Simulation Development and Analysis Branch

Programmer's Guide

Contract NNL06AA74T

September 21, 2010

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Simulation Development and Analysis Branch

Prepared by:
Unisys Corporation
Hampton, Virginia

Submitted by: _____Date: _____
SAFITS Program Manager, Unisys

Approved by: _____Date: _____
Software Group Lead, SDAB

Concurred by: _____Date: _____
Branch Head, SDAB

Change History

Date	Description
April 21, 1995	1) Initial release.
April 27, 1995	1) Correct quotation marks on item 5 for user-defined header files 2) Rewrite item 1 concerning C++ mode in GNU emacs
October 6, 1995	1) Reformat with book document class 2) Separate implementation rules from style rules and put implementation rules in LaSRS++ Implementation Guide.
June 25, 1996	1) Merge Style Guide and Implementation Guide into Programmer's Guide 2) Enhance source code comment section 3) Add abbreviation, acronym, and symbol expansion section 4) Update instructions for emacs C++ formatting 5) Add debug level descriptions 6) Add assume() preprocessor macro description 7) Update warning and error message descriptions
February 18, 1997	1) Added Acknowledgments section 2) Change all LaSRS++ Architecture Team references to LaSRS++ Configuration Control Board (CCB) references 3) Remove all references to .ipp files (all inline member function definitions now in header files) 4) Added detailed requirements for internal and external include guards 5) Added guidelines for static variables at class scope 6) Added guidelines for global function comments 7) Added subsection detailing ObjectManual comments for classes 8) Added more guidelines concerning C++ preprocessor 9) More detail on default items generated by the compiler 10) Added discussion of object identity to subsection on assignment operators 11) Changed preference on inlined constructors and destructors 12) Added requirement to use new cast operators 13) Added section dedicated to inline functions 14) Added chapter on portability issues

Date	Description
June 30, 1997	1) Abbreviation character limits also apply to acronyms 2) Class naming conventions also apply to struct 3) Remove discussion of external include guards – g++ handles this implicitly provided the internal include guards are of the proper format 4) ObjectManual method signatures currently must be only one line 5) Restore stream formatting flags after use
September 2, 1997	1) Change Style part to Coding Standard part 2) Comment on use of acronyms from vendor supplied code 3) Added detail on use of acronyms in names 4) Do not use suffixes on object, constant, or variable names to indicate that the identifier is a reference or a pointer 5) Added detail on argument alignment for functions 6) Placement of operators on multi-line statements 7) Specify format for endless loops – use while (true); do not use for(;;) or while (1) 8) Added detail on resetting format flags when using streams
August 1, 2000	1) LaSRS++ and projects covered under the same CMP 2) General naming conventions also apply to namespaces 3) Added requirement to take names from requirements documents 4) Changed directory path for acronym list 5) Class and struct naming rules also apply to namespaces 6) Added detail on establishing traceability 7) Mutator arguments should reflect dimensional quantity 8) Use typedefs to improve readability 9) Boolean accessor names should pose a question with a positive sense 10) Specify rule for boolean mutator names 11) Replaced alphabetical with lexicographical 12) Put using declarations after include file declarations in header files 13) Added example of forward declaring a template class 14) Only one namespace declaration per header file 15) Detail on public and protected inline member function definitions 16) Added detail on member comments 17) Added listing of standard pre-processor variables 18) Details on arguments for applicationError() 19) Changed reinitialize to initialize 20) Accessor return types can be by value for intrinsic types 21) Use unsigned variables for loop counters 22) Added detail on inline accessor functions 23) Added detail on inline mutator functions 24) Added detail on inline functions

Date	Description
	25) Added detail on arguments to sqrt() function 26) Added detail on arguments to acos() and asin() functions 27) Added detail on debug code 28) Added chapter ``Namespace Declarations" 29) Added section ``Limiting Namespace Pollution" 30) Added section ``Limiting Include File Coupling"
August 29, 2000	1) Added detail on header file specifics 2) Added detail on expressions 3) Added detail on code reuse 4) Added detail on portability 5) Added detail on memory management 6) Added a section on exception handling 7) Added detail on casting 8) Added a section on typeid() 9) Added a chapter on optimizations
October 6, 2000	1) Added Appendix for preprocessor variables
July 12, 2001	1) Replaced ObjectManual section with Doc++ Section 2) Replaced lost macros with equivalents, LaSRS++ and C++
September 8, 2001	1) Added legal notice to standard LaSRS++ banner
February 15, 2002	1) Added subsection ``Use of Friends" 2) Replaced outdated ``Sample Source Files" with current version of PositionalModel
August 16, 2006	1) Converted to MS Word 2) Added item 6 to section 6.5, Part II 3) Added new subsection 6.4.3 Methods to Part II 4) Split vehicle model identifiers into a separate appendix 5) Renamed to Flight Simulation and Software Branch Programmer’s Guide
September 21, 2010	1) Renamed to Simulation Development and Analysis Branch Programmer’s Guide 2) Changed organization name from FSSB to SDAB 3) Removed the requirement for the “Description” part of the LaSRS++ banner at the top of the code files

Acknowledgments

The LaSRS++ Programmer's Guide is the culmination of the efforts of many people. The original LaSRS++ Architecture Team

Richard A. Leslie
David W. Geyer
P. Sean Kenney
Kevin Cunningham
Michael M. Madden
Patricia C. Glaab

decided from the beginning to create and maintain a clear and consistent set of guidelines for developing code in the LaSRS++ Application Framework. This living document has been further enhanced by contributions from Kenneth S. Graham, Jr., David M. Cooke, and by feedback from the LaSRS++ user community.

Contents

1 Introduction

This document is a collection of style and implementation rules for developing code in the LaSRS++ Application Framework. The rules are based upon the experience of the LaSRS++ Configuration Control Board (CCB) and on rules and recommendations published in the cited reference documents. The rules in this document take precedence over all cited references.

This document shall be consulted for questions concerning coding style and implementation in LaSRS++. The cited documents often provide additional explanations concerning the reasons for a given rule. If these guidelines do not directly address a particular question, the question shall be directed to the LaSRS++ CCB for consideration. New guidelines that result from user feedback will be added to the LaSRS++ Programmer's Guide.

These rules shall be applied as code is written. This will save time in the long run as it will avoid the need to go back and make additional source code changes to match these guidelines. Note that it is a requirement of the LaSRS++ Software Development Plan and the LaSRS++ Configuration Management Plan (CMP) that all code conform to the LaSRS++ Programmer's Guide before it will be accepted into the framework.

It is **required** that these guidelines be used for project specific code as well as LaSRS++ baseline code. Exceptions must be approved by the LaSRS++ CCB.

Exceptions that are granted by the LaSRS++ CCB **must** be documented in the source code in the form of comments.

Part I

Coding Standard

2 Naming Conventions

2.1 General Naming Conventions

1. Choose names that suggest the usage of the variable, class, object, function, or namespace.

2. Remember that names matter more to the reader of the code than to the writer.[6]

3. The dereference operator `'*'` and the address-of operator `'&'` shall immediately follow the *type name* in all declarations and definitions. This stresses that the character `'*'` or `'&'` is part of type definition and not the name.
   For example,

   ```
   int  i;
   int& reference_to_i = i;
   int* j;
   ```

4. Do not use identifiers that begin or end with underscores.

5. Use `bool` variables instead of binary integers that emulate boolean variables.

6. Variables of the `bool` type shall be named with a "positive" sense.
   For example,

   ```
   bool no_hud;      ← incorrect, negative sense
   bool hud_on;      ← correct, positive sense
   ```

7. Where reasonable and appropriate, names should be taken from the vocabulary present in the requirements documents. This aids traceability between the source code and the requirements.

2.2 Abbreviations, Acronyms, and Symbol Expansions

1. Unabbreviated names are always accepted and obviously preferred. In modern computer languages like C++, you have almost no reason to shorten meaningful names.[6] Software should be written as carefully as English prose, with consideration given to the reader as well as to the computer.[5]

2. Abbreviations and acronyms shall **not** be used in names where the full, spelled out name is less than or equal to 20 characters. The number 20 is a subjective choice made by the LaSRS++ CCB. This choice is based on personal experience and discussions found in various texts.

3. Abbreviations are acceptable only if found in the cited Webster's dictionary.[2] Acronyms and symbol expansions (e.g., `q_bar`) are acceptable only if found in the LaSRS++ supplemental name list. The official supplemental name list is in the following file:

/vobs/Simulation/Documents/Process/SupplementalNameList/index.html

4. Each acronym and symbol expansion in the supplemental name list is only valid for its specified definition.

5. When a collection of classes makes use of a well recognized standard, a separate list of acronyms will be maintained in the supplemental name list. These acronyms are only to be used within the context of the aforementioned collection of classes. The ARINC specific list is an example of a separate sublist of acronyms.

6. Additions to the supplemental name list are subject to the approval of the LaSRS++ CCB. Any request for a new acronym or symbol expansion must be submitted to the LaSRS++ CCB chairman in writing via e-mail. The submission **must** contain a generally accepted reference for each acronym or symbol expansion. The reference shall include title, author or editor (if applicable), organization (if applicable), publisher (if applicable), and year.

7. Abbreviations, acronyms, and symbol expansions provided by third-party supplied source code **must** be commented as such at the first point in any file where the acronym is used. This applies to **all** names in the third-party supplied source code.

```
// TPS is a struct defined in "vendor.h"
TPS third_party_struct;
```

typedef aliases can be used to expand third-party abbreviations, acronyms, and symbol expansions into more readable names.

```
typedef TPS ThirdPartyStruct;
```

2.3 Class, Struct and Namespace Names

1. Class, struct and namespace names shall be one or more words with each word beginning with an *uppercase* letter followed by all *lowercase* letters.

2. Words in class, struct and namespace names shall be written together; i.e., no underscores. For example,

```
class LinkedList;
```

3. Acronyms used in class, struct and namespace names shall begin with an *uppercase* letter followed by all *lowercase* letters.
For example,

```
class Hsct;
```

2.4 Object, Constant, and Variable Names

1. Object, constant, and variable names shall be all *lowercase* letters.

2. When an object, constant, or variable name consists of more than one word, subwords shall be separated by underscores.

3. Loop index variables may have simple names; e.g., i, j, or k. If a variable is to be used outside the loop, it shall be given a more meaningful name. [6]

4. Acronyms used in object, constant, and variable name shall be all *lowercase.*

5. Do **not** use prefixes or suffixes on object, constant, or variable names to indicate that the identifier is a reference or a pointer. Names describe the role of objects, not their types. The type of an object is evident from its declaration and its syntactical context.

6. In some circumstances, traceability between program statements and their source requirements (i.e., block diagrams, documents, or re-engineered source code) is important. If names used in the source requirements are poorly constructed by LaSRS++ standards, the developer can use one of the following techniques to establish traceability. If the developer can produce a meaningful name for the variable, then the developer will use the meaningful name and place the poorly worded name and its source in the variable's comment (see Section 2.3).
   For example,

   ```
   // Change in lift due to aileron deflection. Non-dimensional.
   // Represents ``DLA'' in document D210T104: Aerodynamics
   // Data.
   double delta_lift_aileron;
   ```

   If meaningful names cannot be produced, then internal variables (i.e., class attributes and local variables) can use the names from the source requirements to aid traceability. For example, a block diagram may contain a signal named pxy3. Although this name conveys no meaning and, thus, violates LaSRS++ style guide rules, it is important to be able to trace its representation in the source code back to the block diagram. Thus, the class implementing the block diagram is allowed to use pxy3 as the name of the attribute that represents the signal.

7. Argument names for mutators that change a dimensional quantity should reflect the dimensions of the quantity. This technique easily identifies the proper dimensions for data passed through the class interface. For example, a mutator for changing velocity could be declared as follows:

   ```
   putVelocity(double feet_per_second);
   ```

2.5 Typedef Names

1. `typedef` names follow the same naming conventions as class names (see Section 2.3).

2. `typedefs` shall **not** be used to shorten the names of fundamental types. typedefs can be used to add domain specific type information to a fundamental type.

3. Use of `typedefs` is encouraged to improve readability by replacing complex type declarations such as template instantiations and function pointers with simpler, meaningful names (See also section 2.5.6, item 11).

2.6 Enumerated Type Names

1. Enumerated types follow the same naming conventions as classes. The enumerators shall be all *uppercase* with underscores separating subwords. For example,

```
enum VehicleType
{
  HSCT,
  B757_BASE,
  F16A,
  GENERIC_FIGHTER
};
```

2. Reminders:

   (a) The names of enumerators must be distinct from those of ordinary variables and other enumerators in the same scope.

   (b) The integral values of the enumerators can be explicitly specified, although these values do **not** need to be distinct.

2.7 Global and Member Function Names

2.7.1 General

1. Single word function names shall be all *lowercase* letters.

2. When a function name consists of more than one word, the words shall be written together, i.e. no underscores. The first word shall be all *lowercase* letters and all subsequent words shall start with an *uppercase* letter followed by all *lowercase* letters. For example,

```
const Angle& getAlpha() const;   // return angle of attack
```

2.7.2 Accessor Functions

1. Accessor member functions (functions that only return the value of or a constant reference to a private member object) that return a non-`bool` shall have the prefix `get`. (see previous example)

2. Accessor member functions that return a `bool` shall have a name that poses a question with a positive sense.
   For example,

```
bool isButtonPressed() const; // is the button pressed?
bool useSpecialFCS() const;   // use the special flight control system?
```

2.7.3 Mutator Functions

1. Mutator member functions (functions that only set the value of a private member object) that change a non-`bool` shall have the prefix put.

2. Mutator member functions that change a `bool` shall have a name that has the prefix put, set, or unset or that commands a change of state.
   For example,

```
void putSomeFlag(const bool& new_flag); // set the flag T/F

void pressButton();                     // sets button to true
void releaseButton();                   // sets button to false

// set use of the special flight control system
void setSpecialFCS(const bool& new_value);
```

2.8 File Names

1. LaSRS++ file names follow these conventions:

   *.**hpp** files - contain class and global declarations and inline member function definitions
   *.**cpp** files - contain non-inline member function and global definitions
   ***.dpp** files - contain file scope data variables; e.g., aero lookup tables

2. Class names and the files that declare and define the class shall have the same name with the appropriate extension. For example, the class LinkedList is defined by the files: LinkedList.hpp and LinkedList.cpp.

3 Code Structure

3.1 Files

3.1.1 General

1. All source code files must have the standard LaSRS++ banner at the top of the file that provides the file name. The file name **must** be filled out in every file. The standard LaSRS++ banner **must** include the standard government ownership notice.

```
/*************************************************************************
 *                                                                       *
 *      _/                    _/_/_/_/  _/_/_/_/  _/_/_/_/               *
 *     _/                    _/        _/    _/  _/          _/      _/   *
 *    _/          _/_/      _/_/_/_/  _/_/_/_/  _/_/_/_/   _/_/_/  _/_/_/  *
 *   _/         _/    _/          _/  _/    _/  _/          _/      _/   *
 *  _/_/_/_/   _/_/_/_/  _/_/_/_/  _/    _/  _/_/_/_/                     *
 *                                                                       *
 *  File:        SomeClass.hpp                                           *
 *                                                                       *
 *************************************************************************
 *                                                                       *
 *  NOTICE:                                                              *
 *                                                                       *
 *    THIS SOFTWARE MAY BE USED, COPIED AND PROVIDED TO OTHERS ONLY AS    *
 *    PERMITTED UNDER THE TERMS OF THE CONTRACT OR OTHER AGREEMENT UNDER  *
 *    WHICH IT WAS ACQUIRED FROM THE U.S. GOVERNMENT.  NEITHER TITLE TO   *
 *    NOR OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.  THIS NOTICE   *
 *    SHALL REMAIN ON ALL COPIES OF THE SOFTWARE.                         *
 *************************************************************************/
```

2. The standard LaSRS++ inline member function separator comment **must** be used in every class declaration header file, even if there are no inline member functions defined. This comment shall immediately follow the class declaration (see example in Appendix A).

```
/*************************************************************************
 *  Inline member functions                                              *
 *************************************************************************/
```

3. Do not rely on included header files to include other header files that are explicitly needed. If an included header file changes for some reason and stops including a header file that is needed in the implementation file, then the implementation file will not compile.

4. Source files shall be restricted to 80 columns of text.

3.1.2 Header File Specifics

1. Header files shall be included in other header files and in implementation files in the following order:

   (a) system header files
   Note: There may be order dependencies between system header files that must be obeyed.

   (b) user-defined header files in lexicographical order
   Exception: In an implementation file containing class member function definitions, the class header file shall be included *first*, followed by the other user-defined header files in lexicographical order. This ensures that the class header file is included *first* in at least one implementation file. This can expose cases where the given class header file does not directly include all of the header files that it requires.

2. Internal include guards shall be used to prevent the same header file from being included multiple times in a given translation unit. [4] Internal include guards shall surround the contents of each header file (see example in Appendix A) and be of the form:

```
#ifndef SOME_CLASS_HPP
#define SOME_CLASS_HPP
          .
          .
          .
#endif
```

3. The identifier in the internal include guard shall be the filename (no extension) in *uppercase* letters, with the suffix `_HPP`. Multiple word macro names shall have underscores between each word; e.g., `SOME_CLASS_HPP`.

4. Each member function shall be declared on its own line. A comment shall accompany the declaration to clarify the purpose of the function unless the name conveys enough useful information.

5. Use directive `#include "filename.hpp"` for user-defined header files; e.g. LaSRS++ class header files.

6. Use directive `#include <filename.hpp>` for the standard and library header files; e.g. math library header file math.h.

7. Using declarations shall appear immediately after include file declarations.

8. A forward declaration shall be used in place of an included header file if the given type is only accessed through a pointer or a reference.
   For example,

```
template <class T> class Vector;
```

Forward declarations shall appear after include file and using declarations.

9. Do not use explicit pathnames in `#include` directives for user-defined header files. [3] `#include` directives for system header files may require an explicit pathname.

10. There shall be at most *one* class or *one* namespace declaration per header file.

11. Do not place definitions of static data and/or static functions at file scope within a header file. These file scope definitions pollute the global name space and consume data space in every translation unit that includes the header file. [4]

12. When including standard C++ and C library header files, the name in Section 17 of the ANSI/ISO standard shall be used. In other words, all names will appear without the .h extension and most C library headers will be prefixed with 'c', e.g. `<cerrno>`.

3.2 Declarations

3.2.1 Class Declarations

1. The public, protected, and private parts of a class shall be declared in that order (see examples).
   Note: The public, protected, and private parts of a class **must** be specified, even if empty.

2. No constructors, destructors, or member functions are to be defined in a class declaration.

3. Definitions of public and protected inline member functions for a class shall immediately follow the class declaration in the appropriate header file (see examples in Appendix A). Private inline member functions shall be defined in the appropriate implementation file before any non-inlined functions are defined.

4. `static const` integral variables may be defined inside a class declaration when the variable is needed to declare the size of a class-scope array. Otherwise, define *all* `static` class-scope variables (`const` or non-`const`) in the appropriate implementation file.

5. All member function and data declarations in a class must have an appropriate inline comment unless the name conveys enough useful information. Units (properly abbreviated) shall be provided if the variable is dimensional.

6. Once a member function has been declared `virtual` in a class hierarchy, it shall be declared `virtual` in all subsequent derived classes. This is not required by C++, but it adds to the understanding and clarity of the code.

7. Do **not** declare `static` variables inside member functions. `static` class variables shall be declared in the class declaration only.

3.2.2 Object, Constant, and Variable Declarations

1. Each object, constant, or variable shall be declared in a separate declaration statement. A descriptive comment must be provided unless the name conveys enough useful information. For example,

```
double alpha = 0.0; // angle of attack - radians
double beta  = 0.0; // sideslip angle  - radians
```

2. Every variable (object member or otherwise) that is declared must be given a value before it is used. Use initialization instead of assignment.
   For example,

```
double alpha;         ← incorrect
        .
        .             ← many lines of code
        .
alpha = 0.0;          ← assign before use

double alpha = 0.0;   ← correct - initialization
```

3.3 Comments

Source code comments are intended to describe what the code does and/or to provide insight into the design of the code. This information can be invaluable for understanding and maintaining the code.

1. The characters `//` shall be used for all comments. The C style comment delimiters `/*  */` are only to be used to make comments out of entire sections of code during development and debugging or in the standard header.

2. Multiple line `//` comments shall precede the line or lines of code to which they apply.
   For example,

```
int variable_1 = 0; // comment line 1   ← incorrect
                    // comment line 2
                    // comment line 3

// comment line 1                       ← correct
// comment line 2
// comment line 3
int variable_1 = 0;
```

3. Each global function definition shall be preceded by a block comment that describes the purpose of the function. Do **not** merely repeat the name of the function in the block comment. The block comment shall be of the form:

```
//=========================================================
// This function has some purpose
```

```
//=========================================================
```

4. As a general rule, one should strive to write code that is clear and unambiguous without comments. Where a comment is required, make it concise and complete. Never repeat information in a comment that is readily available in the code. [8]

5. Misspelled, ungrammatical, ambiguous, or incomplete comments defeat their usefulness. If a comment is worth adding, it is worth adding correctly. [8]

6. Use comments to emphasize the structure of the code and to draw attention to deliberate and necessary violations of standards. [8]

7. Avoid lengthy explanations within the body of the code. If long explanations are necessary, place them in the LaSRS++ banner under Description: or in the block comment before the function body. [8]

8. Indent comments to conform to the indentation of the code so as not to obscure the code's structure or readability. [8]

9. Use blocked comments to highlight divisions between different sections of the code. [8]

10. Inline comments in a given file shall be lined up into orderly columns. Different groups of declarations may line up to a different column but comments within a group should line up to the same column.
    For example,

```
int variable_1 = 0;              // relevant comment
int variable_2 = 0;              // relevant comment
int variable_3 = 0;              // relevant comment

double variable_group2_1 = 0.0;  // relevant comment
double variable_group2_2 = 0.0;  // relevant comment
double variable_group2_3 = 0.0;  // relevant comment
```

11. Variable and/or object names in comments shall be enclosed in single quotes.
    For example,

```
const int array_size = 7;      // size of array 'some_array'
double some_array[array_size]; // some array
```

3.4 DOC++ Comments

DOC++ is a tool that generates HTML output for online browsing of documentation. This tool searches for specially formatted comments in C++ header files, and generates a manual entry for the next declaration in the code.

The developer is responsible for adding the DOC++ comments to the .hpp file for the class. (No DOC++ comments are added to the .cpp since the tool only operates on the header files.)

Processing of the LaSRS++ DOC++ documentation from the source, however, is the responsibility of the VOB librarians at merge time. So, though temporary generation of the output is necessary to validate syntax, developers should not retain permanent copies of the HTML output and should not modify the VOB file system in any way to accommodate their DOC++ class documentation.

Note, also, that this section is not meant to be a tutorial for DOC++, but merely a guideline for the conventions for its usage in the LaSRS++ framework. The DOC++ website, accessible via the wormhole web page, should be consulted for technical guidance.

3.4.1 Usage Requirements

Guidelines for the use of DOC++ (vs. standard C++) comments were chosen to provide comprehensive coverage of class functionality without rendering the code overly difficult to read. With this goal, the following rules govern when to use DOC++ as well as when **NOT** to use DOC++:

1. DOC++ comments are only used in the header (.hpp) file (required by the tool). Source (.cpp) files are commented with standard C++ comment conventions.

2. A useful description of the class must be provided with DOC++ comment preceding the class declaration. For example:

```
/** This class implements the NASA Marshall Engineering Thermosphere Model
    version 2.0. This model is a modified Jacchia 1970 model and is given
    in the subroutine J70.  All of the other subroutines were designed to
    allow flexible use of this model so that various input parameters
    could be varied within a driving program with very little software
    development. Thus, for example, driving routines can be written quite
    easily to facilitate the plotting of output as line or contour plots.
    Control is achieved by setting the values of four switches in the
    driving program, as described in subroutine ATMOSPHERES.
*/
class MetAtmosphere : public EarthAtmosphere
```

3. All class methods, public, protected, and private, must be commented with DOC++. Also, all parameters in the signature must be documented using the DOC++ "`@param`" feature. All return values must be documented using "`@return`":

```
/** Description of someClassMethod
    @param name_1 Description of the name_1 parameter
    @param name_2 Description of the name_2 parameter
    @return Description of the value returned by the method
*/
bool someClassMethod(int name_1, double name_2);
```

   [Redundancy issues concerning this style are addressed at the end of this section.]

4. Individual variable declarations shall **NOT** use DOC++ comments.

DOC++ does not allow comments to occupy the same line as the code they document. Therefore, blocks of variables become significantly less readable when formatted for DOC++. For example:

```
// Standard C++ format for groups of variables

double example_variable_1; // Definition and units of this variable
double example_gain_2;     // Definition and units of this gain
int    example_counter_3;  // Explanation of this counter
bool   example_flag_4;     // Explanation of this boolean
```

vs.

```
// DOC++ comment format for same variables loses visual recognition

/// Definition and units of this variable
double example_variable_1;
/// Definition and units of this gain
double example_gain_2;
/// Explanation of this counter
int    example_counter_3;
/// Explanation of this boolean
bool   example_flag_4;
```

5. Units must be included somewhere in the DOC++ comment (where applicable).

6. DOC++ comments shall **NOT** be used to document groups of declarations. (The tool implementation is simplistic, and it associates the DOC++ comment with the next code declaration, rather than with the group as a whole. Special formatting attempts at grouping in this way can produce undesirable effects in the output.)

7. Comments formatted for other documentation tool packages (e.g. ObjectManual) must be either removed or reformatted to C++ or DOC++ style, as appropriate.

3.4.2 Formatting Requirements

Guidelines in this section are either requirements of the DOC++ tool, or style decisions intended to provide uniformity of code.

1. The DOC++ comment must immediately precede the line it documents (a constraint of the tool).

2. Single line comments use “`///`”

   ```
   /// Description of simple class method
   void simpleClassMethod();
   ```

   Multiple line comments use “`/** ...*/`”

   ```
   /** Properly formatted multiple line descriptions begins on the same
   ```

```
    line as the opening comment delimiter.  The closing comment
    delimiter is placed on a separate line, aligned with the opening
    delimiter.
*/

/** This style of multiple line comment is not allowed because the
    closing delimiter does not occupy its own line (uniformity of
    code) */

/// This style of multiple line comment is not allowed because DOC++
/// mistakenly interprets the second line to be a declaration, and
/// generates strange looking output.
```

3.4.3 Testing Requirements

Header files with DOC++ comments must be tested to validate syntax. This is done by running DOC++ on the test file header, and viewing the output with a web browser. Several options for viewing output are available. The following is a quick way to generate and view a local copy of the HTML produced by DOC++ comments in a single modified file:

[from the directory containing the modified file]

```
% doc++ --dir temp_directory -p -a -H test_filename
% netscape temp_directory/index.html
```

This command sequence generates HTML in a temporary subdirectory (*temp_directory*) off the directory in which you are working. Remember, though, that the final output version of DOC++ will be generated by the VOB librarian as part of the LaSRS++ framework document. Developers should remove temporary directories and HTML outputs when their testing is complete to prevent view clutter:

```
% rm temp_directory/*
% rmdir temp_directory
```

3.4.4 Redundancy Issues

The requirement to use "`@param`" and "`@return`" in the documentation of all class methods has the potential for redundant statements within descriptions that add no value if the developer does not apply common sense. Remember that code, even of a simple accessor method, is probably not as obvious to the person using it as to the person writing it. Inclusion of the signature fields is desired for code uniformity and to eliminate subjective decisions by developers. The onus is on the developer to provide useful definitions.

Example of a poor use of DOC++ fields:

```
/** Returns altitude in feet
    @return Altitude in feet
*/
double getAltitudeInFeet() const;
```

A better use of DOC++ fields:

```
/** Provides pressure altitude measured at the aircraft center of gravity
    @return Altitude in feet, rounded to the nearest hundred
*/
double getAltitudeInFeet() const;
```

3.5 The Preprocessor

1. Preprocessor macros (`#define` directives that contain code) shall not be used; instead, use inline functions.

2. The `#define` directive shall **not** be used to define constants. Such constants are not type-safe.

3. Preprocessor identifiers follow the same naming conventions as enumerators (see 2.6).

4. The `#if defined(`*`identifier`*`)` form of the conditional compilation directive shall be used instead of `#ifdef` *`identifier`*. Similarly, the `#if !defined(`*`identifier`*`)` form of the conditional compilation directive shall be used instead of `#ifndef` *`identifier`* Note: the only exception is the conditional compilation directive used for internal include guards. These directives must use `#infdef` in order for the g++ compiler to implicitly achieve the functionality of external include guards.

5. Do **not** define macros used for conditional compilation in source code. Always define them using a compiler option; e.g., GNU C++ compiler option -D
   Note: this rule does not apply to the `#define` macros used with internal include guards.

6. A number of preprocessor variables have become defacto LaSRS++ standards. See Appendix B for a description of each of these variables.

