

CSM TRD 2010 11 15

# **Community Sensor Model (CSM)**

## **Technical Requirements Document**

### **Appendix C**

#### **Application Program Interface (API)**

15 November 2010

Version 3.0

## REVISION HISTORY

(Re-baseline upon final approval)

Date	Version	Comments
29 September 2010	3.0 For Review	Adds comments and changes from 31 October 2006 through 29 September 2010  Removed MTI TargetModel and MTI SensorModel
15 November 2010	3.0	Adds changes from review
30 November 2010	3.0	Put under configuration management

## **ACKNOWLEDGEMENTS**

Information and text in this document were extracted from copyrighted documents provided by BAE SYSTEMS while under contract to General Dynamics Advanced Information Systems for the Community Sensor Model program.

In addition, this document includes inputs provided by Harris Corporation and Northrop Grumman, also under contract to General Dynamics Advanced Information Systems for the Community Sensor Model program.

This document is a compilation of information and specific content from the Community Sensor Model Government and contractor team.

## NOTES

*Table of Contents*

<b>1</b>	<b>SCOPE .....</b>	<b>1</b>
<b>2</b>	<b>REFERENCE DOCUMENTS.....</b>	<b>2</b>
2.1	<b>Government Documents.....</b>	<b>2</b>
2.1.1	<i>Normative.....</i>	<i>2</i>
2.1.2	<i>Non-Normative.....</i>	<i>3</i>
2.2	<b>Industry Documents/Technical References .....</b>	<b>3</b>
<b>3</b>	<b>TERMS AND DEFINITIONS.....</b>	<b>4</b>
3.1.1	<i>Sensor Exploitation Tool.....</i>	<i>4</i>
3.1.2	<i>Community Sensor Model Definition.....</i>	<i>4</i>
3.1.3	<i>Application Program Interface.....</i>	<i>4</i>
3.1.4	<i>Parameter .....</i>	<i>4</i>
3.1.5	<i>Math Model .....</i>	<i>4</i>
3.1.6	<i>Plug-In.....</i>	<i>5</i>
3.1.7	<i>CSM Library .....</i>	<i>5</i>
3.1.8	<i>CSMPlugin Class .....</i>	<i>5</i>
3.1.9	<i>CSMPlugin Derived Class.....</i>	<i>5</i>
3.1.10	<i>CSMSensorModel Class.....</i>	<i>6</i>
3.1.11	<i>CSMSensorModel Derived Class .....</i>	<i>6</i>
3.1.12	<i>CSM Plugin Name.....</i>	<i>6</i>
3.1.13	<i>CSM Sensor Model Name .....</i>	<i>6</i>
3.1.14	<i>CSM Manufacturer Name .....</i>	<i>6</i>
3.1.15	<i>CSM_ISD Class .....</i>	<i>6</i>
3.1.16	<i>CSM Sensor Model State .....</i>	<i>7</i>
3.1.17	<i>Sensor Mode and Type.....</i>	<i>7</i>
<b>4</b>	<b>GENERAL REQUIREMENTS .....</b>	<b>8</b>
4.1	<b>Guiding Principles.....</b>	<b>8</b>
4.1.1	<i>Application Program Interface.....</i>	<i>8</i>
4.1.2	<i>Community Sensor Model.....</i>	<i>8</i>
4.2	<b>System Architecture.....</b>	<b>8</b>
4.3	<b>Backward Compatibility.....</b>	<b>10</b>
4.3.1	<i>Defining New Base Class .....</i>	<i>10</i>
4.3.2	<i>Deriving New Classes.....</i>	<i>10</i>
4.3.3	<i>Impacts on the SET.....</i>	<i>11</i>
<b>5</b>	<b>SPECIFIC REQUIREMENTS .....</b>	<b>13</b>
5.1	<b>Coordinate Systems.....</b>	<b>13</b>
5.1.1	<i>Image Coordinate System .....</i>	<i>13</i>
5.1.2	<i>Ground Coordinate System .....</i>	<i>14</i>
5.2	<b>Adjustable Sensor Model Parameters.....</b>	<b>14</b>
5.3	<b>Proximate Imaging Locus / Remote Imaging Locus.....</b>	<b>14</b>
5.4	<b>Desired Precision / Achieved Precision.....</b>	<b>15</b>
5.5	<b>Sensor Model State .....</b>	<b>16</b>

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
 Technical Requirements Document (TRD), Version 3.0

<b>5.6</b>	<b>Sensor Model Naming Convention</b> .....	<b>16</b>
<b>5.7</b>	<b>Environment Variables</b> .....	<b>17</b>
<b>5.8</b>	<b>Conventions used in Describing Functions (Methods)</b> .....	<b>17</b>
<b>5.9</b>	<b>CSMPlugin Class</b> .....	<b>18</b>
5.9.1	<i>CSMPlugin Class Object Definition</i> .....	18
5.9.2	<i>Plugin Physical Requirements</i> .....	18
5.9.3	<i>Sensor Model Selection and Construction</i> .....	20
5.9.4	<i>Adding and Removing Plugins</i> .....	21
5.9.5	<i>Explanation of “Plugin Registration”</i> .....	23
5.9.6	<i>Application Developer Responsibilities</i> .....	23
5.9.7	<i>Compiling and Linking with the CSMPlugin Base Classes</i> .....	24
5.9.8	<i>Plugin Developer Responsibilities</i> .....	25
5.9.9	<i>Functionality Required within the Plugin</i> .....	26
<b>5.10</b>	<b>Detailed Plug-in Method Descriptions</b> .....	<b>28</b>
5.10.1	<i>CSMPlugin::getList();</i> .....	29
5.10.2	<i>CSMPlugin::findPlugin();</i> .....	31
5.10.3	<i>CSMPlugin::removePlugin();</i> .....	33
5.10.4	<i>CSMPlugin::canISDBeConvertedToSensorModelState();</i> .....	35
5.10.5	<i>CSMPlugin::canSensorModelBeConstructedFromState();</i> .....	37
5.10.6	<i>CSMPlugin::canSensorModelBeConstructedFromISD();</i> .....	39
5.10.7	<i>CSMPlugin::constructSensorModelFromState();</i> .....	41
5.10.8	<i>CSMPlugin::constructSensorModelFromISD();</i> .....	43
5.10.9	<i>CSMPlugin::convertISDToSensorModelState();</i> .....	45
5.10.10	<i>CSMPlugin::getManufacturer();</i> .....	47
5.10.11	<i>CSMPlugin::getNSensorModels();</i> .....	49
5.10.12	<i>CSMPlugin::getReleaseDate();</i> .....	51
5.10.13	<i>CSMPlugin::getSensorModelName();</i> .....	53
5.10.14	<i>CSMPlugin::getSensorModelNameFromSensorModelState();</i> .....	55
5.10.15	<i>CSMPlugin::getSensorModelVersion();</i> .....	57
5.10.16	<i>CSMPlugin::getPluginName();</i> .....	59
5.10.17	<i>CSMPlugin::getCSMVersion();</i> .....	61
<b>5.11</b>	<b>Image Support Data (ISD)</b> .....	<b>63</b>
5.11.1	<i>NITF 2.0 ISD</i> .....	64
5.11.2	<i>NITF 2.1 ISD</i> .....	65
5.11.3	<i>Filename ISD</i> .....	66
5.11.4	<i>Bytestream ISD</i> .....	67
<b>5.12</b>	<b>Sensor Model Functions</b> .....	<b>68</b>
5.12.1	<i>csmSensorModel::groundToImage(</i> .....	70
5.12.2	<i>csmSensorModel::imageToGround(</i> .....	73
5.12.3	<i>csmSensorModel::imageToProximateImagingLocus(</i> .....	77
5.12.4	<i>csmSensorModel::imageToRemoteImagingLocus(</i> .....	79
5.12.5	<i>csmSensorModel::computeGroundPartials(</i> .....	81
5.12.6	<i>csmSensorModel::computeSensorPartials(</i> .....	83
5.12.7	<i>csmSensorModel::computeAllSensorPartials(</i> .....	86
5.12.8	<i>csmSensorModel::getCurrentParameterCovariance(</i> .....	89
5.12.9	<i>csmSensorModel::setCurrentParameterCovariance(</i> .....	91
5.12.10	<i>csmSensorModel::setOriginalParameterCovariance(</i> .....	93
5.12.11	<i>csmSensorModel::getOriginalParameterCovariance(</i> .....	95
5.12.12	<i>csmSensorModel::getTrajectoryIdentifier(</i> .....	97
5.12.13	<i>csmSensorModel::getReferenceDateAndTime(</i> .....	99
5.12.14	<i>csmSensorModel::getImageTime(</i> .....	102

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
 Technical Requirements Document (TRD), Version 3.0

5.12.15	<i>csmSensorModel::getSensorPosition()</i> .....	104
5.12.16	<i>csmSensorModel::getSensorVelocity()</i> .....	106
5.12.17	<i>csmSensorModel::setCurrentParameterValue()</i> .....	108
5.12.18	<i>csmSensorModel::getCurrentParameterValue()</i> .....	110
5.12.19	<i>csmSensorModel::getParameterName()</i> .....	112
5.12.20	<i>csmSensorModel::getNumParameters()</i> .....	114
5.12.21	<i>csmSensorModel::setOriginalParameterValue()</i> .....	116
5.12.22	<i>csmSensorModel::getOriginalParameterValue()</i> .....	118
5.12.23	<i>csmSensorModel::getOriginalParameterType()</i> .....	120
5.12.24	<i>csmSensorModel::getCurrentParameterType()</i> .....	124
5.12.25	<i>csmSensorModel::getPedigree()</i> .....	128
5.12.26	<i>csmSensorModel::getImageIdentifier()</i> .....	131
5.12.27	<i>csmSensorModel::setImageIdentifier()</i> .....	133
5.12.28	<i>csmSensorModel::getSensorIdentifier()</i> .....	135
5.12.29	<i>csmSensorModel::getPlatformIdentifier()</i> .....	137
5.12.30	<i>csmSensorModel::getImageSize()</i> .....	139
5.12.31	<i>csmSensorModel::getSensorModelState()</i> .....	141
5.12.32	<i>csmSensorModel::getValidHeightRange()</i> .....	143
5.12.33	<i>csmSensorModel::getValidImageRange()</i> .....	145
5.12.34	<i>csmSensorModel::getIlluminationDirection()</i> .....	147
5.12.35	<i>csmSensorModel::getNumGeometricCorrectionSwitches()</i> .....	149
5.12.36	<i>csmSensorModel::getGeometricCorrectionName()</i> .....	151
5.12.37	<i>csmSensorModel::setCurrentGeometricCorrectionSwitch()</i> .....	153
5.12.38	<i>csmSensorModel::getCurrentGeometricCorrectionSwitch()</i> .....	155
5.12.39	<i>csmSensorModel::getReferencePoint()</i> .....	157
5.12.40	<i>csmSensorModel::setReferencePoint()</i> .....	159
5.12.41	<i>csmSensorModel::getSensorModelName()</i> .....	161
5.12.42	<i>csmSensorModel::setOriginalParameterType()</i> .....	163
5.12.43	<i>csmSensorModel::setCurrentParameterType()</i> .....	165
5.12.44	<i>Covariance Model</i> .....	167
5.12.45	<i>csmSensorModel::getUnmodeledError()</i> .....	174
5.12.46	<i>csmSensorModel::getUnmodeledCrossCovariance()</i> .....	176
5.12.47	<i>csmSensorModel::getCollectionIdentifier()</i> .....	178
5.12.48	<i>csmSensorModel::isParameterShareable()</i> .....	180
5.12.49	<i>csmSensorModel::getParameterSharingCriteria()</i> .....	182
5.12.50	<i>CSMSensorModel::getVersion()</i> .....	185
5.12.51	<i>csmSensorModel::getSensorTypeAndMode()</i> .....	186
<b>5.13</b>	<b>Error Control</b> .....	<b>190</b>
<b>5.14</b>	<b>Memory Management</b> .....	<b>194</b>
<b>6</b>	<b>APPENDIX A HEADER FILES</b> .....	<b>1</b>
<b>6.1</b>	<b>CSMPlugin.h</b> .....	<b>1</b>
<b>6.2</b>	<b>CSMISD</b> .....	<b>6</b>
6.2.1	<i>CSMImageSupportData.h</i> .....	6
6.2.2	<i>CSMISDNITF21.h</i> .....	7
6.2.3	<i>CSMISDNITF20.h</i> .....	10
6.2.4	<i>CSMISDByteStream.h</i> .....	13
6.2.5	<i>CSMISDFilename.h</i> .....	14
<b>6.3</b>	<b>CSMSensorModel.h</b> .....	<b>15</b>
<b>6.4</b>	<b>CSM Warnings and Errors</b> .....	<b>30</b>
6.4.1	<i>CSMWarning.h</i> .....	30

6.4.2	<i>CSMError.h</i> .....	32
6.5	<b>CSMMisc.h</b> .....	34
6.6	<b>CSMParameterSharing.h</b> .....	37
<b>7</b>	<b>APPENDIX B ADDITIONAL EXPLANATION OF EXPORT SYMBOLS</b> .....	<b>1</b>
7.1	<b>Introduction</b> .....	1
7.2	<b>Discussion</b> .....	1
<b>8</b>	<b>APPENDIX C COMPILING</b> .....	<b>1</b>
8.1	<b>Sun Solaris Forte or Workshop compiler:</b> .....	1
8.2	<b>GCC compiler:</b> .....	6
8.3	<b>Instructions for compiling and Testing in Microsoft Windows:</b> .....	11
8.3.1	<i>The Application Project vts</i> .....	12
8.3.2	<i>The CSMAPI Project</i> .....	12
8.3.3	<i>The Plugin Projects</i> .....	12
8.3.4	<i>Frequently Asked Questions</i> .....	12
<b>9</b>	<b>APPENDIX D EXAMPLE CPP FILES</b> .....	<b>15</b>
9.1	<b>CSMPlugin.cpp</b> .....	15

## LIST OF FIGURES

Figure 1 - CSM Context Diagram .....	9
Figure 2 - Image Coordinate System .....	13
Figure 3 - Sensor Model Parameter Components.....	14
Figure 4 - Imaging locus for Optical Imagery .....	15

## LIST OF TABLES

Table 1 - Applicable Government Documents.....	2
Table 2 - Reference Government Documents.....	3
Table 3 - Applicable Industrial Documents/Technical References .....	3
Table 4 - Sensor Model Functions .....	68
Table 5 - Date and Time Format .....	100
Table 6 - Pedigree Convention Components .....	129

# 1 SCOPE

This document supplements the functional requirements set forth in the Community Sensor Model (CSM) Technical Requirements Document (TRD) and establishes the requirements placed on the sensor model elements to interface with applications that use the photogrammetric operations (math libraries) contained in the sensor model. Both documents collectively establish the requirements allocated to the sensor model.

This document defines the Application Program Interface (API) between a plug-in Community Sensor Model (CSM) and the host Sensor Exploitation Tool (SET). It outlines the function calls that are available to the SET and describes how these facilities are used. Additionally, this document specifies a minimum set of capabilities that the CSM must provide for SET use.

While this interface supports a broad array of single image operations, only minimal support to multi-image operations is available. The Community Sensor Model API does not provide management of multi-image information, such as joint image error covariance. Derived images produced through such operations as chipping, warping, magnification, and mosaicing are supported by the Community Sensor Model API, but only in the sense that the original image coordinate frame is referenced and relevant support data is applied.

## 2 REFERENCE DOCUMENTS

### 2.1 Government Documents

#### 2.1.1 Normative

**Table 1 - Applicable Government Documents**

Document No.	Title
MIL-STD-2500A	National Imagery Transmission Format Version 2.0 for the National Imagery Transmission Format Standard NOTE: Version 2.0 is for legacy data and has been updated to Version 2.1.
MIL-STD-2500C	National Imagery Transmission Format Version 2.1 for the National Imagery Transmission Format Standard
STDI-0001	National Support Data Extensions (SDE) Version 1.3 for the National Imagery Transmission Format (NITF)
STDI-0002	Compendium of Controlled Extensions (CE) for the National Imagery Transmission Format (NITF) Version 3.0
TR 8350.2	NIMA Technical Report 8350.2, DoD World Geodetic System 1984 – Its Definition and Relationship with Local Geodetic Systems
NIST Special Publication 811	NIST Guide for the Use of the International System of Units (SI)
DCID 6/3 Manual	Director Central Intelligence Directive (DCID) 6/3 - Protecting Sensitive Compartmented Information Within Information Systems
JDCSISS	Joint DODIIS / Cryptologic SCI Information Systems Security Standards
	Community Sensor Model Technical Requirements Document
	Sensor Model Glossary

**2.1.2 Non-Normative**

**Table 2 - Reference Government Documents**

<b>Document No.</b>	<b>Title</b>
N0105-98	NITFS Standards Compliance and Interoperability Certification Test and Evaluation Program Plan
DoDI 5000.61	DoD Modeling and Simulation Verification, Validation, and Accreditation
NUG-B	USIGS Glossary Revision B
	Applicable Platform Developer Documents ORDs
	Applicable Application Developer Documents APIs

**2.2 Industry Documents/Technical References**

**Table 3 - Applicable Industrial Documents/Technical References**

<b>Document No.</b>	<b>Title</b>
ANSI IEEE 754-1985	Floating Point Arithmetic
ISO 8601:2000	ISO 8601:2000 (international standard for date representation)
ISO/IEC 14882:1998	ISO/IEC 14882:1998: Programming Languages - C++
ISO/IEC 14882:2003(E)	ISO/IEC 14882:2003(E): Programming Languages - C++

## 3 TERMS AND DEFINITIONS

### 3.1.1 Sensor Exploitation Tool

A Sensor Exploitation Tool (SET) is any software with the capabilities to make use of imagery resulting from a sensor. Examples of operations commonly performed by a SET include mensuration, feature projection, extraction, registration, and uncertainty propagation.

### 3.1.2 Community Sensor Model Definition

A Community Sensor Model (CSM) is a plug-in software library that provides support for photogrammetric operations on imagery produced by a particular sensor. Underlying a CSM is a mathematical model defining a coordinate transformation from that sensor's image space (2-dimensional) to ground space (3-dimensional). Through the phenomenology, physics, and geometry of the image formation process, an imaging ray from the sensor can be mapped onto the ground through a set of rigorous equations. It is the responsibility of the CSM to perform this mapping through the abstract interface provided by the application program interface. **Please note the term "Tactical Sensor Model, (TSM)" is used in this document and is synonymous with "Community Sensor Model, (CSM)".** TSM was the name of the predecessor USAF program.

### 3.1.3 Application Program Interface

The Application Program Interface (API) provides a generalized, abstract interface between a SET and a Community Sensor Model plug-in. This interface allows a SET to utilize the functionality provided by a Community Sensor Model plug-in with minimal knowledge about the image formation details of the sensor. The API is designed to support a variety of military imaging sensors, and although this interface definition can support classified sensor models, the interface itself is generic enough to remain unclassified.

Through the API, the sensor exploitation tool is provided a complete set of single image photogrammetric operations in a sensor independent manner.

### 3.1.4 Parameter

The parameters referred to in this document are elements of the CSM math models, which can be adjusted by the user.

### 3.1.5 Math Model

The CSM math models are the equations that translate the sensor system data into information supporting imagery exploitation. This includes the equations required to accurately map the ground coordinates of the image, as well as the support data depicting the uncertainty in the calculations based on the system elements as well as their combined effects.

### **3.1.6 Plug-In**

The term Plug-in refers to an independent software program with defined interfaces which will operate with other applications but will not interfere with the operation of other co-resident applications.

### **3.1.7 CSM Library**

A CSM Library is a runtime linked library file conformant to the physical description in section 5.9 that contains all functions and interface support specified in this API for CSM plugins. This library shall include the implementation of concrete classes that derive from the CSMPlugin and CSMSensorModel base classes, defined below, as well as a static instance of the CSMPlugin-derived class.

Within the context of this document, the CSM Library refers to the single physical file containing the compiled plugin object code, whereas the CSMPlugin and CSMSensorModel classes refer to the set of functions contained within the library.

Note that the phrase “CSM plugin” is also used within this document to refer to the combination of the physical entity of the CSM Library file together with the functions provided by the library. Hence, CSM plugin is used almost synonymously with CSM Library.

### **3.1.8 CSMPlugin Class**

The CSMPlugin class is the base class for all CSM plugin derived classes that are associated with real tactical sensors. It consists of a set of pure virtual methods that provide functional support for managing CSM plugins, selecting among the available CSMPlugin derived classes for any particular image, and for constructing a CSMSensorModel object that can be used to perform sensor modeling operations for a given tactical image. Pure virtual methods will need to be implemented by classes that derive from this class.

This class also contains non-virtual methods to allow the application to manage the static list of plugins. These list methods should not be used directly by the plugins.

This class is labeled as a plugin since it provides mechanisms that allow classes that derive from it to be added easily to an installation without the need to recompile the base application. Hence, the derived types are “plugins” to the base class. The CSMPlugin class is compiled and linked into a separate runtime library, and is needed by the application as well as the plugins at runtime. Classes that derive from it and that are included in a CSM Library are “plugins” since they can be added at run time without the need to recompile source code. Section 5.9 and its subparagraphs provide functional detail and requirements for the CSMPlugin base and derived classes.

### **3.1.9 CSMPlugin Derived Class**

Within this specification, classes that derive from the CSMPlugin class, that are associated with a real tactical sensor, that implement the pure virtual methods defined within the CSMPlugin class, and that meet other requirements in this specification are

called CSMPPlugin Derived Classes (or also CSMPPlugin derived types). In addition to implementing all pure virtual methods defined in the base class, there are several additional requirements that must be met by each CSMPPlugin derived class in order for it to properly plug into the base class. These additional requirements are specified in paragraph 5.9 and its subparagraphs.

### **3.1.10 CSMSensorModel Class**

The CSMSensorModel class is the abstract base class for all Community Sensor Models that are governed by this specification. This class will get compiled and linked with the application at compile time, whereas classes that derive from it and that are part of a CSM library can be added at run time without the need for any recompiling of code. Section 5.12 and its subparagraphs provide functional detail and requirements for the CSMSensorModel base and derived classes.

### **3.1.11 CSMSensorModel Derived Class**

Within this specification, classes that derive from the CSMSensorModel class, that are associated with a real tactical sensor, that implement the pure virtual methods defined within the CSMSensorModel base class and that meet all other requirements for CSMs are called CSMSensorModel derived classes (or derived types).

### **3.1.12 CSM Plugin Name**

The CSM plugin name is a name string associated with a real CSMPPlugin derived class that uniquely identifies the CSMPPlugin from all other CSMPPlugin derived classes. See paragraph 5.9.10.1 for requirements associated with this string.

### **3.1.13 CSM Sensor Model Name**

A CSMPPlugin may be capable of constructing multiple types of sensor models, each of which is a CSMSensorModel derived class. The CSM Sensor Model Name is a name string that uniquely identifies any given CSMSensorModel derived class regardless of which CSM plugin creates it. See paragraphs 5.6 and 5.9.10.2 for requirements associated with this string.

### **3.1.14 CSM Manufacturer Name**

The CSM manufacturer name is a name string that uniquely identifies the manufacturer of a CSM plugin. See paragraph 5.9.10.3 for requirements associated with this string.

### **3.1.15 CSM\_ISD Class**

The CSM\_ISD class is an externally defined class that provides a means to pass image support data across the CSM API interface within a sensor independent structure. The CSM\_ISD class can be used for sensor model selection and/or construction. The CSM\_ISD class is used when processing an image from the native image file, or when sensor model state data for the image is not available. The CSM\_ISD class is the base class for derived types of ISD classes, where derived types might be associated with

particular image file formats (such as NITF 2.0, or NITF 2.1). The CSM\_ISD may contain more information than is required to create a given sensor model. It may also not contain all of the information required by a given sensor model. The application software is responsible for creating CSM\_ISD objects. See paragraph 5.11.

### **3.1.16 CSM Sensor Model State**

CSM sensor model state defines a single image geometry in a single form, specific to a particular image and sensor model type. The CSM sensor model state contains all information required to create a specific sensor model for a given image. The sensor model state includes no additional information. Sensor model state data is formatted as a null-terminated ASCII character `std::string` where the first set of characters in the `std::string` (up to and including the first newline character) shall be the sensor model name, where the sensor model name is as defined above. See paragraph 5.5.

### **3.1.17 Sensor Mode and Type**

The `CSMSensorTypeAndMode` class is returned by the `getSensorTypeAndMode` method of the `CSMSensorModel` class. This method can be used by a SET to determine the sensor type (e.g. electro-optical, SAR) and mode (e.g. frame, pushbroom, whiskbroom) that is modeled by the sensor model. For convenience, a set of common mode/type combinations such as EO/frame, SAR/spot are defined in the `CSMSensorModel.h` header file.

Section 5.12.51 describes this method in detail.

## 4 GENERAL REQUIREMENTS

### 4.1 Guiding Principles

This section outlines the guiding principles underlying the application program interface to the Community Sensor Models that will be used by the sensor exploitation software. The characteristics and interactions of these two components can be found in the Technical Requirements Document (TRD) which governs the CSM implementation. The guiding principles behind each item are collected here for convenience.

#### 4.1.1 Application Program Interface

The API is designed to provide a generic interface between a SET and a CSM plug-in. The API should be generic enough such that any sensor providing a 2-dimensional image representation of its environment and the necessary support data can be exploited via the functions defined by the API.

The API is generalized in the sense that it is common across the various types of sensor models supported and does not require the SET to have detailed knowledge of image formation process for a specific sensor. In addition, the functions defined for the sensor models allow them to be plugged into the SET such that sensor models can be added or removed from a particular configuration without modification of the host application code.

API design uses object oriented concepts.

The API is sensor (i.e. SAR vs. EO), operating system (i.e. Windows vs. UNIX) and hardware (i.e. PC vs. Sun) independent.

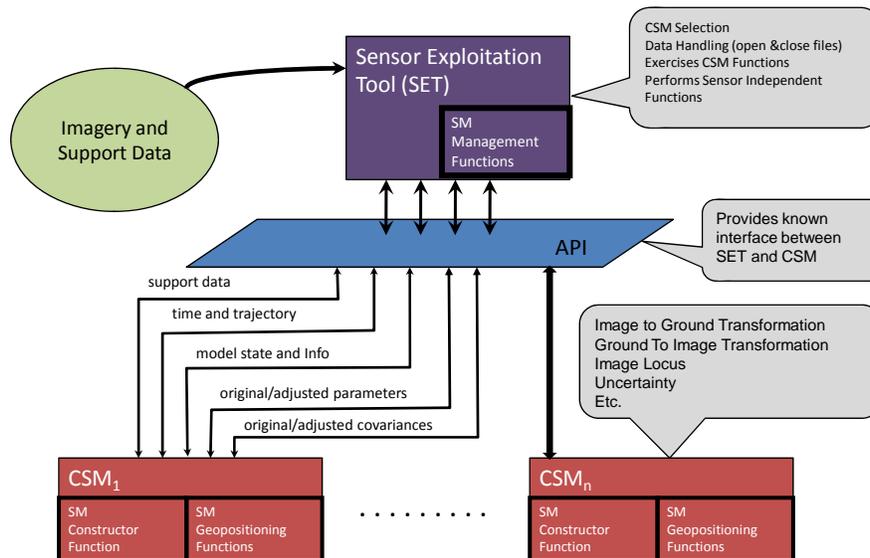
#### 4.1.2 Community Sensor Model

Initially, the SETs may require modifications to access the functionality of the CSMs in a sensor independent manner, but will not require additional changes as more CSMs are produced and released.

The CSM is responsible for extracting any necessary parameters from the sensor image product.

### 4.2 System Architecture

Figure 1 displays the perspective of the CSM in its operational environment, specifically, showing its relationship to the SET via the API. The figure shows example data that may be passed between a given CSM and the SET and example functions. The figure is not all encompassing.



**Figure 1 - CSM Context Diagram**

Below the API, the CSM has two distinct sets of functions. The CSM constructor functions include those functions required to instantiate the sensor model and prepare it to respond to inputs from the SET. While the CSM geopositioning functions perform the image to ground and ground to image transformations. Associated functions support these transformations and provide additional information used by the SET to perform its exploitation functions. The CSM provides a list of specific parameters, their values and uncertainties (variances and covariances) and a means for adjusting these parameters to obtain more accurate solutions. The CSM also integrates information regarding the time of collection and the trajectory.

Above the API, the SET understands the local environment and performs data handling functions (i.e. opens/closes files, opens/closes data streams, etc.). The SET uses the CSM constructor functions to create the required CSM. And the SET exercises the transformation functions between the ground and image spaces to exploit the imagery. The SET can also use this information to perform other sensor independent functions such as mensuration, registration, feature extraction, etc. Using the associated CSM functions, the SET adjusts selected CSM parameters to obtain a more accurate solution.

Furthermore, the SET performs sensor model management functions as required. The SET also selects the appropriate CSM if more than one model is available for the imagery data in use.

Note that there may be multiple CSMs for the same sensor image; therefore the CSM must provide sufficient information to allow the SET or its user to select the most appropriate model.

### 4.3 Backward Compatibility

The CSM API has been designed to allow for some backwards compatibility between versions. Versions of the API that are backwards compatible will be noted in the associated CSM API document.

The method designated for use for backwards compatibility is to add new classes to the API without altering existing class definitions. This is done by either defining a completely new base, or a new derived class of an existing base class. Examples of each alternative are given below.

#### 4.3.1 Defining New Base Class

Adding support for a new model type is accomplished by defining a new plug-in and model class for the CSM API. For example, in order to support GMTI, a new base class (e.g. CSMTargetModel) would be created for the new target model, along with a new corresponding plug-in class (e.g. CSMTargetPlugin).

#### 4.3.2 Deriving New Classes

Adding a new capability to an existing model would be done with derived classes. For example, adding a new method, `getValidImageRange`, to the `CSMSensorModel` class might be done as follows:

```
// Derived new class from version 3.0 of CSMSensorModel

class CSMSensorModel31 : public CSMSensorModel
{
    public:
    CSMSensorModel31() { _csmVersion = 31; }
    virtual ~CSMSensorModel31() {}

    // New function
    virtual CSMWarning* getValidImageRange (
    double&          minRow, double&          maxRow,
    double&          minCol, double&          maxCol)
    throw (CSMError) = 0;
}
```

It is also possible to overload methods of the existing `CSMSensorModel` class by using derived classes. A new declaration of the method name would be added to the derived

class. It is assumed that this declaration would include a different parameter list from that of the base class method to allow overloading.

