INSTRUMENT DESCRIPTION
FILE SYSTEM DEFINITION

(IDFS)

Version 2.1
Document Release R

Department of Space Science
Southwest Research Institute ® (SwRI ®)
6220 Culebra Road
San Antonio, TX 78238-5166

Document Maintained By:
Sandee Jeffers
sjeffers@swri.org

Carrie Gonzalez
cgonzalez@swri.org

The development of this Software was supported by contracts NAG5-3148, NAG5-6855, NAS8-36840, NAG5-2323, and NAG5-7043 issued on behalf of the United States Government by its National Aeronautics and Space Administration.

THIS SOFTWARE. IN NO EVENT SHALL SwRI BE LIABLE FOR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGE (INCLUDING LOSS OF BUSINESS, REVENUE, PROFITS, USE, DATA OR OTHER ECONOMIC ADVANTAGE) HOWEVER IT ARISES, WHETHER FOR BREACH OF IN TORT, EVEN IF SwRI HAS BEEN PREVIOUSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. YOU HAVE SOLE RESPONSIBILITY FOR ADEQUATE PROTECTION AND BACKUP OF DATA AND/OR EQUIPMENT USED IN CONNECTION WITH THE SOFTWARE AND WILL NOT MAKE A CLAIM AGAINST SwRI FOR LOST DATA, RE-RUN TIME, INACCURATE OUTPUT, WORK DELAYS OR LOST PROFITS RESULTING FROM THE USE OF THIS SOFTWARE. YOU AGREE TO HOLD SwRI HARMLESS FROM, AND YOU COVENANT NOT TO SUE SwRI FOR, ANY CLAIMS BASED ON USING THE SOFTWARE.

Local Laws: Export Control

You acknowledge and agree this Software is subject to the U.S. Export Administration Laws and Regulations. Diversion of such Software contrary to U.S. law is prohibited. You agree that none of the Software, nor any direct product therefrom, is being or will be acquired for, shipped, transferred, or reexported, directly or indirectly, to proscribed or embargoed countries or their nationals, nor be used for nuclear activities, chemical biological weapons, or missile projects unless authorized by U.S. Government. Proscribed countries are set forth in the U.S. Export Administration Regulations. Countries subject to U.S embargo are: Cuba, Iran, Iraq, Libya, North Korea, Syria, and the Sudan. This list is subject to change without further notice from SwRI, and you must comply with the list as it exists in fact. You certify that you are not on the U.S. Department of Commerce's Denied Persons List or affiliated lists or on the U.S. Department of Treasury's Specially Designated Nationals List. You agree to comply strictly with all U.S. export laws and assume sole responsibilities for obtaining licenses to export or reexport as may be required.

General

These Terms represent the entire understanding relating to the use of the Software and prevail over any prior or contemporaneous, conflicting or additional, communications. SwRI can revise these Terms at any time without notice by updating this posting.

Trademarks

The SwRI logo is a trademark of SwRI in the United States and other countries.

Revision Log

Revision	Release Date	Changes to Document
Version 2.1 Release G	1/16/03	• Revision Log was added. • SwRI logo was updated since now Registered. • Spacecraft Potential definitions were added in Token-tagged section (section 5.6).
Version 2.1 Release H	3/23/04	• Calibration Data Scope definition was added in Token-tagged section (5.7). • Maximum packing size for scalar instruments was added in Token-tagged section (5.8). • Updated table in section 6 to add the fields cal_scope, max_packing and nano_defined. • Orbiting_body VIDF field has new value definitions. • Added pot_src_defined token for Section 5.6 to be consistent with Euler Angle Rotation information (Section 5.5). • Updated section 4.5.5 (d_type) to clarify that this applies only to sensor data, not to any calibration data which may be defined within the IDFS data file.
Version 2.1 Release I	5/17/05	• Start of Spin IDFS data source definitions were added in Token-tagged section (section 5.9). • Clarifications made for **crit_act_sz**, **crit_status**, **crit_action**, **tbl_off, cal_scope, max_nss,** and **hdr_off** fields. • Clarifications made for timing issues for Multiple VIDF files (section 6).
Version 2.1 Release J	6/06/05	• Fixed example provided for **cal_scope** (section 5.7)
Version 2.1 Release K	1/03/06	• Updated section 5.6 to add **replace_value** to the list of tokens for the Spacecraft Potential definition. • Updated section 4.1.4.6 (tbl_var) to add new value to be used to specify that the table is a function of spacecraft potential data. • Updated section 5.7 to define **d_type** for calibration data sets.
Version 2.1 Release L	6/05/06	• Updated section 5.6 to make use of the format field since constant option is now part of VIDF definition; also renamed **replace_value** to **constant_value** in section 5.6.
Version 2.1 Release M	12/12/06	• Updated section 4.15.1 for addition of constant id 14 and for clarification of usage of constant ids 12 and 13. • Updated section 5.9 to match new constant id 14 text.
Version 2.1 Release N	07/16/07	• Updated section 5.5.4. to indicate scalar IDFS source.

Revision	Release Date	Changes to Document
Version 2.1 Release O	10/01/09	• Revised section 5.5 (Euler Angle Rotation Information) since Euler is now one of 2 possible sections described for coordinate system transformation, with Celestial Position Information being the newest section related to coordinate system transformations.
Version 2.1 Release P	12/28/12	• Background definitions were added in Token-tagged section (section 5.10). • Updated sections 4.10.12 and 5.5.2.14 to clarify examples of utilizing constants for the Pitch Angle and Euler Angle values. • Updated section 4.1.4.6 (tbl_var) to add new value to be used to specify that the table is a function of background data.
Version 2.1 Release Q	03/19/13	• Corrected the second example in section 5.5.3.16
Version 2.1 Release R	04/21/14	• Clarification in section 4.15.1 (**const_id**) for initial and final elevation angle constants

1. Instrument Data File Set (IDFS) Overview

The IDFS is a self-documenting scientific data storage format which is serviced by a set of access routines referred to as the IDFS data access software. The IDFS paradigm and corresponding data access routines were developed and are maintained by Southwest Research Institute (SwRI). The IDFS format was created so that analysis routines could be developed which would have usage over a wide range of data sets. The IDFS data access software provides both a low and high level set of routines which allow access to the IDFS data from application programs. See the **IDFS Programmers Manual** (*http://www.idfs.org/idfs_prog_man.ps*) for more information and details concerning the IDFS data access routines. A **complete** IDFS implementation consists of five files:

1. the Virtual Instrument Description File (VIDF),
2. the Header File (HF),
3. the Data File (DF),
4. the Plot (or Applications) Interface Definition File (PIDF), and
5. the Special Computation File (SCF).

The first three of these files (VIDF, HF, & DF) are necessary within the IDFS formalism while the latter two (PIDF & SCF) are optional. The PIDF contains interface definitions between the IDFS information and the display / application programs (see *http://www.idfs.org/Editors/pidfdoc.html*). The SCF contains user defined analysis algorithms which are calculated at run-time using IDFS sources for input and generate "virtual" IDFS output values (see *http://www.idfs.org/scf_paper.html* and *http://www.idfs.org/scf_docu.PS*). SCF files are linked with all four other IDFS file types and it is assumed that only experienced IDFS users would make use of them. The contents and creation of the first three above-mentioned files (VIDF, HF, & DF) are discussed at length within this document.

The IDFS is meant to be general enough to encompass a majority of scientific data sets, including raw telemetry, processed data, simulation data, and theoretical data. The inclusion of raw telemetry in this set is of primary importance and indeed was the driving goal in the development of the IDFS formalism. The storage of processed data is often fraught with problems. Often times a measurement can be expressed in several sets of units. Saving only one set of units places undue restraints on many analysis projects while saving all conceivable units puts undue strain on the archiving ability of the host computer. The gradual refinement of calibration data often results in changes in the scaling parameters used to create the processed data causing at times multiple re-creations of a single data set. Storing and using raw data alleviates many, if not all, of these problems. The IDFS keeps the decompression and scaling algorithms for each data set within the VIDF as a set of tables which are applied singularly or as a set to the data as it is accessed to achieve multiple sets of scientific units. Thus, one can easily update scaling parameters, efficiency tables, etc. within the VIDF as necessary and so insure that any subsequent access of the data will be correctly converted to the set of physical units requested.

1.1 Virtual Instrument Concept

The architecture of the IDFS is based upon the concept of a virtual instrument. A virtual instrument is a subset of the data taken by an instrument which is grouped together for a number of reasons, such as similar characteristics, similar data rate, similar word lengths, etc. The characteristics of the data are defined within the VIDF and the data itself is contained within both the Header and Data files. A well designed virtual instrument, and hence IDFS data set, is based on a good understanding of the measured data which includes how the data was acquired, how it is generally used, and how it is converted to physical units.

All data is stored within the IDFS Header and Data files and is considered to be **raw** data within the IDFS paradigm even though it may already have been highly processed (e.g., the output of a computer simulation). The term **raw** is used to delineate the stored IDFS data from IDFS data which has undergone any modification, such as the conversion to a set of physical units. It is important to note that all IDFS Header and Data records need to be written in *network order*. This is done to ensure portability of the IDFS data between platforms (Little Endian vs. Big Endian). The following section further describes *network order*.

1.2 Network Order

"*Network order*" is the canonical form of multi-byte data quantities when they are being transferred across a network. Multi-byte data quantities are short integers, long integers, and long long integers in C notation. These types of data quantities are transferred with the most numerically significant byte sent first. "*Host order*" is the order in which a particular type of computer stores multi-byte data quantities internally. [For original citation, see "Internet Protocol", *RFC 791*, **Information Sciences Institute**, Sept. 1981, Appendix B.] Most flavors of UNIX (including Linux) provide a standard set of C library routines for making conversions between *network order* and *host order*:

Function	Action
long htonl(long)	Converts a host long to a network ordered long
long ntohl(long)	Converts a network long to a host long
short htons(short)	Converts a host short to a network ordered short
short ntohs(short)	Converts a network short to a host short

Using these routines isolates the user from having to know anything about the byte order used on a given machine. Note that character strings are considered arrays of single byte quantities, so they do not require conversion. Since the VIDF is created as an ASCII file, conversions are not necessary.

1.3 IDFS Lineage

Every measurement with an IDFS inherits a lineage, and through this lineage, can be uniquely identified and accessed among all of the defined IDFS sets. This lineage is outlined below.

measurement → virtual instrument → instrument→ experiment→ mission→ project

The lineage is nothing more than an identification hierarchy and it is not uncommon for one or more of its levels to be redundant. All data placed within an IDFS are defined under the broad heading **measurement**. These are of three types: mode data, primary sensor data, and secondary sensor data. The latter two are often referred to simply as **sensor** and **calibration** data, respectively. Each **measurement** is associated with a **virtual instrument**, which is a grouping of measurements all of which are obtained from a common **instrument**. A single instrument can be the "parent" of multiple virtual instruments, or equivalently, a virtual instrument does not necessarily contain all of the measurements generated by the instrument. Atop the instrument level is the **experiment**. Each instrument is associated with a single experiment which is defined to be an assemblage of instruments generally headed by a single PI. It is not unusual for the instrument and the experiment to be one and the same, and if the virtual instrument is created to contain the total data output of the instrument, it too may be identical to the instrument and experiment. Each experiment is associated with a **mission** which is an aggregate of experiments generally brought together to study a common problem. The highest level of the lineage is the **project** which is a coordinated set of missions which may be performed simultaneously or spread out in time. In many cases, the mission and project are identical.

The levels in a measurement's lineage are identified through acronyms. The measurement itself is identified through the VIDF. For example, consider the set of x component magnetic field measurements acquired by the Vector Magnetometer Experiment on the Upper Atmosphere Research Satellite. This set of measurements is part of the virtual instrument **VMMA** which contains the magnetic field components and necessary correction data. The virtual instrument was formed from the Vector Magnetometer Experiment termed **VMAG** which in turn is part of the Particle Environment Monitor (**PEM**) experiment on the **UARS-1** satellite (mission). The overall project in this case is **UARS**. The lineage specification of the measurement is then given by:

$$VMMA \rightarrow VMAG \rightarrow PEM \rightarrow UARS\text{-}1 \rightarrow UARS$$

The orbit/attitude data for a satellite provides a good example of redundant levels in the lineage. The orbit/attitude data for the Upper Atmosphere Research Satellite (UARS) has the following IDFS lineage specification:

$$OAUR \rightarrow OAUR \rightarrow OAUR \rightarrow UARS\text{-}1 \rightarrow UARS$$

1.4 Measurement Classifications

As mentioned earlier, there are three classes of data within the IDFS. All information returned by an instrument does not have the same function and this is recognized by the IDFS. Some are classified as **primary measurements**, those which return information on the object being studied. Some information concerns the state of the primary data. This is ancillary data which is necessary to interpret the primary data correctly. These data are termed **secondary measurements**. They contain information that is needed to calibrate the primary measurement, and thus, are often referred to as **calibration measurements**. The third type of measurement concerns data which defines various states of the instrument and which are sometimes necessary in the processing or use of the primary sensor data. This data is called **mode** data. Both primary

and secondary data are stored in the IDFS Data File. Mode data, on the other hand, is placed in the Header record as status bytes because this type of data is not usually rapidly changing data.

Primary data is, by its very nature, multi-dimensional. Each measurement, by default, is a function of three spatial coordinates (position) and time. The IDFS transparently handles the temporal dimension through the creation of time ordered data sets, and if it exists, an azimuthal or spin angle dependence. Other spatial dimensions, where necessary, are handled through the creation of position sensors, as generally occurs in virtual instruments containing orbit/attitude data. More complex measurements, however, may have additional dimensional dependencies (e.g., energy, mass, frequency, wavelength, etc.). Several common instruments include the E/q spectrometer which measures particles as a function of density and the imager which measures intensity as a function of scan position or pixel number. Because such dependencies are common among many measurements, the IDFS is designed with two categories of primary sensor data, **vector** and **scalar**. Scalar data are measurements which depend at most only on position and time. Vector data are more complicated and allow for the sensor measurements to have a functional dependence on a single variable. This variable in IDFS terminology is called the **scanning** variable. There is one scanning value per vector element. The set of scanning elements for a vector are stored in the Header file as a set of indices into a look-up table (kept in the VIDF) which converts these to appropriate units. All vectors within a logical instrument are assumed to use the same set of scanning INDICES, i.e., each sensor does not have a unique set of indices; however, each sensor can have a different set of scan VALUES stored in the VIDF where the scan indices map to a different set of scan values per sensor. The details on how to accomplish this are provided in section 4.14 where the "TABLE BLOCK" is described. A vector sensor also differs from a scalar sensor in that it has associated with it the concept of length; the number of elements in a single scan.

1.5 Overview of IDFS Files

The three IDFS files, Header, Data, and VIDF, are sufficiently intertwined with enough inter-dependencies that it is virtually impossible to discuss one of the files without at least some mention of the other two.

1.5.1 Data File

The vast majority of all of the telemetered data is stored within the Data File. Each Data file consists of a set of fixed-size Data records. The Data records contain a small set of general information, which includes the starting accumulation time of the first measurement in the data array, information on the angular orientation of the virtual instrument, and information on how to obtain the current state of the virtual instrument. This is followed by the data array. The data array contains all of the sensor measurements contained within the Data record.

The data array is a set of two-dimensional matrix pairs. In each matrix pair the first matrix contains the primary sensor data and the second matrix contains the secondary or calibration data. Not all virtual instruments have secondary (calibration) data, in which case, a secondary matrix is non-existent. Together these two matrices form what is termed a **sensor set**. A sensor set is a logical block of data containing measurements from some, but possibly not all,

of the sensors defined within the virtual instrument. Each sensor set has associated with it a byte offset pointer into the Header file. This offset points to the Header record containing the instrument state data for the information in this sensor set. Within a sensor set then, the instrument state is assumed to be constant.

The primary data matrix within the sensor set is of order **n_sample** × **n_sen** (both variables being fields within the Header record). Each column then represents data from one of the virtual instrument sensors and each row represents a measurement associated with the sensor. If the virtual instrument is a vector instrument, then the number of rows in the matrix is equivalent to the vector length, otherwise, the number of rows is simply the number of consecutive measurements which form the sensor set. The secondary matrix within the sensor set is of order CAL_SAMPLE × **n_sen**. Each column contains the calibration data for a single sensor with the number of calibration elements present (CAL_SAMPLE) being determined from the number of calibration sets defined and the number of elements per calibration set. The calculation to obtain CAL_SAMPLE is described in detail under the description of the **cal_use** field in the VIDF. The secondary matrix (if present) has the same number of columns as the primary matrix (one column for each sensor), where each column in the secondary matrix contains data for the same sensor as the corresponding column in the primary matrix. In other words, sensor A has its primary measurement values in column 1 of the primary matrix and its calibration values in column 1 of the secondary matrix. However, the length of the columns (# rows) may be different (**n_sample** rows in the primary matrix and CAL_SAMPLE rows in the secondary matrix as described above). The two matrices which constitute a sensor set are shown pictorially below in Figure 1.

PRIMARY DATA MATRIX										
	N_SEN									
N_SAMPLE										
	D	D	D	D	D	D	D	D	D	
	a	a	a	a	a	a	a	a	a	
	t	t	t	t	t	t	t	t	t	
	a	a	a	a	a	a	a	a	a	
	S	S	S	S	S	S	S	S	S	
	e	e	e	e	e	e	e	e	e	
	n	n	n	n	n	n	n	n	n	
	A	B	C	D	E	F	G	H	I	

SECONDARY DATA MATRIX										
	N_SEN									
CAL_SAMPLE										
	C	C	C	C	C	C	C	C	C	
	a	a	a	a	a	a	a	a	a	
	l	l	l	l	l	l	l	l	l	
	S	S	S	S	S	S	S	S	S	
	e	e	e	e	e	e	e	e	e	
	n	n	n	n	n	n	n	n	n	
	A	B	C	D	E	F	G	H	I	

Figure 1. IDFS Sensor Set

To allow for cases when not all data is returned for every Data record, the **data_array** has **max_nss** sensor sets defined (since Data records are fixed length), but there are **nss** sensor sets returned within a Data record. The VIDF contains the **max_nss** definition and **nss** is one of the general information fields within the Data record, where **nss** is less than or equal to **max_nss** for any given Data record. Each sensor set has its own associated Header record. The information within the Header record not only contains the current state of the virtual instrument but also the number of sensors (columns) returned in the sensor set and the number of elements (rows) returned per sensor (all sensors within a sensor set must return the same number of measurements). Identical sensor sets, that is, sensor sets which return data for the same sensors and return data for periods of time in which the virtual instrument is in the same state, have identical offsets into the Header file. Thus, a given Header record is used by multiple sensor sets.

1.5.2 Header File

A brief introduction to the Header file has already been given in the above description of the Data file. Each Header record within the Header file contains the state of the instrument, a set of timing information for the sensors, and the makeup of the Data record sensor sets to which the Data record is associated. Unlike the Data records, the Header records can have variable sizes. The first two bytes of any Header record contain the record size. The data and information contained within the Header file should be slowly varying so that a minimum number of Header records are actually needed to describe the instrument data over a long period of time. While this is generally achievable, there are times when it is not and the Header file can be lengthy. Many times a virtual instrument may switch states between successive sensor sets creating a change in the Header pointer at each sensor set. If the virtual instrument state is rotating between a finite set of states, there is only the need for one Header record per state with the Header offsets assigned to the sensor sets pointing to the appropriate Header record.

The only telemetered data which is stored within the Header records is found in the **mode_index** field. This is a linear array of defined instrument status bytes (modes). These status bytes, in general, define conditions or states of the virtual instrument which may be pertinent in the processing of the primary sensor data. Other Header data fields include sensor set timing values, number of sensors returned and their order, number of scanning steps returned and their offsets, and data quality flags per sensor. These fields may be preset values, and all or some of which may be functions of the instrument status.

1.5.3 VIDF File

The last of the three necessary IDFS files is the VIDF file. The VIDF describes the IDFS virtual instrument and sets up all of the parameters which are needed to interface with the IDFS data and header files. Its purpose is to provide both a description of the measurements being stored in the IDFS and to provide information to the IDFS data access software which extracts data from the IDFS files and converts each of the measurements into one or more sets of physical units. The VIDF is meant to be easily updated and to contain all of the data which may be periodically updated due to either refinements in the instrument calibration or due to degradation within the instrument. There must be at least one VIDF file defined for each IDFS virtual instrument. If data within the VIDF changes with time, for example calibration coefficients, additional VIDF files can be established, the period of applicability of each being given within the VIDF file itself. Algorithms within the VIDF are designed to be used by the IDFS data access software to transform the raw IDFS data into physical units. Each step within an algorithm can be applied or omitted which allows a single algorithm to sometimes generate several sets of physical units depending on the steps included. For example, the output of a temperature sensor may be converted to volts in the first step of an algorithm, to degrees C in the second step and to degrees F in the third. Each can be obtained by the user through the IDFS data access software by knowing which algorithm steps to include in the conversion process.

When the IDFS paradigm was first developed, all three files – the header, data and VIDF file – were created and archived as binary files, which by definition, resulted in fixed-formatted files. To ease the effort entailed in maintaining the VIDF files, a shift was made to create the

VIDF file as an ASCII file; however, the IDFS data access software still expected a binary file. A conversion program was written and is utilized to transform the VIDF ASCII file into a binary file for use with the IDFS data access software. While the VIDF file became easier to maintain, the file itself is still considered a fixed-formatted file since new fields cannot be added due to the binary format reliance.

As more and more data sets were being converted into the IDFS storage format, the need to expand the current set of fields defined within the VIDF file finally became a reality. In order to preserve backwards compatibility with data sets already in IDFS format, software was modified to enable the parsing of a token-tagged, field-extensible VIDF file, while maintaining the ability to parse the old, fixed-formatted binary VIDF files. Although a converter program has been developed to transform a fixed-formatted ASCII VIDF file into a token-tagged VIDF file, there is no mandate at the present time to force the migration to the new format. Keep in mind that although new fields can be defined and stored in the token-tagged VIDF files, these fields are not automatically utilized by an end-user application or the IDFS data access software. Code changes must be made to the software which parses the VIDF file and returns information contained in the VIDF file in order to look for these newly defined fields.

1.6 File Naming Conventions

There is a simple file naming convention used when creating a set of IDFS files. Each virtual instrument is assigned an acronym up to 8 characters in length. Since this acronym is used system-wide when accessing a virtual instrument, all IDFS file names begin with the virtual instrument acronym. For Header, Data, and VIDF files this is followed by 12 characters, the first 11 specifying the starting time of the file and the final character indicating the file type. The 11 characters used in specifying the starting time of the file have the format

YYYYDDDHHMM

where YYYY is the year (e.g., 1992), DDD is the day of year (e.g., 020), HH is the hour and MM is the minute. The last character in the file name is a **D** if the file is a Data File, an **H** if the file is a Header file, and a **V** if the file is a fixed-formatted VIDF file. An example IDFS file set for the logical instrument TDIE might appear as:

TDIE19922302034D
TDIE19922302034H
TDIE19922302034V

where each file has a starting time of 20:34 (HH:MM) on day 230 of year 1992. The first file is the Data file, the second is the Header, and the third is the VIDF. If the VIDF file is a token-tagged VIDF file, the suffix ".v3" is appended to the filename, e.g. TDIE19922302034V.v3.

1.7 IDFS Assumptions

Within the IDFS paradigm, the assumption is that the spacecraft spins around Axis C, as illustrated in Figure 2 below, and all calculations assume that Axis C is the spin axis. In

addition, the assumption is that all calculations are made with respect to a right-handed coordinate system. These assumptions are important for pitch angle computations (see sections 4.10 and 4.15.1)

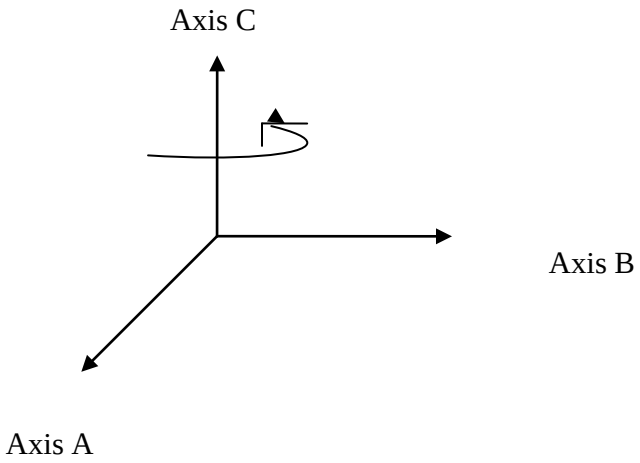

Figure 2. IDFS Axes Definition

For most of the data sets that have been stored in IDFS format, the x-axis of the spacecraft has been defined as Axis A, the y-axis of the spacecraft has been defined as Axis B and the z-axis of the spacecraft has been defined as Axis C, resulting in the +z axis being defined as the spin axis. However, this spacecraft naming convention is not always followed by all space exploration missions. For example, with the Cluster mission, the following spacecraft situation is defined:

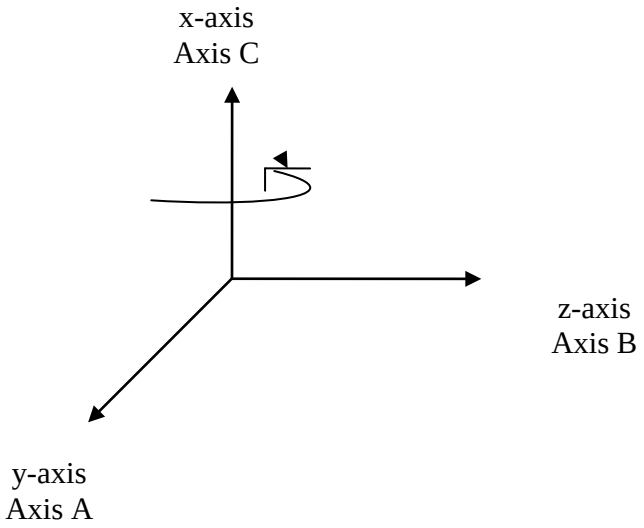

Figure 3. Cluster Axes Definitions

For Cluster, the spacecraft defines the spin axis as the x-axis. Thus, the axes definition for IDFS are Axis A is the spacecraft y-axis, Axis B is the spacecraft z-axis and Axis C is the spacecraft x-axis. In order to avoid confusion, references to axes through out this document will be made with respect to Axis A, B and C as opposed to x, y and z.

2. Structure for Fixed-Formatted Virtual Instrument Description File (VIDF)

The fixed-formatted VIDF file is created as an ASCII file, but must be converted to a binary file before it can be used by the IDFS data access software developed at SwRI (See IDFS Programmer's Manual). The program used to convert the VIDF from ASCII to binary uses *network order* protocols when writing the binary VIDF file. The ASCII VIDF file is considered the "master" and is the archived file. If modifications to the VIDF are necessary, they are done in the ASCII file which should replace the old ASCII VIDF in the archive. The binary VIDF file is generated from the ASCII file using the binary conversion software, *mk_idf*. The SwRI-developed database and catalog system automatically runs the *mk_idf* filter when fixed-formatted VIDF files are promoted from the archive. For version control, *mk_idf* first writes out an 8-byte version number into the binary VIDF before converting and writing the ASCII VIDF to the binary VIDF.

Each line in the VIDF file consists of three blocks as shown below:

FORMAT | ENTRY | COMMENTS

The FORMAT field is a single ASCII character which defines how the ENTRY field is interpreted by the binary conversion program (*mk_idf*). The FORMAT specifier may be a character defining how the entry data is to be stored in the binary VIDF, an information flag signifying the beginning of an array in the VIDF, or that the field is to be ignored. The ENTRY block contains the VIDF data value(s) for the particular VIDF field being defined. The block may contain several entries which are separated by spaces. The ENTRY block is the only block which is stored in the binary version of the VIDF. The COMMENTS block is a set of optional text not processed by *mk_idf*. The COMMENTS block is generally used to identify the VIDF lines, and must be preceded by the characters "/*" to be recognized by *mk_idf* as comments.

The FORMAT block must be one of seven characters recognized by *mk_idf*. The characters are defined in the table below.

FORMAT CHARACTER DEFINITIONS		
CHARACTER	DEFINITION	USAGE
n	null entry	information
m	beginning of array entry	information
l	entries are stored as 4 bytes	entry size
s	entries are stored as 2 bytes	entry size
b	entries are stored as 1 byte	entry size
t	long text information stored as 79 bytes	entry size
T	short text information stored as 20 bytes	entry size

The FORMAT **n** is used to define a field entry which is null, that is, which has no entry in the VIDF. The binary conversion program expects to see nothing in the ENTRY block after this format. Null entries are used when the VIDF parameter or data is not used for the virtual

instrument being defined. For example, a virtual instrument which does not have a fill value defined (*fill_flg = 0*) will have the null entry for the *fill* value:

```
b       0                                       /*   fill_flg    */
n                                               /*   fill        */
```

The FORMAT **m** is used to define the beginning of an array entry in the VIDF. The binary conversion program expects to find two entries in the ENTRY block on this line. The first is the number of elements in the array and the second is the number of elements per line. The entries occur on the next N lines. The number of entries on the last line only need to complete the array. An example showing how an array of 18 elements, 5 elements per line, is specified in the VIDF is shown below. The 18 values will be stored as 1 byte quantities.

```
m       18  5                                   /*   array       */
b       0       0       1       0       2      /*   00000-00004 */
b       2       2       6       4       4      /*   00005-00009 */
b       0       0       8       8       8      /*   00010-00014 */
b       3       3       3                      /*   00015-00017 */
```

In the VIDF, an array of L elements (ex: 18) begins indexing at 0 and ends at element L-1 (ex: 17).

An overview of the VIDF fields is shown in the tables below. The VIDF file consists of a set of fields which are either individual entries or block entries. Block entries are themselves sets of individual fields. Each individual field is characterized by a base length (byte size), an array size and a repetition number. In these tables, all fields whose names are given in all lower case letters represent individual field entries while fields whose names given in all upper case characters represent blocks of fields. These blocks of fields are defined in tables below the main VIDF definition. Note that TABLES and CONSTANTS blocks of fields do not have either a base size or array size associated with them, but they do have a repetition size.

VIDF FILE FORMAT				
FIELD NAME	FIELD DESCRIPTION	BASE SIZE (Bytes)	ARRAY SIZE	FIELD REPETITION
project	Project description	1	79	1
mission	Mission dexription	1	79	1
experiment	Experiment description	1	79	1
v_inst	Virtual instrument description	1	79	1
contact	Five line contact address	1	79	5
num_comnts	Number of comment lines	2		
comments	General comments	1	79	num_comnts
ds_year	Start year	2		
ds_day	Start day	2		
ds_msec	Start time (msec)	4		
ds_usec	Offset from ds_msec	2		

VIDF FILE FORMAT				
FIELD NAME	FIELD DESCRIPTION	BASE SIZE (Bytes)	ARRAY SIZE	FIELD REPETITION
de_year	End year	2		
de_day	End day	2		
de_msec	End time (msec)	4		
de_usec	Offset from de_msec	2		
smp_id	Instrument sweep identifier	1		
sen_mode	Data matrix format	1		
n_qual	Number of quality definitions	1		
cal_sets	Number of calibration sets	1		
num_tbls	Number of entered tables	1		
num_consts	Number of entered constants	1		
status	Number of instrument status defs	1		
pa_defined	Pitch angle information status	1		
sen	Number of sensor definitions	2		
swp_len	Elements in full sample sequence	2		
max_nss	Maximum number of sensor sets	2		
data_len	Size of data record	4		
fill_flg	Fill data flag	1		
fill	Fill data code	4		
da_method	Data accumulation code	1		
status_names	Status definitions	1	79	status
states	Definitions per status	2	status	1
sen_name	Sensor definitions	1	79	sen
cal_names	Calibration definitions	1	79	cal_sets
qual_name	Data quality definitions	1	79	n_qual
pa_format	Pitch angle computation format	2		
pa_project	Project of magnetic field data	1	20	1
pa_mission	Mission of magnetic field data	1	20	1
pa_exper	Exper of magnetic field data	1	20	1
pa_inst	Inst of magnetic field data	1	20	1
pa_vinst	Virtual inst of magnetic field data	1	20	1
pa_b1b2b3	B1, B2, B3 sensor numbers	2	3	1
pa_apps	Number of tables to apply	2		
pa_tbls	Table numbers	2	pa_apps	1
pa_ops	Defined operation for tables	2	pa_apps	1
d_type	Data format	1	sen	1
tdw_len	Data word length (bits)	1	sen	1
sen_status	Sensor status	1	sen	1
time_off	Time correction (msec)	4	sen	1
cal_use	Calibration table usage def.	2	cal_sets	1
cal_wlen	Calibration word length (bits)	1	cal_sets	1
cal_target	Target of calibration set	1	cal_sets	1

VIDF FILE FORMAT				
FIELD NAME	FIELD DESCRIPTION	BASE SIZE (Bytes)	ARRAY SIZE	FIELD REPETITION
TABLES	Put Table Information Here			num_tbls
CONSTANTS	Put Constants Information Here			num_consts

The TABLES field represents multiple fields which define a single table block entry in the VIDF. There are **num_tbls** blocks. The structure of the TABLE block is shown below.