3.6 Standard Forms

3.6.1 General

1. Do not use structures with member functions; instead, use classes. Structures shall only contain data.

3.6.2 Indentation

1. The standard indentation is two ordinary spaces.

2. Use ordinary spaces instead of tabs (different editors treat tab characters differently).

3.6.3 Blank Lines

1. Always use at least one blank line between function definitions within a given file.

2. Use blank lines to separate logically related sections of code. Logically related sections of code include:

   - Groups of declarations in a class header (constructor, destructors, accessors, mutators, operators, etc.)
   - A group of related variable declarations in a class header
   - A group of objects and functions that work together to compute a specific result. For example,

```
//========================================================================
// This function NEEDS blank lines
//========================================================================
double someFunction(double& argument_1, double& argument_2)
{
  // add the function arguments
  double temp_1 = argument_1 + argument_2;
  // take the square root of the result
  double temp_2 = sqrt(temp_1);
  return temp_2;
}

//========================================================================
// This function HAS blank lines
//========================================================================
double someFunction(double& argument_1, double& argument_2)
{
  // add the function arguments
  double temp_1 = argument_1 + argument_2;

  // take the square root of the result
  double temp_2 = sqrt(temp_1);

  return temp_2;
}
```

3.6.4 Emacs C++ Mode

Use the C++ mode in GNU Emacs to format code. An emacs lisp library is available to set up your environment and speed your code formatting. To use it you need to add /usr/local/site-elisp to emacs' load path and load the elisp library by placing the following lines in your ~/.emacs file:

```
;; Add the local emacs lisp directory to the load path.
(setq load-path
      (cons (expand-file-name "/usr/local/site-elisp")
            load-path))

;; Load the LaSRS++ elisp library.
(load-library "LaSRS++")
```

Once this file is loaded the LaSRS++ style is available in C and C++ modes. You can switch styles at any time like this:

switch to LaSRS++ style:	M-x c-set-style <RET> lasrs++ <RET>
switch to gnu style:	M-x c-set-style <RET> gnu <RET>

Type a ? when asked for the style and you will get a list of available styles. To make emacs use the LaSRS++ style by default, you should add a c-mode-common-hook to your .emacs file to set this along with any other useful settings you prefer (after the point at which the above load library is executed); i.e.,

```
(add-hook 'c-mode-common-hook
 '(lambda ()

    ;; Switch to the LaSRS++ style of code formatting.
    (c-set-style "lasrs++")

    ;; Other c-mode preferences...
  ) t)
```

The following bindings are provided:

```
C-x l i   if statement stub
C-x l f   for statement stub
C-x l w   while statement stub
C-x l d   do-while statement stub
C-x l s   switch statement stub
```

3.6.5 Functions

1. Always provide the *return type* of a function explicitly. [3]

2. Always write the left parenthesis directly after a function name. [3]

3. When defining functions, the leading parenthesis and the first argument (if any) are to be written on the *same line* as the function name and return type. If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument). [3]
   For example,

   ```
   double SomeClass::functionName(double alpha, double beta,
                                  double gamma, double delta,
                                  double epsilon);
   ```

4. Line up argument types and names in separate columns in function declarations and definitions.
   For example,

```
double SomeClass::functionName(bool          some_bool,
                               unsigned int& some_unsigned_int,
                               double&       alpha)
{
 ...
}
```

When this rule causes one or more of the lines to extend beyond column 80, use the following rules, in order, until all lines end at or before column 80. **Note:** an example follows each rule to show the result of applying all previous rules.

- place one or more of the following on the line preceding the function name: inline keyword (if applicable), template <...> (if applicable), return type

```
inline void
SomeClass::putSomeDoubleVectorIntoThisObject(const Vector<double> new_value)
{
  ...
}
```

- if a member function definition, place the class scope qualifier on the line preceding the function name

```
void SomeClass::
putSomeAngularValueVectorIntoThisObject(const Vector<AngularValue> new_value)
{
  ...
}
```

- do not line up the names of the arguments in a separate column

```
double SomeClass::
calculateSomeDoubleValue(const Vector<AngularValue> short_argument_name,
                         const double& very_very_very_long_argument_name)
{
  ...
}
```

- keep all arguments as far to the right as possible.

```
double SomeClass::
calculateAnotherDoubleValue(
                      const Vector<AngularValue> very_very_long_argument_name,
                      const double& very_very_very_long_argument_name)
```

The purpose of these rules is to show a visual distinction between the function name and the arguments to enhance readability.

5. Functions with no arguments shall have an empty argument list; do **not** use void in the argument list of a function with no arguments.

6. The names of **formal arguments** to functions are to be specified and are to be the **same** both in the function declaration and in the function definition. Argument name(s) shall be omitted in member function **definitions** when the name(s) is(are) not used inside the definition. [3]

7. When declaring and defining operators, do **not** put whitespace between the word operator and the operator symbol.

3.6.6 Expressions

1. Use parentheses to clarify the order of evaluation for operators in expressions. [3]

2. Do **not** use spaces around the member access operator `.` or the dereference operator `->`

```
object . x = 1.0;            ← incorrect
object.x = 1.0;              ← correct

object -> x = 1.0;           ← incorrect
object->x = 1.0;             ← correct
```

3. Do not use spaces between unary operators and operands. [3]
   For example,

```
if (! error_1)               ← incorrect
{
  ...
}
else if (!error_2)           ← correct
{
  ...
}
```

4. Use single spaces between binary operators and operands.
   For example,

```
if (alpha&&beta)             ← incorrect
{
  ...
}
else if (gamma && delta)     ← correct
{
  ...
}
```

5. Line up types, names, equal signs, and initial values into orderly columns. Observe the following initial value conventions:

   - Line up integral values by the semicolon
   - Line up floating point values by the decimal point

- Line up string values by the left quotation mark

```
long some_long = 10000;              ← incorrect
int increment = 0;
float some_length = 9.81;
double x = 0.5;
char string[12] = "abcdefg";

long   some_long   = 10000;          ← correct
int    increment   =     0;
float  some_length = 9.81;
double x           = 0.5;
char*  string[12]  = "abcdefg";
```

6. Do **not** use multiple assignments in the same statement.
   For example, do not use statements like:

```
x = y = z = 0;
```

7. Do **not** use spaces between the last token in a statement and the semicolon.

```
long n = 0  ;    ← incorrect
long n = 0;      ← correct
```

8. Float or double literal constants shall neither begin nor end with a decimal point.

```
float x = 5.;    ← incorrect
float x = 5.0;   ← correct
float y = .5;    ← incorrect
float y = 0.5;   ← correct
```

9. Always capitalize the E in scientific notation. Use upper case alphabetic characters representing digits in bases above 10.

10. Use symbolic values instead of hard coded numbers wherever possible; i.e., avoid the use of "*magic numbers*".

```
for (int i = 0; i < 10; i++)                  ←incorrect
{
...
}

const int maximum_missiles = 10;
for (int i = 0; i < maximum_missiles; i++)    ← correct
{
...
}
```

11. Use typedef to simplify the syntax when declaring function pointers. [3]

```
// Pointer to global function
int someIntegerFunction(float some_float);

typedef int (*IntegerFunction)(float);
IntegerFunction someFunction = &someIntegerFunction;

// Pointer to member function in class ClassName
typedef int (ClassName::*IntegerFunction)(float);

ClassName class_name;
IntegerFunction someFunction = &class_name::print;
```

12. When statements are long enough to require multiple lines, place the operators at the right hand side of each subexpression.

```
some_long_result_variable = first_long_variable *
  (second_long_variable + third_long_variable) *
  (fourth_long_variable - fifth_long_variable);
```

13. If it is necessary to break a long member function call at the scope operator `::`, the member access operator `.` or the dereference operator `->`, keep the operator with the scope or object name.

```
double result = very_long_object_name->
  someFunctionName(const double& very_very_long_argument_name);
```

3.6.7 Standard Form for Selection Statements

```
if (...)
{
  ...
}
```

```
if (...)
{
  ...
}
else
{
  ...
}
```

```
if (...)
{
  ...
}
else if (...)
{
  ...
}
else
{
  ...
}
```

```
switch (...)
{
  case 1:
    ...
    break;

  case 2:
    ...
    break;

  default:
    ...
    break;
}
```

3.6.8 Standard Forms for Iteration Statements

```
for (...; ...; ...)
{
  ...
}
```

```
do
{
  ...
}
```

```
while (...)
{
  ...
}
```

```
while (...);
```

3.6.9 Notes on Selection and Iteration Forms

1. Braces { } which enclose a block shall be placed in the same column, on separate lines directly before and after the block.[3]

2. Flow control primitives `if`, `else`, `while`, `for` and `do` shall be followed by a block, even if it is empty. [3]

3. The code following a `case` label shall always be followed by a `break` statement unless the given `case` is sharing a section of code with other cases. [3]

4. A `switch` statement shall always have a `default` branch to handle unexpected cases. [3]

5. Use `break` to exit iteration statements only if it will remove the need for a flag specifically designed to test for such an exit condition. [3]

6. Do **not** use `goto` statements.

7. When writing endless loops, use `while (true);` do **not** use `for (;;)` or `while (1)`

3.7 Warning and Error Messages

The following is the standard form for LaSRS++ warning and error messages:

Line 1 WARNING! or ERROR!
Line 2 LOCATION: function name (scoped by class if appropriate)
Line 3+ DESCRIPTION: (multiple lines if necessary)

LaSRS++ supplies a class that shall be used to issue standard warning and error messages. The TerminalIO class (part of the LaSRS++ Toolbox) contains the static methods applicationWarning() and applicationError(). These methods take two arguments, a string containing the function name and a string containing a description of the warning or error. Neither string shall contain formatting characters; the TerminalIO methods handle the formatting for the programmer.

applicationError() now has an argument that will cause it to call abort() and terminate the program. It also has an argument that can cause it to retrieve and print the system error. Please refer to the TerminalIO class for more information about these arguments.

The TerminalIO class contains other methods besides applicationWarning() and applicationError(). Programmers are encouraged to make use of this class (instead of iostreams) where appropriate.

```
// Example using TerminalIO
#include "TerminalIO.hpp"
```

```
void SomeArrayClass::someArrayFunction(const int index)
{
  // if the array pointer is null
  if (array == 0)
  {
    TerminalIO::applicationError(“SomeArrayClass::someArrayFunction()”,
                                 “An attempt was made to use an array “
                                 “class which has not yet allocated any “
                                 “array space.”);
    return;
  }
  else if (index == 0)
  {
    TerminalIO::applicationWarning("SomeArrayClass::someArrayFunction()",
                                   "The first element of the array is "
                                   "used for some internal class “
                                   “function. Accessing it in this “
                                   “manner could cause unpredictable “
                                   “behavior.");
  }
}
```

4 Classes

4.1 General

1. Do **not** declare *any public or protected member data* in a class. This ensures that only the member functions of the class can *directly* modify the data.

   This guideline is in accordance with the fundamental concept of the object model known as *encapsulation*. It ensures that the underlying data representation of a class remains hidden behind a well-specified public (for non-derived classes) and/or protected (for derived classes) interface. As a result, the underlying implementation (but not behavior) of the given class can change without requiring code changes in using or derived classes. [3][7]

2. A constructor shall be defined for every class that contains member data.

3. If there is a requirement for an object to be reinitialized, i.e., reset to a known state, then the given class shall declare and define an `initialize` member function. A constructor of this class may call this member function for first time initialization.
   Note: The reinitialize function for a class **must** be called `initialize`.

4. The following items **must** be defined for any class that dynamically allocates memory:

   (a) destructor
   (b) copy constructor
   (c) assignment operator

5. If a class defines `new`, then it must define `delete`. [7]

6. Accessor member functions may return intrinsic types by constant reference or by value. All other types must be return by constant reference.

7. Mutator member functions shall have a return type of `void`; i.e., they do **not** return a value.

8. Put class specific `typedefs` and `enums` within the class header. If they are designed to be used by other classes then place them in the public interface, otherwise place them in the private part of the class.

9. Inlined member functions that are not part of a class public/protected interface shall be placed in the appropriate ***.cpp** file.

10. Do **not** use `this` when referring to member objects.

11. Remember that default public implementations of the following will be generated, if needed, by the compiler: [7]

(a) default constructor

(b) destructor (if derived from a class that has a destructor)

(c) copy constructor

(d) assignment operator

(e) address-of operators (const and non-const)

Note: The default constructor will **not** be generated by the compiler if *any* other constructor is declared. The default implementations of the other items can be suppressed by declaring them to be `private`.

12. A user-defined assignment operator shall: [7]

(a) return a reference to *this

(b) assign to all data members

(c) check for an assignment to self

i. Immediate return is usually more efficient

ii. Assignment operators typically free resources allocated to an object before allocating new resources. When assigning to self, the old resources might be needed during the process of allocating new ones.

Note: The issue of *object identity* must be considered when writing the actual comparison in the check for assignment to self. If *value equality* is used to determine identity, then two objects are the same if they have the same value (checked via operator==). If *address equality* is used, two objects are the same if they have the same address. Address equality is usually more efficient and more commonly used but can fail in the case of multiple inheritance (see [7] for a more detailed discussion).

For LaSRS++, address equality shall be used unless there is a compelling reason to use value equality.

```
class Point
{
public:
  // operators
  Point& operator=(const Point& right_hand_side);
  ...
private:
  long x; // x coordinate
  long y; // y coordinate
```

```
};

Point& Point::operator=(const Point& right_hand_side)
{
  if (this == &right_hand_side)     // check for self - address equality
  {
    return *this;                   // return reference to *this
  }

  x = right_hand_side.x;            // assign to all data members
  y = right_hand_side.y;

  return *this;                     // return reference to *this
}
```

The value equality check looks like:

```
if (*this == right_hand_side)  // check for self - value equality
{                              // NOTE: assumes   operator== exists
  return *this;                // return reference to *this
}
```

4.2 Constructors

1. Every member object in a class **must** be initialized in the constructor.

2. When initializing attributes of a class, prefer the member initialization list to assignment statements in constructors. [3] [7]

3. For the member initialization list, put the : on the line following the constructor name and argument list. The : shall be indented two spaces.

4. The use of inlined constructors is highly discouraged. If an inlined constructor is deemed necessary, it **must** be well documented. Remember that a constructor always invokes the constructors of its base classes and member data before executing its own code. This means that a constructor can potentially execute a large amount of code, making the constructor ineligible for inlining. [3]

5. A single argument constructor specifies a conversion from the argument type to the type of its class. This type of constructor is known as a conversion constructor. The function-specifier `explicit` shall be used with conversion constructors. This allows the compiler to enforce that a conversion constructor will only be used where a constructor call is explicitly indicated by the syntax.
   For example,

```
class Z
{
public:

  explicit Z(int i);
  ...
```

```
};

...

Z z = 1;        // error, implicit conversion
Z z = Z(1);     // ok, explicit use of constructor
```

4.3 Destructors

1. Destructors in base classes that contain virtual functions shall be `virtual`. [7]

2. The use of inlined destructors is highly discouraged. If an inlined destructor is deemed necessary, it **must** be well documented. As with constructors, destructors can potentially execute a large amount of code, making the destructor ineligible for inlining. [3]

5 Namespace Declarations

5.1 General

1. Namespaces will only contain related enumerations, constant variables, class declarations, and functions.

2. All namespaces declared in a header file will be named.

3. The namespace declaration in the namespace's header file will only contain namespace elements intended as part of the namespace's external interface. Namespace elements intended for internal use will be included in a second namespace declaration in the namespace's implementation file. [Note: By including the namespace header in the namespace implementation file, external elements do not need to be redeclared.]

4. Enumerations, constant variables, and functions should only appear in a namespace to make it easier to share them among classes that are not related through inheritance. Otherwise, the namespace members more appropriately belong in the common base class. Namespaces are not to be used as a substitution for classes although there is a very fine line between namespaces with functions and class utilities. Namespace functions may only work with static constants and function arguments. If a function requires mutable static data, then the function must be placed in a class utility. [All mutable data must be contained within a class. Moreover, this rule makes it easier to implement mutual exclusion protection for the static data in order to make the functions that act on it thread-safe.]

5. Definitions of inline namespace functions shall appear immediately following the namespace declaration.

6. Only static constant integral variables may be defined inside a namespace declaration. All other constant variables must be defined in the implementation file.

7. All declarations in a namespace must have an appropriate inline comment unless the declaration conveys enough useful information. Units (properly abbreviated) shall be provided if the declaration is a dimensional constant.

Part II

Implementation

6.0 General

6.1 Unsigned Variables

1. Do **not** use `unsigned` to gain one more bit to represent positive integers. Use the next largest signed integral type instead. [9]

2. Do **not** attempt to ensure that an integral variable will be positive by declaring it to be `unsigned`. This will typically be defeated by the implicit conversion rules. Assigning a negative value to an unsigned variable is legal C++, although the compiler may issue a warning. [9]

3. Unsigned variables **should** be used for loop counters that do not have negative values and for other positively valued data that undergoes simple, repeated arithmetic operations. Most compilers can better optimize unsigned arithmetic using bit shifts.

6.2 Casting

1. The use of casting is highly discouraged. If a cast is deemed necessary, one of the three cast operators: `static_cast<T>`, `reinterpret_cast<T>`, or `const_cast<T>`, shall be used. [9] This makes the intent of the cast explicit.
   Note: The use of a cast **must** be well documented.

2. Only `dynamic_cast` may be used to downcast. All other forms of downcasting are disallowed. Downcasting is the casting of an object (or pointer or reference to an object) of a base class to an object (or pointer or reference to an object) of a derived class.

3. The return value resulting from a `dynamic_cast` of a pointer argument will always be tested for zero before it is dereferenced. If the value is zero, the cast failed.

4. A `dynamic_cast` of a reference argument throws a `bad_cast` exception if the cast fails. Code that applies a `dynamic_cast` to a reference must handle the exception.

5. The C++ standard requires that the argument of a `dynamic_cast` must be a pointer or reference to a polymorphic type (i.e., have virtual functions). Developers will not place a virtual method in a class for the sole reason to allow `dynamic_cast` of an object of that type. This is a sign of poor design.

6.3 Typeid()

1. Developers cannot assume the naming convention of names stored in the `typeinfo` returned by `typeid()`. Names generated for `typeinfo` are implementation dependent.

2. Use of `dynamic_cast` is preferred over `typeid()` for type identification.

6.4 Functions

6.4.1 Return Values

1. Do **not** return a reference or a pointer to a local variable. Memory for local variables in a function is deallocated when the function returns. [3]

2. Pass and return objects by reference instead of by value; however, do not try to return a reference when you must return an object. [7]

6.4.2 Inline Functions

1. Remember that declaring a function `inline` is just a **hint** to the compiler that the author wants the particular function to be inlined. Functions can be too complex for the compiler to successfully inline. [3][7][9]
Note: GNU C++ does **not** inline the **first invocation** of a given template function.

2. Accessor functions that simply return a class member shall be `inline`. [3] Accessor functions that become more complex than a simple `return` shall be reevaluated to see if they should remain inlined.

3. Mutator functions that consist of a single assignment statement shall be `inline`. [3] Mutator functions that become more complex than a simple assignment shall be reevaluated to see if they should remain inlined.

4. Forwarding functions (functions that do nothing more than call another function) shall be `inline`. [3]

5. Virtual functions shall **not** be inlined.

6. Inline functions should not contain more than a half-dozen statements unless it can be demonstrated that inlining the function produces a meaningful and necessary performance benefit under normal use.

6.4.3 Methods

1. When taking the address of a method, the method name shall be scope qualified. The C++ standard requires that a method be scope qualified when its address is taken, even if the address is taken within a different method of the class. For example:

```
SomeClass::someMethod()
{
  functionTakingMethodPointer(&otherClassMethod);             ← Incorrect
  functionTakingMethodPointer(*&SomeClass::otherClassMethod); ← Correct
```

```
}
```

6.5 Expressions

1. Always test floating point numbers with <= or >=. Never perform a direct comparison of floating point numbers using == or !=. Such direct comparisons can be performed using the global function:

```
// Return true if query_value is within the range
// target_value +/- epsilon,
// otherwise return false
bool withinEpsilonOf(const double query_value,
                     const double target_value,
                     const double epsilon = 1.0E-12);
```

2. Always check for a divide by zero if it is possible for a denominator to be 0.

3. Always verify that the argument to the `sqrt()` function is positive or limit the argument to zero or greater if it is possible for the argument to be negative.

4. Always verify that the argument to the `acos()` and `asin()` functions lie between [-1,1] or limit the argument to [-1,1] if it is possible for the argument to lie outside these limits.

5. Unless necessary for mathematical correctness, use floating point literals in expressions that are assigned to floating point variables. Also avoid integer sub-expressions in expressions assigned to floating point values by explicitly converting integer variables to double precision floating-point values. Otherwise, the result may not evaluate as expected. For example, 2/3 * 10.0 = 0 not 6.666.

6. Always use `std::fabs()` to compute the absolute value of a floating point argument. Although C++ provides floating point signatures for `std::abs()`, C++ also makes it too easy for `std::abs()` to resolve to the integer signature instead of the floating point signature. The signatures for `std::abs()` are split across two header files. Integer signatures are found in `<cstdlib>`. Floating point signatures are found in `<cmath>`. Since `<cstdlib>` is commonly included in other system header files, there is a good chance that a call to `std::abs()` with a floating point argument will resolve to an integer `std::abs()` if the `<cmath>` header is missing. Not all compilers warn the developer when this occurs. `std::fabs()` will always resolve to a floating point signature or will generate a compile-time error if `<cmath>` is missing.

6.6 Classes

1. Constants and reference members must be initialized through the member initialization list; they cannot be initialized by assignment inside the constructor body. [9]

2. Do **not** call destructors explicitly. Allow destructors to be called implicitly when the given object goes out of scope.

3. Remember that `static` class variables are accessible by all instances of a class. If the previous value of a variable is needed, explicitly save the past value in its own class scope non-`static` variable.

4. Remember that a `virtual` function called in a constructor (or) will be the one defined in the constructor's (or destructor's) own class or its bases, but **not** any function overriding it in a derived class. [9]
   Note: Virtual functions calls in a constructor shall be made explicit by qualifying the function call with the class name and scope resolution operator.

6.7 Limiting Namespace Pollution

One purpose of the namespace is to prevent name collisions, especially in large systems where the same name may appear in different contexts to represent different concepts. Unless the developer is careful, the developer can easily disable the protection that namespaces provide against name collision. Abuse of the “using namespace” directive in header files will quickly reduce namespace members to the equivalent of global scope; this dilution of the namespace scope is called “namespace pollution”. The following rules are partially based on Stoustrup's advice in Chapter 8.2 of the C++ Programming Language:

1. Avoid exposing an entire namespace with a using directive:

   ```
   using namespace a_namespace.
   ```

2. Never deploy a using directive at file scope in a header file. That is,

   ```
   using namespace a_namespace
   ```

   is not allowed at file scope in header files.

3. Place using declarations local to the scope of their use:

   ```
   using a_namespace::namespace_member
   ```

   Using declarations can be placed nearly anywhere a typedef can be placed.

4. Deploy a using declaration at file scope in a header file only to expose namespace members that are necessary to interact with global functions that are declared in the header file or with the public or protected interface of the class that is declared in the header file:

   ```
   using a_namespace::namespace_member
   ```

5. Namespace members that appear in the private section of a class or the internals of an inlined function will be identified using a scope qualifier or, for the function, a using declaration within the scope of the function.

6.8 iostreams

1. Use iostreams instead of the I/O functions found in stdio.h. Iostreams are type safe, type extensible, and significantly more powerful. [1]

2. If formatting flags are altered, they shall be restored after the specific usage is complete. For example,

```
ios::fmtflags previous_format_flags = cout.flags();

// set flags and output to cout...
cout.flags(previous_format_flags);
```

6.9 Use of const

1. Constants are to be defined using the keyword const or with an enumeration type. [3] Note: an enumeration type should be used for a group of related constants.

```
const int some_constant = 7;
enum Colors
{
  RED   = 1,
  GREEN = 2,
  BLUE  = 3
};
```

2. Avoid having public member functions return a non-`const` reference or pointer to member data. [3] [7]

3. Use constant references (`const &`) instead of call-by-value for arguments when the argument is not to be modified by the function. [3] [7]

4. All member functions that do not alter the state of the object to which they refer shall be declared `const`; e.g., accessor functions. [3] [7]

```
const Angle& getAlpha() const;   // return angle of attack
```

5. Remember that `const` member functions are the *only* member functions that can be invoked on a `const` object. [3]

6. Use of `const` with pointers: [7]

```
      char*      p = "Hello";    // pointer to data
const char*      p = "Hello";    // pointer to constant data
```

```
      char* const p = "Hello";    // constant pointer to data
const char* const p = "Hello";    // constant pointer to constant data
```

7. Use `const` wherever possible. [7]

6.10 Compiler Warnings

1. All compilations shall be done with full warnings enabled. Any warnings shall be removed from code (baseline or project specific).
   Exception: Warnings that indicate the compiler is unable to inline certain functions are allowable. This usually occurs when optimization is enabled.

6.11 Debug Levels

Debug code that is required to verify correct simulation operation and error-free computation shall always be active. Other debug code shall be enclosed in preprocessor conditional compilation directives of the form:

```
#if defined(DEBUG_LEVEL_X)

...  debug code ...

#endif
```

Debug code is classified by its relevance to each stage of testing and its potential effect on program execution, particularly during real-time critical segments (i.e., during OPERATE mode). Debug code is classified into three levels generally corresponding to the three main phases of testing: unit testing, integration testing, and system testing. By separating debug code in this manner, later phases of testing are not bombarded with information that was analyzed and reconciled in earlier phases of testing. The LaSRS++ debug levels are:

- DEBUG_LEVEL_1

  Debug level one exposes debug code designed to identify problems during system testing (aka checkout). This code constitutes simple statements and integrity checks (e.g. assume() statements) which can be used during real-time operation. Examples include testing successful memory allocation or validating function arguments. For code that will run during real-time critical segments, this level of debug code will not produce any file output. It may only produce simple screen output if the code will cause immediate program termination.

- DEBUG_LEVEL_2

  Debug level two exposes debug code designed to capture problems during integration testing (aka batch testing). This code focuses on potential errors in communication between objects and also provides additional status on simulation progress. The debug code has minimal or moderate impact on program performance. Simulations running debug level two code should

complete within a reasonable amount of time. Any code that generates a LaSRS++ standard warning or error and that does not terminate the program must be placed at level two or higher.

- DEBUG_LEVEL_3

Debug level three exposes debug code designed to capture defects during unit testing. The code runs detailed diagnostic tests and progress reporting on an individual class (or small subsystem of classes). Code that severely impacts the performance of the simulation also falls under this debug level.

6.12 assume() Preprocessor Macro

LaSRS++ replaces the standard C preprocessor macro assert() with a new preprocessor macro called `assume()`. Like `assert()`, `assume()` takes a boolean argument. If the argument evaluates to false, `assume()` will print an error message and terminate the program.

```
// example code using assume()
#include "assume.hpp"

void someArrayFunction(void* array, const int size, const int index)
{
  // Protect against null pointers
  assume(array != 0);

  // Some code appears here
  // ...

  // Protect against an out-of-bounds index
  assume((-1 < index) && (index < size));
}
```

To use `assume()` in LaSRS++, one must include the file `assume.hpp`.

`assume()` has two major advantages over `assert()`:

1. The compiler generates code for `assume()` only when the preprocessor conditional compilation directive DEBUG_LEVEL_1 is defined. Otherwise, `assume()` is treated like a null statement.

2. `assume()` uses iostreams when printing its error message. `assert()` uses stdio. Mixing stdio with iostreams can produce unexpected results on rare occasions.

7. Memory Management

1. Do not use `malloc, realloc`, or `free`. Memory shall be dynamically allocated with `new` and deallocated with `delete`. [3] [7]

2. Always use the same form in corresponding calls to `new` and `delete`. If `new` has no `[]`, then `delete` shall have no `[]`. If `new` has `[],` then delete shall have `[]`.

3. If `new` is redefined, then a corresponding `delete` must also be defined. [3] [7]

4. Do not allocate memory and expect someone else to deallocate it later. [3]

5. Always assign a value of 0 to a pointer that points to deallocated memory.

6. Destructors should call `delete` on all pointer members that point to dynamically allocated memory.

7. In destructors, call `delete` on member pointers to objects in the opposite order than they were allocated, if possible. This maintains consistency with compiler generated order of destroying static objects.

8. Always check the return value of `new` to see if memory allocation was successful. This check can be done inside an `assume()` call. [7]

9. In real-time critical code, the developer should not take actions that could result in memory allocation through the operating system. Memory allocation is a non-deterministic operation that can take as long as several milliseconds to complete. Thus memory allocation can cause real-time code to randomly miss a deadline. For LaSRS++-based simulations, real-time critical code runs during OPERATE mode. The following actions can result in memory allocation:

   (a) Explicit use of the ordinary `new` operator (i.e. not placement syntax).

   (b) Creation of temporary objects of any classes whose constructor performs dynamic memory allocation. The following actions will cause the creation of a temporary object:

      i. Passing or returning the object by value.

      ii. Declaring an object, which is not static and constant, within function scope.

      iii. Implicit or explicit conversion. Implicit conversion is difficult to identify. However, implicit conversion should only occur with the use of standard C++ library classes. Section 3.2 Bullet 5, requires that all single argument constructors in LaSRS++ be given the `explicit` qualifier to prevent implicit conversion.

(c) The `std::string` class's constructor performs dynamic allocation and its use in real-time critical code must adhere to the guidelines in 9.b. The following guidelines also apply to use of `std::string` in real-time critical code:

i. The concatenation operator (`'+'`) and `string::substr()` shall not be used. These functions return a temporary string by value.

ii. Before entering real-time, the developer must use `string::reserve()` to reserve enough memory for the largest possible string if string operations that add characters will be used during real-time. Without reserving memory, these operations can cause memory allocation. Operations that add characters to a string are `string::append()`, `string::+=()`, `string::=()`, `string::assign()`, `string::insert()`, `string::replace()`, `string::resize()`, and `string::push_back()`.

iii. All comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) shall not be used with character arrays. (Be particularly mindful of string literals.) The comparison operators will force an implicit conversion of the character array to a string. Instead, compare strings and character arrays using `string::compare()`. This rule should also be applied outside of real-time critical code because it leads to better computational efficiency.

(d) All Standard Template Library (STL) containers perform dynamic memory allocation and their use in real-time critical code must adhere to the guidelines in 9.b. The following guidelines also apply to the use of STL containers in real-time:

i. If a developer wants to add elements to a vector that uses the default allocator while in real-time, the developer must use vector::reserve() to reserve enough memory accommodate the largest size of the vector. Without reserving memory, the vector can cause memory allocation to occur.

ii. If using the default allocator, elements shall not be added to containers other than vector during real-time. Adding elements to containers other than vector (even if an equal or greater number of elements are removed first) can result in memory allocation. These containers do not have a reserve() method.

8. Code Reuse

1. Use existing baseline objects whenever possible. Do **not** reimplement baseline code in order to enhance or alter your object's functionality. The appropriate action is to inherit from baseline classes and either modify existing member functions or add new member functions. If there is no baseline class that applies to the object in question, then appropriate baseline classes shall be provided by the LaSRS++ CCB.

2. New baseline classes shall be added and/or existing baseline classes updated if new functionality is discovered that can benefit all users.

3. A given class **must** have the same interface/behavior on any platform that is supported by LaSRS++. If the interface/behavior of a class changes across platforms, then new classes shall be created to express this different interface/behavior.

4. The C++ library shall be used where reasonable and appropriate. Use of standard library features is preferred over similar operating system functions. Developers will not re-invent features available in the standard library.

5. The standard C library shall be used only if an equivalent feature cannot be found in the standard C++ library. [See Section 17 of the ANSI/ISO C++ Standard for the separation of standard C++ and C libraries.]

6. STL containers and std::string will not be re-invented to handle special memory situations. Placement new syntax and custom allocators should be sufficient to handle special memory needs.

9. Exception Handling

1. C++ exception handling will be used in place of `setjmp()/longjmp()`.

2. Code should throw an exception instead of calling `exit()` or `abort()`. Code that calls `exit()` and `abort()` cannot be reused in software that is not allowed to terminate in the presence of an error.

