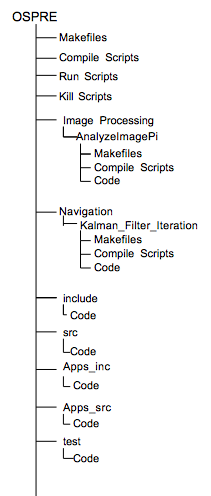
**OSPRE Scripts, OSPRE makefiles, OSPRE Config Files, OSPRE Test Mode vs OSPRE Live Mode, OSPRE SIM Test Directories**

Seth Zegelstein

4/25/17

**OSPRE Directory Structure:**

****

**Makefiles:**

OSPRE/makefile

**Description:**

This makefile is responsible for making the main OSPRE Servers.

OSPRE/makeTestScripts

**Description:**

This makefile is responsible for making the OSPRE test scripts.

OSPRE/ImageProcessing/AnalyzeImagePi/analyzeImage\_rtw.mk

**Description:**

This makefile is responsible for making the Image Processing Library.

OSPRE/GNC/KalmanFilterIteration/Kalman\_Filter\_Iteration\_rtw.mk

**Description:**

This makefile is responsible for making the Navigation library.

**Scripts:**

OSPRE/KalmanFilterIteration/makeGNCLib.sh

**Description:**

This bash script is responsible for making, cleaning and placing the GNC library in the OSPRE/lib directory. This script uses relative paths, but can be called from any directory and function properly.

OSPRE/ImageProcessing/AnalyzeImagePi/makeImageLib.sh

**Description:**

This bash script is responsible for making, cleaning and placing the Image Processing library in the OSPRE/lib directory. This script uses relative paths, but can be called from any directory and function properly.

OSPRE/makeAll.sh

**Description:**

This script is responsible for making all the OSPRE Servers. It places them in the location OSPRE/bin.

OSPRE/makeAllSOM.sh

**Description:**

This script is responsible for making all the OSPRE Servers on the SOM. It places them in the location OSPRE/bin. It differs from makeAll.sh because makeAll.sh uses the –j option (multiple core compile) and this will use up all the memory in the SOM causing a compile error.

OSPRE/makeAllTestScripts.sh

**Description:**

This script is responsible for making all the OSPRE test scripts. It places them in the location OSPRE/bin\_test.

OSPRE/makeAllTestScriptsSOM.sh

**Description:**

This script is responsible for making all the OSPRE test scripts on the SOM. It places them in the location OSPRE/bin\_test. It differs from makeAllTestScripts.sh because makeAll.sh uses the –j option (multiple core compile) and this will use up all the memory in the SOM causing a compile error.

OSPRE/runAll.sh

**Description:**

This bash script runs and backgrounds all the OSPRE Servers (6 total). This script uses relative paths, but can be called from any directory and function properly.

OSPRE/runAllTestScripts.sh

**Description:**

This bash script runs all the OSPRE test scripts. This script uses relative paths, but can be called from any directory and function properly.

**Configuration Files:**

OSPRE/TextData/Camera\_Controller\_Config.txt

**Description:**

This file is read in by CameraController process and will house the specific details of the Camera. Currently the only the Camera’s pixel per degree in the x and y directions exists within this file. When OSPRE gains Camera Drivers, this configuration file will need additional information.

OSPRE/TextData/GNC\_Config.txt

**Description:**

This file is read in by the GNC process and holds specific details of the estimated Trajectory. This file contains an initial position estimate of where OSPRE will turn on, and the distances away from earth that OSPRE will switch from Earth Ranging to Angles, and Angles to Moon Ranging.

OSPRE/TextData/OSPRE\_Server\_Config.txt

**Description:**

All OSPRE Servers will read this file to determine whether OSPRE is in test or live mode and a corresponding test directory if in test mode.

OSPRE/TextData/Skyfire\_J2000\_7\_ECI\_Epsecs.txt

**Description:**

The GNC process will read this file in. It contains the reference trajectory generated by STK which is needed by the Kalman filter. The reference trajectory is trajectory specific and will need to be updated to upon the use of a new trajectory.

OSPRE/\*TEST\_DIRECTORY\_HERE\*/Test\_Data/KalmanFilter\_InitialState.txt

**Description:**

The GNC process will read this file in test mode. This file contains the initial state of a Kalman filter that has been simulated in software up to a certain point in the trajectory. This allows one image to be read in, analyzed and have a solution computed.

OSPRE/\*TEST\_DIRECTORY\_HERE\*/Test\_Data/Satellite\_Data.txt

**Description:**

The Spacecraft process will read this file in during test mode. This file contains all of OSPRE’s inputs from the spacecraft that correspond order wise to the images listed in OSPRE/\*TEST\_DIRECTORY\_HERE\*/Test\_Images/ Image\_Order.txt.

OSPRE/\*TEST\_DIRECTORY\_HERE\*/Test\_Data/TruthData.txt

**Description:**

This file is currently unused.

OSPRE/\*TEST\_DIRECTORY\_HERE\*/Test\_Images/ Image\_Order.txt

**Description:**

The CameraController process will read this file in during test mode. This file contains all of the image names that OSPRE is to analyze ordered top to bottom. The image order corresponds to the satellite data in file OSPRE/\*TEST\_DIRECTORY\_HERE\*/Test\_Data/Satellite\_Data.txt.