VIDF TABLE FORMAT				
FIELD NAME	FIELD DESCRIPTION	BASE SIZE (Bytes)	ARRAY SIZE	FIELD REPETITION
tbl_sca_sz	Number of elements in scaling table	4		
tbl_ele_sz	Number of elements in table	4		
tbl_type	Table type	1		
tbl_comnts	Lines in table description	2		
tbl_desc	Table description	1	79	tbl_comnts
tbl_var	Variables table operates on	1		
tbl_expand	Expand to look-up format flag	1		
crit_act_sz	Size of critical action array	4		
crit_status	Sensor critical status bytes	1	sen	1
crit_off	Sensor offsets into critical action array	2	sen	1
crit_action	Critical action array	4	crit_act_sz	1
tbl_fmt	Table format	1	sen or status	1
tbl_off	Sensor offset into table	4	sen or status	1
tbl_sca	Scaling table	1	|tbl_sca_sz|	1
tbl	Table	4	tbl_ele_sz	1

The CONSTANTS field represents multiple fields which define a single constant block entry in the VIDF. There are **num_consts** blocks. The structure of the CONSTANTS block is shown below. Only defined CONSTANTS identifiers are allowed as constant block information. Refer to the discussion in the "CONSTANTS BLOCK" section.

VIDF CONSTANT FORMAT				
FIELD NAME	FIELD DESCRIPTION	BASE SIZE (Bytes)	ARRAY SIZE	FIELD REPETITION
const_id	Constant type identifier	1		
const_comnts	Lines in constant description	2		
const_desc	Constant description	1	79	const_comnts
const_sca	Scaling for constants	1	sen	1
const	Constants	4	sen	1

3. Structure for Token-Tagged Virtual Instrument Description File (VIDF)

The token-tagged VIDF file is created as an ASCII file consisting of multiple sections. The token-tagged VIDF file consists of all the same fields that are listed in the VIDF File Format Table defined in section 2, with the exception of the following fields:

1. **num_comnts** – not needed since the comments are simply included in the VIDF file, using the same syntax as C.
2. **pa_defined** – this flag is set based upon the existence of a structure which contains pitch angle information contained within the token-tagged VIDF.
3. **tbl_comnts** – not needed since the comments are simply included in the VIDF file, using the same syntax as C.
4. **const_comnts** – not needed since the comments are simply included in the VIDF file, using the same syntax as C.

While the same information is contained in both VIDF formats, some of the information has been restructured in the token-tagged VIDF file in the following manner:

1. **sen_name, d_type, tdw_len, sen_status**, and **time_off** – these fields are grouped together in a structure that is defined to hold all information pertinent to each individual IDFS sensor.
2. **pa_format**, **pa_project**, **pa_mission**, **pa_exper**, **pa_inst**, **pa_vinst**, **pa_bxbybz**, **pa_apps**, **pa_tbls** and **pa_ops** – these fields are grouped together in a structure that is defined to hold all pitch angle information.
3. **status_names** and **states** – these fields are grouped together in a structure that is defined to hold all information pertinent to each individual instrument status byte.
4. **cal_names, cal_use, cal_wlen**, and **cal_target** – these fields are grouped together in a structure that is defined to hold all information pertinent to each individual IDFS calibration set.
5. **VIDF Tables definition** - these fields are grouped together in a structure that is defined to hold all information pertinent to each individual table definition.
6. **VIDF Constants definition** - these fields are grouped together in a structure that is defined to hold all information pertinent to each individual constant definition.

The token-tagged VIDF format defines one additional field (**version**) that is not contained within the fixed-formatted VIDF ASCII file.

When it exists, the token-tagged VIDF file is named according to the filename convention defined in section 1.6 appended by “v3”. Each token-tagged VIDF file consists of a set of fields, each field occupying a single line. Comments are allowed and are defined as any text following the “**/*** ” symbols and ending with the “***/**” symbols. Comments may be continued across multiple lines. The comment will be considered complete when the “***/**” combination is encountered.

Starting with version 3.0 of the token-tagged VIDF file, all VIDF information is kept in a structure like format. The token-tagged VIDF file starts with the keyword "**vidf**" and then the virtual instrument name followed by an open brace ("**{**"). All other information is below the opening brace. The token-tagged VIDF file is ended with a closure brace ("**}**") indicating the end of the file. Each section is broken up into another enclosing set of braces. Entries within each section contain five fields. These five fields are: 1) type, 2) name, 3) equal sign, 4) value and 5) semicolon. Type describes the class of variable name. Valid types must be one of the four values listed below. The name field contains a keyword which is defined in the description of each field in section 4. The equal sign field is the assignment operator and is a "=". The value field contains the quantity to assign to the name keyword. Values enclosed within double quotes (") are taken as character strings, while those enclosed within apostrophes (') are interpreted as single characters. Numeric values without a decimal have an integer value and those with a decimal have a real or float value. Examples may be found below under valid types.

Valid types are:

- string (e.g. string mission = "D1HE";)
- int (e.g. int s_year = 1940;)
- float (e.g. float version = 3.0;)

For example:

```
vidf ABCD {
    float version = 3.0;
    int n_sensors = 1;
    struct Sensor0 {
            string name = "First Sensor";
            int d_type = 0;
            int status = 1;
            int tdw_len = 8;
            int time_offset = 0;
    };
}
```

Entries within a block may be in any order and values may also be in any order or even absent; however, the order of appearance of some fields may be critical in obtaining a proper number result. Any omitted entries may cause the plotting / analysis software to fail.

If this seems unclear, examples are provided for each of the VIDF fields described in section 4. Both a fixed-formatted VIDF and a token-tagged VIDF entry is provided for each field.

4. Fields Common to Fixed-Formatted and Token-Tagged VIDFs

All of the fields that are defined in the fixed-formatted VIDF file are identically defined in the token-tagged VIDF file; however, the converse is not true. There are a few fields that have been added for the token-tagged VIDF files. These fields will be described in section 5. This section will describe in detail the individual fields common to both VIDF file formats. Where appropriate, fields which are linked together by a common basis will be grouped together under a single heading. Sample VIDF entries are included for each field or group of fields for both the fixed-formatted and token-tagged VIDF files. These are shown exactly as they would appear in the ASCII VIDF file.

For fixed-formatted VIDF files, keep in mind that while the detailed descriptions are not necessarily in the order that they appear in the table described in section 2, the fields MUST be in the order outlined in section 2 in the actual ASCII VIDF file. In addition, all fields MUST have an entry (use null, **n**, if not applicable).

4.1 LINEAGE

There are four fields within the VIDF that document the lineage of the virtual instrument, including the virtual instrument itself. The lineage allows one to trace a virtual instrument back to its roots so to speak. It should be noted that the lineage of a virtual instrument within the VIDF is slightly different than the lineage of a virtual instrument within the software that has been developed to interface with the IDFS (IDFS database and access routines). The IDFS archival, catalog, and data access software operate on a five field lineage, adding an *instrument* field between the *experiment* and *virtual instrument* fields in the VIDF. The four VIDF lineage fields are described below.

4.1.1 project

A maximum 79 character description which identifies the particular project with which the virtual instrument is associated. It is meant to be a brief global description of the goal or goals of the project which is generally derived from the project acronym. This field should be identical for all virtual instruments associated with instruments within this project. The project acronym as used in the VIDF lineage should be included.

4.1.2 mission

A maximum 79 character description which identifies the particular mission within a project with which the virtual instrument is associated. In many projects there is only a single mission and the project and mission name and descriptions are identical. This field should be identical for all virtual instruments associated with instruments in this mission. The mission acronym as used in the VIDF lineage should be included.

4.1.3 experiment

A maximum 79 character description which identifies the particular experiment within a mission from which the virtual instrument is derived. This field should be identical for all virtual instruments which have been derived from this experiment. The experiment acronym as used in the VIDF lineage should be included.

4.1.4 v_inst

A maximum 79 character description of the virtual instrument and its contents. The virtual instrument acronym as used in the VIDF lineage should be included.

4.1.5 Example LINEAGE Entries

Shown below are two example VIDF lineage field entries for a fixed-formatted VIDF file. The lineage entries are contiguous within the VIDF. The first example is for an engineering virtual instrument from the UARS satellite and the second example is for an electromagnetic wave spectrometer from the TSS-1 satellite.

```
t    Upper Atmospheric Research Satellite (UARS)                  /*  project    */
t    Initial UARS Flight (UARS-1)                                 /*  mission    */
t    UARS Particle Environment Monitor (PEM)                      /*  exper      */
t    PEM 2 Second Current Monitors (ENIA)                         /*  v_inst     */

t    Tethered Satellite System (TSS)                              /*  project    */
t    First Tethered Satellite System Flight (TSS-1)               /*  mission    */
t    Satellite Plasma and Electromagnetic Wave Inst (RETE)        /*  exper      */
t    Band A: Low Frequency Electromagnetic Wave Data (RTLA)       /*  v_inst     */
```

The same two examples for a token-tagged VIDF file are as follows:

```
string mission = "Upper Atmospheric Research Satellite (UARS)";          /*  mission     */
string spacecraft = "Initial UARS Flight (UARS-1)";                      /*  spacecraft  */
string experiment = "UARS Particle Environment Monitor (PEM)";           /*  exp_desc    */
string instrument = "PEM 2 Second Current Monitors (ENIA)";              /*  imst_desc   */

string mission = "Tethered Satellite System (TSS)";                      /*  mission     */
string spacecraft = "First Tethered Satellite System Flight (TSS-1)";    /*  spacecraft  */
string experiment = "Satellite Plasma and Electromagnetic Wave Inst (RETE)";  /*  exp_desc    */
string instrument = "Band A: Low Frequency Electromagnetic Wave Data (RTLA)"; /*  inst_desc   */
```

4.2 CONTACT Information

4.2.1 contact

This is a set of five lines, each a maximum of 79 characters, which contains the name and address of someone who can act as a focus for questions which might arise concerning the experiment, the data, or the design of the VIDF. If less than five lines are utilized, an empty definition must be provided for the missing lines.

4.2.2 Example CONTACT Information Entries

Shown below are two example VIDF contact field entries for a fixed-formatted VIDF file.

```
m   5   1                                                   /*  contact     */
t   Dr. David Winningham                                    /*  000         */
t   Southwest Research Institute                            /*  001         */
t   6220 Culebra Road                                       /*  002         */
t   San Antonio, Texas 78238-0510                           /*  003         */
t   PH: (210) 522-3075 email:david@cluster.space.swri.edu   /*  004         */

m   5   1                                                   /*  contact     */
t   Dr. David Winningham                                    /*  000         */
t   Southwest Research Institute                            /*  001         */
t   6220 Culebra Road                                       /*  002         */
t   San Antonio, Texas 78238-0510                           /*  003         */
t                                                           /*  004         */
```

The same two examples for a token-tagged VIDF file are as follows:

```
string  contact = “Dr. David Winningham”;
string  contact = “Southwest Research Institute”;
string  contact = “6220 Culebra Road”;
string  contact = “San Antonio, Texas 78238-0510”;
string  contact = “PH: (210) 522-3075 email:david@cluster.space.swri.edu”;

string  contact = “Dr. David Winningham”;
string  contact = “Southwest Research Institute”;
string  contact = “6220 Culebra Road”;
string  contact = “San Antonio, Texas 78238-0510”;
string  contact = “”;
```

4.3 COMMENTS

The free-form comments portion of the VIDF is comprised of two fields which are described below.

4.3.1 num_comnts

The number of comment lines within the comment field.

4.3.2 comments

A set of free-form text, each line being a maximum of 79 characters in length. The number of lines of text in the comment field is defined in the **num_comnts** entry. Comment lines may be used for any general documentation. At a minimum, the comment field should contain the five field IDFS software reference, any caveats and limitations of the data set, a reference to an experimental paper, any algorithms which are defined for the conversion of the raw VIDF data into physical units and the physical units. If the data is already in physical units and no conversion is necessary the units of the data should be given.

4.3.3 Example COMMENTS Entry

Shown below is an example VIDF comment block entry for a fixed-formatted VIDF file for an electron spectrometer which was flown on the Tethered Satellite. Note the detail in describing the algorithms available. For each algorithm, the type of data that is affected, which tables are to be applied, what mathematical operations are associated with each table, and the resulting units after conversion are described. Although the table information is needed by the IDFS data access routine for proper unit conversion, this information is NOT accessed through the VIDF, but rather the applications programs need to provide the IDFS data access routines with this information based on the comments given here. SwRI has developed a "Plot (or Applications) Interface Definition File" (PIDF) format and interface which includes the ability to store the table order / operator information and for applications programs to access this information as well as other useful information. The PIDF is intended to work in conjunction with the VIDF. See *http://www.idfs.org/Editors/pidfdoc.html* for more information about the PIDF. To understand how tables are applied and available operations, see the "TABLE BLOCK" section.

```
s   31                                                              /*  num comnts  */
m   31  1                                                           /*  comment     */
t                          TSS TSS-1 ROPE ROPE RPEA                 /*  000         */
t                                                                   /*  001         */
t   The ROPE SPES particle data are known to have two sources of noise.  /*  002  */
t   One is a light leak and the 2nd is a plasma entry not through the    /*  003  */
t   collimators.  Both occur as localized positions in the orbit and neither  /*  004  */
t   is detrimental to the data.                                     /*  005         */
t                                                                   /*  006         */
t   The following is a list of tables which are in this VIDF        /*  007         */
t   TABLE 0:  center energies (eV)                                  /*  008         */
t   TABLE 1:  telemetry decom table                                 /*  009         */
t   TABLE 2:  1/(detector efficiencies)                             /*  010         */
t   TABLE 3:  geometry factors (cm**2-str)                          /*  011         */
t   TABLE 4:  dE/E                                                  /*  012         */
t   TABLE 5:  center energies (ergs)                                /*  013         */
t   TABLE 6:  constant needed in going to dist. fn                  /*  014         */
t   TABLE 7:  (center energies)**2 (ergs**2)                        /*  015         */
t   TABLE 8:  ascii definitions of status states                    /*  016         */
t                                                                   /*  017         */
t   The following units can be derived from the tables.             /*  018         */
t   The format is to give the tables applied followed by the        /*  019         */
t   operations and unit definitions                                 /*  020         */
t                                                                   /*  021         */
t    DATA      TABLES        OPERS              UNIT                /*  022         */
t   Scan       0             0                  eV                  /*  023         */
t   Sensor     1             0                  cnts/accum          /*  024         */
t   Sensor     1,2           0,3                cnts/accum (eff. cor)  /*  025      */
t   Sensor     1,2           0,153              cnts/sec            /*  026         */
t   Sensor     1,2,3,4       0,153,4,4          cnts/(cm**2-str-s)  /*  027         */
```

```
t   Sensor    1,2,3,4,0     0,153,4,4,4     cnts/(cm**2-str-s-eV)     /*  028     */
t   Sensor    1,2,3,4,0,5   0,153,4,4,4,3   ergs/(cm**2-str-s-eV)     /*  029     */
t   Sensor    1,2,3,4,6,7   0,153,4,4,3,4   sec**3/km**6              /*  030     */
```

The same example for a token-tagged VIDF file is as follows:

```
/*
 *                              TSS TSS-1 ROPE ROPE RPEA
 *
 *    The ROPE SPES particle data are known to have two sources of noise.
 *    One is a light leak and the 2nd is a plasma entry not through the
 *    collimators.  Both occur as localized positions in the orbit and neither
 *    is detrimental to the Data.
 *
 *    The following is a list of tables which are in this VIDF
 *    TABLE 0:  center energies (eV)
 *    TABLE 1:  telemetry decom table
 *    TABLE 2:  1/(detector efficiencies)
 *    TABLE 3:  geometry factors (cm**2-str)
 *    TABLE 4:  dE/E
 *    TABLE 5:  center energies (ergs)
 *    TABLE 6:  constant needed in going to dist. fn
 *    TABLE 7:  (center energies)**2 (ergs**2)
 *    TABLE 8:  ascii definitions of status states
 *
 *    The following units can be derived from the tables.
 *    The format is to give the tables applied followed by the
 *    operations and unit definitions
 *
 *    DATA    TABLES      OPERS              UNIT
 *    Scan    0           0                  eV
 *    Sensor  1           0                  cnts/accum
 *    Sensor  1,2         0,3                cnts/accum (eff. cor)
 *    Sensor  1,2         0,153              cnts/sec
 *    Sensor  1,2,3,4     0,153,4,4          cnts/(cm**2-str-s)
 *    Sensor  1,2,3,4,0   0,153,4,4,4        cnts/(cm**2-str-s-eV)
 *    Sensor  1,2,3,4,0,5 0,153,4,4,4,3      ergs/(cm**2-str-s-eV)
 *    Sensor  1,2,3,4,6,7 0,153,4,4,3,4      sec**3/km**6
 */
```

4.4 TIME Information

Each VIDF file has associated with it a beginning and an ending time. This time describes the time period over which the information in the VIDF is valid for the virtual

instrument. The times are given in two sets of identical data fields describing the beginning time and ending time. Each set of fields has the general format:

year / day / millisecond of day / microsecond

It is normal to define the first and last VIDF files defined for a virtual instrument to have a beginning time earlier than the start of the accumulation of data and an ending time later than the instrument turn off.

The start and stop time definitions are defined in the eight fields below, and the entries are contiguous in the VIDF in the order given below.

4.4.1 ds_year

The starting year field with valid entries running from 1 to 9999

4.4.2 ds_day

The starting day field with valid entries running from 1 to 365 or 366 depending if the year is a leap year or not.

4.4.3 ds_msec

The starting millisecond of day field with valid entries running from 1 to 86400000.

4.4.4 ds_usec

The starting microsecond field. This is an offset from the millisecond of day field such that seconds of day is obtained by:

$$sec = ds\_msec * 10^{-3} + ds\_usec * 10^{-6}$$

4.4.5 de_year

The ending year field with valid entries running from 1 to 9999. If only one (1) VIDF is to be used for the whole duration of the mission, set this value to -1 to expedite processing.

4.4.6 de_day

The ending day field with valid entries running from 1 to 365 or 366 depending if the year is a leap year or not. If only one (1) VIDF is to be used for the whole duration of the mission, set this value to -1 to expedite processing.

4.4.7 de_msec

The ending millisecond of day field with valid entries running from 1 to 86400000. If only one (1) VIDF is to be used for the whole duration of the mission, set this value to -1 to expedite processing.

4.4.8 de_usec

The ending microsecond field. This is an offset from the millisecond of day field such that seconds of day is obtained by:

$$sec = de\_msec * 10^{-3} + de\_usec * 10^{-6}$$

If only one (1) VIDF is to be used for the whole duration of the mission, set this value to -1 to expedite processing.

4.4.9 Example TIME Information Entry

Shown below is an example set of VIDF time information fields. The data actually begins in 1981, but the example shows an earlier beginning time (which is okay and normal to do). At the ending time there is a change in detector efficiency that is addressed by a second VIDF which begins at the ending time of this VIDF. Note that it is normal and desirable for the ending time to be several years after the predicted mission termination if this VIDF is the last one for the virtual instrument. In most cases, there is only one VIDF defined for the duration of the virtual instrument's mission.

```
s   1980                /*   ds_year    */
s   1                   /*   ds_day     */
l   0                   /*   ds_msec    */
s   0                   /*   ds_usec    */
s   1994                /*   de_year    */
s   52                  /*   de_day     */
l   1240                /*   de_msec    */
s   860                 /*   de_usec    */
```

The same example for a token-tagged VIDF file is as follows:

```
int   s_year = 1980;    /*   ds_year    */
int   s_day = 1;        /*   ds_day     */
int   s_msec = 0;       /*   ds_msec    */
int   s_usec = 0;       /*   ds_usec    */
int   e_year = 1994;    /*   de_year    */
int   e_day = 52;       /*   de_day     */
int   e_msec = 1240;    /*   de_msec    */
int   e_usec = 860;     /*   de_usec    */
```

4.5 SENSOR Information

The IDFS sensors are described within the VIDF through a set of seven fields, which are scattered throughout the VIDF file. These fields have two purposes; (1) to establish the interface and control parameters necessary for the IDFS data access software to seamlessly interface with the IDFS, and (2) to provide a definition of what is contained within each sensor. The seven

sensor related fields – **smp_id**, **swp_len**, **sen**, **sen_name**, **d_type**, **tdw_len** and **sen_status** - are described below.

4.5.1 smp_id

This field defines the type of sensor associated with the virtual instrument. A virtual instrument can only contain one type of sensor. The three recognized sensor types are defined in the table below.

SMP_ID FIELD DEFINITIONS	
VALUE	DEFINITION
0	Partial vector data
1	Full vector data
2	Scalar data

A vector sensor differs from a scalar sensor in that it has known functional dependencies other than time or position. The IDFS can accommodate one of these functional dependencies through the **scan_index** field in the header record which is nothing more than a set of indices into an optional look-up table which must be defined within the VIDF. The length of the **scan_index** array is identical to the number of elements in the vector being returned. An example of a vector sensor is one made from a particle spectrometer which returns counts as a function of energy in eV. The data has the form CR(eV) and the values in the **scan_index** are offsets into a table containing the center energy at which each data sample was obtained. The base measurement for a vector sensor in this case is one scan of data.

If only fractional portions of the vectors are stored under any given sensor then the IDFS is defined to be stored as "partial vector data". The "full vector data" definition is used when the entire sweep is stored.

4.5.2 swp_len

This field defines the maximum number of vector elements which can be defined for any sensor and is not necessarily the vector length of the sensors returned in the IDFS data record. The value is used when determining the length of look-up tables associated with the sensor scan index. Another way of putting this is, the **swp_len** field should be set to one more than the maximum value which could be placed in the **scan_index** array in the IDFS header record.

As an example, a vector sensor always returns 32 elements which are indexed from 0 through 63. It returns either the odd or the even 32 elements in the set. In this case, **swp_len** must be set to 64 even though the sensor will never return 64 elements in a single data record. This is because the **scan_index** array will have values 0, 2, 4 ... 62 when the even elements are returned and 1, 3, 5 ... 63 when the odd elements are returned.

For scalar virtual instruments (see **smp_id**) **swp_len** should be set to 1, and for scanning instruments **swp_len** should be greater than 1.

4.5.3 sen

This field defines the maximum number of sensors which will be defined in the VIDF. Any or all of these sensors may be returned within the IDFS data records. Not all of the sensors which are defined need to be returned.

4.5.4 sen_name

This field is an array of **sen** text fields each a maximum of 79 characters in length. Each field is a description of one of the sensors defined within the VIDF. The order of the definitions in the array is assumed to correspond to the sensor number. The first text field describing sensor 0 and so forth. These numbers are used to indicate in the IDFS header record field, **sensor_index**, which sensors have data being returned in a particular data record.

4.5.5 d_type

This field is an array of sen elements which indicates the data formats associated with each sensor, that is, how the data is represented within the IDFS data files for each of the defined IDFS sensors. This allows sensors within the same virtual to have different data formats. This field does not apply to any calibration data which may also be stored within the IDFS data files. Unless otherwise specified, calibration data is always interpreted as unsigned integer, binary data (refer to section 5.7.3). The seven recognized formats are listed below: The floating point IDFS representations are detailed in the next section.

D_TYPE FIELD DEFINITIONS			
VALUE	DEFINITIONS	EXPONENT BASE	WORD LENGTH (bits)
0	unsigned integer, binary data	-	**tdw_len**
1	signed integer, binary data	-	**tdw_len**
2	single precision, floating point data	10	32
3	double precision, floating point data	10	64
4	half precision 1, floating point data	10	16
5	half precision 2, floating point data	2	16
6	half precision 3, floating point data	2	16

4.5.5.1 IDFS Floating Point Formats

Within the IDFS, all floating point values are stored in internally defined formats that are expanded upon to use the native floating point format of the computer on which the data is being extracted.

There are several common points between the IDFS floating point formats. In each format, the mantissa has an inherent decimal point to the LEFT of the first digit. Relative motion of the decimal point from its default location is controlled by the exponent. Both the exponent and mantissa have their sign bits in the most significant bit of their respective bit fields, unless otherwise specified, where a 0 bit value represents the positive sign and a 1 bit value represents a negative sign. The floating point storage formats are described below.

The *single precision floating point data* is stored as a 32-bit integer according to the following format:

Mantissa																Exponent							
Byte 3								Byte 2								Byte 1							
31																7	6						0

The mantissa is formed by the most significant 25 bits giving 7 digits of precision (0 to ±9999999). All seven digits are used in the representation of any mantissa. The exponent is located in the least significant 7 bits of the 32-bit word and has a range of ±63. Under these guidelines, 1.57 would be written as a mantissa of +1570000 and an exponent of +1.

The *double precision floating point data*, which has yet to be implemented in the IDFS data access software, will be stored as a 64-bit integer according to the following format:

Mantissa																																Exponent							
Byte 7								Byte 6								Byte 5								Byte 4								Byte 3								Byte 2								Byte 1								Byte 0							
63																																						9	8	7					0

The mantissa is formed by the most significant 55 bits giving 16 digits of precision (0 to ±9999999999999999). All sixteen digits are used in the representation of any mantissa. The exponent is located in the least significant 9 bits of the 64-bit field and has a range of ±255. Under these guidelines, -9.9734×10^{-6} would be written as a mantissa of -9973400000000000 and an exponent of -5.

There are 3 half precision floating point formats. The difference between them is in the accuracy of the mantissa and the range of the exponent. Exponents differ in their base, being either base 10 or base 2. The latter give a smaller exponent range but better representation of the floating point values being stored.

Both the *half precision 1* and *half precision 2 floating point data* (**d_type** 4 and **d_type** 5) are stored as 16-bit integers. They are defined similar to the 32-bit single precision float with the exception that the mantissa is only 9 bits in width. *Half precision 1* uses a base-10 exponent representation, and *half precision 2* uses a base-2 exponent representation. The base-10 representation has a larger range with less accuracy, while the base-2 representation has a smaller range with greater precision. The storage format of these two data representations is shown below.

Mantissa									Exponent							
Byte 1									Byte 0							
15									7	6						0

The mantissa is formed by the most significant 9 bits with a range of ±255. The exponent is located in the least significant 7 bits of the 16-bit word and has a range of ±63.

The *half precision 3 floating point data* (**d_type** 6) is stored as a 16-bit integer with 9 bits for the mantissa and 7 bits for the exponent. The exponent uses a base-2 representation and has the same dynamic range and accuracy as the *half precision 2 floating point data*. The difference between the two is in the storage format. The storage format for *half precision 3* is shown below.

Signs		Exponent			Mantissa		
Byte 1					Byte 0		
15	14	13		8	7		0

The mantissa is formed by the least significant 8 bits and a range of ±255. The sign of the mantissa is represented in bit 14. The exponent is located in bits 8-13 and has a range of ±63. Bit 15 represents the exponent sign.

There are three error conditions that are recognized by the floating point conversions and are indicated in the 0 state of the integer representation (0 mantissa magnitude, 0 exponent magnitude). The four possible zero states are shown below together with the conditions that they represent and the handling by the IDFS data acquisition routines.

0 STATE FLOAT: MANTISSA AND EXPONENT MAGNITUDES = 0			
MANTISSA SIGN	EXPONENT SIGN	CONDITION	GENERIC ACQUISITION RETURN VALUE
+	+	valid data	0.0
+	-	not a number	0.0
-	+	positive infinity	largest positive value
-	-	negative infinity	largest negative value

4.5.6 tdw_len

This field is an array of **sen** elements which gives the bit lengths associated with data of each sensor as it is stored in the IDFS data record. Data stored under **d_type** 0 or 1 can have any bit lengths up to 32 bits. Data stored under **d_type** 2 through 6 must have the bit length indicated under the **d_type** definition for them.

In the IDFS data record, the data corresponding to each sensor can be stored with different bit lengths, however, physically the data is stored in 1, 2, 4, 8, 16, or 32 bit lengths. Note that the 64-bit length (double precision, floating point) is not supported yet. There is a single base data length used for all data (sensor and calibration) within an IDFS data file. The base word length is defined by rounding up the largest defined sensor **tdw_len** value and calibration **cal_wlen** value to one of the "physical" lengths defined above.

4.5.7 sen_status

An array of **sen** elements which gives the status of each sensor. There are three defined states:

0 - sensor is inoperative and any data returned should be ignored
1 - sensor is operating nominally
3 - sensor is operating erratically and data may be questionable

Note that the IDFS data access software does not use the information in the **sen_status** field at this time.

4.5.8 Example SENSOR Information Entry

Shown below is an example of the VIDF sensor information entries for a fixed-formatted VIDF file for a virtual instrument containing three scalar sensors. Note that these entries are not contiguous in the VIDF, except for the last three (**d_type, tdw_len, sen_status**). The non-contiguous entries are separated with lines in the example below.

```
b   2                                   /*  smp_id      */
s   3                                   /*  sen         */
s   1                                   /*  swp_len     */
m   3  1                                /*  sen_name    */
t   Potential Calculation Method        /*  00000       */
t   Average Spacecraft Potential        /*  00001       */
t   Variance                            /*  00002       */
m   3  3                                /*  d_type      */
b   0        0        0                 /*  0000 - 0002 */
m   3  3                                /*  tdw_len     */
b   4        8        8                 /*  0000 - 0002 */
m   3  3                                /*  sen_status  */
b   1        1        1                 /*  0000 - 0002 */
```

In the above example (**tdw_len**), the base data length will be 8 bits, and thus, the data for sensor 0 should be written in the data file as an 8-bit quantity where the 4 most significant bits will be ignored by the IDFS data access software.

The same example for a token-tagged VIDF file is as follows. The non-contiguous entries are separated with lines in the example below.

```
int     smp_id = 2;                                 /*  smp_id      */
int     n_sensors = 3;                              /*  sen         */
int     swp_len = 1;                                /*  swp_len     */
struct  Sensor0 {
        string name = "Potential Calculation Method"; /*  name      */
        int d_type = 0;                             /*  d_type      */
        int status = 1;                             /*  status      */
        int tdw_len = 4;                            /*  tdw_len     */
        int time_offset = 0;                        /*  time_offset */
};
struct  Sensor1 {
        string name = "Average Spacecraft Potential"; /*  name      */
        int d_type = 0;                             /*  d_type      */
        int status = 1;                             /*  status      */
        int tdw_len = 8;                            /*  tdw_len     */
        int time_offset = 0;                        /*  time_offset */
```

```
};
struct    Sensor2 {
          string name = "Variance";          /*  name        */
          int d_type = 0;                    /*  d_type      */
          int status = 1;                    /*  status      */
          int tdw_len = 8;                   /*  tdw_len     */
          int time_offset = 0;               /*  time_offset */
};
```

Note that for a token-tagged VIDF file, information that is defined per sensor is grouped into a Sensor block structure. There are 5 elements within the Sensor block structure, four of which have been defined in this section and are identified by bold-faced text in the comment string. The last element (**time_offset**) is defined in section 4.6.3.

4.6 DATA TIMING Information

General timing information within the IDFS is defined through a set of three fields in the VIDF coupled with active timing information in the IDFS header and data files. These three VIDF fields, which are described below, are not contiguous within the VIDF.

4.6.1 sen_mode

This field defines how time advances within a sensor set. A sensor set is a 2D matrix of data in which each column holds data from a single vector or scalar sensor. Within a sensor set, time can advance either across the rows or down the columns. In addition, time can advance sequentially, that is element by element, or in parallel, all elements within a row or column advancing at the same time. For a given virtual instrument, time can advance in one prescribed manner.

The recognized values for the **sen_mode** field together with a brief description of each are given in the table below.

SEN_MODE DEFINITIONS			
SEN_MODE VALUE	TIME ADVANCES IN ROW	TIME ADVANCES IN COLUMN	TIME ADVANCES
0	sequential	sequential	down column
1	sequential	parallel	down column
2	parallel	sequential	down column
3	parallel	parallel	down column
4	sequential	sequential	across row
5	sequential	parallel	across row
6	parallel	sequential	across row
7	parallel	parallel	across row

Note that several of the definitions in the above table are redundant. As an example, if the data in both the row and column are taken in parallel then as far as the timing goes, it doesn't matter whether time is advancing across the rows (**sen_mode** = 7) or down the columns (**sen_mode** = 3).

The usage of this field is illustrated through several examples. To begin, the table below illustrates the generic picture of a sensor set which contains five sensors each with six data samples. This sensor set will form the basis for the examples below.

EXAMPLE SENSOR SET					
SENSORS → DATA ↓	SEN 0	SEN 1	SEN 2	SEN 3	SEN 4
DATA 1					
DATA 2					
DATA 3					
DATA 4					
DATA 5					
DATA 6					

If the sensor set is defined to have a **sen_mode** of either 2 or 6, the timing runs as shown below. Here, T1 through T6 are successive times.