3. Do not use exceptions when local control structures are sufficient. [9] In other words, an exception should not be thrown and caught in the same function scope. This is an inefficient means of identifying and handling exceptions locally. In fact, it is akin to using a `goto` construct in the code. `If-then-else` and other control structures are better suited for this purpose.

4. The main program shall not throw exceptions. [This is actually a special case of (3). If the main program throws an exception, it must catch it locally or the program will terminate.]

5. All exceptions will be objects; intrinsic types will not be used as exceptions. The class of the exception will be dedicated to communicating information about the exception. The class will have no other purpose than to define exceptions.

6. Exception classes should not contain code for handling the exception. Their purpose is to report the exception. Handling the exception is the responsibility of the object that catches the exception.

7. All exception classes must allow copy construction. An exception will always be copied when thrown.

8. Exception classes should not throw exceptions. Thus, exception classes that use `new` must handle the `bad_alloc` exception.

9. All exceptions shall be caught by reference. If a derived class exception is caught by-value as an ancestor class, then the virtual methods of the derived class will be inaccessible in the handler.

10. A handler (i.e. catch block) for a derived exception class must appear before the handler of its superclass(es) if present. Otherwise, the handler for the super class will trigger before the handler for the derived class; the handler for the derived class will never execute.

11. Throwing exceptions should be avoided in copy constructors and assignment operators. These operations are implicitly used in many situations and code that invokes these operations implicitly is unlikely to catch the exception.

12. Do use exceptions for reporting failure in constructors other than copy constructors. [9]

13. Destructors should not throw exceptions and should handle all exceptions thrown by functions that it invokes. Destructors are likely to be called as an exception unwinds the stack. If a destructor throws an exception while another one is active, the program will terminate.

14. Functions that throw an exception must specify the exceptions that they will throw when they are declared and defined. (In this context, an exception thrown by a function is any exception that can be returned through the function; this includes exceptions directly thrown by the function or indirectly thrown by other functions that the given function may call.) The exception specification list identifies the possible exceptions that a client can expect to receive as a result of calling the function.
    For example,

    ```
    void Propulsion::addEngine(Engine* new_engine) throw(MaxEngines);
    ```

    The exception specification must be identical between the function's declaration and definition.
    Note: The exception specification restricts the function to throwing the named exceptions or exceptions derived from the named exceptions. Note: For virtual functions, this restricts the list of exceptions that definitions in derived classes can throw. Note: This function will terminate the program rather than allow other exceptions not on the list (or derived from the list) to pass through it unless `std::bad_exception` is on the list. Then, `std::bad_exception` will replace the exception not on the list.

15. Functions that throw exceptions and use the standard C++ library will include `std::exception` on their exception specification list if they let this exception pass through (i.e. the function does not handle the exception internally). All standard C++ library exceptions are derived from `std::exception`.

16. The developer must take steps to ensure that memory allocated in a scope where an exception can occur, will be deallocated. When an exception is thrown, memory is automatically deallocated only through the following mechanisms. The destructors of local objects (i.e. objects on the stack) will be called when the exception is thrown; class members contained by value are treated as locally constructed objects within the class constructor. Memory allocated for an object or an array of objects using the normal new operator (i.e. not the placement syntax) will be freed if the object's constructor throws an exception. Otherwise, the developer must guarantee deallocation of dynamic memory. For example, developers can use `std::auto_ptr` to prevent memory leaks associated with dynamic creation of objects assigned to local pointers.

17. Developers must also take steps to ensure that resources, acquired in a scope where an exception can occur, will be released. This can be accomplished by using “resource acquisition is initialization.” [9]

10. Portability

10.1 Sizes of Fundamental Types

C++ does not make any guarantees concerning the exact sizes (in bits) of the fundamental types. However, C++ does guarantee that the sizes shall conform to the following constraints: [9]

- `1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)`
- `sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)`
- `sizeof(I) ≤ sizeof(signed I) ≤ sizeof(unsigned I)`

where I can be `char, short, int,` or `long`

- `char is at least 8 bits`
- `short is at least 16 bits`
- `long is at least 32 bits`

Remember that `char, signed char,` and `unsigned char` are distinct types.
Note: depending on the hardware, char is a `signed` or an `unsigned integer`.

10.2 Integral Types of Exact Sizes

A collection of platform specific `typedefs` are maintained in the file `typedef.hpp`. The name of a given `typedef` captures the size (in bits) of the underlying integral type. The `typedefs` currently in the file are:

- Signed8Bits
- Signed16Bits
- Signed32Bits
- Signed64Bits
- Unsigned8Bits
- Unsigned16Bits
- Unsigned32Bits
- Unsigned64Bits

These typedefs shall be used in all cases when an integral type of an exact size is required.

10.3 Miscellaneous

1. Do not write code that assumes the size of a class. The point of encapsulation is to protect client code from the underlying implementation. A class's size is implementation dependent and can even change from compiler to compiler. Relying on class size is not portable.

2. For UNIX platforms, POSIX functions shall be preferred over UNIX or vendor-specific operating system functions unless a compelling reason exists otherwise.

11 Optimizations

11.1 Run-time Optimizations

1. All polynomial equations can be decomposed into a series of efficient, nested subexpressions. For example, $5x^6 - 7x^4 - 3x^3 + 4x + 12$ can be reduced to the more efficient expression, `12.0 + x * ( 4.0 – x * x * ( 3.0 + x * ( 7.0 - 5.0 * x * x ))).`

2. Prefer pass-by-reference over pass-by-value for objects.

3. Use unsigned integers for counters and loop indices that are always positive. Most compilers use bit shifting to optimize the increment operation.

4. Do not qualify methods in a superclass as virtual unnecessarily. A virtual method invocation has more overhead than a non-virtual method invocation.

5. Try to order case statements and if-then-else statements by probability of occurrence. The most probable execution path should appear at the top.

6. Try to place branching statements (case and if-then-else) outside of loops. (Compilers will do this for the developer, but only in simple cases. For example, if the branch logic depends on a function call, it will not place the branch outside the loop because it must assume that the function may have side effects.)

7. Unless required for mathematical correctness, always use integer literals in arithmetic expressions containing only integer variables. Also, explicitly convert floating point variables to integers when the floating point variable appears in an integer expression. The compiler is forced to convert the sub-expression containing the floating-point value to floating-point arithmetic (which is more computationally expensive than integer arithmetic). Moreover, the floating-point conversion spreads to all sub-expressions that use the result of this initial floating-point sub-expression. The end result will be a floating-point value and must be converted before it can be used as an integer.

8. If a sub-expression appears more than once in a function, store the result of the sub-expression in a local variable and use the variable in place of the sub-expression if the sub-expression does not call functions with side effects. Note: compilers will typically recognize and optimize repeated sub-expressions involving intrinsic operations and variables. But compilers will not optimize sub-expressions that include function calls because the compiler must assume that the function has side effects, which must be exercised with each call.

9. When comparing a character, character array, or string literal with a string object, use the `string::compare()` method rather than the comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`). The comparison operators will force an implicit conversion of the character-based expression to a string. The `string::compare()` method does not require this conversion; therefore, it is more computationally efficient.

11.2 Build-time Optimizations

Developers must take actions to limit the number of include files that appear in header files. Otherwise, the system will, over time, develop a complex web of include file dependencies that will cause a large number of unrelated items to be re-compiled for changes made to a header file. This can slow down development immensely. The following simple actions will limit include file coupling:

1. Prefer forward declarations over include files. See section 3.1.2, Item 8.

2. If a header file must appear to resolve the definition of an infrequently called inline function in the header, remove the inline qualifier from the function and move both the header and inline function to the body (.cpp). [Note: Private inline headers are automatically placed in the implementation file according to section 3.2.1, Item 3.]

3. Prefer aggregation by reference over aggregation by value. In other words, the class aggregates objects using a pointer or reference to the object rather than containing the object directly. This allows forward declaration of the class of the aggregate object.

12 Simulation Specific Issues

12.1 Framework Issues

1. Use `SimControl::getMode()` to determine the current mode. Do not check to see if time is 0.0 to determine if the simulation is in `RESET`.

2. The use of friends in framework level code is highly discouraged. If friends are deemed necessary, they **must** be well documented.

3. Multiple inheritances in framework code are highly discouraged. If multiple inheritance is deemed necessary, it **must** be well documented.

12.2 Aircraft Specific Issues

12.2.1 Use of Friends

The use of friends for data transfer between aircraft systems is not allowed in LaSRS++ because of data encapsulation requirements. For example, F16aAero is not allowed to friend F16aTurboFan.

However, the use of friends **is** allowed for classes that only provide a user interface to one specific project class. This generally occurs in two circumstances -- class specific GUI dialogs and class specific data recording sets. This judicious use of friends prevents data that is solely targeted for user-interfaces from being publicly exposed through accessors and mutators for modification by other clients. The developer should treat the friend as an extension of the class interface visible only to users. The friend should control write access to class data in a manner that preserves data integrity of the class. In other words, the developer should code write access to class data via the friend with the same controls they would place into a public mutator of the class itself. This can be accomplished within the friend or via a private mutator that the friend calls. [In cases where direct assignment to class data does not destroy data integrity of the class, no controls are necessary.] By prohibiting friended classes from sharing acquired data with other classes, the friend does not provide a backdoor that allows the other classes to circumvent encapsulation.

The friend declaration shall be the first active statement of the public section. (The behavior of the friend declaration does not depend on the section in which it is located, but putting it at the top of the public section allows immediate identification of the relationship upon inspection.)

The following example shows an allowed use of friends:

```
class Example
{
public:

  friend class ExampleGui;           // GUI dialog for this class only
  friend class ExampleRecordingSet;  // Data recording for this class only
  ...
```

```
};
```

