet from the ground station; the TDMA manager process records this start time and opens the transmission window by posting a semaphore to the communications manage task. The communications management task then posts the Comm TX SWI if data needs to be transmitted. The Comm TX SWI sends one complete packet at a time, since the hardware requires that packets be delivered without long interruptions (no more than 2ms between bytes).

As illustrated in Figure G.21, transmission data can be created by a wide range of processes. The periodic housekeeping and telemetry tasks create state of health and state telemetry packets continuously. The controller send a confirmation every time a test is started. The scientist can create data in practically any process, including the periodic controller, program and test initialization, and background tasks. This data can be in the form of standard SPHERES packages by using the publicly available function

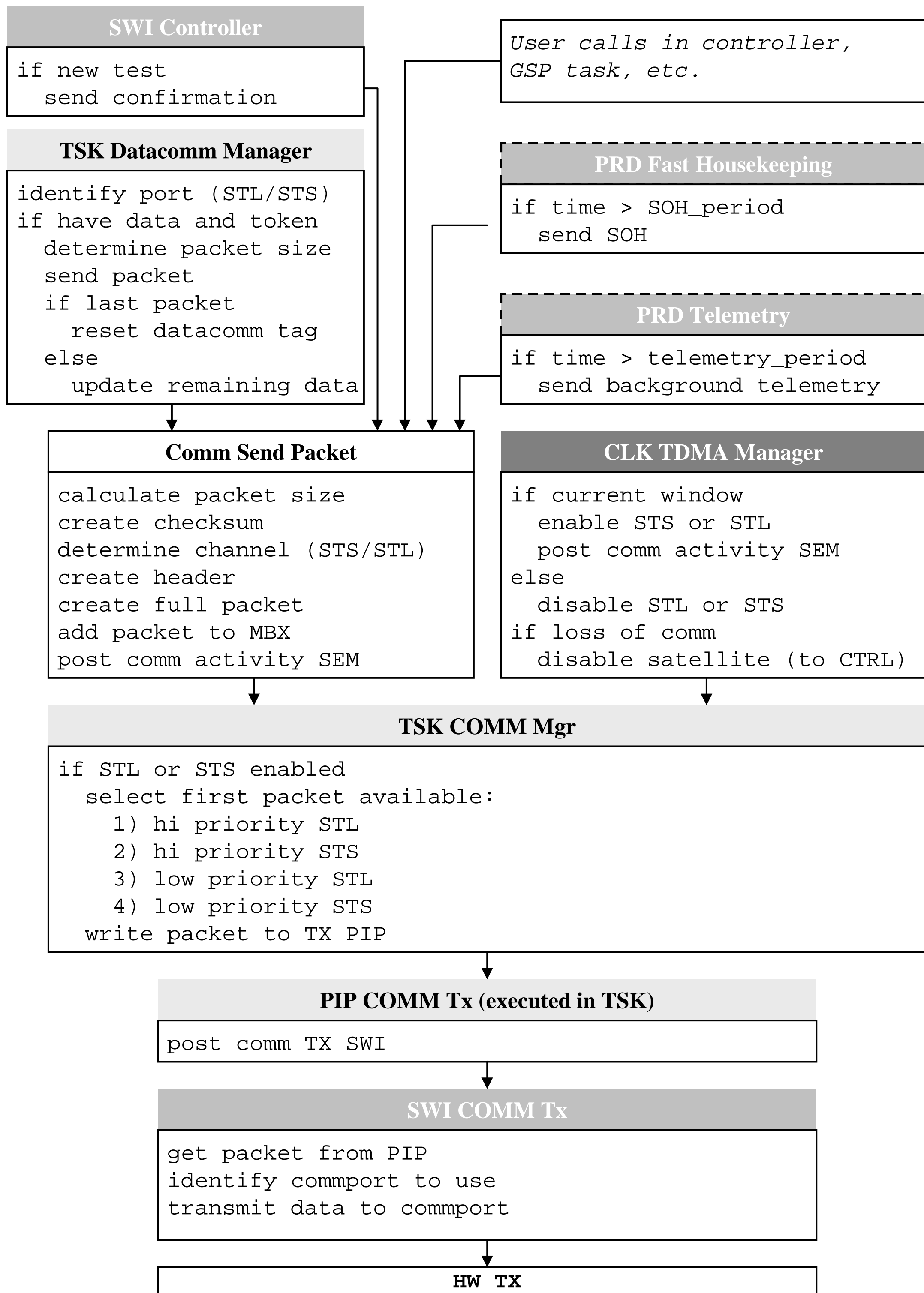

Figure G.21 Communications data transmission process

`CommSendPacket`, or by using the large data transfer tools (datacomm, which used send packet itself). The interface function `CommSendPacket` is the only interface to the communications transmission procedures to ensure that the TDMA protocol is maintained.

The `CommSendPacket` function, which can execute in any type of thread, utilizes the DSP/BIOS supplied mailboxes to store the data for the communications management thread to process. Mailboxes provide the necessary atomic operations and controls to maintain the data safe regardless of what type of thread made the post to the mailbox. After placing the data on the mailbox the function posts a semaphore to indicate to the communications management task that new data is ready for transmission.

The communications management task checks four mailboxes for data transmission when a TDMA window is available: a high and allow priority mailbox for each of the STS and STL channels. Figure G.21 indicates the order in which the mailboxes are searched. When a packet is found, the task sends the data to the transmission pipe. This causes a Comm TX SWI to be posted until the pipe is empty; the Comm TX SWI transmits the packet to the hardware.

Source Files

- comm.c
- comm_interrupt.c
- comm_process_rx_packet.c

Internal Header Files

- comm_internal.h
- comm_interrupt.h
- comm_process_rx_packet.h
- commands.h

Public Header Files

- comm.h

G.2.4.1 DR200x Driver