EXAMPLE SENSOR SET (SEN_MODES 2 or 6)					
SENSORS → DATA ↓	SEN 0	SEN 1	SEN 2	SEN 3	SEN 4
DATA 1	T1	T1	T1	T1	T1
DATA 2	T2	T2	T2	T2	T2
DATA 3	T3	T3	T3	T3	T3
DATA 4	T4	T4	T4	T4	T4
DATA 5	T5	T5	T5	T5	T5
DATA 6	T6	T6	T6	T6	T6

In the above picture, the data for each sensor is taken at the same time, in parallel. Time progresses down the columns of the sensor set, each row containing data taken at a successively later time. Since the data is taken in parallel across the sensor set rows, it does not matter if time advances down the column or across the row -- the same time for any given element will be arrived at in each case. Thus, the redundancy between **sen_mode** 2 and 6.

The next example shows a sensor set defined to have a **sen_mode** of either 1 or 5. Again, T1 through T5 are successive times.

EXAMPLE SENSOR SET (SEN_MODES 1 or 5)					
SENSORS → DATA ↓	SEN 0	SEN 1	SEN 2	SEN 3	SEN 4
DATA 1	T1	T2	T3	T4	T5
DATA 2	T1	T2	T3	T4	T5
DATA 3	T1	T2	T3	T4	T5
DATA 4	T1	T2	T3	T4	T5
DATA 5	T1	T2	T3	T4	T5
DATA 6	T1	T2	T3	T4	T5

Now all measurements within a sensor column are taken at the same time and time advances from column to column, that is from sensor to sensor. These modes (**sen_mode** 1 and **sen_mode** 5) should not be used for scalar sensors, which generally make sequential point measurements.

The last two examples below show successive times running from T1 to T30 (**sen_mode** 0 and 4).

EXAMPLE SENSOR SET (SEN_MODE 0)					
SENSORS → DATA ↓	SEN 0	SEN 1	SEN 2	SEN 3	SEN 4
DATA 1	T1	T7	T13	T19	T25
DATA 2	T2	T8	T14	T20	T26
DATA 3	T3	T9	T15	T21	T27
DATA 4	T4	T10	T16	T22	T28
DATA 5	T5	T11	T17	T23	T29
DATA 6	T6	T12	T18	T24	T30

When **sen_mode** is set to 0, the sensors make measurements one after the other where time advances down the columns. Data is acquired column by column such that the data for sensor 0 is acquired totally before the data for sensor 1 is acquired, etc.

EXAMPLE SENSOR SET (SEN_MODE 4)					
SENSORS → DATA ↓	SEN 0	SEN 1	SEN 2	SEN 3	SEN 4
DATA 1	T1	T2	T3	T4	T5
DATA 2	T6	T7	T8	T9	T10
DATA 3	T11	T12	T13	T14	T15
DATA 4	T16	T17	T18	T19	T20
DATA 5	T21	T22	T23	T24	T25
DATA 6	T26	T27	T28	T29	T30

When **sen_mode** is set to 4, the sensors make measurements one after the other in time across the sensor set rows. Data is acquired sensor by sensor as time advances across the rows.

4.6.2 da_method

This field describes the format of how time changes down the column of a sensor set when dealing with a vector sensor. It has no meaning when dealing with scalar sensors or when dealing with vector sensors if **sen_mode** indicates that the column timing is parallel (**sen_mode** 1, 3, 5, or 7). In these cases **da_method** should be set to 0.

Before going over this section it is beneficial to review the definitions of **data_accum**, **data_lat**, **n_sample**, and **scan_index** fields which are given in the section on the IDFS header record.

For a vector instrument, there are four recognized types of data acquisition which are described below. The value introducing each definition is the appropriate **da_method** value to use in the VIDF.

0 - Each element in the vector is acquired within the time **data_accum** and the time between successive elements is given by

$$\Delta t = \textbf{data\_accum} + \textbf{data\_lat}$$

Each element in the vector is assumed to be acquired sequentially, one after the other despite possible discontinuities in the **scan_index** values. The time contained within a complete vector is then

$$\Delta T = \textbf{n\_sample} * \Delta t$$

As an example: A vector sensor has 10 elements whose **scan_index** values are 1, 5, 9, ... 37. The beginning acquisition of first element (step 1) is found to be T_0. The beginning time of the second element (step 5) is then $T_0 + \Delta t$, the beginning time of the third element (step 9) is $T_0 + 2\Delta t$ and so on. The time contained within the complete vector is $10 * \Delta t$.

1 - Each element in the vector is acquired within the time **data_accum** and the time between successive elements is given by

$$\Delta t = \textbf{data\_accum} + \textbf{data\_lat}$$

The difference between this and the above situation is that it is assumed that all possible sweep steps are acquired but only a subset have been returned. The steps not returned then form an effective dead time or additional data latency between the returned steps. The time contained within a complete vector is then

$$\Delta T = \textbf{swp\_len} * \Delta t$$

Using the same example as above: A vector sensor has 10 elements returned out of 64 possible (**swp_len**). The **scan_index** values are 1, 5, 9, ... 37. The beginning time of the sweep is found to be T_0. The beginning acquisition time of first element (step

1) is then $T_0 + \Delta t$, the beginning time of the second element (step 5) is $T_0 + 5\Delta t$, the beginning time of the third element (step 9) is $T_0 + 9\Delta t$ and so on. The time contained within the complete vector is 64 * Δt.

2 - Each element in the vector is acquired within the time **data_accum** and the time between successive elements is given by

$$\Delta t = \textbf{data\_accum} + \textbf{data\_lat}$$

This situation is very similar to the situation defined for **da_metho**d of 1, with one distinction. In this situation it is assumed that all of the possible sweep steps are acquired, but only a subset have been returned. The subset sampled is contained between the first and last steps defined within the **scan_index** array. Again, the steps not returned in the data form an effective dead time or additional data latency between the returned steps. The time contained within a complete vector is then

$$\Delta T = \Delta t * \text{NSTEPS}$$

where

$$\text{NSTEPS} = \textbf{scan\_index}[\textbf{n\_sample}\text{-}1] - \textbf{scan\_index}[0] + 1$$

Returning to the same example as above: A vector sensor has 10 elements returned out of 64 possible (**swp_len**). The **scan_index** values are 1, 5, 9, ... 37. The beginning acquisition time of first element (step 1) is found to be T_0. The beginning time of the second element (step 5) is then $T_0 + 4\Delta t$ (5-1 = 4), the beginning time of the third element (step 9) is $T_0 + 8\Delta t$ (9-1 = 8), and so on. The time contained within the complete vector is 37 $*$ Δt (37 - 1 + 1 = 37).

3 - This definition of **da_method** is restricted to vector sensors whose elements are evenly spaced (SKIP) within the total number of scan steps available. This is equivalent to requiring that each element in the **scan_index** be able to be determined by an algorithm of the form:

$$\textbf{scan\_index}[J] = J * \text{SKIP} + \textbf{scan\_index}[0]$$

Under this definition each element in the vector is acquired within the time **data_accum** $*$ SKIP and the time between successive elements is given by

$$\Delta t = \text{SKIP} * \textbf{data\_accum} + \textbf{data\_lat}$$

The time contained within a complete vector is then

$$\Delta T = \textbf{n\_sample} * \Delta t$$

This is identical to **da_method** of 0 with a different definition of Δt. This defines the situation where the accumulation of a single element is held for the length of time

between its step and the next returned step. This poses a problem for the last step, where there is no "ending" step from which to determine its duration. The last step by definition is assumed to be acquired in **data_accum**. This can be defeated by defining a dummy ending element in the vector whose **scan_index** value is used to define the duration of the true ending value and whose data value is set to the current fill value.

As an example: A vector sensor has 10 elements returned out of 64 possible (**swp_len**). The **scan_index** values are 1, 5, 9, ... 37. The beginning acquisition time of the first element (step 1) is found to be T_0 and

$$\Delta t = 4 * \textbf{data\_accum} + \textbf{data\_lat}$$

The beginning time of the second element (step 5) is then $T_0 + \Delta t$, the beginning time of the third element (step 9) is $T_0 + 2\Delta t$ and so on. The time contained within the complete vector is $10 * \Delta t$.

4.6.3 time_off

This is an array of **sen** elements which holds a set of offsets to be applied to the derived time for any given measurement for each sensor. The offsets are given in milliseconds and can be positive or negative.

These offsets allow for small corrections to the sensor timing which may arise from internal buffering of data in an instrument or any other effects. A specific example of the usage of this field is the following: A set of voltages and currents are returned from an experiment and would like to be kept as a set of data in a single IDFS. They are, however, taken at slightly different times within a block of transmitted data, their time of acquisition actually being determined by their position in the data stream. The data can be kept together by declaring all sensors within the IDFS to take their data in parallel at times corresponding to the beginning of the data blocks and then using the **time_off** field to correctly offset this time to the actual acquisition times.

4.6.4 Example DATA TIMING Information Entry

Shown below is an example of the VIDF timing fields for a fixed-formatted VIDF file for a virtual instrument containing three sensors. These entries are not contiguous within the VIDF, so they are separated with lines in the example below.

```
b    2                                  /*  sen_mode     */
b    0                                  /*  da_method    */
m    3   3                              /*  time_off     */
l    10          0          -10         /*  000 - 002    */
```

This virtual instrument has a **sen_mode** of 2 which means the data for all sensors are acquired at the same time (in parallel). However, the **time_off** values indicate that the first sensor's data is actually acquired 10 msecs. after the beginning data block time, the second sensor's data is

acquired at the beginning data block time, and the third sensor's data is actually acquired 10 msecs. before the beginning data block time.

The same example for a token-tagged VIDF file is as follows. The non-contiguous entries are separated with lines in the example below.

```
int     sen_mode = 2;                                   /*  sen_mode      */
int     da_method = 0;                                  /*  da_method     */
struct  Sensor0 {
        string name = "Potential Calculation Method";   /*  name          */
        int d_type = 0;                                 /*  d_type        */
        int status = 1;                                 /*  status        */
        int tdw_len = 4;                                /*  tdw_len       */
        int time_offset = 10;                           /*  time_offset   */
};
struct  Sensor1 {
        string name = "Average Spacecraft Potential";   /*  name          */
        int d_type = 0;                                 /*  d_type        */
        int status = 1;                                 /*  status        */
        int tdw_len = 8;                                /*  tdw_len       */
        int time_offset = 0;                            /*  time_offset   */
};
struct  Sensor2 {
        string name = "Variance";                       /*  name          */
        int d_type = 0;                                 /*  d_type        */
        int status = 1;                                 /*  status        */
        int tdw_len = 8;                                /*  tdw_len       */
        int time_offset = -10;                          /*  time_offset   */
};
```

Note that for a token-tagged VIDF file, information that is defined per sensor is grouped into a Sensor block structure. There are 5 elements within the Sensor block structure, one of which has been defined in this section and is identified by bold-faced text in the comment string. The other four elements are individually defined in section 4.5.

4.7 QUALITY Definitions

Two fields in the VIDF are used to define the values of the quality flags found in the IDFS header record. These fields are not contiguous within the VIDF and are described below.

4.7.1 n_qual

The number of data quality definitions within the VIDF field **qual_name**.

4.7.2 qual_name

An array of **n_qual** lines of text each a maximum of 79 characters in length. Each line of text describes one of the possible numerical quality definitions which are given in the IDFS header record for this particular VIDF. The order of the definitions in the array are assumed to correspond to the numerical values (starting at 0) of the quality flags they define. For example, a numerical value of 0 corresponds to the first definition in the array, a numerical value of 1 corresponds to the second definition, and so on.

4.7.3 Example QUALITY Definintions Entry

Shown below is an example entry of the two fields which comprise the VIDF quality block for a fixed-formatted VIDF file. These entries are not contiguous in the VIDF, so they are separated with a line in the example below.

b	3	/*	n_qual	*/
m	3 1	/*	qual_name	*/
t	No Fill Data In Sensor Set	/*	00000	*/
t	Some Fill Data In Sensor Set	/*	00001	*/
t	Questionable Data In Sensor Set	/*	00002	*/

The same example for a token-tagged VIDF file is as follows. The non-contiguous entries are separated with lines in the example below.

int	n_qual = 3;	/*	n_qual	*/
string	qual_names = “No Fill Data In Sensor Set”;	/*	name	*/
string	qual_names = “Some Fill Data In Sensor Set”;	/*	name	*/
string	qual_names = “Questionable Data In Sensor Set”;	/*	name	*/

It should be noted that there is only one assumed definition for a given quality flag within an IDFS. A quality flag is assigned to each sensor within a given sensor set, so if sensors need to have different quality definitions, they should be given unique values.

4.8 CALIBRATION SET Information

The descriptions of the calibration sets which may be attached to each of the sensors are contained in five fields within the VIDF. The five calibration fields are described below.

4.8.1 cal_sets

The total number of calibration sets which are associated with the virtual instrument.

4.8.2 cal_names

An array of **cal_sets** text fields each a maximum of 79 characters in length. Each field is a description of one of the defined calibration sets for this virtual instrument. The order of the definitions in the array is assumed to correspond to the order in which the data for the calibration sets are written to the IDFS data file.

4.8.3 cal_use

An array of **cal_sets** elements which define the number of successive sensor or scan elements (according to **cal_target**) to which each calibration value is to be applied. While it is not necessary for a calibration set to have one value for each sensor measurement defined in a sensor set, it is necessary that each measurement be linked to a calibration set value. It is the function of **cal_use** to describe this linkage.

For vector sensors, **cal_use** can have any value from 0 to the sensor vector length (**swp_len**), while for scalar sensors **cal_use** must be either 0 or 1. The 0 **cal_use** value is reserved and indicates that a single calibration value is present which is to be applied to all sensor measurements in the sensor set.

The best method to illustrate the use of this field is through specific examples.

4.8.3.1 cal_use Example-1

A vector sensor of length 19 has a calibration set associated with it where cal_use is set to 3. This means that each element in the calibration data will apply to 3 elements in the sensor data. The first calibration value will apply to the first 3 sensor elements, the next calibration value to the next 3, and so on. Since there are 19 elements in the sensor vector, there must be 7 calibration values where the last calibration value is used only for the last sensor element. If cal_use is set to 0, then there would be a single calibration value to be applied to all 19 sensor elements.

4.8.3.2 cal_use Example-2

A scalar sensor has 12 successive measurements in each sensor set and a calibration set associated with it where **cal_use** is set to 0. There is then only one calibration value which is applied to each of the 12 sensor measurements. If **cal_use** is set to 1 in this example, then there would be 12 calibration values, one for each measurement.

4.8.4 cal_wlen

This field is an array of **cal_sets** elements which define the bit lengths associated with the data of each calibration set. This is not the base word length used for all data within the data file, but the number of valid bits defined for each calibration set (similar to **tdw_len**). Refer to the description of **tdw_len** in section 4.5.6.

4.8.5 cal_target

An array of **cal_sets** elements which define the target data to which the calibration data applies. The defined values are shown in the table below.

CAL_TARGET DEFINITIONS	
CAL_TARGET	DEFINITION
0	Sensor data
1	Scanning data

Scalar sensors can only have a **cal_target** value of 0 by definition. Note that in the IDFS data record, calibration data must be written with calibration sets targeted to the scan data preceding those that apply to the sensor data. This should be manifested in the **cal_target** entries.

4.8.6 Example CALIBRATION SET Information Entries

Shown below are two examples of the VIDF calibration data block entries. The five fields are disjointed within the VIDF with only the final three fields being contiguous. Note that the non-contiguous VIDF entries are separated with lines.

If there are no calibration data associated with this IDFS, the VIDF calibration entries for a fixed-formatted VIDF file would be as follows:

```
b       0                                                   /*  cal_sets    */
n                                                           /*  cal_names   */
n                                                           /*  cal_use     */
n                                                           /*  cal_wlen    */
n                                                           /*  cal_target  */
```

The same example for a token-tagged VIDF file is as follows:

```
int     n_cal_sets = 0;                                     /*  cal_sets    */
```

The following example shows calibration entries for a virtual instrument that returns vector data. There are 7 defined calibration sets where the first two sets are to be applied to the scan data (**cal_target** = 1), and the last five sets are to be applied to the sensor data (**cal_target** = 0). There is a single value for each calibration set (**cal_use** = 0), where the first two calibration values are to be applied to all elements of the scan data, and the last five values are to be applied to all elements of the sensor data. The first three and the seventh calibration values have bit lengths of 16, and the fourth, fifth, and sixth values have bit lengths of 1, 2, and 3, respectively (**cal_wlen**). However, all values are stored in 16-bit quantities as long as the maximum **tdw_len** value does not exceed 16.

```
b       7                                                   /*  cal_sets    */
m       7   1                                               /*  cal_names   */
t       Scanline Number                                     /*  00000       */
t       Scan Offset                                         /*  00001       */
t       Gain Code                                           /*  00002       */
t       Gain Format (Log/Linear)                            /*  00003       */
t       Gain Sub Mode                                       /*  00004       */
t       Photomultipier Calibration                          /*  00005       */
t       T-Channel Gain                                      /*  00006       */
m       7   7                                               /*  cal_use     */
s       0       0       0       0       0       0       0   /*  0000-0006   */
m       7   7                                               /*  cal_wlen    */
b       16      16      16      1       2       3       16  /*  0000-0006   */
m       7   7                                               /*  cal_target  */
b       1       1       0       0       0       0       0   /*  0000-0006   */
```

The same example for a token-tagged VIDF file is as follows:

```
int     n_cal_sets = 7;                              /*  cal_sets      */
struct  CalSet0 {
        string name = “Scanline Number”;             /*  name          */
        int use = 0;                                 /*  use           */
        int word_len = 16;                           /*  word length   */
        int target = 1;                              /*  target        */
};
struct  CalSet1 {
        string name = “Scan Offset”;                 /*  name          */
        int use = 0;                                 /*  use           */
        int word_len = 16;                           /*  word length   */
        int target = 1;                              /*  target        */
};
struct  CalSet2 {
        string name = “Gain Code”;                   /*  name          */
        int use = 0;                                 /*  use           */
        int word_len = 16;                           /*  word length   */
        int target = 0;                              /*  target        */
};
struct  CalSet3 {
        string name = “Gain Format (Log/Linear)”;    /*  name          */
        int use = 0;                                 /*  use           */
        int word_len = 1;                            /*  word length   */
        int target = 0;                              /*  target        */
};
struct  CalSet4 {
        string name = “Gain Sub Mode”;               /*  name          */
        int use = 0;                                 /*  use           */
        int word_len = 2;                            /*  word length   */
        int target = 0;                              /*  target        */
};
struct  CalSet5 {
        string name = “Photomultipier Calibration”;  /*  name          */
        int use = 0;                                 /*  use           */
        int word_len = 3;                            /*  word length   */
        int target = 0;                              /*  target        */
};
struct  CalSet6 {
        string name = “T-Channel Gain”;              /*  name          */
        int use = 0;                                 /*  use           */
        int word_len = 16;                           /*  word length   */
        int target = 0;                              /*  target        */
};
```

Note that for a token-tagged VIDF file, information that is defined per calibration set is grouped into a CalSet block structure. There are 4 elements within the CalSet block structure. If no calibration sets are defined (**n_cal_sets** = 0), there are no CalSet structures contained within the token-tagged VIDF file.

4.9 INSTRUMENT STATUS (MODE) Information

The descriptions of the status or mode bytes which occur in the IDFS header records are contained within three fields of the VIDF. The three fields are described below.

4.9.1 status

The number of status or mode bytes contained in the header record field **mode_index**. Up to 255 modes can be defined for a given virtual instrument. This value should be identical to the header record field **i_mode**.

4.9.2 status_name

An array of **status** text fields each a maximum of 79 characters in length. Each field is a description of one of the defined status bytes for this virtual instrument. The order of the definitions in the array is assumed to correspond to the order in which the status bytes are written in the IDFS header file **mode_index** array.

4.9.3 states

An array of **status** elements each of which gives the number of states for its corresponding status. These numbers define the actual values of the modes (status bytes) as written to the Header File (**mode_index[i_mode]**) to be from 0 to **states[i_mode]**-1.

4.9.4 Example INSTRUMENT STATUS (MODE) Information Entries

Shown below are two examples of the VIDF status information entries. The three fields are disjointed within the VIDF with only the final two fields (**status_names** and **states**) being contiguous. The non-contiguous entries are separated with lines.

The first example shows no status data defined in this fixed-formatted VIDF file.

```
b     0                                   /*     status          */
n                                         /*     status_name     */
n                                         /*     states          */
```

The same example for a token-tagged VIDF file is as follows:

```
int     n_status = 0;                     /*     status          */
```

The second fixed-formatted VIDF example shows an IDFS which has two statuses (or modes) defined. The first status ("Current Gain Range") has 4 states (0-3), and the second status ("Voltage Gain Range") has 2 states (0-1). States can further be defined in ASCII tables if so

desired (e.g., the "Voltage Gain Range" could have ASCII table entries defining state 0 to be "Low" and state 1 to be "High").

```
b       2                                   /*  Status          */
m       2   1                               /*  status_names    */
t       Current Gain Range                  /*  00000           */
t       Voltage Gain Range                  /*  00001           */
m       2   2                               /*  States          */
s       4           2                       /*  00000-00001     */
```

The same example for a token-tagged VIDF file is as follows:

```
int     n_status = 2;                       /*  status          */
struct  Status0 {
        string name = "Current Gain Range";  /*  name           */
        int state = 4;                       /*  state          */
};
struct  Status1 {
        string name = "Voltage Gain Range";  /*  name           */
        int state = 2;                       /*  state          */
};
```

Note that for a token-tagged VIDF file, information that is defined per status is grouped into a Status block structure. There are 2 elements within the Status block structure. If no status bytes are defined (**n_status** = 0), there are no Status structures contained within the token-tagged VIDF file.

4.10 PITCH ANGLE Information

The description of how to form pitch angles for each sensor defined is contained within eleven fields of the VIDF. The eleven fields are disjointed within the VIDF with the first field separated from the last ten contiguous fields.

The pitch angle is determined by the dot product of the outward directed unit normal to the detector aperture with the local magnetic field as:

$$\alpha = \cos^{-1}\left(\frac{-N \cdot B}{|N||B|}\right)$$

where α is the pitch angle and N is the unit normal. The magnetic field, B, is assumed to be given in the same coordinate system as the unit normal.

The eleven pitch angle fields are described below.

4.10.1 pa_defined

This field indicates whether pitch angle information has been placed within the VIDF. If not, then the next ten fields are not used; otherwise, the next ten fields are considered active and it is assumed that the unit normals to the detector apertures have been defined in the CONSTANTS section of the VIDF, identified as having **const_id** values of 6, 7, and 8.

4.10.2 pa_format

This integer field is used to indicate which algorithm is to be used in computing the pitch angle. Currently, the only "algorithm" defined is the one shown above and therefore, the value for this field should be set to 1. However, if the value of this field is set to 0, the pitch angle computation is overridden and the pitch angle values that are returned by the IDFS data access software are the values that are defined in the CONSTANTS section of the VIDF, identified as having a **const_id** value of 11.

4.10.3 pa_project

This is the ASCII project acronym under which the magnetic field elements to be used in the pitch angle computation are found. It must match the magnetic field IDFS project acronym and is limited to 20 characters in length. If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.4 pa_mission

This is the ASCII mission acronym under which the magnetic field elements to be used in the pitch angle computation are found. It must match the magnetic field IDFS mission acronym and is limited to 20 characters in length. If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.5 pa_exper

This is the ASCII experiment acronym under which the magnetic field elements to be used in the pitch angle computation are found. It must match the magnetic field IDFS experiment acronym and is limited to 20 characters in length. If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.6 pa_inst

This is the ASCII instrument acronym under which the magnetic field elements to be used in the pitch angle computation are found. It must match the magnetic field IDFS instrument acronym and is limited to 20 characters in length. If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.7 pa_vinst

This is the ASCII virtual instrument acronym in which the magnetic field elements to be used in the pitch angle computation are found. It must match the magnetic field IDFS virtual instrument acronym and is limited to 20 characters in length. If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.8 pa_b1b2b3

An array of length three giving the sensor numbers of the B1, B2, and B3 components of the magnetic field as they are defined in the magnetic field IDFS. The notation B1, B2 and B3 refer to Axis A, Axis B and Axis C components, respectively (refer to the diagram provided in section 1.7). If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.9 pa_apps

The number of tables which must be applied to the magnetic field components to bring them into the appropriate units necessary for the pitch angle computation. If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.10 pa_tbls

An array of **pa_apps** which gives the table numbers which will need to be applied to the magnetic field components. Note that these table numbers are obtained from the VIDF pertaining to the virtual instrument in which the magnetic field components are defined. If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active

4.10.11 pa_ops

An array of **pa_apps** which gives the table operations which will need to be applied to each of the tables specified in **pa_tbls** (see Appendix A of the PIDF documentation *http://www.idfs.org/Editors/pidfdoc.html* for valid table operation entries). If the **pa_format** value indicates that the pitch angle computation is to be overridden with pre-defined constants, this field should be set to indicate that it is non-active.

4.10.12 Example PITCH ANGLE Information Entries

Shown below are three examples of the VIDF pitch angle information entries. The entries that are not contiguous within the VIDF are separated with lines.

The first example shows VIDF entries when no pitch angle computation is defined for a fixed-formatted VIDF file.

b	0	/*	pa_defined	*/
n		/*	pa_format	*/
n		/*	pa_project	*/

```
n                                   /*  pa_mission      */
n                                   /*  pa_exper        */
n                                   /*  pa_inst         */
n                                   /*  pa_vinst        */
n                                   /*  pa_b1b2b3       */
n                                   /*  pa_apps         */
n                                   /*  pa_tbls         */
n                                   /*  pa_ops          */
```

For a token-tagged VIDF file, pitch angle computations are specified by the presence of a PitchAngle block structure. If this structure is not present, no pitch angle computations are defined for the IDFS in question.

The second example shows pitch angle information defined, making use of the dot product algorithm. The pitch angle computations are based on the magnetic field data contained in the TSS / TSS-1 / TEMAG / TEMAG / TMMO IDFS. The TMMO IDFS has the B1, B2, and B3 values stored in sensors 0, 1, and 2, respectively. Only one table is to be applied to convert the stored B1, B2, and B3 values to the same coordinate system as the unit normal.

```
b    1                                  /*  pa_defined      */
s    1                                  /*  pa_format       */
T    TSS                                /*  pa_project      */
T    TSS-1                              /*  pa_mission      */
T    TEMAG                              /*  pa_exper        */
T    TEMAG                              /*  pa_inst         */
T    TMMO                               /*  pa_vinst        */
m    3  3                               /*  pa_b1b2b3       */
s    0          1          2            /*  0000-0002       */
s    1                                  /*  pa_apps         */
m    1  1                               /*  pa_tbls         */
s    0                                  /*  00000           */
m    1  1                               /*  pa_ops          */
s    0                                  /*  00000           */
```

The same example for a token-tagged VIDF file is as follows:

```
struct   PitchAngle {
         int format = 1;                /*  pa_format       */
         string project = "TSS";        /*  pa_project      */
         string mission = "TSS-1";      /*  pa_mission      */
         string experiment = "TEMAG";   /*  pa_exper        */
         string instrument = "TEMAG";   /*  pa_inst         */
         string vinstrument = "TMMO";   /*  pa_vinst        */
         int b1 = 0;                    /*  b1              */
         int b2 = 1;                    /*  b2              */
         int b3 = 2;                    /*  b3              */
```

```
        int num_tbls = 1;                       /*  num_tbls        */
        int tbls = 0;                           /*  tbl  0          */
        int opers = 0;                          /*  oper 0          */
};
```

The last example shows VIDF entries when pitch angle values are to be returned, but the values are defined as constants for a fixed-formatted VIDF file.

```
b   1                                           /*  pa_defined      */
s   0                                           /*  pa_format       */
n                                               /*  pa_project      */
n                                               /*  pa_mission      */
n                                               /*  pa_exper        */
n                                               /*  pa_inst         */
n                                               /*  pa_vinst        */
n                                               /*  pa_b1b2b3       */
n                                               /*  pa_apps         */
n                                               /*  pa_tbls         */
n                                               /*  pa_ops          */
```

The same example for a token-tagged VIDF file is as follows:

```
struct  PitchAngle {
        int format = 0;                         /*  pa_format       */
};
```

Note that while the PitchAngle block structure is defined, it does not contain the information that is pertinent for dot product pitch angle computations. The user is reminded that when this format is selected, the pitch angle constants (**const_id** = 11) must also be defined in the VIDF (refer to section 4.15).

4.11 DATA RECORD Information

The VIDF contains two fields which pass information concerning the IDFS data record to the IDFS data access software. These fields are described below.

4.11.1 max_nss

This field sets the maximum dimension of the **hdr_off** array in any given data record, and thus, must be greater than 0. In some cases, **max_nss** defines the maximum number of sensor sets which can be contained within a given data record. Each sensor set can have a unique offset into the IDFS header file and this offset is specified in the **hdr_off** array in the data record. However, in some cases, all sensor sets defined within an IDFS data record may utilize the same header record. If this is always true, the creator of the IDFS data set may choose to set the value of **nss** in the data records to indicate this scenario and should set the value of **max_nss** to 1 since

only one header offset value will be written in the **hdr_off** array. For more information, see the definition of **nss** in the IDFS data record section.

4.11.2 data_len

This field gives the size (in bytes) of the IDFS data records for this virtual instrument. IDFS data records are fixed length, so all data records for this virtual instrument must be **data_len** bytes.

4.11.3 Example DATA RECORD Information Entry

Shown below is an example of the fixed-formatted VIDF data information field entries for a virtual instrument which has a maximum of 12 header offsets within a data record and a data record length of 1268 bytes.

```
s       12                              /*  max_nss   */
l       1268                            /*  data_len  */
```

The same example for a token-tagged VIDF file is as follows:

```
int   max_nss = 12;                     /*  max_nss       */
int   data_len = 1268;                  /*  data_len      */
```

4.12 FILL Information

A single fill data value for the IDFS data can be defined within the VIDF through the fields **fill_flg** and **fill**. The two fields are described below.

4.12.1 fill_flg

Indicates whether or not a fill value has been defined. A value of 0 indicates that no fill value is defined while a value of 1 indicates that the **fill** field has a valid fill value.

4.12.2 fill

When indicated by **fill_flg**, this field contains the designated fill data value used in the IDFS data records to indicate missing or fill data.

4.12.3 Example FILL Information Entries

Shown below are two examples of the VIDF fill information entries.

This example shows the fixed-formatted VIDF entries when a fill value is present.

```
b   1                                   /*  fill_flg      */
l   255                                 /*  fill          */
```

The same example for a token-tagged VIDF file is as follows:

```
int     fill_flag = 1;                  /*  fill_flg    */
int     fill = 255;                     /*  fill        */
```

This example shows the fixed-formatted VIDF entries when no fill value is given.

```
b   0                                   /*  fill_flg    */
n                                       /*  fill        */
```

The same example for a token-tagged VIDF file is as follows:

```
int   fill_flag = 0;                    /*  fill_flg    */
```

4.13 BLOCK Information

There are two major repetitive blocks of fields within the VIDF file. The first describes the included tables and the second describes the included constants. The number of each contained in the VIDF are indicated through two fields described below.

4.13.1 num_tbl

This field defines the number of tables which are contained within the VIDF.

4.13.2 num_consts

This field defines the number of constants which are contained within the VIDF.

4.13.3 Example BLOCK Information Entry

Shown below is an example of the block information fields for a fixed-formatted VIDF which has three tables and two constants included within it.

```
b     3                             /*    num_tbls    */
b     2                             /*    num_consts  */
```

The same example for a token-tagged VIDF file is as follows:

```
int   n_tbls = 3;                   /*  num_tbls      */
int   n_consts = 2;                 /*  num_consts    */
```

4.14 TABLE BLOCK

A VIDF table block consists of 15 fields which describe the table, its function, and its dependencies. Each table definition can contain a mixture of look-up tables and sets of polynomial coefficients. A different table per sensor can be defined within a given table block provided that all have a common functional dependence. In addition, each sensor may switch

between defined tables based on the state of a defined status byte. This is done through the use of the critical value fields.

Most of the table block fields simply give direction to the IDFS data access software on how to acquire the table entries for each sensor defined within the VIDF. If the VIDF entry **num_tbls** is zero, there are no TABLE BLOCK definitions within the VIDF. If it is non-zero, there are **num_tbls** TABLE BLOCK definitions within the VIDF. The block of table fields are described below.

4.14.1 tbl_sca_sz

This field determines the number of scaling parameters defined in the table block scaling array (**tbl_sca**) and how they are to be applied to the actual table values (**tbl**). The absolute value of this entry determines the number of elements in the **tbl_sca** array. There are three formats to this entry.

> 0 The field specifies the total number of scaling values in the scaling array (**tbl_sca**) of which there must be one scaling value for each table value present. In this case, the two fields **tbl_sca_sz** and **tbl_ele_sz** must be identical.

= 0 There is no scaling array present and all table values are assumed to be scaled as entered. This is used primarily when the table being defined is an ASCII look-up table.

< 0 There is only one scaling coefficient per defined sensor or per status depending on **tbl_var**. If **tbl_var** is 4 or 5, the scaling applies to the statuses (modes) defined; otherwise, the scaling applies to the defined sensors. The scaling value for each sensor or status applies to all the table elements defined for that sensor or status.

4.14.2 tbl_ele_sz

This field defines the number of elements in the table array **tbl**. For ASCII tables in a fixed-formatted VIDF, there can be a maximum of 1000 entries in **tbl**.