12.2.2 Use of Multiple Inheritance

1. The use of multiple inheritance is highly discouraged in aircraft code.

Appendices

Appendix A Sample Source Files

A.1 Base Class Header File

The following is an example of a header file for a base class called PositionalModel:

```
/*****************************************************************************
 *                                                                           *
 *      _/                    _/_/_/_/  _/_/_/_/  _/_/_/_/                   *
 *     _/                    _/        _/    _/  _/            _/      _/     *
 *    _/         _/_/      _/_/_/_/  _/_/_/_/  _/_/_/_/   _/_/_/  _/_/_/     *
 *   _/        _/   _/           _/  _/  _/           _/     _/      _/        *
 *  _/_/_/_/   _/_/_/_/  _/_/_/_/  _/    _/  _/_/_/_/                         *
 *                                                                           *
 *  File:              PositionalModel.hpp                                   *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *  NOTICE:                                                                  *
 *                                                                           *
 *    THIS SOFTWARE MAY BE USED, COPIED AND PROVIDED TO OTHERS ONLY AS       *
 *    PERMITTED UNDER THE TERMS OF THE CONTRACT OR OTHER AGREEMENT UNDER     *
 *    WHICH IT WAS ACQUIRED FROM THE U.S. GOVERNMENT.  NEITHER TITLE TO      *
 *    NOR OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.  THIS NOTICE      *
 *    SHALL REMAIN ON ALL COPIES OF THE SOFTWARE.                            *
 *                                                                           *
 *****************************************************************************/

#ifndef POSITONAL_MODEL_HPP
#define POSITONAL_MODEL_HPP

#include <string>

using std::string;

#include "EulerAngles.hpp"
#include "GeodeticCoordinates.hpp"
#include "IntegrandTemplate.hpp"
#include "Mode.hpp"
#include "RotationMatrix.hpp"
#include "SimulationModel.hpp"
#include "UnitQuaternion.hpp"
#include "Vector.hpp"

class AsciiDisplaySystem;
class DataRecordingSystem;
class GeoRefPoint;
class GeoRefRelInfo;
class GeoRefRelInfoHandle;
class GeoRefRelInfoList;
class InitializationFileProcessor;
class Pilot;
class PositionalModelPlayback;
class Timer;
class Universe;
class World;
class ostream;
```

```
/** A positional model is the most basic, independent model in a simulation.
    A positional model has location, orientation, and velocity (translational
    and angular).  These are the minimum characteristics required to calculate
    relative geometry between any two objects and to place the object within
    a virtual reality (e.g. CGI display).  The positional model maintains
    relative states between itself and a reference world and between itself and
    a primary reference point on the reference world.  Positional models have
    health data; thus, they can be damaged, disabled, or destroyed.  In
    addition to encapsulating the concept of a positional model, the class
    contains data and methods required by LaSRS++ for control, operation, and
    identification of positional models. The class also provides the ability
    to record the dynamics of a positional model for playback later.  The
    class can move a model based on a playback file (or a file built using
    the playback format).  The class contains no other event-based dynamics or
    physical modeling.  Derived classes provide these enhancements.
*/
class PositionalModel : public SimulationModel
{
public:
  /** This constructor for the Positional Model class is most commonly called
      by derived classes.  However, it can be called to create a positional
      model which has a fixed position and orientation.
      @param universe Pointer to the parent universe.
      @param model_type String representing the actual positional model type.
      @param model_name String containing the name of this positional model.
      @param cpu_number The number of the CPU that created this positional
      model.
      @param is_alive True if the positional model is still functional.
  */
  PositionalModel(Universe*    universe,
                  const string model_type,
                  const string model_name,
                  int          cpu_number = 0,
                  bool         is_alive   = true);

  /** This constructor for the Positional Model class creates static, playback,
      and traffic models.
      @param universe Pointer to the parent universe.
      @param model_type String representing the actual positional model type.
      @param model_name String containing the name of this positional model.
      @param new_geo_ref_point The primary geographic reference point.
      @param latitude The initial latitude of the model.  The geodetic
      latitude is measured from equatorial plane to surface normal (positive
      North).
      @param longitude The initial longitude of the model.  The longitude is
      the angle in equatorial plane positive east from prime meridian.
      @param altitude The initial altitude of the model.  The altitude is the
      perpendicular height above reference ellipsoid in feet.
      @param pitch_attitude The initial pitch angle of the playback positional
      model.
      @param bank_attitude The initial roll angle of the playback positional
      model.
      @param heading_attitude The initial heading of the playback positional
      model.
      @param model_cg_bias The vertical length that represents how much higher
      or lower the cg of this model is compared to the model used to record a
```

```
    playback tape in feet. For example, if a 747 is recorded but is played
    back as a Cessna, then the cg must be biased so that the Cessna's gear
    are on the ground at the appropriate moments.
    @param cpu_number The number of the CPU that created this positional
    model.
    @param playback_auto_repeat A flag that causes InterpolatingPlayback to
    repeat the playback when it reaches the end of the recording.
    @param is_traffic If traffic selected then create a Playback Traffic
    object, otherwise create a PositionalModelPlayback object which can be a
    static model or a playback model. Traffic automatically repeats itself
    and performs anti - collision computations.
    @param is_incursion A logical that dictates whether or not this
    model will perform an incursion upon the ownship. Defaults to false.
    @param calc_derived_states If true, the playback model will calculate
    all derived states.
    @param delay_before_rewind A double indicating how long the traffic
    playback model should pause when it reaches the end of the playback
    recording before rewinding and starting the playback over in seconds.
    @param disappear_loop An integer that when set tells on which complete
    iteration of the playback sequence that this model should 'disappear'.
    This variable has a default value of -1 which means that the traffic
    playback will not disappear.
    @param is_alive True if the positional model is still functional.
    @param playback_filename The name of the file to use as the source of
    playback data.
*/
PositionalModel(Universe*      universe,
                const string   model_type,
                const string   model_name,
                GeoRefPoint*   new_geo_ref_point,
                double         latitude,
                double         longitude,
                double         altitude,
                double         pitch_attitude,
                double         bank_attitude,
                double         heading_attitude,
                double         model_cg_bias        = 0.0,
                int            cpu_number           = 0,
                bool           playback_auto_repeat = false,
                bool           is_traffic           = false,
                bool           is_incursion         = false,
                bool           calc_derived_states  = false,
                double         delay_before_rewind  = 0.0,
                int            disappear_loop       = -1,
                bool           is_alive             = true,
                const char*    playback_filename    = 0);

/** The destructor for the PositionalModel class deallocates dynamically
    allocated members.
*/
virtual ~PositionalModel();

//*************************************************************************
// POSITIONAL MODEL SYSTEMS AND COMPONENTS
//*************************************************************************

/** This method allows setting the pointer to the data recording system
```

```
    class.
    @param new_data_recording_system The pointer to the data recording
    system being referenced.
*/
void putDataRecordingSystem(DataRecordingSystem* new_data_recording_system);

/** This method allows access to the pointer to the data recording system
    class.  The pointer is exported as a constant object.
    @return The pointer to the data recording system being referenced.
*/
const DataRecordingSystem* getDataRecordingSystem() const;

/** This method allows access to the pointer to the data recording system
    class.  The pointer is exported as a mutable object.
    @return The pointer to the data recording system being referenced.
*/
DataRecordingSystem* getDataRecordingSystem();

//*************************************************************************
// PLAYBACK
//*************************************************************************

/** The ability to record positional model dynamics and play them back is
    encapsulated in the Playback class.  This method provides access to that
    class.
    @return The pointer to the positional model playback class.
*/
PositionalModelPlayback* getPositionalModelPlayback();

//*************************************************************************
// SIMULATION CONTROL DATA
//*************************************************************************

/** This method returns a string that identifies the actual class type of
    the positional model object.  This type identifies the dynamic model;
    i.e., how the model behaves.
    @return String representing the actual positional model type.
 */
const string& getPositionalModelType() const;

/** This method returns a string that identifies the class type to display
    for this positional model object.  This type identifies the visual model;
    i.e., how the model is seen in the virtual environment.  The visual
    representation of the model may be different from its physical dynamics.
    For example, this allows developers to use one aircraft class (dynamic
    model) for may aircraft models in the visual scene (visual models).
    @return String representing the positional model type to display.
 */
const string& getPositionalModelDisplayType() const;

/** This method allows the client to change the visual model of the
    positional model object.  This is useful for running the dynamics of one
    aircraft while the displays represent it as a different aircraft.
    @param display_type String representing the positional model type to
    display.
*/
void putPositionalModelDisplayType(const string& display_type);
```

```
/** Each positional model object is given a unique numeric identifier.  This
    allows the simulation control layer and the GUI to differentiate and
    identify the positional model objects.  This method provides this
    identifier.
    @return The numeric identifier for this positional model.
*/
int  getID() const;

/** Each positional model object is given a unique numeric identifier.  This
    allows the simulation control layer and the GUI to differentiate and
    identify the positional model objects.  This method allows the client to
    set this identifier.
    @param new_id The numeric identifier for this positional model.
*/
void putID(int new_id);

/** This method provides the number of the real-time CPU which created the
    positional model object.
    @return The number of the CPU that created this positional model.
*/
int  getCpuNumber() const;

//*************************************************************************
// INITIAL CONDITIONS
//*************************************************************************

/** When an initial condition has been modified, the vehicle must
    re-initialize. This method provides the flag that signals a change in
    inital conditions.
    @return True if an initial condition has been modified.
*/
bool getModifiedICs() const;

/** When an initial condition has been modified, the vehicle must
    re-initialize. This method allows clients to signal that an initial
    condition has been modified.
    @param new_value True if an initial condition has been modified.
*/
void putModifiedICs(const bool new_value);

/** This method provides the current initial condition case number.
    Positional models may contain hard-coded scenarios that are identified
    using a case number.
    @return The number of the initial condition case currently selected.
*/
unsigned int getCaseNumber() const;

/** This method provides the initialization file name.  The initialization
    file defines a starting scenario for the model.
    @return The name of the current initialization file selected.
*/
const string& getInitializationFileName() const;

/** This method provides the flag that indicates whether a full
    initialization is needed.
    @return True if the initial condition case must be re-evaluated.
```

```
*/
bool isFullInitializationNeeded() const;

/** This method allows clients to set the initialization case number.
    @param new_value The new case number to use.  Positional models may
    contain hard-coded scenarios that are identified using a case number.
*/
void putCaseNumber(unsigned int new_value);

/** This method allows clients to set the initialization file name.  The
    initialization file defines a starting scenario for the model.
    @param new_name The name of the new initialization file to use.  The name
    must be a valid filename for the operating system.  The name may contain
    an absolute or relative path.
*/
void putInitializationFileName(const string& new_name);

/** This method allows clients to signal re-evaluation of the initial
    condition case.
    @param new_value True if a full initialization is needed.
*/
void setFullInitializationNeeded(bool new_value = true);

/** An abstract method that is responsible for applying states stored in the
    'InitializationFileProcessor' to the internal states of the object.  This
    method allows the initial distance from the reference point and the
    initial Euler angles to be set from the initialization file.
    @param file_processor A reference to the InitializationFileProcessor
    object that reads the initialization file to define the initial state of
    the model.
*/
virtual void
   processInitializationFile(InitializationFileProcessor& file_processor);

//*************************************************************************
// HEALTH DATA
//
// This data defines the "health" of the positional model object. Simulations
// that track damage to a positional model use this data (e.g. combat
// simulations).
//*************************************************************************

/** This method provides the flag that indicates whether this positional
    model is still functional.
    @return True if the positional model is alive or false if the positional
    model has been destroyed or disabled.
*/
bool getAlive() const;

/** This method allows access to the number of "hits" this positional model
    has remaining.  This is set to the number of hits needed to kill this
    model and is decremented when a hit is made by another positional model.
    Once this number reaches zero, the model is "killed" and its alive flag
    is set false.
    @return The number of hits this model has remaining.
*/
double getHitPoints() const;
```

```
/** This method provides the number of hits that a target can take before
    it is "killed".
    @return The number of hits needed to kill this positional model.
*/
double getInitialHitPoints() const;

/** This method provides the dimensions of the box around the target in feet.
    This box is centered at the target's CG with the first element of the
    vector the length along the body x axis, the second element the length
    along the body y axis and the third element the height along the body z
    axis.
    @return Dimensions of the box around the target in feet.
*/
const Vector<double>& getStrikeZoneDimensions() const;

/** This method allows clients to set the flag that indicates whether this
    positional model is still functional.
    @param new_alive True if the positional model is alive or false if the
    positional model has been destroyed or disabled.
*/
void putAlive(const bool new_alive);

/** This method allows the client to set the number of "hits" this positional
    model has remaining.  Hit points are initially set to the number of hits
    needed to kill this model and is decremented when a hit is made by
    another positional model.  Once this number reaches zero, the model is
    "killed" and its alive flag is set false.
    @param new_hit_points The number of hits this model has remaining.
*/
void putHitPoints(const double new_hit_points);

/** This method allows the client to set the number of hits a target can
    take before it is "killed".
    @param new_initial_hit_points  The number of hits needed to kill this
    positional model.
*/
void putInitialHitPoints(const double new_initial_hit_points);

/** This method allows the client to set the dimensions of the box around the
    target.  This box is centered at the target's CG with the first element
    of the vector the length along the body x axis, the second element the
    length along the body y axis and the third element the height along the
    body z axis.
    @param new_length New values for the dimensions of the box around the
    target in feet.
*/
void putStrikeZoneDimensions(const Vector<double>& new_dimensions);

//*************************************************************************
// POSITIONAL MODEL DYNAMICS
//
// Mode-based behaviors.
//*************************************************************************

/** This method defines the behavior of the positional model in RESET mode.
    The default behavior resets the alive flag for the model if configured
```

```
    for auto-alive in reset.  The method initializes playback.  If playback
    is in playback mode, places the model at the first condition of the
    playback file.
*/
virtual void doResetCalc();

/** This method defines the behavior of the positional model in TRIM mode.
    The default behavior is no activity.
    @param til_converged The boolean argument determines whether trim is
    calculated in one frame or incrementally over multiple frames.
*/
virtual void doTrimCalc(bool til_converged);

/** This method defines the behavior of the positional model in HOLD mode.
    The default behavior is no activity.
*/
virtual void doHoldCalc();

/** This method defines positional model behavior in OPERATE mode prior to
    the updating of the world states.  The default behavior is no activity.
*/
virtual void doOperateCalc();

/** This method defines positional model behavior in OPEERATE mode after the
    updating of world states.  The default behavior provides only playback
    capabilities.  If playback is active and in record mode, playback records
    the model states.  If playback is active and in playback mode, playback
    sets the model to the next recorded state.
*/
virtual void propagateState();

/** This method defines the positional model behavior in LINEAR_MODEL mode.
    The default behavior is no activity.
*/
virtual void generateLinearModel();

/// This method initializes the positional model to its initial condition.
virtual void initialize();

/// This method sets the initial condition variables to a default state.
virtual void resetInitialConditions();

/** This method creates any systems that a derived model may define.  A
    system is an abstraction of a subsystem of a model such as a propulsion
    system or an electrical system.  By default, this method creates an
    AsciiDisplaySystem object.  This system aids the ASCII user interface in
    displaying the model's states.
*/
virtual void createPositionalModelSystems();

/** This method calculates the world relative states based on the inertial
    states of the positional model and its reference world.
    @param geodetic_coordinates The world relative states of the positional
    model in world coorindates.
    @param world_rel_vel_body The world relative velocity is a combination of
    the difference between the inertial velocities of the positional model
    and the world and the cross  product of the world's rotation and the
```

```
    positional model's position relative to the world in ft/sec.
    @param euler_angles The Euler angles representing rotation from the body
    coordinate system to the local vertical coordinate system.
    @param local_vertical_rel_angular_vel_body The angular velocity of the
    positional model relative to the local vertical frame and expressed in
    body coordinates in angle/sec.
*/
void calcWorldRelStates(
  GeodeticCoordinates&  geodetic_coordinates,
  Vector<double>&       world_rel_vel_body,
  EulerAngles&          euler_angles,
  Vector<AngularValue>& local_vertical_rel_angular_vel_body);

/** This method provides the flag that indicates whether a trim solution
    has been reached.
    @return True when trim has converged.
*/
bool getTrimHasConverged() const;

/** This method provides the flag that indicates whether a state has been
    changed upon which other derived states depend.
    @return False when a state has been changed upon which other derived
    states depend.
*/
bool getDerivedStatesConsistent() const;

/** This method allows the client to set the flag indicating that a trim
    solution has been reached.
    @param new_trim_has_converged True when trim has converged.
*/
void putTrimHasConverged(const bool new_trim_has_converged);

//*****************************************************************************
// INITIAL CONDITIONS
//
// This section provides accessors and mutators for the initial condition
// variables.  The true positional model states are inertial position,
// velocity, orientaton, and angular velocity.  All other states are derived
// from these.  However, the initial condition attempts to maintain initial
// world relative states:  geodetic coordinates, euler angles, world
// relative velocity in body coordinates, and world relative angular
// velocity in body coordinates.  These are the governing initial conditions.
// Thus, changing any initial condition will cause changes in the inertial
// initial conditions in an attempt to maintain the "governing initial
// conditions".  The governing initial conditions are also maintained in the
// order of position, orientation, translational velocity, and angular
// velocity.  Thus, one should specify initial conditions in that order to
// be assured that the positional model will begin with exactly the
// requested state.
//*****************************************************************************

// -----------------------------------------------------------------------------
// The inertial initial conditions.
// -----------------------------------------------------------------------------

/** This method provides the initial inertial position of the model.
    @return The initial inertial position in feet.
```

```
*/
const Vector<double>& getPositionInitialCondition() const;

/** This method sets the initial inertial position.  The method also sets the
    initial geodetic coordinates and the initial distance from the primary
    reference so that they are consistent with the inertial position.  The
    method also changes the initial inertial orientation to maintain initial
    euler angles (i.e., the initial orientation relative to the world).
    @param new_position The new initial inertial position in feet.
*/
void putPositionInitialCondition(const Vector<double>& new_position);

/** This method provides the initial inertial velocity of the model.
    @return The initial inertial velocity in ft/sec.
*/
const Vector<double>& getVelocityInitialCondition() const;

/** This method sets the initial inertial velocity.  Initial world relative
    velocity is also modified assuming that initial geodetic coordinates and
    Euler angles are held constant.
    @param new_velocity The new inertial velocity in ft/sec.
*/
void putVelocityInitialCondition(const Vector<double>& new_velocity);

/** This method provides the initial inertial orientation of the model.
    @return The initial inertial orientation as a quaternion.
*/
const UnitQuaternion& getOrientationInitialCondition() const;

/** This method changes the initial inertial orientation.  The initial Euler
    angles are modified as a result.  The Positional model also attempts to
    maintain the initial geodetic coordinates, world relative velocity in
    body coordinates, and the local vertical angular velocity in body
    coordinates.
    @param new_orientation The new initial inertial orientation as a
    quaterion.
*/
void putOrientationInitialCondition(const UnitQuaternion& new_orientation);

/** This method provides the initial inertial angular velocity of the model
    in body coordinates.
    @return The initial inertial angular velocity in body coordinates in
    angle/sec.
*/
const Vector<AngularValue>& getAngularVelBodyInitialCondition() const;

/** This method changes the initial value for the inertial angular velocity
    in body coordinates. The initial local vertical angular velocity is
    changed as a result.  The PositionalModel also attempts to maintain
    initial geodetic coordinates, initial Euler angles, and initial world
    relative velocity in body coordinates.
    @param new_angular_vel_body The new initial inertial angular velocity
    in angle/sec.
*/
void putAngularVelBodyInitialCondition(
  const Vector<AngularValue>& new_angular_vel_body);
```

```
/** This method allows all inertial initial conditions to be modified with
    one call.
    @param position_ic The new initial position in feet.
    @param velocity_ic The new initial velocity in ft/sec.
    @param orientation_ic The new orientation as a quaterion.
    @param angular_vel_body_ic The new angular velocity in angle/sec.
*/
void putStateICs(const Vector<double>&       position_ic,
                 const Vector<double>&       velocity_ic,
                 const UnitQuaternion&       orientation_ic,
                 const Vector<AngularValue>& angular_vel_body_ic);

// ----------------------------------------------------------------------------
// Governing initial conditions
// ----------------------------------------------------------------------------

/** This method provides the initial distance from primary reference point
    in feet.  The distance is returned as a vector relative to the topodetic
    coordinate system of the reference point.
    @return The initial distance from the primary reference point in feet.
*/
const Vector<double>& getInitDistanceFromRefPoint() const;

/** This method sets the initial distance from the primary geographic
    reference point, if it exists.  If it does not exist, only the initial
    altitude is changed based on the vertical component of the distance
    vector.  The initial geodetic coordinates are calculated from the
    initial distance and set.  Since geodetic coordinates are a derived
    state, the method changes the initial inertial coordinates to be
    consistent with the initial geodetic coordinates.  PositionalModel
    attempts to maintain initial euler angles, initial world relative
    velocity in body coordinates, and initial local vertical angular
    velocity.
    @param new_value The new initial distance from the primary geographic
    reference point in feet.
*/
void putInitDistanceFromRefPoint(const Vector<double>& new_value);

/** This method provides the initial geodetic coordinates.
    @return The initial geodetic coordinates for this positional model.
*/
const GeodeticCoordinates& getInitGeodeticCoordinates() const;

/** This method sets the initial geodetic coordinates.  If a primary
    geographic reference point exists, the the initial distance from the
    reference point is calculated from the initial geodetic coordinate.
    Since geodetic coordinates are a derived state, the method changes the
    initial inertial coordinates to be consistent with the initial geodetic
    coordinates. Positional model attempts to maintain initial euler angles,
    initial world relative velocity in body coordinates, and initial local
    vertical angular velocity.
    @param new_value The new initial geodetic coordinates.
*/
void putInitGeodeticCoordinates(const GeodeticCoordinates& new_value);

/** This method provides the initial Euler angles.
    @return The initial Euler angles for this positional model.
```

```
*/
const EulerAngles& getInitEulerAngles() const;

/** This method sets the initial Euler angles.  Since the Euler angles are
    derived states, the method changes the initial inertial orientation to
    be consistent with the initial Euler angles. PositionalModel also
    attempts to maintain initial geodetic coordinates, initial world
    relative velocity, and initial local vertical relative angular velocity.
    @param new_value The new initial Euler angles.
*/
void putInitEulerAngles(const EulerAngles& new_value);

/** This method provides the initial world-relative velocity represented in
    body coordinates; i.e., initial u, v, and w in ft/sec.
    @return world_rel_vel_body_init The initial u, v, w for this positional
    model in ft/sec.
*/
const Vector<double>& getWorldRelVelBodyInit() const;

/** This method sets the initial world relative velocity in body coordinates.
    Since the world relative velocity is a derived state, the method changes
    the initial inertial velocity to be consistent with the initial world
    relative velocity. PositionalModel also attempts to maintain initial
    Euler  angles, initial geodetic coordinates, and initial local vertical
    angular velocity.
    @param new_velocity The new initial world relative velocity in body
    coordinates in ft/sec.
*/
void putWorldRelVelBodyInit(const Vector<double>& new_velocity);

/** This method provides the initial world relative angular velocity
    represented in body coordinates.
    @return The initial world relative angular velocity represented in body
    coordinates in angle/sec.
*/
const Vector<AngularValue>& getLocalVerticalAngularVelBodyInit() const;

/** This method sets the initial local vertical relative angular velocity in
    body coordinates.  Since the local vertical relative angular velocity is
    is a derived state, the method changes the initial inertial angular
    velocity to be consistent with the initial local vertical relative
    angular velocity. PositionalModel also attempts to maintain initial
    euler angles, initial geodetic coordinates, and initial world relative
    velocity.
    @param new_velocity The new initial local vertical relative angular
    velocity in body coordinates in angle/sec.
*/
void putLocalVerticalAngularVelBodyInit(
  const Vector<AngularValue>& new_velocity);

// ----------------------------------------------------------------------------
// Derived initial conditions.
//
// These are not true initial conditions.  Instead these initial conditions
// modify the governing initial conditions to increase the likelihood that
// a derived state will begin with the given initial value.
// ----------------------------------------------------------------------------
```

```
/** This method returns the initial track angle.
    @return The initial track angle as an angle.
*/
const Angle& getTrackInit() const;

/** This method sets the initial track angle.  Since track is a derived
    state, the method changes the initial inertial states to be consistent
    with the initial track angles. PositionalModel also attempts to maintain
    initial geodetic coordinates, initial world relative velocity in body
    coordinates, and initial local vertical relative angular velocity.
    @param new_track The new initial track angle as an angle.
*/
void putTrackInit(const Angle& new_track);

/** This method returns true if the positional model's initial position was
    last specified as a geodetic coordinate.  Otherwise, the initial position
    was specified relative to the primary reference point.
    @return With a round world, if a primary reference point is set, the
    init_distance_from_ref_point and the init_geodetic_coordinates are kept
    consistent even if one or the other is changed.  If the primary reference
    point is changed, this bool indicates which version of the initial
    position to maintain.  If true, the initial geodetic coordinate is
    maintained.  Otherwise, the initial distance from the primary reference
    point is maintained.
*/
bool isInitPositionInGeodeticCoordinates() const;

//************************************************************************
// POSITIONAL MODEL STATES
//************************************************************************

// ------------------------------------------------------------------------
// ROTATION MATRICES
//
// These rotation matrices transform vectors between inertial, world,
// topodetic, and body axes.
// ------------------------------------------------------------------------

/** This method returns the rotation matrix that transforms vectors from
    inertial coordinates to body coordinates.
    @return The rotation matrix to transform from the inertial axis system
    to the body axis system.
*/
const RotationMatrix& getInertialToBody() const;

/** This method returns the rotation matrix that transforms vectors from
    inertial coordinates to topodetic coordinates.
    @return The rotation matrix to transform from the inerital axis system
    to the topodetic axis system.
*/
const RotationMatrix& getInertialToTopodetic() const;

/** This method returns the rotation matrix that transforms vectors from body
    coordinates to inertial coordinates.
    @return The rotation matrix to transform from the body axis system to
    the inertial axis system.
```

```
*/
const RotationMatrix& getBodyToInertial() const;

/** This method returns the rotation matrix that transforms vectors from body
    coordinates to world coordinates.
    @return The rotation matrix to transform from the body axis system to
    the world axis system.
*/
const RotationMatrix& getBodyToWorld() const;

/** This method returns the rotation matrix that transforms vectors from
    body coordinates to topodetic coordinates.
    @return The rotation matrix to transform from the body axis system to
    the topodetic axis system.
*/
const RotationMatrix& getBodyToTopodetic() const;

/** This method returns the rotation matrix that transforms vectors from
    topodetic coordinates to inertial coordinates.
    @return The rotation matrix to transform from the topodetic axis system
    to the inertial axis system.
*/
const RotationMatrix& getTopodeticToInertial() const;

/** This method returns the rotation matrix that transforms vectors from
    topodetic coordinates to body coordinates.
    @return The rotation matrix to transform from the topodetic axis system
    to the body axis system.
*/
const RotationMatrix& getTopodeticToBody() const;

/** This method returns the rotation matrix that transforms vectors from
    topodetic coordinates to world coordinates.
    @return The rotation matrix to transform from the topodetic axis system
    to the world axis system.
*/
const RotationMatrix& getTopodeticToWorld() const;

/** This method returns the rotation matrix that transforms vectors from
    world coordinates to body coordinates.
    @return The rotation matrix to transform from the world axis system to
    the body axis system.
*/
const RotationMatrix& getWorldToBody() const;

/** This method returns the rotation matrix that transforms vectors from
    world coordinates to topodetic coordinates.
    @return The rotation matrix to transform from the world axis system to
    the topodetic axis system.
*/
const RotationMatrix& getWorldToTopodetic() const;

// ----------------------------------------------------------------------
// INERTIAL AXIS
//
// The most basic states are the position, orientation, translational
// velocity and angular velocity mesaured in inertial coordinates.
```

```
// (NOTE: The inertial angular velocity is represented in body coordinates.)
// ------------------------------------------------------------------------

/** This method provides the inertial position of this positional model.
    @return The positional model inertial position in feet.
*/
const Vector<double>& getPosition() const;

/** This method provides the inertial translational velocity of this
    positional model.
    @return The positional model inertial translational velocity in ft/sec.
*/
const Vector<double>& getVelocity() const;

/** This method provides the inertial orientation of this positional model.
    @return The positional model inertial orientation.
*/
const UnitQuaternion& getOrientation() const;

/** This method provides the inertial angular velocity of this positional
    model represented in body coordinates.
    @return The inertial angular velocity of the positional model represented
    in body coordinates with units of angle/sec.
*/
const Vector<AngularValue>& getAngularVelBody() const;

/** This method allows the client to set the complete inertial state:
    position, orientation, velocity, and angular velocity.
    @param position The inertial position of this positional model in feet.
    @param velocity The inertial translation velocity of this positional
    model in ft/sec.
    @param orientation The inertial orientation of this positional model.
    @param angular_vel_body The inertial angular velocity in body
    coordinates of this positional model in angle/sec.
*/
void putStates(const Vector<double>&       position,
               const Vector<double>&       velocity,
               const UnitQuaternion&       orientation,
               const Vector<AngularValue>& angular_vel_body);

// ------------------------------------------------------------------------
// WORLD AXIS
//
// World relative position is stored in the GeodeticCoordinates object (see
// TOPODETIC AXIS).
// ------------------------------------------------------------------------

/** This method returns the distance between the positional model's CG and
    the center of the reference world.
    @return The distance of the PositionalModel CG from the center of the
    reference world in feet.
*/
double getGeocentricAltitude() const;

/** This method provides the world relative velocity in world coordinates.
    @return The world relative velocity in world coordinates in ft/sec.
*/
```

```
const Vector<double>& getWorldRelVel() const;

/** This method provides the magnitude of the world relative velocity; i.e.,
    the world relative speed.
    @return The magnitude of the world relative velocity; i.e., the world
    relative speed in ft/sec.
*/
double getWorldRelVelMag() const;

/** This method provides the rate of change of the world relative speed.
    @return The rate of change of the world relative speed (not velocity) in
    ft/sec^2.
*/
double getWorldRelVelRate() const;

/** This method sets the world relative position.  Since the world relative
    position is a derived state, this method changes the underlying inertial
    states so that the new world relaitve position will be calculated the
    next time calcDerivedStates() is called.  World relative velocity in
    body coordinates (u, v, w), the Euler angles, and the local vertical
    relative angular velocity are held constant.
    @param new_value The new world relative position of thie positional
    model in feet.
*/
void putWorldRelVector(const Vector<double>& new_value);

/** This method sets the rate of change of the world relative speed.
    @param new_value The new value of the world relative acceleration in
    ft/sec^2.
*/
void putWorldRelVelRate(double new_value);

/** This method sets the underlying inertial states based on the world
    relative state arguments.
    @param new_geodetic_coordinates The new position of the PositionalModel
    relative to the world.
    @param new_world_rel_vel_body The new world relative velocity expressed
    in body coordinates in ft/sec.  The literature commonly refers to the
    vector as [U, V, W]
    @param new_euler_angles The new Euler angles representing rotation from
    the body coordinate system to the local vertical coordinate system.
    @param new_local_vertical_rel_angular_vel_body The new angular velocity
    of the positional model relative to the local vertical frame and
    expressed in body coordinates in angle/sec.
*/
void putWorldRelStates(
  const GeodeticCoordinates&  new_geodetic_coordinates,
  const Vector<double>&       new_world_rel_vel_body,
  const EulerAngles&          new_euler_angles,
  const Vector<AngularValue>& new_local_vertical_rel_angular_vel_body);

// ----------------------------------------------------------------------------
// TOPODETIC AXIS
// ----------------------------------------------------------------------------

/** This method returns the orientation of the positional model relative to
    the topodetic coordinate system.
```

```
    @return The orientation of the body axis relative to the topodetic axis.
*/
const UnitQuaternion& getTopodeticRelOrientation() const;

/** This method returns the geodetic coordinates of the CG of the positional
    model object.
    @return The position of the PositionalModel relative to the world.
    GeodeticCoordinates actually contains both topodetic and world axis
    positions.
*/
const GeodeticCoordinates& getGeodeticCoordinates() const;

/** This method returns the true altitude above sea level.
    @return The geodetic altitude in feet.
*/
double getAltitude() const;

/** This method returns the rate of change of latitude.
    @return The rate of change of latitude in angle/sec.
*/
const AngularValue& getLatitudedot() const;

/** This method returns the rate of change of longitude.
    @return The rate of change of longitude in angle/sec.
*/
const AngularValue& getLongitudedot() const;

/** This method returns the world relative velocity in topodetic coordinates.
    @return The world relative velocity expressed in topodetic coordinates
    in ft/sec.
*/
const Vector<double>& getWorldRelVelTopodetic() const;

/** This method returns the world relative acceleration in topodetic
    coordinates.
    @return The world relative acceleration expressed in topodetic
    coordinates in ft/sec^2.
*/
const Vector<double>& getWorldRelAccelTopodetic() const;

/** This method returns the world relative angular velocity of the local
    vertical frame in topodetic coordinates.
    @return The world relative angular velocity of the local vertical frame
    expressed in topodetic coordinates in angle/sec.
*/
const Vector<AngularValue>& getWorldRelLocalVertAngularVelTopo() const;

/** This method returns the climb rate; i.e. the rate of change of altitude.
    @return The climb rate in ft/sec.
*/
double getAltitudeDot() const;

/** This method returns the ground speed.
    @return The speed of the positional model parallel to the ground in
    ft/sec.
*/
double getGroundSpeed() const;
```

```
/** This method returns the track angle.  The tangent of the track angle is
    the ratio of the East velocity over the North velocity of the positional
    model.
    @return The velocity track; i.e., tangent of East velocity over North
    velocity as an angle.
*/
const Angle& getTrack() const;

/** This method sets the world relative position.  Since the world relative
    position is a derived state, this method changes the underlying inertial
    states so that the new world relative position will be calculated the
    next time calcDerivedStates() is called.  World relative velocity in
    body coordinates (u, v, w), the Euler angles, and the local vertical
    relative angular velocity are held constant.
    @param new_value The new world relative position.
*/
void putGeodeticCoordinates(const GeodeticCoordinates& new_value);

/** This method sets the local vertical angular velocity.  Since the local
    vertical angular velocity is a derived state, the method changes the
    underlying inertial states so that the new local vertical angular
    velocity will be calculated the next time calcDerivedStates() is called.
    This routine keeps the world relative velocity in body coordinates
    (u, v, w), the Euler angles, and the geodetic coordinates constant.
    @param new_value the new local vertical anglular velocity in angle/sec.
*/
void putLocalVerticalRelAngularVelBody(
  const Vector<AngularValue>& new_value);

/** This method sets the local vertical roll rate.  Since the local vertical
    roll rate is a derived state, the method changes the underlying inertial
    states so that the new local vertical roll rate will be calculated the
    next time calcDerivedStates() is called.  This routine keeps the world
    relative velocity in body coordinates (u, v, w), the Euler angles, and
    the geodetic coordinates constant.
    @param new_value The new value of the local vertical roll rate in
    angle/sec.
*/
void putLocalVerticalRelP(const AngularValue& new_value);

/** This method sets the local vertical pitch rate.  Since the local
    vertical pitch rate is a derived state, the method changes the
    underlying inertial states so that the new local vertical pitch rate
    will be calculated the next time calcDerivedStates() is called.  This
    routine keeps the world relative velocity in body coordinates (u, v, w),
    the world relative  orientation (Euler angles), and the geodetic
    coordinates constant.
    @param new_value The new value of the local vertical pitch rate in
    angle/sec.
*/
void putLocalVerticalRelQ(const AngularValue& new_value);

/** This method sets the local vertical yaw rate.  Since the local vertical
    yaw rate is a derived state, the method changes the underlying inertial
    states so that the new local vertical yaw rate will be calculated the
    next time calcDerivedStates() is called.  This routine keeps the world
```

```
    relative velocity in body coordinates (u, v, w), the Euler angles, and
    the geodetic coordinates constant.
    @param new_value The new value of the local vertical yaw rate in
    angle/sec.
*/
void putLocalVerticalRelR(const AngularValue& new_value);

/** This method sets the altitude.  Since the altitude is a derived state,
    this method changes the underlying inertial states so that the new
    altitude will be calculated the next time calcDerivedStates() is called.
    World relative velocity in body coordinates (u, v, w), the Euler angles,
    and the local vertical relative angular velocity are held constant.
    @param new_altitude The new positional model altitude in feet.
*/
void putAltitude(double new_altitude);

/** This method sets the climb rate.  Since climb rate is a derived state,
    this method changes the underlying inertial states so that the new climb
    rate will be calculated the next time calcDerivedStates() is called.
    World relative -position (geodetic coordinates), the Euler angles, and
    the local vertical relative angular velocity are held constant.
    @param feet_per_second The new climb rate in ft/sec.
*/
void putAltitudeDot(double feet_per_second);

/** This method sets the track angle.  Since track is a derived state, this
    method changes the underlying inertial states so that the new track will
    be calculated the next time calcDerivedStates() is called.  World
    relative position (geodetic coordinates), the world relative velocity in
    body coordinates, and the local vertical relative angular velocity are
    held constant.
    @param new_track The new track angle.
*/
void putTrack(const Angle& new_track);

// ----------------------------------------------------------------------------
// BODY AXIS
// ----------------------------------------------------------------------------

/** This method returns the world relative velocity in body coordinates.
    @return The world relative velocity expressed in body coordinates in
    ft/sec.   The literature commonly refers to the vector as [U, V, W].
*/
const Vector<double>& getWorldRelVelBody() const;

/** This method returns the rate of change of the world relative velocity
    from the perspective of the body coordinate system.  This vector accounts
    for both the world relative accelerations and apparent changes in the
    body-axis components of the world relative velocity due to the rotation
    of the body coordinate axis.  The literature commonly refers to this
    vector as [udot, vdot, wdot].
    @return The rate of change of the world relative velocity from the
    perspective of the body coordinate system in ft/sec^2.
*/
const Vector<double>& getBodyDerivWorldRelVelBody() const;

/** This method returns the world relative angular velocity in body
```

```
    coordinates.
    @return The world relative angular velocity expressed in body
    coordinates in angle/sec. The literature refers to this vector as
    [p, q, r].
*/
const Vector<AngularValue>& getWorldRelAngularVelBody() const;

/** This method returns the local vertical relative angular velocity in body
    coordinates.
    @return The angular velocity of the positional model relative to the
    local vertical frame and expressed in body coordinates in angle/sec.
*/
const Vector<AngularValue>& getLocalVerticalRelAngularVelBody() const;

/** This method returns the Euler angles.  The Euler angles represent the
    rotation from body to topodetic coordinates (or the instantaneous local
    vertical coordinate system).
    @return The Euler angles representing rotation from the body coordinate
    system to the local vertical coordinate system.
*/
const EulerAngles& getEulerAngles() const;

/** This method returns the rate of change of the Euler angles.  The
    literature refers to this collection of angular rates as
    [phi_dot, theta_dot, psi_dot].
    @return The rate of change of the Euler angles in angle/sec.
*/
const Vector<AngularValue>& getEulerAngleRates() const;

/** This method returns the rate of change of phi.
    @return The rate of change of the roll angle in angle/sec.
*/
const AngularValue& getPhidot() const;

/** This method returns the rate of change of theta.
    @return The rate of change of the pitch angle in angle/sec.
*/
const AngularValue& getThetadot() const;

/** This method returns the rate of change of psi.
    @return The rate of change of the heading in angle/sec.
*/
const AngularValue& getPsidot() const;

/**This method sets the world relative velocity in body coordinates.  Since
   the world relative velocity in body coordinates is a derived state, this
   method changes the underlying inertial states so that the new world
   relative velocity in body coordinates will be calculated the next time
   calcDerivedStates() is called.  World relative position (geodetic
   coordinates), the Euler angles, and the local vertical relative angular
   velocity are held constant.
   @param new_value The new value of the world relative velocity in body
   coordinates in ft/sec.
*/
void putWorldRelVelBody(const Vector<double>& new_value);

/** This method sets world relative velocity along the X body axis (u).
```

```
    Since u is a derived state, this method changes the underlying inertial
    states so that the new u will be calculated the next time
    calcDerivedStates() is called.  World relative position (geodetic
    coordinates), the Euler angles, and the local vertical relative angular
    velocity are held constant.
    @param feet_per_second The new value of the world relative velocity along
    the X body axis in ft/sec.
*/
void putU(double feet_per_second);

/** This method sets world relative velocity along the Y body axis (v).
    Since v is a derived state, this method changes the underlying inertial
    states so that the new v will be calculated the next time
    calcDerivedStates() is called.  World relative position (geodetic
    coordinates), the Euler angles, and the local vertical relative angular
    velocity are held constant.
    @param feet_per_second The new value of the world relative velocity along
    the Y axis in ft/sec.
*/
void putV(double feet_per_second);

/** This method sets world relative velocity along the Z body axis (w).
    Since w is a derived state, this method changes the underlying inertial
    states so that the new w will be calculated the next time
    calcDerivedStates() is called.  World relative position (geodetic
    coordinates), the Euler angles, and the local vertical relative angular
    velocity are held constant.
    @param feet_per_second The new value of the world relative velocity along
    the Z body axis in ft/sec.
*/
void putW(double feet_per_second);

/** This method sets the rate of change of the world relative velocity from
    the perspective of the body coordinate system. The literature commonly
    refers to this vector as [udot, vdot, wdot].
    @param new_value The new value of the rate of change of the world
    relative velocity from the perspective of the body coordinate system in
    ft/sec^2.
*/
void putBodyDerivWorldRelVelBody(const Vector<double>& new_value);

/** This method sets the Euler angles.  Since the Euler angles are derived
    states, the method changes the underlying inertial states so that the
    new euler angles will be calculated the next time calcDerivedStates() is
    called.  World relative position (geodetic coordinates), world relative
    velocity (u, v, w), and local vertical relative angular velocity are
    held constant.
    @param new_value The new Euler angles.
*/
void putEulerAngles(const EulerAngles& new_value);

/** This method sets psi.  Since psi is a derived state, the method changes
    the underlying inertial states so that the new psi will be calculated
    the next time calcDerivedStates() is called.  World relative position