The communications module also implements several procedures to configure and interface with the commports and the DR200x modules. The functions of the communication driver sub-section are to:

- Manage the communications port
  - Read input buffers
  - Empty input buffers
  - Write to output buffers
  - Prevent output buffer overflow
- Send initialization commands to the DR200x
  - Send commands
  - Wait for response
- Reset DR200x modules

Source Files

- comm_driver.c

Internal Header Files

- comm_driver.h

G.2.4.2 Background Telemetry

The background telemetry sub-module manages the transmission of state information between satellites automatically at a specified rate. The scientist can define the state vector variable to use for state transmission at the start of a program or test. The background telemetry functions will then transmit that state periodically without further intervention by the scientist. The procedures automatically stores the received information in local structures, which can be accessed by the guest scientist using the function `commBackgroundStateGet`. The background telemetry utilizes the following thread:

- **PRD Telemetry** - sends the telemetry information at a periodic rate
- **TSK Comm Mgr** - processes received packets by executing the background telemetry unpack function

Source Files

- comm_background.c

Internal Header Files

- comm_intrnal.h

Public Header Files

- comm.h

G.2.4.3 Datacomm

The datacomm sub-module allows scientists to transfer data or arbitrary size and format to ground and between satellites. The datacomm functions split large packets into standard SPHERES packets, manage the transmission of the multiple packets, and (if necessary) assembly the original data structure in the receiving satellite. To use the datacomm functions the scientist must initialize the transmission and reception buffers at the start of a program, then they only need command a new transmission; reception occurs automatically. The datacomm system allows scientists to poll the state of a transmission. The module triggers the GSP task when new data has been received.

The datacomm sub-module utilizes a background task for each channel to divide and transmit packets:

- **TSK_gspdata_manager** - Implements a heuristic leaky-bucket scheme to manage flow control between multiple datacomm transmission requests. This allows multiple datacomm request to be made simultaneously.

Data reception procedures are executed completely within the general communications management task.