4.14.3 tbl_type

This field indicates the type of table being defined. There are three general table categories which are defined below.

0 - All the table entries are 4-byte integers. This is the most common table type where the scaled values are used as look-up parameters or polynomial coefficients in algorithms to convert the stored IDFS data to physical units.

1 - All the table entries are ASCII strings, each of which must be a maximum of 20 characters in length. They are stored in 21 byte fields where the last byte always being a NULL (string terminator). The ASCII strings in the VIDF table entries must be in quotes (e.g., "string") for the VIDF binary conversion program (*mk_idf*) to work properly.

2 - All the table entries are 4-byte integers as in definition **0**. The difference is that here there is one look-up table or set of polynomial coefficients per scan step. This table type is only valid for VIDF's defining vector sensors (**swp_len** > 1). This **tbl_type** value specifies to the IDFS data access software that **swp_len** look-up tables or sets of polynomial coefficients must be accessed with each corresponding to data associated with a single scan step. This is used in cases where each step in a vector (scanning) sensor requires a unique expansion or correction.

There is a restriction on the placement of tables in the VIDF based on **tbl_type**, that being, tables with a **tbl_type** value of 1 must be placed after all other tables (i.e., ASCII tables must be defined after all integer tables have been defined in the VIDF).

4.14.4 tbl_comnts

The number of comment lines within the table comment field (**tbl_desc**).

4.14.5 tbl_desc

This is the table comment field which has **tbl_comnts** number of lines of free-form text where each line is a maximum of 79 characters in length. Comment lines are generally used to give a brief description of the table contents and usage.

4.14.6 tbl_var

This field indicates the functional dependence of the table which is based on the data type and whether the data is raw or processed. The recognized variable types are given in the table below.

TBL_VAR DEFINITIONS	
TBL_VAR	DEFINITION
-N	Table is a function of raw calibration set N - 1
0	Table is a function of raw sensor data
1	Table is a function of current processed data
2	Table is a function of raw scan step data
3	Table is a function of spacecraft potential data
4	Table is a function of raw mode data
5	Table is a function of processed mode data
7	Table is a function of background data

The table can be a function of processed data for sensor or status (mode) data types only. For processed data dependencies, the table must be a polynomial expansion and not a look-up table. If the table is a function of processed data, then the values are obtained by using the current processed data buffer values as input to the polynomial expression given in the table.

If the table is a function of raw data, the value indicates which IDFS data values (calibration, sensor, scan, or mode) are used either in the polynomial expansion of the table or as offsets into the look-up table. The **tbl_fmt** VIDF entry determines whether the table is a set of polynomial coefficients or look-up values.

If the table is a function of spacecraft potential or background data, the same rules as processed data must be observed, as described above, since the spacecraft potential / background data is returned by the IDFS data access software as a floating point value.

There is a restriction on the placement of tables in the VIDF based on **tbl_var**, that being, that tables with **tbl_var** values of 4 or 5 (mode-dependent tables) must be placed after all other integer tables with a **tbl_type** value of 0 or 2. Mode-dependent tables may be placed either before or after ASCII tables.

4.14.7 tbl_expand

This field specifies to the IDFS data access software whether or not polynomial coefficients should be expanded into look-up table format. The field definitions are shown below.

TBL_EXPAND DEFINITIONS	
TBL_EXPAND	DEFINITION
0	Keep as polynomial coefficients and apply as such
1	Expand to look-up table format

This field is ignored if the sensor table entries for a given sensor are already in look-up format (**tbl_fmt** = 0) or if the **tbl_var** indicates that the table is a function of processed data in which case it cannot be used to index into a look-up table.

The creation of a look-up table is done under the assumption that it is much quicker to use a precalculated value in an expression than to have to calculate the value each time it is used. The decision of whether to expand a polynomial or not is based on many considerations. Three of the more important are given here:

1. The polynomial must be a function of raw data. If not, it cannot be built into a look-up table since there would be no way to index into it.

2. The size of the table which must be built is 2^B where B is the bit length of the data of which the polynomial is a function. If B is much larger than 10-12, the number of elements needed to create the look-up table becomes large enough that it may not be advantageous to compute so many values based both on time spent in creating the table and the memory it will occupy.

3. If in most applications there will be less conversions than there are positions in the look-up table, then it is faster to leave the polynomial unexpanded and do the expansion on demand. More explicitly, an 8-bit data value which is returned once per 16 seconds will require over an hour of data to be processed before 256 conversions will have to be made. If in general this data is analyzed in shorter periods of time, it is more advantageous to leave the polynomial unexpanded than to expand it.

4.14.8 crit_act_sz

This field gives the number of elements which are found in the critical action array (**crit_action**) in the VIDF. If there is not a critical action block associated with the table, then this value is set to 0 and the next three fields are ignored. The critical action fields allow switching between tables defined in the VIDF based on current values of selected IDFS header record status (mode) bytes. The size of the **crit_action** field is dependent upon the number of states defined for the status byte flagged as the critical status byte and upon whether or not the sensors utilize the same indexes into the **tbl** field. If each sensor defines unique **tbl** offset values, then the maximum size of the **crit_action** field would be equal to the number of sensors times the number of states defined for the status byte in question.

4.14.9 crit_status

This field is a set of **sen** values each of which is an offset into the status bytes array (**mode_index**) in the header record. The indicated status byte values control which table is to be acquired at any given time and applied to the data. If a particular sensor does not use a critical status byte, then its **crit_status** element is set to –1 and the IDFS data access software will use the **tbl_off** value for the sensor in question to determine which portion of the table is to be applied to the data.

4.14.10 crit_off

This is a set of **sen** values where for each sensor which has a critical status byte defined, this is the offset into the **crit_action** array. If a particular sensor has a -1 entry for its **crit_status** element, then it should also have a -1 for its **crit_off** entry.

4.14.11 crit_action

This field is a set of **crit_act_sz** offsets into the VIDF table values defined by the field **tbl**. The offsets for each sensor are pointed to by the **crit_off** values. Each entry in the **crit_action** array is an offset into the **tbl** field and points to the first element of a set of polynomial coefficients or the first element in a look-up table. For each sensor, there must be one offset defined for each possible state of the status byte flagged as the critical status byte. Which offset to use depends on the actual value of the status byte (e.g., value of 0 uses first offset, value of 1 uses second offset, etc.). Each sensor that utilizes a critical status byte can access different locations within the **tbl** field, which is accomplished by providing unique offset values in the **crit_action** field. The status bytes (modes) are defined using the **status**, **status_names**, and **states** fields in the VIDF. When the mode-dependent offsets are defined for a given sensor, the table index value defined by **tbl_off** is ignored; however, the **tbl_off** value still needs to be defined and it is suggested that the value be set to –1.

4.14.12 tbl_fmt

This field is a set of **sen** or **status** values depending on **tbl_var**. If **tbl_var** is 4 or 5, then the table is a function of the status flags (mode data) and **tbl_fmt** has **status** elements. Otherwise, the table is a function of sensor data and **tbl_fmt** has **sen** elements. Each **tbl_fmt** value defines the format of the data in the **tbl** field for one of the sensors or one of the status bytes (modes) defined in the VIDF. The recognized definitions are described below.

-1 The sensor or status byte has no defined table entries in the **tbl** field.

0 The sensor or status has a table defined for it in the form of an expanded look-up table which is assumed to have 2^{tdw_len} elements where *tdw_len* is the bit length defined for the sensor in the VIDF **tdw_len** field.

N The sensor or status has a table defined for it in the form of an (N-1) order polynomial.

For ASCII tables, the **tbl_fmt** value should be set to 0.

4.14.13 tbl_off

This field is a set of **sen** or **status** values depending on **tbl_var**. If **tbl_var** is 4 or 5, then the table is a function of the status flags (mode data) and **tbl_off** has **status** elements. Otherwise, the table is a function of sensor data and **tbl_off** has **sen** elements. Each value in **tbl_off** is an index into the **tbl** array to the beginning table entry for the sensor or status byte. Each sensor or status has a unique index value allowing each to have its own tabular values associated with it. Multiple sensors or status bytes can index into the same value indicating that one table applies to multiple sensors or statuses.

If a sensor or status does not have a defined table in this table block (**tbl_fmt** = -1), then its offset (**tbl_off**) is set to -1. If the sensor or status has several tables defined through the use of the critical action fields, then the table offset value **tbl_off** should be set to –1. The IDFS data access software will dynamically switch between the tables depending on the currently defined critical status state and the table offset value defined by **crit_action**.

4.14.14 tbl_sca

This field is a set of **|tbl_sca_sz|** elements containing the scale factors which will be applied to the elements in the **tbl**. If there are no scaling factors (**tbl_sca_sz** = 0), then this field is ignored and the table values are used as read (not scaled). Each scaling factor is a power of ten which is used to convert the integer values in the **tbl** field into floating point values as:

$$VAL = tbl\_value * 10^{tbl\_sca}$$

If **tbl_sca_sz** is negative, there is one scaling factor per sensor which is applied to all of the defined **tbl** elements for that sensor. Otherwise, there is one scaling value for each element in the **tbl** field.

4.14.15 tbl

This field is a set of **tbl_ele_sz** entries. Non-ASCII tables consist of 4-byte integer values. These values are converted (where required) into floating point values through the use of the scaling factors found in the field **tbl_sca**. ASCII table entries are strings each of which are a maximum of 20 characters in length. Strings can have multiple words, and spaces count as 1 character in the character count. The ASCII strings must be in quotes (e.g., "two strings") in the VIDF table entries. The **tbl** values constitute the sum total of all the look-up tables and sets of polynomial coefficients or ASCII strings defined in this TABLE BLOCK.

4.14.16 Example TABLE BLOCK Entries

Shown below are three examples of different VIDF table block entries. The first example is a fixed-formatted VIDF table block definition for a virtual instrument which has three sensors all returning voltage measurements. The raw IDFS data is converted to voltage by means of polynomial expansion in each case and the coefficients are not expanded to full look-up tables. Each sensor uses one scale factor for all coefficients, and there are no critical dependencies. Sensors 0 and 2 use the same polynomial expansion which has 2 coefficients, and sensor 1 uses a different polynomial expansion that has 4 coefficients.

```
l   -3                                                          /*  tbl_sca_sz      */
l   6                                                           /*  tbl_ele_sz      */
b   0                                                           /*  tbl_type        */
s   3                                                           /*  tbl_comnts      */
m   3   1                                                       /*  tbl_desc        */
t   Three sets of polynomial coefficients which expand the raw  /*  000             */
t   sensor data to millivolts for sensors 0 and 2 and to        /*  001             */
t   microvolts for sensor 1.                                    /*  002             */
b   0                                                           /*  tbl_var         */
b   0                                                           /*  tbl_expand      */
l   0                                                           /*  crit_act_sz     */
n                                                               /*  crit_status     */
n                                                               /*  crit_off        */
n                                                               /*  crit_action     */
m   3   3                                                       /*  tbl_fmt         */
b   2           4           2                                   /*  000-002         */
m   3   3                                                       /*  tbl_off         */
l   0           2           0                                   /*  0000-0002       */
m   3   3                                                       /*  tbl_sca         */
b   -3          -6          -3                                  /*  000-002         */
m   6 3                                                         /*  tbl             */
l   4500        500         10000000                            /*  000-002         */
l   5000000     -10000      300                                 /*  003-005         */
```

The same example for a token-tagged VIDF file is as follows:

```
struct  TableN {
            int tbl_sca_sz = -3;                /*  tbl_sca_sz      */
            int tbl_ele_sz = 6;                 /*  tbl_ele_sz      */
            int tbl_type = 0;                   /*  tbl_type        */
/*
 *          Three sets of polynomial coefficients which expand the raw
 *          sensor data to millivolts for sensors 0 and 2 and to
 *          microvolts for sensor 1.
 */
            int tbl_var = 0;                    /*  tbl_var         */
            int tbl_expand = 0;                 /*  tbl_expand      */
```

```
        int crit_act_sz  = 0;                                  /*  crit_act_sz     */
        int format [3] = {2, 4, 2};                            /*  format          */
        int offset [3] = {0, 2, 0};                            /*  offsets         */
        int scale [3] = {-3, -6, -3};                          /*  scale factor    */
        int values [6] = {                                     /*  values          */
          4500, 500, 10000000, 5000000, -10000, 300            /*  000 – 005       */
        };
};
```

Note that for a token-tagged VIDF file, information that is defined per table is grouped into a Table block structure. In the example shown above, the structure is labeled TableN, where N would be replaced by the table number, starting with 0. If no tables are defined (**n_tbls** = 0), there are no Table structures contained within the token-tagged VIDF file.

The next fixed-formatted VIDF example is nearly the same as the first with the exception that for sensor 1, the data is converted to physical units differently depending on whether the instrument has been placed in high or low gain mode. This mode has been stored in the first status byte in the header records. The polynomial coefficients to use for the expansion of the data are determined through the use of the critical action information. When the status byte (mode) value is 0, the polynomial coefficients begin at **tbl** offset 2 (first **crit_action** value), and when the status byte is 1, the polynomial coefficients begin at **tbl** offset 6 (second **crit_action** value). There is also now one scaling factor per table value.

```
l    10                                                        /*  tbl_sca_sz      */
l    10                                                        /*  tbl_ele_sz      */
b    0                                                         /*  tbl_type        */
s    5                                                         /*  tbl_comnts      */
m    5   1                                                     /*  tbl_desc        */
t    Four sets of polynomial coefficients which expand the raw /*  000             */
t    sensor data to millivolts for sensors 0 and 2 and to      /*  001             */
t    microvolts for sensor 1. There are 2 sets of coefficients /*  002             */
t    for sensor 1 depending on the gain. Switching between the /*  003             */
t    coefficients is handled by the critical action info       /*  004             */
b    0                                                         /*  tbl_var         */
b    0                                                         /*  tbl_expand      */
l    2                                                         /*  crit_act_sz     */
m    3   3                                                     /*  crit_status     */
b    -1          0           -1                                /*  000-002         */
m    3   3                                                     /*  crit_off        */
s    -1          0           -1                                /*  000-002         */
m    2   2                                                     /*  crit_action     */
l    2           6                                             /*  000-001         */
m    3   3                                                     /*  tbl_fmt         */
b    2           4           2                                 /*  000-002         */
m    3   3                                                     /*  tbl_off         */
l    0           2           0                                 /*  000-002         */
```

```
m     10   5                                                    /*   tbl_sca      */
b     -3         -3          -5          -5          -5        /*   000-004      */
b     -5         -6          -6          -6          -6        /*   005-009      */
m     10   5                                                    /*   tbl          */
l     4500       500         10000000    5000000     -10000    /*   000-004      */
l     300        10000000    5000000     -10000      300       /*   005-009      */
```

The same example for a token-tagged VIDF file is as follows:

```
struct    TableN {
          int tbl_sca_sz = 10;                                  /*   tbl_sca_sz   */
          int tbl_ele_sz = 10;                                  /*   tbl_ele_sz   */
          int tbl_type = 0;                                     /*   tbl_type     */
/*
 *        Four sets of polynomial coefficients which expand the raw
 *        sensor data to millivolts for sensors 0 and 2 and to
 *        microvolts for sensor 1. There are 2 sets of coefficients
 *        for sensor 1 depending on the gain. Switching between the
 *        coefficients is handled by the critical action info
 */
          int tbl_var = 0;                                      /*   tbl_var      */
          int tbl_expand = 0;                                   /*   tbl_expand   */
          int crit_act_sz  = 2;                                 /*   crit_act_sz  */
          struct CriticalAction {
              int status [3] = {-1, 0, -1};                     /*   status       */
              int offset [3] = {-1, 0, -1};                     /*   offset       */
              int table [2] = {                                 /*   table        */
            2, 6                                                /*   000 – 001    */
              };
          };
          int format [3] = {2, 4, 2};                           /*   format       */
          int offset [3] = {0, 2, 0};                           /*   offsets      */
          int scale [10] = {                                    /*   scale_factor */
            -3, -3, -5, -5, -5, -5                              /*   000 – 005    */
            -6, -6, -6, -6};                                    /*   006 – 009    */
          int values [10] = {                                   /*   values       */
            4500, 500, 10000000, 5000000, -10000, 300           /*   000 – 005    */
            10000000, 5000000, -10000, 300                      /*   006 – 009    */
          };
};
```

The last fixed-formatted VIDF example shows the construction of an ASCII table. The table is a function of raw status bytes (modes). There are two status bytes each of which have two discrete states. The ASCII table defines the states.

```
l     0                                                       /*   tbl_sca_sz   */
l     4                                                       /*   tbl_ele_sz   */
```

```
b     1                                                          /*  tbl_type     */
s     1                                                          /*  tbl_comnts   */
m     1   1                                                      /*  tbl_desc     */
t     ASCII definitions of status states                         /*  000          */
b     4                                                          /*  tbl_var      */
b     0                                                          /*  tbl_expand   */
l     0                                                          /*  crit_act_sz  */
n                                                                /*  crit_status  */
n                                                                /*  crit_off     */
n                                                                /*  crit_action  */
m     2   2                                                      /*  tbl_fmt      */
b     0             0                                          /*  000-002      */
m     2   2                                                      /*  tbl_off      */
l     0             2                                          /*  0000-0002    */
n                                                                /*  tbl_sca      */
m     4   4                                                      /*  tbl          */
T     “Hi”          “Low”       “Off”       “Standby”          /*  000-003      */
```

The same example for a token-tagged VIDF file is as follows:

```
struct    TableN {
          int tbl_sca_sz = 0;                                    /*  tbl_sca_sz   */
          int tbl_ele_sz = 4;                                    /*  tbl_ele_sz   */
          int tbl_type = 1;                                      /*  tbl_type     */
/*
 *        ASCII definitions of status states
 */
          int tbl_var = 4;                                       /*  tbl_var      */
          int tbl_expand = 0;                                    /*  tbl_expand   */
          int crit_act_sz  = 0;                                  /*  crit_act_sz  */
          int format [2] = {0, 0};                               /*  format       */
          int offset [2] = {0, 2};                               /*  offsets      */
          string values [4] = {                                  /*  values       */
            “Hi”, “Low”, “Off”, “Standby”                        /*  000 – 003    */
          };
};
```

4.15 CONSTANT BLOCK

A VIDF constant block consists of 5 fields which describe the constant entries associated with it. Unlike table definitions each of which can hold several tables of various formats per IDFS sensor, each constant definition holds only one value per IDFS sensor and all values must represent the same quantity (**const_id** is the same for all sensors).

There are **num_consts** CONSTANT BLOCK definitions where each constant definition has a different **const_id**. If **num_consts** is 0, then there are no defined constants for this IDFS.

There are no required constant definitions within the IDFS paradigm. If a constant which is needed for a computation in the IDFS data access software is not found, either the computation will be skipped or default values will be used.

One specific example where constants are used is in the generation of pitch angle values. The dot product computation for pitch angle requires the existence of the 3 constants defining the sensor normal vector to the instrument aperture (**const_id** = 6,7,8) given with respect to the magnetometer used for computing pitch angles. If these are not present, then the pitch angles are simply not calculated. For the case where the VIDF indicates that the pitch angles are defined as constants within the VIDF file (**const_id** = 11), the pitch angle values are simply not returned if the constants are not present.

4.15.1 const_id

This field contains a value which identifies the defined constants as being within one of several predefined categories. The field identifies the constant to the IDFS data access software which may require it in certain calculations. The defined categories are shown in the table below. The generic category is used for any constant which fits none of the other categories.

CONST_ID DEFINITIONS	
CONST_ID	DEFINITION
0	Generic
1	elevation angle
2	azimuthal angle offsets
3	azimuthal field of view (FWHM)
4	initial aperture elevation angle
5	final aperture elevation angle
6	Axis A component of aperture normal vector
7	Axis B component of aperture normal vector
8	Axis C component of aperture normal vector
9	initial azimuthal angle
10	final azimuthal angle
11	pitch angle
12	euler angle
13	euler angle rotation axis
14	start of spin azimuthal angle offset
15	declination angle
16	right ascension angle
17	background

The elevation angle constant (**const_id** = 1) assumes that the elevation angle runs from 0-180 degrees with 0 degrees being parallel to Axis C in a right-handed 3-D coordinate system (refer to diagram in section 1.7). The azimuthal angle offsets (**const_id** = 2), when present, are used by the IDFS data access software to compute offsets in the returned spin angle for each sensor. The initial and final aperture elevation angle values (**const_id** = 4,5) must be defined between the range of 0 to 180 degrees. In addition, the values must not be the same value; in

other words, a theta range, not a point, must be defined for each sensor, with the initial elevation angle value being less than the final elevation angle value for each sensor. The initial and final azimuthal angle constants (**const_id** = 9,10) are used in conjunction with the token-tagged VIDF extensible field **phi_method.** When the **phi_method** field value indicates that the azimuthal angles have pre-defined values, the IDFS data access software will pick up the initial and final azimuthal angle constants and return these values.

If pitch angles are defined, there are basically two options available for the specification of the pitch angle values: (1) the dot product of the outward directed unit normal to the detector aperture with the local magnetic field can be computed or (2) pre-defined pitch angle values for each sensor can be defined as a constant in the VIDF. For option 1, **const_id**'s 6,7 and 8 must be provided to define the aperture normal vector. When the **pa_format** field value indicates that the second option is specified, the IDFS data access software will pick up the pitch angle constants (**const_id** = 11) defined for each sensor and return these values. The user is referred to section 4.10 for a more in-depth explanation of pitch angle definitions.

If euler angle information has been defined within the VIDF, there are basically two options available for the specification of the euler angles: (1) the euler angles are identified as sensors values defined within a specific IDFS data source or (2) pre-defined euler angle values for each sensor can be defined as constants in the VIDF. When the **pmi_format** field value indicates that the second option is specified, routines will pick up the euler angle constants (**const_id** = 12) and the euler angle rotation axes (**const_id** = 13) defined for each sensor. There must be **num_pmi_angles** euler angle constants and rotation axes defined within the VIDF, the values must be already in units of degrees and the order in which the constants appear in the VIDF file defines the order of application.

If celestial position (declination and right ascension) angle information has been defined within the VIDF, there are basically two options available for the specification of the celestial position angles: (1) the celestial position angles are identified as sensors values defined within a specific IDFS data source or (2) pre-defined celestial position angle values for each sensor can be defined as constants in the VIDF. When the **cp_format** field value indicates that the second option is specified, routines will pick up the declination angle constants (**const_id** = 15) and the right ascension angle constants (**const_id** = 16) defined for each sensor. The values must be already in units of degrees.

If background information has been defined within the VIDF, there are basically two options available for the specification of the background values: (1) the background values are identified as sensor values defined within a specific IDFS data source or (2) pre-defined background values for each sensor can be defined as constants in the VIDF. When the **bkgd_format** field value indicates that the second option is specified, routines will pick up the background constants (**const_id** = 17) defined for each sensor. The values must already be in the units desired.

The start of spin azimuthal angle offsets (**const_id** = 14), when present, are used by the IDFS data access software in order to determine the start of spin using azimuthal angle information. The default method within the IDFS paradigm flags the start of spin as the point at

which the azimuthal angle crosses the 0 degree position (this is the point at which the instrument looks towards the sun). In other words, when the instrument has rotated to point toward the sun, then the start of spin has been found. If the start of spin is to be defined as the point where the sun sensor views the sun, then the values defined for the start of spin azimuthal angle offset (**const_id** = 14) represent the azimuthal angle offset when the start of spin has been encountered. The IDFS data access software flags the point at which the azimuthal angle crosses this angle (**const_id** = 14), specified in hundredths of a degree, to mark the start of spin instead of 0 degrees. Note, the contents of **const_id** = 14 should be the same as **const_id** = 2 if start of spin occurs when the sun sensor points toward the sun.

Since the azimuthal angle values that are computed and returned for each sweep may not return the **exact** angle specified as the marker for the start of a spin, the IDFS data access software takes into account a tolerance factor. The tolerance factor is based on the instrument timing and is computed as follows:

(360.0 [deg/rev] / **spin** [rev/msec]) * (**swp_time** [msec] **/ n_sample**)

where **spin** is the azimuthal rate of rotation for the virtual instrument, **swp_time** is the time duration of the sweep, and **n_sample** is the number of data samples returned. This tolerance factor is added to the angle value that marks the start of a spin and this angular range is what the IDFS data access software is looking for when determining when a new spin has begun. For more information on start of spin, refer to section 5.9.

4.15.2 const_comnts

The number of comment lines within the constant description field.

4.15.3 const_desc

A set of free-form text, each line being a maximum of 79 characters in length. The number of lines of text in the comment field is defined in the **const_comnts** entry. Comment lines are generally used to give a brief description of the constant contents and usage.

4.15.4 const_sca

This field is a set of **sen** elements containing the scale factors which will be applied to the elements in the **const** array. Each scaling factor is a power of ten which is used to convert the integer values in the **const** field into floating point values as:

$$VAL = const\_value * 10^{const\_sca}$$

4.15.5 const

This field is a set of **sen** entries stored in 4-byte integers. These values are converted into floating point values through the use of the scaling factors found in the field **const_sca**.

4.15.6 Example CONSTANT BLOCK Entry

Shown below is an example of a VIDF constant block entry for a fixed-formatted VIDF file. This example shows the x-component of the detector aperture normal for a virtual instrument which has five sensors.

```
b    6                                                    /*  const_id        */
s    1                                                    /*  const_comnts    */
m    1   1                                                /*  const_desc      */
t    The x-component of the unit normal aperture vectors  /*  00001           */
m    5   5                                                /*  const_sca       */
b    -5          -5          -5          -5          -5   /*  0000-0004       */
m    5   5                                                /*  const           */
l    91234       87621       77472       45200       27481 /*  0000-0004       */
```

The same example for a token-tagged VIDF file is as follows:

```
struct    ConstantN {
          int id = 6;                                          /*  const_id        */
/*
 *        The x-component of the unit normal aperture vectors
 */
          int scale [5] = {-5, -5, -5, -5, -5};                /*  scale_factor    */
          int values [5] = {91234, 87621, 77472, 45200, 27481}; /*  values          */
};
```

Note that for a token-tagged VIDF file, information that is defined per constant is grouped into a Constant block structure. In the example shown above, the structure is labeled ConstantN, where N would be replaced by the constant number, starting with 0. If no constants are defined (**n_consts** = 0), there are no Constant structures contained within the token-tagged VIDF file.

5. Fields Pertinent only to Token-Tagged VIDFs

As more and more data sets were being converted into the IDFS storage format, the need to expand the current set of fields defined within the VIDF file finally became a reality. In order to preserve backwards compatibility with data sets already in IDFS format, software was modified to enable the parsing of a token-tagged, field-extensible VIDF file, while maintaining the ability to parse the old, fixed-formatted binary VIDF files. The token-tagged VIDF file format allows for the additional definition of new fields within the ASCII VIDF file. However, keep in mind that although new fields can be defined and stored in the token-tagged VIDF files, these fields are not automatically utilized by an end-user application or the IDFS data access software. Code changes must be made to the software which parses the VIDF file and returns information contained in the VIDF file in order to look for these newly defined fields.

The following fields have been added to the recognized list of VIDF fields. If there is a need to utilize any of the newly defined fields, the ASCII VIDF file must be created in the token-tagged format. If a fixed-formatted VIDF file already exists, one can utilize the converter program **vidftov3** to translate the fixed-formatted VIDF file into a token-tagged VIDF file. The original fixed-formatted VIDF file is preserved since the token-tagged VIDF files are renamed with a ".v3" extension added to the end of the filename.

5.1 AZIMUTHAL COMPUTATION Information

The IDFS data access software automatically computes and returns initial and final azimuthal angles for each data value returned for a given IDFS sensor. If the spin rate indicates a parked or otherwise non-rotating status, the **sun_sen** field in the data record holds the angle. If the experiment is spinning, the computation of these angle values is performed. The single VIDF field **phi_method** defines how the angles are computed.

5.1.1 phi_method

This field defined how the azimuthal angle values are to be computed by the IDFS data access software. If the value for this field is set to 1, the initial and final azimuthal angle constants (**const_id** = 9,10) are retrieved from the VIDF file and these values are returned. If the value for this field is set to 0, the IDFS data access software will compute the angles utilizing the spin information contained in the data record and timing information pertinent to the IDFS data source (refer to section 10.2). For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as 0 by the IDFS data access software which parses the VIDF files.

5.1.2 Example phi_method Entry

The example shown below defines the azimuthal computation scheme as one in which VIDF constants are defined.

```
int     phi_method = 1;                         /*      phi_method – use consts      */
```

5.1.3 spin_time_offset

There may be instances where the sensors are mounted in such a way that the sensors cross the azimuthal zero degree position at different times. This azimuthal zero degree position is referred to as the **sun_sen** value and is located as a single value within the IDFS data record (refer to section 10.2.2). Since the computation for the azimuthal angles for each sensor are computed using this information, there may be the need to specify an azimuthal timing correction factor per sensor. This field is defined in terms of milliseconds. For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as 0 by the IDFS data access software which parses the VIDF files. This is also the case if the **spin_time_offset** field is not defined within the Sensor block within the token-tagged VIDF file.

5.1.4 Example spin_time_offset Entry

The example shown below defines a 10-millisecond azimuthal timing correction factor for sensor zero.

```
struct      Sensor0 {
            string name = "Potential Calculation Method";
            int d_type = 0;
            int status = 1;
            int tdw_len = 4;
            int time_offset = 0;
            int spin_time_offset = 10;
};
```

5.2 NANOSECOND TIME ADJUSTMENT Information

The finest resolution supported by the IDFS data access software is down to the nanosecond. However, the field **dr_time** in the data record holds the relative beginning time of day in milliseconds for the first data element of the first sensor set in the data record. In order to define the time tag down to the nanosecond precision, the VIDF field **nano_defined** was created.

5.2.1 nano_defined

This field serves as a flag to indicate if a nanosecond time adjustment value is contained within the **data_array** matrix in the data record. If the value for this field is set to 0, no correction to the time tag value **dr_time** is made. In other words, millisecond resolution is adequate for the data being processed. If the value for this field is set to 1, the IDFS data access software will interpret the first four bytes contained within the **data_array** matrix in the data record as a nanosecond time adjustment factor and will use this value when computing the time tags associated with the data (refer to section 10.1). The nanosecond time adjustment factor is interpreted as a signed, 4-byte quantity and therefore, can reach a maximum of 2,147,483,647. However, since this value contains the resolution between nanosecond and millisecond precision, the value should be no larger than 999,999. For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as 0 by the software which parses the VIDF files.

5.2.2 Example NANOSECOND TIME ADJUSTMENT Information Entry

The example shown below indicates that a nanosecond time adjustment value is defined within the data record.

```
int     nano_defined = 1;               /*    nanosecond precision enabled    */
```

5.3 HEADER RECORD TIME ADJUSTMENT Information

The finest resolution supported by the IDFS data access software is down to the nanosecond. The **data_accum** field in the header record has an associated scaling factor (**time_units**) which allows the accumulation time to be expressed in terms of nanoseconds. However, the **data_lat**, **swp_reset** and **sen_reset** fields can be expressed only in microseconds. In order to allow for these three fields to be expressed in the same manner as the **data_accum** field, the following three VIDF fields have been added to the token-tagged, field-extensible VIDF file definition..

5.3.1 data_lat_units

This field is a 1-byte quantity that together with the **data_lat** field describes the dead time between successive data acquisitions. This field contains the scaling given as a power of ten, which is used to take the value in the **data_lat** field to units of seconds. Since the finest resolution supported by the IDFS data access software is down to the nanosecond, **data_lat_units** CANNOT be less than -9. The conversion is given below.

$$DATA\_LATENCY\_TIME = data\_lat * 10^{data\_lat\_units}$$

For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as -6 by the software which parses the VIDF files to represent microseconds.

5.3.2 Example data_lat_units Entry

The example shown below indicates that the **data_lat** value in the header record is expressed in terms of milliseconds.

```
int     data_lat_units = -3;            /*    data_lat in milliseconds    */
```

5.3.3 swp_reset_units

This field is a 1-byte quantity that together with the **swp_reset** field describes the dead time between successive columns of data within the data matrix when time is advancing down the data columns or between successive data rows when time is advancing across the rows. This field contains the scaling given as a power of ten, which is used to take the value in the **swp_reset** field to units of seconds. Since the finest resolution supported by the IDFS data access software is down to the nanosecond, **swp_reset_units** CANNOT be less than -9. The conversion is given below.

$$SWP\_RESET\_TIME = swp\_reset * 10^{swp\_reset\_units}$$

For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as -6 by the software which parses the VIDF files to represent microseconds.

5.3.4 Example swp_reset_units Entry

The example shown below indicates that the **swp_reset** value in the header record is expressed in terms of nanoseconds.

```
int     swp_reset_units = -9;                   /*      swp_reset in nanoseconds     */
```

5.3.5 sen_reset_units

This field is a 1-byte quantity that together with the **sen_reset** field describes the dead time between successive sensor sets of data. This field contains the scaling given as a power of ten, which is used to take the value in the **sen_reset** field to units of seconds. Since the finest resolution supported by the IDFS data access software is down to the nanosecond, **sen_reset_units** CANNOT be less than -9. The conversion is given below.

$$SEN\_RESET\_TIME = sen\_reset * 10^{sen\_reset\_units}$$

For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as -6 by the software which parses the VIDF files to represent microseconds.