Methods from the base class with the same name need to be duplicated in the derived class. For example, in order to overload the `imageToGround` method in class `CSMSensorModel` shown here:

```
class CSMSensorModel
{
public:
virtual CSMWarning* imageToGround(argA, argB) = 0;
}
```

The derived class needs to duplicate all base class declarations with the same method name, in addition to the updated method. For example,

```
class CSMSensorModel31 : public CSMSensorModel
{
public:
virtual CSMWarning* imageToGround(argA, argB) = 0;
virtual CSMWarning* imageToGround(argB, argC, argD) = 0;
}
```

### 4.3.3 Impacts on the SET

In order for a SET to make use of a new method, the SET must first check the model's version to make sure the method is available. The SET has two options to verify the version of a model:

1. Call the `getVersion` method to obtain the version number

The following example code shows how a SET would determine the version of a sensor model before making a particular CSM API method call:

```
int csmVersion = 0;
CSMSensorModel* csmModel = NULL;

// Assuming csmModel is created here
csmModel->imageToGround (argA, argB);           // Calling CSM 3.0 API
csmModel->getVersion ( csmVersion );           // Determine CSM version of
                                                // model
if ( csmVersion >= 31 )                       // For CSM 3.1 and later
{
```

## NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
// Cast to the appropriate version of CSMSensorModel

CSMSensorModel31* csmModel31 =
    dynamic_cast<CSMSensorModel31*>(csmModel);

// Calling CSM 3.1 API
csmModel31->imageToGround (argB, argC, argD);
}
else
{
    // Method not supported
}
```

2. Alternately, the SET could use RTTI (Run-Time Type Identification) `dynamic_cast` instead of calling `getVersion`.

The following sample code shows how a SET would cast to the appropriate `CSMSensorModel` version, and determine the correct version of a sensor model before making a CSM API method call:

```
CSMSensorModel* csmModel = NULL;

// Assuming csmModel is created here
csmModel->imageToGround (argA, argB); // Calling CSM 3.0 API

// Cast to the appropriate version of CSMSensorModel
CSMSensorModel31* csmModel31 =
dynamic_cast<CSMSensorModel31*>(csmModel);
if (csmModel31 != NULL) // Casting to correct version?
{
    // Calling CSM 3.1 API
    csmModel31->imageToGround (argB, argC, argD);
}
else
{
    // Method not supported
}
```

## 5 SPECIFIC REQUIREMENTS

### 5.1 Coordinate Systems

#### 5.1.1 Image Coordinate System

Any point in an image can be described by two coordinates, the line (or row) and the sample (or column). The origin of the coordinate system is taken to be at the upper left corner of the upper left pixel. The line coordinate is positive in the downward direction on the image, and the sample coordinate is positive to the right. The pixel at the origin will have the coordinates of (0,0).

Image coordinates are measured in units of pixels. Only coordinates referenced to the full image resolution are used in the Sensor Model interface.

The image coordinates at the center of any pixel will have a fractional part of 0.5. (See Figure 2)

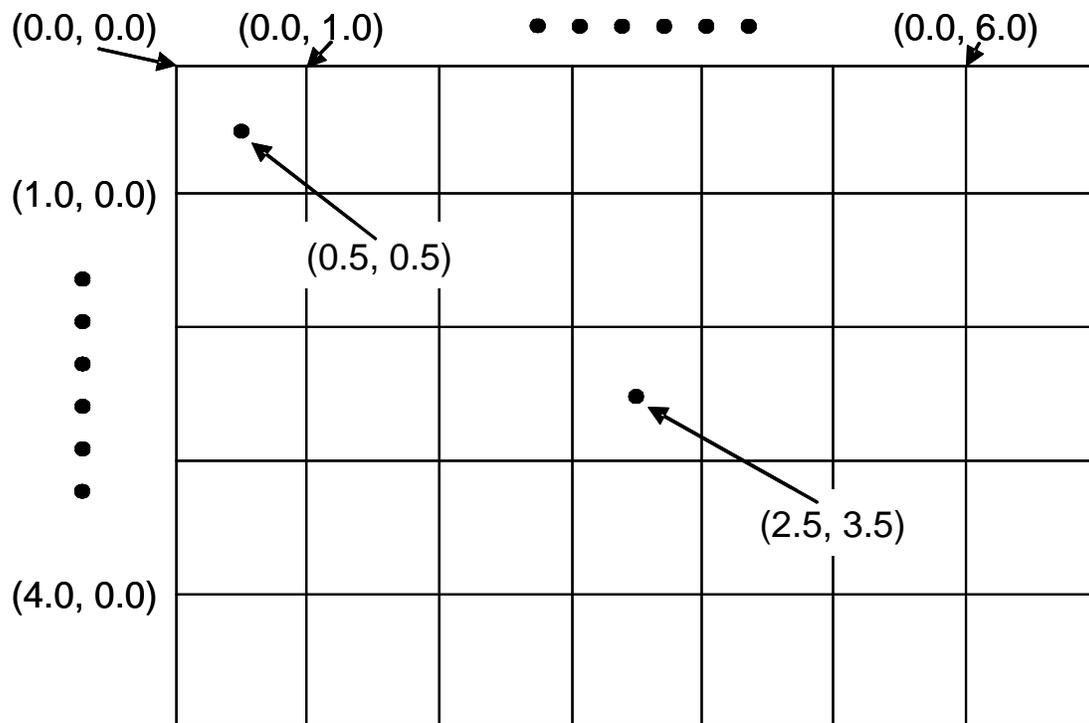


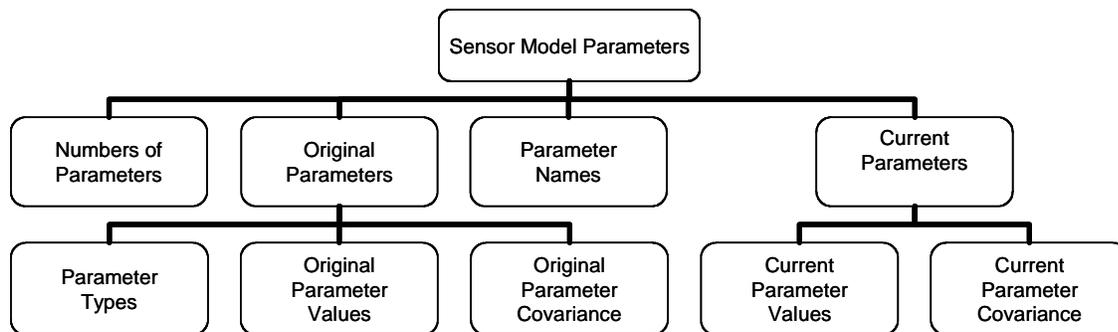
Figure 2 - Image Coordinate System

### 5.1.2 Ground Coordinate System

The API shall use a rectangular coordinate system referenced to the Earth Centered Earth Fixed (ECEF) coordinate frame referenced to WGS-84. All lengths are measured in meters.

## 5.2 Adjustable Sensor Model Parameters

Each sensor model contains two versions of the adjustable parameter information (See Figure 3). The original set of these parameters contains the values that characterize the original image acquisition. The current set of sensor model parameters contains the values that are used to perform all photogrammetric computations and may be altered by the application.



**Figure 3 - Sensor Model Parameter Components**

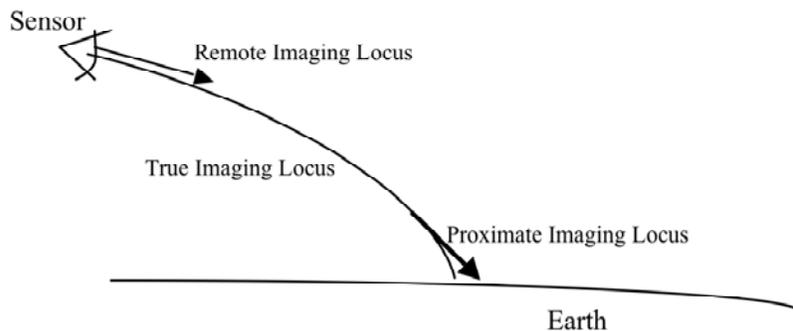
Sensor model parameters refer to those parameters that need to be set in order to run the image-to-ground and ground-to-image functionality of the sensor model. Two values, original and current, are stored (maintained in local memory) for each sensor model parameter. The current value can be updated to the best estimate of the parameter value. The original value is usually derived from support data, in which case it cannot be changed. A covariance matrix is associated with the original parameter values. A second covariance matrix is associated with the current parameter values. Each of these two covariance matrices contains entries for every pair of parameters. The current covariance matrix elements are sufficient to compute rigorous error estimates for the parameters.

### 5.3 Proximate Imaging Locus / Remote Imaging Locus

There is a one-to-one mapping going from ground space to image space. However, going from image to ground space the mapping is one to many. Each pixel in image space corresponds to a line of sight that traverses a set of ground points. That set of ground points is termed the imaging locus. The imaging locus is useful in a variety of

photogrammetric operations. For example, an image-to-ground calculation can be made by finding the intersection of the imaging locus with a ground surface.

In general, the imaging locus is a complicated path due to the effects of physical phenomena. Therefore, the sensor models return an approximation to the imaging locus termed the proximate imaging locus. The proximate imaging locus consists of a point on the locus together with the direction of the locus at that point. In effect, the proximate imaging locus at a given ground point is the line tangent to the imaging locus at that point. Any arbitrary imaging locus can then be approximated using multiple proximate imaging loci.



**Figure 4 - Imaging locus for Optical Imagery**

For optical imagery, the imaging locus follows the line of sight from the camera to the ground. Figure 4 depicts the imaging locus for optical imagery. The effects of refraction on the line of sight are greatly exaggerated in this picture. For SAR imagery, the imaging locus follows the curve of intersection of a constant range sphere with the Doppler cone and also includes the effects of refraction.

In cases where a ground point is not available to compute the proximate imaging locus, a remote imaging locus can be obtained. The remote imaging locus is the line of sight approximated by a straight line. The complicated effects of physical phenomena are ignored. This gives a good approximation of the actual imaging locus and can be used to obtain ground points at which a proximate imaging locus can be computed.

## **5.4 Desired Precision / Achieved Precision**

Desired precision and achieved precision are measures of the precision of implicit (iterative) algorithms. Explicit algorithms find a precise answer for a given input, while implicit algorithms often require some iterative technique in order to produce the desired output. The resulting precision is a trade-off with speed: a more precise result requires additional iterations.

Desired precision is an input to functions that request the result to be within a certain range of the exact value, whether it is in pixels, meters, or some other measure. It has also been referred to as a convergence tolerance.

Achieved precision is an output of functions that accept desired precision as their input. It is the precision that is actually obtained by the algorithm. The intent is to obtain a value equal to or less than the desired precision value.

It is important not to confuse precision with accuracy. "Accuracy" is defined as the degree to which a value may be in error. "Precision" is defined as the resolution or granularity (i.e., number of decimal places, for instance) in which the value is reported. Thus, if a value is accurate to  $\pm 1$  unit of measure, precision to multiple decimal places may be of limited usefulness. Accuracy (i.e., error uncertainty) is represented using covariance matrices for certain CSM API methods.

## 5.5 Sensor Model State

The state of a sensor model is the set of data containing the information required to complete the sensor modeling functions. The sensor model state data must be sufficient for defining a complete and operable sensor model. This includes all unadjustable parameters (such as mode), adjustable parameters (both original and current), and all covariance (both original and current). See paragraph 5.12.31 for more detailed description.

A CSM can only produce a sensor model from a saved state if that saved state was produced by it.

The content of the data is not specified by this document and is the responsibility of the creator of the sensor model. The transfer of the state data from the SET to the CSM and from the CSM to the SET is specified as a byte stream.

## 5.6 Sensor Model Naming Convention

The name of a CSM library must uniquely identify it. The Sensor Model files shall be named by concatenating the platform name or abbreviation (e.g. GLOBAL\_HAWK\_RQ4A), the specific sensing device identifier (e.g. SAR), the development contractor name (e.g., HARRIS), the version release number (see 5.10.15 CSMPugin::getSensorModelVersion), the computing platform operating system (e.g. solaris7), a CSM version number followed by a decimal point and the appropriate extension (e.g. dll, so). Each of the concatenated fields before the decimal point shall be separated by an underscore. Only the sensor name, the decimal point separator and the extension are required, but enough optional fields will be used to completely identify the CSM. Examples of file names:

GlobalHawk\_SAR\_Harris\_2\_solaris9gcc\_csm30.so

GlobalHawk\_SAR\_Harris\_2\_solaris9sun\_csm30.so

GlobalHawk\_SAR\_Harris\_2\_win2k\_csm30.dll

## 5.7 Environment Variables

The CSM shall identify and document environment variables within the installation instructions.

## 5.8 Conventions used in Describing Functions (Methods)

Each public function of the top-level interface to the Sensor Models is individually described in the next sections. For each function, the following items are described:

NAME	The full function name including the class it is found in. Function names begin with lower case. Constants are denoted with all capitalized characters.
SYNOPSIS	The function prototype as it would be seen in the C++ code. Code fragments are displayed in a courier font.
DESCRIPTION	The function and each function parameter are described.
INPUTS	The inputs to the function, if any, are described.
OUTPUTS	The function's outputs upon successful completion, if any, are described.
ERRORS & WARNINGS	Any known conditions that cause an error to be raised or a warning to be issued are identified.
NOTES	Other text used to amplify the understanding of this function
SEE ALSO	Related functions are listed.

## 5.9 CSMPugin Class

The `CSMPugin` class encapsulates all of the functionality contained within a single CSM plugin shared object. These capabilities include providing applications with the ability to describe plugins, select sensor models, process image support data, and create sensor models. The plugin descriptor functions provide applications with information that is associated with the entire associated plugin. Sensor Model availability functions identify the sensor models that are supported by the associated plugin. Sensor Model descriptor functions provide characteristics that are associated with an entire given sensor model type. For sensor models, image support data conversions take given image support data and provide sensor model state results. Sensor Model constructors produce sensor models of the given sensor model type. The resultant sensor models support a wide variety of capabilities required to exploit the support data provided, e.g. sensor models support imaging geometry associated with a particular image.

The `CSMPugin` class is an abstract base class. `CSMPugin` uses sensor model state strings. The class is dependent upon the following externally defined classes: `CSMError`, `CSMWarning`, and `CSM_ISD`, and a model definition class (`CSMSensorModel`) for each model type.

### 5.9.1 CSMPugin Class Object Definition

The code for the `CSMPugin` class is listed in section 6.1.

### 5.9.2 Plugin Physical Requirements

CSM plugins are always provided in a form that can be distributed to potential users and installed in applications that require sensor models. Each plugin takes the form of a runtime linkable library that can be incorporated and by applications at runtime.

#### 5.9.2.1 General Format

Each plugin is deployed as a runtime link library in a single file. The format of that file is dependent upon the operational environment where the plugin is to be used. The following table identifies both the physical formats and the filename suffix associated with the possible operational environments.

Environment	Physical Format	Filename Suffix (Extension)
Sun (SPARC) / Solaris	ELF 32-bit MSB dynamic link library, for SPARC, version 1	.so
	ELF 64-bit MSB dynamic link library, for SPARC, version 1	
Silicon Graphics (MIPS) / IRIX	ELF N32 MSB mips-3 dynamic link library, for MIPS, version 1	.so
	ELF N64 MSB mips-3 dynamic link library, for MIPS, version 1	
PC / Linux	ELF 32-bit LSB shared object, Intel 80386, version 1	.so
	ELF 64-bit LSB shared object, Intel 80386, version 1	
PC / Windows	32-bit segmented executable dynamic link library (DLL) for Release and with Multi-threading (/MD)	.DLL
	64-bit segmented executable dynamic link library (DLL) for Release and with Multi-threading (/MD)1	

Although a plugin may be stored on any system, it will operate only in the prescribed environment.

### 5.9.2.2 Internal Structure

Each plugin contains the elements necessary to implement one or more sensor models. This includes CSM plugin interface software, sensor model construction support, and sensor model operations software.

### 5.9.2.3 External Dependencies

Because each plugin contains an implementation of a derived `csmPlugin` class, all plugins are dependent upon the identical implementation of the `csmPlugin` base class (as specified in section 5.9.1 and 5.14). The `csmPlugin` base class implementation is not provided in a CSM plugin, but must be provided by a plugin-using application. As a result, compatibility among plugins is regulated by the `csmPlugin` base class implementation.

Generally, runtime link libraries are capable of identifying other runtime link libraries that are needed for proper operation. Because the names and capabilities that are potentially provided by these plugin-external libraries cannot be effectively controlled,

each plugin may not contain any dependencies upon any software not already included in the plugin or the plugin-using application. As a result, the external software available to a plugin for use must either come as a direct result of the CSM plugin interface or the compiler environment (see section Appendix C Compiling) that is common to both the application and the plugins.

Plugin operating environments are expected to provide a variety of features that could cause problems for proper plugin operation. The use of these features is prohibited. Problem features include file access (particularly with shared files), environment variables, and various operating system controls.

It is recommended that a CSM plugin be self contained and not dependent upon other objects (that cannot be controlled) — compatibility and proper operation are enhanced.

### 5.9.3 Sensor Model Selection and Construction

#### 5.9.3.1 Sensor Model Construction

A sensor model is generally constructed for a particular image or set of support data. The information necessary to perform that construction is contained within the image support data associated with an image. This support data can either be in the form of an Image Support Data (ISD) structure, or it can be in string form known as sensor model state data. Reference the externally defined class `csm_ISD` for a definition and description of the ISD structure. The sensor model state defines a single image geometry in a single form specific to a particular image. Note that image support data is suitable for storage or transfer as needed, while the sensor model produced from this data is not.

The overall sensor model construction process will involve two types of activities: 1) sensor model *selection* and 2) sensor model *construction*. These activities could be done in series or they may overlap. The details of sensor model *selection* are discussed further in the following section, while the remainder of this section will be devoted to sensor model *construction*.

There are two ways to construct a sensor model. First, an application may construct a sensor model using image support data that has been prepared by the application to be in the form specified by the `csm_ISD` class. In this case, the application calls the `CSMPlugin` method `constructSensorModelFromISD`, providing a pointer to the `csm_ISD` structure as an input. Prior to making this call, the application should first determine which plugin can produce a model from the given ISD. The section “Sensor Model Selection” addresses how to determine this.

The second way to create a sensor model, specific to sensor models only, is through the use of a sensor model state. The state can be obtained by calling the `CSMPlugin` method `convertISDToSensorModelState`. Then, this state can be used to create a sensor model by calling the `CSMPlugin` method `constructSensorModelFromState`. Again, as in the previous method, the application will need to determine which plugin can convert the ISD into a valid state as

part of this process. The section “Sensor Model Selection” addresses how to do this. It is useful to mention at this point that since the state data can be transferred and stored, it can be saved and then used at a later time to create a sensor model that will be valid to perform sensor modeling operations on the original image.

In summary, a sensor model can be constructed in one of two ways: (1) using the image support data (ISD) as defined by the `CSM_ISD` class; and (2) using a sensor model state. The second method can be used to create a sensor model from a state that has been generated previously and stored to disk.

### 5.9.3.2 Sensor Model Selection

Selection of sensor models can take place at several points in the sensor model construction process. Selection can be made with respect to which ISD conversions will be exercised. Additionally, sensor model selection can be applied after ISD conversion has resulted in a sensor model state to control the sensor model constructions that are performed. Functions are provided to identify which plug-ins can construct a sensor model from either an ISD or a state. Sensor `mModel` selection can even extend to the selection among multiple sensor model objects that are producible. The `CSMPlugin` class provides some of the criteria that can be applied such as plug-in manufacturer, plugin release date, and sensor model version. Instantiated sensor models support additional criteria such as expected uncertainty and processing speed.

Note: Although expected uncertainty is directly available from the instantiated sensor model, processing speed is not a predefined metric that is directly available. If speed is needed by the application, as a selection criterion, then the application must devise its own way of measuring speed using the model. For example, the application might call the same method a number of times and measure the total elapsed time, or the application might make a sequence of related calls and measure the time for the sequence.

In addition to providing selection criteria, the plugin `CSMPlugin` classes also provides some shortcut functions that can be used to assess whether or not a particular ISD conversion or sensor model construction can be performed. Although these functions cannot be relied upon to provide definitive answers, they can quickly identify impossible alternatives and assist in the refinement of the possibilities.

### 5.9.4 Adding and Removing Plugins

The `CSMPlugin` plugin interfaces are designed such that derived classes will be plug-and-play. Specifically, this means that CSM plugins can be added and removed from the system configuration very easily without the need to recompile or re-link any source code. Of course, applications will first need to be integrated with the appropriate `CSMPlugin` plugin base class(es). Once this has been done, `CSMPlugin` sensor models can be added or removed without recompiling code, allowing sensor model addition and removal to be easily accomplished “in the field”.

At this point, it is useful to mention that CSM plugins may be added and removed from a system either while the system is running or while it is not running.

#### 5.9.4.1 Adding and Removing CSMPlugins from an Application Configuration

This section addresses the simple case where plugins are added and removed while the application is not running, as this may likely be a common scenario. The section to follow discusses adding and removing plugins while the application is running.

The application software looks for dynamically link libraries in certain places. It can look in a centralized location for all CSMPlugin plugin libraries, or it can look in several pre-set directories. For the purpose of this specification, the “plugin directory” will be known as the location in which the application software looks for libraries to load, keeping in mind that this might refer to multiple directories. To add a CSMPlugin sensor model, the library for that model will need to be placed in the “plugin directory”. The next time the application starts up, the new plugin will be included in the lists of available plugins owned by the CSMPlugin base classes and will be available for use by the application. See section 5.9.5 for a more detailed explanation of how plugins get added to the base class list.

To remove a CSM plugin while the application software is not running, the library for that plugin simply needs to be removed from the “plugin directory”.

Applications will typically load shared libraries among the first processing tasks that they perform when they first startup. Therefore, any new plugins added to the application configuration will be available for use the next time that the application software is started. Likewise, any plugins that have been removed from the “plugin directory” will not be available.

#### 5.9.4.2 Adding and Removing CSMPlugins while the Application is Running

As previously described, it will be typical for applications to be designed such that CSMPlugin sensor models plugins are added and removed while the application is not running, such that changes take effect the next time that the application software is started. However, this does not need to be the case. Some applications may need to remain operational for long periods of time, during which “plugin maintenance” including addition and removal of plugins might be necessary. The CSMPlugin class supports this requirement.

If desired, an application could load new plugin libraries at any time while the application is running. How this is done is left to the application developer. (In other words, the application developer is responsible for providing a means by which the application becomes aware of the new library. This can be through automatic detection of changes in the “plugin directory”, or a GUI menu allowing an operator to identify the new plugin.) Regardless of the method, whenever an application loads a CSMPlugin Library, the plugin derived class will automatically register itself with the CSMPlugin plugin base class (i.e., add itself to the list of available CSM plugins contained in the base class). No explicit method calls are required by the application other than to load the new CSMPlugin plugin library.

However, removing a CSMPlugin plugin from the system while the application is running is more complicated. Within the context of this document, plugin removal will be called

“expulsion”. Expelling a plugin will involve removing the plugin from the base class list, as well as closing the library associated with that plugin. After the library has been closed, it should be removed from the plugin directory if it is not desired for use the next time that the application starts and/or loads libraries.

Application developers are cautioned regarding the need to make sure that all processes dependent on the candidate library are ended before the library is closed. Furthermore, applications will need to keep track of which library is associated with each plugin in order to make sure that the correct library is closed when a plugin is to be removed.

### 5.9.5 Explanation of “Plugin Registration”

The CSMPlugin base class owns a list of all derived sensor model plugins that are currently installed and available for image processing. In actuality, this list is a list of pointers to the factory classes for the respective derived plugins. Client software can access this list of pointers through the CSMPlugin::getList() function.

Registration is the process whereby derived plugins get added to the base class list. The CSMPlugin base class provides a mechanism allowing derived plugins to “self-register” with the base class, thereby adding themselves to the base class list. This mechanism is described below.

The CSMPlugin base class (see header files CSMPlugin.h) has the a constructor CSMPlugin::CSMPlugin() that can only be called by objects that derive from the base class. This constructor adds the calling object’s pointer to a list of known derived type plugins. An example for CSMPlugin construction, as follows:

```
    csmPlugin::csmPlugin()
    {
        if (!theList)
        {
            theList = new csmPluginList;
        }

        if (theList)
        {
            theList->push_back(this);
        }
    }
}
```

Derived plugins must contain a static instance of themselves (i.e., a static instance of the derived type plugin). The effect of having the static instance is that when the runtime library gets loaded by the application, the static instance gets initialized resulting in the base class constructor adding the derived plugin to its list.

### 5.9.6 Application Developer Responsibilities

#### 5.9.6.1 Functionality Expected within the Application

The application software is responsible for providing the following functionality:

- Loading CSMPlugin plugin library files (CSMPlugin for applications using sensor models)
- Providing image support data to the sensor model selection and construction functions. A `csm_ISD` class object is provided when processing an image from native file format (or when a sensor model state is not available).

Note that the following convention should be observed by the Application when constructing `csm_ISD` objects. The Application should create ISD standard forms such as NITF 2.0 or 2.1, if possible. The next preferred form is BYTESTREAM, followed by FILENAME. Some plug-ins may not support file access operations.

- Providing a sensor model state to the sensor model selection and functions.
- Selecting among multiple possible sensor models (either before construction or after) using the sensor model selection and construction methods provided by the CSMPlugin plugin class
- If the application has the requirement to be able to expel a CSMPlugin plugin while the application is running, then the application is responsible for tracking which CSMPlugin plugin library is associated with each sensor model plugin by associating the library file handle with the unique CSMPlugin plugin name string.
- The CSMPlugin plugin classes can be used in applications that are single threaded or multithreaded. However, in multithreaded cases the application is responsible for using the methods defined within the CSMPlugin plugin class in such a way that provides the desired degree of thread safety. With the exception of mutually exclusive (mutex) locks that are used within some functions (see implementation of `removePlugin()`, for example), no thread safety mechanisms have been built into the CSMPlugin plugin classes. The functions within the CSMPlugin plugin classes having the internal mutex lock are so noted in the "Notes" section of the applicable method definitions in Section 5.10 and 5.15.

### 5.9.7 Compiling and Linking with the CSMPlugin Base Classes

#### 5.9.7.1 Supported Platforms and Environments

The CSMPlugin plugin classes have been designed not to rely on code that would cause it to be platform, operating system, or compiler dependent, or to require special compile time options. As a result, the CSMPlugin plugin classes can be used on any combination of platform, operating system, or compiler environment provided that a standard, fully compliant C++ compiler is used.

#### 5.9.7.2 General Instructions for Compiling and Linking

The CSMPlugin plugin base class is provided as source code files `CSMPlugin.cpp` and `CSMPlugin.h` (see Appendix D Example CPP Files and Appendix A Header Files). The application developer will need to compile this code into a shared library and then link with the library in order to incorporate the CSMPlugin appropriate base class(es) functionality into the application, and in order for the application to use the CSMPlugin's

associated derived classes. For clarification, example makefiles are provided in Appendix C Compiling for both the Unix and Windows/PC environments. These examples were written for the Sun Forte C++ and GNU C++ compilers in the Unix case and for the Visual C++ compiler in the Windows/PC case.

### 5.9.7.3 Compile Time Dependencies

In addition to linking with the CSMPugin plugin shared library as described above, applications are also required to link with the following library for proper operation of the CSMPugin class for the respective environments:

Unix	Windows/PC
libdl.so	Multi-threading and Dynamic Linking (/MD)

The Windows operating systems require that the main program and all dynamic link libraries (dll's) be compiled as either Release or Debug not a mixture of both. It is a project requirement that all sensor model dll's be delivered as a release. All integrator testing will be done with release libraries. Successful integration between the sensor model plugin dll's and the SET application require that many if not ALL of the compiler flags match. A listing of the compiler flag setting for the plugin shall be included in the Plugin Summary Form.

The second requirement is that the model libraries be compiled with the Multi-threading switch enabled (/MD). See the project files included with the VTS distribution for examples.

The third requirement, which is driven by Windows library conventions, is that the CSM library be built with a standard name. This is because the name of the CSM library is stored in the sensor model library. This uniform name is CSMAPI.dll.

## 5.9.8 Plugin Developer Responsibilities

### 5.9.8.1 CSMPugin Naming

All CSMPugin plugin derived classes will need to have a name that is unique among all CSMPugin CSM sensor models. If the plugin name is not specified within contractual documentation, then the plugin developer shall coordinate with the procuring government office to obtain the name for the plugin. This name shall be a null-terminated ASCII character string consisting of only the following characters: the digits '0' through '9', the letters 'A' through 'Z' (upper case only), and the underscore '\_'.

### 5.9.8.2 CSMSensorModelName Naming

All CSMSensorModel derived classes will need to have a name that is unique among all CSM sensor models of the same type. Since a CSM plugin may be capable of creating

more than one sensor model, each sensor model type that can be created must have its own unique name. This name not only must be unique within the models produced by that plugin, but it must also be unique across all models of the same type produced by all other CSM plugins. If sensor model name strings are not specified within contractual documentation, then the plugin developer shall coordinate with the procuring government office to obtain the names to be used for all sensor models that can be created by the CSM plugin. These names shall be a null-terminated ASCII character string consisting of only the following characters: the digits '0' through '9', the letters 'A' through 'Z' (upper case only), and the underscore '\_'.

#### 5.9.8.3 CSMPugin Manufacturer Name

All CSMPugin plugin derived classes will need to have a manufacturer name string that uniquely identifies the manufacturer of the plugin. For proper operation, this string must be unique for each organization that produces Community Sensor Models. If the manufacturer name string is not specified by contractual documentation, then the plugin developer shall contact the procuring government office to coordinate this string.

The CSMPugin plugin manufacturer name shall be a null-terminated ASCII character string consisting of only the following characters: the digits '0' through '9', the letters 'A' through 'Z' (upper case only) and the underscore '\_'.

#### 5.9.8.4 Plug-In Supporting Classes Naming

All supporting classes for the plugin will need to have names that are unique among all CSMPugin plugins to avoid name collisions with other plugins as well as with the SET. A recommended method for supporting this is to require the plugin to use its own C++ namespace.

```
namespace CSM_PLUGIN_UNIQUE_NAMESPACE
{
    class support_class_1_for_csm_plugin { ... }
}
```

If the plugin's namespace is not specified within contractual documentation, then the plugin developer shall coordinate with the procuring government office to obtain the name for the plugin's namespace. This name shall be a null-terminated ASCII character string consisting of only the following characters: the digits '0' through '9', the letters 'A' through 'Z' (upper case only), and the underscore '\_'.

### 5.9.9 Functionality Required within the Plugin

The CSMPugin plugin derived class is responsible for providing the following functionality:

#### 5.9.9.1 implementation of CSMPugin Class Pure Virtual Methods

The CSMPugin derived class must provide an implementation for all functions defined as virtual by the CSMPugin base class.

#### 5.9.9.2 Static Instance of Own Type

The derived class must contain a static instance of its own type. This static instance will get initialized when the derived library is loaded by the application, resulting in the base class constructor being called thereby adding this derived class to the base class list.

#### 5.9.9.3 Win32 Export Code

The plugin developer must include the following code in the header for the derived class, except with the real CSM plugin name described by paragraph 5.9.8.1 inserted in place of 'PLUGIN\_NAME' below. The symbol defined from this code should be used accordingly when writing the derived class header. See the additional background discussion provided in Appendix B Additional Explanation of Export Symbols.

This code is required for the proper operation of CSMPugin plugin class in Windows/PC environments. This declaration is part of the CSMMisc.h file so it will be included in every class.