To utilize datacomm scientists use the following procedures:

- Sending satellite
  - `datacommSendData (tag, *buffer, size, channel, to, mode, *TX_done_flag);`

    `tag` - unique identifier of the data

    `*buffer` - pointer to data location

`size` - size of the data in bytes

`channel` - STS or STL channel

`to` - destination (use 0 to broadcast)

`mode` - low or high priority

`*TX_done_flag` - user flag to poll until transmission is done

- Receiving satellite (optional)
  - In the program or task initialization (but **not** in test initialization), allocate space to assemble new data:

    `datacommInitializeTag(tag, buffer_size, timeout);`

    `tag` - unique identifier of the data

    `buffer_size` - size, in bytes, of assembly buffer (size of data transfer)

    `timeout` - timeout, in milliseconds, to cancel assembly of this tag
  - Before the data is received (i.e., preferable in the initialization routines), the scientist must allocate space for the assembled data to be copied into by using the following function:

    `int datacommRegisterBuffer(tag, *buffer);`

    `tag` - unique identifier of the data

    `*buffer` - pointer to the destination memory space allocated by the scientist

Source Files

- comm_datacomm.c

Internal Header Files

- comm_datacomm_internal.h

Public Header Files

- comm_datacomm.h

G.2.5 Metrology

The metrology module performs two tasks: collects metrology data and provides the necessary threads to process this data. To achieve this, the metrology module utilizes high priority interrupts to collect the data and low priority tasks to enable processing, as illustrated in Figure G.22. The metrology module utilizes the following threads:

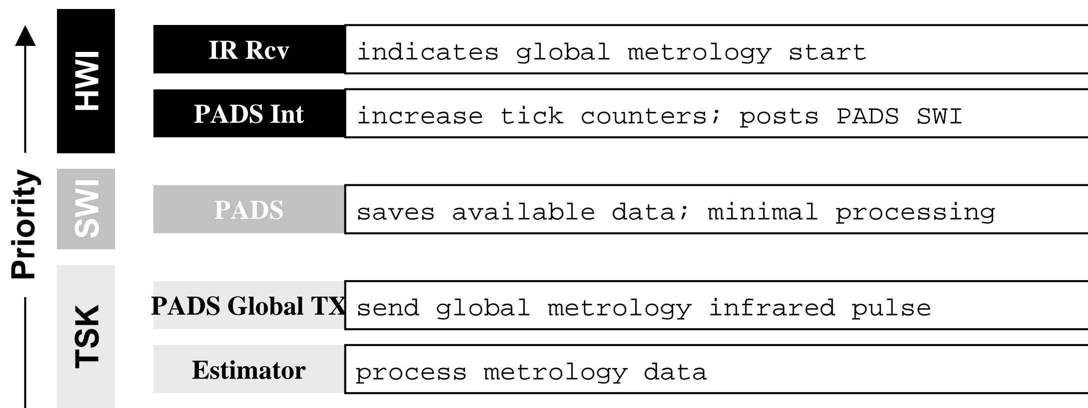

Figure G.22 SCS metrology module threads

- **Hardware Interrupts**
  - **IR Rcv** - The reception of an infrared pulse indicates the start of a global metrology cycle. The metrology FPGA collects the time-of-flight measurements of the ultrasound signals; to achieve the highest precision possible, the IR signals are connected to the FPGA externally via hardware, independently of the DSP. Actually, it is the FPGA which creates the signal that interrupts the FPGA. The IR Rcv hardware interrupt allows the SCS to transmit commands to the onboard beacon or external expansion items and to turn the thrusters off. Because no other tasks are performed, this interrupt is relatively short and very fast.
  - **PADS Int** - The hardware metrology interrupt is essential for the operations of the SCS. The FPGA creates a precise 1kHz clock used to control the satellite system time, which is maintained through the PADS Int HWI. The same interrupt drives the periodic software interrupts created by DSP/BIOS (PRD). But the interrupt does not perform any metrology data handling or processing directly, instead, it posts the PADS software interrupt, which collects the data.
- **Software Interrupts**

- **PADS -** The PADS SWI collects the data available in the FPGA. The FPGA will always interrupt at 1kHz and will always have IMU data available. But it will only have global metrology data available during a global cycle. Therefore, the PADS SWI checks the status of the FPGA to determine whether to save global data or not.