5.3.6 Example sen_reset_units Entry

The example shown below indicates that the **sen_reset** value in the header record is expressed in terms of seconds.

```
int     sen_reset_units = 0;                    /*      sen_reset in seconds         */
```

5.4 TRANSFORMATION Information

Rotation angles are direction dependent. For purposes of the transform code, positive angles are measured counter-clockwise in the right handed sense. Basically, a rotation is defined about an axis. This axis remains the same within a transform. The angle of rotation describes the relation of the rotated axes relative to the initial axis. Thus, the three combinations (one for rotations about each axis) are shown in Figures 4, 5 and 6 for the positive angle theta:

$$\begin{bmatrix} 1' \text{ axis} \\ 2' \text{ axis} \\ 3' \text{ axis} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\text{Psi}) & \sin(\text{Psi}) \\ 0 & -\sin(\text{Psi}) & \cos(\text{Psi}) \end{bmatrix} \begin{bmatrix} 1 \text{ axis} \\ 2 \text{ axis} \\ 3 \text{ axis} \end{bmatrix}$$

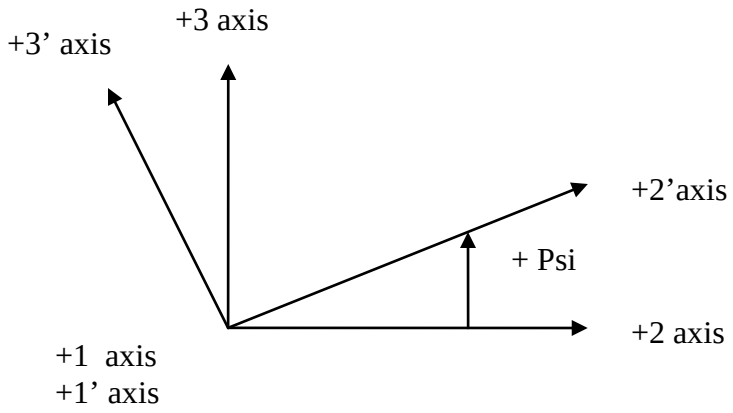

Figure 4. Rotation about +1 axis

$$\begin{bmatrix} 1' \text{ axis} \\ 2' \text{ axis} \\ 3' \text{ axis} \end{bmatrix} = \begin{bmatrix} \cos(\text{Psi}) & 0 & -\sin(\text{Psi}) \\ 0 & 1 & 0 \\ \sin(\text{Psi}) & 0 & \cos(\text{Psi}) \end{bmatrix} \begin{bmatrix} 1 \text{ axis} \\ 2 \text{ axis} \\ 3 \text{ axis} \end{bmatrix}$$

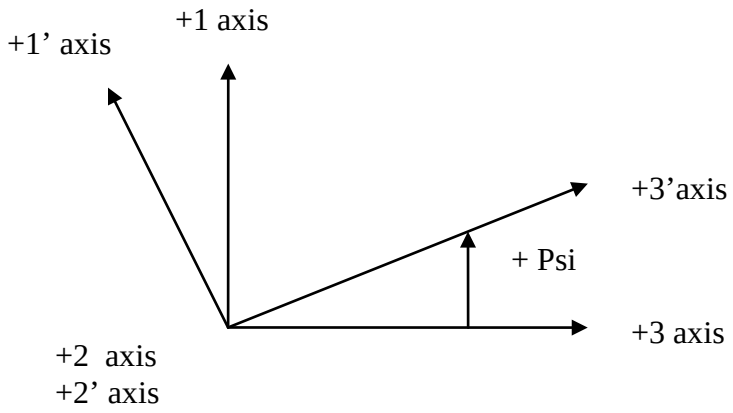

Figure 5. Rotation about +2 axis

$$\begin{pmatrix} 1' \text{ axis} \\ 2' \text{ axis} \\ 3' \text{ axis} \end{pmatrix} = \begin{pmatrix} \cos(Psi) & \sin(Psi) & 0 \\ -\sin(Psi) & \cos(Psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \text{ axis} \\ 2 \text{ axis} \\ 3 \text{ axis} \end{pmatrix}$$

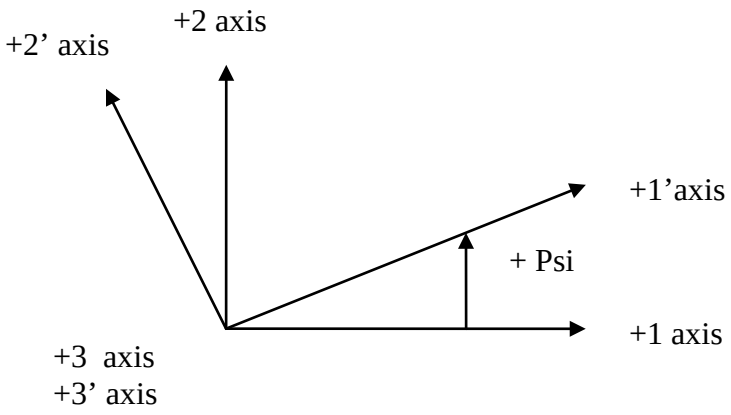

Figure 6. Rotation about +3 axis

Since you need to know from where you are starting your transform, it matters as to where your spacecraft is orbiting. Transforms from interstellar space where the spacecraft is orbiting around the sun will have a different orientation than spacecraft orbiting about a planet, say Earth. The inertial frame is different depending on the orbiting body. If you are orbiting the sun, then your inertial frame is in a heliocentric system. If the planet is Mars, then the inertial system will be relative to Mars. The VIDF field **orbiting_body** provides this information.

Another factor that affects transformations is reference sensor signal delay. When a reference sensor views the object, there is some delay between when the object is detected and when this information is signaled. The delay is normally described as a time difference. If the spacecraft is spinning, then the reference delay may be expressed as an angle. A positive angle difference is measured as the azimuthal difference between the reference sensor and the reference sensor signal (e.g. when the signal from the reference signal is delayed in time). A negative angle indicates that the signal from the reference sensor occurs before the object is detected by the reference sensor.

It is recognized that some sensors need an opposite change in a quantity to decide when the object is detected. By the time the detection is made, decision is concluded, and signals are transmitted, the sensor could no longer be viewing the object. Thus, a delay time (**ref_sen_delay**) is needed so that time and distance may be changed to reflect the appropriate position. This information is needed to align the time of the reference signal for coordinate system transforms into an inertial reference system.

5.4.1 orbiting_body

This field specifies where your spacecraft is orbiting. The values recognized by the IDFS software are defined as:

ORBITING_BODY FIELD DEFINITIONS	
VALUE	DEFINITION
0	Not Defined
1	Earth
2	Sun
3	Mars
4	Venus
5	Jupiter
6	Mercury
7	Saturn
8	Uranus
9	Neptune
10	Pluto

Another example of an orbiting body might be Mars. Note, that interstellar probes often orbit different bodies through out their lifetime. Spacecraft launched from Earth start their orientation with respect to the Earth. At some point in time, this will change to the Sun as the spacecraft heads out into interstellar space. Once the spacecraft reach their destination, then there could be another body that the spacecraft orbits. Thus, when changing the **orbiting_body** definition in the VIDF, there needs to be a VIDF boundary and change with file close-outs at the appropriate time.

5.4.2 Example orbiting_body Entry

The example shown below indicates that the **orbiting_body** value is Earth.

```
int     orbiting_body = 1;                  /*     orbits around Earth          */
```

5.4.3 ref_sen_delay

This field is an adjustment factor that is defined so that the reference signal occurs exactly at the time when the reference object was observed. The field **ref_sen_delay_unit** is used to specify the units in which **ref_sen_delay** is expressed.

Positive values of **ref_sen_delay** make physical sense and represent a delay in the signal. Negative values are used to adjust for something unique with the data. For example, when processing the data, the operator selects the sensor closest to when the reference signal occurs and this sensor may generate its data just before the reference sensor signal.

5.4.4 ref_sen_delay_unit

This field defines the unit for the value of **ref_sen_delay**. The values recognized by the IDFS software are defined as:

REF_SEN_DELAY_UNIT FIELD DEFINITIONS	
VALUE	DEFINITION
1	Seconds
2	Degrees
3	Radians

5.4.5 Example ref_sen_delay and ref_sen_delay_unit Entries

The time delay of the reference sensor is normally expressed in terms of time, referred to in the equations below as *time_ref_delay*. However, the delay value specified in the VIDF may be expressed in terms of time or angle. In the first example shown below, a delay value of 1.28 microseconds would be defined using the following 2 VIDF field entries:

```
float   ref_sen_delay = 1.28e-6;        /*   1.28 microseconds      */
int     ref_sen_delay_unit = 1;         /*   unit is seconds        */
```

therefore,

time_ref_delay [sec] = ref_sen_delay [sec]

where [sec] specifies that the unit of time measurement is seconds.

For a spinning spacecraft, the reference sensor delay time may also be expressed in degrees. In the second example shown below, a delay value of 3.2 degrees would be defined using the following 2 VIDF field entries:

```
float   ref_sen_delay = 3.2;            /*   3.2 degrees            */
int     ref_sen_delay_unit = 2;         /*   unit is degrees        */
```

therefore,

time_ref_delay [sec] = ref_sen_delay [deg] / (360 [deg/rev] * spin_rate [rev/sec])

For a spinning spacecraft, the reference sensor delay time may also be expressed in radians. In the third example shown below, a delay value of 0.1024 radians would be defined using the following 2 VIDF field entries:

```
float   ref_sen_delay = 1024.0e-4;      /*   3.2 degrees            */
int     ref_sen_delay_unit = 3;         /*   unit is radians        */
```

therefore,

time_ref_delay [sec] = ref_sen_delay [rad] / (2pi [rad/rev] * spin_rate [rev/sec])

In all three examples shown above, the exact time when the reference object was sensed can be computed using the equation:

adjusted current time [sec] = current time [sec] - time_ref_delay [sec]

5.5 COORDINATE SYSTEM TRANSFORMATION Information

To aid in the analysis of the data, it may be necessary to convert the data that is stored in the IDFS data set into a different coordinate system that is of importance to the Space Physics community. Data may need to be converted into coordinate systems that have geocentric origins such as Geocentric Equatorial Inertial (GEI), Geocentric Solar Ecliptic (GSE), Geographic (GEO), Geocentric Solar Magnetospheric (GSM), Solar Magnetic (SM), and/or Geomagnetic (MAG). Data may need to be converted into coordinate systems that have heliocentric origins such as Heliocentric Earth Ecliptic (HEE), Heliocentric Aries Ecliptic (HAE), and/or Heliocentric Earth Equatorial (HEEQ). Data may need to be converted into a coordinate system that have spacecraft origins such as Principal Moments of Inertia (PMI).

In order to facilitate this data conversion, the VIDF definition has been extended to allow for the specification of the information that is needed to transform the data into the various coordinate systems that are supported by the IDFS data access software and these fields are described below in sections 5.5.1, 5.5.2 and 5.5.3. Currently, within the IDFS paradigm, coordinate system transformation is only applicable for the moments computations since true vector data is available for conversion. For an overview and brief description of the supported coordinate systems, the user is referred to the webpage http://cluster/vector_convert.html.

5.5.1 BASIC COORDINATE SYSTEM Information

5.5.1.1 coord_system_defined

This field indicates whether coordinate system transformation information has been defined within the VIDF. For a token-tagged VIDF file, coordinate system transformation definitions are specified by the presence of a CoordinateSystem block structure**.** If this structure is not present, coordinate system transformation information is not defined for the IDFS in question and the remainder of the fields in section 5.5.1, 5.5.2, and 5.5.3 are not used; otherwise, the next field in this section and the fields defined in sections 5.5.2 and 5.5.3 are considered active.

5.5.1.2 coord_system

This field indicates which coordinate system the data is stored under in the IDFS data set. If the field is not defined within the VIDF file, the IDFS data access software defaults the coordinate system to indicate that coordinate system transformation information is not available.

COORDINATE SYSTEM DEFINITIONS	
ID STRING	DEFINITION
SPACECRAFT	Spacecraft coordinates
PMI	Principal Moments of Inertia coordinates
GEI	Geocentric Equatorial Inertial coordinates
GEO	Geographic coordinates
GSE	Geocentric Solar Ecliptic coordinates
GSM	Geocentric Solar Magnetospheric coordinates
SM	Solar Magnetic coordinates

COORDINATE SYSTEM DEFINITIONS	
ID STRING	DEFINITION
MAG	Geomagnetic coordinates
HEE	Heliocentric Earth Ecliptic coordinates
HAE	Heliocentric Aries Ecliptic coordinates
HEEQ	Heliocentric Earth Equatorial coordinates

5.5.2 EULER ANGLE ROTATION Information

With some spacecraft, the spin axis is different than the spacecraft reference axis. The offset of the spin axis from the spacecraft reference axis is given by a set of angles, referred to as Euler angles. These angles describe the position of the spin axis relative to the spacecraft reference axis and may be used to describe vectors relative to the spacecraft reference frame in the frame of the spin axis. The Euler angles are specified in degrees with directions defined in Section 5.4.

Not all spacecraft have Euler angles defined and not all spacecraft have the same Euler angles. With some spacecraft, the reference axis of the spacecraft and the frame which contains the spin axis are the same. Thus, the Euler angle rotations are not defined, as is the case for the Viking mission. When these reference frames differ, then there is the need to define Principal Moment of Inertia (PMI) reference Euler offset angles, as is the case for the Cluster mission. Figure 7 illustrates how the Euler angles are defined for the Cluster mission:

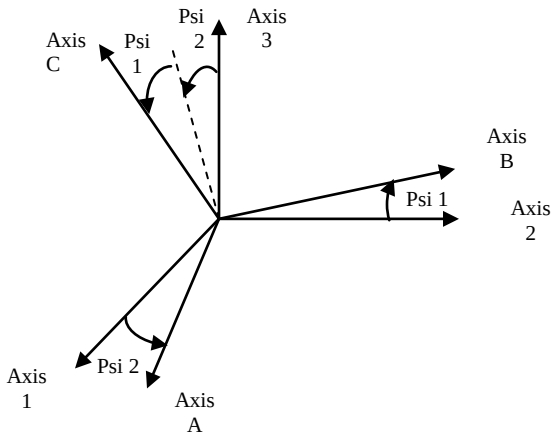

Figure 7. Euler Angle definitions for the Cluster Mission

The description of how to form Euler angles for the virtual instrument being defined is contained within thirteen fields of the VIDF. The thirteen Euler angle fields are described below.

5.5.2.1 pmi_defined

This field indicates whether Euler angle information has been defined within the VIDF. For a token-tagged VIDF file, Euler angle definitions are specified by the presence of a **PMI** block structure within a CoordinateSystem block structure. If this structure is not present, Euler angles are not defined for the IDFS in question and the next twelve fields are not used; otherwise, the next twelve fields are considered active.

5.5.2.2 pmi_format

This integer field is used to indicate which algorithm is to be used in computing the Euler angle values. Currently, the only "algorithm" defined is a set of rotations built up from the rotation definitions described in section 5.4 Therefore, the value for this field should be set to 1. However, if the value of this field is set to 0, the Euler angle rotations are overridden and the Euler angle values that are returned are the values that are defined in the CONSTANTS section of the VIDF, identified as having a **const_id** value of 12 and the rotation axes associated with the pre-defined Euler angle values are defined in the CONSTANTS section of the VIDF, identified as having a **const_id** value of 13.

5.5.2.3 num_pmi_angles

This field is used to indicate the number of Euler angles that are defined for the spacecraft in question. The value for this field should never be less than or equal to 0. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field indicates the number of constants and rotation axes to pick up from the VIDF file (all having the **const_id** value of 12 and 13, respectively). Since the pre-defined Euler angle values have the same **const_id** value, the order in which the constants appear in the VIDF file defines the order of application.

5.5.2.4 pmi_project

A satellite program which can determine the Euler angles necessary for defining the wandering of the spin axis needs to include these angles as part of the data. All of the necessary Euler angles must be defined by and within a single scalar IDFS data source. The IDFS lineage of the data file location for the Euler angles is specified utilizing the fields **pmi_project**, **pmi_mission**, **pmi_exper**, **pmi_inst** and **pmi_vinst**. The **pmi_project** field is the ASCII project acronym under which the Euler angle values are found. It is limited to 20 characters in length. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.5 pmi_mission

This field is the ASCII mission acronym under which the Euler angle values are found. It is limited to 20 characters in length. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.6 pmi_exper

This field is the ASCII experiment acronym under which the Euler angle values are found. It is limited to 20 characters in length. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.7 pmi_inst

This field is the ASCII instrument acronym under which the Euler angle values are found. It is limited to 20 characters in length. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.8 pmi_vinst

This field is the ASCII virtual instrument acronym in which the Euler angle values are found. It is limited to 20 characters in length. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.9 pmi_sensors

This field is an integer array of length **num_pmi_angles**, giving the sensor number(s) within the virtual instrument which contains the Euler angles. The order of specification should be the order of application. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.10 pmi_rotation_axis

Euler angles are defined so that they are rotations about an axis. The convention of labeling is the same as defined in section 5.4, namely 1-axis, 2-axis, and 3-axis. This field is an integer array of length **num_pmi_angles**, specifying the rotation axis for each Euler angle in the order of Euler angle application. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.11 pmi_apps

Tables may need to be applied to the Euler angles to convert the storage values into the correct orientation of degree measure. This field holds the number of tables to be applied. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.12 pmi_tbls

This field, which is an integer array of length **pmi_apps**, specifies the table numbers which will need to be applied to the Euler angles to convert the storage values into the correct orientation of degree measure. Note that these table numbers are obtained from the VIDF pertaining to the virtual instrument in which the Euler angle values are defined. If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.13 pmi_ops

This field, which is an integer array of length **pmi_apps,** specifies the table operations which will need to be applied to each of the tables specified in **pmi_tbls** (see Appendix A of the PIDF documentation *http://www.idfs.org/Editors/pidfdoc.html* for valid table operation entries). If the **pmi_format** value indicates that the Euler angle values are pre-defined constants, this field should be omitted from the PMI block structure.

5.5.2.14 Example EULER ANGLE ROTATION Information Entries

Shown below are two examples of the VIDF Euler angle information entries. The first example shows Euler angle information defined, making use of the sensor data contained in the CLUSTERII / CLUSTER-3 / AUXILIARY / OA / CSATT IDFS. The CSATT IDFS has two Euler angles. They are written in the virtual so that the angle represented by sensor 7 is applied first, then sensor 8 (pmi_sensors [2] = {7, 8}). The Euler angle represented by sensor 7 rotates about the 2-axis and the Euler angle represented by sensor 8 rotates about the 1-axis. Two tables are needed to convert the stored Euler angle values to the correct orientation of degree measure.

```
struct    CoordinateSystem {
  string  coord_system = "SPACECRAFT";          /*  coord_system        */
  struct  PMI {
          int format = 1;                       /*  pmi_format          */
          int num_angles = 2;                   /*  num_pmi_angles      */
          string project = "CLUSTERII";         /*  pmi_project         */
          string mission = "CLUSTER-3";         /*  pmi_mission         */
          string experiment = "AUXILIARY";      /*  pmi_exper           */
          string instrument = "OA";             /*  pmi_inst            */
          string vinstrument = "CSATT";         /*  pmi_vinst           */
          int sensors [2] = {7, 8};             /*  pmi_sensors         */
          int rot_axis [2] = {2, 1};            /*  pmi_rotation_axis   */
          int num_tbls = 2;                     /*  pmi_apps            */
          int tbls [2] = {0, 1};                /*  pmi_tbls            */
          int opers [2] = {0, 4};               /*  pmi_ops             */
    };
};
```

The second example shows the VIDF entries for the PMI block structure when Euler angle values are to be returned, but the values are defined as constants. The user is reminded that when this format is selected, the euler angle constants (**const_id** = 12) and the euler angle rotation axes constants (**const_id** = 13) must also be defined in the VIDF (refer to section 4.15).

```
struct    CoordinateSystem {
  string  coord_system = "SPACECRAFT";          /*  coord_system        */
  struct  PMI {
          int format = 0;                       /*  pmi_format          */
          int num_angles = 2;                   /*  num_pmi_angles      */
    };
};
```

Note that while the PMI block structure is defined, it does not contain the information that is pertinent for the extraction of the data from a specified IDFS data source. In addition, the values defined as constants in the VIDF file must already be specified in units of degrees.

5.5.3 CELESTIAL POSITION Information

Two angles are required to specify the spacecraft spin axis in the Geocentric Equatorial Inertial (GEI) coordinate system – the right ascension angle and the declination angle. The right ascension angle defines the right ascension of the spacecraft spin axis on the celestial sphere. The declination angle defines the declination of the spacecraft spin axis on the celestial sphere. These angles specify the relationship between the PMI coordinate system in the GEI coordinate system and must be expressed in the inertial system of J2000.0. Together, these two angles are referred to as celestial position angles.

The description of how to define celestial position angles for the virtual instrument being defined is contained within fifteen fields of the VIDF. The fifteen celestial position angle fields are described below.

5.5.3.1 cp_defined

This field indicates whether celestial position angle information has been defined within the VIDF. For a token-tagged VIDF file, celestial position angle definitions are specified by the presence of a CelestialPosition block structure within a CoordinateSystem block structure. If this structure is not present, celestial position angles are not defined for the IDFS in question and the next fourteen fields are not used; otherwise, the next fourteen fields are considered active.

5.5.3.2 cp_format

This integer field is used to indicate which algorithm is to be used in computing the celestial position angle values. Currently, the only "algorithm" defined is the retrieval of the values from a specified IDFS data source and the conversion of those values to physical units. Therefore, the value for this field should be set to 1. However, if the value of this field is set to 0, the celestial position angle values that are returned are the values that are defined in the CONSTANTS section of the VIDF, identified as having a **const_id** value of 15 for the declination angles and a **const_id** value of 16 for the right ascension angles.

5.5.3.3 cp_project

A satellite program which can determine the celestial position angles needs to include these angles as part of the data. All of the necessary celestial position angles must be defined by and within a single scalar IDFS data source. The IDFS lineage of the data file location for the celestial position angles is specified utilizing the fields **cp_project**, **cp_mission**, **cp_exper**, **cp_inst** and **cp_vinst**. The **cp_project** field is the ASCII project acronym under which the celestial position angle values are found. It is limited to 20 characters in length. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.4 cp_mission

This field is the ASCII mission acronym under which the celestial position angle values are found. It is limited to 20 characters in length. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.5 cp_exper

This field is the ASCII experiment acronym under which the celestial position angle values are found. It is limited to 20 characters in length. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.6 cp_inst

This field is the ASCII instrument acronym under which the celestial position angle values are found. It is limited to 20 characters in length. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.7 cp_vinst

This field is the ASCII virtual instrument acronym in which the celestial position angle values are found. It is limited to 20 characters in length. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.8 cp_declination_sensor

This field specifies the sensor number for the declination angle defined within the virtual instrument which contains the celestial position angles. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.9 cp_declination_apps

Tables may need to be applied to the celestial position angles to convert the storage values into the correct orientation of degree measure. This field holds the number of tables to be applied for the declination angle. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.10 cp_declination_tbls

This field, which is an integer array of length **cp_declination_apps**, specifies the table numbers which will need to be applied to the declination angle to convert the storage values into the correct orientation of degree measure. Note that these table numbers are obtained from the VIDF pertaining to the virtual instrument in which the celestial position angle values are defined. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.11 cp_declination_ops

This field, which is an integer array of length **cp_declination_apps,** specifies the table operations which will need to be applied to each of the tables specified in **cp_declination_tbls** (see Appendix A of the PIDF documentation *http://www.idfs.org/Editors/pidfdoc.html* for valid table operation entries). If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.12 cp_rt_ascension_sensor

This field specifies the sensor number for the right ascension angle defined within the virtual instrument which contains the celestial position angles. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.13 cp_rt_ascension_apps

Tables may need to be applied to the celestial position angles to convert the storage values into the correct orientation of degree measure. This field holds the number of tables to be applied for the right ascension angle. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.14 cp_rt_ascension_tbls

This field, which is an integer array of length **cp_rt_ascension_apps**, specifies the table numbers which will need to be applied to the right ascension angle to convert the storage values into the correct orientation of degree measure. Note that these table numbers are obtained from the VIDF pertaining to the virtual instrument in which the celestial position angle values are defined. If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.15 cp_rt_ascension_ops

This field, which is an integer array of length **cp_rt_ascension_apps,** specifies the table operations which will need to be applied to each of the tables specified in **cp_rt_ascension_tbls** (see Appendix A of the PIDF documentation *http://www.idfs.org/Editors/pidfdoc.html* for valid table operation entries). If the **cp_format** value indicates that the celestial position angle values are pre-defined constants, this field should be omitted from the CelestialPosition block structure.

5.5.3.16 Example CELESTIAL POSITION Information Entries

Shown below are two examples of the VIDF celestial position angle information entries. The first example shows celestial position angle information defined, making use of the sensor data contained in the CLUSTERII / CLUSTER-1 / AUXILIARY / OA / CSATT IDFS. Two tables are needed to convert the stored celestial position angle values to the correct orientation of degree measure. In this example, the same unit conversion is applied to both the declination and right ascension angles, although this does not have to be the case.

```
struct    CoordinateSystem {
  string  coord_system = “SPACECRAFT”;          /*  coord_system      */
  struct  CelestialPosition {
          int format = 1;                       /*  cp_format         */
          string project = "CLUSTERII";         /*  cp_project        */
          string mission = "CLUSTER-1";         /*  cp_mission        */
          string experiment = "AUXILIARY";      /*  cp_exper          */
          string instrument = "OA";             /*  cp_inst           */
          string vinstrument = "CSATT";         /*  cp_vinst          */
```

```
            int declination_sensor = 1;                 /*  cp_declination_sensor    */
            int declination_num_tbls = 2;               /*  cp_declination_apps      */
            int declination_tbls [2] = {0, 1};          /*  cp_declination_tbls      */
            int declination_opers [2] = {0, 4};         /*  cp_declination_ops       */
            int rt_ascension_sensor = 0;                /*  cp_rt_ascension_sensor   */
            int rt_ascension_num_tbls = 2;              /*  cp_rt_ascension_apps     */
            int rt_ascension_tbls [2] = {0, 1};         /*  cp_rt_ascension_tbls     */
            int rt_ascension_opers [2] = {0, 4};        /*  cp_rt_ascension_ops      */
    };
};
```

The second example shows the VIDF entries for the **CelestialPosition** block when celestial position angle values are to be returned, but the values are defined as constants. The user is reminded that when this format is selected, the declination angle constants (**const_id** = 15) and the right ascension angle constants (**const_id** = 16) must also be defined in the VIDF (refer to section 4.15).

```
  struct    CoordinateSystem {
    string  coord_system = "SPACECRAFT";        /*  coord_system             */
    struct  CelestialPosition {
            int format = 0;                     /*  cp_format                */
    };
};
```

Note that while the CelestialPosition block structure is defined, it does not contain the information that is pertinent for the extraction of the data from a specified IDFS data source. In addition, the values defined as constants in the VIDF file must already be specified in units of degrees.

5.6 SPACECRAFT POTENTIAL Information

The IDFS data access software now has the capability of returning spacecraft potential data for the IDFS data source in question. The spacecraft potential data may be used to adjust the sensor data or scan data for calculations that determine absolute quantities. The spacecraft potential data may be defined as a constant value or as a dynamic value from a separate IDFS data source. The specification of a dynamic value is similar to the VIDF definition for pitch angle information (section 4.10) and Euler angle rotation information (section 5.5.2). A dynamic spacecraft potential value must be contained within a scalar IDFS data source. The specification of the spacecraft potential data is defined using the twelve fields that are described below.

5.6.1 pot_src_defined

This field indicates whether spacecraft potential information has been defined within the VIDF. For a token-tagged VIDF file, spacecraft potential definitions are specified by the presence of a PotentialSource block structure. If this structure is not present, spacecraft potential information is not defined for the IDFS in question and the next eleven fields are not used; otherwise, the next eleven fields are considered active.

5.6.2 pot_src_format

This integer field is used to indicate whether the spacecraft potential is a constant value or a dynamic value. If the spacecraft potential is a constant value, the value for this field should be set to 0 and the value that is returned by the IDFS data access software is the value defined in the **pot_constant_val** field. If the spacecraft potential is a dynamic value coming from a separate IDFS data source, the value for this field should be set to 1.

5.6.3 pot_src_project

This is the ASCII project acronym under which the spacecraft potential sensor is to be found for a dynamic spacecraft potential. It is limited to 20 characters in length. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.4 pot_src_mission

This is the ASCII mission acronym under which the spacecraft potential sensor is to be found for a dynamic spacecraft potential. It is limited to 20 characters in length. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.5 pot_src_exper

This is the ASCII experiment acronym under which the spacecraft potential sensor is to be found for a dynamic spacecraft potential. It is limited to 20 characters in length. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.6 pot_src_inst

This is the ASCII instrument acronym under which the spacecraft potential sensor is to be found for a dynamic spacecraft potential. It is limited to 20 characters in length. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.7 pot_src_vinst

This is the ASCII virtual instrument acronym in which the spacecraft potential sensor is to be found for a dynamic spacecraft potential. It is limited to 20 characters in length. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.8 pot_src_sen

This integer field is the sensor number of the source for the spacecraft potential as defined in the designated IDFS source. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.9 pot_src_apps

This integer field defines the number of tables which must be applied to the defined sensor (the source for the spacecraft potential) to bring its value to the appropriate units (volts) necessary for computation. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.10 pot_src_tbls

An integer array of **pot_src_apps** which gives the table numbers which will need to be applied to the sensor defined as the source of the spacecraft potential. Note that these table numbers are obtained from the VIDF pertaining to the virtual instrument in which the spacecraft potential sensor is defined. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.11 pot_src_ops

An integer array of **pot_src_apps** which gives the table operations which will need to be applied to each of the tables specified in **pot_src_tbls** (see Appendix A of the PIDF documentation *http://www.idfs.org/Editors/pidfdoc.html* for valid table operation entries). Note that the spacecraft potential must have the electrical sign associated in order to generate the correct type of spectral shifting for the source. This should be handled through the use of tables if the potential located within the virtual description only contains a positive magnitude. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this field should be omitted from the PotentialSource block structure.

5.6.12 pot_constant_val

If the **pot_src_format** value indicates that the spacecraft potential is a dynamic value, this floating point field is used to specify a value that is to be returned by the IDFS data access software if there is no data available (i.e. data gaps) from the spacecraft potential data source for the time period(s) being processed. If the **pot_src_format** value indicates that the spacecraft potential is a constant value, this floating point field is used to specify the constant value that is to be returned by the IDFS data access software for the spacecraft potential. In either case, the value for this field must be defined in units of volts and must include the sign of the spacecraft charge.

5.6.13 Example SPACECRAFT POTENTIAL Information Entries

Shown below are two examples of the Spacecraft Potential information entries. The first example defines a dynamic spacecraft potential value that is retrieved from sensor 0 within the CLUSTERII / CLUSTER-1 / EFW / POTENTIAL / SCPOT IDFS data set.

```
struct    PotentialSource {
          int format = 1;                          /*   pot_src_format      */
          string project = “CLUSTERII”;            /*   pot_src_project     */
          string mission = “CLUSTER-1”;            /*   pot_src_mission     */
          string experiment = “EFW”;               /*   pot_src_exper       */
          string instrument = “POTENTIAL”;         /*   pot_src_inst        */
          string vinstrument = “SCPOT”;            /*   pot_src_vinst       */
```

```
        int potential_sensor = 0;                /*  pot_src_sen        */
        int num_tbls = 1;                        /*  pot_src_apps       */
        int tbls = 0;                            /*  pot_src_tbls       */
        int opers = 0;                           /*  pot_src_ops        */
        float constant_value = 0.0;              /*  pot_constant_val   */
};
```

The second example shows the VIDF entries for the PotentialSource block when a constant spacecraft potential value is defined.

```
struct  PotentialSource {
        int format = 0;                          /*  pot_src_format     */
        float constant_value = -1.0;             /*  pot_constant_val   */
};
```

5.7 CALIBRATION SET Expansion Information

The original definition of the IDFS data storage format attached calibration sets to each of the IDFS sensors defined for the virtual instrument in question. If the same calibration data was to be used by all sensors, the calibration data had to be written into the data record **n_sen** times per sensor set, with **n_sen** being defined in the header record. The definition of calibration sets has been expanded such that a calibration set can be written to apply to all sensors within a sensor set of the data record, instead of once per sensor.

As defined in Section 4.8.5 (**cal_target**), calibration sets which target the scan data must precede those that apply to the sensor data. The following demonstrates how calibration data would be ordered within the data array:

Calibration Sets written once per sensor set - used by all sensors - (**cal_scope** = 1)		Calibration Sets written once per sensor (**cal_scope** = 0)	
Calibration Sets which target scan data (**cal_target** = 1)	Calibration Sets which target sensor data (**cal_target** = 0)	Calibration Sets which target scan data (**cal_target** = 1)	Calibration Sets which target sensor data (**cal_target** = 0)

This usage of calibration sets can be achieved through the use of the VIDF field **cal_scope**. This field must be defined within the CalSet block structure which is defined for token-tagged VIDF files only.