    (geodetic coordinates), world relative velocity (u, v, w), and local
    vertical relative angular velocity are held constant.
    @param new_value The new heading as an angle.
```

```
*/
void putPsi(const Angle& new_value);

/** This method sets theta.  Since theta is a derived state, the method
    changes the underlying inertial states so that the new theta will be
    calculated the next time calcDerivedStates() is called.  World relative
    position (geodetic coordinates), world relative velocity (u, v, w), and
    local vertical relative angular velocity are held constant.
    @param new_value The new pitch angle as an angle.
*/
void putTheta(const Angle& new_value);

/** This method sets phi.  Since phi is a derived state, the method changes
    the underlying inertial states so that the new phi will be calculated
    the next time calcDerivedStates() is called.  World relative position
    (geodetic coordinates), world relative velocity (u, v, w), and local
    vertical relative angular velocity are held constant.
    @param new_value The new roll angle as an angle.
*/
void putPhi(const Angle& new_value);

/** This method sets the world relative angular velocity in body coordinates.
    Since the world relative angular velocity in body coordinates is a
    derived state, the method changes the underlying inertial states so that
    the new world relative angular velocity in body coordinates will be
    calculated the next time calcDerivedStates() is called.  The Euler
    angles, the world relative position (geodetic coordinates), and world
    relative velocity (u, v, w) are held constant.
    @param new_value The new world relative angular velocity in body
    coordinates in angle/sec.
*/
void putWorldRelAngularVelBody(const Vector<AngularValue>& new_value);

/** This method sets the world relative roll rate.  Since the world relative
    roll rate is a derived state, the method changes the underlying inertial
    states so that the new world relative roll rate will be calculated the
    next time calcDerivedStates() is called.  The Euler angles, the world
    relative position (geodetic coordinates), and world relative velocity
    (u, v, w) are held constant.
    @param new_value The new roll rate in angle/sec.
*/
void putP(const AngularValue& new_value);

/** This method sets the world relative pitch rate.  Since the world
    relative pitch rate is a derived state, the method changes the
    underlying inertial states so that the new world relative pitch rate
    will be calculated the next time calcDerivedStates() is called.  The
    Euler angles, the world relative position (geodetic coordinates), and
    world relative velocity (u, v, w) are held constant.
    @param new_value The new pitch rate in angle/sec.
*/
void putQ(const AngularValue& new_value);

/** This method sets the world relative yaw rate.  Since the world relative
    yaw rate is a derived state, the method changes the underlying inertial
    states so that the new world relative yaw rate will be calculated the
    next time calcDerivedStates() is called.  The Euler angles, world
```

```
    relative position (geodetic coordinates), and world relative velocity
    (u, v, w) are held constant.
    @param new_value The new yaw rate in angle/sec.
*/
void putR(const AngularValue& new_value);

/** This method sets the Euler angle rates.  Since the Euler angle rates are
    derived state, the method changes the underlying inertial states so that
    the new euler angle rates will be calculated the next time
    calcDerivedStates() is called.  The geodetic coordinates, world relative
    velocity (u, v, w), and Euler angles are held constant.
    @param new_euler_angle_rates The new Euler angle rates in angle/sec.
*/
void putEulerAngleRates(const Vector<AngularValue>& new_euler_angle_rates);

/** This method sets the rate of change of phi.  Since phidot is a derived
    state, the method changes the underlying inertial states so that the new
    "phidot" will be calculated the next time calcDerivedStates() is called.
    The method attempts to hold geodetic coordinates, world relative
    velocity, and Euler angles constant.
    @param new_phidot The new rate of change of the roll angle in angle/sec.
*/
void putPhidot(const AngularValue& new_phidot);

/** This method sets the rate of change of theta.  Since thetadot is a
    derived state, the method changes the underlying inertial states so
    that the new "thetadot" will be calculated the next time
    calcDerivedStates() is called.  The method attempts to hold geodetic
    coordinates, world relative velocity, and Euler angles constant.
    @param new_thetadot The new value of the rate of change of the pitch
    angle in angle/sec.
*/
void putThetadot(const AngularValue& new_thetadot);

/** This method sets the rate of change of psi.  Since psidot is a derived
    state, the method changes the underlying inertial states so that the new
    "psidot" will be calculated the next time calcDerivedStates() is called.
    The method attempts to hold geodetic coordinates, world relative
    velocity, and Euler angles constant.
    @param new_psidot The new value of the rate of change of the heading in
    angle/sec.
*/
void putPsidot(const AngularValue& new_psidot);

//************************************************************************
// REFERENCE POINT
//************************************************************************

/** This method returns the primary geographic reference point.
    @return The pointer to the primary geographic reference point.
*/
const GeoRefPoint* getPrimaryGeoRefPoint() const;

/** This method returns the "handle" to the relative geometry information
    between the positional model object and its primary reference point.
    @return The "handle" to the relative geometry between the primary
    geographic reference point and the positional model.
```

```
*/
const GeoRefRelInfoHandle* getPrimaryGeoRefRelInfoHandle() const;

/** This method returns the velocity relative to the primary geographic
    reference point. The vector is represented in the "local" (i.e. body)
    coordinate system of the reference point.
    @return The velocity relative to primary reference point in the "local"
    coordinate system anchored on the reference point in ft/sec.  In LaSRS,
    this vector was called [sxdot, sydot, szdot].
*/
const Vector<double>& getRefPointRelVel() const;

/** This method returns the position relative to the primary geographic
    reference point. The vector is represented in the "local" (i.e. body)
    coordinate system of the reference point.
    @return The position of the positional model relative to the primary
    reference point represented in the "local" coordinate of the reference
    point in feet.  In LaSRS, this vector was called [sx, sy, sz].
*/
const Vector<double>& getPrimaryRefPointRelativePosition()  const;

/** This method sets the primary geographic reference point.
    @param geo_ref_point The pointer to the primary geographic reference
    point.
*/
void  putPrimaryGeoRefPoint(const GeoRefPoint* geo_ref_point);

/** This method sets the world relative position as a distance from the
    primary reference point.  Since the world relative position is a derived
    state, this method changes the underlying inertial states so that the
    new distance from the primary reference point will be calculated the
    next time calcDerivedStates() is called.  World relative velocity in
    body coordinates (u, v, w), the Euler angles, and the local vertical
    relative angular velocity are held constant.  The vector is measured in
    the "local" (i.e. body) coordinate system of the primary reference point.
    @param new_value The new position as a vector distance from the primary
    reference point measured from the primary reference point's local
    coordinate system.  Units are feet.
*/
void putDistanceFromRefPoint(const Vector<double>& new_value);

/** This method sets the X distance from the primary reference point.  The
    distance is measured along the X axis of the "local" (i.e., body)
    coordinate system of the primary reference point.  Since the X distance
    from the primary reference point is a derived state, this method changes
    the underlying inertial states so that the new X distance from the
    primary reference point will be calculated the next time
    calcDerivedStates() is called.  World relative velocity in body
    coordinates (u, v, w), the Euler angles, and the local vertical relative
    angular velocity are held constant.
    @param distance_feet The new distance from the primary reference point
    along the X axis of the "local" coordinate system of the primary
    reference point.  Units are in feet.
*/
void putSx(double distance_feet);

/** This method sets the Y distance from the primary reference point.  The
```

```
    distance is measured along the Y axis of the "local" (i.e., body)
    coordinate system of the primary reference point.  Since the Y distance
    from the primary reference point is a derived state, this method changes
    the underlying inertial states so that the new Y distance from the
    primary reference point will be calculated the next time
    calcDerivedStates() is called.  World relative velocity in  body
    coordinates (u, v, w), the Euler angles, and the local vertical relative
    angular velocity are held constant.
    @param distance_feet The new distance from the primary reference point
    along the Y axis of the "local" coordinate system of the primary
    reference point.  Units are in feet.
*/
void putSy(double distance_feet);

/** This method sets the vertical distance from the primary reference point.
    Since the vertical distance from the primary reference point is a derived
    state, this method changes the underlying inertial states so that the
    new vertical distance from the primary reference point will be calculated
    the  next time calcDerivedStates() is called.  World relative velocity in
    body coordinates (u, v, w), the Euler angles, and the local vertical
    relative angular velocity are held constant.  The vector is measured in
    the "local" (i.e. body) coordinate system of the primary reference point.
    @param distance_feet The new vertical distance from the primary
    reference point in feet.
*/
void putSz(double distance_feet);

/** This method sets the velocity relative to the primary reference point.
    Since the reference point relative velocity is a derived state, the
    method changes the underlying inertial states so that the new reference
    point relative velocity will be calculated the next time
    calcDerivedStates() is called.  The Euler angles, the world relative
    positon (geodetic coordinates), and the local vertical relative angular
    velocity are held constant.  The vector is measured in the local
    coordinate system of the primary reference point.
    @param new_value The new velocity relative to the primary reference
    point in ft/sec.
*/
void putRefPointRelVel(const Vector<double>& new_value);

/** This method sets the X velocity relative to the primary reference
    point.  The velocity is measured along the X axis of the "local" (i.e.,
    body) coordinate system of the primary reference point.  Since the
    reference point relative X velocity is a derived state, the method
    changes the underlying inertial states so that the new reference point
    relative X velocity will be calculated the next time calcDerivedStates()
    is called.  The Euler angles, the world relative position (geodetic
    coordinates), and local vertical relative angular velocity are held
    constant.
    @param feet_per_second The new velocity relative to the primary
    reference point measured along the X axis of the primary reference
    point's local coordinate system.  Units are in ft/sec.
*/
void putSxdot(double feet_per_second);

/** This method sets the Y velocity relative to the primary reference point.
    The velocity is measured along the Y axis of the "local" (i.e., body)
```

```
    coordinate system of the primary reference point.  Since the reference
    point relative Y velocity is a derived state, the method changes the
    underlying inertial states so that the new reference point relative Y
    velocity will be calculated the next time calcDerivedStates() is called.
    The Euler angles, the world relative position (geodetic coordinates),
    and the local vertical relative angular velocity are held constant.
    @param feet_per_second The new velocity relative to the primary
    reference point measured along the Y axis of the primary reference
    point's local coordinate system.  Units are in ft/sec.
*/
void putSydot(double feet_per_second);

/** This method sets the vertical velocity relative to the primary reference
    point. Since the reference point relative vertical velocity is a
    derived state, the method changes the underlying inertial states so that
    the new reference point relative vertical velocity will be calculated
    the next time calcDerivedStates() is called.  The Euler angles, the
    world relative position (geodetic coordinates), and the local vertical
    relative angular velocity are held constant.  The vector is measured in
    the local coordinate system of the primary reference point.
    @param feet_per_second The new vertical velocity relative to the primary
    reference point in ft/sec.
*/
void putSzdot(double feet_per_second);

//*************************************************************************
// HEIGHT ABOVE TERRAIN
//*************************************************************************

/** This method returns the height above terrain.
    @return The distance between the terrain and the positional model CG as
    measured along the topodetic Z-axis in feet.
*/
double getHeightAboveTerrain() const;

/** This method calculates and returns the current height above terrain.
    @param body_rel_position The position of the positional model relative to
    the primary geographic reference point in feet.
    @return The distance between the terrain and the positional model CG as
    measured along the topodetic Z-axis in feet.
*/
double getHeightAboveTerrain(const Vector<double>& body_rel_position) const;

/** This method allows the client to set the data relevant to the height
    above terrain.
    @param new_height_above_terrain The new distance between the terrain and
    the positional model CG as measured along the topodetic Z-axis in feet.
    @param new_polygon_normal Normalized polygon directly "below" the motion
    system.
    @param new_height_above_terrain_time_tag Timer used to determine if the
    height above terrain calculation is valid.
*/
void putHeightAboveTerrainData(
  const double&         new_height_above_terrain,
  const Vector<double>& new_polygon_normal,
  const Timer*          new_height_above_terrain_time_tag);
```

```
/** This method returns the flag indicating that a valid height above the
    terrain was computed.
    @return True if a valid value for the height above the terrain was
    returned.
*/
bool heightAboveTerrainValid() const;

/** This method returns a z-axis unit vector in topodetic coordinates.
    @return A z-axis unit vector in topodetic coordinates.
*/
Vector<double> getTerrainPolygonNormal() const;

/** This method returns the height above terrain time tag.
    @return The time tag used in the determination of the height above
    terrain validity.
*/
const Timer* getHeightAboveTerrainTimeTag() const;

//************************************************************************
// RELATIVE GEOMETRY SETTINGS
//
// These flags deterimine how relative geometry is conducted between this
// positional model and other positional models in the simulation.
//************************************************************************

/** This method determines whether relative geometry information will be
    calculated from the CG of the positional model to the CG of the target.
    @param new_value True to compute the relative geometry information
    between the positional model CG and the target CG.
*/
void setComputeCGToCGRelGeomFlag(bool new_value);

/** This method determines whether relative geometry information will be
    calculated from the pilot location of the positional model to the CG of
    the target.  (For target image generation displays).
    @param new_value True to compute relative geometry between pilot
    location and positional model CG.
*/
void setComputePilotToTargetCGRelGeomFlag(bool new_value);

/** This method returns true if relative geometry is calculated between the
    CG of the positional model and the CG of the target.
    @return True if relative geometry is calculated between the CG of the
    positional model and the CG of the target.
*/
bool computeCGToCGRelGeomFlag() const;

/** This method returns true if relative geometry is calculated between the
    pilot location on the positional model and the CG of the target.
    @return True if relative geometry is calculated between the pilot
    location on the positional model and the CG of the target.
*/
bool computePilotToTargetCGRelGeomFlag() const;

//************************************************************************
// PILOT
//
```

```
// Pilot models information about human operators.  This information
// facilitates relative geometry between the pilot and other important
// objects in the environment (for accurate visual displays).  Forces at the
// pilot station are also maintained for motion simulation and blackout.
// A client or derived class sets the pilot.
//**************************************************************************

/** This method returns a pointer to the Pilot object in the positional
    model; the Pilot is exported as a constant object.
    @return The pointer to the pilot as a constant object.
*/
const Pilot* getPilot() const;

/** This method returns a pointer to a non-constant Pilot object in the
    positional model.
    @return The pointer to the pilot object.
*/
Pilot* getPilot();

/** This method attaches a pilot to the positional model.
    @param new_pilot The new pilot associated with this positional model.
*/
void putPilot(Pilot* new_pilot);

//**************************************************************************
// ENVIRONMENT
//**************************************************************************

/** This method returns a pointer to the parent Universe.  The Universe is
    exported as a constant object.
    @return Reference to the parent universe as a constant object.
*/
const Universe* getUniverse() const;

/** This method returns a pointer to the parent Universe.  The Universe is
    exported as a mutable object.
    @return Reference to the parent universe.
*/
Universe* getUniverse();

/** This method returns a pointer to the reference world of the positional
    model object.  The world is exported as a constant object.
    @return Reference to the parent world as a constant object.
*/
const World* getWorld() const;

/** This method returns a pointer to the reference world of the positional
    model object.  The world is exported as a mutable object.
    @return Reference to the parent world.
*/
World* getWorld();

/** This method returns the magnetic variation at the CG location of the
    positional model.  Magnetic variation is positive magnetic north west of
    true north.
    @return Magnetic variation at the current geodetic coordinates.
*/
```

```
  AngularValue getMagneticVariation() const;

  //************************************************************************
  // ASCII DISPLAY SYSTEM
  //************************************************************************

  /** This method allows external access to the ascii display system.
      @return Pointer to the ascii display system.
  */
  AsciiDisplaySystem* getAsciiDisplaySystem();

  //************************************************************************
  // DEPRECATED METHODS
  //************************************************************************
  // These methods are obsolete and are retained for backward compatibility
  // only.  They will be removed in the future.
  virtual void copyDataToDataRamFile();
  virtual void copyDataRamFileToPhysicalFile();

protected:
  //************************************************************************
  // INITIAL CONDITION
  //************************************************************************

  /** This method calculates the inertial and derived initial conditions from
      the "governing" initial conditions.  The method is designed to keep all
      initial condition variables physically consistent.  When a derived class
      introduces a new initial condition, the mutator for the new condition
      should back out one of the governing initial conditions and call this
      method.  The derived class should also redefine this method to add the
      equations that calculate the new initial condition from the governing
      initial conditions.
  */
  virtual void calcInitialConditions();

  //************************************************************************
  // MODEL DYNAMICS
  //************************************************************************

  /** This method calculates world relative and primary reference point
      relative states from the inertial states.
  */
  virtual void calcDerivedStates();

  /** This method calculates the world relative angular velocity in body
      coordinates given the geodetic coordinates, the world relative velocity
      in body coordinates, the Euler angles, and the local vertical relative
      angular velocity in body coordinates.
      @param geodetic_coordinates The world relative location of the positional
      model in world coorindates.
      @param world_rel_vel_body The world relative velocity is a combination of
      the difference between the inertial velocities of the positional model
      and the world and the cross  product of the world's rotation and the
      positional model's position relative to the world.  Units are ft/sec.
      @param euler_angles The Euler angles represent the rotation from the body
      coordinate system to the local vertical coordinate system.
      @param local_vertical_rel_ang_vel_body The angular velocity of the
```

```
    positional model relative to the local vertical frame.  It is expressed
    in body coordinates.  Units are angle/sec.
*/
Vector<double> calcWorldRelAngularVelBody(
  const GeodeticCoordinates&  geodetic_coordinates,
  const Vector<double>&       world_rel_vel_body,
  const EulerAngles&          euler_angles,
  const Vector<AngularValue>& local_vertical_rel_ang_vel_body) const;

/** This method calculates the local vertical relative angular velocity in
    body coordinates based on the geodetic coordinates, the world relative
    velocity in body coordinates, the Euler angles, and the world relative
    angular velocity in body coordinates.
    @param geodetic_coordinates The world relative location of the positional
    model in world coorindates.
    @param world_rel_vel_body The world relative velocity is a combination of
    the difference between the inertial velocities of the positional model
    and the world and the cross  product of the world's rotation and the
    positional model's position relative to the world.  Units are ft/sec.
    @param euler_angles The Euler angles represent the rotation from the body
    coordinate system to the local vertical coordinate system.
    @param world_rel_angular_vel_body The world relative angular velocity
    expressed in body coordinates.  Units are angle/sec.
*/
Vector<double> calcLocalVerticalRelAngularVelBody(
  const GeodeticCoordinates&  geodetic_coordinates,
  const Vector<double>&       world_rel_vel_body,
  const EulerAngles&          euler_angles,
  const Vector<AngularValue>& world_rel_angular_vel_body) const;

/// Compute the position integral for the positional model.
void integratePosition();

/// Compute the velocity integral for the positional model.
void integrateVelocity();

/// Compute integral for the orientation quaternion of the positional model.
void integrateOrientation();

/// Compute the integral for the angular velocity of the positional model.
void integrateAngularVelBody();

//************************************************************************
// POSITIONAL MODEL STATES
//
// These methods allow derived classes to set the integrands that represent
// the inertial states.
//************************************************************************

/** This method sets the integrand representing the inertial position.
    @param new_integrand The new integrand that computes the inertial
    position of the positional model in feet.
*/
void putPosition(IntegrandTemplate< Vector<double> >* new_integrand);

/** This method sets the integrand representing the inertial velocity.
    @param new_integrand The new integrand that compute the inertial
```

```
    velocity of the positional model in ft/sec.
*/
void putVelocity(IntegrandTemplate< Vector<double> >* new_integrand);

/** This method sets the integrand representing the inertial orientation.
    @param new_integrand The new integrand that computes the inertial
    orientation of the positional model.
*/
void putOrientation(IntegrandTemplate< UnitQuaternion >* new_integrand);

/** This method sets the integrand representing the inertial angular velocity
    in body coordinates.
    @param new_integrand The new integrand that computes the inertial
    angular velocity of the positional model in body coordinates in
    angle/sec.
*/
void putAngularVelBody(
IntegrandTemplate< Vector<AngularValue> >* new_integrand);

// ----------------------------------------------------------------------------
// These methods allow derived classes to change the current value of the
// inertial state.
// ----------------------------------------------------------------------------

/** This method sets the inertial position in feet.
    @param new_position The new inertial position in feet.
*/
void putPosition(const Vector<double>& new_position);

/** This method sets the inertial velocity in ft/sec.
    @param new_velocity The new inertial velocity in ft/sec.
*/
void putVelocity(const Vector<double>& new_velocity);

/** This method sets the inertial orientation as a quaternion.
    @param new_orientation The new inertial orientation.
*/
void putOrientation(const UnitQuaternion& new_orientation);

/** This method sets the inertial angular velocity in body coordinates.
    @param new_angular_vel_body The new angular velocity in body coordinates
    in angle/sec.
*/
void putAngularVelBody(const Vector<AngularValue>& new_angular_vel_body);

/** This method allows derived classes to set the world relative acceleration
    in topodetic coordinates.
    @param new_value The new world relative acceleration in topodetic
    coordinates in ft/sec^2.
*/
void putWorldRelAccelTopodetic(const Vector<double>& new_value);

/** This method is generally used with an argument value of false.  This
    indicates that a state was changed and the derived states must be
    recalculated. However, derived classes will set this value to true after
    derived states have been calculated.
    @param new_value True if the derived states have been calculated after
```

```
        the last change to the inertial states.
    */
    void putDerivedStatesConsistent(bool new_value);

    //************************************************************************
    // REFERENCE POINT
    //************************************************************************

    /** This method allows derived class to retrieve the list of reference point
        relative state objects from the base class.
        @return List of relative geometry information between the positional
        model and geographic reference points.
    */
    GeoRefRelInfoList* getGeoRefRelInfoList();

    //************************************************************************
    // ASCII DISPLAY SYSTEM
    //************************************************************************

    /** This method allows derived classes to insert a new ascii display system
        into this class for use.
        @param new_system The new ascii display system to use.
    */
    void putAsciiDisplaySystem(AsciiDisplaySystem* new_system);

  private:
    //************************************************************************
    // POSITIONAL MODEL SYSTEMS AND COMPONENTS
    //
    // These are systems commonly attached to a PositionalModel object.
    //************************************************************************

    // Pointer to the data recording system.
    DataRecordingSystem* data_recording_system;

    //************************************************************************
    // ENVIRONMENT
    //************************************************************************

    Universe* universe;  // Reference to the parent universe.
    World*    world;     // Reference to the "reference" world for the model.

    // Magnetic variation at the PositionalModel's current position.
    AngularValue magnetic_variation;

    //************************************************************************
    // SIMULATION CONTROL DATA
    //************************************************************************

    // The actual type of positional model object.
    string simulation_object_type;

    // The display type of the positional model object.  Defaults to the
    // simulation object type.
    string simulation_display_type;

    // Each positional model object is given a unique numeric identifier.  This
```

```
// allows the simulation control layer and the GUI to differentiate and
// uniquely identify the positional model objects.
int  id_number;

// The number of the real-time CPU which created the positional model object.
int  cpu_number;

//*************************************************************************
// INITIAL CONDITION MEMBERS
//*************************************************************************

// The current initial condition set selection.
unsigned int case_number;

// This variable flags the need to re-select the initial condition set.
bool full_initialization_needed;

// This variable flags the need to re-initialize the vehicle.
bool modified_ics;

// This is the name of the initialization file.
string initialization_file_name;

//*************************************************************************
// HEALTH DATA
//
// This data defines the "health" of the positional model object. Simulations
// that track damage to a positional model use this data (e.g. combat
// simulations).
//*************************************************************************

// Flag which activates/deactivates object.
bool alive;

// Flag that indicates to reset the alive flag to true when in RESET
bool auto_reset_alive;

// Number of damage points remaining that are required to kill the
// positional model object.
double hit_points;

// Initial number of hit points for this positional model
double initial_hit_points;

// Size of the box around the target, in feet,  used to determine if a
// bullet hit has occurred
Vector<double> strike_zone_dimensions;

//*************************************************************************
// POSITIONAL MODEL DYNAMICS
//*************************************************************************

bool trim_has_converged;   // Flag indicating trim has converged

// Flag indicates wether a state has changed that would require the
// re-computation of the derived states.
bool derived_states_consistent;
```

```
//***************************************************************************
// INITIAL CONDITIONS
//
// Governing Initial Conditions.  The PositionalModel attempts to maintain
// these initial condtions.
//***************************************************************************

// Initial distance from primary refrence point in feet.  The distance is
// measured in the topodetic coordinate system of the primary reference
// point.
Vector<double> init_distance_from_ref_point;

// The initial geodetic coordinates.
GeodeticCoordinates init_geodetic_coordinates;

// The initial euler angles.
EulerAngles init_euler_angles;

// The initial world relative velocity in body coordinates in ft/sec.
Vector<double> world_rel_vel_body_init;

// The initial world relative angular velocity in body coordinates ft/sec.
Vector<AngularValue> local_vertical_angular_vel_body_init;

// -------------------------------------------------------------------------
// Derived Initial Conditions.
// These are not true initial conditions.  Instead these initial conditions
// modify the governing initial conditions to increase the likelihood that
// a derived state will begin with the given initial value.
// -------------------------------------------------------------------------

// The initial velocity track as an angle.
Angle track_init;

// With a round world, if a primary reference point is set, the
// init_distance_from_ref_point and the init_geodetic_coordinates are kept
// consistent even if one or the other is changed.  If the primary reference
// point is changed, this bool indicates which version of the initial
// position to maintain.
bool maintain_init_position_as_geodetic_coordinates;

//***************************************************************************
// POSITIONAL MODEL STATES
//***************************************************************************

// ROTATION MATRICES AND QUATERNIONS
// These rotation matrices transform vectors between body, inertial,
// topodetic, and world axes.
RotationMatrix body_to_inertial;
RotationMatrix body_to_topodetic;
RotationMatrix body_to_world;
RotationMatrix inertial_to_body;
RotationMatrix inertial_to_topodetic;
RotationMatrix topodetic_to_body;
RotationMatrix topodetic_to_inertial;
RotationMatrix topodetic_to_world;
```

```
RotationMatrix world_to_body;
RotationMatrix world_to_topodetic;

// Orientation of the body axis relative to the topodetic axis.
UnitQuaternion topodetic_rel_orientation;

// Orientation of the topodetic axis relative to the world axis.
UnitQuaternion world_rel_topodetic_orientation;

// Orientation of the body axis relative to the world axis.
UnitQuaternion world_rel_orientation;

// INERTIAL AXIS
// The inertial states (position, orientation, velocity, and angular
// velocity) are integrands.  In self propagating models, these are the
// most basic states and they are integrated to propagate the model through
// time.
IntegrandTemplate< Vector<double> >*       position;          // ft.
IntegrandTemplate< UnitQuaternion >*       orientation;       // quaternion
IntegrandTemplate< Vector<double> >*       velocity;          // ft/s.
IntegrandTemplate< Vector<AngularValue> >* angular_vel_body;  // angle/s.

// WORLD AXIS
// The world relative velocity in world coordinates in ft/sec.
Vector<double> world_rel_vel;

// The magnitude of the world relative velocity; i.e., the world relative
// speed in ft/sec.
double world_rel_vel_mag;

// The rate of change of the world relative speed (not velocity).
double world_rel_vel_rate;

// The distance of the PositionalModel CG from the center of the reference
// world in feet.
double geocentric_altitude;

// BODY AXIS
// The Euler angles representing rotation from the body coordinate system to
// the local vertical coordinate system.
EulerAngles euler_angles;

// The rate of change of the euler angles.  The literature refers to this
// collection of angular rates as [phi_dot, theta_dot, psi_dot] in angle/sec.
Vector<AngularValue> euler_angle_rates;

// The world relative velocity expressed in body coordinates.  The literature
// commonly refers to the vector as [u, v, w] in ft/sec.
Vector<double> world_rel_vel_body;

// The rate of change of the world relative velocity from the perspective of
// the body coordinate system.  This vector accounts for both the world
// relative accelerations and apparent changes in the body-axis components of
// the world relative velocity due to the rotation of the body coordinate
// axis.  The literature commonly refers to this vector as [udot, vdot,
// wdot] in ft/sec^2.
Vector<double> body_deriv_world_rel_vel_body;
```

```
// The world relative angular velocity expressed in body coordinates. The
// literature refers to this vector as [p, q, r].
Vector<AngularValue> world_rel_angular_vel_body;

// The angular velocity of the positional model relative to the local
// vertical frame and expressed in body coordinates.
Vector<AngularValue> local_vertical_rel_angular_vel_body;

// TOPODETIC AXIS
// The position of the PositionalModel relative to the world.
// GeodeticCoordinates actually contains both topodetic and world axis
// positions.
GeodeticCoordinates geodetic_coordinates;

// The rate of change of laittude in angle/sec.
AngularValue latitude_dot;

// The rate of change of longitude in angle/sec.
AngularValue longitude_dot;

// The world relative velocity expressed in topodetic coordinates in ft/sec.
Vector<double> world_rel_vel_topodetic;

// The world relative acceleration expressed in topodetic coordinates in
// ft/sec^2.
Vector<double> world_rel_accel_topodetic;

// The world relative angular velocity of the local vertical frame expressed
// in topodetic coordinates in angle/sec.
Vector<AngularValue> world_rel_local_vert_angular_vel_topo;

// The ground speed track angle; i.e., the tangent of East velocity over
// North velocity.
Angle  track;

double ground_speed; // The ground speed in ft/sec.

//*************************************************************************
// REFERENCE POINT
//*************************************************************************

// List of relative geometry information between the PositionalModel and
// geographic reference points.
GeoRefRelInfoList* geo_ref_rel_info_list;

// Handle to the relative geometry information between the PositionalModel
// and the primary geogrpahic reference point.
GeoRefRelInfoHandle* primary_geo_ref_rel_info_handle;

/** This function is available so that the GeoRefRelInfoHandle can register
    itself onto the PositionalModel's GeoRefRelInfoList.
    @param handle The "handle" to the relative geometry between the primary
    geographic reference point and the positional model.
    @param geo_ref_point The pointer to the primary geographic reference
    point.
*/
```

```
const GeoRefRelInfo*
registerWithGeoRefRelInfoList(const GeoRefRelInfoHandle& handle,
                              const GeoRefPoint*         geo_ref_point);

/** This function is available so that the GeoRefRelInfoHandle can unregister
    itself from the PositionalModel's GeoRefRelInfoList.
    @param handle The "handle" to the relative geometry between the primary
    geographic reference point and the positional model.
    @param geo_ref_rel_info Structure containing the positional model's
    geographical reference information.
*/
void unregisterFromGeoRefRelInfoList(const GeoRefRelInfoHandle& handle,
                                     const GeoRefRelInfo* geo_ref_rel_info);

// The above two functions require friendship for access.
friend class GeoRefRelInfoHandle;

//************************************************************************
// HEIGHT ABOVE TERRAIN
//************************************************************************

// If true, the relative altitude with the primary reference point is used
// for height above terrain.
bool use_sz_for_hat;

// The current height above terrain in feet.
double height_above_terrain;

// Vector normal to the ground
Vector<double> terrain_polygon_normal;

// Timer used to determine if the height above terrain computation is valid.
Timer* height_above_terrain_time_tag;

//************************************************************************
// RELATIVE GEOMETRY SETTINGS
//************************************************************************

// If true, compute relative geomtry between the CGs of the two
// PositionalModel objects.
bool compute_cg_to_cg_rel_geom_flag;

// If true, compute the relative geometry between the pilot location of the
// PositionalModel object and the CG of the target.
bool compute_pilot_to_target_cg_rel_geom_flag;

//************************************************************************
// PILOT
//************************************************************************

// The reference to the pilot object.  A PositionalModel object may or may
// not have a pilot object
Pilot* pilot;

//************************************************************************
// PLAYBACK
//************************************************************************
```

```
  // The Playback class encapsulates the recording of and playing back the
  // movement of objects derived from PositionalModel.
  PositionalModelPlayback* playback;

  // For playback models, we may optionally make a call to calcDerivedStates()
  // on each call to propagateState() if this is true.
  bool calc_derived_states;

  //************************************************************************
  // ASCII DISPLAY SYSTEM
  //************************************************************************

  // The AsciiDisplaySystem encapsulates the printing of model states,
  // the print frequency, etc...
  AsciiDisplaySystem* ascii_display_system;


  /** Copy construction of PositionalModel objects is not allowed.
      Private version of this constructor creates error if attempted outside
      this class.
      @param object A reference to the PositionalModel attempting to be copied.
  */
  PositionalModel(const PositionalModel&);

  /** Assignment of PositionalModel objects is not allowed.
      @param object A reference to the PositionalModel attempting to be
      assigned.
      @return A PositionalModel object after assignment.
  */
  PositionalModel& operator=(const PositionalModel&);
};

/************************************************************************
 * Inline member functions                                              *
 ************************************************************************/

inline void PositionalModel::putDataRecordingSystem(
  DataRecordingSystem* new_data_recording_system)
{
  data_recording_system = new_data_recording_system;
}

inline DataRecordingSystem* PositionalModel::getDataRecordingSystem()
{
  return data_recording_system;
}

inline
const DataRecordingSystem* PositionalModel::getDataRecordingSystem() const
{
  return data_recording_system;
}

inline const Vector<double>& PositionalModel::getPosition() const
{
  return position->getValue();
```

```
}

inline const Vector<double>& PositionalModel::getVelocity() const
{
  return velocity->getValue();
}

inline const UnitQuaternion& PositionalModel::getOrientation() const
{
  return orientation->getValue();
}

inline const Vector<AngularValue>& PositionalModel::getAngularVelBody() const
{
  return angular_vel_body->getValue();
}

inline const EulerAngles& PositionalModel::getEulerAngles() const
{
  return euler_angles;
}

inline const RotationMatrix& PositionalModel::getInertialToBody() const
{
  return inertial_to_body;
}

inline const RotationMatrix& PositionalModel::getBodyToInertial() const
{
  return body_to_inertial;
}

inline const RotationMatrix& PositionalModel::getInertialToTopodetic() const
{
  return inertial_to_topodetic;
}

inline const RotationMatrix& PositionalModel::getTopodeticToInertial() const
{
  return topodetic_to_inertial;
}

inline bool PositionalModel::isInitPositionInGeodeticCoordinates() const
{
  return maintain_init_position_as_geodetic_coordinates;
}

inline void PositionalModel::putPosition(
  IntegrandTemplate< Vector<double> >* new_integrand)
{
  position = new_integrand;
}

inline void PositionalModel::putVelocity(
  IntegrandTemplate< Vector<double> >* new_integrand)
{
  velocity = new_integrand;
```

```
}

inline void PositionalModel::putOrientation(
  IntegrandTemplate< UnitQuaternion >* new_integrand)
{
  orientation = new_integrand;
}

inline void PositionalModel::putAngularVelBody(
  IntegrandTemplate< Vector<AngularValue> >* new_integrand)
{
  angular_vel_body = new_integrand;
}

inline void PositionalModel::putPosition(const Vector<double>& new_position)
{
  position->putValue(new_position);
}

inline void PositionalModel::putVelocity(const Vector<double>& new_velocity)
{
  velocity->putValue(new_velocity);
}

inline
void PositionalModel::putOrientation(const UnitQuaternion& new_orientation)
{
  orientation->putValue(new_orientation);
}

inline void PositionalModel::putAngularVelBody(
  const Vector<AngularValue>& new_angular_vel_body)
{
  angular_vel_body->putValue(new_angular_vel_body);
}

inline const string& PositionalModel::getPositionalModelType() const
{
  // This method returns a string that identifies the actual class type
  // of the positional model object.
  return simulation_object_type;
}

inline const string& PositionalModel::getPositionalModelDisplayType() const
{
  // This method returns a string that identifies the class type
  // to display for this positional model object
  return simulation_display_type;
}

inline
void PositionalModel::putPositionalModelDisplayType(const string& display_type)
{
  // This method allows the string that identifies the class type
  // to display for the positional model object to be changed.  This is
  // useful for running the dynamics of one aircraft while the displays
  // represent it as a different aircraft
```

```
  simulation_display_type = display_type;
}

inline int PositionalModel::getID() const
{
  return id_number;
}

inline  int PositionalModel::getCpuNumber() const
{
  return cpu_number;
}

inline unsigned int PositionalModel::getCaseNumber() const
{
  return case_number;
}

inline const string& PositionalModel::getInitializationFileName() const
{
  return initialization_file_name;
}

inline bool PositionalModel::isFullInitializationNeeded() const
{
  return full_initialization_needed;
}

inline bool PositionalModel::getAlive() const
{
  return alive;
}

inline double PositionalModel::getHitPoints() const
{
  return hit_points;
}

inline double PositionalModel::getInitialHitPoints() const
{
  return initial_hit_points;
}

inline const Vector<double>& PositionalModel::getStrikeZoneDimensions() const
{
  return strike_zone_dimensions;
}

inline void PositionalModel::putStrikeZoneDimensions(
  const Vector<double>& new_dimensions)
{
  strike_zone_dimensions = new_dimensions;
}

inline void PositionalModel::putModifiedICs(const bool new_value)
{
  modified_ics = new_value;
```

```
}

inline bool PositionalModel::getModifiedICs() const
{
  return modified_ics;
}

inline void PositionalModel::putCaseNumber(unsigned int new_value)
{
  case_number                = new_value;
  full_initialization_needed = true;
}

inline void PositionalModel::putInitializationFileName(const string& new_name)
{
  initialization_file_name   = new_name;
  full_initialization_needed = true;
}

inline void PositionalModel::setFullInitializationNeeded(bool new_value)
{
  full_initialization_needed = new_value;
}

inline void PositionalModel::putAlive(const bool new_alive)
{
  alive = new_alive;
}

inline bool PositionalModel::getTrimHasConverged() const
{
  return trim_has_converged;
}

inline
void PositionalModel::putTrimHasConverged(const bool new_trim_has_converged)
{
  trim_has_converged = new_trim_has_converged;
}

inline void PositionalModel::putHitPoints(const double new_hit_points)
{
  hit_points = new_hit_points;
}

inline bool PositionalModel::getDerivedStatesConsistent() const
{
  return derived_states_consistent;
}

inline void PositionalModel::putDerivedStatesConsistent(bool new_value)
{
  derived_states_consistent = new_value;
}

inline bool PositionalModel::computeCGToCGRelGeomFlag() const
{
```

```
  return compute_cg_to_cg_rel_geom_flag;
}

inline bool PositionalModel::computePilotToTargetCGRelGeomFlag() const
{
  return compute_pilot_to_target_cg_rel_geom_flag;
}

inline const Pilot* PositionalModel::getPilot() const
{
  return pilot;
}

inline Pilot* PositionalModel::getPilot()
{
  return pilot;
}

inline void
PositionalModel::putStates(const Vector<double>&       new_position,
                           const Vector<double>&       new_velocity,
                           const UnitQuaternion&       new_orientation,
                           const Vector<AngularValue>& new_angular_vel_body)
{
  putPosition(new_position);
  putVelocity(new_velocity);
  putOrientation(new_orientation);
  putAngularVelBody(new_angular_vel_body);

  derived_states_consistent = false;
}

inline
const Vector<AngularValue>& PositionalModel::getWorldRelAngularVelBody() const
{
  return world_rel_angular_vel_body;
}

inline const Vector<AngularValue>&
PositionalModel::getLocalVerticalRelAngularVelBody() const
{
 return local_vertical_rel_angular_vel_body;
}

inline const Vector<AngularValue>&
PositionalModel::getWorldRelLocalVertAngularVelTopo() const
{
  return world_rel_local_vert_angular_vel_topo;
}

inline const Vector<double>& PositionalModel::getWorldRelVelBody() const
{
  return world_rel_vel_body;
}

inline const Vector<double>& PositionalModel::getWorldRelVel() const
{
```

```
  return world_rel_vel;
}

inline double PositionalModel::getWorldRelVelMag() const
{
  return world_rel_vel_mag;
}

inline double PositionalModel::getWorldRelVelRate() const
{
  return world_rel_vel_rate;
}

inline void PositionalModel::putWorldRelVelRate(double new_value)
{
  world_rel_vel_rate = new_value;
}

inline const Vector<double>& PositionalModel::getWorldRelVelTopodetic() const
{
  return world_rel_vel_topodetic;
}

inline AngularValue PositionalModel::getMagneticVariation() const
{
  return magnetic_variation;
}

inline AsciiDisplaySystem* PositionalModel::getAsciiDisplaySystem()
{
  return ascii_display_system;
}

inline
const Vector<double>& PositionalModel::getBodyDerivWorldRelVelBody() const
{
  return body_deriv_world_rel_vel_body;
}

inline void
PositionalModel::putBodyDerivWorldRelVelBody(const Vector<double>& new_value)
{
  body_deriv_world_rel_vel_body = new_value;
}

inline const RotationMatrix& PositionalModel::getWorldToBody() const
{
  return world_to_body;
}

inline const RotationMatrix& PositionalModel::getBodyToWorld() const
{
  return body_to_world;
}

inline const RotationMatrix& PositionalModel::getTopodeticToBody() const
{
```

```
  return topodetic_to_body;
}

inline const RotationMatrix& PositionalModel::getBodyToTopodetic() const
{
  return body_to_topodetic;
}

inline const RotationMatrix& PositionalModel::getTopodeticToWorld() const
{
  return topodetic_to_world;
}

inline const RotationMatrix& PositionalModel::getWorldToTopodetic() const
{
  return world_to_topodetic;
}

inline World* PositionalModel::getWorld()
{
  return world;
}

inline const World* PositionalModel::getWorld() const
{
  return world;
}

inline
const GeodeticCoordinates& PositionalModel::getGeodeticCoordinates() const
{
  return geodetic_coordinates;
}

inline GeoRefRelInfoList* PositionalModel::getGeoRefRelInfoList()
{
  return geo_ref_rel_info_list;
}

inline
void PositionalModel::putAsciiDisplaySystem(AsciiDisplaySystem* new_system)
{
  ascii_display_system = new_system;
}

inline const AngularValue& PositionalModel::getLatitudedot() const
{
  return latitude_dot;
}

inline const AngularValue& PositionalModel::getLongitudedot() const
{
  return longitude_dot;
}

inline const Universe* PositionalModel::getUniverse() const
{
```

```
  return universe;
}

inline Universe* PositionalModel::getUniverse()
{
  return universe;
}

inline const
GeoRefRelInfoHandle* PositionalModel::getPrimaryGeoRefRelInfoHandle() const
{
  return primary_geo_ref_rel_info_handle;
}

inline
const UnitQuaternion& PositionalModel::getTopodeticRelOrientation() const
{
  return topodetic_rel_orientation;
}

inline const Vector<AngularValue>& PositionalModel::getEulerAngleRates() const
{
  return euler_angle_rates;
}

inline const AngularValue& PositionalModel::getPhidot() const
{
  return euler_angle_rates[0];
}

inline const AngularValue& PositionalModel::getThetadot() const
{
  return euler_angle_rates[1];
}

inline const AngularValue& PositionalModel::getPsidot() const
{
  return euler_angle_rates[2];
}

inline double PositionalModel::getGeocentricAltitude() const
{
  return geocentric_altitude;
}

inline const Vector<double>& PositionalModel::getWorldRelAccelTopodetic() const
{
  return world_rel_accel_topodetic;
}

inline void PositionalModel::putWorldRelAccelTopodetic(
  const Vector<double>& new_value)
{
  world_rel_accel_topodetic = new_value;
}

inline PositionalModelPlayback* PositionalModel::getPositionalModelPlayback()
```

```
{
  return playback;
}

inline double PositionalModel::getHeightAboveTerrain() const
{
  if( use_sz_for_hat )
  {
    return -getPrimaryRefPointRelativePosition()[2];
  }
  else
  {
    return height_above_terrain;
  }
}

inline bool PositionalModel::heightAboveTerrainValid() const
{
  return !use_sz_for_hat;
}

inline Vector<double> PositionalModel::getTerrainPolygonNormal() const
{
  if (use_sz_for_hat)
  {
    return -k_vector;
  }
  else
  {
    return terrain_polygon_normal;
  }
}

inline const Timer* PositionalModel::getHeightAboveTerrainTimeTag() const
{
  return height_above_terrain_time_tag;
}

inline double PositionalModel::getAltitude() const
{
  return getGeodeticCoordinates().getAltitude();
}

inline const EulerAngles& PositionalModel::getInitEulerAngles() const
{
  return init_euler_angles;
}

inline const Vector<double>& PositionalModel::getWorldRelVelBodyInit() const
{
  return world_rel_vel_body_init;
}

inline const Vector<AngularValue>&
PositionalModel::getLocalVerticalAngularVelBodyInit() const
{
  return local_vertical_angular_vel_body_init;
```

```
}

inline const Angle& PositionalModel::getTrack() const
{
  return track;
}

inline double PositionalModel::getGroundSpeed() const
{
  return ground_speed;
}

inline double PositionalModel::getAltitudeDot() const
{
  return   -world_rel_vel_topodetic[2];
}

inline void PositionalModel::integratePosition()
{
  assume(position);
  position->integrate();
}

inline void PositionalModel::integrateVelocity()
{
  assume(velocity);
  velocity->integrate();
}

inline void PositionalModel::integrateOrientation()
{
  assume(orientation);
  orientation->integrate();
}

inline void PositionalModel::integrateAngularVelBody()
{
  assume(angular_vel_body);
  angular_vel_body->integrate();
}

#endif
```