- **Tasks** - The availability of two tasks for data processing enable scientists to compare their algorithms with the standard SCS estimators.
  - **Estimator** - The estimator task is reserved for use with the SPHERES standard estimator which is part of the SCS. The estimator task, which runs in the background, perform both long calculations with the global metrology data and short updates with the IMU data.
  - **GSP Task** - The GSP task is provided to scientists to handle multiple events, including the reception of metrology data (the full range of events is described below, in the GSP section). Scientists can implement their own algorithms in this thread utilizing any combination of inertial and global data.
  - **Global TX** - This task sleeps in the background until triggered to command an infrared transmission. The transmission of an IR should only be done by one satellite, although there are no restrictions from the SCS programming.

The flow of data through the metrology system is illustrated in Figure G.23. The inertial and global data are transmitted through the FPGA to the DSP via the same HWI/SWI combination; the software checks which data is available. The SWI makes the data available to the scientist. The PADS SWI collects all the inertial data and immediately converts the digitized analog values to floating point values in units of [m/s^2] for accelerations and [rad/s] for rotation rates. The inertial data is stored in buffers configured during program initialization; the scientist can choose to receive the data at any rate in period increments of one millisecond. Further, the scientist can receive all the data (e.g., an array of 10 data sets every 10ms) or just the single data collected at that rate (e.g., one array of data every 10ms). The global metrology data is transferred as N packets of 24 measurements and one time stamp (i.e. 25 words total), where N is the number of beacons programmed during program initialization. The GSP interface functions to the inertial and global data execute within the SWI, therefore these functions must completely quickly.