**OSPRE Test vs Live Mode:**

**OSPRE Test Mode:**

OSPRE Test mode was developed to work with the OSPRE test setup. OSPRE test mode is synonymous with OSPRE SIM mode. The OSPRE test setup is a physical test that was built to validate that the solutions OSPRE outputs are accurate. This test setup provides 40 images at the same location on the trans lunar trajectory.

**OSPRE Live Mode**

OSPRE was originally developed for live mode, which is what would be used during the mission. OSPRE test mode became the focus of our senior project after taking a off ramp to not use the built in camera due to lack of software drivers, and not having enough time to develop SPI software capability. OSPRE live mode was never fully implemented because it became de-scoped from our project. In the following documentation, I will state where known future changes need to occur to get OSPRE live mode functioning.

**OSPRE Test Directories:**

**Test Directory Description:**

The OSPRE Software in TEST mode reads inputs and outputs results files to the pre-configured test directory. All test directories must have the same predefined structure in order to work properly. This structure is given as follows:

OSPRE/OSPRE\_FULL\_MOON\_TEST\_DIR

**Description:**

Contains images from the full moon test setup.

OSPRE/OSPRE\_GIBBOUS\_MOON\_TEST\_DIR

**Description:**

Contains images from the gibbous moon test setup.

OSPRE/OSPRE\_HALF\_MOON\_TEST\_DIR

**Description:**

Contains images from the half moon test setup.

OSPRE/OSPRE\_SMALL\_CRESENT\_MOON\_TEST\_DIR

**Description:**

Contains images from the small cresent moon test setup.

OSPRE/ OSPRE\_TILTED\_CRESENT\_MOON\_TEST\_DIR

**Description:**

Contains images from the tilted cresent moon test setup.

**OSPRE Components**

Seth Zegelstein

4/25/17

**Class Name:**

Selector

**File Paths:**

OSPRE/include/Components/ Selector.h

OSPRE/src/Components/ Selector.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

* Complete
* Tested through use

**Future Development:**

* Consider using UNIX/LINUX poll() instead of select().
* If continued use of select(), consider an optimization to set the MAX\_FDS to the minimum required value.

**Description:**

This Selector class is our implementation of the JAVA Selector class. Every OSPRE Server application has a Selector instance. The Selector controls the event loop of an Application. Selector is a wrapper around the UNIX/LINUX select() function. The UNIX/LINUX select() function blocks until a registered file descriptor has an event, or a time out occurs. Selector relies on Service object to handle the I/O on each socket and manage what events select() is waiting on for each socket. A Service manages an open connection (file descriptor) to a client. Each Service has a reference to the Application’s Selector. Services register themselves with the Selector instance to be interested in read and/or write events. When a read or write event occurs, the Selector calls the registered Service’s callback in order to handle the event. If multiple events occur, the Selector will handle them all. If there is a read, and write event on the same file descriptor, both handler methods will get called. In the instance of a timeout, the Selector’s select() method returns a 0, to allow the application to handle timeouts.

**Members:**

**Private:**

Service \*services[FD\_SETSIZE];

* An array holding pointers to the Services registered with the Selector object. Maximum size, FD\_SETSIZE is the maximum number of file descriptors that the UNIX/LINUX select() can monitor.

fd\_set readFds, writeFds;

* File descriptor masks for read and write events

fd\_set tempReadFds, tempWriteFds;

* Temporary file description masks for read and write events passed to select() that gets reinitialize every time prior to entering the UNIX/LINUX select().

**Constructors:**

Selector();

* Initializes file descriptors and Service array to NULL.

**Methods:**

Public void registerService(int fd, Service \*srv);

* Allows a Service to register itself with the Selector.

Public void unregisterService(int fd);

* Allows a Service to unregister itself with the Selector.

Public void interestInRead(int fd);

* Allows a Service to register its interest in read events

Public void interestInWrite(int fd);

* Allows a Service to register its interest in read Write

Public void noInterestInRead(int fd);

* Allows a Service to unregister itself for read events

Public void noInterestInWrite(int fd);

* Allows a Service to unregister itself for write events

Public int select(timeval \*timeout);

* This is the main function of the Selector class that holds the UNIX/LINUX select function call. Inputs a pointer timeval structure which determines the timeout time for UNIX/LINUX select(). Returns the number of events that have occurred. Returns 0 on timeout. Returns -1 on error, otherwise returns the number of events processed.

Public void printFds(fd\_set\* set);

* Prints off the set file descriptors for debugging

**Class Name:**

Server

**File Paths:**

OSPRE/include/Components/Server.h

OSPRE/src/Components/Server.cpp

**Super Classes:**

None

**Subclasses:**

ServerInternal

**Status:**

* Complete
* Tested through use

**Future Development:**

May want to additional core server application functionality that gets passed onto all subclasses.

**Description:**

The Server class is an abstract class containing basic functionality for all OSPRE Servers (a.k.a. Applications). All Servers contain a Selector, signal handling, log files, a static pointer back to the Server and a run method. The Server class also contains a method to read in an OSPRE configuration file to determine if OSPRE is in test or live mode. The Server run method holds the main loop calling the Selector’s select() method. An application is allowed to only have one instance of Server object.

**Members:**

**Protected:**

timeval t\_val;

* The timeout passed into select()

FILE\* logFile;

* The Servers log file

static Server\* appl;

* A pointer back to the application class

bool liveMode;

* True if OSPRE in Live mode, False if OSPRE is in test mode

std::string testDIR;

* Test Directory for OSPRE while in test mode, empty string otherwise.