A.2 Base Class Member Function Definition File

The following is the member function definition file for the base class PositionalModel:

```
/*****************************************************************************
 *                                                                           *
 *      _/                    _/_/_/_/  _/_/_/_/  _/_/_/_/                   *
 *     _/                    _/        _/    _/  _/            _/      _/     *
 *    _/         _/_/      _/_/_/_/  _/_/_/_/  _/_/_/_/   _/_/_/  _/_/_/     *
 *   _/        _/   _/          _/  _/  _/           _/     _/      _/        *
 *  _/_/_/_/   _/_/_/_/  _/_/_/_/  _/    _/  _/_/_/_/                         *
 *                                                                           *
 *  File:        PositionalModel.cpp                                         *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *  NOTICE:                                                                  *
 *                                                                           *
 *    THIS SOFTWARE MAY BE USED, COPIED AND PROVIDED TO OTHERS ONLY AS       *
 *    PERMITTED UNDER THE TERMS OF THE CONTRACT OR OTHER AGREEMENT UNDER     *
 *    WHICH IT WAS ACQUIRED FROM THE U.S. GOVERNMENT.  NEITHER TITLE TO      *
 *    NOR OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.  THIS NOTICE      *
 *    SHALL REMAIN ON ALL COPIES OF THE SOFTWARE.                            *
 *                                                                           *
 *****************************************************************************/

#include "PositionalModel.hpp"

#include "AB2Integrand.hpp"
#include "Angle.hpp"
#include "AsciiDisplaySystem.hpp"
#include "Constants.hpp"
#include "DataRecordingSystem.hpp"
#include "EulerAngles.hpp"
#include "GeoRefPoint.hpp"
#include "GeoRefRelInfoHandle.hpp"
#include "GeoRefRelInfoList.hpp"
#include "GeodeticCoordinates.hpp"
#include "InitializationFileProcessor.hpp"
#include "Pilot.hpp"
#include "PositionalModelPlayback.hpp"
#include "RequestStatus.hpp"
#include "TerminalIO.hpp"
#include "Timer.hpp"
#include "TrafficPlayback.hpp"
#include "UnitQuatIntegrand.hpp"
#include "UnitQuaternion.hpp"
#include "Universe.hpp"
#include "Vector.hpp"
#include "World.hpp"
#include "assume.hpp"
#include "miscMath.hpp"

//=============================================================
// This positional model constructor is called by derived classes.
//=============================================================
PositionalModel::PositionalModel(Universe*    universe,
```

```
                                const string model_type,
                                const string model_name,
                                int          object_cpu_number,
                                bool         is_alive)
: SimulationModel(universe->getMode(),
                  universe->getTimer(),
                  model_name.c_str()),
data_recording_system(0),
universe(universe),
world(universe->getWorld()),
simulation_object_type(model_type),
simulation_display_type(model_type),
id_number(0),
cpu_number(object_cpu_number),
case_number(0),
full_initialization_needed(true),
initialization_file_name(""),
alive(is_alive),
auto_reset_alive(alive),
initial_hit_points(20),
strike_zone_dimensions(Vector<double>(50.0, 8.0, 5.0)),
trim_has_converged(true),
derived_states_consistent(false),
init_distance_from_ref_point(zero_vector),
init_geodetic_coordinates(universe->getWorld()->getWorldShapeData()),
world_rel_vel_body_init(zero_vector),
local_vertical_angular_vel_body_init(zero_vector),
track_init(0.0),
maintain_init_position_as_geodetic_coordinates(true),
position(0),
orientation(0),
velocity(0),
angular_vel_body(0),
world_rel_vel(zero_vector),
world_rel_vel_mag(0.0),
world_rel_vel_rate(0.0),
geocentric_altitude(0.0),
euler_angle_rates(zero_vector),
world_rel_vel_body(zero_vector),
body_deriv_world_rel_vel_body(zero_vector),
world_rel_angular_vel_body(zero_vector),
local_vertical_rel_angular_vel_body(zero_vector),
geodetic_coordinates(universe->getWorld()->getWorldShapeData()),
latitude_dot(0.0),
longitude_dot(0.0),
world_rel_vel_topodetic(zero_vector),
world_rel_accel_topodetic(zero_vector),
world_rel_local_vert_angular_vel_topo(zero_vector),
track(0.0),
ground_speed(0.0),
use_sz_for_hat(true),
height_above_terrain(-10000.0),
terrain_polygon_normal(zero_vector),
compute_cg_to_cg_rel_geom_flag(false),
compute_pilot_to_target_cg_rel_geom_flag(false),
pilot(0),
playback(0),
```

```
  calc_derived_states(true),
  ascii_display_system(0)
{
  // Add the positional model to the list in Universe and retrieve its unique
  // identifier in the list.
  id_number = universe->addPositionalModel(this);

  if ( world->getWorldShapeData()->worldIsFlat() )
  {
    // By default, initialize position using the geodetic coordinates if the
    // world is flat.
    maintain_init_position_as_geodetic_coordinates = true;
  }
  else
  {
    // If the world is spherical or ellipsoidal, use the initial position
    // from the primary reference point to initialize position.
    maintain_init_position_as_geodetic_coordinates = false;
  }

  // Create an empty list of relative geometry information for geographic
  // refence points of interest.
  geo_ref_rel_info_list = new GeoRefRelInfoList(*this);
  assume(geo_ref_rel_info_list);

  // Create a handle to the relative geometry information for the primary
  // reference point.  By default, no geograhic reference point is assigned to
  // it.
  primary_geo_ref_rel_info_handle = new GeoRefRelInfoHandle(*this,0);
  assume(primary_geo_ref_rel_info_handle);

  height_above_terrain_time_tag = new Timer;
  assume(height_above_terrain_time_tag);

  PositionalModel::initialize();

  // Create the playback object, linking it to the inertial basic states.
  playback = new PositionalModelPlayback( getTimer(),
                                          "/dev/null",
                                          0.0,
                                          false,
                                          &position,
                                          &orientation,
                                          &velocity,
                                          &angular_vel_body,
                                          inertial_to_body,
                                          geodetic_coordinates );
}

//=================================================================
// This positional model constructor creates static, playback, and traffic
// models.
//=================================================================
PositionalModel::PositionalModel(Universe*    universe,
                                 const string model_type,
                                 const string model_name,
                                 GeoRefPoint* new_geo_ref_point,
```

```
                                double       latitude,
                                double       longitude,
                                double       altitude,
                                double       pitch_attitude,
                                double       bank_attitude,
                                double       heading_attitude,
                                double       model_cg_bias,
                                int          cpu_number,
                                bool         playback_auto_repeat,
                                bool         is_traffic,
                                bool         is_incursion,
                                bool         calc_derived_states,
                                double       delay_before_rewind,
                                int          disappear_loop,
                                bool         is_alive,
                                const char*  playback_filename)
: SimulationModel(universe->getMode(),
                  universe->getTimer(),
                  model_name.c_str()),
data_recording_system(0),
universe(universe),
world(universe->getWorld()),
simulation_object_type(model_type),
simulation_display_type(model_type),
id_number(0),
cpu_number(cpu_number),
initialization_file_name(""),
alive(is_alive),
auto_reset_alive(is_alive),
initial_hit_points(20),
strike_zone_dimensions(Vector<double>(50.0, 8.0, 5.0)),
trim_has_converged(true),
derived_states_consistent(false),
init_distance_from_ref_point(zero_vector),
init_geodetic_coordinates(universe->getWorld()->getWorldShapeData()),
world_rel_vel_body_init(zero_vector),
local_vertical_angular_vel_body_init(zero_vector),
track_init(0.0),
maintain_init_position_as_geodetic_coordinates(true),
position(0),
orientation(0),
velocity(0),
angular_vel_body(0),
world_rel_vel(zero_vector),
world_rel_vel_mag(0.0),
world_rel_vel_rate(0.0),
geocentric_altitude(0.0),
euler_angle_rates(zero_vector),
world_rel_vel_body(zero_vector),
body_deriv_world_rel_vel_body(zero_vector),
world_rel_angular_vel_body(zero_vector),
local_vertical_rel_angular_vel_body(zero_vector),
geodetic_coordinates(universe->getWorld()->getWorldShapeData(),
                     latitude * deg_to_rad,
                     longitude * deg_to_rad,
                     altitude),
latitude_dot(0.0),
```

```
  longitude_dot(0.0),
  world_rel_vel_topodetic(zero_vector),
  world_rel_accel_topodetic(zero_vector),
  world_rel_local_vert_angular_vel_topo(zero_vector),
  track(0.0),
  ground_speed(0.0),
  use_sz_for_hat(true),
  height_above_terrain(-10000.0),
  terrain_polygon_normal(zero_vector),
  compute_cg_to_cg_rel_geom_flag(false),
  compute_pilot_to_target_cg_rel_geom_flag(false),
  pilot(0),
  playback(0),
  calc_derived_states(calc_derived_states),
  ascii_display_system(0)
{
  // Add the positional model to the list in Universe and retrieve its unique
  // identifier in the list.
  id_number = universe->addPositionalModel(this);

  if ( world->getWorldShapeData()->worldIsFlat() )
  {
    // By default, initialize position using the geodetic coordinates if the
    // world is flat.
    maintain_init_position_as_geodetic_coordinates = true;
  }
  else
  {
    // If the world is spherical or ellipsoidal, use the initial position
    // from the primary reference point to initialize position.
    maintain_init_position_as_geodetic_coordinates = false;
  }

  // Create an empty list of relative geometry information for geographic
  // reference points of interest.
  geo_ref_rel_info_list = new GeoRefRelInfoList(*this);
  assume(geo_ref_rel_info_list);

  // Create a handle to the relative geometry information for the primary
  // reference point.  By default, no geographic reference point is assigned to
  // it.
  primary_geo_ref_rel_info_handle = new GeoRefRelInfoHandle(*this,0);
  assume(primary_geo_ref_rel_info_handle);

  height_above_terrain_time_tag = new Timer;
  assume(height_above_terrain_time_tag);

  // Must create Integrands for storage of position, velocity, etc...
  velocity =
    new AB2Integrand< Vector<double> >(zero_vector, zero_vector,
                                       getMode(), getTimer());
  assume(velocity);

  position =
    new AB2Integrand<Vector<double> >(geodetic_coordinates.getWorldRelVector(),
                                      getVelocity(), getMode(), getTimer());
  assume(position);
```

```
angular_vel_body = new AB2Integrand< Vector<AngularValue> >(
  zero_vector, zero_vector, getMode(), getTimer());
assume(angular_vel_body);

euler_angles = EulerAngles(pitch_attitude * deg_to_rad,
                           bank_attitude * deg_to_rad,
                           heading_attitude * deg_to_rad);

orientation = new UnitQuatIntegrand( UnitQuaternion(euler_angles),
                                     getAngularVelBody(),
                                     zero_vector,
                                     getMode(),
                                     getTimer());
assume(orientation);

inertial_to_body = RotationMatrix( orientation->getValue() );

// Set all matrices to zero
PositionalModel::initialize();

// If the user has selected a geo ref point then use it
if ( new_geo_ref_point )
{
  putPrimaryGeoRefPoint(new_geo_ref_point);
}

// Update all derived states
PositionalModel::calcDerivedStates();

// Set initial EulerAngles to those passed into the constructor
putTheta(pitch_attitude * deg_to_rad);
putPhi(bank_attitude * deg_to_rad);
putPsi(heading_attitude * deg_to_rad);

// If traffic selected then create a Playback Traffic object, otherwise
// create a PositionalModelPlayback object which can be a static model
// or a playback model. Traffic automatically repeats itself and performs
// anti - collision computations.
if ( is_traffic )
{
  playback = new TrafficPlayback( universe,
                                  getTimer(),
                                  id_number,
                                  playback_filename,
                                  model_cg_bias,
                                  &position,
                                  &orientation,
                                  &velocity,
                                  &angular_vel_body,
                                  inertial_to_body,
                                  geodetic_coordinates,
                                  new_geo_ref_point,
                                  &euler_angles,
                                  delay_before_rewind,
                                  disappear_loop,
                                  is_incursion);
```

```
    playback->initialize();

    // Need to allow relative geometry computations
    compute_cg_to_cg_rel_geom_flag = true;
  }
  else
  {
    playback = new PositionalModelPlayback( getTimer(),
                                            playback_filename,
                                            model_cg_bias,
                                            playback_auto_repeat,
                                            &position,
                                            &orientation,
                                            &velocity,
                                            &angular_vel_body,
                                            inertial_to_body,
                                            geodetic_coordinates);

    playback->initialize();

    // Need to allow relative geometry computations
    compute_cg_to_cg_rel_geom_flag = true;
  }
}

//=====================================================================
// The PositionalModel destructor deallocates dynamically allocated members.
//=====================================================================
PositionalModel::~PositionalModel()
{
  delete orientation;
  delete angular_vel_body;
  delete position;
  delete velocity;

  orientation      = 0;
  angular_vel_body = 0;
  position         = 0;
  velocity         = 0;

  delete pilot;
  pilot = 0;

  delete height_above_terrain_time_tag;
  height_above_terrain_time_tag = 0;

  delete primary_geo_ref_rel_info_handle;
  primary_geo_ref_rel_info_handle = 0;

  delete geo_ref_rel_info_list;
  geo_ref_rel_info_list = 0;

  delete playback;
  playback = 0;

  delete ascii_display_system;
```

```
  ascii_display_system = 0;
}

//=================================================================
// This method calculates derived states from the four basic inertial states:
// position, velocity, orientation, and angular velocity in body coordinates.
// The derived states can be grouped into the following categories:
//    - Rotation matrices and orientation quaternions.  The rotation
//      matrices are used to transform other states into various
//      coordinate systems.
//    - World relative states. Position, velocity, orientation, and
//      angular velocity are computed relative to the reference world.  The
//      world relative states are transformed into various coordinate systems.
//    - Magnetic variation is computed at the location of the positional model
//      (or from the navigation data base if flat earth).
//    - Relative geometry between the positional model and geographic reference
//      points of interest.
//    - Pilot derived states.
//    - Height above terrain.
//=================================================================
void PositionalModel::calcDerivedStates()
{
  inertial_to_body = RotationMatrix(getOrientation());
  body_to_inertial = ~inertial_to_body;

  // Compute the world-relative position of the positional model in world
  // coordinates and set the geodetic coordinates using the result.
  geodetic_coordinates.putWorldRelVector(
    world->getInertialToWorld() * ( getPosition() - world->getPosition() ) );

  // The distance between the center of the world and the positional model is
  // the magnitude of the world relative position.
  geocentric_altitude = geodetic_coordinates.getWorldRelVector().mag();

  // Based on the positional model's location relative to the world, calculate
  // the rotation matrix between the topodetic coordinate system (tangent to
  // the world surface) and the world coordinate system.
  world_to_topodetic = geodetic_coordinates.calcWorldToTopodetic();
  topodetic_to_world = ~world_to_topodetic;

  world_rel_topodetic_orientation = UnitQuaternion(world_to_topodetic);

  inertial_to_topodetic = world_to_topodetic * world->getInertialToWorld();
  topodetic_to_inertial = ~inertial_to_topodetic;

  world_rel_orientation =~world->getOrientation()* getOrientation();

  world_to_body = RotationMatrix(world_rel_orientation);
  body_to_world = ~world_to_body;

  topodetic_rel_orientation =
    ~world_rel_topodetic_orientation * world_rel_orientation;

  topodetic_to_body = RotationMatrix(topodetic_rel_orientation);
  body_to_topodetic = ~topodetic_to_body;

  euler_angles = EulerAngles(topodetic_to_body);
```

```
// The world relative velocity is a combination of the difference between the
// inertial velocities of the positional model and the world and the cross
// product of the world's rotation and the positional model's position
// relative to the world.
world_rel_vel =
  world->getInertialToWorld() * (getVelocity() - world->getVelocity()) -
  ( world->getAngularVelWorld() ^ geodetic_coordinates.getWorldRelVector() );

world_rel_vel_body = world_to_body * world_rel_vel;

world_rel_vel_mag = world_rel_vel.mag();

world_rel_vel_topodetic = world_to_topodetic * world_rel_vel;

// The world relative angular velocity is the difference between the
// inertial angular velocities of the positional model and the world.
world_rel_angular_vel_body =
  getAngularVelBody() - world_to_body * world->getAngularVelWorld();

// The rates of latitude and longitude are a function of the positional
// model's world relative position and velocity.
geodetic_coordinates.calcGeodeticCoordinateRates(latitude_dot,  // output
                                                 longitude_dot, // output
                                                 world_rel_vel_topodetic);

world_rel_local_vert_angular_vel_topo = Vector<AngularValue>(
  longitude_dot * geodetic_coordinates.getLatitude().getCosine(),
  -latitude_dot,
  -longitude_dot * geodetic_coordinates.getLatitude().getSine());

local_vertical_rel_angular_vel_body = world_rel_angular_vel_body -
  topodetic_to_body * world_rel_local_vert_angular_vel_topo;

euler_angle_rates = calcEulerAngleRates(euler_angles,
                                        local_vertical_rel_angular_vel_body);

// Compute ground speed and track.
double velocity_north = world_rel_vel_topodetic[0];
double velocity_east  = world_rel_vel_topodetic[1];

// The ground speed is magnitude of the north and east components of the
// world relative velocity.
ground_speed =
  sqrt(velocity_north * velocity_north + velocity_east * velocity_east);

if ( fabs(velocity_east) > 1.0e-8 || fabs(velocity_north) > 1.0e-8 )
{
  // A reliable calculation of the track angle requires that either the
  // north or east components of the world relative velocity be outside the
  // neighborhood of zero.
  track = atan2(velocity_east , velocity_north);
}
else
{
  // The vehicle is either not moving or is only traveling in the
  // z-direction.  Approximate the track using psi.
```

```
  track = euler_angles.getPsi();
}

// Along a line (agonic) that stretches from a place in Florida to a place
// in the Mid West, the magnetic variation is zero degrees.  Along this line,
// the magnetic and true poles are aligned.  As time passes and this line
// shifts, the magnetic variations need to be adjusted.

// Magnetic Variation is + or West if you are east of this line (the magnetic
// pole is west of the line from where you are to the north pole).  Magnetic
// variation is - or East if you are west of this line (the magnetic pole is
// east of the line from where you are to the north pole).  There is a saying
// associated with this: 'east is least (-); west is best (+)' but it must be
// remembered that the east/west does not refer to east/west coast.

// Following this convention, the magnetic heading can be found by summing
// the true heading and the magnetic variation.
if ( world->getWorldShapeData()->worldIsFlat() )
{
  // Since not using latitude/longitude, use the magnetic variation as
  // defined in the navigation data base for its reference point (if there
  // is one).
  if ( primary_geo_ref_rel_info_handle->getGeoRefPoint() )
  {
    magnetic_variation =
      primary_geo_ref_rel_info_handle->getGeoRefPoint()->
      getMagneticVariation();
  }
  else
  {
    magnetic_variation = 0.0;
  }
}
else
{
  // Magnetic variation is obtained from the world given the positional
  // model's current location.
  magnetic_variation = world->calcMagneticVariation(geodetic_coordinates);
}

// Relative geometry is computed for each geographic reference point of
// interest.
geo_ref_rel_info_list->updateRelPositionsVelocities();

// The derived states of the positional model have been updated.
derived_states_consistent = true;

if ( pilot )
{
  // If a pilot is attached to the positional model, compute its derived
  // states.
  pilot->calcAuxEqns();
}

if ( !use_sz_for_hat &&
     (((getTimer().getElapsedTime() -
        height_above_terrain_time_tag->getElapsedTime()) >
```

```
         5.0 * getTimer().getTimeStep()) ||
         getTimer().getElapsedTime() <
         height_above_terrain_time_tag->getElapsedTime()) )
  {
    use_sz_for_hat         = true;
    height_above_terrain   = -100000.0;
    terrain_polygon_normal = zero_vector;
  }
}

//=========================================================================
// Initialize sets the basic inertial states to their initial conditions and
// sets the velocity derivatives to zero.
//=========================================================================
void PositionalModel::initialize()
{
  // Set accelerations and velocity rates to zero.
  body_deriv_world_rel_vel_body = zero_vector;
  world_rel_vel_rate            = 0.0;
  world_rel_accel_topodetic     = zero_vector;

  // Set the position, orientation, translational velocity, and angular
  // velocity to their initial conditions.
  if ( position )
  {
    position->initialize();
  }

  if ( orientation )
  {
    orientation->initialize();
  }

  if ( velocity )
  {
    velocity->initialize();
  }

  if ( angular_vel_body )
  {
    angular_vel_body->initialize();
  }

  geo_ref_rel_info_list->updateRelAccelerations();

  // Reset the "hit_points" to the "initial_hit_points"
  hit_points = initial_hit_points;
}

//=========================================================================
// This method creates any systems that models may define
// Currently only creates an AsciiDisplaySystem
//=========================================================================
void PositionalModel::createPositionalModelSystems()
{
  // Create the basic ASCII display system
  ascii_display_system = new AsciiDisplaySystem( this, getTimer() );
```

```
}

//=========================================================================
// Reset the init values of position model to their default states.  If the
// model has been assigned a primary geographic reference point, the
// positional model is placed there.  Otherwise, the position model is placed
// on the ground at zero latitude and longitude.  It is oriented parallel to
// the surface of the world.  It has no translational or angular velocity.
//=========================================================================
void PositionalModel::resetInitialConditions()
{
  // The initial position is set either relative to the reference world or
  // relative to the geographic reference point.  The geographic reference
  // point is used as the starting location if it exists.
  if ( getPrimaryGeoRefPoint() )
  {
    // Put the vehicle at the primary geographic reference point.  This will
    // also modify the initial geodetic coordinate and initial position.
    putInitDistanceFromRefPoint(zero_vector);
  }
  else
  {
    // The initial geodetic coordinate places the aircraft at 0.0 latitude, 0.0
    // longitude and 0.0 altitude of the current reference world.
    GeodeticCoordinates zero_coordinates(world->getWorldShapeData());
    // This call will also set the initial distance from the primary reference
    // point and the initial inertial position.
    putInitGeodeticCoordinates(zero_coordinates);
  }

  // Set the default orientation along the local vertical frame.  The
  // euler angles are used to set the inertial orientation.  It is better to
  // define the initial euler angles after the initial geodetic coordinates
  // because the formula to convert the euler angles to inertial orientation
  // requires the geodetic coordinates.
  EulerAngles parallel_to_world(0.0,0.0,0.0);
  // This method will also change the inertial orientation.
  putInitEulerAngles(parallel_to_world);

  // Give the vehicle an initial world relative velocity of zero.
  putWorldRelVelBodyInit(zero_vector);

  // Give the vehicle an initial world relative angular velocity of zero.
  putLocalVerticalAngularVelBodyInit(zero_vector);
}

//=========================================================================
// This method applies states stored in the 'InitializationFileProcessor' to
// the internal states of the object.
//=========================================================================
void PositionalModel::processInitializationFile(
  InitializationFileProcessor& file_processor)
{
  SimulationModel::processInitializationFile(file_processor);

  Vector<double> relative_position;
```

```
  if ( file_processor
       .getInitializationValue("reference point relative position:",
                               relative_position) )
  {
    putInitDistanceFromRefPoint(relative_position);
  }

  EulerAngles euler_angles;

  if ( file_processor.getInitializationValue("euler angles:", euler_angles) )
  {
    putInitEulerAngles(euler_angles);
  }

  // Allow 'pilot' to be initialized with the same file.
  if ( pilot )
  {
    pilot->processInitializationFile(file_processor);
  }
}

//====================================================================
// This routine calculates the world relative variables using only the
// positional model state variables and the current state of the world.
//====================================================================
void PositionalModel::calcWorldRelStates(
  GeodeticCoordinates&  geodetic_coordinates,
  Vector<double>&       world_rel_vel_body,
  EulerAngles&          euler_angles,
  Vector<AngularValue>& local_vertical_rel_angular_vel_body)
{
  RotationMatrix inertial_to_body = RotationMatrix(getOrientation());

  // Compute the world-relative position of the positional model in world
  // coordinates and set the geodetic coordinates using the result.
  geodetic_coordinates.putWorldRelVector(
    world->getInertialToWorld() * (getPosition() - world->getPosition()));

  // The world relative velocity is a combination of the difference between the
  // inertial velocities of the positional model and the world and the cross
  // product of the world's rotation and the positional model's position
  // relative to the world.
  Vector<double> world_rel_vel =
    world->getInertialToWorld()* (getVelocity() - world->getVelocity()) -
    (world->getAngularVelWorld() ^ geodetic_coordinates.getWorldRelVector());

  RotationMatrix world_to_body =
    inertial_to_body * world->getWorldToInertial();

  RotationMatrix topodetic_to_body =
    world_to_body * ~(geodetic_coordinates.calcWorldToTopodetic());

  world_rel_vel_body = world_to_body * world_rel_vel;

  euler_angles = EulerAngles(topodetic_to_body);

  // The world relative angular velocity is the difference between the
```

```
  // inertial angular velocities of the positional model and the world.
  Vector<AngularValue> world_rel_angular_vel_body =
    getAngularVelBody() - world_to_body * world->getAngularVelWorld();

  local_vertical_rel_angular_vel_body =
    calcLocalVerticalRelAngularVelBody(geodetic_coordinates,
                                       world_rel_vel_body,
                                       euler_angles,
                                       world_rel_angular_vel_body);
}

//=============================================================================
// This method sets the underlying inertial states based on the world relative
// state arguments. (Velocity in fps; angular velocity in angle/second.)
//=============================================================================
void PositionalModel::putWorldRelStates(
  const GeodeticCoordinates&  new_geodetic_coordinates,
  const Vector<double>&       new_world_rel_vel_body,
  const EulerAngles&          new_euler_angles,
  const Vector<AngularValue>& new_local_vert_rel_angular_vel_body)
{
  geodetic_coordinates               = new_geodetic_coordinates;
  world_rel_vel_body                 = new_world_rel_vel_body;
  euler_angles                       = new_euler_angles;
  local_vertical_rel_angular_vel_body = new_local_vert_rel_angular_vel_body;

  // First calculate the inertial orientation from the geodetic coordinates and
  // the Euler angles.
  topodetic_rel_orientation = UnitQuaternion(euler_angles);

  world_to_topodetic = geodetic_coordinates.calcWorldToTopodetic();

  world_rel_topodetic_orientation = UnitQuaternion(world_to_topodetic);

  world_rel_orientation =
    world_rel_topodetic_orientation * topodetic_rel_orientation;

  putOrientation(world->getOrientation() * world_rel_orientation);

  // Compute the rotation matrices between the world and body coordinate
  // systems and between the inertial and body coordinate systems.
  world_to_body    =  RotationMatrix(world_rel_orientation);
  body_to_world    = ~world_to_body;
  inertial_to_body = RotationMatrix(getOrientation());
  body_to_inertial = ~inertial_to_body;

  // Compute the inertial angular velocity in body coordinates.  It is a
  // function of all the world-relative state arguments.
  world_rel_angular_vel_body =
    calcWorldRelAngularVelBody(geodetic_coordinates,
                               world_rel_vel_body,
                               euler_angles,
                               local_vertical_rel_angular_vel_body);

  putAngularVelBody(world_rel_angular_vel_body
                    + world_to_body * world->getAngularVelWorld());
```

```
  // Compute the inertial velocity
  world_rel_vel     = body_to_world * world_rel_vel_body;
  world_rel_vel_mag = world_rel_vel.mag();

  putVelocity(world->getVelocity() + world->getWorldToInertial() *
              (world_rel_vel + (world->getAngularVelWorld() ^
                                geodetic_coordinates.getWorldRelVector())));

  // Compute the inertial posiiton.
  putPosition(
    world->getWorldToInertial() * geodetic_coordinates.getWorldRelVector());

  // This method does not recalculate all derived states.
  derived_states_consistent = false;
}

//=================================================================
// This method sets the local vertical angular velocity.  Since the local
// vertical angular velocity is a derived state, the method changes the
// underlying inertial states so that the new local vertical angular velocity
// will be calculated the next time calcDerivedStates() is called.  This
// routine keeps the world relative velocity in body coordinates (U,V,W), the
// Euler angles, and the geodetic coordinates constant.
// (Argument has units of angle/second.)
//=================================================================
void PositionalModel::putLocalVerticalRelAngularVelBody(
  const Vector<AngularValue>& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Vector<AngularValue> new_angular_velocity = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  local_vertical_rel_angular_vel_body = new_angular_velocity;

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=================================================================
// This method sets the local vertical roll rate.  Since the local vertical
// roll rate is a derived state, the method changes the underlying inertial
// states so that the new local vertical roll rate will be calculated the next
// time calcDerivedStates() is called.  This routine keeps the world relative
// velocity in body coordinates (U,V,W), the Euler angles, and the geodetic
// coordinates constant. (Argument has units of angle/second)
//=================================================================
void PositionalModel::putLocalVerticalRelP(const AngularValue& new_value)
{
```

```
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue new_p = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  local_vertical_rel_angular_vel_body[0] = new_p;

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=================================================================
// This method sets the local vertical pitch rate.  Since the local vertical
// pitch rate is a derived state, the method changes the underlying inertial
// states so that the new local vertical pitch rate will be calculated the
// next time calcDerivedStates() is called.  This routine keeps the world
// relative velocity in body coordinates (U,V,W), the world relative
// orientation (Euler angles), and the geodetic coordinates constant.
// (Argument has units of angle/second.)
//=================================================================
void PositionalModel::putLocalVerticalRelQ(const AngularValue& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue new_q = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  local_vertical_rel_angular_vel_body[1] = new_q;

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=================================================================
// This method sets the local vertical yaw rate.  Since the local vertical yaw
// rate is a derived state, the method changes the underlying inertial states
// so that the new local vertical yaw rate will be calculated the next time
// calcDerivedStates() is called.  This routine keeps the world relative
// velocity in body coordinates (U,V,W), the Euler angles, and the geodetic
// coordinates constant. (Arguments have units of angle/second.)
//=================================================================
```

```
void PositionalModel::putLocalVerticalRelR(const AngularValue& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue new_r = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  local_vertical_rel_angular_vel_body[2] = new_r;

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//===========================================================================
// This method sets the Euler angle rates.  Since the Euler angle rates are
// derived state, the method changes the underlying inertial states so that
// the new euler angle rates will be calculated the next time calcDerived
// states is called.  The method holds constant the geodetic coordinates,
// world relative velocity, and Euler angles constant.  (Argument has units of
// angle/second.)
//===========================================================================
void PositionalModel::putEulerAngleRates(
  const Vector<AngularValue>& new_euler_angle_rates)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Vector<AngularValue> temp_euler_angle_rates = new_euler_angle_rates;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  euler_angle_rates = temp_euler_angle_rates;

  local_vertical_rel_angular_vel_body =
    calcAngularVelBody(euler_angles, euler_angle_rates);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//===========================================================================
// This method sets the rate of change of phi.  Since phi is a derived state,
// the method changes the underlying inertial states so that the new "phidot"
```

```
// will be calculated the next time calcDerived states is called.  The method
// attempts to hold geodetic coordinates, world relative velocity, and Euler
// angles constant.  (Argument has units of angle/second.)
//=====================================================================
void PositionalModel::putPhidot(const AngularValue& new_phidot)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue temp_phidot = new_phidot;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  euler_angle_rates =
    calcEulerAngleRates(euler_angles, local_vertical_rel_angular_vel_body);

  euler_angle_rates[0] = temp_phidot;

  local_vertical_rel_angular_vel_body =
    calcAngularVelBody(euler_angles, euler_angle_rates);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=====================================================================
// This method sets the rate of change of theta.  Since theta is a derived
// state, the method changes the underlying inertial states so that the new
// "thetadot" will be calculated the next time calcDerived states is called.
// The method attempts to hold geodetic coordinates, world relative velocity,
// and Euler angles constant. (Argument has units of angle/second.)
//=====================================================================
void PositionalModel::putThetadot(const AngularValue& new_thetadot)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue temp_thetadot = new_thetadot;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);


  euler_angle_rates = calcEulerAngleRates(euler_angles,
                                          local_vertical_rel_angular_vel_body);

  euler_angle_rates[1] = temp_thetadot;
```

```
  local_vertical_rel_angular_vel_body = calcAngularVelBody(euler_angles,
                                                           euler_angle_rates);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body );
}

//=================================================================
// This method sets the rate of change of psi.  Since psi is a derived state,
// the method changes the underlying inertial states so that the new "psidot"
// will be calculated the next time calcDerived states is called.  The method
// attempts to hold geodetic coordinates, world relative velocity, and Euler
// angles constant. (Argument has units of angle/second.)
//=================================================================
void PositionalModel::putPsidot(const AngularValue& new_psidot)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue temp_psidot = new_psidot;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);


  euler_angle_rates = calcEulerAngleRates(euler_angles,
                                          local_vertical_rel_angular_vel_body);

  euler_angle_rates[2] = temp_psidot;


  local_vertical_rel_angular_vel_body = calcAngularVelBody(euler_angles,
                                                           euler_angle_rates);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body );
}

//=================================================================
// This method sets the velocity relative to the primary reference point.
// Since the reference point relative velocity is a derived state, the method
// changes the underlying inertial states so that the new reference point
// relative velocity will be calculated the next time calcDerivedStates() is
// called.  The Euler angles, the world relative position (geodetic
// coordinates), and the local vertical relative angular velocity are held
// constant.  (Argument has units of feet/second.  The vector is measured in
// the local coordinate system of the primary reference point.)
//=================================================================
void PositionalModel::putRefPointRelVel(const Vector<double>& new_value)
```

```
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Vector<double> new_relative_velocity = new_value;

  inertial_to_body = RotationMatrix(getOrientation());
  body_to_inertial = ~inertial_to_body;

  geodetic_coordinates.putWorldRelVector(
    world->getInertialToWorld() * ( getPosition() - world->getPosition() ) );

  world_to_topodetic = geodetic_coordinates.calcWorldToTopodetic();

  topodetic_to_body =
    inertial_to_body * world->getWorldToInertial() * (~world_to_topodetic);

  if ( !primary_geo_ref_rel_info_handle->getGeoRefPoint() )
  {
    putWorldRelVelBody(topodetic_to_body*new_relative_velocity);
  }
  else
  {
    putWorldRelVelBody(
      topodetic_to_body*
      (primary_geo_ref_rel_info_handle->getGeoRefPoint()->getLocalToTopodetic()
       * new_relative_velocity));
  }
}

//===================================================================
// This method sets the north velocity relative to the primary reference point.
// Since the reference-point-relative, north velocity is a derived state, the
// method changes the underlying inertial states so that the new reference-
// point-relative, north velocity will be calculated the next time
// calcDerivedStates() is called.  The Euler angles, the world relative
// position (geodetic coordinates), and local vertical relative angular
// velocity are held constant. (Argument has units of feet/second.  The vector
// is measured in the local coordinate system of the primary reference point.)
//===================================================================
void PositionalModel::putSxdot(double feet_per_second)
{
  Vector<double> ref_point_rel_vel =
    primary_geo_ref_rel_info_handle->getRelVelocityLocal();
  ref_point_rel_vel[0] = feet_per_second;
  putRefPointRelVel(ref_point_rel_vel);
}

//===================================================================
// This method sets the east velocity relative to the primary reference point.
// Since the reference-point-relative, east velocity is a derived state, the
// method changes the underlying inertial states so that the new reference-
// point-relative, east velocity will be calculated the next time
// calcDerivedStates() is called.  The Euler angles, the world relative
// position (geodetic coordinates), and the local vertical relative angular
// velocity are held constant. (Argument has units of feet/second.  The vector
```

```
// is measured in the local coordinate system of the primary reference point.)
//=====================================================================
void PositionalModel::putSydot(double feet_per_second)
{
  Vector<double> ref_point_rel_vel =
    primary_geo_ref_rel_info_handle->getRelVelocityLocal();
  ref_point_rel_vel[1] = feet_per_second;
  putRefPointRelVel(ref_point_rel_vel);
}

//=====================================================================
// This method sets the vertical velocity relative to the primary reference
// point. Since the reference-point-relative, vertical velocity is a derived
// state, the method changes the underlying inertial states so that the new
// reference-point-relative, vertical velocity will be calculated the next
// time calcDerivedStates() is called.  The Euler angles, the world relative
// position (geodetic coordinates), and the local vertical relative angular
// velocity are held constant. (Argument has units of feet/second.  The vector
// is measured in the local coordinate system of the primary reference point.)
//=====================================================================
void PositionalModel::putSzdot(double feet_per_second)
{
  Vector<double> ref_point_rel_vel =
    primary_geo_ref_rel_info_handle->getRelVelocityLocal();
  ref_point_rel_vel[2] = feet_per_second;
  putRefPointRelVel(ref_point_rel_vel);
}

//=====================================================================
// This method sets the world relative angular velocity in body coordinates.
// Since the world relative angular velocity in body coordinates is a derived
// state, the method changes the underlying inertial states so that the new
// world relative angular velocity in body coordinates will be calculated the
// next time calcDerivedStates() is called.  The Euler angles, the world
// relative position (geodetic coordinates), and world relative velocity
// (U, V, W) are held constant. (Argument has units of angle/second.)
//=====================================================================
void PositionalModel::putWorldRelAngularVelBody(
  const Vector<AngularValue>& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Vector<AngularValue> new_angular_velocity = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                      euler_angles,
                      local_vertical_rel_angular_vel_body);

  world_rel_angular_vel_body = new_angular_velocity;

  local_vertical_rel_angular_vel_body =
    calcLocalVerticalRelAngularVelBody(geodetic_coordinates,
                                       world_rel_vel_body,
                                       euler_angles,
```

```
                                        world_rel_angular_vel_body);


  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=====================================================================
// This method sets the world relative roll rate.  Since the world relative
// roll rate is a derived state, the method changes the underlying inertial
// states so that the new world relative roll rate will be calculated the
// next time calcDerivedStates() is called.  The Euler angles, the world
// relative position (geodetic coordinates), and world relative velocity (U,
// V, W) are held constant.  (Argument has units of angle/second)
//=====================================================================
void PositionalModel::putP(const AngularValue& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue new_p = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  world_rel_angular_vel_body =
    calcWorldRelAngularVelBody(geodetic_coordinates,
                               world_rel_vel_body,
                               euler_angles,

local_vertical_rel_angular_vel_body);

  world_rel_angular_vel_body[0] = new_p;

 local_vertical_rel_angular_vel_body =
    calcLocalVerticalRelAngularVelBody(geodetic_coordinates,
                                       world_rel_vel_body,
                                       euler_angles,
                                       world_rel_angular_vel_body);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=====================================================================
// This method sets the world relative pitch rate.  Since the world relative
// pitch rate is a derived state, the method changes the underlying inertial
// states so that the new world relative pitch rate will be calculated the
// next time calcDerivedStates() is called.  The Euler angles, the world
// relative position (geodetic coordinates), and world relative velocity
```

```
// (U, V, W) are held constant. (Argument has units of angle/second.)
//=================================================================
void PositionalModel::putQ(const AngularValue& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue new_q = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  world_rel_angular_vel_body =
    calcWorldRelAngularVelBody(geodetic_coordinates,
                               world_rel_vel_body,
                               euler_angles,
                               local_vertical_rel_angular_vel_body);

  world_rel_angular_vel_body[1] = new_q;

  local_vertical_rel_angular_vel_body =
    calcLocalVerticalRelAngularVelBody(geodetic_coordinates,
                                       world_rel_vel_body,
                                       euler_angles,
                                       world_rel_angular_vel_body);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=================================================================
// This method sets the world relative yaw rate.  Since the world relative
// yaw rate is a derived state, the method changes the underlying inertial
// states so that the new world relative yaw rate will be calculated the
// next time calcDerivedStates() is called.  The Euler angles, world relative
// position (geodetic coordinates), and world relative velocity (U, V, W) are
// held constant.  (Argument has units of angle/second.)
//=================================================================
void PositionalModel::putR(const AngularValue& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  AngularValue new_r = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  world_rel_angular_vel_body =
```

```
    calcWorldRelAngularVelBody(geodetic_coordinates,
                               world_rel_vel_body,
                               euler_angles,
                               local_vertical_rel_angular_vel_body);

  world_rel_angular_vel_body[2] = new_r;

  local_vertical_rel_angular_vel_body =
    calcLocalVerticalRelAngularVelBody(geodetic_coordinates,
                                       world_rel_vel_body,
                                       euler_angles,
                                       world_rel_angular_vel_body);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//===================================================================
// This method calculates the world relative angular velocity in body
// coordinates as a function of the geodetic coordinates, the world relative
// velocity in body coordinates, the Euler angles, and the local vertical
// relative velocity in body coordinates.
//===================================================================
Vector<double> PositionalModel::calcWorldRelAngularVelBody(
  const GeodeticCoordinates&  geodetic_coordinates,
  const Vector<double>&       world_rel_vel_body,
  const EulerAngles&          euler_angles,
  const Vector<AngularValue>& local_vertical_rel_angular_vel_body) const
{
  RotationMatrix topodetic_to_body(euler_angles);
  RotationMatrix body_to_topodetic = ~topodetic_to_body;

  Vector<double> world_rel_vel_topodetic =
    body_to_topodetic * world_rel_vel_body;

  AngularValue latitude_dot;
  AngularValue longitude_dot;

  geodetic_coordinates.calcGeodeticCoordinateRates(latitude_dot,  // output
                                                   longitude_dot, // output
                                                   world_rel_vel_topodetic);

  Vector<AngularValue> world_rel_local_vert_angular_vel_topo =
    Vector<AngularValue>(longitude_dot*
                         geodetic_coordinates.getLatitude().getCosine(),
                         -latitude_dot,
                         -longitude_dot*
                         geodetic_coordinates.getLatitude().getSine());