5.7.1 cal_scope

This field identifies how the calibration data is written in the IDFS data record for the calibration set being defined. If the value for this field is set to 0, the IDFS data access software expects that the calibration set is written once per sensor within the data record, which is the original IDFS definition for calibration data. If this field is set to 1, the IDFS data access software expects that the calibration set is written once per sensor set within the data record. For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this

field will be returned as 0 by the IDFS data access software which parses the VIDF files. This is also the case if the **cal_scope** field is not defined within the token-tagged VIDF file for the calibration set in question.

5.7.2 cal_d_type

The original definition of the IDFS data storage format did not allow for the specification of the data format for the calibration data. It was mandated that the calibration data be represented as unsigned integer binary data. With the new token-tagged VIDF file, how the calibration data is represented within the IDFS data files for each of the defined calibration sets can be specified using the **cal_d_type** field. The values for this field are identical to the values defined for the **d_type** field (refer to section 4.5.5). However, for the sake of clarity, the definitions are repeated in the table below.

CAL_D_TYPE FIELD DEFINITIONS			
VALUE	DEFINITIONS	EXPONENT BASE	WORD LENGTH (bits)
0	unsigned integer, binary data	-	**cal_wlen**
1	signed integer, binary data	-	**cal_wlen**
2	single precision, floating point data	10	32
3	double precision, floating point data	10	64
4	half precision 1, floating point data	10	16
5	half precision 2, floating point data	2	16
6	half precision 3, floating point data	2	16

For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as 0 by the IDFS data access software which parses the VIDF files. This is also the case if the **cal_d_type** field is not defined within the token-tagged VIDF file.

Remember, when writing an IDFS data file, the largest word size of both sensor and calibration data must be used as the word size throughout the data record.

5.7.3 Example CALIBRATION SET Expansion Information Entry

The example below shows 4 defined calibration sets where the first two sets are to be written once per sensor set (**cal_scope** = 1) and the last two sets are to be written once per sensor (**cal_scope** = 0). Notice that within the CalSet2 block structure, **cal_scope** is not defined, which defaults to the original IDFS definition for calibration data. The first and third calibration sets are applied to the scan data (**cal_target** = 1), and the second and fourth calibration sets are applied to the sensor data (**cal_target** = 0). This definition complies with the order requirements as defined in Section 5.7. In addition, the data format for the first calibration set is being set to indicate that the data for calibration set zero is represented within the IDFS data files as single precision, floating point data. Since the other calibration sets do not specify the data format, they will be defaulted to the original IDFS definition for calibration data.

```
int     n_cal_sets = 4;                          /*  cal_sets    */
struct  CalSet0 {
        string name = “Scanline Number”;         /*  name        */
```

```
        int use = 0;                                    /*  use          */
        int word_len = 16;                              /*  word length  */
        int target = 1;                                 /*  target       */
        int scope = 1;                                  /*  scope        */
        int d_type = 2;                                 /*  d_type       */
};
struct  CalSet1 {
        string name = "Gain Code";                      /*  name         */
        int use = 0;                                    /*  use          */
        int word_len = 16;                              /*  word length  */
        int target = 0;                                 /*  target       */
        int scope = 1;                                  /*  scope        */
};
struct  CalSet2 {
        string name = "Scan Offset";                    /*  name         */
        int use = 0;                                    /*  use          */
        int word_len = 16;                              /*  word length  */
        int target = 1;                                 /*  target       */
};
struct  CalSet3 {
        string name = "Gain Format (Log/Linear)";       /*  name         */
        int use = 0;                                    /*  use          */
        int word_len = 1;                               /*  word length  */
        int target = 0;                                 /*  target       */
        int scope = 0;                                  /*  scope        */
};
```

5.8 SCALAR PACKING Information

Within the IDFS paradigm, scalar data are singular measurements which depend at most only on position and time. In order to condense the size of a data file, the IDFS data storage format allows multiple measurements from a scalar sensor to be packed into a single sensor set before being written into the data file. The actual number of samples packed into a sensor set is contained in the **n_sample** field within the header record, not within the VIDF file. When packing scalar data, the maximum packing size for scalar data must be specified using the VIDF field **max_packing**. This piece of information is vital when there is pitch angle data to be computed for the data source in question. The packing size is needed in order to allocate space to hold the information pertinent to pitch angle computation since there will be one pitch angle value computed for each of the packed scalar values contained within the data record.

5.8.1 max_packing

This field defines the maximum number of data samples which can be packed into a single sensor set for scalar sensors (**smp_id** = 2). In other words, the **max_packing** field should be set to the maximum value which could be placed in the **n_sample** field in the IDFS header record for the data source in question. For backwards compatibility, when fixed-formatted VIDF files are being processed, the value for this field will be returned as 1 by the IDFS data access

software which parses the VIDF files. This is also the case if the **max_packing** field is not defined within the token-tagged VIDF file.

5.8.2 Example SCALAR PACKING Information Entry

The example shown below indicates that the maximum number of packed scalar sensors contained within a sensor set is 20.

```
int     max_packing = 20;                                   /*      max_packing      */
```

5.9 START OF SPIN Information

Within the IDFS paradigm, there are two methods utilized to determine the start of spin for an IDFS data source. The first method is referred to as the angular method since the start of spin is flagged as the point at which the azimuthal angle crosses a specified angle value (either 0 degrees or some other constant value defined as **const_id** = 14). Since the azimuthal angle values that are computed and returned for each sweep may not return the **exact** angle specified as the marker for the start of a spin, the IDFS data access software takes into account a tolerance factor. This tolerance factor is based on the instrument timing and is computed as follows:

(360.0 [deg/rev] / **spin** [rev/msec]) * (**swp_time** [msec] **/ n_sample**)

where **spin** is the azimuthal rate of rotation for the virtual instrument, **swp_time** is the time duration of the sweep, and **n_sample** is the number of data samples returned. This tolerance factor is added to the angle value that marks the start of a spin to form an angular range and when an azimuthal angle value from the current sweep is found within this angular range, this element of the sweep is flagged as the start of spin. This method is the default method that is used within the IDFS paradigm.

The angular method works in most cases; however, when there is a gap in spin, there is the possibility that data from different spin periods can be returned together as a single composite spin. In order to remedy this problem, another method has been provided which uses time for the determination of start of spin. This method allows the time of each spin to be explicitly defined and is specified by defining an IDFS data source that is to be used to determine the spin periods. Within the VIDF, the spin timing definition is similar to the VIDF definition for pitch angle information (section 4.10) and is defined using the nine fields that are described below. Upon start of execution, if any problems are encountered in accessing the specified IDFS data source, the method that is used to determine the start of spin is reverted to the angular method.

5.9.1 start_spin_defined

This field indicates whether start of spin information has been defined within the VIDF. For a token-tagged VIDF file, start of spin definitions are specified by the presence of a StartOfSpin block structure. If this structure is not present, start of spin information is not defined for the IDFS in question and the next eight fields are not used; otherwise, the next eight fields are considered active.

5.9.2 start_spin_project

The IDFS lineage of the data file location for the start of spin information is specified utilizing the fields **start_spin_project**, **start_spin_mission**, **start_spin_exper**, **start_spin_inst** and **start_spin_vinst**. The **start_spin_project** field is the ASCII project acronym under which the start of spin values are found. It is limited to 20 characters in length.

5.9.3 start_spin_mission

This is the ASCII mission acronym under which the start of spin information is to be found. It is limited to 20 characters in length.

5.9.4 start_spin_exper

This is the ASCII experiment acronym under which the start of spin information is to be found. It is limited to 20 characters in length.

5.9.5 start_spin_inst

This is the ASCII instrument acronym under which the start of spin information is to be found. It is limited to 20 characters in length.

5.9.6 start_spin_vinst

This is the ASCII virtual instrument acronym in which the start of spin information is to be found . It is limited to 20 characters in length.

5.9.7 start_spin_sensor

This integer field is the sensor number of the source for the start of spin information as defined in the designated IDFS source.

5.9.8 start_spin_msec_adj

Part of the start of spin information that is retrieved from the specified IDFS source is the start time and the stop time associated with each identified spin. The IDFS data access software returns every time tag in 2 parts, one which is expressed in milliseconds of the day and one which contains the remaining nanoseconds of the day. The milliseconds of the day time element can be adjusted by specifying a non-zero adjustment factor for the **start_spin_msec_adj** field. This value can be either positive or negative. If there is no adjustment necessary, the value should be set to zero.

5.9.9 start_spin_nsec_adj

Part of the start of spin information that is retrieved from the specified IDFS source is the start time and the stop time associated with each identified spin. The IDFS data access software returns every time tag in 2 parts, one which is expressed in milliseconds of the day and one which contains the remaining nanoseconds of the day. The residual nanoseconds of the day time element can be adjusted by specifying a non-zero adjustment factor for the **start_spin_nsec_adj** field. This value can be either positive or negative. If there is no adjustment necessary, the value should be set to zero. The nanosecond time adjustment factor is interpreted as a signed, 4-byte quantity and therefore, can reach a maximum of 2,147,483,647.

However, since this value contains the resolution between nanosecond and millisecond precision, the absolute value of this field should be no larger than 999,999.

5.9.10 Example START OF SPIN Information Entries

The example shown below defines the IDFS source that is to be used for the start of spin determination.

```
struct    StartOfSpin {
          string project = "CLUSTERII";              /*  start_spin_project      */
          string mission = "CLUSTER-2";              /*  start_spin_mission      */
          string experiment = "AUXILIARY";           /*  start_spin_exper        */
          string instrument = "OA";                  /*  start_spin_inst         */
          string vinstrument = "CSPSPIN";            /*  start_spin_vinst        */
          int sensor = 0;                            /*  start_spin_sensor       */
          int msec_adj = 0;                          /*  start_spin_msec_adj     */
          int nsec_adj = 0;                          /*  start_spin_nsec_adj     */
};
```

5.10 BACKGROUND Information

The IDFS data access software now has the capability of returning background data for the IDFS data source in question. The background data may be used to adjust the sensor data for calculations that determine absolute quantities. The background data may be defined as a constant value or as a dynamic value from a separate IDFS data source. The specification of a dynamic value is similar to the VIDF definition for pitch angle information (section 4.10) and Euler angle rotation information (section 5.5.2). A dynamic background value must be contained within a scalar IDFS data source. The specification of the background data is defined using the eleven fields that are described below.

5.10.1 bkgd_defined

This field indicates whether background information has been defined within the VIDF. For a token-tagged VIDF file, background definitions are specified by the presence of a Background block structure. If this structure is not present, background information is not defined for the IDFS in question and the next ten fields are not used; otherwise, the next ten fields are considered active.

5.10.2 bkgd_format

This integer field is used to indicate which algorithm is to be used in computing the background values. Currently, the only "algorithm" defined is the retrieval of the values from a specified IDFS data source and the conversion of those values to physical units. Therefore, the value for this field should be set to 1. However, if the value of this field is set to 0, the background values that are returned are the values that are defined in the CONSTANTS section of the VIDF, identified as having a **const_id** value of 17.

5.10.3 bkgd_project

This is the ASCII project acronym under which the background sensor is to be found for a dynamic background value. It is limited to 20 characters in length. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.4 bkgd_mission

This is the ASCII mission acronym under which the background sensor is to be found for a dynamic background value. It is limited to 20 characters in length. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.5 bkgd_exper

This is the ASCII experiment acronym under which the background sensor is to be found for a dynamic background value. It is limited to 20 characters in length. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.6 bkgd_inst

This is the ASCII instrument acronym under which the background sensor is to be found for a dynamic background value. It is limited to 20 characters in length. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.7 bkgd_vinst

This is the ASCII virtual instrument acronym in which the background sensor is to be found for a dynamic background value. It is limited to 20 characters in length. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.8 bkgd_sensors

This field is an integer array of length **sen**, giving the sensor number of the source for the background as defined in the designated IDFS source. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.9 bkgd_apps

This field is an integer array of length **sen**, defining the number of tables which must be applied to the defined sensor (the source for the background) to bring its value to the appropriate units necessary for computation. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.10 bkgd_tbls

An integer array, whose size is defined by the summation of the **bkgd_apps** values, which gives the table numbers which will need to be applied to the sensor defined as the source of the background. Note that these table numbers are obtained from the VIDF pertaining to the virtual instrument in which the background sensor is defined. If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.11 bkgd_ops

An integer array, whose size is defined by the summation of the **bkgd_apps** values, which gives the table operations which will need to be applied to each of the tables specified in **bkgd_tbls** (see Appendix A of the PIDF documentation *http://www.idfs.org/Editors/pidfdoc.html* for valid table operation entries). If the **bkgd_format** value indicates that the background is a constant value, this field should be omitted from the Background block structure.

5.10.12 Example BACKGROUND Information Entries

Shown below are two examples of the Background information entries. The first example defines a dynamic background value that is retrieved from the MARS / Mars_Express / ASPERA-3 / ELS / ELS05BK IDFS data set.

```
struct    Background {
          int format = 1;                              /*  bkgd_format      */
          string project = "MARS";                     /*  bkgd_project     */
          string mission = "Mars_Express";             /*  bkgd_mission     */
          string experiment = "ASPERA-3";              /*  bkgd_exper       */
          string instrument = "ELS";                   /*  bkgd_inst        */
          string vinstrument = "ELS05BK";              /*  bkgd_vinst       */
          int sensors [16] = {                         /*  bkgd_sensors     */
                1,  2,  3,  4,  5,  6,  7,  8,         /*  000 - 007        */
                9, 10, 11, 12, 13, 14, 15, 16          /*  008 - 015        */
          };                                           /*                   */
          int num_tbls [16] = {                        /*  bkgd_apps        */
                2,  2,  2,  2,  2,  2,  2,  2,         /*  000 - 007        */
                2,  2,  2,  2,  2,  2,  2,  2          /*  008 - 015        */
          };                                           /*                   */
          int tbls [32] = {                            /*  bkgd_tbls        */
                0,  4,  0,  5,  0,  6,  0,  7,         /*  0000 - 0007      */
                0,  8,  0,  9,  0, 10,  0, 11,         /*  0008 - 0015      */
                0, 12,  0, 13,  0, 14,  0, 15,         /*  0016 - 0023      */
                0, 16,  0, 17,  0, 18,  0, 19          /*  0024 - 0031      */
          };                                           /*                   */
          int opers [32] = {                           /*  bkgd_ops         */
                0,  4,  0,  4,  0,  4,  0,  4,         /*  0000 - 0007      */
                0,  4,  0,  4,  0,  4,  0,  4,         /*  0008 - 0015      */
                0,  4,  0,  4,  0,  4,  0,  4,         /*  0016 - 0023      */
```

```
            0,  4,  0,  4,  0,  4,  0,  4          /*    0024 - 0031        */
        };                                         /*                       */
};
```

The second example shows the VIDF entries for the **Background** block when background values are to be returned, but the values are defined as constants. The user is reminded that when this format is selected, the background constants (**const_id** = 17) must also be defined in the VIDF (refer to section 4.15).

```
struct    Background {
          int format = 0;                          /*    bkgd_format        */
};
```

Note that while the Background block structure is defined, it does not contain the information that is pertinent for the extraction of the data from a specified IDFS data source. In addition, the values defined as constants in the VIDF file must already be specified in the desired units.

6. Multiple VIDF files for a Given Instrument

If data within the VIDF file changes with time, for example the calibration coefficients need to be modified, multiple VIDF files for the same virtual instrument can be defined. Each VIDF file must specify a unique, valid time period over which the information in the VIDF is applicable to the IDFS data set. In other words, the time periods covered by the multiple VIDF files must not overlap and should define a time continuum. For example, if the first VIDF file specifies:

ds_year	1991	de_year	1991
ds_day	256	de_day	256
ds_msec	0	de_msec	8180000
ds_usec	0	de_usec	0

and the next VIDF file specifies:

ds_year	1991	de_year	1991
ds_day	256	de_day	257
ds_msec	81790000	de_msec	0
ds_usec	0	de_usec	0

there is an overlap in time between the two VIDF files and **this is not allowed**. In addition, there is a lower limit to the minimum timespan that a single VIDF file may contain. For database and archiving purposes, VIDF file names contain the start year, day, hour, and minute for which the VIDF file is defined. Therefore, each VIDF file must be defined to span at least one minute in time; otherwise, an overwrite of multiple VIDF files would happen.

If the multiple VIDF files are divided at data/header file boundaries, there are no further restrictions that are imposed upon the creation of the VIDF files. The state of the instrument at each of the designated time segments can be defined by the creation of multiple VIDF files.

A different situation arises when more than one VIDF file corresponds to a single data file; that is, the cross over to a new VIDF file happens within a data record in the middle of a data file. There are some defined parameters within the VIDF file that **can not** change from VIDF to VIDF, even though the data set may require that change. The following table lists the VIDF fields that must preserve the same value across multiple VIDF files:

VIDF FIELD	COMMENT
smp_id	
d_type	
tdw_len	
sen_mode	
da_method	
cal_sets	necessary in order to preserve the size of the data record.
cal_use	necessary in order to preserve the size of the data record.
cal_wlen	necessary in order to preserve the size of the data record.

VIDF FIELD	COMMENT
cal_scope	
cal_target	
cal_d_type	
max_nss	
data_len	
status	
sen	use **n_sen** and **sensor_index**[] in the header record to specify which sensors are actually returned. In addition, make sure that sensor definitions are consistent from VIDF to VIDF; that is, make sure sensor X is the same parameter definition across all defined VIDF's.
num_tbls	since PIDF contains the definition of how to construct the units using the tables and the PIDF file is not "time sensitive", only one PIDF file defined per virtual instrument.
swp_len	use **n_sample** in the header record to indicate the actual number of transmitted values
time_off	values do not have to stay the same BUT all time_off values should be set so that the end time of the data sample(s) for all sensors contained within a single sensor set reference a single VIDF file.
max_packing	
nano_defined	

Any changes to the values for any of these VIDF parameters mandates that the use of multiple VIDF files be abandoned and that the data set be constructed as a different, new virtual instrument. However, there are some values which can be modified between the multiple instances of the VIDF file. These include:

- the time period over which the information in the VIDF is valid
- the definition of a fill value
- the status of each defined sensor
- the inclusion / exclusion of pitch angle information, including modifications to the IDFS source that is used to compute the pitch angle values
- the contents of the tables that are defined within the VIDF (note that while the contents may change, the number of tables cannot change due to PIDF restrictions / interactions)
- the number of constants defined and the values for these constants
- the azimuthal angle computation flag

7. FIXED-FORMATTED VIDF EXAMPLE

The fixed-formatted VIDF file is built as an ASCII file and then converted into a binary file that is used by the IDFS data access software (the binary file allows for faster access). The ASCII file consists of an entry definition field followed by a VIDF data field value and an optional comment field. The following example includes all standard entries plus many of the optional fields. This VIDF describes the nadir pointing electron sensors on the Medium Energy Particle Spectrometer (MEPS) portion of the Particle Environment Monitor (PEM) aboard the Upper Atmosphere Research Satellite (UARS).

There are twelve (12) tables of varying types including an ASCII table. The detector efficiencies table (TABLE 2) makes use of the critical action feature where there are actually two tables defined because the efficiencies vary depending on the applied bias voltage --"HVPS3 State" (**crit_status**=1 for each sensor and "HVPS3 State" is the second defined **status_names** in the VIDF [indexes and offsets start at 0]). There are two states defined for "HVPS3 State" (**states**[1]=2) where the first table is used for lookup when the mode value is 0, and when the mode value is 1, the second table is used for lookup. If a table had been defined depending on the "Satellite Aspect" mode, there could potentially be 4 tables to switch between (**states**[0]=4). The mode values (status bytes) are written to the Header File (**mode_index[i_mode]**) since these are slowly varying parameters.

This VIDF also has pitch angle defined where the UARS/PEM detected B1, B2, and B3 values (pitch angle source) are used in the calculations. The three (3) defined constants are the Axis A, Axis B, and Axis C components, respectively, of the aperture normals with respect to the PEM magnetometer (VMAG). These constants along with the pitch angle source are necessary for pitch angle computations.

```
t    Upper Atmospheric Research Satellite                  /*  mission      */
t    UARS                                                  /*  spacecraft   */
t    Medium Energy Electron and Ion Measurements           /*  exp_desc     */
t    Nadir Pointing Electron Sensors                       /*  inst_desc    */
m    5   1                                                 /*  contact      */
t    Dr. J. David Winningham                               /*  00000        */
t    Southwest Research Institute                          /*  00001        */
t    6220 Culebra Road                                     /*  00002        */
t    San Antonio, Texas 78238                              /*  00003        */
t    david@cluster.space.swri.edu                          /*  00004        */
s    35                                                    /*  num_comnts   */
m    35   1                                                /*  comments     */
t    The following is a list of tables which are in this vidf  /*  00000     */
t    TABLE 0:  center energies (eV)                        /*  00001        */
t    TABLE 1:  telemetry decom table                       /*  00002        */
t    TABLE 2:  detector efficiencies                       /*  00003        */
t    TABLE 3:  geometry factors (cm**2-str)                /*  00004        */
t    TABLE 4:  dE/E                                        /*  00005        */
t    TABLE 5:  center energies (ergs)                      /*  00006        */
```