```
#ifdef _WIN32
# ifdef PLUGIN_NAME_LIBRARY
#  define PLUGIN_NAME_EXPORT_API __declspec(dllexport)
# else
#  define PLUGIN_NAME_EXPORT_API __declspec(dllimport)
# endif
#else
# define SENSOR_MODEL_NAME_EXPORT_API
#endif
```

## **5.10 Detailed Plug-in Method Descriptions**

The following pages provide detailed information about each method provided by the CSMPugin class.

## NAME

5.10.1 CSMPlugin::getList();

## SYNOPSIS

```
static CSMWarning* getList(  
    CSMPluginList*&    aCSMPluginList)  
    throw              (CSMError);
```

## DESCRIPTION

The `getList()` method provides access to the list of all CSMPlugin derived classes that are currently registered with the base class.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`aCSMPluginList` is a reference to a list of pointers to all CSMPlugin derived classes that have registered with the base class. This list can be used to access the implementations to CSMPlugin base class virtual methods that have been implemented by derived classes.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

### – Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR :

Use this error if no other error is suitable. Its use is discouraged.

NOTES

This method should be ignored by plugins; it is meant for applications to access all loaded plugins.

The list returned by the method is private data and the application should not attempt to alter (add/remove items) the list.

SEE ALSO

None

## NAME

5.10.2 CSMPlugin::findPlugin();

## SYNOPSIS

```
static CSMWarning* findPlugin(  
                                const std::string&    pluginName,  
                                CSMPlugin*&          aCSMPlugin)  
                                throw                (CSMError);
```

## DESCRIPTION

The `findPlugin` method is a convenience function that accepts a CSM Plugin Name as defined by paragraph 5.9 of this document and returns a pointer to the CSMPlugin derived class that has that string as its name.

## INPUTS

`pluginName` is a ACSII character string that uniquely identifies a CSMPlugin.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`aCSMPlugin` a pointer to the CSMPlugin derived class.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The CSMPlugin class can be used in applications that are single threaded or multithreaded. However, in multithreaded cases the application is responsible for using the methods defined within the CSMPlugin class in such a way that provides the desired degree of thread safety. With the exception of mutually exclusive (mutex) locks that are used within some functions (see implementation of removePlugin(), for example), no thread safety mechanisms have been built into the CSMPlugin class.

The list returned by the method is private data and the application should not attempt to alter (add/remove items) the list.

## SEE ALSO

`CSMPlugin::removePlugin()`

## NAME

5.10.3 `CSMPlugin::removePlugin()`;

## SYNOPSIS

```
static CSMWarning* removePlugin(  
                                const std::string&    pluginName)  
                                throw                (CSMError);
```

## DESCRIPTION

This method searches the list of pointers to available CSM plugin derived classes and removes the pointer corresponding to the derived class whose name is equal to `pluginName`.

## INPUTS

`pluginName` is a null-terminated ACSII character string that uniquely identifies a `CSMPlugin` derived class, as defined by 5.9.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The CSMPlugin class can be used in applications that are single threaded or multithreaded. However, in multithreaded cases the application is responsible for using the methods defined within the CSMPlugin class in such a way that provides the desired degree of thread safety. With the exception of mutually exclusive (mutex) locks that are used within some functions (see implementation of removePlugin(), for example), no thread safety mechanisms have been built into the CSMPlugin class.

## SEE ALSO

`CSMPlugin::findPlugin()`

## NAME

5.10.4 CSMPlugin::canISDBeConvertedToSensorModelState()

## SYNOPSIS

```
virtual CSMWarning* canISDBeConvertedToSensorModelState(  
    const csm_ISD&          image_support_data,  
    const std::string&      sensor_model_name,  
    bool&                   convertible)  
    const throw             (CSMError) = 0;
```

## DESCRIPTION

The `canISDBeConvertedToSensorModelState()` indicates whether or not given image support data can be converted into a sensor model state for the given sensor model by the associated CSM plugin.

## INPUTS

`image_support_data` is a reference to the image support data that provides the source for a potential conversion.

`sensor_model_name` is a null-terminated ASCII character string that identifies one of the sensor models within the plugin.

Pointers to the functional outputs are also provided.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`convertible` is a boolean integer that indicates whether or not the identified sensor model state can be produced using the given image support data. A return of “false” indicates that the sensor model state for the given sensor model is definitely not producible and a return of “true” indicates that the sensor model state for the given sensor model is potentially producible with the associated CSM plugin.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function does not definitively indicate whether or not the associated plugin can be used for the production of usable sensor model state from the given image support data. Definitive results may be obtained using

`CSMPlugin::convertISDToSensorModelState()`.

## SEE ALSO

`CSMPlugin::convertISDToSensorModelState()`

## NAME

5.10.5 CSMPugin::canSensorModelBeConstructedFromState()

## SYNOPSIS

```
virtual CSMWarning* canSensorModelBeConstructedFromState(  
    const std::string& sensor_model_name,  
    const std::string& sensor_model_state,  
    bool& constructible)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `canSensorModelBeConstructedFromState()` indicates whether or not a given sensor model can be constructed by the associated CSM plugin.

## INPUTS

`sensor_model_name` is a null-terminated ASCII character string that identifies one of the sensor models within the plugin.

`sensor_model_state` is a null-terminated ASCII character string that contains the sensor model state data for one of the sensor models supported by the associated CSM plugin.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of NULL indicates no warning is present.

`constructible` is a boolean integer that indicates whether or not the associated sensor model is constructible. A return of "False" indicates that the given sensor model is definitely not constructible and a return of "True" indicates that the given sensor model is potentially constructible with the associated CSM plugin.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function does not definitively indicate whether or not the associated plugin can be used for the construction of a usable sensor model with any sensor model state.

Definitive results may be obtained using

`CSMPlugin::constructSensorModelFromState()`.

## SEE ALSO

`CSMPlugin::constructSensorModelFromState()`

## NAME

5.10.6 CSMPugin::canSensorModelBeConstructedFromISD()

## SYNOPSIS

```
virtual CSMWarning* canSensorModelBeConstructedFromISD (
    const csm_ISD&          image_support_data,
    const std::string&      sensor_model_name,
    bool&                   constructible)
    const throw             (CSMError) = 0;
```

## DESCRIPTION

The `canSensorModelBeConstructedFromISD()` indicates whether or not given image support data can be used to construct the sensor model specified by `sensor_model_name` by the associated CSM plugin. Note that the calling application must first get the number of models constructible by the associated plugin and also retrieve their respective names.

## INPUTS

`image_support_data` is a reference to the image support data that provides the source for a potential conversion.

`sensor_model_name` is a null-terminated ASCII character string that identifies one of the sensor models within the plugin.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`constructible` is a boolean integer that indicates whether or not the identified sensor model state can be produced using the given image support data. A return of "False" indicates that the sensor model state for the given sensor model is definitely not producible and a return of "True" indicates that the sensor model state for the given sensor model is potentially producible with the associated CSM plugin.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

Errors and warnings documented in section 6.4, must be handled, but not all conditions are errors for this method. Specifically, certain errors should not be used, as these will return an exception to the SET when setting the `constructible` argument value to `false` is the correct behavior. These errors are `ISD_NOT_SUPPORTED`, `SENSOR_MODEL_NOT_CONSTRUCTIBLE`, `SENSOR_MODEL_NOT_SUPPORTED`, and `UNKNOWN_SUPPORT_DATA`. Use of these errors remains appropriate for the `constructSensorModelFromISD()` method.

The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- `UNKNOWN_WARNING:`

- Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

- `UNKNOWN_ERROR:`

- Use this error if no other error is suitable. Its use is discouraged.

## NOTES

Note that before calling this method, the calling application will probably first need to get the number of sensor models constructible by the associated plugin, and then also get each of their names. Then, this method would be called ‘N’ times, once for each sensor model that could be built by the plugin.

## SEE ALSO

`CSMPlugin::getNSensorModels()`, `CSMPlugin::getSensorModelName()`,  
`CSMPlugin::convertISDToSensorModelState()`,  
`CSMPlugin::constructSensorModelFromISD()`

## NAME

5.10.7 CSMPugin::constructSensorModelFromState()

## SYNOPSIS

```
virtual CSMWarning* constructSensorModelFromState(  
    const std::string&    sensor_model_state,  
    CSMSensorModel*&    sensor_model)  
    const throw          (CSMError) = 0;
```

## DESCRIPTION

The `constructSensorModelFromState()` creates a sensor model object from the given sensor model state.

## INPUTS

`sensor_model_state` is a null-terminated ASCII character string that contains the sensor model state data for one of the sensor models supported by the associated CSM plugin.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`sensor_model` is a pointer to a sensor model object pointer that can be used to perform a variety of sensor modeling operations. Upon failure, a null pointer is returned.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- UNKNOWN\_WARNING:

- Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INVALID\_SENSOR\_MODEL\_STATE:

The sensor model state provided is not recognizable.

SENSOR\_MODEL\_NOT\_SUPPORTED:

A constructor for the given sensor model name is not available within the associated CSM plugin.

SENSOR\_MODEL\_NOT\_CONSTRUCTIBLE:

A constructor for the given sensor model is available, but is not able to successfully create a sensor model object.

## NOTES

None

## SEE ALSO

`CSMPlugin::canSensorModelBeConstructedFromState()`

## NAME

5.10.8 CSMPugin::constructSensorModelFromISD()

## SYNOPSIS

```
virtual CSMWarning* constructSensorModelFromISD(  
    const csm_ISD&          image_support_data,  
    const std::string&     sensor_model_name,  
    CSMSensorModel*&      sensor_model)  
    const throw            (CSMError) = 0;
```

## DESCRIPTION

The `constructSensorModelFromISD()` creates a sensor model object from the given image support data. The `sensor_model_name` is used to identify which sensor model within the plugin is to be constructed.

## INPUTS

`image_support_data` is a reference to the image support data that provides the source for model construction.

`sensor_model_name` is a null-terminated ASCII character string that identifies one of the sensor models within the plugin.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`sensor_model` is a pointer to a sensor model object pointer that can be used to perform a variety of sensor modeling operations. Upon failure, a null pointer is returned.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING :

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR :

Use this error if no other error is suitable. Its use is discouraged.

ISD\_NOT\_SUPPORTED :

The sensor model support data provided is not supported by this plugin.

MEMORY :

The allocated state string is insufficient to hold the entire sensor model state that would be produced by this function.

SENSOR\_MODEL\_NOT\_SUPPORTED :

A constructor for the given sensor model name is not available within the associated CSM plugin.

SENSOR\_MODEL\_NOT\_CONSTRUCTIBLE :

A constructor for the given sensor model is available, but is not able to successfully create a sensor model object.

NOTES

None

SEE ALSO

`CSMPlugin::canSensorModelBeConstructedFromISD()`

## NAME

5.10.9 CSMPugin::convertISDToSensorModelState()

## SYNOPSIS

```
virtual CSMWarning* convertISDToSensorModelState(  
    const csm_ISD&          image_support_data,  
    const std::string&      sensor_model_name,  
    std::string&            sensor_model_state)  
    const throw             (CSMError) = 0;
```

## DESCRIPTION

The `convertISDToSensorModelState()` converts the given image support data into a sensor model state string for the given sensor model.

## INPUTS

`image_support_data` is a reference to the image support data that provides the source for the conversion to sensor model state.

`sensor_model_name` is a null-terminated ASCII character string that contains the name of the sensor model associated with the desired sensor model state.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`sensor_model_state` is a pointer to a ASCII character string that contains sensor model state.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR :

Use this error if no other error is suitable. Its use is discouraged.

SENSOR\_MODEL\_NOT\_SUPPORTED :

An ISD conversion for the given sensor model is not available within the associated CSM plugin.

ISD\_NOT\_CONVERTIBLE :

An ISD conversion for the given sensor model is available, but is not able to successfully create a sensor model state.

MEMORY :

The allocated state string is insufficient to hold the entire sensor model state that would be produced by this function.

## NOTES

## SEE ALSO

`CSMPlugin::canISDBeConvertedToSensorModelState()`

## NAME

5.10.10 CSMPlugin::getManufacturer()

## SYNOPSIS

```
virtual CSMWarning* getManufacturer(  
    std::string& manufacturer_name)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getManufacturer()` function provides the name of the manufacturer who built the associated CSM plugin. The manufacturer name is copied to the location specified by the given character pointer.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`manufacturer_name` is a null-terminated, human-readable, ASCII character string constructed from only the following ASCII characters: the digits '0' through '9', the letters 'A' through 'Z', and the underscore '\_'. The manufacturer name is unique and corresponding for each organization that produces sensor model plugins.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

#### NOTES

To ensure the uniqueness of manufacturer name (and proper plugin operation), the manufacturer name must be managed across the procurements of all CSM plugins.

#### SEE ALSO

None

## NAME

5.10.11 CSMPugin::getNSensorModels()

## SYNOPSIS

```
virtual CSMWarning* getNSensorModels(  
    int& n_sensor_models)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getNSensorModels()` function provides the number of sensor models available in the associated CSM plugin.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`n_sensor_models` is the number of sensor models supported in the associated CSM plugin.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

Each sensor model corresponds to a single set of functional sensor model operations and a single sensor model state form.

## SEE ALSO

`CSMPlugin::getSensorModelName()`

## NAME

5.10.12 CSMPugin::getReleaseDate()

## SYNOPSIS

```
virtual CSMWarning* getReleaseDate(  
    std::string&         release_date)  
    const throw         (CSMError) = 0;
```

## DESCRIPTION

The `getReleaseDate()` function provides the release date of the associated CSM plugin. The release date is defined as the date on which the associated CSM plugin completed the construction process that resulted in a deliverable implementation. The release date is copied to the location specified by the given character pointer.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`release_date` is a null-terminated, ASCII character string of decimal digits in compliance with ISO 8601. The release date string will have a length of exactly 8 characters (not including the terminating null character) and be of the form `yyyymmdd` where `yyyy` is the year, `mm` is the month of the year, and `dd` is the day of the month.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

NOTES

None

SEE ALSO

None

## NAME

5.10.13 CSMPlugin::getSensorModelName()

## SYNOPSIS

```
virtual CSMWarning* getSensorModelName(  
    const int&          sensor_model_index,  
    std::string&       sensor_model_name)  
    const throw        (CSMError) = 0;
```

## DESCRIPTION

The `getSensorModelName()` function provides a human readable name for the given sensor model.

## INPUTS

`sensor_model_index` is the identifying number of a sensor model. This index is a nonnegative integer ranging from zero to one less than the number of sensor models provided by the associated CSM plugin.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`sensor_model_name` is a null-terminated, human-readable, ASCII character string constructed from only the following ASCII characters: the digits '0' through '9', the letters 'A' through 'Z', and the underscore '\_'. The sensor model name is unique and corresponding for each sensor model.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR :

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE :

The sensor model index given does not correspond to a sensor model that is supported by the associated plugin.

## NOTES

Each sensor model corresponds to a single set of functional sensor model operations and a single sensor model state form.

A sensor model name will be reported by a plugin if there is either an ISD conversion or a sensor model constructor associated with the sensor model. It is not necessary that a plugin contain both of these capabilities for a given sensor model.

## SEE ALSO

`CSMPlugin::getNSensorModels()`

## NAME

5.10.14 CSMPlugin::getSensorModelNameFromSensorModelState()

## SYNOPSIS

```
virtual CSMWarning* getSensorModelNameFromSensorModelState(  
    const std::string& sensor_model_state,  
    std::string& sensor_model_name)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getSensorModelNameFromSensorModelState()` extracts the name of the sensor model associated with the given sensor model state.

## INPUTS

`sensor_model_state` is a pointer to a ASCII character string that contains sensor model state.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`sensor_model_name` is a null-terminated ASCII character string that contains the name of the sensor model associated with the given sensor model state.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INVALID\_SENSOR\_MODEL\_STATE :

The sensor model state provided is not recognizable.

#### NOTES

This function is common to all CSM plugins and does not vary according to the other capabilities that are available within any plugin.

#### SEE ALSO

None

## NAME

5.10.15 CSMPugin::getSensorModelVersion()

## SYNOPSIS

```
virtual CSMWarning* getSensorModelVersion(  
    const std::string& sensor_model_name,  
    int& version)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getSensorModelVersion()` function provides the version number of the associated sensor model software. This software includes an associated constructor from sensor model state (`CSMPugin::constructSensorModel()` for the associated sensor model name) and all of the sensor model operations associated with the given sensor model type (as defined for the `CSMSensorModel` associated with a sensor model name). Version numbers always increase with time and always imply backwards compatibility.

## INPUTS

`sensor_model_name` is a ASCII character string that indicates the sensor model for which the version is requested.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`version` is a positive integer (strictly greater than zero) that indicates the version number of the given sensor model.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING :

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR :

Use this error if no other error is suitable. Its use is discouraged.

SENSOR\_MODEL\_NOT\_SUPPORTED :

A version number for the given sensor model is not available within the associated CSM plugin.

## NOTES

None

## SEE ALSO

`CSMPlugin::constructSensorModel()`, `CSMSensorModel`

## NAME

5.10.16 CSMPlugin::getPluginName()

## SYNOPSIS

```
virtual CSMWarning* getPluginName(  
                                std::string&      pluginName)  
                                const throw      (CSMError) = 0;
```

## DESCRIPTION

The `getPluginName()` function provides a human readable name for the CSM plugin. The plugin name data is copied to the location specified by the given character pointer.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`pluginName` is a null-terminated, human-readable, ASCII character string constructed from only the following ASCII characters: the digits '0' through '9', the letters 'A' through 'Z', and the underscore '\_'. The plugin name is unique and corresponding for each plugin.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

Each plugin corresponds to a single set of functional plugin operations and a single sensor model state form.

## SEE ALSO

`CSMPlugin::getSensorModelName()`, `CSMPlugin::findPlugin()`

## NAME

5.10.17 CSMPlugin::getCSMVersion()

## SYNOPSIS

```
virtual CSMWarning* getCSMVersion (
    int&          csmVersion)
    const throw  (CSMError) = 0;
```

## DESCRIPTION

The `getCSMVersion()` method returns the CSM API version that the plug-in was written to. CSM API v3.0 is specified by version equal to 3000. Use the static const `CURRENT_CSM_VERSION` defined in `CSMMisc.h` to return the current CSM API version that the plug-in is compiled to.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`csmVersion` is an integer describing the software version. The first digit indicates a major release, the following two digits represent minor updates and the least significant digit is always set to zero indicating a standard version. Setting the least significant digit to anything other than zero is reserved for testing subclassing versions .

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR :

Use this error if no other error is suitable. Its use is discouraged.

NOTES

None.

SEE ALSO

`csmSensorModel::getVersion`

## 5.11 Image Support Data (ISD)

ISD is provided by the SET for the plugins to make state data and sensor models. ISD is encapsulated in a C++ class for transfer through the CSM interface. ISD is passed as a pointer to a simple ISD base class (for example, `csm_ISD *isd`). The ISD base class is defined as follows:

The code for the image support data is located in section 6.2.1.

Classes derived from the base class are used for specific ISD formats. The CSM API includes ISD definitions for commonly used formats including:

1. NITF 2.0
2. NITF 2.1
3. ISD passed as the name of a file holding ISD
4. ISD passed as an unspecified byte stream

In each case, some code is provided in the header file to facilitate memory management. It is not the intention to provide any data processing in the ISD classes.

The ISD definition provided here is meant to be expandable. Additional formats will be defined, as needed. Any particular plugin may support a subset of these formats.

#### 5.11.1 NITF 2.0 ISD

### SYNOPSIS

The code of the NITF 2.0 ISD is listed in section 6.2.3.

### DESCRIPTION

This class is designed to hold the sections of an NITF 2.0 image file that may potentially hold ISD. These include the fileheader, image subheader and tagged record extensions (TRE). The TRE data may be byte data that is not NULL terminated. Therefore, it is held as a char pointer with a length associated with it. Although the TRE class allows direct access to data, it is strongly suggested that the set method be used to fill in the TRE data.

Multiple images are set only when it is desired to produce one sensor model for the set of images.

#### 5.11.2 NITF 2.1 ISD

### SYNOPSIS

The code for NITF 2.1 ISD is listed in section 6.2.2.

### DESCRIPTION

This class is designed to hold the sections of an NITF 2.1 image file that may potentially hold ISD. These include the fileheader, image subheader and tagged record extensions (TRE). The TRE data may be byte data that is not NULL terminated. Therefore, it is held as a char pointer with a length associated with it. Although the TRE class allows direct access to data, it is strongly suggested that the set method be used to fill in the TRE data.

Multiple images are set only when it is desired to produce one sensor model for the set of images.

### 5.11.3 Filename ISD

#### SYNOPSIS

The code for Filename ISD is listed in section 6.2.5.

#### DESCRIPTION

This class is designed allow a string indicating the name of a file that contains ISD. The field `_filename` should be set to the full path name of the file.

#### **5.11.4 Bytestream ISD**

##### **SYNOPSIS**

The code for Bytestream ISD is listed in section 6.2.4.

##### **DESCRIPTION**

This class is designed to hold ISD in a string of unspecified format. The field `_isd` is set with the ISD.

## 5.12 Sensor Model Functions

The sensor model member functions operate upon `csmSensorModel` objects. Table 4 gives a functional grouping of the various sensor model functions.

**Table 4 - Sensor Model Functions**

Group	Description	Methods	Page
Core Photogrammetry	These methods provide the basic photogrammetric operations.	<a href="#">groundToImage</a>	70
		<a href="#">imageToGround</a>	73
		<a href="#">imageToProximateImagingLocus</a>	77
		<a href="#">imageToRemoteImagingLocus</a>	79
Uncertainty Propagation	These methods provide for the propagation of photogrammetric uncertainty.	<a href="#">ComputeGroundPartials</a>	81
		<a href="#">computeSensorPartials</a>	83
		<a href="#">getCurrentParameterCovariance</a>	89
		<a href="#">setCurrentParameterCovariance</a>	91
		<a href="#">getCovarianceModel</a>	167
		<a href="#">getCurrentCrossCovarianceMatrix</a>	171
		<a href="#">getOriginalCrossCovarianceMatrix</a>	93
		<a href="#">setOriginalParameterCovariance</a>	95
		<a href="#">getOriginalParameterCovariance</a>	174
<a href="#">getUnmodeledError</a>	176		
<a href="#">getUnmodeledCrossCovariance</a>			
Time and Trajectory	These methods provide information regarding imaging time and trajectory for real sensors.	<a href="#">getTrajectoryIdentifier</a>	95
		<a href="#">getReferenceDateAndTime</a>	99
		<a href="#">getImageTime</a>	102
		<a href="#">getSensorPosition</a>	104
		<a href="#">getSensorVelocity</a>	106
Sensor Model Parameters	These methods provide information regarding sensor model parameters, their adjustment, and their associated	<a href="#">setCurrentParameterValue</a>	108
		<a href="#">getCurrentParameterValue</a>	110
		<a href="#">getParameterName</a>	112
		<a href="#">getNumParameters</a>	114

Group	Description	Methods	Page
	covariance.	<a href="#">setOriginalParameterValue</a> <a href="#">getOriginalParameterValue</a> <a href="#">setOriginalParameterType</a> <a href="#">getOriginalParameterType</a> <a href="#">setCurrentParameterType</a> <a href="#">getCurrentParameterType</a>	118 118 163 120 165 124
Sensor Model Information	These methods provide basic information about the imaging process, sensor model type, and area imaged.	<a href="#">getPedigree</a> <a href="#">getSensorModelName</a> <a href="#">getImageIdentifier</a> <a href="#">setImageIdentifier</a> <a href="#">getSensorIdentifier</a> <a href="#">getPlatformIdentifier</a> <a href="#">getReferencePoint</a> <a href="#">setReferencePoint</a> <a href="#">getImageSize</a> <a href="#">getCollectionIdentifier</a> <a href="#">isParameterShareable</a> <a href="#">getParameterSharingCriteria</a>	128 161 131 133 135 137 157 159 139 178 180 182
Sensor Model State	These methods create, set, or get major portions of sensor model state.	<a href="#">getSensorModelState</a>	141
Monoscopic Mensuration	These methods provide support for ground measurements that are made using only a single image.	<a href="#">getValidHeightRange</a> <a href="#">getIlluminationDirection</a>	143 147

## NAME

### 5.12.1 csmSensorModel::groundToImage()

Two implementations of this method are provided. The first implementation performs coordinate conversions only. The second provides for passing and returning accuracy information associated with the image and ground coordinates. The returned image coordinate error covariance shall include the effects from ground coordinate error covariance, adjustable sensor parameter error covariance, and unmodeled error covariance. This method shall include corrections for systematic errors as required by the Community Sensor Model Technical Requirements Document.

## SYNOPSIS

```
virtual CSMWarning* groundToImage(           const
double& x,           const double& y,
const double& z,           double& line,
double& sample,           double&
    achieved_precision,           const double&
    desired_precision = 0.001)
    const throw (CSMError) = 0;
```

```
virtual CSMWarning* groundToImage(           const
double& x,           const double& y,
const double& z,           const double
    groundCovariance[9],           double&
    line,           double& sample,
double imageCovariance[4],           double&
    achieved_precision,           const double&
    desired_precision = 0.001)
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `groundToImage()` function converts  $x$ ,  $y$  and  $z$  (meters) in ground space (ECEF) to line and sample (pixels) in image space. Two implementations of this method are provided. The first implementation performs coordinate conversions only. The second provides for passing and returning accuracy information associated with the image and ground coordinates.

## INPUTS

$x$ ,  $y$ , and  $z$  are ground coordinates in meters.

`groundCovariance` is an array of doubles consisting of the 3x3 covariance of the passed in ground point.

`desired_precision` (pixels) is the requested precision of the calculation. The default is 0.001 pixels.

## OUTPUTS

Upon successful completion, `groundToImage()` assigns values to line and sample. It also returns the `achieved_precision` of the calculation in pixels.

The overloaded implementation taking covariance information in `ground` (`groundCovariance`) additionally returns the image covariance (`imageCovariance`).

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`line` and `sample` are image coordinates in units of pixels.

`imageCovariance` is an array of doubles (pixels squared) consisting of the 2x2 covariance of the resultant image point. The output covariance includes the contribution from unmodeled error.

`achieved_precision` is the precision, in pixels, to which the calculation is achieved.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

### – Warnings:

#### `UNKNOWN_WARNING:`

The `desired_precision` is not met.

Use this warning if no other warning is suitable. Its use is discouraged.

#### `PRECISION_NOT_MET:`

The desired precision is not met.

#### `IMAGE_COORD_OUT_OF_BOUNDS:`

The specified ground location is not found within the image.

Since many sensor models are capable of returning coordinates outside of the image, this should only be a warning and not an error. This way the SET is responsible for handling this situation appropriately.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

NOTES

This method converts a ground coordinate to a line and sample in image space.

SEE ALSO

`csmSensorModel::imageToGround,`  
`csmSensorModel::getUnmodeledError,`  
`csmSensorModel::getValidHeightRange,`  
`csmSensorModel::getValidImageRange`

## NAME

### 5.12.2 csmSensorModel::imageToGround()

Two implementations of this method are provided. The first implementation performs coordinate conversions only. The second provides for passing and returning accuracy information associated with the image and ground coordinates. The returned ground coordinate error covariance shall include the effects from image coordinate error covariance, adjustable sensor parameter error covariance, and unmodeled error covariance. This method shall include corrections for systematic errors as required by the Community Sensor Model Technical Requirements Document.

## SYNOPSIS

```
virtual CSMWarning* imageToGround(           const
double& line,                               const double& sample,
const double& height,                       double& x,
double& y,                                  double& z,
double& achieved_precision,                const
double& desired_precision = 0.001)
      const throw (CSMError) = 0;
virtual CSMWarning* imageToGround(           const
double& line,                               const double& sample,
const double imageCovariance[4],           const
double& height,                            const double&
      heightVariance,                       double& x,
double& y,                                  double& z,
double groundCovariance[9],                double&
      achieved_precision,                   const double&
      desired_precision = 0.001)
      const throw (CSMError) = 0;
```

## DESCRIPTION

The `imageToGround()` function computes the ground coordinates ( $x$ ,  $y$ , and  $z$ ) where the imaging locus for the given line and sample (pixels) intersects the geodetic surface defined by the height. Two implementation of this method are provided. The first implementation performs coordinated conversions only. The second provides for passing and returning accuracy information associated with the image and ground coordinates. The image uncertainty is given in pixels squared and is passed in via a double array representing a 2 x 2 covariance matrix (`imageCovariance`). The line, sample and height inputs must lie within the range of validity of the sensor model as reported by the methods `getValidImageRange()` and `getValidHeightRange()`.

## INPUTS

`line` and `sample` are image coordinates in units of pixels.

`height` (meters) is measured with respect to the WGS-84 ellipsoid. The uncertainty in elevation is assumed to be zero when using an implementation that does not provide accuracy information. When using an implementation providing accuracy information, the uncertainty in meters squared is passed in via `heightVariance`.

`imageCovariance` is an array of doubles (pixels squared) consisting of the 2x2 covariance of the resultant image point. The input covariance is assumed to include the contribution from unmodeled error.

`heightVariance` is a double representing the variance of the passed in height measurement.

`desired_precision` (meters) is the requested precision of the calculation. The default is 0.001 meters.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`x`, `y`, and `z` are ground coordinates in units of meters.

`groundCovariance` is an array of doubles (meters squared) consisting of the 3x3 covariance of the passed in ground point.

`achieved_precision` is the precision, in meters, that the calculation achieved.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING`:

Use this warning if no other warning is suitable. Its use is discouraged.

`PRECISION_NOT_MET`:

The `desired_precision` is not met.

`IMAGE_COORD_OUT_OF_BOUNDS`:

The specified pixel location (line, sample, height) is not within the sensor model's range of validity. It is optional whether a sensor model implementation chooses to handle this condition and attempt to return ground coordinates. If a ground point cannot be determined, then an error is issued (see below) instead of this warning.

**NO\_INTERSECTION:**

No intersection of the imaging locus with ground is possible. The sensor model has computed the closest point of intersection and returned this value. For example, the sensor model may allow the height input to be variable and compute the intersection using an alternate value. If a ground point cannot be determined, then an error is issued (see below) instead of this warning.

**DATA\_NOT\_AVAILABLE:**

The imageToGround() calculation was not able to complete due to incomplete or invalid image support data. The image does not include all of the support data that the sensor model needs for this calculation.

– Errors:

**BOUNDS:**

The input pixel location is not within the sensor model's range of validity. The ground point calculation could not be completed. If the calculation is completed, a warning is issued instead of this error.

**UNKNOWN\_ERROR:**

The calculation was not able to complete due to causes not included in any of the more specific errors. Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function converts a given line and sample in image space to a corresponding ground point at the specified height on the Earth's surface. There are situations in which no corresponding ground point exists. An example of this would be when the ground point would be located behind the focal plane of an optical sensor. In this case, an error should be issued. Another example would be imaging over the horizon. In this case, a choice would exist between issuing an error or returning the point of closest approach along with a warning. In general, any situation that prevents a ground point from being calculated requires issuing an error. Any situation that prevents a calculated ground point from being trusted requires issuing a warning.

There is also a possible situation in which multiple ground points could correspond to the same image point and height combination. This method must determine the correct result based on the imaging geometry. If the correct result cannot be determined, an error would be issued.

#### SEE ALSO

`csmSensorModel::groundToImage,`  
`csmSensorModel::getUnmodeledError,`  
`csmSensorModel::getValidHeightRange,`  
`csmSensorModel::getValidImagRange`

NAME

5.12.3 `csmSensorModel::imageToProximateImagingLocus()`

## SYNOPSIS

```
virtual CSMWarning* imageToProximateImagingLocus(  
const double& line,                const double& sample,  
const double& x,                    const double& y,  
const double& z,                    double locus[6],  
double& achieved_precision,          const  
double& desired_precision = 0.001)  
const throw (CSMError) = 0;
```

## DESCRIPTION

The `imageToProximateImagingLocus()` function computes a proximate imaging locus, a vector approximation of the imaging locus for the given line and sample nearest the given `x`, `y` and `z`. The precision of this calculation refers to the locus's origin and does not refer to the locus's orientation. For an explanation of the proximate imaging locus section, see paragraph 5.3.

## INPUTS

`line` and `sample` are image coordinates in units of pixels.

`x`, `y`, and `z` are ground coordinates, in meters, near which the locus will be computed.

`locus` is an array of six doubles: a position and a direction vector.

`desired_precision` (meters) is the precision used for `groundToImage()` calls, if any, within the function. The default is 0.001 meters.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, the `imageToProximateImagingLocus()` function returns `locus` as follows:

```
locus[0] = position x    locus[1] = position y    locus[2] = position z  
locus[3] = direction x  locus[4] = direction y    locus[5] = direction z
```

`achieved_precision` (meters) is the precision returned from the computation  
`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition  
and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- `UNKNOWN_WARNING:`

- Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

- `UNKNOWN_ERROR:`

- An embedded `imageToGround()` call was unable to complete due to bad sensor model data.

- Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The `locus` must be allocated by the calling function as an array of at least six doubles.

The proximate imaging locus is computed under the assumption that the ground coordinate system is fixed to the Earth. This is important for SAR imagery since the SAR imaging process is sensitive to the relative velocity of sensor and target parallel to the line of sight. This means that the proximate imaging locus will be usable for finding the ground coordinates of objects fixed with respect to the Earth.

## SEE ALSO

`csmSensorModel::imageToRemoteImagingLocus`

## NAME

5.12.4 `csmSensorModel::imageToRemoteImagingLocus()`

## SYNOPSIS

```
virtual CSMWarning* imageToRemoteImagingLocus(  
const double& line,                const double& sample,  
double locus[6],                    double&  
    achieved_precision,            const double&  
    desired_precision = 0.001)  
    const throw (CSMError) = 0;
```

The `imageToRemoteImagingLocus()` function computes `locus`, a vector approximation of the imaging locus for the given `line` and `sample`. The precision of this calculation refers only to the origin of the locus vector and does not refer to the locus's orientation. For an explanation of the remote imaging locus, see the section at the beginning of this document.

## INPUTS

`line` and `sample` are in units of pixels.

`desired_precision` (meters) is the precision used for `groundToImage()` calls, if any, within the function. The default is 0.001 meters.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`locus` is an array of six doubles: a position and a direction vector.

Upon successful completion, the `imageToRemoteImagingLocus()` function returns `locus` as follows:

```
locus[0] = position x    locus[1] = position y    locus[2] = position z  
locus[3] = direction x  locus[4] = direction y    locus[5] = direction z
```

`achieved_precision` (meters) is the precision returned from the computation

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

UNKNOWN\_ERROR:

An embedded `imageToGround()` call was unable to complete due to bad sensor model data.

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The `locus` must be allocated by the calling function as an array of at least six doubles.

## SEE ALSO

`csmSensorModel::imageToProximateImagingLocus`

## NAME

5.12.5 `csmSensorModel::computeGroundPartials()`

## SYNOPSIS

```
virtual CSMWarning* computeGroundPartials(  
    const double& x,                const double& y,  
    const double& z,                double  
        partials[6])  
        throw                        (CSMError) = 0;
```

## DESCRIPTION

The `computeGroundPartials` function calculates the partial derivatives (`partials`) of image position (both line and sample) with respect to ground coordinates at the given ground position `x`, `y`, `z`. (Units for the partials are pixels per meter.).

## INPUTS

`x`, `y`, and `z` are ground coordinates in meters.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, `computeGroundPartials()` produces the partial derivatives as follows:

```
partials[0] = line wrt x,           partials[1] = line wrt y,  
partials[2] = line wrt z,           partials[3] = sample wrt x,  
partials[4] = sample wrt y,         partials[5] = sample wrt z.
```

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

The partial derivatives cannot be computed due to problems in the `groundToImage()` calculation.

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function calculates the partial derivatives of image position with respect to ground space at the given ground position.

The `partials` array must be allocated by the calling function to hold at least six doubles.

## SEE ALSO

`csmSensorModel::computeSensorPartials`,  
`csmSensorModel::groundToImage`, `csmSensorModel::imageToGround`

## NAME

5.12.6 `csmSensorModel::computeSensorPartials()`

## SYNOPSIS

```
virtual CSMWarning* computeSensorPartials(
const int&      index,                const double& x,
const double& y,                    const double& z,
double&        line_partial,        double&
    sample_partial,                double&
    achieved_precision,            const double&
    desired_precision = 0.001)
    throw (CSMError) = 0;
virtual CSM Warning* computeSensorPartials(
const int&      index,                const double&
    line,                const double& sample,
const double& x,                const double& y,
const double& z,                double&
    line_partial,            double&
    sample_partial,            double&
    achieved_precision,        const double&
    desired_precision = 0.001)
    throw (CSMError) = 0;
```

## DESCRIPTION

The `computeSensorPartials()` function calculates the partial derivatives of image position (both line and sample) with respect to the given sensor parameter (`index`) at the given ground position. (Units for the partials are pixels per parameter units.)

## INPUTS

`index` selects the sensor parameter.

`line` and `sample` are in units of pixels.

`x`, `y`, and `z` are ground coordinates in meters.

`desired_precision` (pixels) is the requested precision of the calculation. The default is 0.001 pixels. The precision refers to the precision of calls to `groundToImage()` made within this function.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`line_partial` is set by the function to the partial of line with respect to the indexed sensor parameter.

`sample_partial` is set by the function to the partial of line with respect to the indexed sensor parameter.

`achieved_precision` is the precision, in pixels, to which the calculation is achieved.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- `UNKNOWN_WARNING:`

- Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

- `INDEX_OUT_OF_RANGE:`

- `index` is less than zero or greater than or equal to the value returned by `getNumParameters()`

- `UNKNOWN_ERROR:`

- `line_partial` and `sample_partial` cannot be computed at the given ground position due to problems in the `groundToImage()` calculation.

- The parameter `tweak` is set too small for a numerical partial to be calculated.

- Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The two versions of this function perform the same operation. The second one exists for efficiency reasons.

Two versions of the function are provided. The first function, `computeSensorPartials()`, takes in only necessary information. It performs `groundToImage()` on the ground coordinate and then calls the second form of the function with the obtained line and sample. If the calling function has already performed

`groundToImage` with the ground coordinate, it may call the second function directly since it may be significantly faster than the first. The results are unpredictable if the line and sample provided do not correspond to the result of calling `groundToImage()` with the given ground position ( $x$ ,  $y$ , and  $z$ ).

See section 5.4 of this document for usage of `desired_precision` and `achieved_precision`.

#### SEE ALSO

`csmSensorModel::computeGroundPartials`,  
`csmSensorModel::groundToImage`, `csmSensorModel::imageToGround`

#### 5.12.7 `csmSensorModel::computeAllSensorPartials()`

##### SYNOPSIS

```
virtual CSMWarning* computeAllSensorPartials(
    const double&          x,
    const double&          y,
    const double&          z,
    std::vector<double>&    line_partials,
    std::vector<double>&    sample_partials,
    double&                achieved_precision,
    const double&          desired_precision = 0.001)
    throw                  (CSMError) = 0;

virtual CSM Warning* computeAllSensorPartials(
    const double&          line,
    const double&          sample,
    const double&          x,
    const double&          y,
    const double&          z,
    std::vector<double>&    line_partials,
    std::vector<double>&    sample_partials,
    double&                achieved_precision,
    const double&          desired_precision = 0.001)
    throw                  (CSMError) = 0;
```

##### DESCRIPTION

The `computeAllSensorPartials()` function calculates the partial derivatives of image position (both line and sample) with respect to each of the adjustable parameters at the given ground position.

##### INPUTS

`line` and `sample` are in units of pixels.

`x`, `y`, and `z` are ground coordinates in meters.

`desired_precision` (pixels) is the requested precision of the calculation. The default is 0.001 pixels. The precision refers to the precision of calls to `groundToImage()` made within this function.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`line_partials` is set by the function to the partial of line with respect to each sensor parameter. The units of each partial is pixels per parameter units. The order of the partials in the vector output follows the order of the associated parameter index.

`sample_partials` is set by the function to the partial of line with respect to the each sensor parameter. The units of each partial is pixels per parameter units. The order of the partials in the vector output follows the order of the associated parameter index.

`achieved_precision` is the precision, in pixels, to which the calculation is achieved.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- `UNKNOWN_WARNING:`

- Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

- `ALGORITHM:`

- `line_partials` and/or `sample_partials` cannot be computed at the given ground position due to problems in the numerical calculation.

- `UNKNOWN_ERROR:`

- Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The two versions of this function perform the same operation. The second one exists for efficiency reasons. The values returned by this method are identical to those that can be determined from multiple calls to `computeSensorPartials()`.

Two versions of this function are provided. The first version only takes in necessary information. It performs `groundToImage()` on the ground coordinate and then calls the second form of the function with the obtained line and sample. If the calling function

has already performed `groundToImage` with the ground coordinate, it may call the second function directly since it may be significantly faster than the first. The results are unpredictable if the line and sample provided do not correspond to the result of calling `groundToImage()` with the given ground position ( $x$ ,  $y$ , and  $z$ ).

See section 5.4 of this document for usage of `desired_precision` and `achieved_precision`.

#### SEE ALSO

`csmSensorModel::computeSensorPartials`

## NAME

5.12.8 `csmSensorModel::getCurrentParameterCovariance()`

## SYNOPSIS

```
virtual CSMWarning* getCurrentParameterCovariance(  
const int&      index1,          const int&  
      index2,  
          double&      covariance)  
      const throw (CSMError) = 0;
```

## DESCRIPTION

The `getCurrentParameterCovariance()` function returns the covariance of the specified parameter pair (`index1`, `index2`). The variance of the given parameter can be obtained by using the same value for `index1` and `index2`.

## INPUTS

`index1` selects the first of the parameter pair.

`index2` selects the second of the parameter pair.

The variance of a given parameter is given when `index1= index2`.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

The `getCurrentParameterCovariance()` function returns a double specifying the covariance value of the specified parameter pair.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

Either `index1` or `index2` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

## NOTES

This method set the covariance of the specified parameter pair. The variance of a single parameter can be set by using the same value for `index1` and `index2`.

## SEE ALSO

`csmSensorModel::getNumParameters`,  
`csmSensorModel::originalParameterCovariance`

## NAME

5.12.9 `csmSensorModel::setCurrentParameterCovariance()`

## SYNOPSIS

```
virtual CSMWarning* setCurrentParameterCovariance(  
const int&          index1,          const int&  
    index2,          const double& covariance)  
    throw            (CSMError) = 0;
```

## DESCRIPTION

The `setCurrentParameterCovariance()` function is used to set the covariance value of the specified parameter pair.

## INPUTS

`index1` selects the first of the parameter pair.

`index2` selects the second of the parameter pair.

The variance of a given parameter is set when `index1= index2`.

`covariance` sets the covariance value of the specified parameter pair.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

The function updates the sensor model covariance values.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

Either `index1` or `index2` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

## NOTES

This method set the covariance of the specified parameter pair. The variance of a single parameter can be set by using the same value for `index1` and `index2`.

## SEE ALSO

`csmSensorModel::getNumParameters`,  
`csmSensorModel::originalParameterCovariance`

## NAME

5.12.10 `csmSensorModel::setOriginalParameterCovariance()`

## SYNOPSIS

```
virtual CSMWarning* setOriginalParameterCovariance(  
const int&          index1,          const int&  
    index2,          const double& covariance)  
    throw            (CSMError) = 0;
```

## DESCRIPTION

The `setOriginalParameterCovariance()` function sets the covariance of the specified parameter pair (`index1`, `index2`). The variance of the given parameter can be set using the same value for `index1` and `index2`.

## INPUTS

`index1` selects the first of the parameter pair.

`index2` selects the second of the parameter pair.

The variance of a given parameter is set when `index1 = index2`.

`covariance` sets the covariance value of the specified parameter pair.

## OUTPUTS

The `setOriginalParameterCovariance()` function updates the sensor model's covariance matrix.

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

Either `index1` or `index2` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

## NOTES

This function sets the original covariance of the specified parameter pair. The variance of a single parameter can be set by using the same value for `index1` and `index2`.

## SEE ALSO

`csmSensorModel::getNumParameters`,  
`csmSensorModel::currentParameterCovariance`

## NAME

5.12.11 `csmSensorModel::getOriginalParameterCovariance()`

## SYNOPSIS

```
virtual CSMWarning*  getOriginalParameterCovariance(  
const int&           index1,                const int&  
    index2,                  
                        double&           covariance)  
const throw         (CSMError) = 0;
```

## DESCRIPTION

The `getOriginalParameterCovariance()` function gets the covariance of the specified parameter pair (`index1`, `index2`). The variance of the given parameter can be obtained using the same value for `index1` and `index2`.

## INPUTS

`index1` selects the first of the parameter pair.

`index2` selects the second of the parameter pair.

The variance of a given parameter is gotten when `index1= index2`.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`covariance` returns a double specifying the covariance value of the specified parameter pair.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

Either `index1` or `index2` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

## NOTES

This function returns the original covariance of the specified parameter `par`. The variance of a single parameter can be obtained by using the same value for `index1` and `index2`.

## SEE ALSO

`csmSensorModel::getNumParameters`,  
`csmSensorModel::currentParameterCovariance`

## NAME

5.12.12 `csmSensorModel::getTrajectoryIdentifier()`

## SYNOPSIS

```
virtual CSMWarning* getTrajectoryIdentifier(  
std::string          &trajectoryId)  
                    const throw          (CSMError) = 0;
```

## DESCRIPTION

The `getTrajectoryIdentifier()` function returns `trajectoryId` to indicate which trajectory was used to acquire the image. This `trajectoryId` is unique for each sensor type on an individual path.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`trajectoryId` is a null-terminated ASCII character `std::string` containing the trajectory name.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

`DATA_NOT_AVAILABLE:`

Warning is returned if the trajectory ID was never set.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function returns a unique identifier to indicate which trajectory was used to acquire the image. This ID is unique for each sensor type on an individual path.

`trajectoryId` must be allocated by the application.

`getTrajectoryIdentifier()` may not have any meaning for sensor models that do not pertain to real sensors (such as a polynomial sensor model).

## SEE ALSO

`csmSensorModel::getSensorIdentifier`,  
`csmSensorModel::getImageIdentifier`

## NAME

5.12.13 `csmSensorModel::getReferenceDateAndTime()`

## SYNOPSIS

```
virtual CSMWarning* getReferenceDateAndTime(  
std::string          &date_and_time)  
                    const throw          (CSMError)=  
0;DESCRIPTION
```

The `getReferenceDateAndTime()` function returns a UTC (Universal Time Coordinated) date and time near the time of the trajectory for the associated image.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

The reference date and time of the image is provided in a null-terminated ASCII character. Table 5 shows examples of the valid forms. Please note that the example table below is for illustration purposes and that the ISO 8601 standard governs the format. It is desirable to report the time in the finest increment available. The fractional second field is not limited to just three decimal points.

**Table 5 - Date and Time Format**

Precision	Format	Examples
Year	yyyy	"1961" "2000"
Month	yyyymm	"196104" "200002"
Day	yyyymmdd	"19610420" "20000229"
Hour	yyyymmddThh	"19610420T20Z" "20000229T11Z"
Minute	yyyymmddThhmm	"19610420T2000Z" "20000229T1130Z"
Second	yyyymmddThhmmss	"19610420T200000Z" "20000229T113000Z"
Fractional Second	yyyymmddThhmmssfff	"19610420T200000123Z" "20000229T113000123Z"

There is no maximum length of the date/time string.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

`UNKNOWN_WARNING`:

Use this warning if no other warning is suitable. Its use is discouraged.

`DATA_NOT_AVAILABLE`:

Warning is returned if the date and time are not set.

- Errors:

`UNKNOWN_ERROR`:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

A single reference date and time are associated with each trajectory and image times are all provided relative to this reference. The returned precision of the reference date and time should be heeded by applications.

The date produced by this function follows the ISO 8601 standard for date representation. Applications should be aware that this standard allows hours in the range 0 to 24 (24 being used strictly for midnight) and seconds in the range 0 to 60 (60 being used strictly for leap seconds).

ISO 8601 recommends that the latin capital letter T be placed between the date and the time.

ISO 8601 assumes the time is in local time. The capital letter Z designates Universal Time (UTC)

## SEE ALSO

`csmSensorModel::getImageTime`,  
`csmSensorModel::getTrajectoryIdentifier`

## NAME

5.12.14 `csmSensorModel::getImageTime()`

## SYNOPSIS

```
virtual CSMWarning* getImageTime(           const  
double& line,                               const double& sample,  
double& time)  
const throw (CSMError) = 0;
```

## DESCRIPTION

The `getImageTime()` function computes the time in seconds at which the pixel specified by `line` and `sample` was imaged. The time provided is relative to the reference date and time given by `getReferenceDateAndTime`.

## INPUTS

`line` and `sample` are in units of pixels.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`time` contains the time in seconds from the reference date and time.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function returns the time in seconds at which the specified pixel was imaged. The time provided is relative to the reference date and time given by the `getReferenceDateAndTime` function. `getImageTime` represents the time offset with the trajectory associated with the given image.

## SEE ALSO

`csmSensorModel::getReferenceDateAndTime`,  
`csmSensorModel::getSensorPosition`,  
`csmSensorModel::getSensorVelocity`,  
`csmSensorModel::getTrajectoryIdentifier`

## NAME

5.12.15 `csmSensorModel::getSensorPosition()`

## SYNOPSIS

```
virtual CSMWarning* getSensorPosition(
const double& line,                const double&
    sample,                        double& x,
double& y,                          double& z)
    const throw (CSMError) = 0;
virtual CSMWarning* getSensorPostion(
    const double& time
    double& x,
    double& y,
    double& z)
    const throw (CSMError) = 0
```

## DESCRIPTION

The `getSensorPosition()` function returns the position of the physical sensor at the given position in the image.

## INPUTS

`line` is an offset in image rows from the image origin.

`sample` is an offset in image columns from the image origin.

`time` is time pixel is imaged as reported by `getImageTime()`.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`x`, `y` and `z` contain the position of the sensor in meters.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

UNSUPPORTED\_FUNCTION:

## NOTES

None

## SEE ALSO

`csmSensorModel::getImageTime`,  
`csmSensorModel::getReferenceDateAndTime`,  
`csmSensorModel::getSensorVelocity`

## NAME

5.12.16 `csmSensorModel::getSensorVelocity()`

## SYNOPSIS

```
virtual CSMWarning* getSensorVelocity(
const double& line,                const double&
    sample,                        double& vx,
double& vy,                        double& vz)
    const throw (CSMError) = 0;

virtual CSMWarning* getSensorVelocity (
    const double& time,
    double& vx,
    double& vy,
    double& vz)
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getSensorVelocity()` function calculates the velocity of the physical sensor at the time a given image pixel is imaged. This can be specified by supplying either the location of the pixel imaged or the time of imaging.

## INPUTS

`line` is an offset in image rows from the image origin.

`sample` is an offset in image columns from the image origin.

`time` is the time the pixel is imaged as reported by `getImageTime()`.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, `getSensorVelocity()` returns the velocity `vx`, `vy` and `vz` of the sensor in meters per second.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

UNSUPPORTED\_FUNCTION:

## NOTES

None

## SEE ALSO

`csmSensorModel::getImageTime`,  
`csmSensorModel::getReferenceDateAndTime`,  
`csmSensorModel::getSensorPosition`

## NAME

5.12.17 `csmSensorModel::setCurrentParameterValue()`

## SYNOPSIS

```
virtual CSMWarning* setCurrentParameterValue(  
const int&          index,          const double&  
    value)          throw          (CSMError) = 0;
```

## DESCRIPTION

The `setCurrentParameterValue()` method is used to set the value of the adjustable parameter indicated by `index`.

## INPUTS

`index` selects the sensor parameter.

`value` contains the value to set the parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, the `setCurrentParameterValue()` method updates the sensor model.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

`index` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

## NOTES

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices).

## SEE ALSO

`csmSensorModel::getNumParameters`,  
`csmSensorModel::getOriginalParameterValue`,  
`csmSensorModel::getCurrentParameterCovariance`

## NAME

5.12.18 `csmSensorModel::getCurrentParameterValue()`

## SYNOPSIS

```
virtual double getCurrentParameterValue(           const  
int&          index,  
              double&          value)  
const throw  (CSMError)= 0;
```

## DESCRIPTION

The `getCurrentParameterValue()` method returns the value of the adjustable parameter given by `index`.

## INPUTS

`index` selects the sensor parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`value` returns a double corresponding to the value of the adjustable parameter.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

`INDEX_OUT_OF_RANGE:`

`index` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

**BOUNDS:**

`parameterType` is less than zero or greater than the maximum specified by the data type `parameterType`.

**NOTES**

This function returns the value of the model parameter indicated by the given `index`. `index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices)

See section 5.13.24 for a definition of the variable `parameterType`.

**SEE ALSO**

`csmSensorModel::getNumParameters`,  
`csmSensorModel::originalParameterValue`,  
`csmSensorModel::currentParameterCovariance`

## NAME

5.12.19 `csmSensorModel::getParameterName()`

## SYNOPSIS

```
virtual CSMWarning* getParameterName(  
const int&          index,          std::string&  
    name)          const throw      (CSMError) = 0;
```

## DESCRIPTION

The `getParameterName()` function returns `name` to indicate the name of the sensor model parameter for the specified index.

## INPUTS

`index` selects the sensor parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`name` is a null-terminated ASCII character string that contains the parameter name.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

The `index` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

## NOTES

`name` must be allocated by the application.

The `getParameterName()` function is intended for reporting purposes only. The parameter units may also be returned as part of the string.

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices).

## SEE ALSO

`csmSensorModel::getNumParameters`

## NAME

5.12.20 `csmSensorModel::getNumParameters()`

## SYNOPSIS

```
virtual CSMWarning* getNumParameters(  
    int& numParams)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getNumParameters()` function gets the number of parameters for the associated sensor model.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, `getNumParameters()` returns the number of parameters defined for the sensor model. The returned value is a one subscripted reference (values start at 1).

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The number of original parameters and the number of current parameters is the same and the function `getNumParameters()` provides the number of parameters for either.

## SEE ALSO

`csmSensorModel::getCurrentParameterValue,`  
`csmSensorModel::getOriginalParameterValue,`

## NAME

5.12.21 `csmSensorModel::setOriginalParameterValue()`

## SYNOPSIS

```
virtual CSMWarning* setOriginalParameterValue(  
const int&          index,          const double&  
    value)          throw          (CSMError) = 0;
```

## DESCRIPTION

The `setOriginalParameterValue()` method is used to set the original parameter value of the indexed parameter.

## INPUTS

`index` selects the sensor parameter.

`value` contains the value to set the parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, the `setOriginalParameterValue()` method updates the sensor model.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- `UNKNOWN_WARNING:`

- Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

- `UNKNOWN_ERROR:`

- Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

`index` is less than zero or greater than or equal to the value returned by `getNumParameters()`

NOTES

The method sets the original parameter value indicated by the index. This form is typically used when it is necessary to correct errors or provide missing information in the original data.

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices).

SEE ALSO

`csmSensorModel::getOriginalParameterCovariance,`  
`csmSensorModel::setOriginalParameterType,`  
`csmSensorModel::getOriginalParameterValue,`  
`csmSensorModel::setCurrentParameterType,`  
`csmSensorModel::getCurrentParameterValue`

## NAME

5.12.22 `csmSensorModel::getOriginalParameterValue()`

## SYNOPSIS

```
virtual CSMWarning* getOriginalParameterValue(  
const int&          index,  
                double&          value)  
const throw        (CSMError)= 0;
```

## DESCRIPTION

The `getOriginalParameterValue()` function returns the value of the adjustable parameter given by `index`.

## INPUTS

`index` selects the sensor parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`value` returns a double corresponding to the value of the indexed adjustable parameter. The value will have the type given by the function `getOriginalParameterType()`.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

index is less than zero or greater than or equal to the value returned by  
getNumParameters()

BOUNDS:

NOTES

This function returns the original parameter value indicated by the index.

See the function `getOriginalParameterType()` for a definition of the variable  
`parameterType`.

SEE ALSO

`csmSensorModel::originalParameterCovariance`,  
`csmSensorModel::getOriginalParameterType`,  
`csmSensorModel::setOriginalParameterValue`, `csmSensorModel::getCu`  
`rrentParameterType`, `csmSensorModel::setCurrentParameterValue`

## NAME

5.12.23 `csmSensorModel::getOriginalParameterType()`

## SYNOPSIS

```
virtual CSMWarning* getOriginalParameterType(  
const int&                index,  
                        CSMMisc::Param_CharType& pType)  
const throw              (CSMError) = 0;
```

## DESCRIPTION

The `getOriginalParameterType()` function returns the original type of the parameter given by `index`.

## INPUTS

`index` selects the sensor parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`pType` returns an enumerated type corresponding to one of the following values:

- 0 = None – the parameter value has not yet been initialized,
- 1 = Fictitious – the parameter value has been calculated by resection or other means,
- 2 = Real – the parameter value has been measured or read from support data,
- 3 = Exact – the parameter value has been specified and is assumed to have no uncertainty.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

index is less than zero or greater than or equal to the value returned by `getNumParameters()`.

## NOTES

The return values, None, Fictitious, Real and Exact, report the status of each adjustable parameter.

Triangulation is a nonlinear optimal adjustment, and one approach to this problem is initialization followed by iterative linearization and solution of the linear equations. Ideally, each adjustable sensor model parameter is initialized by a measurement and probable error estimate (based on metadata and/or look-up tables), and the size of the error estimate is such that the linearization is a good initial approximation. This can be considered as the norm, the "Real" parameter type case.

The "None" type can be considered as a means for the sensor model to signal to the SET that the original value of an adjustable parameter together with its error estimate are such that the linear approximation is not likely to be valid. In the worst case, there might be no measurement at all. A sensor model that has any current "None" parameters can be considered to be unusable for geopositioning, until the situation is fixed.

In the initialization phase of triangulation, the "None" type can be used as a signal to initialize the sensor model in the process called "resection", also known to others as "camera calibration". The resection uses the measurements input to triangulation and the sensor projective model to solve for new values and error estimates for the adjustable parameters, which were of type "None" going into the adjustment. If the resection is successful, the SET changes the current parameter type from "None" to "Fictitious". Thus "Fictitious" is a signal that the parameter value and error estimate are such that the linear approximation is likely to be good. The model could be used for geopositioning at this point. Still, the numerical value of a "Fictitious" parameter is not a

measurement that should carry any weight in triangulation, because it is based on the other measurements input to triangulation. If it were given weight, then that would amount to double-counting the other measurements. That is why "Fictitious" is a separate type from "Real".

Once the triangulation has been completed successfully, the current parameter types could be changed to "Real" by the SET for further exploitation.

If there is another, subsequent triangulation involving the sensor model under discussion, the original parameter type needs to be consulted to determine whether each parameter should have any weight in the triangulation. Note that the "original" parameter type is not available in the CSM 2.0 API, but is being added to the CSM 3.0 API.

The concept for the "Exact" type is to signal to the SET an adjustable parameter whose value is to be considered an exact constraint. It should carry infinite weight in triangulation, which corresponds to an uncertainty of zero. Existing SETs built to CSM 2.0 will break if zero variance values are encountered. So the workaround for CSM 2.0 is for the sensor model builder to assign the uncertainty according to a 0.001 pixel equivalent sigma in image space. In the CSM 3.0 API, SET builders shall handle the "Exact" parameter by omitting it entirely from the set of adjustable parameters.

A sensor model builder may classify their adjustable parameters as being "Exact", for at least two reasons. First, the adjustable parameter may be a calibrated value, such as focal length, which should not be adjusted in a typical triangulation scenario. However, given sufficiently strong geometry with overlapping images, a relatively sophisticated SET may attempt a bundle adjustment with self-calibration, thereby overriding the typical handling of the "Exact" type. This SET would override the sensor model's original "Exact" type by: 1) calling the newly proposed `setCurrentParameterType()` method and assigning the type to "Real", and 2) calling the `setCurrentParameterCovariance()` method and assigning its variance to a non-zero number for use in its triangulation solution. The SET would not be able to rely on the sensor model in order to directly determine an appropriate a priori variance; however, it could determine it indirectly, e.g. by calculating how much variance in the adjustable parameter equates to a desired variance in image space. A second reason why a sensor model builder would classify the adjustable parameter as "Exact" is that it may correspond to a higher order correction which should only be attempted by a relatively sophisticated SET in a block adjustment scenario with extensive redundancy and widespread point distribution. In order for the SET to know why the sensor model developer classified the adjustable parameter as "Exact", the sensor model developer shall provide such written explanation in a document that accompanies their delivery of the sensor model.

**SEE ALSO**

`csmSensorModel::getOriginalParameterCovariance,`  
`csmSensorModel::getOriginalParameterValue,`  
`csmSensorModel::getCurrentParameterType,`  
`csmSensorModel::setCurrentParameterType`

## NAME

5.12.24 `csmSensorModel::getCurrentParameterType()`

## SYNOPSIS

```
virtual CSMWarning* getCurrentParameterType(  
const int&                index,  
                        CSMMisc::Param_CharType& pType)  
const throw              (CSMError) = 0;
```

## DESCRIPTION

The `getCurrentParameterType()` function returns the current type of the parameter given by `index`.

## INPUTS

`index` selects the sensor parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`pType` returns an enumerated type corresponding to one of the following values:

- 0 = None – the parameter value has not yet been initialized,
- 1 = Fictitious – the parameter value has been calculated by resection or other means,
- 2 = Real – the parameter value has been measured or read from support data,
- 3 = Exact – the parameter value has been specified and is assumed to have no uncertainty.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

index is less than zero or greater than or equal to the value returned by  
`getNumParameters()`.

## NOTES

The return values, None, Fictitious, Real and Exact, report the status of each adjustable parameter.

Triangulation is a nonlinear optimal adjustment, and one approach to this problem is initialization followed by iterative linearization and solution of the linear equations. Ideally, each adjustable sensor model parameter is initialized by a measurement and probable error estimate (based on metadata and/or look-up tables), and the size of the error estimate is such that the linearization is a good initial approximation. This can be considered as the norm, the "Real" parameter type case.

The "None" type can be considered as a means for the sensor model to signal to the SET that the original value of an adjustable parameter together with its error estimate are such that the linear approximation is not likely to be valid. In the worst case, there might be no measurement at all. A sensor model that has any current "None" parameters can be considered to be unusable for geopositioning, until the situation is fixed.

In the initialization phase of triangulation, the "None" type can be used as a signal to initialize the sensor model in the process called "resection", also known to others as "camera calibration". The resection uses the measurements input to triangulation and the sensor projective model to solve for new values and error estimates for the adjustable parameters, which were of type "None" going into the adjustment. If the resection is successful, the SET changes the current parameter type from "None" to "Fictitious". Thus "Fictitious" is a signal that the parameter value and error estimate are such that the linear approximation is likely to be good. The model could be used for geopositioning at this point. Still, the numerical value of a "Fictitious" parameter is not a measurement that should carry any weight in triangulation, because it is based on the other measurements input to triangulation. If it were given weight, then that would

amount to double-counting the other measurements. That is why "Fictitious" is a separate type from "Real".

Once the triangulation has been completed successfully, the current parameter types could be changed to "Real" by the SET for further exploitation.

If there is another, subsequent triangulation involving the sensor model under discussion, the original parameter type needs to be consulted to determine whether each parameter should have any weight in the triangulation. Note that the "original" parameter type is not available in the CSM 2.0 API, but is being added to the CSM 3.0 API.

The concept for the "Exact" type is to signal to the SET an adjustable parameter whose value is to be considered an exact constraint. It should carry infinite weight in triangulation, which corresponds to an uncertainty of zero. Existing SETs built to CSM 2.0 will break if zero variance values are encountered. So the workaround for CSM 2.0 is for the sensor model builder to assign the uncertainty according to a 0.001 pixel equivalent sigma in image space. In the CSM 3.0 API, SET builders shall handle the "Exact" parameter by omitting it entirely from the set of adjustable parameters.

A sensor model builder may classify their adjustable parameters as being "Exact", for at least two reasons. First, the adjustable parameter may be a calibrated value, such as focal length, which should not be adjusted in a typical triangulation scenario. However, given sufficiently strong geometry with overlapping images, a relatively sophisticated SET may attempt a bundle adjustment with self-calibration, thereby overriding the typical handling of the "Exact" type. This SET would override the sensor model's original "Exact" type by: 1) calling the newly proposed `setCurrentParameterType()` method and assigning the type to "Real", and 2) calling the `setCurrentParameterCovariance()` method and assigning its variance to a non-zero number for use in its triangulation solution. The SET would not be able to rely on the sensor model in order to directly determine an appropriate a priori variance; however, it could determine it indirectly, e.g. by calculating how much variance in the adjustable parameter equates to a desired variance in image space. A second reason why a sensor model builder would classify the adjustable parameter as "Exact" is that it may correspond to a higher order correction which should only be attempted by a relatively sophisticated SET in a block adjustment scenario with extensive redundancy and widespread point distribution. In order for the SET to know why the sensor model developer classified the adjustable parameter as "Exact", the sensor model developer shall provide such written explanation in a document that accompanies their delivery of the sensor model.

**SEE ALSO**

`csmSensorModel::getOriginalParameterCovariance,`  
`csmSensorModel::getOriginalParameterValue,`  
`csmSensorModel::getOriginalParameterType,`  
`csmSensorModel::setOriginalParameterType`

## NAME

5.12.25 `csmSensorModel::getPedigree()`

## SYNOPSIS

```
virtual CSMWarning* getPedigree(  
    std::string& pedigree)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getPedigree()` function returns a character string that identifies the sensor, the model type, its mode of acquisition and processing path. For example, an image that could produce either an optical sensor model or a cubic rational polynomial model would produce different pedigrees for each case. See the notes section below for more information on how to construct a pedigree.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`pedigree` is a null-terminated ASCII character string that contains the pedigree information.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

In order to make the pedigree most useful, a set of conventions that should be followed when constructing a pedigree are presented here. These conventions are not mandatory, but following them will increase the usefulness of the sensor model.