**Private:**

Selector sel;

* The Selector to be used by the server application

int flushTime;

* The log file is written using buffered writes. Every flushTime seconds the log gets flush()’ed to disk. As this implies there is a trade off between performance and a log entry being guaranteed to make it to disk.

**Constructors:**

Server();

* Ignores SIG\_PIPE’s so that application will not exit if it attempts to write on a broken socket. Determining a socket has been closed by the remote party is done via receiving a read notification from select() and no data being able to be read().
* Initializes timeout structure to default value of 5 seconds. This can be override by the derived class.

**Methods:**

Public int run();

* Contains the main loop which calls Selector’s select() method and handles timeouts until the server exits

Public Selector& getSelector();

* Returns a reference to the Server’s Selector

Public friend void signalHandler(int);

* Gracefully handles SIG\_TERMs

Public static Server\* getAppl();

* Returns a pointer to the Server object

Public FILE\* getLogFileID()

* Returns a pointer to the log file

Protected static void setAppl(Server\* srv);

* Allows derived Servers to set the application pointer to themselves. All derived classes must do this for the application to function correctly.

Protected virtual void handleTimeout() = 0;

* All derived Servers must implement a timeout handler. This function gets called in between successive calls to select(). There method is called if the Selector object times out or if the Selector handled one more socket events. It is up to the derived class to determine whether or not timeouts or wait for a subsequent call before taking action and what actions to take.

Protected void setTimeoutTime(int sec, int micro);

* Allows Derived Server’s to customize Selector timeout

Protected void flushLog();

* Attempts to get Server to Flush Log, Server will only flush log once every 10 seconds for efficiency

Protected void readOSPREServerConfigFile();

* Reads in the OSPRE Server Configuration file located in OSPRE/Text\_Data. This file path is hard coded, so changing will break all OSPRE Servers.

**Class Name:**

ServerInternal

**File Paths:**

OSPRE/include/Components/ServerInternal.h

OSPRE/src/Components/ServerInternal.cpp

**Super Classes:**

Server

**Subclasses:**

WatchDog

ScComms

CameraController

ImageProcessor

GNC

**Status:**

* Complete
* Tested through use

**Future Development:**

Currently all ImageProcessor and CameraController Services have huge buffers capable of receiving 5 13MP Image Messages. Ideally, we only need 2 of these Huge Buffers. There is ample memory on the current SOM for this unnecessary use of memory, but efficiency can be improved.

**Description:**

The abstract ServerInternal class is meant to hold shared functionality that is common between all server applications. This includes having an Acceptor to accept connections, an array of Services to manage multiple client connections, a message handle callback function to handle OSPRE messaging, and virtual message handlers for all internal messages. Every ServerInternal subclass is required to provide a handler for all message types including the message types that they do not except to receive. The ServerInternal class was developed to take message handling from the Service layer and move it to the application layer.

**Members:**

**Public:**

enum { MaxClients = 20 };

* The compiled constant MaxClients defines the maximum amount of connections a ServerInternal can manage.

ProcessID p\_ID;

* The process ID of the application.

static ProcessID p\_ID\_Static;

* The process ID of the application needed for static methods.

**Protected:**

Acceptor accept;

std::string hostName;

* Hostname that the Acceptor is accepting connections on.

int localPort;

* Port number to open the Acceptor on

static ServiceInternal \*connections[MaxClients];

* Array to hold all Services

**Constructors:**

ServerInternal(std::string hostName, int localPort, ProcessID p\_ID);

**Methods:**

Public static void handleConnectionRequest(int fd);

* This is the method that the Acceptor will use as a callback for all ServerInternals when they get a connection request.

Public static void handleMessage(Message\* msg, ServiceInternal\* service);

* This is the method is the callback that the ServerInternal registers with the ServiceInternal to handle messages at the application level.

Public virtual void open() = 0;

* Abstract method that states all derived classes must have an open call.

Public bool connectToAppl(std::string host, int port, ServiceInternal\*\* service);

* This is method allows classes derived from ServerInternal to connect to other Applications.

Public virtual void handleCaptureImageRequest(CaptureImageRequest\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleDataMessage(DataMessage\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleImageAdjustment(ImageAdjustment\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleImageMessage(ImageMessage\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleOSPREStatus(OSPREStatus\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handlePointingRequest(PointingRequest\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleProcessHealthAndStatusRequest(ProcessHealthAndStatusRequest\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleProcessHealthAndStatusResponse(ProcessHealthAndStatusResponse\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleSolutionMessage(SolutionMessage\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

Public virtual void handleProcessedImageMessage(ProcessedImageMessage\* msg, ServiceInternal\* service) = 0;

* Abstract method that states all derived classes must handle this type of message.

**Class Name:**

Service

**File Paths:**

OSPRE/include/Components/Service.h

OSPRE/src/Components/Service.cpp

**Super Classes:**

None

**Subclasses:**

ServiceInternal

ServiceExternal

Acceptor

**Status:**

* Complete
* Tested through use

**Future Development:**

**Description:**

A Service is an abstract wrapper class around a UNIX/LINUX TCP/IP socket. An application contains one Service for every connection it has. A Service fully encapsulates the TCP/IP networking from the application. Services are capable of handling reading and writing from the socket, determine if the socket is connected and close the connection. The Service class also contains the methods to connect to a Server and to open a server socket. The Service class has two derived classes, ServiceInternal to handle all internal OSPRE communication using TCP/IP communication and ServiceExternal to communicate through SPI protocol.