t	TABLE 6: (center energies)**2 (ergs**2)				/*	00007	*/
t	TABLE 7: constant needed in going to dist. fn				/*	00008	*/
t	TABLE 8: Threshold count level				/*	00009	*/
t	TABLE 9: Dark count level				/*	00010	*/
t	TABLE 10: Amplifier Dead Time Factor				/*	00011	*/
t	TABLE 11: ASCII descriptions of status states				/*	00012	*/
t					/*	00013	*/
t	The following are units which can be derived from the tables.				/*	00014	*/
t	The format is to give the tables applied followed by the				/*	00015	*/
t	operations and unit definition				/*	00016	*/
t					/*	00017	*/
t	DATA TYPE	TABLES	OPERS	UNIT	/*	00018	*/
t	Scan	0	0	eV	/*	00019	*/
t	Sensor	1	0	cnts/accum	/*	00020	*/
t	Sensor	1,2	0,4	cnts/accum (eff. cor)	/*	00021	*/
t	Sensor	1,2	0,154	cnts/sec	/*	00022	*/
t	Sensor	1,2,3,4	0,154,4,4	cnts/(cm**2-str-s)	/*	00023	*/
t	Sensor	1,2,3,4,0	0,154,4,4,4	cnts/(cm**2-str-s-eV)	/*	00024	*/
t	Sensor	1,2,3,4,0,5	0,154,4,4,4,3	ergs/(cm**2-str-s-eV)	/*	00025	*/
t	Sensor	1,2,3,4,7,6	0,154,4,4,3,4	sec**3/km**6	/*	00026	*/
t					/*	00027	*/
t	The following is a list of constant values which are in this vidf				/*	00028	*/
t	CONST 0: These are the Axis A components of each sensor's				/*	00029	*/
t	aperture expressed relative to the PEM magnetometer.				/*	00030	*/
t	CONST 1: These are the Axis B components of each sensor's				/*	00031	*/
t	aperture expressed relative to the PEM magnetometer.				/*	00032	*/
t	CONST 2: These are the Axis C components of each sensor's				/*	00033	*/
t	aperture expressed relative to the PEM magnetometer.				/*	00034	*/
s	1980				/*	ds_year	*/
s	1				/*	ds_day	*/
l	0				/*	ds_msec	*/
s	0				/*	ds_usec	*/
s	2020				/*	de_year	*/
s	1				/*	de_day	*/
l	0				/*	de_msec	*/
s	0				/*	de_usec	*/
b	1				/*	smp_id	*/
b	2				/*	sen_mode	*/
b	4				/*	n_qual	*/
b	2				/*	cal_sets	*/
b	12				/*	num_tbls	*/
b	3				/*	num_consts	*/
b	2				/*	status	*/
b	1				/*	pa_defined	*/
s	3				/*	sen	*/
s	31				/*	swp_len	*/

```
s     48                                                    /*  max_nss      */
l     3952                                                  /*  data_len     */
b     0                                                     /*  fill_flg     */
n                                                           /*  fill value   */
b     0                                                     /*  da_method    */
m     2  1                                                  /*  status_name  */
                                                                s
t     Satellite Aspect                                      /*  00000        */
t     HVPS3 State                                           /*  00001        */
m     2  2                                                  /*  states       */
s           4        2                                      /*  00000-       */
                                                                00001
m     3  1                                                  /*  sen_name     */
t     ESensor 10: 126.3 degrees                             /*  00000        */
t     ESensor 12: 156.3 degrees                             /*  00001        */
t     ESensor 14: -158.7 degrees                            /*  00002        */
m     2  1                                                  /*  cal_names    */
t     Fill Flags Per Energy Step                            /*  00000        */
t     CRC Flags Per Energy Step                             /*  00001        */
m     4  1                                                  /*  qual_name    */
t     No Fill Data                                          /*  00000        */
t     Fill Data With Energy Sweep                           /*  00001        */
t     Possible Fill Data With Energy Sweep                  /*  00002        */
t     Fill And Possible Fill Data With Energy Sweep         /*  00003        */
s     1                                                     /*  pa_format    */
T     UARS                                                  /*  pa_project   */
T     UARS-1                                                /*  pa_mission   */
T     PEM                                                   /*  pa_exper     */
T     VMAG                                                  /*  pa_inst      */
T     VMMA                                                  /*  pa_vinst     */
m     3  3                                                  /*  num_pa_sen   */
s           0        1        2                             /*  b1-b2-b3     */
s     1                                                     /*  pa_apps      */
m     1  1                                                  /*  num_pa_tbls  */
s     1                                                     /*  pa_tbl_nums  */
m     1  1                                                  /*  num_pa_ops   */
s     0                                                     /*  pa_opers     */
m     3  3                                                  /*  d_type       */
b           0        0        0                             /*  00000-       */
                                                                00002
m     3  3                                                  /*  tdw_len      */
b           8        8        8                             /*  00000-       */
                                                                00002
m     3  3                                                  /*  sen_status   */
b           1        1        1                             /*  00000-       */
                                                                00002
```

```
m    3  3                                                    /*  time_off      */
l              0        0        0                           /*  00000-        */
                                                                00002
m    2  2                                                    /*  cal_use       */
s              8        8                                    /*  00000-        */
                                                                00001
m    2  2                                                    /*  cal_wlen      */
b              8        8                                    /*  00000-        */
                                                                00001
m    2  2                                                    /*  cal_target    */
b              1        1                                    /*  00000-        */
                                                                00001
l    -3                                                      /*  tbl_sca_sz    */
l    93                                                      /*  tbl_ele_sz    */
b    0                                                       /*  tbl_type      */
s    1                                                       /*  num comnts    */
m    1  1                                                    /*  tbl_desc      */
t    This table contains the center energies in eV           /*  00000         */
b    2                                                       /*  tbl_var       */
b    1                                                       /*  tbl_expand    */
l    0                                                       /*  crit_act_sz   */
n                                                            /*  crit_status   */
n                                                            /*  crit_sen_off  */
n                                                            /*  crit_offs     */
m    3  3                                                    /*  tbl_fmt       */
b              0        0        0                           /*  00000-        */
                                                                00002
m    3  3                                                    /*  tbl_off       */
l              0       31       62                           /*  00000-        */
                                                                00002
m    3  3                                                    /*  tbl_sca       */
b             -3       -3       -3                           /*  00000-        */
                                                                00002
m    93  5                                                   /*  tbl           */
l        28776058 20010449 13907228  9658524  6711579        /*  00000-        */
                                                                00004
l         4664744  3232809  2252238  1560147  1087850        /*  00005-        */
                                                                00009
l          758814   526591   367631   255315   177274        /*  00010-        */
                                                                00014
l          120717    81964    59402    41316    28677        /*  00015-        */
                                                                00019
l           19983    13920     9642     6730     4631        /*  00020-        */
                                                                00024
l            3263     2276     1557     1072      674        /*  00025-        */
                                                                00029
```

```
l              523    29629464    20603894    14319672     9944965     /*  00030-00034  */
l          6910623     4803084     3328684     2319031     1606416     /*  00035-00039  */
l          1120114      781317      542208      378535      262886     /*  00040-00044  */
l           182531      124296       84396       61165       42541     /*  00045-00049  */
l            29527       20576       14333        9928        6931     /*  00050-00054  */
l             4769        3361        2344        1603        1105     /*  00055-00059  */
l              694         538    29917599    20804259    14458925     /*  00060-00064  */
l         10041676     6977826     4849792     3361053     2341583     /*  00065-00069  */
l          1622038     1131005      788916      547481      382216     /*  00070-00074  */
l           265442      184307      125505       85217       61759     /*  00075-00079  */
l            42955       29815       20777       14472       10024     /*  00080-00084  */
l             6998        4815        3394        2367        1618     /*  00085-00089  */
l             1115         700         544                             /*  00090-00092  */
l    -3                                                                 /*  tbl_sca_sz   */
l    256                                                                /*  tbl_ele_sz   */
b    0                                                                  /*  tbl_type     */
s    2                                                                  /*  num comnts   */
m    2 1                                                                /*  tbl_desc     */
t    This table contains the decompress from telemetry to               /*  00000        */
t    counts per accumulation period                                     /*  00001        */
b    0                                                                  /*  tbl_var      */
b    1                                                                  /*  tbl_expand   */
l    0                                                                  /*  crit_act_sz  */
n                                                                       /*  crit_status  */
n                                                                       /*  crit_sen_off */
n                                                                       /*  crit_offs    */
m    3  3                                                               /*  tbl_fmt      */
b         0       0       0                                            /*  00000-00002  */
m    3  3                                                               /*  tbl_off      */
l         0       0       0                                            /*  00000-00002  */
m    3  3                                                               /*  tbl_scale    */
```

```
b        -1     -1     -1                              /* 00000-00002 */
m   256  5                                             /* tbl */
l         0     10     20     30     40                /* 00000-00004 */
l        50     60     70     80     90                /* 00005-00009 */
l       100    110    120    130    140                /* 00010-00014 */
l       150    165    175    185    195                /* 00015-00019 */
l       205    215    225    235    245                /* 00020-00024 */
l       255    265    275    285    295                /* 00025-00029 */
l       305    315    330    350    370                /* 00030-00034 */
l       390    410    430    450    470                /* 00035-00039 */
l       490    510    530    550    570                /* 00040-00044 */
l       590    610    630    660    700                /* 00045-00049 */
l       740    780    820    860    900                /* 00050-00054 */
l       940    980   1020   1060   1100                /* 00055-00059 */
l      1140   1180   1220   1260   1320                /* 00060-00064 */
l      1400   1480   1560   1640   1720                /* 00065-00069 */
l      1800   1880   1960   2040   2120                /* 00070-00074 */
l      2200   2280   2360   2440   2520                /* 00075-00079 */
l      2640   2800   2960   3120   3280                /* 00080-00084 */
l      3440   3600   3760   3920   4080                /* 00085-00089 */
l      4240   4400   4560   4720   4880                /* 00090-00094 */
l      5040   5280   5600   5920   6240                /* 00095-00099 */
l      6560   6880   7200   7520   7840                /* 00100-00104 */
l      8160   8480   8800   9120   9440                /* 00105- */
```

```
                                                              00109
l      9760    10080    10560    11200    11840          /*   00110-       */
                                                              00114
l     12480    13120    13760    14400    15040          /*   00115-       */
                                                              00119
l     15680    16320    16960    17600    18240          /*   00120-       */
                                                              00124
l     18880    19520    20160    21120    22400          /*   00125-       */
                                                              00129
l     23680    24960    26240    27520    28800          /*   00130-       */
                                                              00134
l     30080    31360    32640    33920    35200          /*   00135-       */
                                                              00139
l     36480    37760    39040    40320    42240          /*   00140-       */
                                                              00144
l     44800    47360    49920    52480    55040          /*   00145-       */
                                                              00149
l     57600    60160    62720    65280    67840          /*   00150-       */
                                                              00154
l     70400    72960    75520    78080    80640          /*   00155-       */
                                                              00159
l     84480    89600    94720    99840   104960          /*   00160-       */
                                                              00164
l    110080   115200   120320   125440   130560          /*   00165-       */
                                                              00169
l    135680   140800   145920   151040   156160          /*   00170-       */
                                                              00174
l    161280   168960   179200   189440   199680          /*   00175-       */
                                                              00179
l    209920   220160   230400   240640   250880          /*   00180-       */
                                                              00184
l    261120   271360   281600   291840   302080          /*   00185-       */
                                                              00189
l    312320   322560   337920   358400   378880          /*   00190-       */
                                                              00194
l    399360   419840   440320   460800   481280          /*   00195-       */
                                                              00199
l    501760   522240   542720   563200   583680          /*   00200-       */
                                                              00204
l    604160   624640   645120   675840   716800          /*   00205-       */
                                                              00209
l    757760   798720   839680   880640   921600          /*   00210-       */
                                                              00214
l    962560  1003520  1044480  1085440  1126400          /*   00215-       */
                                                              00219
l   1167360  1208320  1249280  1290240  1351680          /*   00220-       */
```

```
                                                                                  00224
l       1433600  1515520  1597440  1679360  1761280                      /*  00225-  */
                                                                                  00229
l       1843200  1925120  2007040  2088960  2170880                      /*  00230-  */
                                                                                  00234
l       2252800  2334720  2416640  2498560  2580480                      /*  00235-  */
                                                                                  00239
l       2703360  2867200  3031040  3194880  3358720                      /*  00240-  */
                                                                                  00244
l       3522560  3686400  3850240  4014080  4177920                      /*  00245-  */
                                                                                  00249
l       4341760  4505600  4669440  4833280  4997120                      /*  00250-  */
                                                                                  00254
l       5160960                                                          /*  00255   */
l    -3                                                                  /*  tbl_sca_sz  */
l    186                                                                 /*  tbl_ele_sz  */
b    0                                                                   /*  tbl_type    */
s    4                                                                   /*  num comnts  */
m    4 1                                                                 /*  tbl_desc    */
t    This table contains channeltron efficiency as a function            /*  00000   */
t    of energy step.  The table is repeated for each of the              /*  00001   */
t    three detectors and then again since the efficiencies vary          /*  00002   */
t    depending on the applied bias voltage applied.                      /*  00003   */
b    2                                                                   /*  tbl_var     */
b    1                                                                   /*  tbl_expand  */
l    6                                                                   /*  crit_act_sz */
m    3  3                                                                /*  crit_status */
b         1       1       1                                            /*  00000-  */
                                                                                  00002
m    3  3                                                                /*  crit_off    */
s         0       2       4                                            /*  00000-  */
                                                                                  00002
m    6  5                                                                /*  crit_action */
l         0      93      31     124      62                            /*  00000-  */
                                                                                  00004
l       155                                                            /*  00005   */
m    3  3                                                                /*  tbl_fmt     */
b         0       0       0                                            /*  00000-  */
                                                                                  00002
m    3  3                                                                /*  tbl_off     */
l         0       0       0                                            /*  00000-  */
                                                                                  00002
m    3  3                                                                /*  tbl_sca     */
b        -6      -6      -6                                            /*  00000-  */
                                                                                  00002
m    186 5                                                               /*  tbl     */
```

```
l       264919  340689  442820  531910  605260          /*  00000-00004  */
l       671389  730780  782180  827130  864849          /*  00005-00009  */
l       894100  917280  933659  944999  951960          /*  00010-00014  */
l       955820  957629  958090  958050  957740          /*  00015-00019  */
l       957360  957009  956709  956480  956300          /*  00020-00024  */
l       956179  956080  956019  955969  955929          /*  00025-00029  */
l       955910  263729  332670  435290  525690          /*  00030-00034  */
l       599629  666339  726310  778280  823819          /*  00035-00039  */
l       861419  891929  915690  932529  944270          /*  00040-00044  */
l       951529  955609  957560  958079  958069          /*  00045-00049  */
l       957769  957390  957040  956730  956499          /*  00050-00054  */
l       956309  956189  956089  956019  955969          /*  00055-00059  */
l       955929  955910  263619  330029  432819          /*  00060-00064  */
l       523620  597760  664659  724829  776989          /*  00065-00069  */
l       822709  860499  891200  915149  932150          /*  00070-00074  */
l       944019  951390  955540  957530  958069          /*  00075-00079  */
l       958069  957780  957400  957050  956740          /*  00080-00084  */
l       956499  956319  956189  956089  956019          /*  00085-00089  */
l       955969  955929  955919  520107  565243          /*  00090-00094  */
l       611256  657460  702879  746668  788259          /*  00095-00099  */
l       825896  859978  888868  912930  932294          /*  00100-00104  */
l       946394  956121  962158  965722  967494          /*  00105-00109  */
l       968152  968415  968423  968325  968203          /*  00110-00114  */
```

```
l       968086  967990  967913  967859  967818          /*  00115-      */
                                                            00119
l       967788  967768  967750  967742  514510          /*  00120-      */
                                                            00124
l       559463  605354  651510  696959  740858          /*  00125-      */
                                                            00129
l       782639  820528  854918  884146  908569          /*  00130-      */
                                                            00134
l       928354  942942  953207  959770  963828          /*  00135-      */
                                                            00139
l       966024  966980  967521  967748  967823          /*  00140-      */
                                                            00144
l       967827  967806  967779  967754  967733          /*  00145-      */
                                                            00149
l       967718  967705  967696  967688  967685          /*  00150-      */
                                                            00154
l       514685  559668  605598  651808  697324          /*  00155-      */
                                                            00159
l       741312  783203  821220  855759  885142          /*  00160-      */
                                                            00164
l       909718  929630  944285  954533  961004          /*  00165-      */
                                                            00169
l       964914  966938  967751  968141  968240          /*  00170-      */
                                                            00174
l       968209  968132  968047  967975  967916          /*  00175-      */
                                                            00179
l       967872  967840  967814  967797  967782          /*  00180-      */
                                                            00184
l       967778                                          /*  00185       */
l   3                                                   /*  tbl_sca_sz  */
l   3                                                   /*  tbl_ele_sz  */
b   0                                                   /*  tbl_type    */
s   2                                                   /*  num comnts  */
m   2   1                                               /*  tbl_desc    */
t   This table contains the detector geometry factors as /*  00000       */
t   (cm**2-s)                                           /*  00001       */
b   2                                                   /*  tbl_var     */
b   1                                                   /*  tbl_expand  */
l   0                                                   /*  crit_act_sz */
n                                                       /*  crit_status */
n                                                       /*  crit_sen_off*/
n                                                       /*  crit_offs   */
m   3   3                                               /*  tbl_fmt     */
b           1       1       1                           /*  00000-      */
                                                            00002
m   3   3                                               /*  tbl_off     */
```

```
l          0       1       2                                   /*  00000-      */
                                                                   00002
m   3  3                                                       /*  tbl_sca     */
b        -8      -8      -8                                    /*  00000-      */
                                                                   00002
m   3  3                                                       /*  tbl         */
l       20551   11009   18882                                  /*  00000-      */
                                                                   00002
l   -3                                                         /*  tbl_sca_sz  */
l   93                                                         /*  tbl_ele_sz  */
b   0                                                          /*  tbl_type    */
s   1                                                          /*  num comnts  */
m   1  1                                                       /*  tbl_desc    */
t   This table contains the energy resolution (dE/E)           /*  00000       */
b   2                                                          /*  tbl_var     */
b   1                                                          /*  tbl_expand  */
l   0                                                          /*  crit_act_sz */
n                                                              /*  crit_status */
n                                                              /*  crit_sen_off*/
n                                                              /*  crit_offs   */
m   3  3                                                       /*  tbl_fmt     */
b        0       0       0                                     /*  00000-      */
                                                                   00002
m   3  3                                                       /*  tbl_off     */
l        0      31      62                                     /*  00000-      */
                                                                   00002
m   3  3                                                       /*  tbl_sca     */
b       -3      -3      -3                                     /*  00000-      */
                                                                   00002
m   93  5                                                      /*  tbl         */
l       344     344     344     344     344                    /*  00000-      */
                                                                   00004
l       344     344     344     344     344                    /*  00005-      */
                                                                   00009
l       344     344     344     344     344                    /*  00010-      */
                                                                   00014
l       344     344     344     344     344                    /*  00015-      */
                                                                   00019
l       344     344     344     344     344                    /*  00020-      */
                                                                   00024
l       344     344     344     344     344                    /*  00025-      */
                                                                   00029
l       344     355     355     355     355                    /*  00030-      */
                                                                   00034
l       355     355     355     355     355                    /*  00035-      */
                                                                   00039
```

```
l       355     355     355     355     355             /*  00040-      */
                                                            00044
l       355     355     355     355     355             /*  00045-      */
                                                            00049
l       355     355     355     355     355             /*  00050-      */
                                                            00054
l       355     355     355     355     355             /*  00055-      */
                                                            00059
l       355     355     349     349     349             /*  00060-      */
                                                            00064
l       349     349     349     349     349             /*  00065-      */
                                                            00069
l       349     349     349     349     349             /*  00070-      */
                                                            00074
l       349     349     349     349     349             /*  00075-      */
                                                            00079
l       349     349     349     349     349             /*  00080-      */
                                                            00084
l       349     349     349     349     349             /*  00085-      */
                                                            00089
l       349     349     349                             /*  00090-      */
                                                            00092
l   -3                                                  /*  tbl_sca_sz  */
l   93                                                  /*  tbl_ele_sz  */
b   0                                                   /*  tbl_type    */
s   1                                                   /*  num comnts  */
m   1   1                                               /*  tbl_desc    */
t   This table contains the center energies in ergs     /*  00000       */
b   2                                                   /*  tbl_var     */
b   1                                                   /*  tbl_expand  */
l   0                                                   /*  crit_act_sz */
n                                                       /*  crit_status */
n                                                       /*  crit_sen_off*/
n                                                       /*  crit_offs   */
m   3   3                                               /*  tbl_fmt     */
b       0       0       0                               /*  00000-      */
                                                            00002
m   3   3                                               /*  tbl_off     */
l       0       31      62                              /*  00000-      */
                                                            00002
m   3   3                                               /*  tbl_sca     */
b       -16     -16     -16                             /*  00000-      */
                                                            00002
m   93  5                                               /*  tbl         */
l   460992448 320567376 222793792 154729554 107519495   /*  00000-      */
                                                            00004
```

```
l        74729198  51789600  36080852  24993554  17427356  /*  00005-00009  */
l        12156200   8435987   5889448   4090146   2839929  /*  00010-00014  */
l         1933886   1313063    951620    661882    459405  /*  00015-00019  */
l          320127    222998    154464    107814     74188  /*  00020-00024  */
l           52273     36461     24943     17173     10797  /*  00025-00029  */
l            8378 474664012 330074381 229401145 159318339  /*  00030-00034  */
l       110708180  76945405  53325517  37150876  25734784  /*  00035-00039  */
l        17944226  12516698   8686172   6064130   4211433  /*  00040-00044  */
l         2924146   1991221   1352023    979863    681506  /*  00045-00049  */
l          473022    329627    229614    159046    111034  /*  00050-00054  */
l           76399     53843     37550     25680     17702  /*  00055-00059  */
l           11117      8618 479279951 333284244 231631978  /*  00060-00064  */
l       160867649 111784772  77693667  53844068  37512159  /*  00065-00069  */
l        25985048  18118700  12638434   8770645   6123100  /*  00070-00074  */
l         4252380   2952598   2010590   1365176    989379  /*  00075-00079  */
l          688139    477636    332847    231841    160584  /*  00080-00084  */
l          112107     77136     54371     37919     25920  /*  00085-00089  */
l           17862     11213      8714                      /*  00090-00092  */
l    93                                                    /*  tbl_sca_sz   */
l    93                                                    /*  tbl_ele_sz   */
b    0                                                     /*  tbl_type     */
s    2                                                     /*  num comnts   */
m    2  1                                                  /*  tbl_desc     */
t    This table contains the square of the center energies /*  00000        */
t    in ergs**2                                            /*  00001        */
b    2                                                     /*  tbl_var      */
b    1                                                     /*  tbl_expand   */
l    0                                                     /*  crit_act_sz  */
```

```
n                                                                   /*  crit_status     */
n                                                                   /*  crit_sen_off    */
n                                                                   /*  crit_offs       */
m    3  3                                                          /*  tbl_fmt         */
b          0       0       0                                        /*  00000-          */
                                                                        00002
m    3  3                                                          /*  tbl_off         */
l          0      31      62                                        /*  00000-          */
                                                                        00002
m    93  5                                                         /*  tbl_sca         */
b         -23     -23     -24     -24     -24                      /*  00000-          */
                                                                        00004
b         -25     -25     -25     -26     -26                      /*  00005-          */
                                                                        00009
b         -26     -27     -27     -27     -28                      /*  00010-          */
                                                                        00014
b         -28     -28     -29     -29     -29                      /*  00015-          */
                                                                        00019
b         -29     -30     -30     -30     -31                      /*  00020-          */
                                                                        00024
b         -31     -31     -32     -32     -32                      /*  00025-          */
                                                                        00029
b         -33     -23     -23     -24     -24                      /*  00030-          */
                                                                        00034
b         -24     -25     -25     -25     -26                      /*  00035-          */
                                                                        00039
b         -26     -26     -27     -27     -27                      /*  00040-          */
                                                                        00044
b         -28     -28     -28     -29     -29                      /*  00045-          */
                                                                        00049
b         -29     -29     -30     -30     -30                      /*  00050-          */
                                                                        00054
b         -31     -31     -31     -32     -32                      /*  00055-          */
                                                                        00059
b         -32     -33     -23     -23     -24                      /*  00060-          */
                                                                        00064
b         -24     -24     -25     -25     -25                      /*  00065-          */
                                                                        00069
b         -26     -26     -26     -27     -27                      /*  00070-          */
                                                                        00074
b         -27     -28     -28     -28     -29                      /*  00075-          */
                                                                        00079
b         -29     -29     -29     -30     -30                      /*  00080-          */
                                                                        00084
b         -30     -31     -31     -31     -32                      /*  00085-          */
                                                                        00089
```

b	-32	-32	-33			/*	00090-00092	*/
m	93 5					/*	tbl	*/
l	212514037	102763442	496370738	239412349	115604419	/*	00000-00004	*/
l	558445315	268216268	130182793	624677787	303712771	/*	00005-00009	*/
l	147773204	711658903	346856049	167292967	806519943	/*	00010-00014	*/
l	373991636	172413517	905580698	438088204	211053449	/*	00015-00019	*/
l	102481718	497282862	238593867	116239879	550395132	/*	00020-00024	*/
l	273249370	132944243	622160231	294927040	116585574	/*	00025-00029	*/
l	701985918	225305924	108949097	526248854	253823331	/*	00030-00034	*/
l	122563011	592059544	284361082	138018763	662279122	/*	00035-00039	*/
l	321995256	156667736	754495866	367736810	177361739	/*	00040-00044	*/
l	855063343	396496472	182796867	960132084	464451544	/*	00045-00049	*/
l	223750322	108654301	527228919	252958081	123286868	/*	00050-00054	*/
l	583686525	289909233	141006858	659465480	313364343	/*	00055-00059	*/
l	123607255	742830237	229709271	111078387	536533733	/*	00060-00064	*/
l	258784006	124958353	603630600	289918376	140716211	/*	00065-00069	*/
l	675222757	328287292	159730021	769242244	374923574	/*	00070-00074	*/
l	180827427	871783575	404247254	186370643	978871159	/*	00075-00079	*/
l	473535419	228136434	110787484	537504531	257873751	/*	00080-00084	*/
l	125681946	595000876	295630132	143787634	671865060	/*	00085-00089	*/
l	319061760	125753795	759491332			/*	00090-00092	*/
l	1					/*	tbl_sca_sz	*/
l	1					/*	tbl_ele_sz	*/
b	0					/*	tbl_type	*/
s	4					/*	num comnts	*/
m	4 1					/*	tbl_desc	*/

```
t     This factor contains the mass dependency in computing          /*  00000  */
t     distribution (needed since we make computation using the       /*  00001  */
t     particle energy and not velocity) and also the necessary       /*  00002  */
t     scaling to put units in s**3/km***6                            /*  00003  */
b     2                                                              /*  tbl_var  */
b     1                                                              /*  tbl_expand  */
l     0                                                              /*  crit_act_sz  */
n                                                                    /*  crit_status  */
n                                                                    /*  crit_sen_off  */
n                                                                    /*  crit_offs  */
m     3  3                                                          /*  tbl_fmt  */
b           1       1       1                                      /*  00000-  */
                                                                     /*  00002  */
m     3  3                                                          /*  tbl_off  */
l           0       0       0                                      /*  00000-  */
                                                                     /*  00002  */
m     1  1                                                          /*  tbl_sca  */
b          -31                                                     /*  00000  */
m     1  1                                                          /*  tbl  */
l          4149605                                                 /*  00000  */
l     -3                                                           /*  tbl_sca_sz  */
l      93                                                          /*  tbl_ele_sz  */
b     2                                                            /*  tbl_type  */
s     8                                                            /*  num_comnts  */
m     8   1                                                        /*  tbl_desc  */
t     Table 8                                                      /*  00000  */
t     Calibration table of the threshold 1 count level.            /*  00001  */
t     In order to generate the threshold level, it is faked        /*  00002  */
t     in the IDFS world by signifying it dependent on a cal        /*  00003  */
t     field.  Threshold count levels are not dependent on          /*  00004  */
t     cal values and represent a count of 1 (one) at each          /*  00005  */
t     sensor energy level.  Zero value means that that energy      /*  00006  */
t     value has not been defined.                                  /*  00007  */
b     -1                                                           /*  tbl_var  */
b     0                                                            /*  tbl_expand  */
l     0                                                            /*  crit_act_sz  */
n                                                                  /*  crit_status  */
n                                                                  /*  crit_sen_off  */
n                                                                  /*  crit_offs  */
m     3  3                                                        /*  tbl_fmt  */
b           1       1       1                                    /*  00000-  */
                                                                   /*  00004  */
m     3  3                                                        /*  tbl_off  */
l           0       31      62                                   /*  00000-  */
                                                                   /*  00004  */
m     3  3                                                        /*  tbl_sca  */
```

```
b           0       0       0                                                   /*  00000-      */
                                                                                    00004
m     93  8                                                                     /*  tbl         */
l           1   1   1   1   1   1   1   1                                       /*  00000-      */
                                                                                    00007
l           1   1   1   1   1   1   1   1                                       /*  00008-      */
                                                                                    00015
l           1   1   1   1   1   1   1   1                                       /*  00016-      */
                                                                                    00023
l           1   1   1   1   1   1   1   1                                       /*  00024-      */
                                                                                    00031
l           1   1   1   1   1   1   1   1                                       /*  00032-      */
                                                                                    00039
l           1   1   1   1   1   1   1   1                                       /*  00040-      */
                                                                                    00047
l           1   1   1   1   1   1   1   1                                       /*  00048-      */
                                                                                    00055
l           1   1   1   1   1   1   1   1                                       /*  00056-      */
                                                                                    00063
l           1   1   1   1   1   1   1   1                                       /*  00064-      */
                                                                                    00071
l           1   1   1   1   1   1   1   1                                       /*  00072-      */
                                                                                    00079
l           1   1   1   1   1   1   1   1                                       /*  00080-      */
                                                                                    00087
l           1   1   1   1   1                                                   /*  00088-      */
                                                                                    00092
l     -3                                                                        /*  tbl_sca_sz  */
l     93                                                                        /*  tbl_ele_sz  */
b     2                                                                         /*  tbl_type    */
s     9                                                                         /*  num_comnts  */
m     9   1                                                                     /*  tbl_desc    */
t     Table 9                                                                   /*  00000       */
t     Dark count as determined by Gallileo Electro-Optics.                      /*  00001       */
t     In order to generate the dark count level, it is faked                    /*  00002       */
t     in the IDFS world by signifying it dependent on a cal                     /*  00003       */
t     field.  Dark count levels are not dependent on cal values                 /*  00004       */
t     and represent a count at each sensor energy level.                        /*  00005       */
t     Zero value means that that energy value has not been                      /*  00006       */
t     defined. Dark counts represent the average counts that the                /*  00007       */
t     CEM sensor will generate by spontaneous emission.                         /*  00008       */
b     -1                                                                        /*  tbl_var     */
b     0                                                                         /*  tbl_expand  */
l     0                                                                         /*  crit_act_sz */
n                                                                               /*  crit_status */
n                                                                               /*  crit_sen_off */
```

```
n                                                                   /*  crit_offs       */
m    3  3                                                          /*  tbl_fmt         */
b          1       1       1                                       /*  00000-          */
                                                                        00004
m    3  3                                                          /*  tbl_off         */
l          0      31      62                                       /*  00000-          */
                                                                        00004
m    3  3                                                          /*  tbl_sca         */
b         -6      -6      -6                                       /*  00000-          */
                                                                        00004
m    93  8                                                         /*  tbl             */
l         957  957  957  957  957  957  957  957                   /*  00000-          */
                                                                        00007
l         957  957  957  957  957  957  957  957                   /*  00008-          */
                                                                        00015
l         957  957  957  957  957  957  957  957                   /*  00016-          */
                                                                        00023
l         957  957  957  957  957  957  957  957                   /*  00024-          */
                                                                        00031
l         957  957  957  957  957  957  957  957                   /*  00032-          */
                                                                        00039
l         957  957  957  957  957  957  957  957                   /*  00040-          */
                                                                        00047
l         957  957  957  957  957  957  957  957                   /*  00048-          */
                                                                        00055
l         957  957  957  957  957  957 11489 11489                 /*  00056-          */
                                                                        00063
l       11489 11489 11489 11489 11489 11489 11489 11489           /*  00064-          */
                                                                        00071
l       11489 11489 11489 11489 11489 11489 11489 11489           /*  00072-          */
                                                                        00079
l       11489 11489 11489 11489 11489 11489 11489 11489           /*  00080-          */
                                                                        00087
l       11489 11489 11489 11489 11489                             /*  00088-          */
                                                                        00095
l    -3                                                            /*  tbl_sca_sz      */
l    256                                                           /*  tbl_ele_sz      */
b    0                                                             /*  tbl_type        */
s    4                                                             /*  num_comnts      */
m    4  1                                                          /*  tbl_desc        */
t    Table 10                                                      /*  00000           */
t    Amplifier Dead Time Correction Factor.  Multiply this         /*  00001           */
t    number by count rate table value to correct the count for     /*  00002           */
t    amplifier dead time.                                          /*  00003           */
b    0                                                             /*  tbl_var         */
b    1                                                             /*  tbl_expand      */
```

```
l     0                                                                 /*  crit_act_sz     */
n                                                                       /*  crit_status     */
n                                                                       /*  crit_sen_off    */
n                                                                       /*  crit_offs       */
m     3  3                                                              /*  tbl_fmt         */
b           0       0       0                                           /*  00000-
                                                                            00004           */
m     3  3                                                              /*  tbl_off         */
l           0       0       0                                           /*  00000-
                                                                            00004           */
m     3  3                                                              /*  tbl_sca         */
b          -6      -6      -6                                           /*  00000-
                                                                            00004           */
m     256  6                                                            /*  tbl             */
l       1000000 1000004 1000009 1000013 1000017 1000022                 /*  00000-
                                                                            00005           */
l       1000026 1000030 1000035 1000039 1000044 1000048                 /*  00006-
                                                                            00011           */
l       1000052 1000057 1000061 1000065 1000072 1000076                 /*  00012-
                                                                            00017           */
l       1000081 1000085 1000089 1000094 1000098 1000102                 /*  00018-
                                                                            00023           */
l       1000107 1000111 1000115 1000120 1000124 1000128                 /*  00024-
                                                                            00029           */
l       1000133 1000137 1000144 1000152 1000161 1000170                 /*  00030-
                                                                            00035           */
l       1000178 1000187 1000196 1000205 1000213 1000222                 /*  00036-
                                                                            00041           */
l       1000231 1000239 1000248 1000257 1000266 1000274                 /*  00042-
                                                                            00047           */
l       1000287 1000305 1000322 1000340 1000357 1000374                 /*  00048-
                                                                            00053           */
l       1000392 1000409 1000427 1000444 1000462 1000479                 /*  00054-
                                                                            00059           */
l       1000496 1000514 1000531 1000549 1000575 1000610                 /*  00060-
                                                                            00065           */
l       1000645 1000680 1000714 1000749 1000784 1000819                 /*  00066-
                                                                            00071           */
l       1000854 1000889 1000924 1000959 1000994 1001029                 /*  00072-
                                                                            00077           */
l       1001064 1001098 1001151 1001221 1001291 1001361                 /*  00078-
                                                                            00083           */
l       1001430 1001500 1001570 1001640 1001710 1001780                 /*  00084-
                                                                            00089           */
l       1001850 1001920 1001990 1002060 1002131 1002201                 /*  00090-
                                                                            00095           */
```

```
l     1002306 1002446 1002586 1002727 1002867 1003008   /* 00096-00101 */
l     1003148 1003289 1003429 1003570 1003711 1003852   /* 00102-00107 */
l     1003993 1004134 1004275 1004416 1004628 1004910   /* 00108-00113 */
l     1005193 1005476 1005759 1006043 1006326 1006610   /* 00114-00119 */
l     1006895 1007179 1007464 1007749 1008034 1008319   /* 00120-00125 */
l     1008605 1008891 1009320 1009893 1010468 1011043   /* 00126-00131 */
l     1011619 1012196 1012775 1013354 1013934 1014515   /* 00132-00137 */
l     1015097 1015681 1016265 1016850 1017436 1018024   /* 00138-00143 */
l     1018907 1020087 1021272 1022461 1023655 1024852   /* 00144-00149 */
l     1026054 1027260 1028470 1029685 1030904 1032127   /* 00150-00155 */
l     1033355 1034587 1035824 1037065 1038935 1041445   /* 00156-00161 */
l     1043973 1046520 1049087 1051672 1054277 1056902   /* 00162-00167 */
l     1059547 1062212 1064898 1067605 1070333 1073082   /* 00168-00163 */
l     1075854 1078647 1082880 1088604 1094423 1100339   /* 00174-00179 */
l     1106356 1112476 1118703 1125041 1131492 1138061   /* 00180-00185 */
l     1144752 1151568 1158514 1165596 1172816 1180182   /* 00186-00191 */
l     1191513 1207180 1223533 1240630 1258534 1277321   /* 00192-00197 */
l     1297073 1317887 1339877 1363171 1387925 1414321   /* 00198-00203 */
l     1442580 1472970 1505823 1541557 1601779 1698837   /* 00204-00209 */
l     1826153 2013567 2431803       0       0       0   /* 00210-00215 */
l           0       0       0       0       0       0   /* 00216-00221 */
l           0       0       0       0       0       0   /* 00222-00227 */
l           0       0       0       0       0       0   /* 00228-00233 */
```

```
l          0       0       0       0       0       0                    /*  00234-      */
                                                                            00239
l          0       0       0       0       0       0                    /*  00240-      */
                                                                            00245
l          0       0       0       0       0       0                    /*  00246-      */
                                                                            00251
l          0       0       0       0                                    /*  00252-      */
                                                                            00255
l    0                                                                  /*  tbl_sca_sz  */
l    6                                                                  /*  tbl_ele_sz  */
b    1                                                                  /*  tbl_type    */
s    1                                                                  /*  num comnts  */
m    1   1                                                              /*  tbl_desc    */
t    ASCII definitions of the status states                             /*  00000       */
b    4                                                                  /*  tbl_var     */
b    0                                                                  /*  tbl_expand  */
l    0                                                                  /*  crit_act_sz */
n                                                                       /*  crit_status */
n                                                                       /*  crit_sen_off*/
n                                                                       /*  crit_offs   */
m    2   2                                                              /*  tbl_fmt     */
b          0       0                                                    /*  00000-      */
                                                                            00002
m    2   2                                                              /*  tbl_off     */
l          0       4                                                    /*  00000-      */
                                                                            00002
n                                                                       /*  tbl_sca     */
m    6   4                                                              /*  tbl         */
T    "N along -X"   "N along +X"   "S along -X"  "S along +X"           /*  00000-      */
                                                                            00003
T          "low"       "high"                                           /*  00004-      */
                                                                            00005
b    6                                                                  /*  const_id    */
s    3                                                                  /*  num_comnts  */
m    3   1                                                              /*  const_desc  */
t    Constant 0                                                         /*  000000      */
t    Axis A component of aperture normals with respect to the PEM       /*  000001      */
t    magnetometer (VMAG).                                               /*  000002      */
m    3   3                                                              /*  const_sca   */
b          -6      -6      -6                                           /*  00000-      */
                                                                            00002
m    3   3                                                              /*  const       */
l        -22690    -17188      5581                                     /*  00000-      */
                                                                            00002
b    7                                                                  /*  const_id    */
s    3                                                                  /*  num_comnts  */
```

```
m    3  1                                                          /*  const_desc   */
t    Constant 1                                                   /*  000000       */
t    Axis B component of aperture normals with respect to the PEM /*  000001       */
t    magnetometer (VMAG).                                         /*  000002       */
m    3  3                                                         /*  const_sca    */
b         -6       -6       -6                                    /*  00000-       */
                                                                   /*  00002        */
m    3  3                                                         /*  const        */
l       804688    398922   -366949                                /*  00000-       */
                                                                   /*  00002        */
b    8                                                            /*  const_id     */
s    3                                                            /*  num_comnts   */
m    3  1                                                         /*  const_desc   */
t    Constant 2                                                   /*  000000       */
t    Axis C component of aperture normals with respect to the PEM /*  000001       */
t    magnetometer (VMAG).                                         /*  000002       */
m    3  3                                                         /*  const_sca    */
b         -6       -6       -6                                    /*  00000-       */
                                                                   /*  00002        */
m    3  3                                                         /*  const        */
l       593265    916824    930224                                /*  00000-       */
                                                                   /*  00002        */
```