The convention defines a set of standard pedigree components to be used constructing a pedigree, as applicable. In addition to these standard components, any particular pedigree will contain elements not defined here.

**Table 6 - Pedigree Convention Components**

Pedigree Component	When Used
_SAR	For Synthetic Aperture Radar Imagery
_SPOT	SAR Spotlight Imagery
_SCAN	SAR Scan Imagery (sometimes call search or strip mode)
_EO	Optical
_FRAME	EO/IR Frame Camera
_PB	EO/IR Pushbroom Camera
_WHISK	EO/IR Whiskbroom Camera
_IR	Infrared Imagery
_MSI	Multi-Spectral Imagery
_HSI	Hyper-Spectral Imagery
_RSM	Replacement Sensor Model
_RPC	Rational Cubic Polynomial used for SM

An example of a pedigree would be “PREDATOR\_IR\_FRAME\_RSM”. This would indicate to an application that a replacement sensor model was made for an infrared image using a frame camera.

Note that the term “PREDATOR” is not listed as standardized pedigree component. It would be impossible to keep this document up to date with all new sensors. In addition, the CSM API is not designed to carry sensor specific information, only generalized information describing the sensor.

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

While CSM does not limit the size of any identifiers, it is advisable to keep them small (<40 characters) since they may be inserted into NITF TREs with length restrictions.

SEE ALSO

None

## NAME

5.12.26 `csmSensorModel::getImageIdentifier()`

## SYNOPSIS

```
virtual CSMWarning* getImageIdentifier(  
std::string& imageId)  
const throw (CSMError)= 0;
```

## DESCRIPTION

The `getImageIdentifier()` function returns `imageId`, a unique identifier associated with the given sensor model.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`imageId` is a null-terminated ASCII character string that contains the image identifier.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

`DATA_NOT_AVAILABLE:`

Warning is returned if the requested data is unknown.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

`imageId` must be allocated by the application.

`imageId` is a unique identifier indicating the imaging operation associated with this sensor model.

## SEE ALSO

`csmSensorModel::getCollectionIdentifier`,  
`csmSensorModel::getSensorIdentifier`,  
`csmSensorModel::getTrajectoryIdentifier`,  
`csmSensorModel::setImageIdentifier`

## NAME

5.12.27 `csmSensorModel::setImageIdentifier()`

## SYNOPSIS

```
virtual CSMWarning* setImageIdentifier(  
const std::string& imageId)  
    throw (CSMError) = 0;
```

## DESCRIPTION

The `setImageIdentifier()` function sets a unique identifier for the image to which the sensor model pertains.

## INPUTS

`imageId` is an ASCII string containing the new image identifier value.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

The image identifier is updated.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

`imageId` is a null-terminated ASCII character string that is the image name. It can be no more than characters including the string termination character.

`imageId` should be a universally unique string.

## SEE ALSO

`csmSensorModel::getImageIdentifier`

## NAME

5.12.28 `csmSensorModel::getSensorIdentifier()`

## SYNOPSIS

```
virtual CSMWarning* getSensorIdentifier(  
std::string& sensorId)  
  
    const throw          (CSMError) = 0;
```

## DESCRIPTION

The `getSensorIdentifier()` function returns `sensorId` to indicate which sensor was used to acquire the image. This `sensorId` is meant to uniquely identify the sensor used to make the image.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`sensorId` is a null-terminated ASCII character string containing the sensor identifier.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

`DATA_NOT_AVAILABLE:`

Warning is returned if the sensor ID is unknown.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function returns a unique identifier to indicate which sensor was used to acquire the image.

`sensorId` must be allocated by the application to hold at least characters.

## SEE ALSO

`csmSensorModel::getCollectionIdentifier,`  
`csmSensorModel::getImageIdentifier,`  
`csmSensorModel::getTrajectoryIdentifier,`  
`csmSensorModel::getPlatformIdentifier`

## NAME

5.12.29 `csmSensorModel::getPlatformIdentifier()`

## SYNOPSIS

```
virtual CSMWarning* getPlatformIdentifier(  
std::string& platformId)  
const throw (CSMError) = 0;
```

## DESCRIPTION

The `getPlatformIdentifier()` function returns `platformId` to indicate which sensor was used to acquire the image. This `platformId` is meant to uniquely identify the platform used to collect the image.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`platformId` is a null-terminated ASCII character string containing the platform identifier.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

`DATA_NOT_AVAILABLE:`

Warning is returned if the platform ID is unknown.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

platformId must be allocated by the application.

## SEE ALSO

csmSensorModel::getCollectionIdentifier,  
csmSensorModel::getImageIdentifier,  
csmSensorModel::getTrajectoryIdentifier,  
csmSensorModel::getSensorIdentifier

## NAME

5.12.30 `csmSensorModel::getImageSize()`

## SYNOPSIS

```
virtual CSMWarning* getImageSize(          int&  
    num_lines,          int&          num_samples)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getImageSize()` function gets the number of lines and samples in the imaging operation.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, the `getImageSize()` function returns `num_lines` and `num_samples`, the numbers of lines and samples in the imaging operation for the sensor model.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

SEE ALSO

None

## NAME

5.12.31 `csmSensorModel::getSensorModelState()`

## SYNOPSIS

```
virtual CSMWarning* getSensorModelState(  
    std::string& state)  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getSensorModelState()` function returns the sensor model's state data. The first value of the sensor model's state is a unique identifier that indicates which CSM, including version number, created the state. The CSM creator defines the content of the state data. The format of the state data is ASCII.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`state` returns a character string that contains the state data.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The state includes the current parameter values for the model.

This method returns the current state of the model in an intermediate form. This intermediate form can then be processed, for example, by saving to file so that this model can be instantiated at a later date. The derived SensorModel is responsible for saving all information needed to restore itself to its current state from this intermediate form. A NULL pointer is returned if it is not possible to save the current state

## SEE ALSO

None

## NAME

5.12.32 `csmSensorModel::getValidHeightRange()`

## SYNOPSIS

```
virtual CSMWarning* getValidHeightRange(  
                                double&      minAltitude,  
double&      maxAltitude)  
                                const throw  (CSMError) = 0;
```

## DESCRIPTION

The `getValidHeightRange()` function returns the minimum and maximum ground heights that describe the range of validity of the model. For example, the model may not be valid at ground heights above the height of the sensor for physical models. This method is used in conjunction with `getValidImageRange()` to determine the full range of applicability of the sensor model.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`minAltitude` is the minimum valid ground height in meters above the WGS-84 ellipsoid.

`maxAltitude` is the maximum valid ground height in meters above the WGS-84 ellipsoid.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

NOTES

If there is no limit in the altitude range, the function will return +- 99,999 meters.

SEE ALSO

`csmSensorModel::getValidImageRange`

5.12.33 `csmSensorModel::getValidImageRange()`

### SYNOPSIS

```
virtual CSMWarning* getValidImageRange(  
                                double&    minRow,  
double&    maxRow,  
                                double&    minCol,  
double&    maxCol)  
                                const throw (CSMError) = 0;
```

### DESCRIPTION

The `getValidImageRange()` function returns the minimum and maximum values for image position (row and column) that describe the range of validity of the model. This range may not always match the physical size of the image. This method is used in conjunction with `getValidHeightRange()` to determine the full range of applicability of the sensor model.

### INPUTS

None.

### OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`minRow` (pixels) is the minimum row value for which the sensor model is valid.  
`maxRow` (pixels) is the maximum row value for which the sensor model is valid.  
`minCol` (pixels) is the minimum column value for which the sensor model is valid.  
`maxCol` (pixels) is the maximum column value for which the sensor model is valid.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

### ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR :

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

If there is no limit in the image range in a particular direction, the function will return a value of +- 99,999 pixels for the corresponding limit.

It is recommended that the valid image range extend to as large a region as is practical. This is useful to many numerical algorithms that extend beyond the limits of the image.

## SEE ALSO

`csmSensorModel::getValidImageRange`

## NAME

5.12.34 `csmSensorModel::getIlluminationDirection()`

## SYNOPSIS

```
virtual CSMWarning* getIlluminationDirection(  
const double&      x,          const double&  
    y,              const double&      z,  
double&           direction_x,      double&  
    direction_y,          double&  
    direction_z)  
    const throw      (CSMError) = 0;
```

## DESCRIPTION

The `getIlluminationDirection()` function calculates the direction of illumination at the given ground position `x`, `y`, `z`. The ground position is given in the coordinate system described in paragraph 5.1.2. The direction vectors are returned in the same coordinate system

## INPUTS

`x`, `y`, and `z` are ground coordinates in meters.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, `getIlluminationDirection()` produces three `doubles`, `direction_x`, `direction_y` and `direction_z`, defining a direction vector that points from the illumination source to the given ground point.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

The sensor model does not contain enough information to compute the direction of illumination.

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

This function calculates the direction of illumination with respect to ground space at the given ground position.

## SEE ALSO

None

## NAME

5.12.35 `csmSensorModel::getNumGeometricCorrectionSwitches ()`

## SYNOPSIS

```
virtual CSMWarning* getNumGeometricCorrectionSwitches (  
    int&    numGcs)  
    const throw    (CSMError) = 0;
```

## DESCRIPTION

The `getNumGeometricCorrectionSwitches ()` function gets the number of geometric correction switches for the associated sensor model.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`numGcs` returns the number of geometric correction switches defined for the sensor model. The returned value is a one subscripted reference (values start at 1).

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

  - `UNKNOWN_WARNING:`

    - Use this warning if no other warning is suitable. Its use is discouraged.

- Errors:

  - `UNKNOWN_ERROR:`

    - Use this error if no other error is suitable. Its use is discouraged.

NOTES

None

SEE ALSO

`csmSensorModel::getGeometricCorrectionName`

## NAME

5.12.36 `csmSensorModel::getGeometricCorrectionName()`

## SYNOPSIS

```
virtual CSMWarning* getGeometricCorrectionName(  
const int&          index,          std::string&  
    name)  
    const throw    (CSMError) =0;
```

## DESCRIPTION

The `getGeometricCorrectionName()` function returns `name` to indicate the name of the sensor model geometric correction for the specified `index`.

## INPUTS

`index` selects the geometric correction parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`name` is a null-terminated ASCII character string that contains the geometric correction name.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

The index is less than zero or greater than or equal to the value returned by `numGeometricCorrectionSwitches ()`

NOTES

`name` must be allocated by the application.

The `getGeometricCorrectionName ()` function is intended for reporting purposes only.

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices)

SEE ALSO

`csmSensorModel::numGeometricCorrectionSwitches`

## NAME

5.12.37 `csmSensorModel::setCurrentGeometricCorrectionSwitch()`

## SYNOPSIS

```
virtual CSMWarning* setCurrentGeometricCorrectionSwitch(  
const int&                index,                const  
bool&                    value,                const  
CSMMisc::Param_CharType& parameterType)        (CSMError)= 0;  
                                throw
```

## DESCRIPTION

The `setCurrentGeometricCorrectionSwitch()` is used to set the switch of the geometric correction indicated by `index`. A geometric correction switch of “False” turns off the associated (by `index`) geometric correction. A value of “True” turns on the associated (by `index`) geometric correction. If any of the geometric correction switches are set to “False”, then the error propagation associated with the `groundToImage()` and `imageToGround()` methods is not valid. This method is not intended to be used under normal SET operations but can be used to perform verification and validation of the sensor model.

## INPUTS

`index` selects the systematic error correction switch.

The “`value`” is a Boolean switch.

`parameterType` contains the value to set the parameter type.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

The index is less than zero or greater than or equal to the value returned by `numGeometricCorrectionSwitches ()`.

## NOTES

This function returns the value of the geometric correction switch indicated by the given `index`.

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices)

## SEE ALSO

`csmSensorModel::numGeometricCorrectionSwitches`,  
`csmSensorModel::getGeometricCorrectionName`

## NAME

5.12.38 `csmSensorModel::getCurrentGeometricCorrectionSwitch()`

## SYNOPSIS

```
virtual CSMWarning* getCurrentGeometricCorrectionSwitch(  
const int& index,  
bool& value)  
const throw (CSMError)= 0;
```

## DESCRIPTION

The `getCurrentGeometricCorrectionSwitch()` method returns the value of the geometric correction switch given by `index`.

## INPUTS

`index` selects the geometric correction switch.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`value` returns a boolean corresponding to the state of the associated systematic error correction.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

The index is less than zero or greater than or equal to the value returned by `getNumGeometricCorrectionSwitches ()`.

NOTES

This function returns the value of the geometric correction switch indicated by the given `index`.

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices)

SEE ALSO

`csmSensorModel::getNumGeometricCorrectionSwitches`,  
`csmSensorModel::getGeometricCorrectionName`

## NAME

5.12.39 `csmSensorModel::getReferencePoint()`

## SYNOPSIS

```
virtual CSMWarning* getReferencePoint(  
                                double&    x,  
                                double&    y,  
                                double&    z)  
    const throw    (CSMError) =0;
```

## DESCRIPTION

The `getReferencePoint()` function returns `x`, `y` and `z` in meters to indicate the general location of the image.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`x`, `y`, and `z` are ground coordinates in meters.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

The reference point was not set.

Use this error if no other error is suitable. Its use is discouraged.

## USAGE

Coordinates are rough and may or may not refer to a point on the image. Typically, the coordinate is an estimate of the center point, but could be the center of the first line, etc.

## SEE ALSO

`csmSensorModel::setReferencePoint`

## NAME

5.12.40 `csmSensorModel::setReferencePoint()`

## SYNOPSIS

```
virtual CSMWarning* setReferencePoint(  
    const double&    x,  
    const double&    y,  
    const double&    z)  
    throw             (CSMError)=0;
```

## DESCRIPTION

The `setReferencePoint()` function sets the reference point to the input `x`, `y` and `z` position in meters to indicate the general location of the image.

## INPUTS

`x`, `y`, and `z` are ground coordinates in meters.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

The reference point of the sensor model is updated.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## USAGE

Coordinates do not need to refer to a point on the image. Typically, the coordinate is an estimate of the center point, but could be the center of the first line, etc.

## SEE ALSO

`csmSensorModel::getReferencePoint`

## NAME

5.12.41 `csmSensorModel::getSensorModelName()`

## SYNOPSIS

```
virtual CSMWarning* getSensorModelName(  
    std::string& name)  
    const throw (CSMError) = 0 ;
```

## DESCRIPTION

The `getSensorModelName()` function returns a string indicating the name of the sensor model. The sensor model's name is described in section 3.1.13.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`name` will return a character string containing the sensor model's name.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

USAGE

SEE ALSO

None

## NAME

5.12.42 `csmSensorModel::setOriginalParameterType()`

## SYNOPSIS

```
virtual CSMWarning* setOriginalParameterType(          const  
int&                index,                          const  
CSMMisc::Param_CharType& parameterType)            (CSMError) = 0;  
                throw
```

## DESCRIPTION

The `setOriginalParameterType()` method is used to set the value of the original parameter type of the indicated adjustable parameter.

## INPUTS

`index` selects the sensor parameter.

`parameterType` contains the value to set the parameter type.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, the `setOriginalParameterType()` method updates the sensor model.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

`index` is less than zero or greater than or equal to the value returned by `numParameters()`.

BOUNDS:

`parameterType` is less than zero or greater than the maximum specified by the data type `parameterType`.

## NOTES

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices)

See section 5.12.23 for a definition of the variable `parameterType`.

## SEE ALSO

`csmSensorModel::numParameters`,  
`csmSensorModel::getOriginalParameterType`,  
`csmSensorModel::getCurrentParameterType`, `csmSensorModel::setCurrentParameterType`

NAME

5.12.43 `csmSensorModel::setCurrentParameterType()`

## SYNOPSIS

```
virtual CSMWarning* setCurrentParameterType(          const  
int&                index,                          const  
CSMMisc::Param_CharType& parameterType)           (CSMError) = 0;  
                throw
```

## DESCRIPTION

The `setCurrentParameterType()` method is used to set the value of the current parameter type of the indicated adjustable parameter.

## INPUTS

`index` selects the sensor parameter.

`parameterType` contains the value to set the parameter type.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

Upon successful completion, the `setCurrentParameterType()` method updates the sensor model.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

INDEX\_OUT\_OF\_RANGE:

`index` is less than zero or greater than or equal to the value returned by `numParameters()`.

BOUNDS:

`parameterType` is less than zero or greater than the maximum specified by the data type `parameterType`.

## NOTES

`index` is a zero subscripted reference (values range from 0 through N-1, with N being the maximum number of indices)

See section 5.12.23 for a definition of the variable `parameterType`.

## SEE ALSO

`csmSensorModel::numParameters`,  
`csmSensorModel::getCurrentParameterType`,  
`csmSensorModel::getOriginalParameterType`,  
`csmSensorModel::setOriginalParameterType`

#### 5.12.44 Covariance Model

Adjustable parameter error covariance matrices between pairs of images can be formulated in one of two possible techniques: direct and indirect. A direct error covariance matrix can for example be obtained as a by-product of a block adjustment which yields a full error covariance matrix pertaining to all images involved in the adjustment. An indirect error covariance matrix constructs the error cross covariance submatrix as a function of the error covariance matrix for each image, the time difference between acquisition, and a temporal de-correlation model.

Note that once the Geopositioning Metadata Model (GMM) becomes a standard, it will provide the means for a SET to access the *a posteriori* full error covariance matrix for an entire block of images that have been triangulated. Hence, for this case of a direct covariance formulation, the SET will not be required to go through the API to access error cross-covariance information from the sensor model.

The long-term goal is for CSM 3.0 to eventually add an album class, which would reside at a higher level than the CSM sensor model plugin class. Such album class is a concept for consideration, and not yet an actual design. The builders of this album class would have to know/understand the details of the cross covariance models used by each sensor model builder, but the SETs would not require this detailed knowledge. The interim solution, until an album class has been established, is to replace the original `getCovarianceModel()` method with new methods which extend the scope of the CSM API to handle modeling of multiple images.

The remainder of this section consists of two sub-sections pertaining to `getCurrentCrossCovarianceMatrix()` and `getOriginalCrossCovarianceMatrix()` which return the current and original, respectively, error cross covariance matrices between the instantiated and specified sensor model.

## NAME

5.12.44.1 `csmSensorModel::getCurrentCrossCovarianceMatrix()`

## SYNOPSIS

```
virtual CSMWarning*  getCurrentCrossCovarianceMatrix(  
const int numSM,  
    const csm_SensorModel** SMs,  
    const double line1,  
    const double sample1,  
    const double* lines,  
    const double* samples,  
    double **&crossCovarianceMatrix,  
    int &M)  
    const throw      (CSMError) = 0;  DESCRIPTION
```

The `getCurrentCrossCovarianceMatrix()` function returns matrices containing all elements of the error cross covariance matrix between the instantiated sensor model and `numSM` other sensor models (SMs). Images may be correlated because they are taken by the same sensor or from sensors on the same platform. Images may also be correlated due to post processing of the sensor models. The data returned here may need to be supplemented with the single image covariance from `getCurrentParameterCovariance()` and `getUnmodeledError()`.

## INPUTS

`numSM` is the number of other sensor models (not counting the currently instantiated model).

`SMs` is a n array of pointers to the other sensor models.

`line1` and `sample1` are the image coordinates of a point in the instantiated image.

`lines` and `samples` are arrays containing the image coordinates of points in the other images.

## OUTPUTS

`crossCovarianceMatrix` is a `numSM` by `M*M` two-dimensional array containing the current error cross covariance matrix between the instantiated image and the other

images. Each row of `crossCovarianceMatrix` contains an array of length  $M \times M$  which can be re-structured to form the associated  $M$  by  $M$  cross covariance matrix.  $M$  is an integer representing the number of rows and columns in the cross covariance matrix.

## ERRORS & WARNINGS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

All errors and warnings documented in section 6.4 must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- `DATA_NOT_AVAILABLE :`

- warning is returned if the requested data is not available.

- Errors:

- `SENSOR_MODEL_NOT_SUPPORTED :`

- The sensor model passed to the function does not support this method of computing cross-covariance.

- `UNSUPPORTED_FUNCTION :`

- The sensor model being called does not support this method of computing cross-covariance.

- `INVALID_USE :`

- The sensor model pointer passed is `NULL` or refers to the same sensor model as the one being called.

- `UNKNOWN_ERROR :`

- Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The calling program is responsible for allocating the array used to return the covariance matrix and for freeing that memory when it is no longer needed. The calling program should call the `getNumParameters()` method on the current sensor model to determine how much space to allocate for the array.

The order of parameters in the matrix is the same as the index parameter in the `getCurrentParameterValue()` method. Therefore, all adjustable parameters, including those that are of type "EXACT", are represented in the `getCurrentCrossCovarianceMatrix` method. The SET has the responsibility to extract the usable columns and rows from the cross covariance matrix, taking into account which adjustable parameters are "EXACT" or are not used in the current scenario. It is the responsibility of the sensor model to ensure that it returns self-consistent and valid results. Regarding self-consistency, if sensor model A is instantiated and a pointer to sensor model B is passed into the method, then the returned error cross covariance matrix must be the transpose of the matrix returned if the roles of sensor model A and B are reversed. Regarding validity of the results, the fully assembled multi-image error covariance matrix must be positive definite. See the introduction section (5.12.44) above.

This method must zero fill the elements of the cross covariance matrices if no cross correlation is present.

#### SEE ALSO

`csmSensorModel::getOriginalCrossCovarianceMatrix`

## NAME

5.12.44.2 `csmSensorModel::getOriginalCrossCovarianceMatrix()`

## SYNOPSIS

```
virtual CSMWarning* getOriginalCrossCovarianceMatrix(  
const int numSM,  
    const csm_SensorModel** SMs,  
    const double line1,  
    const double sample1,  
    const double* lines,  
    const double* samples,  
    double **&crossCovarianceMatrix,  
    int &M)  
    const throw (CSMError) = 0; DESCRIPTION
```

The `getOriginalCrossCovarianceMatrix()` function returns matrices containing all elements of the error cross covariance matrix between the instantiated sensor model and `numSM` other sensor models (SMs). Images may be correlated because they are taken by the same sensor or from sensors on the same platform. Images may also be correlated due to post processing of the sensor models. The data returned here may need to be supplemented with the single image covariance from `getOriginalParameterCovariance()` and `getUnmodeledError()`.

## INPUTS

`numSM` is the number of other sensor models (not counting the currently instantiated model).

`SMs` is a `n` array of pointer to the other sensor models.

`line1` and `sample1` are the image coordinates of a point in the instantiated image.

`lines` and `samples` are the arrays containing image coordinates of points in the other images.

## OUTPUTS

`crossCovarianceMatrix` is a `numSM` by `M*M` two-dimensional array containing the original error cross covariance matrix between the instantiated image and the other images. Each row of `crossCovarianceMatrix` contains an array of length `M*M` which can be re-structured to form the associated `M` by `M` cross covariance matrix.

$M$  is an integer representing the number of rows and columns in the cross covariance matrix.

## ERRORS & WARNINGS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

All errors and warnings documented in section 6.4 must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

- Warnings:

- `DATA_NOT_AVAILABLE :`

- warning is returned if the requested data is not available.

- Errors:

- `UNKNOWN_ERROR :`

- Use this error if no other error is suitable. Its use is discouraged.

## NOTES

The calling program is responsible for allocating the array used to return the covariance matrix and for freeing that memory when it is no longer needed. The calling program should call the `getNumParameters()` method on the original sensor model to determine how much space to allocate for the array

The order of parameters in the matrix is the same as the index parameter in the `getCurrentParameterValue()` method. Therefore, all adjustable parameters, including those that are of type “EXACT”, are represented in the `getOriginalCrossCovarianceMatrix` method. The SET has the responsibility to extract the usable columns and rows from the cross covariance matrix, taking into account which adjustable parameters are “EXACT” or are not used in the current scenario. It is the responsibility of the sensor model to ensure that it returns self-consistent and valid results. Regarding self-consistency, if sensor model A is instantiated and a pointer to sensor model B is passed into the method, then the returned error cross covariance matrix must be the transpose of the matrix returned if

the roles of sensor model A and B are reversed. Regarding validity of the results, the fully assembled multi-image error covariance matrix must be positive definite.

See the introduction section (5.12.44) above.

This method must zero fill the elements of the cross covariance matrices if no cross correlation is present.

#### SEE ALSO

`csmSensorModel::getCurrentCrossCovarianceMatrix`

## NAME

5.12.45 `csmSensorModel::getUnmodeledError()`

## SYNOPSIS

```
virtual CSMWarning* getUnmodeledError(  
    const double    line,  
    const double    sample,  
    double          covariance[4] )  
    const throw     (CSMError) = 0;
```

## DESCRIPTION

The `getUnmodeledError` function gives a sensor specific error for the given input image point. The error is reported as the four terms of a 2x2 covariance mensuration error matrix. This error term is meant to map error terms that are not modeled in the sensor model to image space for inclusion in error propagation. The extra error is added to the mensuration error that may already be in the matrix.

## OUTPUTS

The matrix is filled in as follows:

```
covariance[0] = line variance.  
covariance[1] = line-sample covariance.  
covariance[2] = line-sample covariance.  
covariance[3] = sample variance.
```

## ERRORS & WARNINGS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

All errors and warnings documented in section 6.4 must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`DATA_NOT_AVAILABLE:`

warning is returned if the requested data is not available.

– Errors:

`UNKNOWN_ERROR:`

Use this error if no other error is suitable. Its use is discouraged.

## USAGE

`covariance` has to have memory allocated by the calling routine to hold at least four doubles. The values should be initialized since this function adds to the uncertainty.

## SEE ALSO

`getUnmodeledCrossCovariance()`

## NAME

5.12.46 `csmSensorModel::getUnmodeledCrossCovariance()`

## SYNOPSIS

```
virtual CSMWarning* getUnmodeledCrossCovariance (
    const double      pt1Line,
    const double      pt1Sample,
    const double      pt2Line,
    const double      pt2Sample,
    double            crossCovariance[4] )
    const throw      (CSMError) = 0;
```

## DESCRIPTION

The `getUnmodeledCrossCovariance` function gives the cross covariance for unmodeled error between two image points on the same image. The error is reported as the four terms of a 2x2 matrix. The unmodeled cross covariance is added to any values that may already be in the cross covariance matrix.

## OUTPUTS

The matrix is filled in as follows:

```
crossCovariance[0] = line variance.
crossCovariance[1] = line-sample covariance.
crossCovariance[2] = line-sample covariance.
crossCovariance[3] = sample variance.
```

## ERRORS & WARNINGS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

All errors and warnings documented in section 6.4 must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`DATA_NOT_AVAILABLE :`

Warning is returned if the requested data is not available.

– Errors:

`UNKNOWN_ERROR :`

Use this error if no other error is suitable. Its use is discouraged.

None.

## USAGE

`crossCovariance` has to have memory allocated by the calling routine to hold at least four doubles. The values should be initialized since this function adds to the uncertainty.

## SEE ALSO

`getUnmodeledError()`

## NAME

5.12.47 `csmSensorModel::getCollectionIdentifier()`

## SYNOPSIS

```
virtual CSMWarning* getCollectionIdentifier(  
std::string& collectionId)  
const throw (CSMError) = 0;
```

## DESCRIPTION

The `getCollectionIdentifier()` function returns `collectionId` to indicate an identifier that uniquely identifies a collection activity by a sensor platform. This `collectionId` will vary depending on the type of sensor and platform. For example, for airborne platforms this may be called “mission ID” (e.g., `ACFT_MISN_ID` field in the NITF ACTFA and ACFTB tags), while for orbital platforms this will likely correspond to some data element of the ephemeris (e.g., pass number).

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`collectionId` is a null-terminated ASCII character string containing the collection identifier.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

`DATA_NOT_AVAILABLE:`

Warning is returned if the collection ID is unknown or not applicable for this sensor model.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

`collectionId` must be allocated by the application.

## SEE ALSO

`csmSensorModel::getImageIdentifier`,  
`csmSensorModel::getPlatformIdentifier`,  
`csmSensorModel::getTrajectoryIdentifier`,  
`csmSensorModel::getSensorIdentifier`

## NAME

5.12.48 `csmSensorModel::isParameterShareable()`

## SYNOPSIS

```
virtual CSMWarning* isParameterShareable(  
const int&          index,  
              bool&          shareable)  
              const throw    (CSMError) = 0;
```

## DESCRIPTION

The `isParameterShareable()` function returns a `shareable` flag to indicate whether or not the sensor model parameter adjustments are shareable across images for the sensor model adjustable parameter referenced by `index`.

## INPUTS

`index` selects the sensor model adjustable parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`shareable` is Boolean flag identifying whether or not the adjustments are shareable across images.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

### – Warnings:

`INDEX_OUT_OF_RANGE:`

`index` is less than zero or greater than or equal to the value returned by `getNumParameters()`.

`UNKNOWN_WARNING:`

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

None.

## SEE ALSO

`csmSensorModel::getNumParameters,`  
`csmSensorModel::getParameterSharingCriteria`

## NAME

5.12.49 `csmSensorModel::getParameterSharingCriteria()`

## SYNOPSIS

```
virtual CSMWarning* getParameterSharingCriteria(  
const int&          index,  
  
std::vector<csm_ParameterSharingCriteria>& criteria)  
    const throw     (CSMError) = 0;
```

## DESCRIPTION

The `getParameterSharingCriteria()` function returns characteristics to indicate how the sensor model adjustable parameter referenced by `index` may be shareable across images.