**Members:**

**Private:**

Selector &sel;

* A reference to the Servers selector

**Constructors:**

Service(Selector &sharedSel);

**Methods:**

public virtual void handleRead() = 0;

* All Services must be capable of reading information.

public virtual void handleWrite() = 0;

* All Services must be capable of writing information.

public virtual bool isConnected() = 0;

* All Services must be capable of determining if their connection is valid. Only applies to connection-based protocols, (TCP/IP not SPI).

public virtual void closeConnection() = 0;

* Services must be able to end connections. Only applies to connection-based protocols, (TCP/IP not SPI).

public Selector& getSelector();

* A getter method for the private Selector reference.

public int openServerSocket(int portNumber);

* The method used to open a ServerSocket on a specific port number. Used by the Acceptor Service to create a server-side server socket for accepting connection requests.

public static int connectToServer(const char \*serverHosts, int serverPort);

* This method allows Services/Servers to innate a client side TCP/IP connection.

**Class Name:**

ServiceExternal

**File Paths:**

OSPRE/include/Components/ ServiceExternal.h

OSPRE/src/Components/ ServiceExternal.cpp

**Super Classes:**

Service

**Subclasses:**

None

**Status:**

* Needs Future Development
  + Temporarily works
* Tested through use

**Future Development:**

Need to switch the TCP/IP communication for SPI communication between ScComms and Spacecraft processes. Right now this ServiceExternal is using TCP/IP networking which works for communication over a network connection (Copied from ServiceInternal). Once SPI software drivers are developed for OSPRE SOM and Zedboard, they should be put into this class.

**Description:**

ServiceExternal was developed to hold the communication between the Spacecraft process and the ScComms process. TCP/IP networking code was put here temporarily for the ScComms process to talk to the Spacecraft process until SPI drivers got developed. Due to time constraints, the SPI drivers did not get developed. All of the implemented methods in this class need to be erased and redone in order to support the SPI connection. I will discuss the members, methods of this class in the ServiceInternal class documentation section beneath this. \*\*\* Currently, the ScComms and Spacecraft processes communicate through the Space Packet Protocal interface. \*\*\*

**Class Name:**

ServiceInternal

**File Paths:**

OSPRE/include/Components/ ServiceInternal.h

OSPRE/src/Components/ ServiceInternal.cpp

**Super Classes:**

Service

**Subclasses:**

None

**Status:**

* Complete
* Tested through use

**Future Development:**

Add more robustness.

**Description:**

**Members:**

**Private:**

int fd;

* The file descriptor associated with the Service’s socket.

ByteBuffer readbuf;

* A ByteBuffer to store bytes read from the socket.

ByteBuffer writebuf;

* A ByteBuffer to store bytes that will be written to the socket.

Builder build;

* Builds messages into a stream of Bytes and places them onto the writebuf.

Parser parse;

* Parses a stream of bytes from the readbuf and creates a message object.

bool partialMessage;

* Used to determine if a partial message is on the readbuf.

void (\*messageCallBack)(Message\*, ServiceInternal\*);

* The function pointer that is called when complete messages are received.

**Constructors:**

ServiceInternal(Selector& sel, int fd = -1, int buffSize = 1024\*1024);

* Initializes a ServiceInternal instance with read and write buffer sizes of 1024\*1024 unless otherwise specified.

**Methods:**

public ~ServiceInternal();

* Destructor

public bool open(int fd);

* Opens the Service by registering the socket with interest in read events with the Selector. Used for server-side accepted connections.

public bool open(std::string hostName, int portNumber);

* Attempts to connect to the Server with the specified hostname and port number. Upon successful connection, opens the Service by registering the socket with interest in read events with the Selector. Used for client-side connections.

public void registerCallback(void (\*messageCallBackFunc)(Message\*, ServiceInternal\*));

* Allows a Server to register a callback to handle received message objects.

public virtual void handleRead();

* Called by the Selector when a read event occurs on the socket. This method reads in the bytes from the socket and puts them in the read ByteBuffer. It then attempts to parse the messages. If the messages are complete, it calls the registered callback function.

public virtual void handleWrite();

* Called by the Selector when a write event occurs on the socket. This method attempts to write all of the contents in the writebuf to the socket. When all the contents of the writebuf have been written to the socket, this method unregisters its interest in writing with the Selector.

public virtual bool isConnected();

* This method determines whether the socket is still connected to the other side.

public virtual void closeConnection();

* This method closes a connection.

public void sendMessage(Message\* msg);

* This method calls the Builder to build a Message object to the writebuf and registers interest in writing with the Selector.

**Class Name:**

Acceptor

**File Paths:**

OSPRE/include/Components/Acceptor.h

OSPRE/src/Components/Acceptor.cpp

**Super Classes:**

Service

**Subclasses:**

None

**Status:**

Complete, Tested and Operating Nominally

**Future Development:**

Can use more robustness in Acceptor methods (Check for NULL, throw more exceptions).

**Description:**

An Acceptor is an implementation of generic Service that handles client connection requests. The Acceptor class is a wrapper class around a server socket and UNIX/LINUX accept() function. The Acceptor is responsible for the house keeping around handling connections and calls a previously registered callback function that the application wants called whenever a client attempts to connect. The application is responsible for deciding which connections to accept and which connections to reject. Once the callback function is called, the application takes ownership of the newly created socket. An application can have zero or more Acceptor instances each accepting connections on a different port. If a server is dealing with many different types of clients, a new Acceptor should be made for each type of client.