  return local_vertical_rel_angular_vel_body +
    topodetic_to_body * world_rel_local_vert_angular_vel_topo;
}

//===================================================================
// This method calculates the local vertical relative angular velocity in
```

```
// body coordinates as a function of the geodetic coordinates, the world
// relative velocity in body coordinates, the Euler angles, and the world
// relative angular velocity in body coordinates.
//===========================================================================
Vector<double> PositionalModel::calcLocalVerticalRelAngularVelBody(
  const GeodeticCoordinates&  geodetic_coordinates,
  const Vector<double>&       world_rel_vel_body,
  const EulerAngles&          euler_angles,
  const Vector<AngularValue>& world_rel_angular_vel_body) const
{
  RotationMatrix topodetic_to_body(euler_angles);
  RotationMatrix body_to_topodetic = ~topodetic_to_body;

  Vector<double> world_rel_vel_topodetic =
    body_to_topodetic * world_rel_vel_body;

  AngularValue latitude_dot;
  AngularValue longitude_dot;

  geodetic_coordinates.calcGeodeticCoordinateRates(latitude_dot,  // output
                                                   longitude_dot, // output
                                                   world_rel_vel_topodetic);

  Vector<AngularValue> world_rel_local_vert_angular_vel_topo =
    Vector<AngularValue>(longitude_dot*
                         geodetic_coordinates.getLatitude().getCosine(),
                         -latitude_dot,
                         -longitude_dot*
                         geodetic_coordinates.getLatitude().getSine());

  return world_rel_angular_vel_body -
            topodetic_to_body *  world_rel_local_vert_angular_vel_topo;
}

//===========================================================================
// This method sets the Euler angles.  Since the euler angles are derived
// states, the method changes the underlying inertial states so that the new
// euler angles will be calculated the next time calcDerivedStates() is
// called.  World relative position (geodetic coordinates), world relative
// velocity (U, V, W), and local vertical relative angular velocity are held
// constant.
//===========================================================================
void PositionalModel::putEulerAngles(const EulerAngles& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  EulerAngles new_euler_angles = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  euler_angles = new_euler_angles;
```

```
  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=====================================================================
// This method sets psi.  Since the psi is a derived state, the method changes
// the underlying inertial states so that the new psi will be calculated the
// next time calcDerivedStates() is called.  World relative position (geodetic
// coordinates), world relative velocity (U, V, W), and local vertical
// relative angular velocity are held constant.
//=====================================================================
void PositionalModel::putPsi(const Angle& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Angle new_angle = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  euler_angles.putPsi(new_angle);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=====================================================================
// This method sets theta.  Since the theta is a derived state, the method
// changes the underlying inertial states so that the new theta will be
// calculated the next time calcDerivedStates() is called.  World relative
// position (geodetic coordinates), world relative velocity (U, V, W), and
// local vertical relative angular velocity are held constant.
//=====================================================================
void PositionalModel::putTheta(const Angle& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Angle new_angle = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  euler_angles.putTheta(new_angle);

  putWorldRelStates(geodetic_coordinates,
```

```
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=========================================================================
// This method sets phi.  Since the phi is a derived state, the method changes
// the underlying inertial states so that the new phi will be calculated the
// next time calcDerivedStates() is called.  World relative position (geodetic
// coordinates), world relative velocity (U, V, W), and local vertical
// relative angular velocity are held constant.
//=========================================================================
void PositionalModel::putPhi(const Angle& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Angle new_angle = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  euler_angles.putPhi(new_angle);

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=========================================================================
// This method sets the world relative velocity in body coordinates.  Since
// the world relative velocity in body coordinates is a derived state, this
// method changes the underlying inertial states so that the new world
// relative velocity in body coordinates will be calculated the next time
// calcDerivedStates() is called.  World relative position (geodetic
// coordinates), the Euler angles, and the local vertical relative angular
// velocity are held constant.  (Argument has units of feet/second.)
//=========================================================================
void PositionalModel::putWorldRelVelBody(const Vector<double>& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Vector<double> new_velocity = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  world_rel_vel_body = new_velocity;
  world_rel_vel_mag  = world_rel_vel_body.mag();
```

```
  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=================================================================
// This method sets world relative velocity along the X body axis (U).  Since
// U is a derived state, this method changes the underlying inertial states
// so that the new U will be calculated the next time calcDerivedStates() is
// called.  World relative position (geodetic coordinates), the Euler angles,
// and the local vertical relative angular velocity are held constant.
// (Argument has units of feet/second.)
//=================================================================
void PositionalModel::putU(double feet_per_second)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  world_rel_vel_body[0] = feet_per_second;
  world_rel_vel_mag = world_rel_vel_body.mag();

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=================================================================
// This method sets world relative velocity along the Y body axis (V).  Since
// V is a derived state, this method changes the underlying inertial states
// so that the new V will be calculated the next time calcDerivedStates() is
// called.  World relative position (geodetic coordinates), the Euler angles,
// and the local vertical relative angular velocity are held constant.
// (Argument has units of feet/second.)
//=================================================================
void PositionalModel::putV(double feet_per_second)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  world_rel_vel_body[1] = feet_per_second;
  world_rel_vel_mag = world_rel_vel_body.mag();

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=================================================================
// This method sets world relative velocity along the Z body axis (W).  Since
```

```
// W is a derived state, this method changes the underlying inertial states
// so that the new W will be calculated the next time calcDerivedStates() is
// called.  World relative position (geodetic coordinates), the Euler angles,
// and the local vertical relative angular velocity are held constant.
// (Argument has units of feet/second.)
//=============================================================
void PositionalModel::putW(double feet_per_second)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  world_rel_vel_body[2] = feet_per_second;
  world_rel_vel_mag = world_rel_vel_body.mag();

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=============================================================
// This method sets the world relative position.  Since the world relative
// position is a derived state, this method changes the underlying inertial
// states so that the new world relative position will be calculated the next
// time calcDerivedStates() is called.  World relative velocity in body
// coordinates (U, V, W), the Euler angles, and the local vertical relative
// angular velocity are held constant.  (Argument has units of feet)
//=============================================================
void PositionalModel::putWorldRelVector(const Vector<double>& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Vector<double> new_position = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  geodetic_coordinates.putWorldRelVector(new_position);


  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=============================================================
// This method sets the world relative position.  Since the world relative
// position is a derived state, this method changes the underlying inertial
// states so that the new world relative position will be calculated the next
// time calcDerivedStates() is called.  World relative velocity in body
```

```
// coordinates (U, V, W), the Euler angles, and the local vertical relative
// angular velocity are held constant.
//====================================================================
void PositionalModel::putGeodeticCoordinates(
  const GeodeticCoordinates& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  GeodeticCoordinates new_position = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  geodetic_coordinates = new_position;


  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//====================================================================
// This method sets the world relative position as a distance from the primary
// reference point.  Since the world relative position is a derived state,
// this method changes the underlying inertial states so that the new distance
// from the primary reference point will be calculated the next time
// calcDerivedStates() is called.  World relative velocity in body coordinates
// (U, V, W), the Euler angles, and the local vertical relative angular
// velocity are held constant.  The vector is measured in the "local" (i.e.
// body) coordinate system of the primary refernce point.  (Argument has
// units of feet.)
//====================================================================
void PositionalModel::putDistanceFromRefPoint(const Vector<double>& new_value)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the PositionalModel class.  This method makes
  // other PositionalModel method calls that can potentially change the value
  // of the argument during the execution of this method.
  Vector<double> new_position = new_value;

  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  if ( !primary_geo_ref_rel_info_handle->getGeoRefPoint() )
  {
    geodetic_coordinates.putAltitude(-new_position[2]);
  }
  else
  {
    geodetic_coordinates.setUsingDistanceFrom(
```

```
      *(primary_geo_ref_rel_info_handle->getGeoRefPoint()),
      new_position);
  }

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=========================================================================
// This method sets the north distance from the primary reference point.
// Since the north distance from the primary reference point is a derived
// state, this method changes the underlying inertial states so that the new
// north distance from the primary reference point will be calculated the next
// time calcDerivedStates() is called.  World relative velocity in body
// coordinates (U, V, W), the Euler angles, and the local vertical relative
// angular velocity are held constant.  The vector is measured in the "local"
// (i.e.body) coordinate system of the primary reference point.  (Argument has
//  units of feet.)
//=========================================================================
void PositionalModel::putSx(double distance_feet)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  Vector<double> distance_from_ref_point;

  geodetic_coordinates.calcHorizontalDistanceFrom(
                       *primary_geo_ref_rel_info_handle->getGeoRefPoint(),
                       distance_from_ref_point);

  distance_from_ref_point[0] = distance_feet;


  putDistanceFromRefPoint(distance_from_ref_point);
}

//=========================================================================
// This method sets the east distance from the primary reference point.
// Since the east distance from the primary reference point is a derived
// state, this method changes the underlying inertial states so that the new
// east distance from the primary reference point will be calculated the next
// time calcDerivedStates() is called.  World relative velocity in body
// coordinates (U, V, W), the Euler angles, and the local vertical relative
// angular velocity are held constant.  The vector is measured in the "local"
// (i.e. body) coordinate system of the primary reference point. (Argument has
//  units of feet.)
//=========================================================================
void PositionalModel::putSy(double distance_feet)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);
```

```
  Vector<double> distance_from_ref_point;

  geodetic_coordinates.calcHorizontalDistanceFrom(
                       *primary_geo_ref_rel_info_handle->getGeoRefPoint(),
                       distance_from_ref_point);

  distance_from_ref_point[1] = distance_feet;


  putDistanceFromRefPoint(distance_from_ref_point);
}

//=====================================================================
// This method sets the vertical distance from the primary reference point.
// Since the vertical distance from the primary reference point is a derived
// state, this method changes the underlying inertial states so that the new
// vertical distance from the primary reference point will be calculated the
// next time calcDerivedStates() is called.  World relative velocity in body
// coordinates (U, V, W), the Euler angles, and the local vertical relative
// angular velocity are held constant.  The vector is measured in the "local"
// (i.e. body) coordinate system of the primary reference point. (Argument has
//  units of feet.)
//=====================================================================
void PositionalModel::putSz(double distance_feet)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  Vector<double> distance_from_ref_point;

  geodetic_coordinates.calcHorizontalDistanceFrom(
                       *primary_geo_ref_rel_info_handle->getGeoRefPoint(),
                       distance_from_ref_point);

  distance_from_ref_point[2] = distance_feet;


  putDistanceFromRefPoint(distance_from_ref_point);
}

//=====================================================================
// This method sets the altitude.  Since the altitude is a derived state, this
// method changes the underlying inertial states so that the new altitude will
// be calculated the next time calcDerivedStates() is called.  World relative
// velocity in body coordinates (U, V, W), the Euler angles, and the local
// vertical relative angular velocity are held constant .  (Argument has units
// of feet.)
//=====================================================================
void PositionalModel::putAltitude(double new_altitude)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);
```

```
  geodetic_coordinates.putAltitude(new_altitude);


  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=========================================================================
// This method sets the climb rate.  Since climb rate is a derived state, this
// method changes the underlying inertial states so that the new climb rate
// will be calculated the next time calcDerivedStates() is called.  World
// relative  position (geodetic coordinates), the Euler angles, and the local
// vertical relative angular velocity are held constant.  (Argument has units
// of feet/second.)
//=========================================================================
void PositionalModel::putAltitudeDot(double feet_per_second)
{
  calcWorldRelStates(geodetic_coordinates,
                     world_rel_vel_body,
                     euler_angles,
                     local_vertical_rel_angular_vel_body);

  RotationMatrix topodetic_to_body_temp(euler_angles);

  Vector<double> world_rel_vel_topodetic_temp =
    ~topodetic_to_body_temp * world_rel_vel_body;

  world_rel_vel_topodetic_temp[2] = -feet_per_second;

  world_rel_vel_body = topodetic_to_body_temp * world_rel_vel_topodetic_temp;

  putWorldRelStates(geodetic_coordinates,
                    world_rel_vel_body,
                    euler_angles,
                    local_vertical_rel_angular_vel_body);
}

//=========================================================================
// This method sets the track angle.  Since track is a derived state, this
// method changes the underlying inertial states so that the new track will
// be calculated the next time calcDerivedStates() is called.  World relative
// position (geodetic coordinates), the world relative velocity in body
// coordinates, and the local vertical relative angular velocity are held
// constant.
//=========================================================================
void PositionalModel::putTrack(const Angle& new_track)
{
  // A temporary copy must be made to avoid side effects when the actual
  // argument is a member of the Vehicle class.  This method makes other
  // Vehicle method calls that can potentially change the value of the
  // argument during the execution of this method.
  Angle temp_track = new_track;

  GeodeticCoordinates  geodetic_coordinates_temp(world->getWorldShapeData());
```

```
  Vector<double>       world_rel_vel_body_temp;
  EulerAngles          euler_angles_temp;
  Vector<AngularValue> local_vertical_rel_angular_vel_body_temp;

  calcWorldRelStates(geodetic_coordinates_temp,
                     world_rel_vel_body_temp,
                     euler_angles_temp,
                     local_vertical_rel_angular_vel_body_temp);

  RotationMatrix topodetic_to_body_temp(euler_angles_temp);

  Vector<double> world_rel_vel_topodetic_temp =
    ~topodetic_to_body_temp * world_rel_vel_body_temp;

  double limited_new_track=fmod(temp_track,two_pi);

  if ( fabs(world_rel_vel_topodetic_temp[1])>1e-8 ||
       fabs(world_rel_vel_topodetic_temp[0])>1e-8 )
  {
    // Ground track angle
    track = atan2( world_rel_vel_topodetic_temp[1] ,
                   world_rel_vel_topodetic_temp[0] );
  }
  else
  {
    track = euler_angles_temp.getPsi();
  }

  double limited_track = fmod(track,two_pi);

  euler_angles_temp.putPsi(
    euler_angles_temp.getPsi()+(limited_new_track-limited_track));

  track = limited_new_track;

  putWorldRelStates(geodetic_coordinates_temp,
                    world_rel_vel_body_temp,
                    euler_angles_temp,
                    local_vertical_rel_angular_vel_body_temp);
}

//=====================================================================
// This method sets the primary geographic reference point.
//=====================================================================
void PositionalModel::putPrimaryGeoRefPoint(const GeoRefPoint* geo_ref_point)
{
  primary_geo_ref_rel_info_handle->putGeoRefPoint(geo_ref_point);

  if ( maintain_init_position_as_geodetic_coordinates )
  {
    putInitGeodeticCoordinates(init_geodetic_coordinates);
  }
  else
  {
    putInitDistanceFromRefPoint(init_distance_from_ref_point);
  }
}
```

```
//=========================================================================
// This method sets the number of hits needed to kill the target to the new
// value if the simulation is not in OPERATE and then sets the "hit_points"
// to this value.  "hit_point" is counted down to determine if the model has
// been killed.
//=========================================================================
void PositionalModel::putInitialHitPoints(
  const double new_initial_hit_points)
{
  if ( getMode() != OPERATE )
  {
    initial_hit_points = new_initial_hit_points;
    hit_points         = initial_hit_points;
  }
}

//=========================================================================
// This method returns the initial value for the inertial position. (feet)
//=========================================================================
const Vector<double>& PositionalModel::getPositionInitialCondition() const
{
  return position->getInitialCondition();
}

//=========================================================================
// This method returns the initial value for the inertial velocity. (fps)
//=========================================================================
const Vector<double>& PositionalModel::getVelocityInitialCondition() const
{
  return velocity->getInitialCondition();
}

//=========================================================================
// This method returns the initial value for the inertial orientation.
// (quaternion)
//=========================================================================
const UnitQuaternion& PositionalModel::getOrientationInitialCondition() const
{
  return orientation->getInitialCondition();
}

//=========================================================================
// This method returns the initial value for the inertial angular velocity in
// body coordinates. (angle/second)
//=========================================================================
const Vector<AngularValue>&
PositionalModel::getAngularVelBodyInitialCondition() const
{
  return angular_vel_body->getInitialCondition();
}

//=========================================================================
// This method returns the initial distance from the primary geographic
// reference point.  (feet)
//=========================================================================
const Vector<double>& PositionalModel::getInitDistanceFromRefPoint() const
```

```
{
  return init_distance_from_ref_point;
}

//====================================================================
// This method returns the initial geodetic coordinates.
//====================================================================
const GeodeticCoordinates& PositionalModel::getInitGeodeticCoordinates() const
{
  return init_geodetic_coordinates;
}


//====================================================================
// This method returns the initial track angle.
//====================================================================
const Angle& PositionalModel::getTrackInit() const
{
  return track_init;
}

//====================================================================
// This method sets the initial geodetic coordinates.  Since geodetic
// coordinates are a derived state, the method changes the initial inertial
// coordinates to be consistent with the initial geodetic coordinates.
// Positional model attempts to maintain initial euler angles, initial world
// relative velocity in body coordinates, and initial local vertical angular
// velocity.
//====================================================================
void PositionalModel::putInitGeodeticCoordinates(
  const GeodeticCoordinates& new_value)
{
  maintain_init_position_as_geodetic_coordinates = true;
  init_geodetic_coordinates                      = new_value;

  // Compute the new initial distance from the primary reference point.
  const GeoRefPoint* reference_point = getPrimaryGeoRefPoint();

  if ( reference_point )
  {
    init_geodetic_coordinates.
      calcHorizontalDistanceFrom(*reference_point,
                                 init_distance_from_ref_point);
  }
  else
  {
    init_distance_from_ref_point[2] = -init_geodetic_coordinates.getAltitude();
  }

  calcInitialConditions();
}

//====================================================================
// This method sets the initial distance from the primary geographic reference
// point, if it exists.  If it does not exist, only the initial altitude is
// changed based on the vertical component of the distance vector.  The
// initial geodetic coordinates are calculated from the initial distance and
```

```
// set.  Since geodetic coordinates are a derived state, the method changes
// the initial inertial coordinates to be consistent with the initial geodetic
// coordinates.  PositionalModel attempts to maintain initial euler angles,
// initial world relative velocity in body coordinates, and initial local
// vertical angular velocity.  (Argument has units of feet.)
//=========================================================================
void PositionalModel::putInitDistanceFromRefPoint(
  const Vector<double>& new_value)
{
  maintain_init_position_as_geodetic_coordinates = false;
  init_distance_from_ref_point                   = new_value;

  const GeoRefPoint* reference_point = getPrimaryGeoRefPoint();

  if ( reference_point )
  {
    init_geodetic_coordinates.
      setUsingDistanceFrom(*reference_point, init_distance_from_ref_point);
  }
  else
  {
    init_geodetic_coordinates.putAltitude(-init_distance_from_ref_point[2]);
  }

  calcInitialConditions();
}

//=========================================================================
// This method sets the initial world relative velocity in body coordinates.
// Since the world relative velocity is a derived state, the method changes
// the initial inertial velocity to be consistent with the initial world
// relative velocity. PositionalModel also attempts to maintain initial euler
// angles, initial geodetic coordinates, and initial local vertical angular
// velocity.   (Argument has units of feet/second.)
//=========================================================================
void PositionalModel::putWorldRelVelBodyInit(
  const Vector<double>& new_velocity)
{
  world_rel_vel_body_init = new_velocity;

  calcInitialConditions();
}


//=========================================================================
// This method sets the initial local vertical relative angular velocity in
// body coordinates.  Since the local vertical relative angular velocity is
// is a derived state, the method changes the initial inertial angular
// velocity to be consistent with the initial local vertical relative angular
// velocity. PositionalModel also attempts to maintain initial euler
// angles, initial geodetic coordinates, and initial world relative velocity.
// (Argument has units of angle/second.)
//=========================================================================
void PositionalModel::putLocalVerticalAngularVelBodyInit(
  const Vector<AngularValue>& new_velocity)
{
  local_vertical_angular_vel_body_init = new_velocity;
```

```
  calcInitialConditions();
}

//===========================================================================
// This method sets the initial Euler angles.  Since the Euler angles are
// derived states, the method changes the initial inertial orientation to be
// consistent with the initial Euler angles. PositionalModel also attempts to
// maintain initial geodetic coordinates, initial world relative velocity, and
// initial local vertical relative angular velocity.
//===========================================================================
void PositionalModel::putInitEulerAngles(const EulerAngles& new_value)
{
  init_euler_angles = new_value;

  calcInitialConditions();
}

//===========================================================================
// This method calculates inertial and derived initial conditions based on
// the "governing" initial conditions.  The governing initial conditions are
// initial geodetic coordinates, initial euler angles, initial world relative
// velocity in body coordinates, and the initial local vertical angular
// velocity in body coordinates.  The method is designed to maintain
// consistency between the initial conditions variables.
//===========================================================================
void PositionalModel::calcInitialConditions()
{
  // Compute the initial inertial position.
  Vector<double> init_inertial_position =
    world->getPosition() + world->getWorldToInertial() *
    init_geodetic_coordinates.getWorldRelVector();
  position->putInitialCondition(init_inertial_position);

  // Compute initial topodetic to body quaternion.
  UnitQuaternion init_topodetic_to_body_quat(init_euler_angles);

  // Compute initial world to body quaternion.
  UnitQuaternion init_world_to_topodetic_quat(
    init_geodetic_coordinates.calcWorldToTopodetic());
  UnitQuaternion init_world_to_body_quat(
    init_world_to_topodetic_quat * init_topodetic_to_body_quat);

  // Compute initial inertial orientation (i.e. inertial to body quaternion).
  orientation->putInitialCondition(
    world->getOrientation() * init_world_to_body_quat);

  // Calculate new inertial angular velocity (hold constant the world
  // relative angular velocity in body coordinates).
  // Convert to an initial inertial velocity.
  // Compute initial topodetic to body quaternion.
  RotationMatrix init_topodetic_to_body(init_euler_angles);
  RotationMatrix init_body_to_topodetic = ~init_topodetic_to_body;
  RotationMatrix init_world_to_body(init_world_to_body_quat);

  // Calculate new inertial velocity (hold world relative velocity in body
  // coordinates constant).
```

```
  Vector<double> world_rel_vel_init =
    ~init_world_to_body * world_rel_vel_body_init;
  Vector<double> world_rel_vector_init =
   init_geodetic_coordinates.getWorldRelVector();
  Vector<double> velocity_init = world->getVelocity() +
    world->getWorldToInertial() *
    (world_rel_vel_init + (world->getAngularVelWorld()^world_rel_vector_init));

  velocity->putInitialCondition(velocity_init);

  // Calculate the new inertial angular velocity in body coordinates. (Hold the
  // local vertical angular velocity in body coordinates constant.)

  // The rates of latitude and longitude are a function of the positional
  // model's world relative position and velocity.
  Vector<double> world_rel_vel_topo_init =
    init_body_to_topodetic * world_rel_vel_body_init;
  AngularValue latitude_dot_init;
  AngularValue longitude_dot_init;
  init_geodetic_coordinates.calcGeodeticCoordinateRates(
    latitude_dot_init, longitude_dot_init, world_rel_vel_topo_init);

  Vector<AngularValue> world_rel_local_vert_angular_vel_topo_init(
    longitude_dot_init * init_geodetic_coordinates.getLatitude().getCosine(),
    -latitude_dot_init,
    -longitude_dot_init * init_geodetic_coordinates.getLatitude().getSine());

  Vector<AngularValue> world_rel_angular_vel_body_init =
    local_vertical_angular_vel_body_init +
    init_topodetic_to_body * world_rel_local_vert_angular_vel_topo_init;

  // Calculate the initial inertial angular velocity in body coordinates.
  angular_vel_body->putInitialCondition(
    world_rel_angular_vel_body_init +
    init_world_to_body * world->getAngularVelWorld());

  // Calculate the initial track.
  double velocity_north = world_rel_vel_topo_init[0];
  double velocity_east  = world_rel_vel_topo_init[1];

  if ( fabs(velocity_east) > 1.0e-8 || fabs(velocity_north) > 1.0e-8 )
  {
    // An reliable calculation of the track angle requires that either the
    // north or east components of the world relative velocity be outside the
    // neighborhood of zero.
    track_init = atan2(velocity_east , velocity_north);
  }
  else
  {
    // The vehicle is either not moving or is only traveling in the
    // z-direction.  Approximate the track using psi.
    track_init = init_euler_angles.getPsi();
  }

  putModifiedICs(true);
}
```

```
//=========================================================================
// This method sets the initial inertial position.  The method also sets the
// initial geodetic coordinates and the initial distance from the primary
// reference so that they are consistent with the inertial position.  The
// method also changes the initial inertial orientation to maintain initial
// euler angles (i.e., the initial orientation relative to the world).
// (Argument has units of feet.)
//=========================================================================
void PositionalModel::putPositionInitialCondition(
  const Vector<double>& new_position)
{
  position->putInitialCondition(new_position);

  // Set the initial geodetic coordinates based on the new inertial position.
  init_geodetic_coordinates.putWorldRelVector(
    world->getInertialToWorld() * (new_position - world->getPosition()));

  // Compute the new initial distance from the primary reference point.
  const GeoRefPoint* reference_point = getPrimaryGeoRefPoint();

  if ( reference_point )
  {
    init_geodetic_coordinates.
      calcHorizontalDistanceFrom(*reference_point,
                                 init_distance_from_ref_point);
  }
  else
  {
    init_distance_from_ref_point[2] = -init_geodetic_coordinates.getAltitude();
  }

  calcInitialConditions();
}

//=========================================================================
// This method sets the initial inertial velocity.  Initial world relative
// velocity is also modified assuming that initial geodetic coordinates and
// Euler angles are held constant.  (Argument has units of feet/second.)
//=========================================================================
void PositionalModel::putVelocityInitialCondition(
  const Vector<double>& new_velocity)
{
  velocity->putInitialCondition(new_velocity);

  // Compute initial world relative velocity in body coordinates.
  Vector<double> init_world_rel_vector =
    init_geodetic_coordinates.getWorldRelVector();
  Vector<double> world_rel_vel_init =
    world->getInertialToWorld() * (new_velocity - world->getVelocity()) -
    (world->getAngularVelWorld() ^ init_world_rel_vector);

  RotationMatrix world_to_body(
    ~world->getOrientation() * orientation->getInitialCondition());
  world_rel_vel_body_init = world_to_body * world_rel_vel_init;

  calcInitialConditions();
}
```

```
//=========================================================================
// This method changes the initial inertial orientation.  The initial Euler
// angles are modified as a result.  The Positional model also attempts to
// maintain the initial geodetic coordinates, world relative velocity in body
// coordinates, and the local vertical angular velocity in body coordinates.
//=========================================================================
void PositionalModel::putOrientationInitialCondition(
  const UnitQuaternion& new_orientation)
{
  orientation->putInitialCondition(new_orientation);

  // Calculate new euler angles based on the new inertial orientation.
  UnitQuaternion init_world_to_topodetic(
    init_geodetic_coordinates.calcWorldToTopodetic());

  UnitQuaternion init_world_to_body =
    ~world->getOrientation() * new_orientation;

  UnitQuaternion init_topodetic_to_body =
    ~init_world_to_topodetic * init_world_to_body;

  init_euler_angles = EulerAngles(init_topodetic_to_body);

  // Recalculate initial conditions based on new initial euler angle.
  calcInitialConditions();
}

//=========================================================================
// This method changes the initial value for the inertial angular velocity in
// body coordinates. The initial local vertical angular velocity is changed as
// a result.  The PositionalModel also attempts to maintain initial geodetic
// coordinates, initial Euler angles, and initial world relative velocity in
// body coordinates.  (Argument has units of angle/second.)
//=========================================================================
void PositionalModel::putAngularVelBodyInitialCondition(
  const Vector<AngularValue>& new_angular_vel_body)
{
  angular_vel_body->putInitialCondition(new_angular_vel_body);

  // Convert to local vertical angular velocity.
  // Compute initial topodetic to body quaternion.
  RotationMatrix init_topodetic_to_body(init_euler_angles);
  RotationMatrix init_body_to_topodetic = ~init_topodetic_to_body;

  // Compute initial world to body quaternion.
  RotationMatrix init_world_to_topodetic(
    init_geodetic_coordinates.calcWorldToTopodetic());
  RotationMatrix init_world_to_body(
    init_topodetic_to_body * init_world_to_topodetic );

  // Calculate the initial world relative angular velocity in body coordinates.
  Vector<AngularValue> world_rel_angular_vel_body_init =
    new_angular_vel_body - init_world_to_body * world->getAngularVelWorld();

  // The rates of latitude and longitude are a function of the positional
  // model's world relative position and velocity.
```

```
  Vector<double> world_rel_vel_topo_init =
    init_body_to_topodetic * world_rel_vel_body_init;
  AngularValue latitude_dot_init;
  AngularValue longitude_dot_init;
  init_geodetic_coordinates.calcGeodeticCoordinateRates(
    latitude_dot_init, longitude_dot_init, world_rel_vel_topo_init);

  Vector<AngularValue> world_rel_local_vert_angular_vel_topo_init(
    longitude_dot_init * init_geodetic_coordinates.getLatitude().getCosine(),
    -latitude_dot_init,
    -longitude_dot_init * init_geodetic_coordinates.getLatitude().getSine());

  local_vertical_angular_vel_body_init = world_rel_angular_vel_body_init -
    init_topodetic_to_body * world_rel_local_vert_angular_vel_topo_init;

  calcInitialConditions();
}

//=========================================================================
// This method allows a client to set the initial values for all the inertial
// states with one call.  (Position has units of feet; velocity has units of
// feet per second; angular velocity has units of angle/second.)
//=========================================================================
void PositionalModel::putStateICs(
  const Vector<double>&       position_ic,
  const Vector<double>&       velocity_ic,
  const UnitQuaternion&       orientation_ic,
  const Vector<AngularValue>& angular_vel_body_ic)
{
  putPositionInitialCondition(position_ic);
  putOrientationInitialCondition(orientation_ic);
  putVelocityInitialCondition(velocity_ic);
  putAngularVelBodyInitialCondition(angular_vel_body_ic);
}

//=========================================================================
// This method sets the initial track angle.  Since track is a derived state,
// the method changes the initial inertial states to be consistent with the
// initial track angles. PositionalModel also attempts to maintain initial
// geodetic coordinates, initial world relative velocity in body coordinates,
// and initial local vertical relative angular velocity.
//=========================================================================
void PositionalModel::putTrackInit(const Angle& new_track)
{
  Angle track_init = fmod(new_track, two_pi);

  RotationMatrix world_to_topodetic_init =
    init_geodetic_coordinates.calcWorldToTopodetic();

  Vector<double> world_rel_vel_init = world->getInertialToWorld()*
    (velocity->getInitialCondition() - world->getVelocity()) -
    (world->getAngularVelWorld()^
     init_geodetic_coordinates.getWorldRelVector());

  Vector<double> world_rel_vel_topodetic_init =
    world_to_topodetic_init * world_rel_vel_init;
```

```
  // The track angle is only meaningful when the velocity the east or north
  // velocity
  if ( fabs(world_rel_vel_topodetic_init[1])>1e-8 ||
       fabs(world_rel_vel_topodetic_init[0])>1e-8 )
  {
    // Ground track angle
    double current_track_init = atan2( world_rel_vel_topodetic_init[1] ,
                                       world_rel_vel_topodetic_init[0] );

    double limited_track = fmod(current_track_init, two_pi);

    init_euler_angles.putPsi(
      init_euler_angles.getPsi() + (track_init - limited_track));
  }
  else
  {
    init_euler_angles.putPsi(track_init);
  }

  putInitEulerAngles(init_euler_angles);
}


//=================================================================
// This method defines the behavior of the positional model in RESET mode.
// The positional model is revived if it had been disabled during the previous
// run.  Playback is initialized.
//=================================================================
void PositionalModel::doResetCalc()
{
  // Revive vehicle.
  if ( auto_reset_alive )
  {
    alive = true;
  }

  playback->initialize();

  if ( playback->isPlayback() )
  {
    world_to_topodetic = geodetic_coordinates.calcWorldToTopodetic();

    world_rel_topodetic_orientation = UnitQuaternion(world_to_topodetic);

    world_rel_orientation =~world->getOrientation()* getOrientation();

    topodetic_rel_orientation = ~world_rel_topodetic_orientation *
      world_rel_orientation;

    topodetic_to_body = RotationMatrix(topodetic_rel_orientation);

    euler_angles = EulerAngles(topodetic_to_body);

    // If a pilot exists then update pilot positions
    if ( pilot )
    {
      pilot->calcAuxEqns();
```

```
    }
  }
}

//=================================================================
// This method defines the behavior of the positional model in HOLD mode.
// It takes no action.
//=================================================================
void PositionalModel::doHoldCalc()
{
}

//=================================================================
// This method defines the behavior of the positional model in the external
// force and moment calculation phase of OPERATE mode.  It takes no action.
//=================================================================
void PositionalModel::doOperateCalc()
{
}

//=================================================================
// This method defines the behavior of the positional model during the state
// propagation phase of OPERATE mode.  This section updates playback.  If
// playback is active, the positional model is moved according to the playback
// file. Pilot properties are updated if the pilot exists.
//=================================================================
void PositionalModel::propagateState()
{
  if ( alive )
  {
    // Instruct the playback object to update the positional models states
    // if in playback mode or record states if in record mode
    playback->update();

    // If in playback mode then update the euler angles - the world rel
    // orientation states are based on the new positional model states
    if ( playback->isPlayback() )
    {
      if ( playback->isPlaybackRewinding() )
      {
        body_deriv_world_rel_vel_body = zero_vector;
        world_rel_vel_rate            = 0.0;
        world_rel_accel_topodetic     = zero_vector;

        geo_ref_rel_info_list->updateRelAccelerations();
      }
      else
      {
        world_to_topodetic = geodetic_coordinates.calcWorldToTopodetic();
        world_rel_topodetic_orientation = UnitQuaternion(world_to_topodetic);
        world_rel_orientation =~world->getOrientation()* getOrientation();

        topodetic_rel_orientation = ~world_rel_topodetic_orientation *
          world_rel_orientation;

        topodetic_to_body = RotationMatrix(topodetic_rel_orientation);
```

```
        euler_angles = EulerAngles(topodetic_to_body);

        // If this playback model is configured to calculate all derived
        // states, do it now.
        if ( calc_derived_states )
        {
          calcDerivedStates();
        }
      }

      // If a pilot exists then update pilot positions
      if ( pilot )
      {
        pilot->calcAuxEqns();
      }
    }
  }
}

//===================================================================
// This virtual method defines how a positional model behaves during TRIM
// mode.  The LaSRS++ currently supports two fundamental means of trimming:
// "trim until converged" and "incremental trim".  The first option trims the
// positional model in first frame of TRIM mode; i.e., multiple passes of
// the trim solver are run in a loop until it converges on a solution.  The
// second option operates only one pass of the trim solver each frame while
// in TRIM mode.  The solver will run until the simulation switches out of
// trim mode.  The trim solver does not stop after the convergence criteria
// is reached though the GUI will signal when the convergence has been
// reached.  The boolean argument determines which trim option is exercised;
// "trim until converged" is used if the boolean is true.  The choice of option
// usually depends on operational requirements rather than performance or
// accuracy.  However, the first option is usually faster since each trim pass
// uses less than a full frame.  The second option can be more accurate since
// the trim solver continues to run even after the convergence criteria is
// reached.  The default behavior is no action.
//===================================================================
void PositionalModel::doTrimCalc(bool)
{
}

//===================================================================
// This virtual method defines how a positional model behaves during
// LINEAR_MODEL mode.  The default behavior is no action.
//===================================================================
void PositionalModel::generateLinearModel()
{
}

//===================================================================
// This method sets the ID number for the positional model.
//===================================================================
void PositionalModel::putID(int new_id)
{
  id_number = new_id;
}
```

```
//=========================================================================
// This method determines whether relative geometry information will be
// calculated from the CG of the positional model to the CG of the target.
//=========================================================================
void PositionalModel::setComputeCGToCGRelGeomFlag(bool new_value)
{
  compute_cg_to_cg_rel_geom_flag = new_value;
}

//=========================================================================
// This method determines whether relative geometry information will be
// calculated from the pilot location of the positional model to the CG of the
// target.
//=========================================================================
void PositionalModel::setComputePilotToTargetCGRelGeomFlag(bool new_value)
{
  compute_pilot_to_target_cg_rel_geom_flag = new_value;
}

//=========================================================================
// This method attaches a pilot to the positional model.
//=========================================================================
void PositionalModel::putPilot(Pilot* new_pilot)
{
  pilot = new_pilot;
}

//=========================================================================
// This method calculates and returns the current height above terrain. (feet)
//=========================================================================
double PositionalModel::getHeightAboveTerrain(
  const Vector<double>& body_rel_position) const
{
  Vector<double> p = body_to_topodetic * body_rel_position;

  double z_cg =
    body_to_topodetic[2] * geodetic_coordinates.getWorldRelVector();

  double rz =  z_cg + p[2];

  const Vector<double>& polygon_normal = getTerrainPolygonNormal();

  return rz - z_cg + getHeightAboveTerrain() -
    (p[2] - (p * polygon_normal) * polygon_normal[2]);
}

//=========================================================================
// This method allows the client to set the data relevant to the height
// above terrain.  (Height above terrain has units of feet; new_polygon_normal
// is a unit vector.)
//=========================================================================
void PositionalModel::putHeightAboveTerrainData(
  const double&         new_height_above_terrain,
  const Vector<double>& new_polygon_normal,
  const Timer*          new_height_above_terrain_time_tag)
{
  assume(withinEpsilonOf(new_polygon_normal.mag(), 1.0));
```

```
  height_above_terrain           = new_height_above_terrain;
  terrain_polygon_normal         = new_polygon_normal;
  *height_above_terrain_time_tag = *new_height_above_terrain_time_tag;
  use_sz_for_hat                 = false;
}

//========================================================================
// This method allows the GeoRefRelInfo objects to register themselves onto
// the positional model list of GeoRefRelInfo objects.
//========================================================================
const GeoRefRelInfo* PositionalModel::registerWithGeoRefRelInfoList(
  const GeoRefRelInfoHandle& handle,
  const GeoRefPoint*         geo_ref_point)
{
  return geo_ref_rel_info_list->registerWithList(handle, geo_ref_point);
}

//========================================================================
// This method allows the GeoRefRelInfo objects to unregister themselves from
// the positional model list of GeoRefRelInfo objects.
//========================================================================
void PositionalModel::unregisterFromGeoRefRelInfoList(
  const GeoRefRelInfoHandle& handle,
  const GeoRefRelInfo*       geo_ref_rel_info)
{
  geo_ref_rel_info_list->unregisterFromList(handle, geo_ref_rel_info);
}

//========================================================================
// This method returns the position relative to the primary geographic
// reference point. The vector is represented in the "local" (i.e. body)
// coordinate system of the reference point.  (feet)
//========================================================================
const Vector<double>&
PositionalModel::getPrimaryRefPointRelativePosition() const
{
  return primary_geo_ref_rel_info_handle->getRelArcLengthPositionLocal();
}

//========================================================================
// This method returns the velocity relative to the primary geographic
// reference point. The vector is represented in the "local" (i.e. body)
// coordinate system of the reference point. (feet/second)
//========================================================================
const Vector<double>& PositionalModel::getRefPointRelVel() const
{
  return primary_geo_ref_rel_info_handle->getRelVelocityLocal();
}

//========================================================================
// This method returns a pointer to the primary geographic reference point.
//========================================================================
const GeoRefPoint* PositionalModel::getPrimaryGeoRefPoint() const
{
  return primary_geo_ref_rel_info_handle->getGeoRefPoint();
}
```

```
//***************************************************************************
// DEPRECATED METHODS
//***************************************************************************

void PositionalModel::copyDataToDataRamFile()
{
}

void PositionalModel::copyDataRamFileToPhysicalFile()
{
  if ( getDataRecordingSystem() )
  {
    getDataRecordingSystem()->saveToPhysicalFile();
  }
}
```