## INPUTS

`index` selects the sensor model adjustable parameter.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`criteria` is a STL vector of `csm_ParameterSharingCriteria` objects. Each object in the vector has a `Type()` method which can be used to determine type using one of the pre-defined types indicated below.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Warnings and Errors, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

### – Warnings:

`INDEX_OUT_OF_RANGE:`

index is less than zero or greater than or equal to the value returned by `getNumParameters()`.

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

When the sharing criteria is `SHARE_BY_DATE_TIME_MATCH` the sharing criteria is transitive regardless of the time limit parameter. For example, if a parameter is shared between images A and B and the same parameter is shared between images B and C, then the parameter must be shareable between images A and C.

Examples:

For a sample airborne platform with multiple sensors, a particular adjustable parameter for the sensor model may be shareable across sensor models, if the images were collected within 3 minutes of each other by sensors on the same aircraft. In this case, the vector returned by `getParameterSharingCriteria` would contain three `csm_ParameterSharingCriteria` objects as shown below. The order of these objects in the vector is not significant.

vector element	Type() returns	Time() returns
0	SHARE_BY_DATE_TIME_MATCH	180.0
1	SHARE_BY_SENSOR_ID	0.0
2	SHARE_BY_PLATFORM_ID	0.0

For a sample orbital platform with multiple sensors, a particular adjustable parameter for the sensor model may be shareable if the images were collected within 10 minutes of each other, with the same collection ID, by the same platform and sensor model and when the trajectories are the same. In this case, the vector would contain six objects as shown below. Again, the order of these objects in the vector is not significant.

vector element	Type() returns	Time() returns
0	SHARE_BY_DATE_TIME_MATCH	600.0
1	SHARE_BY_MODEL_NAME	0.0
2	SHARE_BY_SENSOR_ID	0.0
3	SHARE_BY_PLATFORM_ID	0.0

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

4	SHARE_BY_TRAJECTORY_ID	0.0
5	SHARE_BY_COLLECTION_ID	0.0
6	SHARE_BY_PASS_ID	0.0
7	SHARE_BY_BLOCK_ID	0.0

**SEE ALSO**

`csmSensorModel::getNumParameters,`  
`csmSensorModel::isParameterShareable`

5.12.50 CSMSensorModel::getVersion()

## SYNOPSIS

```
virtual CSMWarning* getVersion(  
    int &version )  
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getVersion()` method returns the CSM API version that the sensor model was written to. The CSM 3.00 API is specified by version equal to 3000. Use the static const `CURRENT_CSM_VERSION` defined in `CSMMisc.h` to return the current CSM API version that the sensor model is compiled to.

## INPUTS

None

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of `NULL` indicates no warning is present.

`version` is an integer describing the software version. The first digit indicates a major release, the following digit represent minor updates.

`CSMError` returns the CSM Error object. A thrown `CSMError` is a terminal condition and must be caught by the SET program.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4, CSM Errors and Warnings, must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

– Warnings:

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

– Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

5.12.51 `csmSensorModel::getSensorTypeAndMode ()`

## SYNOPSIS

```
virtual CSMWarning* getSensorTypeAndMode (
    CSMSensorTypeAndMode&
        sensorTypeAndMode)
    const throw (CSMError) = 0;
```

## DESCRIPTION

The `getSensorTypeAndMode()` function returns two pieces of information, the type, and collection mode. Sensor Types and Assumptions are described in the CSM TRD Section 1.4. To summarize, type refers to the method or phenomenology by which the imaging sensor collects an image, for example electro-optical sensors detect reflected light while radar sensor detect reflected microwave radiation resulting from active transmission of radar energy. Within a sensor type, the mode further refines the sensor description. For example, electro-optical sensors can collect an entire “frame” of pixels at once, i.e. framing mode, while other EO sensors employ the motion of the platform to move an array of detectors across the ground, i.e. pushbroom mode. Details on sensor types and modes can be found in Appendix A of the CSM TRD.

## INPUTS

None.

## OUTPUTS

`CSMWarning` is a pointer to a CSM warning object. A value of NULL indicates no warning is present.

`sensorTypeAndMode` is an object defined in `CSMSensorModel.h` and shown below.

```
class CSMSensorTypeAndMode
{
public:
    CSMSensorTypeAndMode (const std::string& sensorType,
                          const std::string&
sensorMode)
    {
        theSensorType = sensorType;
        theSensorMode = sensorMode;
    }
}
```

```
CSMSensorTypeAndMode() {}

CSMSensorTypeAndMode & operator=(const CSMSensorTypeAndMode&x)
{
    this->theSensorType = x.type();
    this->theSensorMode = x.mode();
    return *this;
}

const std::string& type() const {return theSensorType;}
const std::string& mode() const {return theSensorMode;}

private:
    std::string theSensorType;
    std::string theSensorMode;
}
```

As shown, this object contains both the sensor type and mode as strings which are populated by the sensor model. For convenience, predefined combinations of common sensor mode and type are provided in CSMSensorModel.h.

## ERRORS & WARNINGS

All errors and warnings documented in section 6.4 must be handled. The following lists of warnings and errors are reasonably likely to occur from this call.

### – Warnings:

INDEX\_OUT\_OF\_RANGE:

index is less than zero or greater than or equal to the value returned by  
getNumParameters().

UNKNOWN\_WARNING:

Use this warning if no other warning is suitable. Its use is discouraged.

### – Errors:

UNKNOWN\_ERROR:

Use this error if no other error is suitable. Its use is discouraged.

## NOTES

Predefined combinations of sensor mode and type are defined in CSMSensorModel.h. When a sensor model is one of these types/modes, the method may be implemented as shown below for example.

```
CSMSensorModel::CSMWarning *getSensorTypeAndMode(  
    CSMSensorTypeAndMode &x) const  
{  
    x = EO_FRAME;  
    return NULL;  
};
```

Add a new class, CSMSensorTypeAndMode to CSMSensorModel.h as defined here:

```
class CSMSensorTypeAndMode  
{  
public:  
    CSMSensorTypeAndMode (const std::string& sensorType,  
                           const std::string& sensorMode)  
    {  
        theSensorType = sensorType;  
        theSensorMode = sensorMode;  
    }  
  
    CSMSensorTypeAndMode() {}  
  
    CSMSensorTypeAndMode & operator=(const CSMSensorTypeAndMode&  
x)  
    {  
        this->theSensorType = x.type();  
        this->theSensorMode = x.mode();  
        return *this;  
    }  
  
    const std::string& type() const {return theSensorType;}  
    const std::string& mode() const {return theSensorMode;}  
  
private:  
    std::string theSensorType;  
    std::string theSensorMode;  
}
```

Add “canned” types and modes to the CSMSensorModel.cc code as shown here:

```
#define TYPE_UNK "UNKNOWN";
#define TYPE_EO "EO";
#define TYPE_IR "IR";
#define TYPE_MWIR "MWIR";
#define TYPE_LWIR "LWIR";
#define TYPE_SAR "SAR";
#define TYPE_EOIRSC "EO_IR_SPECIAL_CASE";
#define MODE_FRAME "FRAME";
#define MODE_PULSE "PULSE";
#define MODE_PB "PUSHBROOM";
#define MODE_WB "WHISKBROOM";
#define MODE_SPOT "SPOT";
#define MODE_STRIP "STRIP";
#define MODE_SCAN "SCAN";
#define MODE_VIDEO "VIDEO";
#define MODE_BODY_POINTING "BODY_POINTING";
```

For convenience, add static definitions of common sensor type/mode combinations to the CSMSensorModel.h header as shown below.

```
static const CSMSensorTypeAndMode EO_FRAME(TYPE_EO, MODE_FRAME);
static const CSMSensorTypeAndMode EO_PUSHBROOM(TYPE_EO, MODE_PB);
static const CSMSensorTypeAndMode EO_WHISKBROOM(TYPE_EO, MODE_WB);
static const CSMSensorTypeAndMode EO_VIDEO(TYPE_EO, MODE_VIDEO);
static const CSMSensorTypeAndMode MWIR_FRAME(TYPE_MWIR, MODE_FRAME);
static const CSMSensorTypeAndMode LWIR_WHISKBROOM(TYPE_LWIR, MODE_WB);
static const CSMSensorTypeAndMode SAR_SPOT(TYPE_SAR, MODE_SPOT);
static const CSMSensorTypeAndMode SAR_STRIP(TYPE_SAR, MODE_STRIP);
static const CSMSensorTypeAndMode SAR_SCAN(TYPE_SAR, MODE_SCAN);
static const CSMSensorTypeAndMode EO_BODY_POINTING(TYPE_EO, MODE_BODY_POINTING);
```

(Note: EO\_BODY\_POINTING refers to agile sensors that can operate in either pushbroom, whiskbroom, or “pointing” mode in which a specified direction is scanned. Examples include GeoEye1 and GeoEye 2)

SEE ALSO  
None

### 5.13 Error Control

Two kinds of operational status are reported back to the calling application: errors and warnings.

An error is a condition that prevents the completion of a function. The value of any return variable is undefined when an error occurs.

A warning is a condition that allows a function to complete, but indicates that the course of action may not have been carried out as expected. Values of return variables may be suspect.

All CSM API functions include as a return an instance of the `CSMWarning` class. This class can return the following data via accessor methods:

- `WarningType`: An enumeration type indicating the nature of the warning. Warnings will be enumerated with positive values starting at 1. Known warnings will have an enumerated value in the header file pre-defined to the text identified in the various functions. As the software is developed the known warnings will be added to the end of their list.
- A string containing a detailed warning message.
- A string indicating the function that produced the warning.

The `WarningType` enumeration type is defined in section 6.4.1 `CSMWarning`.

When there is no warning condition on a method then a `NULL` pointer must always be returned.

All CSM API functions include a thrown instance of the `CSMError` class. This class can return the following data via accessor methods:

- `ErrorType`: An enumeration type indicating the nature of the error. Errors will be enumerated with positive values starting at 1. Known errors will have an enumerated value in the header file pre-defined to the text identified in the various functions. As the software is developed the known errors will be added to the end of their list.
- A string containing a detailed error message.
- A string indicating the function that produced the error.

The `ErrorType` enumeration type is defined in section 6.4.2 `CSMError`.

The various API interfaces provide a SET with various options for handling errors and warnings returned by the API. For example, consider the status checking from the following API call:

```
Status = CSMPugin::convertISDToSensorModelState (
                                                    ISD,
                                                    SensorModelName,
                                                    SensorModelState);
```

The following example represents a method for handling the status checking.

```
try {
    Status = CSMPugin::convertISDToSensorModelState (
                                                    ISD,
                                                    SensorModelName,
                                                    SensorModelState);

    } catch (CSMError::ErrorType error)
        // An error occurred
        {
            ...
        }

if(Status == NULL)
// Success
    {
        ...
    }

else
// A warning occurred
    {if (Status.getWarning() ==...
    }
```

The following table contains description conditions for each of the defined warning enumerations:

**Table 7 - Warnings**

Warning Name	Description
UNKNOWN_WARNING	A warning was issued that was not contained on this list.
DATA_NOT_AVAILABLE	Additional data is needed.
PRECISION_NOT_MET	Achieved precision is greater (less precise) than desired precision.
NEGATIVE_PRECISION	Desired precision is negative.
IMAGE_COORD_OUT_OF_BOUNDS	The image coordinate (line and/or sample) is not in the imaging operation.
IMAGE_ID_TOO_LONG	String input is too long.
NO_INTERSECTION	Intersection (i.e. <code>imageToGround()</code> ) computed the closest approach rather than true intersection.
DEPRECATED_FUNCTION	This warning indicates the requested method is no longer available.

The following table contains description conditions for each of the defined error enumerations:

**Table 8 - Errors**

Error Name	Description
ALGORITHM	TBD
BOUNDS	One or more of the input parameters had a value outside the valid range.
FILE_READ	A file read error occurred
FILE_WRITE	A file write error occurred
ILLEGAL_MATH_OPERATION	TBD ( <i>When would this be needed? Shouldn't there be bounds checking for data values that would prevent illegal math?</i> )
INDEX_OUT_OF_RANGE	One or more of the input parameters had an index value outside the valid range.
INVALID_SENSOR_MODEL_STATE	The input sensor model state is invalid for this sensor model.
INVALID_USE	TBD
ISD_NOT_SUPPORTED	The input ISD contains data not supported for this sensor model.
MEMORY	Insufficient system memory exists to perform the desired operation.
SENSOR_MODEL_NOT_CONSTRUCTIBLE	The sensor model could not be constructed with the input data (ISD or state) provided.
SENSOR_MODEL_NOT_SUPPORTED	The sensor model name indicated is not supported by this CSMPugin.
UNKNOWN_ERROR	An unknown error condition has occurred.
UNSUPPORTED_FUNCTION	Requested API function not supported by this CSMPugin or CSMSensorModel.
UNKNOWN_SUPPORT_DATA	The input ISD contains insufficient support data to

## 5.14 Memory Management

Since a SET may open multiple images simultaneously, it should be expected for a given plugin to initialize the same sensor model multiple times, creating multiple `csmSensorModel::objects`. The construction of the plugin and the model(s) contained therein will support instantiation of multiple independent models, including multiple independent models using the same image as input. This will require that the plugin shared object to be able to operate with a SET that may be multi-process and/or multi threaded; however, it is the responsibility of the SET for managing the multi-process/multi-thread environment if the SET is so designed

This section references functions that need to be addressed in section 5.10 Detailed Plug-in Method Descriptions and section 5.12 Sensor Model Functions.

All output parameters shall be allocated by the calling application except for:

A `CMSensorModel` object returned from:

```
CSMPlugin::constructSensorModelFromState()
```

A `CMSensorModel` object returned from:

```
CSMPlugin::constructSensorModelFromISD()
```

An array of covariance model parameters returned from:

```
csmSensorModel::getCovarianceModel ()
```

In addition, many functions have a `csmWarning*` return type. In normal operation, the function will return a NULL pointer. However, when a condition arises that calls for a warning to be issued, the memory for the warning is allocated by the function issuing the warning.

The return `csmWarning` parameters, which are allocated within the function, will be done so using `new()`. They must be deallocated by the calling application using `delete()`.

## 6 APPENDIX A HEADER FILES

**Warning: Extracting data from a Microsoft Word document to create code may introduce errors. Use the source code from the VTS distribution. In the event of a conflict the source code distribution take precedence over code examples in this document.**

### 6.1 CSMPugin.h

```
//#####  
//  
// FILENAME: CSMPugin.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for abstract base class that is to provide a common interface from  
// which all Tactical Sensor Model (CSM) plugin factories shall inherit.  
//  
// LIMITATIONS: None  
//  
// SOFTWARE HISTORY:  
//  
// Date Author Comment  
// -----  
// May-2003 J. Olson Received initial version from BAE.  
// 20-Jun-2003 KFM Revised to incorporate plugin list and automatic  
// registration for derived types.  
// 01-Jul-2003 KFM Updated signatures.  
// 06-Feb-2004 KRW Incorporates changes approved by  
// the January and February 2004  
// configuration control board.  
// 08-JUN-2004 TWC API 3.1  
// 19-Aug-2004 PW Add throws  
//  
// NOTES:  
//  
// Initial coding of this class was accomplished by BAE Corporation. This  
// version contains modifications by Harris Corporation with the primary  
// goal of altering the method by which derived factories are "registered"  
// with the base plugin class.  
//  
// To use this for a plugin, the developer must simply inherit from this  
// class providing, at least, implementation for each pure virtual function.  
// In order to allow the plugin to self-register itself and be recognized  
// by the system as a "plugin", a static instance of the derived class must  
// invoke the CSMPugin constructor.  
//  
//#####  
#ifndef __CSMPLUGIN_H  
#define __CSMPLUGIN_H  
  
#ifdef _WIN32  
#pragma warning( disable : 4290 )  
#endif
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
#include <list>
#include <string>
#include "CSMImageSupportData.h"
#include "CSMMisc.h"
#include "CSMError.h"

class CSMWarning;
class CSMSensorModel;
class csm_ISD;

//-----
// This is an example factory for plug ins. In the real world, we might have
// multiple different classes in each shared library that are made to work
// together. All these classes must be created by this factory class.
//-----

class CSM_EXPORT_API CSMPugin
{
public:

    //-----
    // Types
    //-----

    typedef std::list < const CSMPugin* > CSMPuginList;

    class Impl;

    //-----
    // Constructors/Destructor
    //-----

    virtual ~CSMPugin() {}

    //-----
    // List Managing Methods
    //-----

    static CSMWarning* getList(CSMPuginList*& aCSMPuginList)
        throw (CSMError);
    //>This method provides access to the list of all plugins that are
    // currently registered.
    //<

    static CSMWarning* findPugin(
        const std::string& puginName,
        CSMPugin*& aCSMPugin)
        throw (CSMError);

    // pre: None.
    // post: Returns a pointer to the first pugin found whose name is
    //      aName; returns NULL if no such pugin found.

    static CSMWarning* removePugin(
        const std::string& puginName)
        throw (CSMError);

    //-----
    // Pugin Interface
    //-----

```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
virtual CSMWarning* getPluginName(
    std::string& pluginName)
    const throw (CSMError) = 0;

    //>This method returns the character std::string that identifies the plugin.
    //<

//---
// CSM Plugin Descriptors
//---

virtual CSMWarning* getManufacturer(
    std::string& manufacturer_name)
    const throw (CSMError) = 0;

virtual CSMWarning* getReleaseDate(
    std::string& release_date)
    const throw (CSMError) = 0;

//This method returns the CSM API version that the plug-in was written to.
//
virtual CSMWarning* getCSMVersion(
    int& csmVersion) const
    // implementation must include the following code:
    // csmVersion = CURRENT_CSM_VERSION; //CURRENT_CSM_VERSION is defined in CSMMisc.h
    throw (CSMError) = 0;

//---
// Sensor Model Availability
//---

virtual CSMWarning* getNSensorModels(int& n_sensor_models) const throw (CSMError) = 0;

virtual CSMWarning* getSensorModelName(
    const int& sensor_model_index,
    std::string& sensor_model_name)
    const throw (CSMError) = 0;

//---
// Sensor Model Descriptors
//---

virtual CSMWarning* getSensorModelVersion(
    const std::string &sensor_model_name,
    int& version)
    const throw (CSMError) = 0;

//---
// Sensor Model Construction
//---

virtual CSMWarning* canSensorModelBeConstructedFromState(
    const std::string& sensor_model_name,
    const std::string& sensor_model_state,
    bool& constructible)
    const throw (CSMError) = 0;

virtual CSMWarning* canSensorModelBeConstructedFromISD(
    const csm_ISD& Image_support_data,
    const std::string& sensor_model_name,
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
    bool& constructible)
    const throw (CSMError) = 0;

virtual CSMWarning* constructSensorModelFromState(
    const std::string& sensor_model_state,
    CSMSensorModel*& sensor_model)
    const throw (CSMError) = 0;

virtual CSMWarning* constructSensorModelFromISD(
    const csm_ISD& image_support_data,
    const std::string& sensor_model_name,
    CSMSensorModel*& sensor_model)
    const throw (CSMError) = 0;

virtual CSMWarning* getSensorModelNameFromSensorModelState(
    const std::string& sensor_model_state,
    std::string& sensor_model_name)
    const throw (CSMError) = 0;

//---
// Image Support Data Conversions
//---

virtual CSMWarning* canISDBeConvertedToSensorModelState(
    const csm_ISD& image_support_data,
    const std::string& sensor_model_name,
    bool& convertible)
    const throw (CSMError) = 0;

virtual CSMWarning* convertISDToSensorModelState(
    const csm_ISD& image_support_data,
    const std::string& sensor_model_name,
    std::string& sensor_model_state)
    const throw (CSMError) = 0;

protected:

//-----
// Constructors
//-----

CSMPlugin();
    //>This special constructor is responsible for registering each plugin
    // by adding it to theList. It is invoked by a special derived class
    // constructor that is only used by the static instance of the derived
    // class. (Refer to the example plugins to see how this is accomplished.)
    //<

private:

//-----
// Data Members
//-----

    static CSMPluginList* theList;
    static Impl* theImpl;

}; // CSMPlugin

#endif // __CSMPLUGIN_H
```



## 6.2 CSMISD

### 6.2.1 CSMImageSupportData.h

```
//#####  
//  
// FILENAME: CSMImageSupportData.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the ISD base class. ISD is encapsulated in a C++ class for  
// transfer through the CSM interface. ISD is passed as a pointer to a  
// simple ISD base class (for example, csm_ISD *isd). i  
//  
// LIMITATIONS: None  
//  
//  
// SOFTWARE HISTORY: Date Author Comment  
// 01-Jul-2003 LMT Initial version.  
// 06-Feb-2004 KRW Incorporates changes approved by  
// January and February 2004  
// Configuration control board.  
//  
// NOTES:  
//  
//#####  
#ifndef __CSMIMAGESUPPORTDATA_H  
#define __CSMIMAGESUPPORTDATA_H  
  
#include <string>  
#include "CSMMisc.h"  
#ifdef _WIN32  
#pragma warning( disable : 4291 )  
#pragma warning( disable : 4251 )  
#endif  
class CSM_EXPORT_API csm_ISD  
{  
public:  
    csm_ISD() { _format = "UNKNOWN"; }  
    virtual ~csm_ISD(){ _format.erase(); }  
  
    void getFormat( std::string &format ) const { format = _format; }  
  
protected:  
    std::string _format;  
};  
  
#endif
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

6.2.2 CSMISDNITF21.h

```
//#####  
//  
// FILENAME: csm_ISDNITF21.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the NITF 2.1 ISD class derived from the csm_ISD base class.  
// ISD is encapsulated in a C++ class for transfer through the CSM  
// interface. ISD is passed as a pointer to a simple ISD base class  
// (for example, csm_ISD *isd).  
//  
// LIMITATIONS: None  
//  
//  
// SOFTWARE HISTORY: Date Author Comment  
// 01-Jul-2003 LMT Initial version.  
// 06-Feb-2004 KRW Incorporates changes approved by  
// January and February 2004  
// Configuration control board.  
// 01-Nov-2004 KRW October 2004 CCB  
// 08-Jan-2005 KRW Multi Image/Frame ? Administrative changes  
//  
// NOTES:  
//  
//#####  
#ifndef __CSM_ISDNITF21_H  
#define __CSM_ISDNITF21_H  
  
#include "CSMImageSupportData.h"  
#include "CSMISDNITF20.h"  
#include "CSMMisc.h"  
  
class CSM_EXPORT_API NITF_2_1_ISD : public csm_ISD  
{  
public:  
NITF_2_1_ISD()  
{ _format = "NITF2.1"; numTREs = 0; numImages = 0;  
fileTREs = NULL; images = NULL; numDESSs = 0; fileDESSs = NULL; }  
  
~NITF_2_1_ISD()  
{ delete [] images; delete [] fileTREs; delete [] fileDESSs;}  
  
std::string filename; // full path filename of NITF file. This is an optional field.  
  
std::string fileHeader;  
int numTREs;  
tre *fileTREs;  
int numDESSs;  
des *fileDESSs;  
int numImages;  
image *images;  
};  
  
#endif
```





NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

6.2.3 CSMISDNITF20.h

```
//#####  
//  
// FILENAME:   csm_ISDNITF20.h  
//  
// CLASSIFICATION:   Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the NITF 2.0 ISD class derived from the csm_ISD base class.  
// ISD is encapsulated in a C++ class for transfer through the CSM  
// interface. ISD is passed as a pointer to a simple ISD base class  
// (for example, csm_ISD *isd).  
//  
// LIMITATIONS:      None  
//  
//  
//          Date          Author   Comment  
// SOFTWARE HISTORY:  01-Jul-2003  LMT       Initial version.  
//                   06-Feb-2004  KRW       Incorporated changes approved by  
//                                     January and February configuration  
//                                     control board.  
//                   01-Oct-2004  KRW       October 2004 CCB  
// NOTES:  
//  
//#####  
#ifndef __csm_ISDNITF20_H  
#define __csm_ISDNITF20_H  
  
#include "CSMImageSupportData.h"  
#include "CSMMisc.h"  
  
class CSM_EXPORT_API des  
{  
public:  
  
    des()  
    {  
        desShLength   = 0;  
        desSh          = NULL;  
        desDataLength = 0;  
        desData        = NULL;  
    }  
  
    ~des()  
    {  
        clear();  
    }  
  
    void setDES  
    (  
        int  des_sh_length,  
        char *des_sh,  
        int  des_data_length,  
        char *des_data )  
    {  
        int i;  
  
        clear();  
    }  
};
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
    desShLength = des_sh_length;
    desSh       = new char[desShLength+1];

    for( i = 0; i < desShLength; i++ )
        desSh[i] = des_sh[i];

    desSh[desShLength] = '\\0'; // in case NULL termination is needed

    desDataLength = des_data_length;
    desData       = new char[desDataLength+1];

    for( i = 0; i < desDataLength; i++ )
        desData[i] = des_data[i];

    desData[desDataLength] = '\\0'; // in case NULL termination is needed
}

void clear()
{
    delete [] desSh;
    delete [] desData;
    desShLength = 0;
    desDataLength = 0;
}

int    desShLength;
char   *desSh;
long   desDataLength;
char   *desData;
};

class CSM_EXPORT_API tre
{
public:
    tre() { record = NULL; length = 0; name[0] = '\\0'; }
    ~tre() { delete [] record; }

    void setTRE( char *tre) // tre includes TRE name, length and data
    {
        int i;
        char lengthString[6];

        clear();
        for( i = 0; i < 6; i++ )
            name[i] = tre[i];

        // in case, NULL termination is needed
        name[6] = '\\0';

        for( i = 6; i < 11; i++ )
            lengthString[i-6] = tre[i];

        // in case, NULL termination is needed
        lengthString[5] = '\\0';

        length = atoi(lengthString);

        record = new char[length+1];
    }
};
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
        for( i = 11; i < length+11; i++ )
            record[i-11] = tre[i];

        // in case, NULL termination is needed
        record[length] = '\\0';
    }

void clear()
{ delete [] record; length = 0; name[0] = '\\0'; }

char *record;
char  name[7];
int   length;
};

class CSM_EXPORT_API image
{
public:
    image() { numTRES = 0; imageTRES = NULL; }
    ~image() { delete [] imageTRES; }

    std::string imageSubHeader;
    tre        *imageTRES;
    int        numTRES;
};

class CSM_EXPORT_API NITF_2_0_ISD : public csm_ISD
{
public:
    NITF_2_0_ISD()
        { _format = "NITF2.0"; numTRES = 0; numImages = 0;
          fileTRES = NULL; images = NULL; numDESSs = 0; fileDESSs = NULL; }
    ~NITF_2_0_ISD()
        { delete [] images; delete [] fileTRES; delete[] fileDESSs; }

    std::string filename; // full path filename of NITF file. This is an optional field.

    std::string fileHeader;
    int         numTRES;
    tre        *fileTRES;
    int         numDESSs;
    des        *fileDESSs;
    int         numImages;
    image      *images;
};

#endif
```

## 6.2.4 CSMISDByteStream.h

```
//#####  
//  
// FILENAME: csm_ISDByteStream.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the ByteStream ISD class derived from the csm_ISD base class.  
// ISD is encapsulated in a C++ class for transfer through the CSM  
// interface. This class is designed to hold ISD in a std::string of unspecified  
// format. The field _isd is set with the ISD.  
//  
// LIMITATIONS: None  
//  
//  
// SOFTWARE HISTORY: Date Author Comment  
// 01-Jul-2003 LMT Initial version.  
// 06-Feb-2004 KRW Incorporates changes approved by  
// January and February 2004  
// Configuration control board.  
//  
// NOTES:  
//  
//#####  
#ifndef __csm_ISDBYTESTREAM_H  
#define __csm_ISDBYTESTREAM_H  
  
#include "CSMImageSupportData.h"  
#include <string>  
#include "CSMMisc.h"  
#ifdef _WIN32  
#pragma warning( disable : 4291 )  
#pragma warning( disable : 4251 )  
#endif  
  
class CSM_EXPORT_API bytestreamISD : public csm_ISD  
{  
public:  
    bytestreamISD() { _format = "BYTESTREAM"; }  
    bytestreamISD(std::string filename);  
    ~bytestreamISD() { _format.erase(); _isd.erase(); }  
  
    std::string _isd;  
};  
  
#endif
```

## 6.2.5 CSMISDFilename.h

```
//#####  
//  
// FILENAME: csm_ISDFilename.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the Filename ISD class derived from the csm_ISD base class.  
// ISD is encapsulated in a C++ class for transfer through the CSM  
// interface. This class is designed allow a std::string indicating the name  
// of a file that contains ISD. The field _filename should be set to the  
// full path name of the file.  
//  
// LIMITATIONS: None  
//  
//  
// SOFTWARE HISTORY: Date Author Comment  
// 01-Jul-2003 LMT Initial version.  
// 06-Feb-2004 KRW Incorporates changes approved by  
// January and February 2004  
// Configuration control board.  
// NOTES:  
//  
//#####  
#ifndef __csm_ISDFILENAME_H  
#define __csm_ISDFILENAME_H  
  
#include "CSMImageSupportData.h"  
#include <string>  
#include "CSMMisc.h"  
#ifdef _WIN32  
#pragma warning( disable : 4291 )  
#pragma warning( disable : 4251 )  
#endif  
  
class CSM_EXPORT_API filenameISD : public csm_ISD  
{  
public:  
filenameISD() { _format = "FILENAME"; }  
~filenameISD() { }  
  
std::string _filename;  
};  
  
#endif
```

### 6.3 CSMSensorModel.h

```
//#####
//
// FILENAME:          CSMSensorModel.h
//
// CLASSIFICATION:    Unclassified
//
// DESCRIPTION:
//
// Header for abstract base class that is to provide a common interface from
// which all Tactical Sensor Model (CSM) plugin models will inherit.
//
// LIMITATIONS:      None
//
// SOFTWARE HISTORY:  Date          Author Comment
//                   27-Jun-2003   LMT   Initial version.
//                   01-Jul-2003   LMT   Remove constants, error/warning
//                                     and make methods pure virtual.
//                                     CharType enum.
//                   31-Jul-2003   LMT   Change calls with a "&" to a "*",
//                                     combined CharType with ParamType
//                                     to create Param_CharType, //reordered
//                                     methods to match API order, added
//                                     systematic error methods.
//                   06-Aug-2003   LMT   Removed all Characteristic calls.
//                   08-Oct 2003   LMT   Added getImageSize calls
//                   06-Feb-2004   KRW   Incorporates changes approved by
//                                     January and February 2004
//                                     configuration control board.
//                   30-Jul-2004   PW    Initail API 3.1 version
//                   01-Nov-2004   PW    October 2004 CCB
//                   22 Oct 2010   DSL   CCB Change add
getCurrentCrossCovarianceMatrix
//                                     and
getOriginalCrossCovarianceMatrix
//                   22 Oct 2010   DSL   CCB Change add
getCurrentCrossCovarianceMatrix
//                                     and
getOriginalCrossCovarianceMatrix
//                   25 Oct 2010   DSL   CCB Change add
getNumGeometricCorrectionSwitches,
//                                     getGeometricCorrectionName,
//
getCurrentGeometricCorrectionSwitch,
//                                     and
setCurrentGeometricCorrectionSwitch
//                   25 Oct 2010   DSL   CCB Change add
getNumGeometricCorrectionSwitches,
//                                     getGeometricCorrectionName,
//
getCurrentGeometricCorrectionSwitch,
//                                     and
setCurrentGeometricCorrectionSwitch

// NOTES:
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
//  
//#####  
#ifndef _WIN32  
#pragma warning( disable : 4290 )  
#endif  
// remove ";" from end of each #define  
#ifndef __CSMSENSORMODEL_H  
#define __CSMSENSORMODEL_H  
  