**Members:**

**Private:**

int port;

* The port number that the Acceptor opens a Server Socket on.

std::string hostName;

* The host interface on the machine that the Acceptor is listening on. Note that this data member is not currently used. Expect it to be used to support servers with multiple network interfaces.

int fd;

* The file descriptor for the Server Socket

void (\*callBack)(int);

* A function pointer to the callback function that must be registered with the Acceptor

**Constructors:**

Acceptor(Selector& sel)

* An Acceptor is a Service, and all Services have a reference to a Selector so that they can register interests in read events. This constructor initializes members to their empty state.

**Methods:**

Public bool open(std::string hostName, int portNumber)

* This method must be called before an Acceptor can be used. This method attempts to open the server socket, and register it with the Selector if opened properly. If opened improperly, this method throws a const char\* exception containing a string on why it failed.

Public void registerCallback(void (\*callBackFunc)(int))

* This method must be called before an Acceptor can be used. This method registers the Acceptor callback function. The callback function sends the file descriptor of an accepted connection to the server so it can handle the connection as it pleases.

Public virtual void handleRead()

* The Selector calls this method when a read event occurs on the server socket that the Acceptor owns. The acceptor attempts to accept the connection, sets the accepted socket to non-blocking and calls the callback with the new connected socket file descriptor.

Public virtual void handleWrite()

* This method should never be called. It throws a const char\* exception if called.

Public virtual bool isConnected()

* This method returns true if the Acceptors Server Socket is connected and false if it is not.

Public virtual void closeConnection()

* This method closes the Acceptors Server Socket.

**Class Name:**

ByteBuffer

**File Paths:**

OSPRE/include/Components/ ByteBuffer.h

OSPRE/src/Components/ByteBuffer.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

* Complete
* Tested with test script OSPRE/test/testByteBuffer.cpp

**Future Development:**

Can make more fool proof by adding read mode and write mode, and not allowing certain actions to be executed in a certain mode. Stop the ByteBuffer from breaking encapsulation (potentially Friend Functions, or a public Read and Write method in the ByteBuffer that takes a file descriptor)

**Description:**

The ByteBuffer is a lightweight FIFO wrapper class around a fixed size char\* array. The ByteBuffer class is designed to be used with sending and receiving messages. The ByteBuffer class has two modes, an insert mode and an extract mode. It is currently the programmer’s responsibility to keep track of which mode that the ByteBuffer is in. The ByteBuffer breaks encapsulation by providing the pointer to the buffers current position so that Services can perform read and write commands to sockets using the ByteBuffers current position pointer. The ByteBuffer is always assumed to be in insert mode (ready to have more bytes inserted into it). After a insert is performed from reading from a socket, the positionRead() method must be called to tell the ByteBuffer how much was put into it. In order to get it into extract mode (ready to have the first byte removed), the flip method must be called. Now, one can take out the messages in the order that they came in. The amountWritten method has to be called in the case of writing the ByteBuffer to a socket. In order to get the ByteBuffer back into insert mode, the compact method followed by the flip method must be called.

**Members:**

**Private:**

char\* buf;

* A pointer to the beginning of the buffer, Never changed to avoid Memory Leaks.

char\* currentPos;

* A pointer to the next byte to insert or exctract

int size;

* Total bytes being occupied in the ByteBuffer

int capacity;

* How much memory (bytes) has been allocated to the buffer

**Constructors:**

Public ByteBuffer(int capacity);

* Initializes Members and allocates memory

**Methods:**

Public void clear();

* Clears the buffer

Public void compact();

* Shifts the left over messages in the buffer to the beginning of the buffer

Public void flip();

* Moves the current position pointer from the tail of the buffer to the head of the buffer

Public int used()

* Returns the amount of items in the buffer

Public int remaining()

* Returns the amount of bytes remaining in the buffer

Public int position();

* Returns the integer position of the currentPosition pointer

Public void positionWrite(int length);

* Adjusts the size and currentPosition pointer after a non-blocking write command off of the currentPosition pointer. Throws constant char\* invalid input exception.

Public void positionRead(int length);

* Adjusts the size and currentPosition pointer after a non-blocking read command off of the currentPosition pointer. Throws constant char\* invalid input exception.

Public void rewind(int length);

* Rewinds the buffer, used in dealing with partial messages. Throws several constant char\* exceptions.

Public char get();

* Retrieve a character from the buffer. Throws constant char\* when there is no object in ByteBuffer.

Public void put(char c);

* Put a character on the buffer. Throws constant char\* exception when ByteBuffer is full.

Public int getInt();

* Retrieve an int from the buffer. Throws constant char\* when there is no object in ByteBuffer.

Public void putInt(int i);

* Put an int on the buffer. Throws constant char\* exception when ByteBuffer is full.

Public long getLong();

* Retrieve a long from the buffer. Throws constant char\* when there is no object in ByteBuffer.

Public void putLong(long d);

* Put a long on the buffer. Throws constant char\* exception when ByteBuffer is full.

Public double getDouble();

* Retrieve a double from the buffer. Throws constant char\* when there is no object in ByteBuffer. Only takes 6 digits passed the decimal, limits very large doubles

Public void putDouble(double d);

* Put a double on the buffer. Throws constant char\* exception when ByteBuffer is full. Only takes 6 digits passed the decimal, limits very large doubles

Public char\* getBuffer()

* Returns the currentPosition pointer (Breaks Encapsulation)

Public void put(char\* c, int length);

* Put a character array on the buffer. Throws constant char\* exception when ByteBuffer is full.