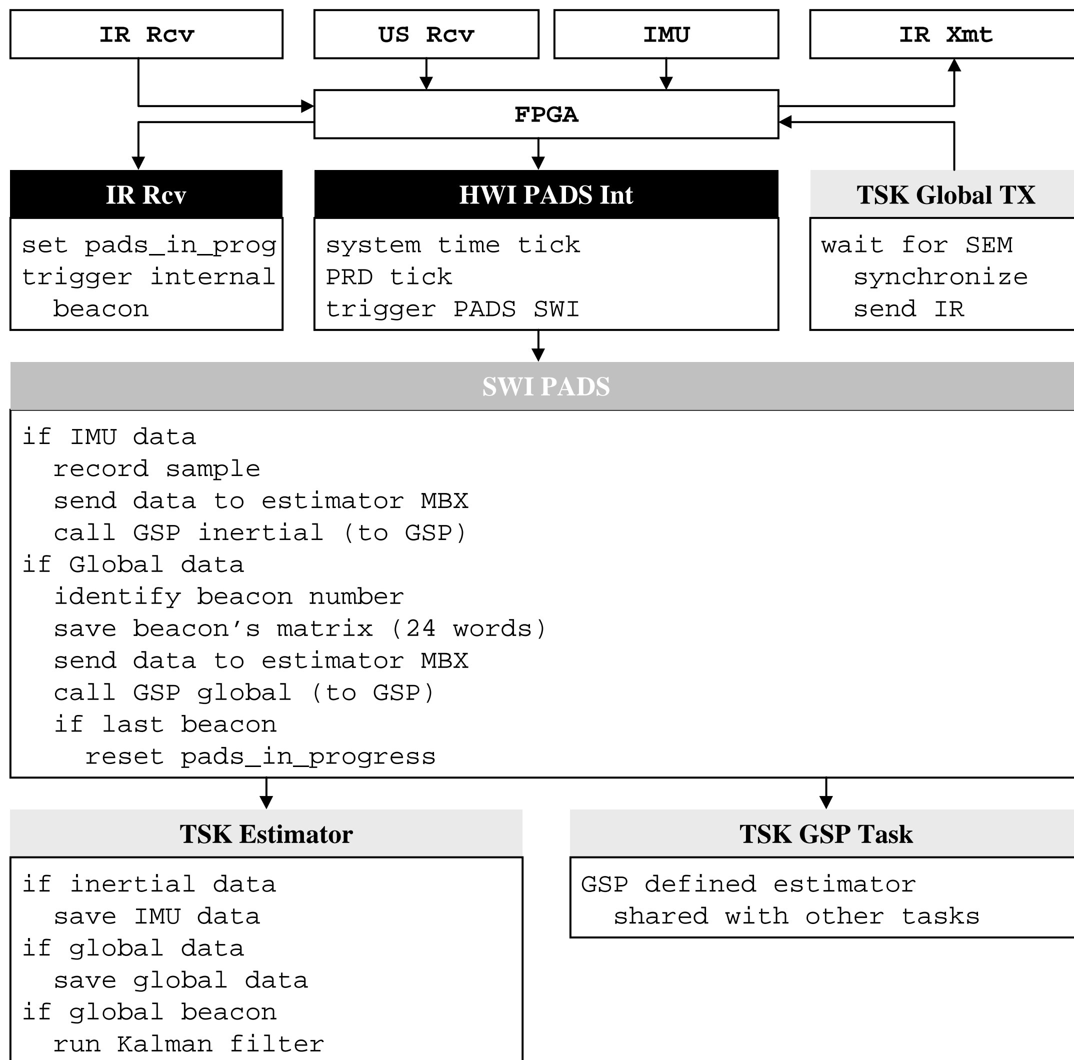

Figure G.23 SCS metrology module general algorithms

The data for the standard SPHERES estimator is placed in DSP/BIOS mailboxes. The estimator task pends on those mailboxes and executes when data is available. This allows the estimator task to remain asleep in the background when no data exists; the use of mailboxes allows the estimator task to run for extended periods of time without loosing data. Therefore, the estimator can process global metrology data over extended periods of time and then use all the inertial data collected throughout the global metrology processing period.

The interface with the GSP uses a semaphore. The semaphore is posted consecutively, but it is the responsibility of the scientist to save the data through the inertial and global data collection functions, since the SCS will not save the data for the scientist otherwise. Figure G.24 presents a sample timing diagram of the scheduling of the metrology treads during both inertial and global data cycles.

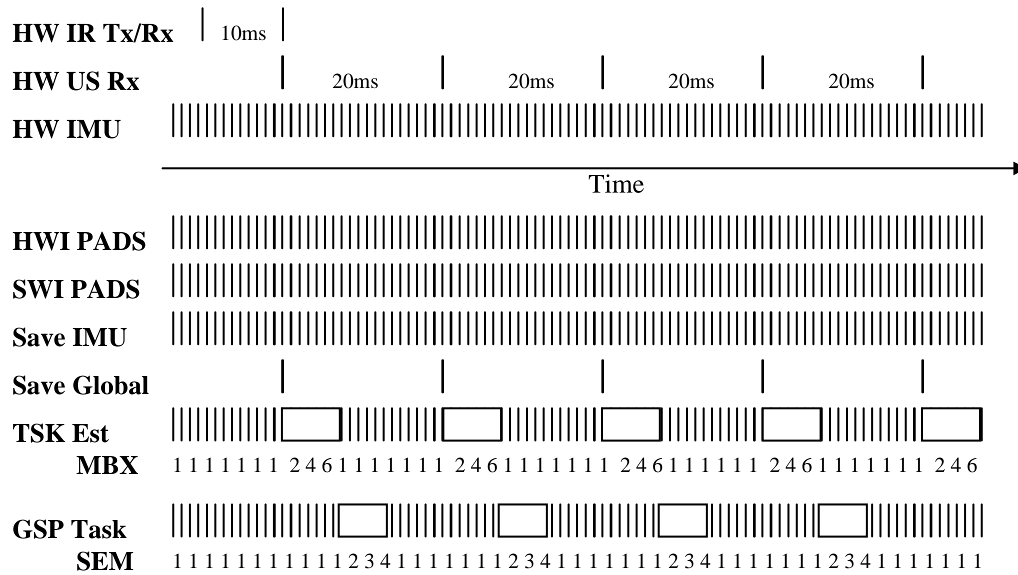

Figure G.24 SCS metrology treads scheduling

Source Files

- pads.c
- pads_correct.c
- pads_estimator.c
- pads_request.c

Internal Header Files

- pads_internal.h
- pads_correct.h
- est_USsubfunc.h

Public Header Files

- pads.h

G.2.6 Housekeeping

The primary function of the housekeeping module is to maintain the ground station informed on the state of health (SOH) of the satellite and save information to the FLASH loader variables. The housekeeping module also checks the status of the commports during boot time and then periodically throughout operations. As shown in Figure G.25, three threads are used to perform these functions:

- **Fast Housekeeping PRD** - The main housekeeping thread, it collects the information periodically and sends out the state of health packets. Because the SOH packets are used to acknowledge commands from the ground station, the housekeeping task is also used to acknowledge commands.
- **FLASH TSK** - Because the boot loader program resides in the FLASH, it is essential to protect that space in memory. Therefore, the SCS provides a single-point interface to the FLASH via this task. The task filters the write addresses to ensure that programs do not overwrite the boot loader (although the could overwrite themselves). This task implements the time delays necessary between FLASH write cycles.
- **CP Monitor TSK** - This task is started upon boot to check the interrupt flags of the communications ports, which exhibits a race condition after a hard reset. The task resets the commport interrupt flags. During operations the task checks that the status of the commports corresponds to the indicated interrupt flags approximately once a second.

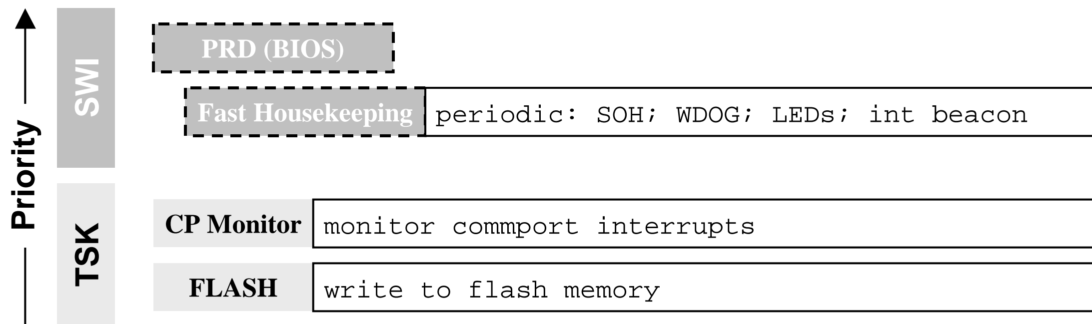

Figure G.25 SCS housekeeping module threads

State of health packets are created and transmitted at 1Hz by default. To create the SOH packets, the housekeeping module collects information from the other modules. The state of health of the satellite is collected as follows:

- System time (from system)
- Program ID (from system)
- Tank usage (from propulsion)
- Test time (from system)
- Maneuver time (from control)
- Last test result (local, set by control)
- Number of received IR pulses (from metrology)
- Communications status (from communications)
- Beacon information (from system)
- Controller state (from control)
- Acknowledgement (local, commanded by communications)

The housekeeping periodic interrupt provides the software interface to the metrology FPGA digital outputs which consist of the Enable LED, the internal watchdog, and the internal beacon. The housekeeping interrupt controls the state of the Enable LED on the SPHERES control panel based on the status of the controller. It provides a software interface to the beacon control which can be used in any other module.

The interface of the watchdog is of special importance, since the watchdog control signal must be continuously flipped to prevent the watchdog circuitry from forcing a hardware reset. If the housekeeping task does not respond (any task with higher priority does not return control) the system will reset. This function is not performed at the lowest priority because it is possible that metrology or autonomy algorithms which run in background tasks take several seconds, which would cause a hardware reset.

The fact that the housekeeping interrupt controls both the watchdog and the FLASH loader variables is used to provide two functions to the satellites: the ability to force a hardware reset and to enter boot loader mode automatically. To force a hardware reset the

housekeeping thread is commanded to no longer flip the watchdog control line. To enter boot loader mode, the housekeeping thread first writes the boot loader variables with `Enter_boot` set to `0x01`, and then forces a hard reset.