#include <vector>  
#include "CSMMisc.h"  
#include "CSMWarning.h"  
#include "CSMError.h"  
#include "CSMParameterSharing.h"  
  
#include <string>  
  
class CSMSensorTypeAndMode  
{  
public:  
    CSMSensorTypeAndMode() {}  
    CSMSensorTypeAndMode (const std::string sensorType,  
                          const std::string sensorMode)  
    {  
        theSensorType = sensorType;  
        theSensorMode = sensorMode;  
    }  
  
    CSMSensorTypeAndMode & operator=(const CSMSensorTypeAndMode& x)  
    {  
        this->theSensorType = x.type();  
        this->theSensorMode = x.mode();  
        return *this;  
    }  
  
    const std::string& type() const {return theSensorType;}  
    const std::string& mode() const {return theSensorMode;}  
  
private:  
    std::string theSensorType;  
    std::string theSensorMode;  
};  
  
#define TYPE_UNK "UNKNOWN"  
#define TYPE_EO "EO"  
#define TYPE_IR "IR"  
#define TYPE_MWIR "MWIR"  
#define TYPE_LWIR "LWIR"  
#define TYPE_SAR "SAR"  
#define TYPE_EOIRSC "EO_IR_SPECIAL_CASE"  
#define MODE_FRAME "FRAME"  
#define MODE_PULSE "PULSE"  
#define MODE_PB "PUSHBROOM"  
#define MODE_WB "WHISKBROOM"  
#define MODE_SPOT "SPOT"  
#define MODE_STRIP "STRIP"  
#define MODE_SCAN "SCAN"  
#define MODE_VIDEO "VIDEO"  
#define MODE_BODY_POINTING "BODY_POINTING"  
  
static const CSMSensorTypeAndMode EO_FRAME(TYPE_EO, MODE_FRAME);
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
static const CSMSensorTypeAndMode EO_PUSHBROOM(TYPE_EO, MODE_PB); //EO_ -> EO,  
static const CSMSensorTypeAndMode EO_WHISKBROOM(TYPE_EO, MODE_WB); //EO_ -> EO,  
static const CSMSensorTypeAndMode EO_VIDEO(TYPE_EO, MODE_VIDEO);  
static const CSMSensorTypeAndMode MWIR_FRAME(TYPE_MWIR, MODE_FRAME);  
static const CSMSensorTypeAndMode LWIR_WHISKBROOM(TYPE_LWIR, MODE_WB);  
static const CSMSensorTypeAndMode SAR_SPOT(TYPE_SAR, MODE_SPOT);  
static const CSMSensorTypeAndMode SAR_STRIP(TYPE_SAR, MODE_STRIP);  
static const CSMSensorTypeAndMode SAR_SCAN(TYPE_SAR, MODE_SCAN);  
static const CSMSensorTypeAndMode BODY_POINTING(TYPE_EO, MODE_BODY_POINTING);
```

```
class CSM_EXPORT_API CSMSensorModel  
{  
public:
```

```
//-----  
// Constructors/Destructor  
//-----
```

```
CSMSensorModel()  
{  
    //EO_FRAME = CSMSensorTypeAndMode(TYPE_EO, MODE_PB);  
}
```

```
virtual ~CSMSensorModel() { }
```

```
//-----  
// Modifier  
//-----
```

```
//---  
// Core Photogrammetry  
//---
```

```
virtual CSMWarning* groundToImage(  
    const double& x,  
    const double& y,  
    const double& z,  
    double& line,  
    double& sample,  
    double& achieved_precision,  
    const double& desired_precision = 0.001)  
const throw (CSMError) = 0;
```

```
//> The groundToImage() method converts x, y and z (meters) in ground  
// space (ECEF) to line and sample (pixels) in image space.  
//<
```

```
virtual CSMWarning* groundToImage(  
    const double& x,  
    const double& y,  
    const double& z,  
    const double groundCovariance[9],  
    double& line,  
    double& sample,  
    double imageCovariance[4],  
    double& achieved_precision,  
    const double& desired_precision = 0.001)  
const throw (CSMError) = 0;
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
    //> This method converts a given ground point into line and sample
    // (pixels) in image space and returns accuracy information
    // associated with the image and ground coordinates.
    //<

virtual CSMWarning* imageToGround(
    const double& line,
    const double& sample,
    const double& height,
    double& x,
    double& y,
    double& z,
    double& achieved_precision,
    const double& desired_precision = 0.001)
const throw (CSMError) = 0;

    //> This method converts a given line and sample (pixels) in image
    // space to a ground point.
    //<

virtual CSMWarning* imageToGround(
    const double& line,
    const double& sample,
    const double imageCovariance[4],
    const double& height,
    const double& heightVariance,
    double& x,
    double& y,
    double& z,
    double groundCovariance[9],
    double& achieved_precision,
    const double& desired_precision = 0.001)
const throw (CSMError) = 0;

    //> This method converts a given line and sample (pixels) in //image space
    // to a ground point and returns accuracy information associated with
    // the image and ground coordinates.
    //<

virtual CSMWarning* imageToProximateImagingLocus(
    const double& line,
    const double& sample,
    const double& x,
    const double& y,
    const double& z,
    double locus[6],
    double& achieved_precision,
    const double& desired_precision = 0.001)
const throw (CSMError) = 0;

    //> The imageToProximateImagingLocus() method computes a proximate
    // imaging locus, a vector approximation of the imaging locus for the
    // given line and sample nearest the given x, y and z or at the given
    // height. The precision of this calculation refers to the locus's
    // origin and does not refer to the locus's orientation.
    //<

virtual CSMWarning* imageToRemoteImagingLocus(
    const double& line,
    const double& sample,
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
        double locus[6],
        double& achieved_precision,
        const double& desired_precision = 0.001)
const throw (CSMError) = 0;

//> The imageToRemoteImagingLocus() method computes locus, a vector
// approximation of the imaging locus for the given line and sample.
// The precision of this calculation refers only to the origin of the
// locus vector and does not refer to the locus's orientation. For an
// explanation of the remote imaging locus, see the section at the
// beginning of this document.
//<

//---
// Uncertainty Propagation
//---

virtual CSMWarning* computeGroundPartials(
        const double& x,
        const double& y,
        const double& z,
        double partials[6])
    throw (CSMError) = 0;

//> The computeGroundPartials method calculates the partial
// derivatives (partials) of image position (both line and sample)
// with respect to ground coordinates at the given ground
// position x, y, z.
// Upon successful completion, computeGroundPartials() produces the
// partial derivatives as follows:
//
// partials [0] = line wrt x
// partials [1] = line wrt y
// partials [2] = line wrt z
// partials [3] = sample wrt x
// partials [4] = sample wrt y
// partials [5] = sample wrt z
//<

virtual CSMWarning* computeSensorPartials(
        const int& index,
        const double& x,
        const double& y,
        const double& z,
        double& line_partial,
        double& sample_partial,
        double& achieved_precision,
        const double& desired_precision = 0.001)
    throw (CSMError) = 0;

virtual CSMWarning* computeSensorPartials(
        const int& index,
        const double& line,
        const double& sample,
        const double& x,
        const double& y,
        const double& z,
        double& line_partial,
        double& sample_partial,
        double& achieved_precision,
        const double& desired_precision = 0.001)
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
        throw (CSMError) = 0;

    //> The computeSensorPartials() method calculates the partial
    // derivatives of image position (both line and sample) with
    // respect to the given sensor parameter (index) at the given
    // ground position.
    // Two versions of the method are provided. The first method,
    // computeSensorPartials(), takes in only necessary information.
    // It performs groundToImage() on the ground coordinate and then
    // calls the second form of the method with the obtained line
    // and sample. If the calling function has already performed
    // groundToImage with the ground coordinate, it may call the second
    // method directly since it may be significantly faster than the
    // first. The results are unpredictable if the line and sample
    // provided do not correspond to the result of calling // //groundToImage()
    // with the given ground position (x, y, and z).
    //<

virtual CSMWarning* computeAllSensorPartials(
    const double&      x,
    const double&      y,
    const double&      z,
    std::vector<double>& line_partials,
    std::vector<double>& sample_partials,
    double&            achieved_precision,
    const double&      desired_precision = 0.001)
    throw (CSMError) = 0;

virtual CSMWarning* computeAllSensorPartials(
    const double&      line,
    const double&      sample,
    const double&      x,
    const double&      y,
    const double&      z,
    std::vector<double>& line_partials,
    std::vector<double>& sample_partials,
    double&            achieved_precision,
    const double&      desired_precision = 0.001)
    throw (CSMError) = 0;

    //> The computeAllSensorPartials() function calculates the
    // partial derivatives of image position (both line and sample)
    // with respect to each of the adjustable parameters at the
    // given ground position.
    //>

virtual CSMWarning* getCurrentParameterCovariance(
    const int& index1,
    const int& index2,
    double& covariance)
    const throw (CSMError) = 0;

    //> The getCurrentParameterCovariance() method
    // returns the covariance of the specified parameter pair
    // (index1, index2). The variance of the given parameter can be
    // obtained by using the same value for index1 and index2.
    //<

virtual CSMWarning* setCurrentParameterCovariance(
    const int& index1,
    const int& index2,
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
        const double& covariance)
throw (CSMError) = 0;

//> The setCurrentParameterCovariance() method is
// used to set the covariance value of the specified parameter pair.
//<

virtual CSMWarning* setOriginalParameterCovariance(
        const int& index1,
        const int& index2,
        const double& covariance)
throw (CSMError) = 0;

virtual CSMWarning* getOriginalParameterCovariance(
        const int& index1,
        const int& index2,
        double& covariance)
const throw (CSMError) = 0;

//> The first form of originalParameterCovariance() method sets
// the covariance of the specified parameter pair (index1, index2).
// The variance of the given parameter can be set using the same
// value for index1 and index2.
// The second form of originalParameterCovariance() method gets
// the covariance of the specified parameter pair (index1, index2).
// The variance of the given parameter can be obtained using the
// same value for index1 and index2.
//<

//---
// Time and Trajectory
//---

virtual CSMWarning* getTrajectoryIdentifier(
        std::string &trajectoryId)
const throw (CSMError) = 0;

//> This method returns a unique identifier to indicate which
// trajectory was used to acquire the image. This ID is unique for
// each sensor type on an individual path.
//<

virtual CSMWarning* getReferenceDateAndTime(
        std::string &date_and_time)
const throw (CSMError) = 0;

//> This method returns the time in seconds at which the specified
// pixel was imaged. The time provided is relative to the reference
// date and time given by the getReferenceDateAndTime() method and
// can be used to represent time offsets within the trajectory
// associated with the given image.
//<

virtual CSMWarning* getImageTime(
        const double& line,
        const double& sample,
        double& time)
const throw (CSMError) = 0;

//> The getImageTime() method returns the time in seconds at which
// the pixel specified by line and sample was imaged. The time
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
// provided is relative to the reference date and time given by
// getReferenceDateAndTime.
//<

virtual CSMWarning* getSensorPosition(
    const double& line,
    const double& sample,
    double& x,
    double& y,
    double& z)
    const throw (CSMError) = 0;

//> The getSensorPosition() method returns the position of
// the physical sensor at the given position in the image.
//<

virtual CSMWarning* getSensorPosition(
    const double& time,
    double& x,
    double& y,
    double& z)
    const throw (CSMError) = 0;

//> The getSensorPosition() method returns the position of
// the physical sensor at the given time of imaging.
//<

virtual CSMWarning* getSensorVelocity(
    const double& line,
    const double& sample,
    double& vx,
    double& vy,
    double &vz)
    const throw (CSMError) = 0;

//> The getSensorVelocity() method returns the velocity
// of the physical sensor at the given position in the image.
//<

virtual CSMWarning* getSensorVelocity(
    const double& time,
    double& vx,
    double& vy,
    double &vz)
    const throw (CSMError) = 0;

//> The getSensorVelocity() method returns the velocity
// of the physical sensor at the given time of imaging.
//<

//---
// Sensor Model Parameters
//---

virtual CSMWarning* setCurrentParameterValue(
    const int& index,
    const double& value)
    throw (CSMError) = 0;

//> The setCurrentParameterValue() is used to set the
// value of the adjustable parameter indicated by index.
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
//<

virtual CSMWarning* getCurrentParameterValue(
    const int& index,
    double& value)
    const throw (CSMError) = 0;

//> The getCurrentParameterValue() returns the value
// of the adjustable parameter given by index.
//<

virtual CSMWarning* getParameterName(
    const int& index,
    std::string& name)
    const throw (CSMError) = 0;

//> This method returns the name for the sensor model parameter
// indicated by the given index.
//<

virtual CSMWarning* getNumParameters(
    int& numParams)
    const throw (CSMError) = 0;

//> This method returns the number of sensor model parameters.
//<

virtual CSMWarning* setOriginalParameterValue(
    const int& index,
    const double& value)
    throw (CSMError) = 0;

//> The setOriginalParameterValue() method is
// used to set the original parameter value of the indexed
// parameter.
//<

virtual CSMWarning* getOriginalParameterValue(
    const int& index,
    double& value)
    const throw (CSMError) = 0;

//> The getOriginalParameterValue() method
// returns the value of the adjustable parameter given by
// index.
//<

virtual CSMWarning* getOriginalParameterType(
    const int& index,
    CSMMisc::Param_CharType &pType)
    const throw (CSMError) = 0;

//> The getOriginalParameterType() method returns the original
// type of the parameter given by index.
//<

virtual CSMWarning* getCurrentParameterType(
    const int& index,
    CSMMisc::Param_CharType &pType)
    const throw (CSMError) = 0;
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
    //> The getCurrentParameterType() method returns the current
    // type of the parameter given by index.
    //<

virtual CSMWarning* setOriginalParameterType(
    const int& index,
    const CSMMisc::Param_CharType &pType)
    throw (CSMError) = 0;

    //> The setOriginalParameterType() method sets the original
    // type of the parameter for the given by index.
    //<

virtual CSMWarning* setCurrentParameterType(
    const int& index,
    const CSMMisc::Param_CharType &pType)
    throw (CSMError) = 0;

    //> The setCurrentParameterType() method sets the current
    // type of the parameter for the given by index.
    //<

//---
// Sensor Model Information
//---

virtual CSMWarning* getPedigree(
    std::string &pedigree)
    const throw (CSMError) = 0;

    //> The getPedigree() method returns a character std::string that
    // identifies the sensor, the model type, its mode of acquisition
    // and processing path. For example, an image that could produce
    // either an optical sensor model or a cubic rational polynomial
    // model would produce different pedigrees for each case.
    //<

virtual CSMWarning* getImageIdentifier(
    std::string &imageId)
    const throw (CSMError) = 0;

    //> This method returns the unique identifier to indicate the imaging
    // operation associated with this sensor model.
    //<

virtual CSMWarning* setImageIdentifier(
    const std::string &imageId)
    throw (CSMError) = 0;

    //> This method sets the unique identifier for the image to which the
    // sensor model pertains.
    //<

virtual CSMWarning* getSensorIdentifier(
    std::string &sensorId)
    const throw (CSMError) = 0;

    //> The getSensorIdentifier() method returns sensorId to indicate
    // which sensor was used to acquire the image. This sensorId is
    // meant to uniquely identify the sensor used to make the image.
    //<
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
virtual CSMWarning* getPlatformIdentifier(
    std::string &platformId)
    const throw (CSMError) = 0;

    //> The getPlatformIdentifier() method returns platformId to indicate
    // which platform was used to acquire the image. This platformId
    // is meant to uniquely identify the platform used to collect the // //image.
    //<

virtual CSMWarning* setReferencePoint(
    const double &x,
    const double &y,
    const double &z)
    throw (CSMError) = 0;

    //> This method returns the ground point indicating the general
    // location
    // of the image.
    //<

virtual CSMWarning* getReferencePoint(
    double &x,
    double &y,
    double &z)
    const throw (CSMError) = 0;

    //> This method sets the ground point indicating the general location
    // of the image.
    //<

virtual CSMWarning* getSensorModelName(
    std::string &name)
    const throw (CSMError) = 0;

    //> This method returns a string identifying the name of the sensor model.
    //<

virtual CSMWarning* getImageSize(
    int& num_lines,
    int& num_samples)
    const throw (CSMError) = 0;

    //> This method returns the number of lines and samples in the imaging
    // operation.
    //<

    //---
    // Sensor Model State
    //---

virtual CSMWarning* getSensorModelState(
    std::string& state)
    const throw (CSMError) = 0;

    //> This method returns the current state of the model in an
    // intermediate form. This intermediate form can then be processed,
    // for example, by saving to file so that this model
    // can be instantiated at a later date. The derived SensorModel
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
// is responsible for saving all information needed to restore
// itself to its current state from this intermediate form.
// A NULL pointer is returned if it is not possible to save the
// current state.
//<

//---
// Monoscopic Mensuration
//---

virtual CSMWarning* getValidHeightRange(
    double& minHeight,
    double& maxHeight)
    const throw (CSMError) = 0;

//> The validHeightRange() method returns the minimum and maximum
// heights that describe the range of validity of the model. For
// example, the model may not be valid at heights above the heights
// of the sensor for physical models.
//<

virtual CSMWarning *getValidImageRange(
    double& minRow,
    double& maxRow,
    double& minCol,
    double& maxCol)
    const throw (CSMError) = 0;

//> The validImageRange() method returns the minimum and maximum
// values for image position (row and column) that describe the
// range of validity of the model. This range may not always match
// the physical size of the image. This method is used in
// conjunction with getValidHeightRange() to determine the full
// range of applicability of the sensor model.
//<

virtual CSMWarning* getIlluminationDirection(
    const double& x,
    const double& y,
    const double& z,
    double& direction_x,
    double& direction_y,
    double& direction_z)
    const throw (CSMError) = 0;

//> The getIlluminationDirection() method calculates the direction of
// illumination at the given ground position x, y, z.
//<

//---
// Error Correction
//---

virtual CSMWarning* getNumGeometricCorrectionSwitches(
    int& numSec)
    const throw (CSMError) = 0;

//> The numGeometricCorrections() method returns the number
// of geometric corrections defined for the sensor model.
//<
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
virtual CSMWarning* getGeometricCorrectionName(
    const int& index,
    std::string &name)
    const throw (CSMError) = 0;

    //> This method returns the name for the sensor model parameter
    // indicated by the given index.
    //<

virtual CSMWarning* setCurrentGeometricCorrectionSwitch(
    const int& index,
    const bool &value,
    const CSMMisc::Param_CharType& parameterType)
    throw (CSMError) = 0;

    //> The setCurrentGeometricCorrectionSwitch() is
    // used to set the switch of the geometric correction
    // indicated by index.
    //<

virtual CSMWarning* getCurrentGeometricCorrectionSwitch(
    const int& index,
    bool &value)
    const throw (CSMError) = 0;

    //> The getCurrentGeometricCorrectionSwitch()
    // returns the value of the geometric correction switch
    // given by index.
    //<

virtual CSMWarning* getCurrentCrossCovarianceMatrix(
    const int numSM,
    const CSMSensorModel** SMs,
    const double line1,
    const double sample1,
    const double* lines,
    const double* samples,
    double **&crossCovarianceMatrix,
    int &M)
    const throw (CSMError) = 0;

    //> The getCurrentCovarianceMatrix() function returns a matrix
    // containing all elements of the error cross covariance matrix
    // between the instantiated sensor model and a specified second
    // sensor model (SM2). This data supplies the data to compute
    // cross covariance between images. Images may be correlated
    // because they are taken by the same sensor or from sensors on
    // the same platform. Images may also be correlated due to post
    // processing of the sensor models. The data returned here may
    // need to be supplemented with the single image covariance from
    // getCurrentParameterCovariance() and getUnmodeledError().
    //<

virtual CSMWarning* getOriginalCrossCovarianceMatrix(
    const int numSM,
    const CSMSensorModel** SMs,
    const double line1,
    const double sample1,
    const double* lines,
    const double* samples,
    double **&crossCovarianceMatrix,
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
        int &M)
const throw (CSMError) = 0;

//> The getOriginalCovarianceMatrix() function returns a matrix
// containing all elements of the error cross covariance matrix
// between the instantiated sensor model and a specified second
// sensor model (SM2). Images may be correlated because they
// are taken by the same sensor or from sensors on the same
// platform. Images may also be correlated due to post
// processing of the sensor models. The data returned here may
// need to be supplemented with the single image covariance from
// getOriginalParameterCovariance() and getUnmodeledError().
//<

virtual CSMWarning* getUnmodeledError(
        const double line,
        const double sample,
        double covariance[4])
const throw (CSMError) = 0;

//> The getUnmodeledError() function gives a sensor specific
// error for the given input image point. The error is reported
// as the four terms of a 2x2 covariance mensuration error
// matrix. This error term is meant to map error terms that are
// not modeled in the sensor model to image space for inclusion
// in error propagation. The extra error is added to the
// mensuration error that may already be in the matrix.
//<

virtual CSMWarning* getUnmodeledCrossCovariance(
        const double pt1Line,
        const double pt1Sample,
        const double pt2Line,
        const double pt2Sample,
        double crossCovariance[4])
const throw (CSMError) = 0;

//> The getUnmodeledCrossCovariance function gives the cross
// covariance for unmodeled error between two image points on
// the same image. The error is reported as the four terms of
// a 2x2 matrix. The unmodeled cross covariance is added to
// any values that may already be in the cross covariance matrix.
//<

virtual CSMWarning* getCollectionIdentifier(
        std::string &collectionId)
const throw (CSMError) = 0;

//> This method returns a unique identifier that uniquely identifies
// a collection activity by a sensor platform. This ID will vary
// depending on the sensor type and platform.
//<

virtual CSMWarning* isParameterShareable(
        const int& index,
        bool& shareable)
const throw (CSMError) = 0;

//> This method returns a flag to indicate whether or not a sensor
// model parameter adjustments are shareable across images for the
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
// sensor model adjustable parameter referenced by index.
//<

/*
virtual CSMWarning* getParameterSharingCriteria(
    const int& index,
    bool& requireModelNameMatch,
    bool& requireSensorIDMatch,
    bool& requirePlatformIDMatch,
    bool& requireCollectionIDMatch,
    bool& requireTrajectoryIDMatch,
    bool& requireDateTimeMatch,
    double& allowableTimeDelta)
    const throw (CSMError) = 0;
*/
virtual CSMWarning* getParameterSharingCriteria(
    const int& index,
    std::vector<CSM_SHARING::csm_ParameterSharingCriteria>&
criteria)
    const throw (CSMError) = 0;

//> This method returns characteristics to indicate how
// the sensor model adjustable parameter referenced by index
// may be shareable accross images.
//<

virtual CSMWarning* getSensorTypeAndMode(
    CSMSensorTypeAndMode &sensorTypeAndMode)
    const throw (CSMError) = 0;

//> This method returns a flag to indicate whether or not a sensor
// ...
//<

virtual CSMWarning* getVersion( int &version )
    // implementation must include the following code:
    // version = CURRENT_CSM_VERSION; //CURRENT_CSM_VERSION is defined in CSMMisc.h
    const throw (CSMError) = 0;

#ifdef TESTAPIVERSION
    virtual CSMWarning* testAPIVersionSubclass(
        std::string &text)
        const throw (CSMError) = 0;

//> The testAPIVersionSubclass method provides a means to
// demonstrate and test the subclass backward compatibility
// for an API release. This method is not a member of a
// standard API compliant released sensor model, but is
// the sole addition to the API release that creates the
// API compliant subclassing test version of a sensor model
// release.
//<
#endif
};

#endif
```

## 6.4 CSM Warnings and Errors

### 6.4.1 CSMWarning.h

```
//#####  
//  
// FILENAME: CSMWarning.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the warning structure used by the CSM.  
//  
// LIMITATIONS: None  
//  
// Date Author Comment  
// SOFTWARE HISTORY: 1 June 2004 Kevin Lam CCB Change  
//  
// NOTES:  
//  
//#####  
  
#ifndef __CSMWARNING_H  
#define __CSMWARNING_H  
  
#include <string>  
#include "CSMMisc.h"  
#ifdef WIN32  
#pragma warning( disable : 4291 )  
#pragma warning( disable : 4251 )  
#endif  
class CSM_EXPORT_API CSMWarning  
{  
public:  
  
//-----  
// Warnings  
//-----  
  
enum WarningType  
{  
UNKNOWN_WARNING = 1,  
DATA_NOT_AVAILABLE,  
PRECISION_NOT_MET,  
NEGATIVE_PRECISION,  
IMAGE_COORD_OUT_OF_BOUNDS,  
IMAGE_ID_TOO_LONG,  
NO_INTERSECTION,  
DEPRECATED_FUNCTION  
};  
  
CSMWarning()  
{  
}
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
CSMWarning(  
    const WarningType& aWarningType,  
    const std::string& aMessage,  
    const std::string& aFunction)  
{  
    setCSMWarning( aWarningType, aMessage, aFunction );  
}  
  
WarningType    getWarning()    { return theWarning; }  
const std::string& getMessage() { return theMessage; }  
const std::string& getFunction() { return theFunction; }  
  
void setCSMWarning(  
    const WarningType& aWarningType,  
    const std::string& aMessage,  
    const std::string& aFunction)  
{  
    theWarning = aWarningType;  
    theMessage = aMessage;  
    theFunction = aFunction;  
}  
  
private:  
  
    WarningType theWarning;  
    //> enumeration of the warning (for application control),  
    //<  
    std::string theMessage;  
    //> string describing the warning.  
    //<  
    std::string theFunction;  
    //> string identifying the function in which the warning occurred.  
    //<  
};  
  
#endif // __CSMWARNING_H
```

## 6.4.2 CSMErrror.h

```
//#####  
//  
// FILENAME: CSMErrror.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the error structure used by the CSM.  
//  
// LIMITATIONS: None  
//  
//  
// Date Author Comment  
// SOFTWARE HISTORY: 1 June 2004 Kevin Lam CCB Change  
// 24 June 2005 Len Tomko CCB Change Added  
// DATA_NOT_AVAILABLE  
// 09 Mar 2010 Don Leonard CCB Change Deleted  
DATA_NOT_AVAILABLE  
//  
*****  
// NOTES:  
//  
//#####  
  
#ifndef __CSMErrror_H  
#define __CSMErrror_H  
  
#include <string>  
#include "CSMMisc.h"  
  
#ifdef WIN32  
#pragma warning( disable : 4291 )  
#pragma warning( disable : 4251 )  
#endif  
  
class CSM_EXPORT_API CSMErrror  
{  
public:  
  
//-----  
// Errors  
//-----  
  
enum ErrorType  
{  
ALGORITHM = 1,  
BOUNDS,  
FILE_READ,  
FILE_WRITE,  
ILLEGAL_MATH_OPERATION,  
INDEX_OUT_OF_RANGE,  
INVALID_SENSOR_MODEL_STATE,  
INVALID_USE,  
ISD_NOT_SUPPORTED,  
MEMORY,
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
    SENSOR_MODEL_NOT_CONSTRUCTIBLE,  
    SENSOR_MODEL_NOT_SUPPORTED,  
    STRING_TOO_LONG,  
    UNKNOWN_ERROR,  
    UNSUPPORTED_FUNCTION,  
    UNKNOWN_SUPPORT_DATA  
};  
  
CSMError()  
{  
}  
  
CSMError(  
    const ErrorType& aErrorType,  
    const std::string& aMessage,  
    const std::string& aFunction)  
{  
    setCSMError( aErrorType, aMessage, aFunction );  
}  
  
ErrorType      getError()      { return theError; }  
const std::string& getMessage() { return theMessage; }  
const std::string& getFunction() { return theFunction; }  
  
void setCSMError(  
    const ErrorType& aErrorType,  
    const std::string& aMessage,  
    const std::string& aFunction)  
{  
    theError      = aErrorType;  
    theMessage    = aMessage;  
    theFunction   = aFunction;  
}  
  
private:  
  
    ErrorType theError;  
    //> enumeration of the error (for application control),  
    //<  
    std::string theMessage;  
    //> string describing the error.  
    //<  
    std::string theFunction;  
    //> string identifying the function in which the error occurred.  
    //<  
};  
  
#endif // __CSMERROR_H
```

## 6.5 CSMMisc.h

```
//#####  
//  
// FILENAME: CSMMisc.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the constants and other definitions used by the CSM.  
//  
// LIMITATIONS: None  
//  
//  
// SOFTWARE HISTORY: Date Author Comment  
// SOFTWARE HISTORY: 01-Jul-2003 LMT Initial version.  
//  
// NOTES:  
//  
//#####  
#ifndef __CSMMISC_H  
#define __CSMMISC_H  
#ifdef _WIN32  
# ifdef CSM_LIBRARY  
# define CSM_EXPORT_API __declspec(dllexport)  
# else  
# define CSM_EXPORT_API __declspec(dllimport)  
# endif  
#else  
# define CSM_EXPORT_API  
#endif  
  
// The getVersion() and getCSMVersion() methods should use CURRENT_CSM_VERSION to  
// return the CSM API version that the sensor model/plugin was written to.  
// The CSM 3.00 API is specified by version equal to 3000.  
static const int CURRENT_CSM_VERSION = 3000;  
  
static const int TEMP_STRING_LENGTH = 2048;  
static const int MAX_NAME_LENGTH = 40;  
static const int MAX_FUNCTION_NAME_LENGTH = 80;  
static const int MAX_MESSAGE_LENGTH = 512;  
  
class CSM_EXPORT_API CSMMisc  
{  
public:  
  
//-----  
// Enumerations  
//-----  
enum Param_CharType  
{  
NONE,  
FICTITIOUS,  
REAL,  
EXACT };  
//>  
// This enumeration lists the possible parameter or characteristic
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
// types as follows.  
//  
// NONE      - Parameter value has not yet been initialized.  
// FICTITIOUS - Parameter value has been calculated by resection  
//            or other means.  
// REAL      - Parameter value has been measured or read from  
//            support data.  
// EXACT     - Parameter value has been specified and is assumed to  
//            have no uncertainty.  
//<  
  
};  
  
#endif
```



## 6.6 CSMPParameterSharing.h

```
//#####  
//  
// FILENAME: CSMPParameterSharing.h  
//  
// CLASSIFICATION: Unclassified  
//  
// DESCRIPTION:  
//  
// Header for the constants and other definitions used by the CSM Parameter  
// Sharing function.  
//  
// LIMITATIONS: None  
//  
// Date Author Comment  
// SOFTWARE HISTORY: 16 Sep 2010 Gene Rose Initial version.  
//  
// NOTES:  
//  
//#####  
#ifndef __CSMPPARAMETERSHARING_H  
#define __CSMPPARAMETERSHARING_H  
  
namespace CSM_SHARING {  
  
static const int NOT_INITIALIZED = 0;  
static const int SHARE_BY_MODEL_NAME = 1;  
static const int SHARE_BY_SENSOR_ID = 2;  
static const int SHARE_BY_PLATFORM_ID = 3;  
static const int SHARE_BY_COLLECTION_ID = 4;  
static const int SHARE_BY_TRAJECTORY_ID = 5;  
static const int SHARE_BY_DATE_TIME_MATCH = 6;  
  
// sharing criteria class is not virtual  
class CSM_EXPORT_API csm_ParameterSharingCriteria  
{  
public:  
// data access  
int Type() const {return m_type;};  
double Time() const {return m_time_limit;};  
  
// constructors  
csm_ParameterSharingCriteria(int t): m_type(t)  
{  
m_time_limit = 0.0;  
}  
csm_ParameterSharingCriteria(double t1)  
{  
m_type = CSM_SHARING::SHARE_BY_DATE_TIME_MATCH;  
m_time_limit = t1;  
}  
csm_ParameterSharingCriteria()  
{  
m_type = CSM_SHARING::NOT_INITIALIZED;  
m_time_limit = 0.0;  
}
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
    }  
protected:  
    int m_type;  
    double m_time_limit;  
};  
}; // namespace CSM_SHARING  
#endif // __CSMPARAMETER_SHARING_H
```



## 7 APPENDIX B ADDITIONAL EXPLANATION OF EXPORT SYMBOLS

### 7.1 Introduction

This section explains the purpose of the "EXAMPLE\_EXPORT\_API" identifier included in the example CSMPlugin derived class header file ExampleSMPlugin.h and, similarly, the "STUB\_EXPORT\_API" identifier in the StubSMPlugin.h header file.

Note that the following discussion applies only to Win32 platforms. By examining the logic, it can easily be seen that the symbols mentioned above evaluate to nothing on non-Win32 platforms.

### 7.2 Discussion

When creating a dynamic-link library (DLL), no symbol is marked for export by default. In effect, this hides the symbol in the DLL from any other DLL or application ("deliverable") that uses the DLL. This can be advantageous if multiple deliverables define the same symbol, as there will be no ambiguity of symbol references.

On the other hand, a DLL is not useful at all if it has no symbols marked for export. Therefore, when including a header file whose implementation is part of the currently-developed DLL, the preferred practice is to mark functions and data members of the public interface for export.