Public void get(char\* location, int length);

* Copies a character array on the buffer to a new location. Throws constant char\* when there is no object in ByteBuffer.

Public void printBuffer();

* Prints the contents of the ByteBuffer in hex for debugging

**Class Name:**

Builder

**File Paths:**

OSPRE/include/Components/Builder.h

OSPRE/src/Components/Builder.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

* Complete
* Tested through use

**Future Development:**

The Builder needs a new method for every new internal message that is going to be sent.

**Description:**

The Builder class is meant to take all internal message objects and flatten them into a stream of bytes stored in a ByteBuffer. The Builder class goes along with the Parser class that takes a stream of bytes and turns it into a message class. The Builder has a unique method in order to build all of the internal messages into a stream of bytes. The Builder relies on the public interface of each Message object only.

**Members:**

**Private:**

ByteBuffer& buf;

* The ByteBuffer that the Builder is working with.

**Constructors:**

Public Builder(ByteBuffer &bufParam);

* Sets private member buf

**Methods:**

Public void buildCaptureImageRequest(CaptureImageRequest &msg);

* Flattens a CaptureImageRequest class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildDataMessage(DataMessage &msg);

* Flattens a DataMessage class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildImageAdjustment(ImageAdjustment &msg);

* Flattens a ImageAdjustment class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildImageMessage(ImageMessage &msg);

* Flattens a ImageMessage class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildOSPREStatus(OSPREStatus &msg);

* Flattens a OSPREStatus class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildPointingRequest(PointingRequest &msg);

* Flattens a PointingRequest class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildProcessHealthAndStatusRequest(ProcessHealthAndStatusRequest &msg);

* Flattens a ProcessHealthAndStatusRequest class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildProcessHealthAndStatusResponse(ProcessHealthAndStatusResponse &msg);

* Flattens a ProcessHealthAndStatusResponse class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildSolutionMessage(SolutionMessage &msg);

* Flattens a SolutionMessage class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Public void buildProcessedImageMessage(ProcessedImageMessage &msg);

* Flattens a ProcessedImageMessage class and puts the byte stream into the ByteBuffer. Throws const char\* exception when there is not enough space left in buffer to build message.

Private void createHeader(int length, MessageID msgID, time\_t timeStamp)

* Creates a message Header containing the message length in bytes, the message ID and the timestamp of the message.

**Class Name:**

Parser

**File Paths:**

OSPRE/include/Components/Parser.h

OSPRE/src/Components/Parser.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

* Complete
* Tested through use

**Future Development:**

The Parser can be upgraded to be more robust. Currently the Parser will break if an invalid message is received. The switch statement in the parseMessage() function can be switched to a callback.

**Description:**

The Parser creates message objects from a stream of bytes. This Parser is only capable of parsing predefine internal messages. The Parser has a pointer to one of every type of internal messages. Every internal message has the same message header. The parser first Parses the header, checks to make sure the full message has arrived, and then parses the data for the specific type of message declared in the message header. When a specific message type gets received for the first time, the Parser allocates that message object in the heap. When a message gets parsed, the Parser returns a generic message type pointer to the parsed message. This message pointer will be valid until the Parser is called again, at which point the old message may get overwritten. The Parser is capable of handling partial messages.

**Members:**

**Private:**

ByteBuffer &buf;

* The ByteBuffer that the Parser parses messages from

int messageLength;

* Length of current message being parsed

MessageID messageID;

* MessageID of current message being parsed

time\_t timeStamp;

* Timestamp of current message being parsed

CaptureImageRequest\* capture;

* Pointer to CaptureImageRequest message, not initialized until message type received

DataMessage\* data;

* Pointer to DataMessage, not initialized until message type received

ImageAdjustment\* adjustment;

* Pointer to ImageAdjustment message, not initialized until message type received

ImageMessage\* image;

* Pointer to ImageMessage, not initialized until message type received

OSPREStatus\* status;

* Pointer to OSPREStatus message, not initialized until message type received

PointingRequest\* pointing;

* Pointer to PointingRequest message, not initialized until message type received

ProcessHealthAndStatusRequest\* request;

* Pointer to ProcessHealthAndStatusRequest message, not initialized until message type received

ProcessHealthAndStatusResponse\* response;

* Pointer to ProcessHealthAndStatusResponse message, not initialized until message type received

SolutionMessage\* solution;

* Pointer to SolutionMessage, not initialized until message type received

ProcessedImageMessage\* processed;

* Pointer to ProcessedImageMessage, not initialized until message type received

**Constructors:**

Parser(ByteBuffer &bufParam);

* Takes a reference to a ByteBuffer and sets internal member to this.

**Methods:**

Public Message\* parseMessage(bool\* partialMessage);

* This method is called by a Service to parse the contents of a ByteBuffer after it receives a read event. This method sets the bool partialMessage to true if it received a partial message. If there are no messages in the buffer, this method returns NULL. All internal messages are derived from Message. This method will return a pointer to a Message object after successfully parsing a message. This method calls one of the methods below to parse the specific type of internal message. In the event of an error, this message throws a const char\* exception containing an explanation for the error.

Private Message\* parseCaptureImageRequest();

* Parses allocates and parses a CaptureImageRequest message.