By default the housekeeping task will save the following information once a second to the FLASH:

- Tank time
- Satellite ID (if changed)
- Beacon locations (if changed)
- IMU calibration data (if changed)
- Enter boot (if commanded)

This information is shared by any program since the physical configuration of the satellite (Satellite ID, IMU calibration data) does not change between programs. Further, since the beacon locations are expected to be constant each test session, they only need to be uploaded when the first program is used. By saving the tank time once a second, the housekeeping module maintains a reasonable estimate of tank usage even if the unit is reset.

Source Files

- housekeeping.c
- util_FlashLib.c
- util_BranchTo.c

Internal Header Files

- housekeeping_internal.h
- util_FlashLib.h
- util_timing.h

Public Header Files

- housekeeping.h

G.2.7 GSP Interface

The scientist code interfaces with the other modules through the GSP interface. The other modules include calls to the functions of the GSP module, so that scientist can concentrate all their work in one module. The GSP interfaces with the control and metrology modules, operating as part of the PADS SWI and the Controller SWI. All the modules can post a semaphore to trigger the GSP Task, although the scientist can select which events trigger the task. These interfaces are illustrated in Figure G.26.

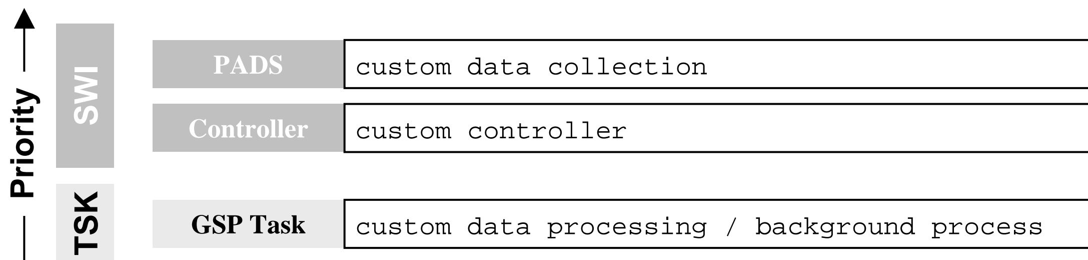

Figure G.26 SCS GSP module threads

The GSP functions available to scientists are:

- **Program Initialization**
  - **gspIdentitySet** - Sets the *logical* identity of the satellite.
  - **gspInitProgram** - Run *once* during boot time, before the DSP/BIOS real-time environment is set, to initialize global variables and perform other functions required before the real-time system is started. These include:
    - Initializing the TDMA windows
    - Allocating the buffers for inertial metrology data
    - Configuring the metrology FPGA for the global metrology setup
    - Setting metrology (inertial and global) rates
    - Setting the background telemetry period
- **Control**
  - **gspInitTest** - This function is run once at the start of each test. The function should initialize any variables used in the GSP control algorithms and set the desired control period for that test.

- **gspControl** - This function implements the control algorithm for the test. It can be programmed directly or it may call other functions developed by the scientist.

- **Metrology**
  - **gspPadsInertial** - This function should collect the data and store it for later processing, since the function is run within the 1kHz PADS SWI. If the scientist can perform quick calculations with the inertial data, this function can also update the system state using the inertial data. If the GSP task takes extended periods of time, this function should provide structures to save data even if the task is not available.
  - **gspPadsGlobal** - This function should only collect the global metrology data and store it for later processing because it is run within the 1kHz PADS SWI. If the GSP task takes extended periods of time, this function should provide structures to save data even if the task is not available.

- **Task**
  - **gspInitTask** - This function executes once when DSP/BIOS sets up the real-time environment, including the GSP Task thread (i.e., it runs shortly after `main()` terminates). The function should initialize any variables needed by the task, as well as setup the masks for events that will trigger the task.
  - **gspTaskRun** - This function executes in the GSP Task thread when an event occurs that posts a semaphore. The possible events which can start this task are:
    - `CTRL_DONE_TRIG` - a controller period is done
    - `DATA_TX_DONE_TRIG` - datacomm has finished transmitting data
    - `DATA_RX_DONE_TRIG` - datacomm has finished assembling data
    - `PADS_GLOBAL_START_TRIG` - an IR pulse was received
    - `PADS_GLOBAL_BEACON_TRIG` - global metrology data received
    - `PADS_INERTIAL_TRIG` - inertial metrology data received
    - `PADS_ESTIMATOR_DONE_TRIG` - the default estimator is done
    - `TASK_TIME_TRIG` - the task sleep period is over
    - `TEST_START_TRIG` - a test was started
    - `GSP_USER_TRIG` - user trigger

    Scientists can select which events will trigger the task by setting a mask at during the task initialization.