Appendix B Preprocessor Variables

AIX_UNIX

This variable indicates that the code only runs on the AIX platform. It also prevents ClearCase from causing code built on other platforms to be winked in on the AIX platform.

BSD_UNIX

This variable indicates that the code only runs on platforms that support the BSD UNIX system calls.

***CLASS_NAME*_TEST**

This variable is used when unit test code is embedded in a class's source file. It isolates the unit test code from the remainder of the class's code and keeps the unit test code inactive during normal use. CLASS_NAME is replaced by the name of the class.

DEBUG_LEVEL_1

This variable exposes debug code designed to identify problems during system testing (aka checkout).

DEBUG_LEVEL_2

This variable exposes debug code designed to capture problems during integration testing (aka batch testing).

DEBUG_LEVEL_3

This variable exposes debug code designed to capture defects during unit testing.

HAVE_LONGLONG

This variable indicates whether the platform supports the "long long" intrinsic type.

INCLUDE_SDB_HARDWARE

This variable determines whether or not support for hardware devices is built into the framework.

IRIX_6_2

This variable is used to indicate that the code is being compiled under IRIX 6.2. It was used to prevent ClearCase from winking in files built on IRIX 6.2 machines to machines running IRIX 6.5.

IRIX_6_5

This variable is used to indicate that the code is being compiled under IRIX 6.5.

_LANGUAGE_C_PLUS_PLUS

This variable activates code that compiles with a standard C++ compiler.

_POSIX_PTHREAD_SEMANTICS

This variable only applies when compiling on the SUN system. It must be defined when compiling code that uses Posix threads on the SUN.

_REENTRANT

This variable activates the re-entrant versions of SGI system calls.

SGI

This variable indicates that the code only runs on the SGI platform. It also prevents ClearCase from causing code built on other platforms to be winked in on the SGI platform.

_SGI_MP_SOURCE

This variable activates a thread safe version of the system calls.

_SGI_REENTRANT_FUNCTIONS

This variable activates a thread safe version of the system calls.

SUN

This variable indicates that the code only runs on the SUN platform. It also prevents ClearCase from causing code built on other platforms to be winked in on the SUN platform.

SYSTEM_V_UNIX

This variable guards code that only runs on platforms that support the System V UNIX system calls.

USE_GUI

This variable determines whether or not GUI libraries are built into the framework.

USE_HEAP_FOR_SCRAMNET

This variable will replace the SCRAMNet+ implementation of the LaSRS++ SCRAMNet software with an implementation that allocates memory from the heap. This is used to debug software, that communicates with SCRAMNet+, on computers that do not have SCRAMNet+.

Appendix C Vehicle Models

The presence of each variable listed below will cause the resulting build to include the specified vehicle type.

```
INCLUDE_B757BASE
INCLUDE_B757AILS
INCLUDE_B757ANOPP
INCLUDE_B757CTAS
INCLUDE_B757CWIN
INCLUDE_B757LAHSO
INCLUDE_B757LVLASO
INCLUDE_B757RIPS
INCLUDE_B757WXAP
INCLUDE_B757SAAP
INCLUDE_B757SLINK2
INCLUDE_BWB
INCLUDE_F15A
INCLUDE_F16A
INCLUDE_F16A_FALLING_LEAF
INCLUDE_F16XL
INCLUDE_F18A
INCLUDE_F18C
INCLUDE_F18ASRA
INCLUDE_F18TV
INCLUDE_F18E
INCLUDE_F18EAWS
INCLUDE_GENERAL_AVIATION
INCLUDE_GENERIC_FIGHTER
INCLUDE_HL20
INCLUDE_HSCT
INCLUDE_PTS
INCLUDE_F18ERPV
INCLUDE_CDU_TEST_VEHICLE
```

Bibliography

[1] B. Eckel. C++ programming style guides. Unix Review, March 1995.

[2] P. B. Gove, PhD., editor. Webster's Third New International Dictionary of the English Language, Unabridged. Merriam-Webster, Inc., Springfield, Massachusetts, 1986.

[3] M. Henricson and E. Nyquist, Programming in C++: Rules and Recommendations. Ellemtel Telecommunications Systems Laboratories, December 1993. Revision C.

[4] J. Lakos. Large Scale C++ Software Design. Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, 1996

[5] C. Lins. A first look at literate programming. Structured Programming, 1989

[6] S. McConnell. Code Complete. Microsoft Press, Redmond, Washington, first edition, 1993.

[7] S. Meyers. Effective C++; 50 Specific Ways to Improve Your Programs and Designs. Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, 1994.

[8] S. Quier, J. Stassi, and T. Anselmo. CERES Software Coding Guidelines. Science Applications International Corporation, November 1994. Release 1.

[9] B. Stroustrup. The C++ Programming Language. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.