Private Message\* parseDataMessage();

* Parses allocates and parses a DataMessage.

Private Message\* parseImageAdjustment();

* Parses allocates and parses a ImageAdjustment message.

Private Message\* parseImageMessage();

* Parses allocates and parses a ImageMessage message.

Private Message\* parseOSPREStatus();

* Parses allocates and parses a OSPREStatus message.

Private Message\* parsePointingRequest();

* Parses allocates and parses a PointingRequest message.

Private Message\* parseProcessHealthAndStatusRequest();

* Parses allocates and parses a ProcessHealthAndStatusRequest message.

Private Message\* parseProcessHealthAndStatusResponse();

* Parses allocates and parses a ProcessHealthAndStatusResponse message.

Private Message\* parseSolutionMessage();

* Parses allocates and parses a SolutionMessage.

Private Message\* parseProcessedImageMessage();

* Parses allocates and parses a ProcessedImageMessage.

**Class Name:**

External\_Builder

**File Paths:**

OSPRE/include/Components/ External\_Builder.h

OSPRE/src/Components/ External\_Builder.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

* Complete
* Tested through use

**Future Development:**

None Yet.

**Description:**

The External\_Builder class works just like the Builder class described above, but it only builds messages that are to be sent externally, between Spacecraft Comms and Spacecraft processes. The External\_Builder also follows Space Packet Protocol as defined in CCSDS 133.0-B-1 published in 2003. The External\_Builder takes an External\_Message, creates a stream of bytes and places it onto its ByteBuffer reference. This class should still function the same when the external protocol communication gets switched to SPI. It will need to be updated to make sure that the messages are getting built with the same implementation of CCSDS that the Spacecraft is using.

**Members:**

**Private:**

ByteBuffer& buf;

* ByteBuffer reference for External\_Builder

unsigned int sequence : 14;

* The sequence number gets incremented every time a message gets built/sent across the channel.

**Constructors:**

External\_Builder(ByteBuffer &bufParam);

**Methods:**

Public void buildExternal\_DataMessage(External\_DataMessage &msg);

* Builds an External\_DataMessage object into a stream of bytes and places it on the associated ByteBuffer.

Public void buildExternal\_OSPREStatus(External\_OSPREStatus &msg);

* Builds an External\_OSPREStatus object into a stream of bytes and places it on the associated ByteBuffer.

Public void buildExternal\_PointingRequest(External\_PointingRequest &msg);

* Builds an External\_PointingRequest object into a stream of bytes and places it on the associated ByteBuffer.

Public void buildExternal\_SolutionMessage(External\_SolutionMessage &msg);

* Builds an External\_SolutionMessage object into a stream of bytes and places it on the associated ByteBuffer.

**Class Name:**

External\_Parser

**File Paths:**

OSPRE/include/Components/ External\_Parser.h

OSPRE/src/Components/ External\_Parser.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

* Complete
* Tested through use

**Future Development:**

None Yet.

**Description:**

The External\_Parser class works just like the Parser class described above, but it only parsers messages that are to be sent externally, between Spacecraft Comms and Spacecraft processes. The External\_Parser also follows Space Packet Protocol as defined in CCSDS 133.0-B-1 published in 2003. The External\_Parser takes a stream of bytes and creates External\_Message objects. It will need to be updated to make sure that the messages are getting built with the same implementation of CCSDS that the Spacecraft is using. This file uses the compile time constant EXTERNAL\_HEADER\_MESSAGE\_SIZE declared in OSPRE/include/Enumerations/ MessageSizes.h.

**Members:**

**Private:**

ByteBuffer &buf;

* A reference to the ByteBuffer holding the stream of bytes

Message\_External messageHeader;

* A placeholder for the message header.

MessageID messageID;

* A placeholder for the messageID

External\_DataMessage\* data;

* A pointer to the External\_Parser External\_DataMessage.

External\_OSPREStatus\* status;

* A pointer to the External\_Parser External\_OSPREStatus.

External\_PointingRequest\* pointing;

* A pointer to the External\_Parser External\_PointingRequest.

External\_SolutionMessage\* solution;

* A pointer to the External\_Parser External\_SolutionMessage.

**Constructors:**

External\_Parser(ByteBuffer &bufParam);

**Methods:**

Public Message\_External\* parseMessage(bool\* partialMessage);

* This method is called by a Service to parse the contents of a ByteBuffer. This method sets the bool partialMessage to true if it received a partial message. If there are no messages in the buffer, this method returns NULL. All internal messages are derived from Message. This method will return a pointer to a Message object after successfully parsing a message. This method calls one of the methods below to parse the specific type of internal message. In the event of an error, this message throws a const char\* exception containing an explanation for the error.

Private Message\_External\* parseExternal\_DataMessage();

Private Message\_External\* parseExternal\_OSPREStatus();

Private Message\_External\* parseExternal\_PointingRequest();

Private Message\_External\* parseExternal\_SolutionMessage();

**Class Name:**

CircularBuffer

**File Paths:**

OSPRE/include/Components/ CircularBuffer.h

OSPRE/src/Components/ CircularBuffer.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

* Complete
* Tested with test script OSPRE/test/testCircularBuffer.cpp

**Future Development:**

Can make more robust in future. Increase robustness around Test mode implementation.