```
class Hello
{
public:
    void __declspec(dllexport) printHello();

private:
    void writeToScreen(const char*);
};
```

Furthermore, for every deliverable that uses that header file in its own code, those symbols need to be marked for import.

```
class Hello
{
public:
    void __declspec(dllimport) printHello();

private:
    void writeToScreen(const char*);
};
```

In addition, as stated above, on non-Win32 platforms, this is irrelevant and no marking needs to be done.

## NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
class Hello
{
public:
    void printHello();

private:
    void writeToScreen(const char*);
};
```

Note the similarities in the code above. For a particular DLL, only the `__declspec` qualifier needs to be changed for the various circumstances. Thus, we utilize the preprocessor for help:

```
#ifdef _WIN32
# ifdef HELLO_LIBRARY
#  define HELLO_EXPORT_API __declspec(dllexport)
# else
#  define HELLO_EXPORT_API __declspec(dllimport)
# endif
#else
# define HELLO_EXPORT_API
#endif
```

This will expand `HELLO_EXPORT_API` to the appropriate `__declspec` symbol only if it is compiled on a Win32 platform. When building the library, the code is compiled with `HELLO_LIBRARY` defined, possibly by placing the following in the source file for this class.

```
// -- Hello.cpp
#define HELLO_LIBRARY
#include "Hello.h"
```

Defining `HELLO_LIBRARY` causes `HELLO_EXPORT_API` to expand to `__declspec(dllexport)`, as it should when building the DLL; otherwise, it will expand to `__declspec(dllimport)`, as it should when building any deliverable that uses the DLL.

The following link has more information:

```
http://msdn.microsoft.com/library/default.asp?
url=/library/en-us/vccore98/html/
_core_using___declspec.28.dllimport.29_.and___declspec.28.dllexport.29.asp
```

## 8 APPENDIX C COMPILING

### 8.1 Sun Solaris Forte or Workshop compiler:

For compiling the shared object (.so) source files to an object files (.o), the -c -PIC -g -o switches are used. For compiling the .o files to .so files, the -G -PIC -g -o switches are used. The -G switch directs the link editor to produce a shared object. The -PIC switch emits position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. The -G switch includes all debugging symbols and is recommended for debugging/developer builds only, not for releases.

For example:

```
CC -c -PIC -g StubSMPlugin.cpp -o StubSMPlugin.o
CC -c -PIC -g StubSensorModel.cpp -o StubSensorModel.o
CC -G -PIC -g StubSMPlugin.o StubSensorModel.o -o StubSMPlugin.so
```

For compiling the main application source files to object files, the -c -g -o switches are used.

For example:

```
CC -c -g vts.cpp -o vts.o
```

To link the main application object to produce an executable, the -o -ldl switches are used. The -l switch causes the link editor to look for files named libdl.a or libdl.so, the shared object handling library and the file named libcstd.a or libcstd.so, the C standard library, in the library search path.

For example:

```
CC vts.o vts_misc.o vts_isd.o SManager.o compareParam.o, recordLog.o_cCstd _lcstd -o
vts -ldl
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

The following is an example Makefile that produces an executable called vts.

```
#####  
#  
#  
# FILENAME:   Makefile  
#  
# DESCRIPTION:  
#  
# This Makefile is used by make to build the vts application and  
# supporting code on Solaris using the WorkShop/Forte CC compiler.  
#  
# NOTES:  
#  
#  
#####  
  
CXX = CC  
  
# The flags needed to compile all  
CXXFLAGS = -PIC -g  
  
# we need to link in the dll handling library  
  
LDLIBS = -ldl -lCstd  
  
#-----  
--  
all : compareParam.o recordLog.o SManager.o vts_isd.o vts_misc.o vts  
  
# the main executable has to be linked with the -rdynamic flag  
# so the plug in libraries can call inherited methods and  
# access vttables in the main executable.  
  
#-----  
--  
vts : vts.o \  
      compareParam.o \  
      recordLog.o \  
      SManager.o \  
      vts_isd.o \  
      vts_misc.o \  
      CSMPugin.o  
      $(CXX) vts.o vts_misc.o vts_isd.o SManager.o compareParam.o \  
      recordLog.o CSMPugin.o -o vts $(LDLIBS)  
      rm -f vts  
      ln -s vts .  
  
# -rdynamic
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
#-----  
--  
vts.o : vts.cpp \  
    Makefile \  
    CSMPugin.h \  
    CSMEror.h \  
    CSMWarning.h \  
    CSMSensorModel.h \  
    CSMImageSupportData.h \  
    CSMISDNITF20.h \  
    CSMISDNITF21.h \  
    CSMISDFilename.h \  
    CSMISDByteStream.h \  
    SMManager.h \  
    VCSMisc.h  
    $(CXX) -c -g vts.cpp -o vts.o  
  
SMManager.o : SMManager.cpp \  
    Makefile \  
    SMManager.h \  
    CSMPugin.h \  
    CSMEror.h \  
    CSMWarning.h  
    $(CXX) -c -g SMManager.cpp -o SMManager.o  
  
CSMPugin.o : CSMPugin.cpp \  
    Makefile \  
    CSMEror.h \  
    CSMWarning.h \  
    CSMMisc.h \  
    CSMPugin.h  
    $(CXX) -c -g CSMPugin.cpp -o CSMPugin.o  
  
compareParam.o : compareParam.cpp \  
    Makefile \  
    CSMMisc.h \  
    VCSMisc.h  
    $(CXX) -c -g compareParam.cpp -o compareParam.o  
  
recordLog.o : recordLog.cpp \  
    Makefile \  
    CSMEror.h \  
    CSMWarning.h \  
    CSMISDFilename.h \  
    CSMISDByteStream.h \  
    CSMISDNITF20.h \  
    CSMISDNITF21.h \  
    VCSMisc.h  
    $(CXX) -c -g recordLog.cpp -o recordLog.o  
  
vts_isd.o : vts_isd.cpp \  
    Makefile \  
    CSMEror.h \  
    CSMWarning.h \  
    CSMImageSupportData.h \  

```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
CSMISDFilename.h \  
CSMISDByteStream.h \  
CSMISDNITF20.h \  
CSMISDNITF21.h \  
VCSMisc.h  
$(CXX) -c -g vts_isd.cpp -o vts_isd.o  
  
vts_misc.o      : vts_misc.cpp \  
Makefile \  
CSMError.h \  
CSMWarning.h \  
CSMImageSupportData.h \  
CSMPlugin.h \  
VCSMisc.h  
$(CXX) -c -g vts_misc.cpp -o vts_misc.o  
#-----  
--  
clean :  
-rm -f *.o vts *.so *~  
-rm -rf SunWS_cache
```

The following is an example MakeFile that produces a shared object file called StubSMPlugin.so .

```
#####  
#  
#  
# FILENAME:   Makefile  
#  
# DESCRIPTION:  
#  
# This Makefile is used by make to build the vts StubSMPlugin plugin  
# (shared object) on Solaris using the WorkShop/Forte CC compiler.  
#  
# NOTES:  
#  
#  
#####  
  
CXX = CC  
  
# The flags needed to compile all  
CXXFLAGS = -PIC -g  
  
# we need to link in the dll handling library  
  
LDLIBS = -ldl -lCstd  
#-----  
--
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
all : StubSMPlugin.so
```

```
#-----  
--
```

```
StubSMPlugin.so : StubSMPlugin.o StubSensorModel.o  
    $(CXX) -G $(CXXFLAGS) StubSMPlugin.o StubSensorModel.o \  
    -o StubSMPlugin.so  
    rm -f StubSMPlugin.so  
    ln -s StubSMPlugin.so .
```

```
#-----  
--
```

```
StubSMPlugin.o : StubSMPlugin.cpp \  
    Makefile \  
    StubSensorModel.h \  
    StubSMPlugin.h \  
    CSMPugin.h \  
    CSMError.h \  
    CSMWarning.h \  
    CSMMisc.h \  
    $(CXX) -c -PIC -g StubSMPlugin.cpp -o StubSMPlugin.o
```

```
#-----  
--
```

```
StubSensorModel.o : StubSensorModel.cpp \  
    Makefile \  
    StubSensorModel.h \  
    CSMSensorModel.h \  
    VCSMisc.h  
    $(CXX) -c -PIC -g StubSensorModel.cpp \  
    -o StubSensorModel.o
```

```
#-----  
--
```

```
clean :  
    -rm -f *.o vts *.so *~  
    -rm -rf SunWS_cache
```

## 8.2 GCC compiler:

For compiling the shared object (.so) source files to an object files (.o), the `-Wall -fPIC -c -o` switches are used. For compiling the .o files to .so files the `-Wall -o` switches are used. The `-Wall` switch turns on all the warning messages. The `-fPIC` switch emits position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table.

For compiling the main application source files to object files the `-Wall -c -o` switches are used.

To link the main application object to produce an executable the `-Wall -o -ldl` switches are used. The `-l` switch causes the link editor to look for files named `libdl.a` or `libdl.so`, the shared object handling library, in the library search path.

The following are example Makefiles that produces an executable called `vts`.

```
#####  
##  
#  
#   FILENAME:   Makefile_gcc  
#  
#   DESCRIPTION:  
#  
#   This is the master Makefile that is used by make to build the vts  
#   application and example plugins on Solaris using the gcc compiler.  
#  
#   NOTES:  
#  
#####  
##  
  
#-----  
--  
all :  
    ( cd ExampleSMPlugin ;make -f Makefile_gcc)  
    ( cd StubSMPlugin ;make -f Makefile_gcc)  
    ( cd CSM;make -f Makefile_gcc)  
    ( cd VTS ;make -f Makefile_gcc)  
    echo "Make has concluded"  
  
#-----  
--  
clean_all :  
    -rm -f *.o vts *.so *~  
    -rm -rf SunWS_cache  
    ( cd ExampleSMPlugin ;make -f Makefile_gcc clean)  
    ( cd StubSMPlugin ;make -f Makefile_gcc clean)
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
( cd CSM;make -f Makefile_gcc clean)
( cd VTS ;make -f Makefile_gcc clean)
echo "Make clean_all has concluded"

#-----
--
clean :
    -rm -f *.o vts *.so *~
    -rm -rf SunWS_cache

#####
##
#
#   FILENAME:   Makefile_gcc
#
#   DESCRIPTION:
#
#   This Makefile is used by make to build CSMPugin.o file
#   on Solaris using the gcc compiler.
#
#   NOTES:
#
#####
##

CXX = g++

VTSPROJDIR = ..

CSMINCDIR = $(VTSPROJDIR)/CSM_include

# The flags needed to compile all
CXXFLAGS = -g -Wall
CSHAREDFLAGS = -shared

# we need to link in the dll handling library

LDLIBS =    -ldl -lCstd

#-----
--
all : CSMPugin.o

CSMPugin.o    : CSMPugin.cpp \
    Makefile \
    $(CSMINCDIR)/CSMError.h \
    $(CSMINCDIR)/CSMWarning.h \
    $(CSMINCDIR)/CSMMisc.h \
    $(CSMINCDIR)/CSMPugin.h
    $(CXX) -c -g -I$(CSMINCDIR) CSMPugin.cpp -o CSMPugin.o

#-----
--
clean :
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
-rm -f *.o vts *.so *~
-rm -rf SunWS_cache

#####
##
#
#   FILENAME:   Makefile_gcc
#
#   DESCRIPTION:
#
#   This Makefile is used by make to build the vts application and
#   supporting code on Solaris using the gcc compiler.
#
#   NOTES:
#
#####
##

CXX = g++

VTSPROJDIR = ..
CSMINCDIR = $(VTSPROJDIR)/CSM_include
VTSINCDIR = $(VTSPROJDIR)/VTS_include
CSMDIR = $(VTSPROJDIR)/CSM

# The flags needed to compile all
CXXFLAGS = -g -Wall
CSHAREDFLAGS = -shared

# we need to link in the dll handling library

LDLIBS = -ldl

#-----
--
all : compareParam.o recordLog.o SMManager.o vts_isd.o vts_misc.o vts

# the main executable has to be linked with the -rdynamic flag
# so the plug in libraries can call inherited methods and
# access vttables in the main executable.

#-----
--
vts : vts.o \
      compareParam.o \
      recordLog.o \
      SMManager.o \
      vts_isd.o \
      vts_misc.o \
      $(CSMDIR)/CSMPlugin.o
      $(CXX) vts.o vts_misc.o vts_isd.o SMManager.o compareParam.o \
      recordLog.o $(CSMDIR)/CSMPlugin.o -o $(VTSPROJDIR)/vts $(LDLIBS)
      rm -f vts
      ln -s $(VTSPROJDIR)/vts .
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
# -rdynamic

#-----
--
vts.o : vts.cpp \
    Makefile_gcc \
    $(CSMINCDIR)/CSMPlugin.h \
    $(CSMINCDIR)/CSMError.h \
    $(CSMINCDIR)/CSMWarning.h \
    $(CSMINCDIR)/CSMSensorModel.h \
    $(CSMINCDIR)/CSMImageSupportData.h \
    $(CSMINCDIR)/CSMISDNITF20.h \
    $(CSMINCDIR)/CSMISDNITF21.h \
    $(CSMINCDIR)/CSMISDFilename.h \
    $(CSMINCDIR)/CSMISDByteStream.h \
    $(VTSINCDIR)/SMMManager.h \
    $(VTSINCDIR)/VCSMisc.h
    $(CXX) -c -g -I$(CSMINCDIR) -I$(VTSINCDIR) vts.cpp -o vts.o

SMMManager.o : SMMManager.cpp \
    Makefile_gcc \
    $(VTSINCDIR)/SMMManager.h \
    $(CSMINCDIR)/CSMPlugin.h \
    $(CSMINCDIR)/CSMError.h \
    $(CSMINCDIR)/CSMWarning.h
    $(CXX) -c -g -I$(CSMINCDIR) -I$(VTSINCDIR) SMMManager.cpp -o SMMManager.o

$(CSMDIR)/CSMPlugin.o :
    ( cd $(CSMDIR);make -f Makefile_gcc)

compareParam.o : compareParam.cpp \
    Makefile_gcc \
    $(CSMINCDIR)/CSMMisc.h \
    $(VTSINCDIR)/VCSMisc.h
    $(CXX) -c -g -I$(CSMINCDIR) -I$(VTSINCDIR) compareParam.cpp -o
compareParam.o

recordLog.o : recordLog.cpp \
    Makefile_gcc \
    $(CSMINCDIR)/CSMError.h \
    $(CSMINCDIR)/CSMWarning.h \
    $(CSMINCDIR)/CSMISDFilename.h \
    $(CSMINCDIR)/CSMISDByteStream.h \
    $(CSMINCDIR)/CSMISDNITF20.h \
    $(CSMINCDIR)/CSMISDNITF21.h \
    $(VTSINCDIR)/VCSMisc.h
    $(CXX) -c -g -I$(CSMINCDIR) -I$(VTSINCDIR) recordLog.cpp -o recordLog.o

vts_isd.o : vts_isd.cpp \
    Makefile_gcc \
    $(CSMINCDIR)/CSMError.h \
    $(CSMINCDIR)/CSMWarning.h \
    $(CSMINCDIR)/CSMImageSupportData.h \
    $(CSMINCDIR)/CSMISDFilename.h \
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
$(CSMINCDIR)/CSMISDByteStream.h \  
$(CSMINCDIR)/CSMISDNITF20.h \  
$(CSMINCDIR)/CSMISDNITF21.h \  
$(VTSINCDIR)/VCSMisc.h  
$(CXX) -c -g -I$(CSMINCDIR) -I$(VTSINCDIR) vts_isd.cpp -o vts_isd.o  
  
vts_misc.o      : vts_misc.cpp \  
Makefile_gcc \  
$(CSMINCDIR)/CSMError.h \  
$(CSMINCDIR)/CSMWarning.h \  
$(CSMINCDIR)/CSMImageSupportData.h \  
$(CSMINCDIR)/CSMPlugin.h \  
$(VTSINCDIR)/VCSMisc.h  
$(CXX) -c -g -I$(CSMINCDIR) -I$(VTSINCDIR) vts_misc.cpp -o vts_misc.o  
#-----  
--  
clean :  
-rm -f *.o vts *.so *~  
-rm -rf SunWS_cache
```

### **8.3 Instructions for compiling and Testing in Microsoft Windows:**

For building under Microsoft Windows, the current standard of the CSM requires that Visual Studio.NET 2003 be used for the API build. It is assumed that this software has been installed on your system. It is also assumed that the person responsible for building is familiar with the operation of this software.

The VS.NET environment requires that project files be used to define build parameters in much the same way as makefiles are used in UNIX environments. A complete set of project files has been developed for building and testing CSM and the VTS (Virtual Test System) test bed application. In addition, example project files are included for several sample plugins and models. All of the information necessary for defining and building the project, soln and proj files, is supplied with the distribution

Project files are fairly complex and fairly easy to break if edited by hand. The environment, VS.NET, provides a very powerful facility for editing these files and for this reason the contents of these files will not be listed here.

The steps needed to gain access to the project files from within the environment will be described. Knowledgeable users will be able to modify the project files as necessary. This system has been tested with XP-Professional and -Home and with Windows 2000.

For this release, the locations of the components on the system have been hard-coded into the project files. It is anticipated that in future releases, the locations will be given as environment variables to make the project files, data, etc. easier to relocate.

This distribution assumes that the tree is rooted directly under C:\ and is named CSM. If you extract the enclosed zip file into C:\ you will get a useable tree (the tree is described below).

After installing (unzipping) the tree, start VS.NET if not already running. In VS.NET Open the solution file csm\_test.sln in C:\CSM\CSM. This in turn will locate all of the other project files, sources, etc and load them. You may do builds at either the global csm\_test level or each of its projects.

Visual Studio.NET requires that all of the components of a build be either debug or release. If the build is mixed with some components release and some debug, an error message will be displayed at link time about a corrupt file. You must change the build to debug or release in both the csm\_test.soln and in the individual project files.

The MT (Multi-Threading) version of the Runtime library must be used for all projects in the build.

Note that VTS does NOT depend on anything but the CSM API library.

It will greatly improve the efficiency of testing models at the integrator if both a debug and release versions of the model are provided. The release version will be used for testing – the debug version will be used for diagnosing integration problems between the model and vts.

### 8.3.1 The Application Project vts

The 'application' project, vts, is the only executable in the system. It is a command-line based program that is designed to test all of the functionality of a model. Once you have done a build, you may run it from a command line window by changing directories to c:\csm\bin and typing vts. You may now type in responses to the prompts. You may also run vts non-interactively with a preprogrammed script file by redirecting a script to stdin, thus

➤ vts < test\_script

A large collection of test scripts is included with the distribution in the bin directory.

Of course, if you build the solution for debug you may set the script filename as the value of the command line in the properties pages for vts. You will then be able to run vts with that script in the debugger.

### 8.3.2 The CSM API Project

The CSM API library project is the only project in the solution that is required by other objects including your model. It contains the configured code described in this manual. No changes should be made to this code without the consent of the CCB. The system solution file automatically builds this, if necessary, before any of the other components of the solution.

As distributed, every other component of the system depends on the CSM API library. For this reason the output file must have the same name in all of the builds, csmapi.dll.

### 8.3.3 The Plugin Projects

Several example models and plugins are provided with this distribution. The integrator uses these to test vts to verify that it functions with simple standard models. These files also make good examples of how to write a conforming model. The only fully functional plugin is StubSensorModel. All of the other example models are used to test loading, releasing, and operation with multiple models present.

### 8.3.4 Frequently Asked Questions

#### 8.3.4.1 Why does our model fail to read properly from stdin?

When vts does console I/O it uses the crt provided by Microsoft. If the model also does console I/O it links in its own crt. As the program transitions from vts to the model it loses sync because of the two instances of the cin object. The way to fix this is to change the vts linker properties so it is also a dll.

#### To set this linker option in the Visual Studio development environment

1. Open the vts project's **Property Pages** dialog box. Click the **C/C++** folder.
2. Click the **Code Generation** property page.
3. Modify the **Runtime Library** property to add dll to the existing property(ies).

#### 8.3.4.2 Why does a build fail with the message: “Fatal Error: Invalid or corrupt file.” when trying to load or link our model?

This can happen when the model is built with a different version of the Runtime library (rtl) than csmapi and vts. Either change the rtl version for vts and csmapi (see 1.1.5.1 for how to do this) and recompile or recompile your model with the same version of rtl as vts and csmapi.

#### 8.3.4.3 Why does the debug version generate a heap/stack error?

VTS when run in debug can fail on a heap or stack error. This may be corrected by adding a value for heap/stack size to the vts debug properties. The default heap size is 1 MB.

#### **To set this linker option in the Visual Studio development environment**

1. Open the vts project's **Property Pages** dialog box. Click the **Linker** folder.
2. Click the **System** property page.
3. Modify the **Heap Commit Size** property.

You may also alter the stack values on this page. The unit is bytes.

#### 8.3.4.4 Why doesn't our model “register” in the main program.

This is usually caused by compiling the model with a different rtl than the main program. Please contact the integrator for further assistance if necessary.

#### 8.3.4.5 Why am I getting memory exceptions?

The same C/C++ runtime must be used when allocating and freeing memory across modules. If you choose to link with the static runtime library, then your module has its own private copy of the C/C++ runtime. When your module calls new or malloc, the memory can only be freed by your module calling delete or free. If another module calls delete or free, that will use the C/C++ runtime of that other module which is not the same as yours. If you choose to link with the DLL version of the C/C++ runtime library, you still have to agree which version of the C/C++ runtime to use. If the CSM plugin or TSAMAPI.DLL uses MSVCRT20.DLL to allocate memory, then anybody who wants to free that memory must also use MSVCRT20.DLL.

This rule is extended to the STL strings that are passed in/out as arguments to CSM API calls. When a std::string is instantiated, it has a default length of 15 characters. If an empty string is passed into a CSM member function which fills the string with more than 15 characters, memory will be allocated by that DLL to extend the std::string. When the variable goes out of scope in the calling application, an exception will be thrown because the memory is being freed using a different C/C++ runtime.

The workaround to this problem is to call `std::string.resize(<MAX>)` before calling the CSM member function. `<MAX>` is the maximum number of characters that could be assigned to the string within the function. This will prevent any memory allocation being done by the CSM DLL to avoid the unhandled exceptions.

For example:

```
std::string str = "";  
str.resize(MAX_NAME_LENGTH);  
warning = plugin->getPluginName(str);
```

## 9 APPENDIX D EXAMPLE CPP FILES

### 9.1 CSMPugin.cpp

```
//#####  
//  
// FILENAME: CSMPugin.cpp  
//  
// DESCRIPTION:  
//  
// This file provides implementation for methods declared in the  
// CSMPugin class.  
//  
// NOTES:  
//  
// Refer to CSMPugin.h for more information.  
//  
//#####  
#define CSM_LIBRARY  
  
#include <algorithm>  
#include <iostream>  
  
#ifdef _WIN32 //exports the symbols to be used (KJR)  
# include <windows.h>  
# include "CSMPugin.h"  
# include "CSMSensorModel.h"  
# include "CSMISDFilename.h"  
# include "CSMISDByteStream.h"  
# include "CSMISDNITF20.h"  
# include "CSMISDNITF21.h"  
#else  
# include <pthread.h>  
#endif  
#include "CSMPugin.h"  
#include "CSMWarning.h"  
#include "CSMError.h"  
  
//*****  
CSMPugin::CSMPuginList* CSMPugin::theList = NULL;  
CSMPugin::Impl* CSMPugin::theImpl = NULL;  
  
//*****  
// CSMPugin::Impl  
//*****  
class CSMPugin::Impl  
{  
public:  
  
//---  
// Modifiers  
//---  
  
void initializeMutex();  
// pre: None.  
// post: The mutex has been initialized.
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
CSMWarning * lockList(void);
    // pre: The list is unlocked.
    // post: The list has been locked.

CSMWarning * unlockList(void);
    // pre: The list is locked.
    // post: The list has been unlocked.

//---
// Data Members
//---

#ifdef _WIN32
    typedef HANDLE          Mutex;
#else
    typedef pthread_mutex_t Mutex;
#endif

    Mutex mutex;
};

/*****
// CSMPlugin::Impl::initializeMutex
*****/
void CSMPlugin::Impl::initializeMutex()
{
#ifdef _WIN32
    mutex = CreateMutex(NULL, FALSE, NULL); // TBD: handle errors
#else
    pthread_mutex_init(&mutex, NULL); // TBD: handle errors
#endif
}

/*****
// CSMPlugin::Impl::lockList
*****/
CSMWarning *CSMPlugin::Impl::lockList(void)
{
#ifdef _WIN32
    WaitForSingleObject(mutex, INFINITE);
#else
    pthread_mutex_lock(&mutex); // TBD: handle error returns
#endif
    return NULL;
}

/*****
// CSMPlugin::Impl::unlockList
*****/
CSMWarning *CSMPlugin::Impl::unlockList(void)
{
#ifdef _WIN32
    ReleaseMutex(mutex); // TBD: handle errors
#else
    pthread_mutex_unlock(&mutex); // TBD: handle error returns
#endif
    return NULL;
}

/*****
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
// CSMPugin::getList
//*****
CSMWarning * CSMPugin::getList(CSMPuginList*& aCSMPuginList) throw (CSMError)
{
    aCSMPuginList = theList;
    return NULL;
}

//*****
// CSMPugin::findPlugin
//*****
CSMWarning *CSMPugin::findPlugin(const std::string& pluginName,
                                CSMPugin*& aCSMPugin) throw (CSMError)
{

    CSMWarning* csmWarn = NULL;
    CSMPugin::CSMPuginList* models = NULL;
    theImpl->lockList();

    try {
        csmWarn = CSMPugin::getList(models);
    }
    catch (CSMError *err) {
        std::cout << err->getError() << '\n';
        std::cout << err->getMessage() << '\n';
    }
    catch (...) {
        std::cout << "&&&& UNKNOWN error thrown by getList\n";
    }

    if (!models)
        return csmWarn;
    bool found = false;
    for (CSMPuginList::const_iterator i = models->begin();
        i != models->end();
        ++i)
    {
        std::string apluginName;

        try {
            csmWarn = (*i)->getPluginName(apluginName);
        }
        catch (CSMError *err) {
            std::cout << err->getError() << '\n';
            std::cout << err->getMessage() << '\n';
        }
        catch (...) {
            std::cout << "&&&& UNKNOWN error thrown by getPluginName\n";
        }

        if (std::string(apluginName) == std::string(pluginName))
        {
            aCSMPugin = const_cast < CSMPugin* > (*i);
            found = true;
            break;
        }
    }

    try {
        csmWarn = theImpl->unlockList();
    }
```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
    }
    catch (...) {
        std::cout << "&&&&& ERROR thrown by unlockList\n";
    }

    if (!found)
    {
        if(!csmWarn)
            csmWarn = new CSMWarning();
        csmWarn->setCSMWarning (CSMWarning::DATA_NOT_AVAILABLE,
                                "No matching plugin found\n",
                                "CSMPlugin::findPlugin");
    }

    return csmWarn;
}

//*****
// CSMPlugin::removePlugin
//*****
CSMWarning *CSMPlugin::removePlugin(const std::string& pluginName) throw (CSMError)
{
    CSMWarning *csmWarn = NULL;
    CSMPlugin* pluginPtr = NULL;

    CSMError csmErr;
    std::string myName("removePlugin");

    try {
        csmWarn = findPlugin(pluginName, pluginPtr);
    }
    catch (CSMError *err) {
        std::cout << err->getError() << '\n';
        std::cout << err->getMessage() << '\n';
    }
    catch (...) {
        std::cout << "&&&&& UNKNOWN error thrown by findPlugin\n";
    }

    if (pluginPtr !=NULL)
    {
        try {
            csmWarn = theImpl->lockList();
        }
        catch (...) {
            std::cout << "&&&&& ERROR thrown by lockList\n";
        }

        // find and remove pointer-to-plugin from theList

        CSMPluginList::iterator pos = std::find(theList->begin(),
                                                theList->end(),
                                                pluginPtr);

        if (theList->end() != pos)
        {
            theList->erase(pos);
        }
        else
        {

```

# NGA.STND.0017\_3.0, Community Sensor Model (CSM) Technical Requirements Document (TRD), Version 3.0

```
std::cout << "CSMPlugin::removePlugin: Plugin " << pluginName
          << " not found" << std::endl;
// THROW A NOT FOUND EXCEPTION
csmErr.setCSMError (
    CSMError::UNKNOWN_ERROR,
    "Plugin Name Not Found",
    myName);
    throw csmErr;
}

try {
    csmWarn = theImpl->unlockList();
}
catch (...) {
    std::cout << "ERROR thrown by unlockList\n";
}
}
else
{
    std::cout << "CSMPlugin::removePlugin: Plugin " << pluginName
          << " not found" << std::endl;
// THROW A NOT FOUND EXCEPTION
csmErr.setCSMError (
    CSMError::UNKNOWN_ERROR,
    "Plugin Name Not Found",
    myName);

    throw csmErr;
}
return csmWarn;
} // removePlugin

/*****
// CSMPlugin::CSMPlugin
*****/
CSMPlugin::CSMPlugin()
{
    //---
    // If the list of registered sensor model factories does not exist yet, then
    // create it.
    //---

    if (!theList)
    {
        theList = new CSMPluginList;
    }

    if (!theImpl)
    {
        theImpl = new Impl;
        theImpl->initializeMutex();
    }

    //---
    // If the list of registered sensor model factories exists now (i.e., no
    // error occurred while creating it), then register the plugin factory in
    // theList by adding a pointer to this list.
    // The pointer points to the static instance of the derived sensor
    // model plugin.
    //---
}
```

NGA.STND.0017\_3.0, Community Sensor Model (CSM)  
Technical Requirements Document (TRD), Version 3.0

```
if (theList)
{
    CSMWarning *csmWarn = NULL;

    try {
        csmWarn = theImpl->lockList();
    }
    catch (...) {
        std::cout << "##### ERROR thrown by lockList\n";
    }

    if (csmWarn == NULL)
    {
        theList->push_back(this);

        try {
            csmWarn = theImpl->unlockList();
        }
        catch (...) {
            std::cout << "##### ERROR thrown by unlockList\n";
        }
    }
}
}
```