> Since all of the events share the same task, the scientists must carefully use this resource and realize that the task may not respond to an event in real-time. For example, if the task is used with a custom estimator, and it is also used to perform post-control data processing, the post-control functions may not be carried in real time. Further, the task has a maximum semaphore depth of 16, therefore no more than 16 events will be accounted for and new events are lost.

Scientists can use any of the functions defined in the *public* header files of the other modules. These provide scientists with full knowledge of the state of the satellite and interfaces to all the hardware.

G.3 Standard Libraries

The following section describe the available standard functions at the time of print of this thesis. These functions can be used by scientists to implement simple algorithms outside their area of interest so that they may concentrate on development of their program. They also serve as baseline algorithms for scientists to compare their results.

G.3.1 Controllers

- **Angular control** - 3D controllers to apply proportional/derivative (PD) control of angular position and control with respect to the global metrology frame with the option to specify the gains.
- **Position control** - 3D controllers to apply proportional/derivative (PD) and proportional/integral/derivative (PID) control laws to the position and velocity of the satellites with respect to the global metrology frame with the option to specify the gains.

  Also provides a function to transform delta-V commands into x-y-z body forces, which can then be transformed to thruster on/off times by the mixers.
- **Switchline control** - 3D switchline controllers for position and attitude control. Two versions are available: one with coupled position and attitude and one with decouple position and attitude.

G.3.2 Estimators

- **Extended Kalman Filter** - An extended Kalman filter which utilizes both inertial and global data to estimate the position and attitude of the satellite in the global frame.
- **Range and bearing** - An extended Kalman filter to obtain the range and bearing between two satellites which use their internal beacons for global metrology. This provides relative state information in multi-satellite systems.

G.3.3 Maneuvers

- **Regulation** - Regulates a constant rotation about one axis while maintaining the satellite in the same position.
- **Open Loop Translation and Rotation** - returns the necessary thruster on/off times for an open loop translation or rotation with respect to the body frame.

G.3.4 Mixers

- **Standard Mixer** - Creates a set of thruster on/off times based on input force and torque parameters, duty cycle, and control period.
- **Mixer with Thruster Correction** - Enhances the standard mixer by correcting for differences in the actual measured thrust of each thruster.

G.3.5 Terminators

- **Timed terminators** - Terminators to end a maneuver or test in a specified period of time after they start.

G.3.6 Math

- **Matrix and Vectors Manipulation Methods**
  - Square of a matrix (B = A * A)
  - Matrix times vector (c = A * b)
  - Matrix times matrix (C = A * B)
  - Matrix time matrix transpose (C = A * B') and (C = A' * B)
  - Matrix add (C = A + B)
  - Vector add (c = a + b)
  - Vector outer product (c = a * b)
  - Vector inner product (c = a' * b)
  - Calculate skew symmetric matrix of A
  - Normalize a vector
  - Calculate the magnitude of a vector
  - Invert a 3x3 matrix
  - Determination of the body to global rotation matrix
  - Determination of the rotation matrix for use with quaternions
- **Matrix Inversion Methods** - provides several methods to invert matrices, including: Cholesky decomposition and LU decomposition.
- **Jacobi** - Computes the eigenvalues and eigenvectors of a matrix.
- **LTI Filter** - Implements a causal form II LTI filter.

G.3.7 Utilities

- **Data compression** - Provides utilities to collect large amounts of data from either the Global or the Inertial metrology system and compress the data before downloading it through the communications module.
- **Serial print** - Enables scientists to transmit serial data through the expansion port (in the satellites) or the simulation debug file (in the simulation) without having to use the standard ANSI C `printf` function which requires substantial memory space.