**Description:**

This Circular Buffer is only meant to hold satellite DataMessages. Satellite DataMessages come at a frequency of 1Hz. The idea behind the CircularBuffer is that you make the CircularBuffer big enough to hold all the DataMessages that you may need, but when they go obsolete, or the buffer runs out of room, it re-writes the new DataMessages over the older DataMessage’s.

**Members:**

**Private:**

DataMessage\* bufferHead;

* Pointer kept at head of the buffer to prevent memory leaks.

DataMessage\* insert;

* Pointer pointing at spot to insert the next DataMessage

DataMessage\* nextSimMessage;

* Used for SIM mode only, not very robust.

int buffSize;

* The total number of DataMessages that the buffer can hold.

**Constructors:**

CircularBuffer(int numDataMessage = 6\*60);

* Allocates and initialize member variables. Defualts the total amount of DataMessages to 6 minutes worth of satellite DataMessages.

**Methods:**

public void put(DataMessage\* msg);

* Copies a DataMessage onto the buffer.

public DataMessage\* get(time\_t timeStamp);

* Finds and returns a Pointer to a DataMessage by its satellite timeStamp. Throws an exception if the DataMessage is not found.

public DataMessage\* getNextSimMessage();

* Only used for TEST Mode, returns DataMessage addresses with a non zero timestamp.

public void printBuffer();

* Prints the DataMessages timestamps in the buffer.

**Class Name:**

ImageNameReader

**File Paths:**

OSPRE/include/Components/ ImageNameReader.h

OSPRE/src/Components/ ImageNameReader.cpp

**Super Classes:**

None

**Subclasses:**

None

**Description:**

**Members:**

**Private:**

std::vector<std::string> data;

int count;

**Constructors:**

ImageNameReader();

**Methods:**

Public std::string getNextImageName();

Public void loadImageNames(std::string testDir);

**Class Name:**

SpacecraftDataReader

**File Paths:**

OSPRE/include/Components/ SpacecraftDataReader.h

OSPRE/src/Components/ SpacecraftDataReader.cpp

**Super Classes:**

None

**Subclasses:**

None

**Description:**

**Members:**

**Public:**

**Protected:**

**Private:**

**Constructors:**

**Methods:**

**OSPRE Applications**

Seth Zegelstein

4/25/17

**Class Name:**

WatchDog

**File Paths:**

OSPRE/ Apps\_inc/ WatchDog.h

OSPRE/ Apps\_src / WatchDog.cpp

**Super Classes:**

ServerInternal

**Subclasses:**

None

**Status:**

Complete, Tested and Operating Nominally

**Future Development:**

Uses hard coded hostnames and port numbers. Should switch this to a configuration file.

**Description:**

**Members:**

**Public:**

**Protected:**

**Private:**

**Constructors:**

**Methods:**

**Class Name:**

ScComms

**File Paths:**

OSPRE/ Apps\_inc/ScComms.h

OSPRE/ Apps\_src /ScComms.cpp

**Super Classes:**

ServerInternal

**Subclasses:**

None

**Status:**

Complete, Tested and Operating Nominally

**Future Development:**

Uses hard coded hostnames and port numbers. Should switch this to a configuration file.

**Description:**

**Members:**

**Public:**

**Protected:**

**Private:**

**Constructors:**

**Methods:**

**Class Name:**

GNC

**File Paths:**

OSPRE/ Apps\_inc/GNC.h

OSPRE/ Apps\_src /GNC.cpp

**Super Classes:**

ServerInternal

**Subclasses:**

None

**Status:**

Complete, Tested and Operating Nominally

**Future Development:**

Uses hard coded hostnames and port numbers. Should switch this to a configuration file.

**Description:**

**Members:**

**Public:**

**Protected:**

**Private:**

**Constructors:**

**Methods:**

**Class Name:**

ImageProcessor

**File Paths:**

OSPRE/ Apps\_inc/ImageProcessor.h

OSPRE/ Apps\_src /ImageProcessor.cpp

**Super Classes:**

ServerInternal

**Subclasses:**

None

**Status:**

Complete, Tested and Operating Nominally

**Future Development:**

Uses hard coded hostnames and port numbers. Should switch this to a configuration file.

**Description:**

**Members:**

**Public:**

**Protected:**

**Private:**

**Constructors:**

**Methods:**

**Class Name:**

CameraController

**File Paths:**

OSPRE/ Apps\_inc/ CameraController.h

OSPRE/ Apps\_src / CameraController.cpp

**Super Classes:**

None

**Subclasses:**

None

**Status:**

Complete, Tested and Operating Nominally

**Future Development:**

Uses hard coded hostnames and port numbers. Should switch this to a configuration file.

**Description:**

**Members:**

**Public:**

**Protected:**

**Private:**

**Constructors:**

**Methods:**

**Class Name:**

Spacecraft

**File Paths:**

OSPRE/ Apps\_inc/ Spacecraft.h

OSPRE/ Apps\_src / Spacecraft.cpp

**Super Classes:**

Server

**Subclasses:**

None

**Status:**

Complete, Tested and Operating Nominally

**Future Development:**

Uses hard coded hostnames and port numbers. Should switch this to a configuration file.

**Description:**

**Members:**

**Public:**

**Protected:**

**Private:**

**Constructors:**

**Methods:**

**OSPRE Messages**

Seth Zegelstein

4/25/17

19 Files

OSPRE Structures, Exceptions and Enumerations

Seth Zegelstein

4/25/17

7 Files

OSPRE Test Files

Seth Zegelstein

4/25/17

17 Files