8. TOKEN-TAGGED VIDF EXAMPLE

The example provided below is the same example that is provided in section 7 but defined in the token-tagged VIDF format.

```
vidf MPSC {
    float version = 3.0;
    string mission = "Upper Atmospheric Research Satellite";
    string spacecraft = "UARS
    string experiment = "Medium Energy Electron and Ion Measurements";
    string instrument = "Nadir Pointing Electron Sensors";
    string contact = "Dr. J. David Winningham";
    string contact = "Southwest Research Institute";
    string contact = "6220 Culebra Road";
    string contact = "San Antonio, Texas 78238";
    string contact = "david@cluster.space.swri.edu";
/*
 *  The following is a list of tables which are in this vidf
 *    TABLE 0:  center energies (eV)
 *    TABLE 1:  telemetry decom table
 *    TABLE 2:  detector efficiencies
 *    TABLE 3:  geometry factors (cm**2-str)
 *    TABLE 4:  dE/E
 *    TABLE 5:  center energies (ergs)
 *    TABLE 6:  (center energies)**2 (ergs**2)
 *    TABLE 7:  constant needed in going to dist. fn
 *    TABLE 8:  Threshold count level
 *    TABLE 9:  Dark count level
 *    TABLE 10: Amplifier Dead Time Factor
 *    TABLE 11: ASCII descriptions of status states
 *
 *  The following are units which can be derived from the tables.
 *  The format is to give the tables applied followed by the
 *  operations and unit definition
 *
 *  DATA TYPE  TABLES          OPERS            UNIT
 *    Scan      0              0                eV
 *    Sensor    1              0                cnts/accum
 *    Sensor    1,2            0,4              cnts/accum (eff. cor)
 *    Sensor    1,2            0,154            cnts/sec
 *    Sensor    1,2,3,4        0,154,4,4        cnts/(cm**2-str-s)
 *    Sensor    1,2,3,4,0      0,154,4,4,4      cnts/(cm**2-str-s-eV)
 *    Sensor    1,2,3,4,0,5    0,154,4,4,4,3    ergs/(cm**2-str-s-eV)
 *    Sensor    1,2,3,4,7,6    0,154,4,4,3,4    sec**3/km**6
 *
 *  The following is a list of constant values which are in this vidf
 *    CONST 0:  These are the Axis A components of each sensor's
 *              aperture expressed relative to the PEM magnetometer.
 *    CONST 1:  These are the Axis B components of each sensor's
 *              aperture expressed relative to the PEM magnetometer.
 *    CONST 2:  These are the Axis C components of each sensor's
 *              aperture expressed relative to the PEM magnetometer.
 */
   int s_year = 1980;                                    /* ds_year      */
   int s_day = 1;                                        /* ds_day       */
```

```
    int s_msec = 0;                                         /* ds_msec      */
    int s_usec = 0;                                         /* ds_usec      */
    int e_year = 2020;                                      /* de_year      */
    int e_day = 1;                                          /* de_day       */
    int e_msec = 0;                                         /* de_msec      */
    int e_usec = 0;                                         /* de_usec      */
    int smp_id = 1;                                         /* smp_id       */
    int sen_mode = 2;                                       /* sen_mode     */
    int n_qual = 4;                                         /* n_qual       */
    int n_cal_sets = 2;                                     /* cal_sets     */
    int n_tbls = 12;                                        /* num_tbls     */
    int n_consts = 3;                                       /* num_consts   */
    int n_status = 2;                                       /* status       */
    int n_sensors = 3;                                      /* sen          */
    int swp_len = 31;                                       /* swp_len      */
    int max_nss = 48;                                       /* max_nss      */
    int data_len = 3952;                                    /* data_len     */
    int fill_flag = 0;                                      /* fill_flg     */
    int da_method = 0;                                      /* da_method    */
    struct Status0 {
        string name = "Satellite Aspect";                   /* name         */
        int state = 4;                                      /* state        */
    };
    struct Status1 {
        string name = "HVPS3 State";                        /* name         */
        int state = 2;                                      /* state        */
    };
    string qual_names = "No Fill Data";                     /* name         */
    string qual_names = "Fill Data With Energy Sweep";      /* name         */
    string qual_names = "Possible Fill Data With Energy Sweep"; /* name     */
    string qual_names = "Fill And Possible Fill Data With Energy Sweep";
    struct PitchAngle {
        int format = 1;                                     /* pa_format    */
        string project = "UARS";                            /* pa_project   */
        string mission = "UARS-1";                          /* pa_mission   */
        string experiment = "PEM";                          /* pa_exper     */
        string instrument = "VMAG";                         /* pa_inst      */
        string vinstrument = "VMMA";                        /* pa_vinst     */
        int b1 = 0;                                         /* b1           */
        int b2 = 1;                                         /* b2           */
        int b3 = 2;                                         /* b3           */
        int num_tbls = 1;                                   /* num_tbls     */
        int tbls = 1;                                       /* tbl 0        */
        int opers = 0;                                      /* oper 0       */
    };
    struct Sensor0 {
        string name = "ESensor 10: 126.3 degrees";          /* name         */
        int d_type = 0;                                     /* d_type       */
        int status = 1;                                     /* status       */
        int tdw_len = 8;                                    /* tdw_len      */
        int time_offset = 0;                                /* time_offset  */
    };
    struct Sensor1 {
        string name = "ESensor 12: 156.3 degrees";          /* name         */
        int d_type = 0;                                     /* d_type       */
        int status = 1;                                     /* status       */
        int tdw_len = 8;                                    /* tdw_len      */
```

```
        int time_offset = 0;                                /* time_offset   */
    };
    struct Sensor2 {
        string name = "ESensor 14: -158.7 degrees";         /* name          */
        int d_type = 0;                                     /* d_type        */
        int status = 1;                                     /* status        */
        int tdw_len = 8;                                    /* tdw_len       */
        int time_offset = 0;                                /* time_offset   */
    };
    struct CalSet0 {
        string name = "Fill Flags Per Energy Step";         /* name          */
        int use = 8;                                        /* use           */
        int word_len = 8;                                   /* word length   */
        int target = 1;                                     /* target        */
    };
    struct CalSet1 {
        string name = "CRC Flags Per Energy Step";          /* name          */
        int use = 8;                                        /* use           */
        int word_len = 8;                                   /* word length   */
        int target = 1;                                     /* target        */
    };
    struct Table0 {
        int tbl_sca_sz = -3;                                /* tbl_sca_sz    */
        int tbl_ele_sz = 93;                                /* tbl_ele_sz    */
        int tbl_type = 0;                                   /* tbl_type      */
/*
 *  This table contains the center energies in eV
 */
        int tbl_var = 2;                                    /* tbl_var       */
        int tbl_expand = 1;                                 /* tbl_expand    */
        int crit_act_sz = 0;                                /* crit_act_ele  */
        int format [3] = {0, 0, 0};                         /* format        */
        int offset [3] = {0, 31, 62};                       /* offsets       */
        int scale [3] = {-3, -3, -3};                       /* scale factor  */
        int values [93] = {                                 /* values        */
          28776058, 20010449, 13907228, 9658524, 6711579, 4664744,
          3232809, 2252238, 1560147, 1087850, 758814, 526591,
          367631, 255315, 177274, 120717, 81964, 59402,
          41316, 28677, 19983, 13920, 9642, 6730,
          4631, 3263, 2276, 1557, 1072, 674,
          523, 29629464, 20603894, 14319672, 9944965, 6910623,
          4803084, 3328684, 2319031, 1606416, 1120114, 781317,
          542208, 378535, 262886, 182531, 124296, 84396,
          61165, 42541, 29527, 20576, 14333, 9928,
          6931, 4769, 3361, 2344, 1603, 1105,
          694, 538, 29917599, 20804259, 14458925, 10041676,
          6977826, 4849792, 3361053, 2341583, 1622038, 1131005,
          788916, 547481, 382216, 265442, 184307, 125505,
          85217, 61759, 42955, 29815, 20777, 14472,
          10024, 6998, 4815, 3394, 2367, 1618,
          1115, 700, 544
        };
    };
    struct Table1 {
        int tbl_sca_sz = -3;                                /* tbl_sca_sz   */
        int tbl_ele_sz = 256;                               /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
```

```
/*
 *  This table contains the decompress from telemetry to
 *  counts per acumulation period
 */
       int tbl_var = 0;                                        /* tbl_var      */
       int tbl_expand = 1;                                     /* tbl_expand   */
       int crit_act_sz = 0;                                    /* crit_act_ele */
       int format [3] = {0, 0, 0};                             /* format       */
       int offset [3] = {0, 0, 0};                             /* offsets      */
       int scale [3] = {-1, -1, -1};                           /* scale factor */
       int values [256] = {                                    /* values       */
         0, 10, 20, 30, 40, 50,                                /*  000 - 005   */
         60, 70, 80, 90, 100, 110,                             /*  006 - 011   */
         120, 130, 140, 150, 165, 175,                         /*  012 - 017   */
         185, 195, 205, 215, 225, 235,                         /*  018 - 023   */
         245, 255, 265, 275, 285, 295,                         /*  024 - 029   */
         305, 315, 330, 350, 370, 390,                         /*  030 - 035   */
         410, 430, 450, 470, 490, 510,                         /*  036 - 041   */
         530, 550, 570, 590, 610, 630,                         /*  042 - 047   */
         660, 700, 740, 780, 820, 860,                         /*  048 - 053   */
         900, 940, 980, 1020, 1060, 1100,                      /*  054 - 059   */
         1140, 1180, 1220, 1260, 1320, 1400,                   /*  060 - 065   */
         1480, 1560, 1640, 1720, 1800, 1880,                   /*  066 - 071   */
         1960, 2040, 2120, 2200, 2280, 2360,                   /*  072 - 077   */
         2440, 2520, 2640, 2800, 2960, 3120,                   /*  078 - 083   */
         3280, 3440, 3600, 3760, 3920, 4080,                   /*  084 - 089   */
         4240, 4400, 4560, 4720, 4880, 5040,                   /*  090 - 095   */
         5280, 5600, 5920, 6240, 6560, 6880,                   /*  096 - 101   */
         7200, 7520, 7840, 8160, 8480, 8800,                   /*  102 - 107   */
         9120, 9440, 9760, 10080, 10560, 11200,                /*  108 - 113   */
         11840, 12480, 13120, 13760, 14400, 15040,             /*  114 - 119   */
         15680, 16320, 16960, 17600, 18240, 18880,             /*  120 - 125   */
         19520, 20160, 21120, 22400, 23680, 24960,             /*  126 - 131   */
         26240, 27520, 28800, 30080, 31360, 32640,             /*  132 - 137   */
         33920, 35200, 36480, 37760, 39040, 40320,             /*  138 - 143   */
         42240, 44800, 47360, 49920, 52480, 55040,             /*  144 - 149   */
         57600, 60160, 62720, 65280, 67840, 70400,             /*  150 - 155   */
         72960, 75520, 78080, 80640, 84480, 89600,             /*  156 - 161   */
         94720, 99840, 104960, 110080, 115200, 120320,         /*  162 - 167   */
         125440, 130560, 135680, 140800, 145920, 151040,       /*  168 - 173   */
         156160, 161280, 168960, 179200, 189440, 199680,       /*  174 - 179   */
         209920, 220160, 230400, 240640, 250880, 261120,       /*  180 - 185   */
         271360, 281600, 291840, 302080, 312320, 322560,       /*  186 - 191   */
         337920, 358400, 378880, 399360, 419840, 440320,       /*  192 – 197   */
         460800, 481280, 501760, 522240, 542720, 563200,       /*  198 – 203   */
         583680, 604160, 624640, 645120, 675840, 716800,       /*  204 - 209   */
         757760, 798720, 839680, 880640, 921600, 962560,       /*  210 - 215   */
         1003520, 1044480, 1085440, 1126400, 1167360, 1208320, /* 216-221 */
         1249280, 1290240, 1351680, 1433600, 1515520, 1597440, /* 222-227 */
         1679360, 1761280, 1843200, 1925120, 2007040, 2088960, /* 228-233 */
         2170880, 2252800, 2334720, 2416640, 2498560, 2580480, /* 234-239 */
         2703360, 2867200, 3031040, 3194880, 3358720, 3522560, /* 240-245 */
         3686400, 3850240, 4014080, 4177920, 4341760, 4505600, /* 246-251 */
         4669440, 4833280, 4997120, 5160960                    /* 252-255 */
       };
    };
```

```
    struct Table2 {
        int tbl_sca_sz = -3;                                /* tbl_sca_sz   */
        int tbl_ele_sz = 186;                               /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
/*
 *  This table contains channeltron efficiency as a function
 *  of energy step.  The table is repeated for each of the
 *  three detectors and then again since the efficiencies vary
 *  depending on the applied bias voltage applied.
 */
        int tbl_var = 2;                                    /* tbl_var      */
        int tbl_expand = 1;                                 /* tbl_expand   */
        int crit_act_sz = 6;                                /* crit_act_ele */
        struct CriticalAction {
            int status [3] = {1, 1, 1};                     /* status       */
            int offset [3] = {0, 2, 4};                     /* offset       */
            int table [6] = {                               /* table        */
          0, 93, 31, 124, 62, 155                           /*  000 - 005   */
            };
        };
        int format [3] = {0, 0, 0};                         /* format       */
        int offset [3] = {0, 0, 0};                         /* offsets      */
        int scale [3] = {-6, -6, -6};                       /* scale factor */
        int values [186] = {                                /* values       */
          264919, 340689, 442820, 531910, 605260, 671389,   /*  000 - 005   */
          730780, 782180, 827130, 864849, 894100, 917280,   /*  006 - 011   */
          933659, 944999, 951960, 955820, 957629, 958090,   /*  012 - 017   */
          958050, 957740, 957360, 957009, 956709, 956480,   /*  018 - 023   */
          956300, 956179, 956080, 956019, 955969, 955929,   /*  024 - 029   */
          955910, 263729, 332670, 435290, 525690, 599629,   /*  030 - 035   */
          666339, 726310, 778280, 823819, 861419, 891929,   /*  036 - 041   */
          915690, 932529, 944270, 951529, 955609, 957560,   /*  042 - 047   */
          958079, 958069, 957769, 957390, 957040, 956730,   /*  048 - 053   */
          956499, 956309, 956189, 956089, 956019, 955969,   /*  054 - 059   */
          955929, 955910, 263619, 330029, 432819, 523620,   /*  060 - 065   */
          597760, 664659, 724829, 776989, 822709, 860499,   /*  066 - 071   */
          891200, 915149, 932150, 944019, 951390, 955540,   /*  072 - 077   */
          957530, 958069, 958069, 957780, 957400, 957050,   /*  078 - 083   */
          956740, 956499, 956319, 956189, 956089, 956019,   /*  084 - 089   */
          955969, 955929, 955919, 520107, 565243, 611256,   /*  090 - 095   */
          657460, 702879, 746668, 788259, 825896, 859978,   /*  096 - 101   */
          888868, 912930, 932294, 946394, 956121, 962158,   /*  102 - 107   */
          965722, 967494, 968152, 968415, 968423, 968325,   /*  108 - 113   */
          968203, 968086, 967990, 967913, 967859, 967818,   /*  114 - 119   */
          967788, 967768, 967750, 967742, 514510, 559463,   /*  120 - 125   */
          605354, 651510, 696959, 740858, 782639, 820528,   /*  126 - 131   */
          854918, 884146, 908569, 928354, 942942, 953207,   /*  132 - 137   */
          959770, 963828, 966024, 966980, 967521, 967748,   /*  138 - 143   */
          967823, 967827, 967806, 967779, 967754, 967733,   /*  144 - 149   */
          967718, 967705, 967696, 967688, 967685, 514685,   /*  150 - 155   */
          559668, 605598, 651808, 697324, 741312, 783203,   /*  156 - 161   */
          821220, 855759, 885142, 909718, 929630, 944285,   /*  162 - 167   */
          954533, 961004, 964914, 966938, 967751, 968141,   /*  168 - 173   */
          968240, 968209, 968132, 968047, 967975, 967916,   /*  174 - 179   */
          967872, 967840, 967814, 967797, 967782, 967778    /*  180 - 185   */
        };
    };
```

```
    struct Table3 {
        int tbl_sca_sz = 3;                                 /* tbl_sca_sz   */
        int tbl_ele_sz = 3;                                 /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
/*
 *  This table contains the detector geometry factors as
 *  (cm**2-s)                                 `
 */
        int tbl_var = 2;                                    /* tbl_var      */
        int tbl_expand = 1;                                 /* tbl_expand   */
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [3] = {1, 1, 1};                         /* format       */
        int offset [3] = {0, 1, 2};                         /* offsets      */
        int scale [3] = {-8, -8, -8};                       /* scale factor */
        int values [3] = {20551, 11009, 18882};             /* values       */
    };
    struct Table4 {
        int tbl_sca_sz = -3;                                /* tbl_sca_sz   */
        int tbl_ele_sz = 93;                                /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
/*
 *  This table contains the energy resolution (dE/E)
 */
        int tbl_var = 2;                                    /* tbl_var      */
        int tbl_expand = 1;                                 /* tbl_expand   */
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [3] = {0, 0, 0};                         /* format       */
        int offset [3] = {0, 31, 62};                       /* offsets      */
        int scale [3] = {-3, -3, -3};                       /* scale factor */
        int values [93] = {                                 /* values       */
          344, 344, 344, 344, 344, 344,                     /*  000 - 005   */
          344, 344, 344, 344, 344, 344,                     /*  006 - 011   */
          344, 344, 344, 344, 344, 344,                     /*  012 - 017   */
          344, 344, 344, 344, 344, 344,                     /*  018 - 023   */
          344, 344, 344, 344, 344, 344,                     /*  024 - 029   */
          344, 355, 355, 355, 355, 355,                     /*  030 - 035   */
          355, 355, 355, 355, 355, 355,                     /*  036 - 041   */
          355, 355, 355, 355, 355, 355,                     /*  042 - 047   */
          355, 355, 355, 355, 355, 355,                     /*  048 - 053   */
          355, 355, 355, 355, 355, 355,                     /*  054 - 059   */
          355, 355, 349, 349, 349, 349,                     /*  060 - 065   */
          349, 349, 349, 349, 349, 349,                     /*  066 - 071   */
          349, 349, 349, 349, 349, 349,                     /*  072 - 077   */
          349, 349, 349, 349, 349, 349,                     /*  078 - 083   */
          349, 349, 349, 349, 349, 349,                     /*  084 - 089   */
          349, 349, 349                                     /*  090 - 092   */
        };
    };
    struct Table5 {
        int tbl_sca_sz = -3;                                /* tbl_sca_sz   */
        int tbl_ele_sz = 93;                                /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
/*
 *  This table contains the center energies in ergs
 */
        int tbl_var = 2;                                    /* tbl_var      */
        int tbl_expand = 1;                                 /* tbl_expand   */
```

```
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [3] = {0, 0, 0};                         /* format       */
        int offset [3] = {0, 31, 62};                       /* offsets      */
        int scale [3] = {-16, -16, -16};                    /* scale factor */
        int values [93] = {                                 /* values       */
          460992448, 320567376, 222793792, 154729554, 107519495, 74729198,
          51789600, 36080852, 24993554, 17427356, 12156200, 8435987,
          5889448, 4090146, 2839929, 1933886, 1313063, 951620,
          661882, 459405, 320127, 222998, 154464, 107814,
          74188, 52273, 36461, 24943, 17173, 10797,
          8378, 474664012, 330074381, 229401145, 159318339, 110708180,
          76945405, 53325517, 37150876, 25734784, 17944226, 12516698,
          8686172, 6064130, 4211433, 2924146, 1991221, 1352023,
          979863, 681506, 473022, 329627, 229614, 159046,
          111034, 76399, 53843, 37550, 25680, 17702,
          11117, 8618, 479279951, 333284244, 231631978, 160867649,
          111784772, 77693667, 53844068, 37512159, 25985048, 18118700,
          12638434, 8770645, 6123100, 4252380, 2952598, 2010590,
          1365176, 989379, 688139, 477636, 332847, 231841,
          160584, 112107, 77136, 54371, 37919, 25920,
          17862, 11213, 8714
        };
    };
    struct Table6 {
        int tbl_sca_sz = 93;                                /* tbl_sca_sz   */
        int tbl_ele_sz = 93;                                /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
/*
 *  This table contains the square of the center energies
 *  in ergs**2
 */
        int tbl_var = 2;                                    /* tbl_var      */
        int tbl_expand = 1;                                 /* tbl_expand   */
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [3] = {0, 0, 0};                         /* format       */
        int offset [3] = {0, 31, 62};                       /* offsets      */
        int scale [93] = {                                  /* scale factor */
          -23, -23, -24, -24, -24, -25,                     /*  000 - 005   */
          -25, -25, -26, -26, -26, -27,                     /*  006 - 011   */
          -27, -27, -28, -28, -28, -29,                     /*  012 - 017   */
          -29, -29, -29, -30, -30, -30,                     /*  018 - 023   */
          -31, -31, -31, -32, -32, -32,                     /*  024 - 029   */
          -33, -23, -23, -24, -24, -24,                     /*  030 - 035   */
          -25, -25, -25, -26, -26, -26,                     /*  036 - 041   */
          -27, -27, -27, -28, -28, -28,                     /*  042 - 047   */
          -29, -29, -29, -29, -30, -30,                     /*  048 - 053   */
          -30, -31, -31, -31, -32, -32,                     /*  054 - 059   */
          -32, -33, -23, -23, -24, -24,                     /*  060 - 065   */
          -24, -25, -25, -25, -26, -26,                     /*  066 - 071   */
          -26, -27, -27, -27, -28, -28,                     /*  072 - 077   */
          -28, -29, -29, -29, -29, -30,                     /*  078 - 083   */
          -30, -30, -31, -31, -31, -32,                     /*  084 - 089   */
          -32, -32, -33                                     /*  090 - 092   */
        };
        int values [93] = {                                 /* values       */
          212514037, 102763442, 496370738, 239412349, 115604419, 558445315,
          268216268, 130182793, 624677787, 303712771, 147773204, 711658903,
```

```
          346856049, 167292967, 806519943, 373991636, 172413517, 905580698,
          438088204, 211053449, 102481718, 497282862, 238593867, 116239879,
          550395132, 273249370, 132944243, 622160231, 294927040, 116585574,
          701985918, 225305924, 108949097, 526248854, 253823331, 122563011,
          592059544, 284361082, 138018763, 662279122, 321995256, 156667736,
          754495866, 367736810, 177361739, 855063343, 396496472, 182796867,
          960132084, 464451544, 223750322, 108654301, 527228919, 252958081,
          123286868, 583686525, 289909233, 141006858, 659465480, 313364343,
          123607255, 742830237, 229709271, 111078387, 536533733, 258784006,
          124958353, 603630600, 289918376, 140716211, 675222757, 328287292,
          159730021, 769242244, 374923574, 180827427, 871783575, 404247254,
          186370643, 978871159, 473535419, 228136434, 110787484, 537504531,
          257873751, 125681946, 595000876, 295630132, 143787634, 671865060,
          319061760, 125753795, 759491332
        };
    };
    struct Table7 {
        int tbl_sca_sz = 1;                                 /* tbl_sca_sz   */
        int tbl_ele_sz = 1;                                 /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
/*
 *  This factor contains the mass dependency in computing
 *  distribution (needed since we make computation using the
 *  particle energy and not velocity) and also the necessary
 *  scaling to put units in s**3/km***6
 */
        int tbl_var = 2;                                    /* tbl_var      */
        int tbl_expand = 1;                                 /* tbl_expand   */
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [3] = {1, 1, 1};                         /* format       */
        int offset [3] = {0, 0, 0};                         /* offsets      */
        int scale [1] = {-31};                              /* scale factor */
        int values [1] = {4149605};                         /* values       */
    };
    struct Table8 {
        int tbl_sca_sz = -3;                                /* tbl_sca_sz   */
        int tbl_ele_sz = 93;                                /* tbl_ele_sz   */
        int tbl_type = 2;                                   /* tbl_type     */
/*
 *  Table 8
 *  Calibration table of the threshold 1 count level.
 *  In order to generate the threshold level, it is faked
 *  in the IDF world by signifing it dependent on a cal
 *  field.  Threshold count levels are not dependent on
 *  cal values and represent a count of 1 (one) at each
 *  sensor energy level.  Zero value means that that energy
 *  value has not been defined.
 */
        int tbl_var = -1;                                   /* tbl_var      */
        int tbl_expand = 0;                                 /* tbl_expand   */
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [3] = {1, 1, 1};                         /* format       */
        int offset [3] = {0, 31, 62};                       /* offsets      */
        int scale [3] = {0, 0, 0};                          /* scale factor */
        int values [93] = {                                 /* values       */
          1, 1, 1, 1, 1, 1,                                 /*  000 - 005   */
          1, 1, 1, 1, 1, 1,                                 /*  006 - 011   */
```

```
          1, 1, 1, 1, 1, 1,                                   /*  012 - 017   */
          1, 1, 1, 1, 1, 1,                                   /*  018 - 023   */
          1, 1, 1, 1, 1, 1,                                   /*  024 - 029   */
          1, 1, 1, 1, 1, 1,                                   /*  030 - 035   */
          1, 1, 1, 1, 1, 1,                                   /*  036 - 041   */
          1, 1, 1, 1, 1, 1,                                   /*  042 - 047   */
          1, 1, 1, 1, 1, 1,                                   /*  048 - 053   */
          1, 1, 1, 1, 1, 1,                                   /*  054 - 059   */
          1, 1, 1, 1, 1, 1,                                   /*  060 - 065   */
          1, 1, 1, 1, 1, 1,                                   /*  066 - 071   */
          1, 1, 1, 1, 1, 1,                                   /*  072 - 077   */
          1, 1, 1, 1, 1, 1,                                   /*  078 - 083   */
          1, 1, 1, 1, 1, 1,                                   /*  084 - 089   */
          1, 1, 1                                             /*  090 - 092   */
        };
    };
    struct Table9 {
        int tbl_sca_sz = -3;                                  /* tbl_sca_sz   */
        int tbl_ele_sz = 93;                                  /* tbl_ele_sz   */
        int tbl_type = 2;                                     /* tbl_type     */
/*
 *  Table 9
 *  Dark count as determined by Gallileo Electro-Optics.
 *  In order to generate the dark count level, it is faked
 *  in the IDF world by signifing it dependent on a cal
 *  field.  Dark count levels are not dependent on cal values
 *  and represent a count at each sensor energy level.
 *  Zero value means that that energy value has not been
 *  defined. Dark counts represent the average counts that the
 *  CEM sensor will generate by spontaneous emission.
 */
        int tbl_var = -1;                                     /* tbl_var      */
        int tbl_expand = 0;                                   /* tbl_expand   */
        int crit_act_sz = 0;                                  /* crit_act_ele */
        int format [3] = {1, 1, 1};                           /* format       */
        int offset [3] = {0, 31, 62};                         /* offsets      */
        int scale [3] = {-6, -6, -6};                         /* scale factor */
        int values [93] = {                                   /* values       */
          957, 957, 957, 957, 957, 957,                       /*  000 - 005   */
          957, 957, 957, 957, 957, 957,                       /*  006 - 011   */
          957, 957, 957, 957, 957, 957,                       /*  012 - 017   */
          957, 957, 957, 957, 957, 957,                       /*  018 - 023   */
          957, 957, 957, 957, 957, 957,                       /*  024 - 029   */
          957, 957, 957, 957, 957, 957,                       /*  030 - 035   */
          957, 957, 957, 957, 957, 957,                       /*  036 - 041   */
          957, 957, 957, 957, 957, 957,                       /*  042 - 047   */
          957, 957, 957, 957, 957, 957,                       /*  048 - 053   */
          957, 957, 957, 957, 957, 957,                       /*  054 - 059   */
          957, 957, 11489, 11489, 11489, 11489,               /*  060 - 065   */
          11489, 11489, 11489, 11489, 11489, 11489,           /*  066 - 071   */
          11489, 11489, 11489, 11489, 11489, 11489,           /*  072 - 077   */
          11489, 11489, 11489, 11489, 11489, 11489,           /*  078 - 083   */
          11489, 11489, 11489, 11489, 11489, 11489,           /*  084 - 089   */
          11489, 11489, 11489                                 /*  090 - 092   */
        };
    };
```

```
    struct Table10 {
        int tbl_sca_sz = -3;                                /* tbl_sca_sz   */
        int tbl_ele_sz = 256;                               /* tbl_ele_sz   */
        int tbl_type = 0;                                   /* tbl_type     */
/*
 *  Table 10
 *  Amplifier Dead Time Correction Factor.  Multiply this
 *  number by count rate table value to correct the count for
 *  amplifier dead time.
 */
        int tbl_var = 0;                                    /* tbl_var      */
        int tbl_expand = 1;                                 /* tbl_expand   */
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [3] = {0, 0, 0};                         /* format       */
        int offset [3] = {0, 0, 0};                         /* offsets      */
        int scale [3] = {-6, -6, -6};                       /* scale factor */
        int values [256] = {                                /* values       */
          1000000, 1000004, 1000009, 1000013, 1000017, 1000022,
          1000026, 1000030, 1000035, 1000039, 1000044, 1000048,
          1000052, 1000057, 1000061, 1000065, 1000072, 1000076,
          1000081, 1000085, 1000089, 1000094, 1000098, 1000102,
          1000107, 1000111, 1000115, 1000120, 1000124, 1000128,
          1000133, 1000137, 1000144, 1000152, 1000161, 1000170,
          1000178, 1000187, 1000196, 1000205, 1000213, 1000222,
          1000231, 1000239, 1000248, 1000257, 1000266, 1000274,
          1000287, 1000305, 1000322, 1000340, 1000357, 1000374,
          1000392, 1000409, 1000427, 1000444, 1000462, 1000479,
          1000496, 1000514, 1000531, 1000549, 1000575, 1000610,
          1000645, 1000680, 1000714, 1000749, 1000784, 1000819,
          1000854, 1000889, 1000924, 1000959, 1000994, 1001029,
          1001064, 1001098, 1001151, 1001221, 1001291, 1001361,
          1001430, 1001500, 1001570, 1001640, 1001710, 1001780,
          1001850, 1001920, 1001990, 1002060, 1002131, 1002201,
          1002306, 1002446, 1002586, 1002727, 1002867, 1003008,
          1003148, 1003289, 1003429, 1003570, 1003711, 1003852,
          1003993, 1004134, 1004275, 1004416, 1004628, 1004910,
          1005193, 1005476, 1005759, 1006043, 1006326, 1006610,
          1006895, 1007179, 1007464, 1007749, 1008034, 1008319,
          1008605, 1008891, 1009320, 1009893, 1010468, 1011043,
          1011619, 1012196, 1012775, 1013354, 1013934, 1014515,
          1015097, 1015681, 1016265, 1016850, 1017436, 1018024,
          1018907, 1020087, 1021272, 1022461, 1023655, 1024852,
          1026054, 1027260, 1028470, 1029685, 1030904, 1032127,
          1033355, 1034587, 1035824, 1037065, 1038935, 1041445,
          1043973, 1046520, 1049087, 1051672, 1054277, 1056902,
          1059547, 1062212, 1064898, 1067605, 1070333, 1073082,
          1075854, 1078647, 1082880, 1088604, 1094423, 1100339,
          1106356, 1112476, 1118703, 1125041, 1131492, 1138061,
          1144752, 1151568, 1158514, 1165596, 1172816, 1180182,
          1191513, 1207180, 1223533, 1240630, 1258534, 1277321,
          1297073, 1317887, 1339877, 1363171, 1387925, 1414321,
          1442580, 1472970, 1505823, 1541557, 1601779, 1698837,
          1826153, 2013567, 2431803, 0, 0, 0,
          0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0,
```

```
          0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0,
          0, 0, 0, 0
        };
    };
    struct Table11 {
        int tbl_sca_sz = 0;                                 /* tbl_sca_sz   */
        int tbl_ele_sz = 6;                                 /* tbl_ele_sz   */
        int tbl_type = 1;                                   /* tbl_type     */
/*
 *  ASCII definitions of the status states
 */
        int tbl_var = 4;                                    /* tbl_var      */
        int tbl_expand = 0;                                 /* tbl_expand   */
        int crit_act_sz = 0;                                /* crit_act_ele */
        int format [2] = {0, 0};                            /* format       */
        int offset [2] = {0, 4};                            /* offsets      */
        string values [6] = {                               /* values       */
          "N along -X",  "N along +X",  "S along -X",       /*  000 – 002   */
          "S along +X", "low",  "high"                      /*  003 - 005   */
        };
    };
    struct Constant0 {
        int id = 6;                                         /* const_id     */
/*
 *  Constant 0
 *  Axis A component of aperture normals with respect to the PEM
 *  magnetometer (VMAG).
 */
        int scale [3] = {-6, -6, -6};                       /* scale factor */
        int values [3] = {-22690, -17188, 5581};            /* values       */
    };
    struct Constant1 {
        int id = 7;                                         /* const_id     */
/*
 *  Constant 1
 *  Axis B component of aperture normals with respect to the PEM
 *  magnetometer (VMAG).
 */
        int scale [3] = {-6, -6, -6};                       /* scale factor */
        int values [3] = {804688, 398922, -366949};         /* values       */
    };
    struct Constant2 {
        int id = 8;                                         /* const_id     */
/*
 *  Constant 2
 *  Axis C component of aperture normals with respect to the PEM
 *  magnetometer (VMAG).
 */
        int scale [3] = {-6, -6, -6};                       /* scale factor */
        int values [3] = {593265, 916824, 930224};          /* values       */
    };
}
```

9. HEADER FILE

The IDFS header file holds data which, for the most part, is slowly varying in time and need not be repeated every data record. Each IDFS data record points to a header record which contains its slowly varying information. Header records have variable length. Their format is shown below in the form of a C data structure.

```
struct
 {
      short           hdr_len;                  /*2-byte integer*/
      short           year;                     /*2-byte integer*/
      short           day;                      /*2-byte integer*/
      char            time_units;               /*1-byte quantity*/
      unsigned char   i_mode;                   /* unsigned 1-byte quantity */
      long            data_accum;               /* 4-byte integer */
      long            data_lat;                 /* 4-byte integer */
      long            swp_reset;                /* 4-byte integer */
      long            sen_reset;                /* 4-byte integer */
      short           n_sen;                    /* 2-byte integer */
      unsigned short  n_sample;                 /* unsigned 2-byte integer */
      short           scan_index[1 or n_sample]; /* array of 2-byte integers */
      short           sensor_index[n_sen];      /* array of 2-byte integers */
      unsigned char   d_qual[n_sen];            /* array of unsigned 1-byte quantities */
      unsigned char   mode_index[i_mode];       /* array of unsigned 1-byte quantities */
};
```

Each of the fields within the header record is described in the following sections. Fields are grouped together into logical blocks much the same as it is done for the VIDF field descriptions.

9.1 HEADER RECORD SIZE

Header records are not fixed in length. The length in bytes of each header record is found in the first 2 bytes of the record. The software reads the first two bytes to determine how many bytes to read to complete the access of the header record.

9.1.1 hdr_len

This is a 2-byte integer which gives the size of the header record in bytes. The length includes the 2 bytes holding the **hdr_len** variable.

9.2 HEADER RECORD TIME Information

Each header record contains the slowly varying time elements of when the data, which points to it, was obtained. These are contained in the following two fields. The change of any value necessitates the writing of a new header record.

9.2.1 year

This is a 2-byte integer that contains the year during which the data was obtained. This value contains the 4-digit representation of the year (e.g., 1985) instead of the traditional 2-digit representation (e.g., 85).

9.2.2 day

A 2-byte integer that contains the day of year during which the data was obtained. The day of year begins at day 1 and runs to 365 for non-leap years and 366 for leap years.

9.3 HEADER INSTRUMENT TIMING Information

Each header record contains a set of times which describe the current accumulation times of the virtual instrument. These times are found in several fields and are used to correctly time tag the given data values found within the data records. These are contained in the following five fields. The change of any value necessitates the writing of a new header record.

9.3.1 time_units

This field is a 1-byte quantity that together with the **data_accum** field describes the time taken to acquire a single data measurement. This field contains the scaling given as a power of ten, which is used to take the value in the **data_accum** field to units of seconds. An example is shown in the **data_accum** section below. Since the finest resolution supported by the IDFS data access software is down to the nanosecond, **time_units** CANNOT be less than -9.

9.3.2 data_accum

This field is a 4-byte integer that together with the **time_units** field describes the time taken to acquire a single data measurement. This field contains an integer value which when scaled according to the value given in the **time_units** field gives the time in seconds required to acquire one data measurement. The conversion is given below.

$$ACC\_TIME = data\_accum * 10^{time\_units}$$

Note that for many scalar data sets, the accumulation time is set to 0 to indicate that the accumulation is instantaneous. In these cases, the **data_lat** field described below is used to give the time between successive measurements.

9.3.3 data_lat

This is a 4-byte integer which contains the number of microseconds of dead time between successive data acquisitions. This data field together with the data fields **data_accum** and **time_units** gives the total time between successive accumulations in seconds. This is computed as:

$$T = data\_accum * 10^{time\_units} + data\_lat * 10^{-6}$$

9.3.4 swp_reset

This field is a 4-byte integer that holds the number of microseconds of dead time between successive columns of data within the data matrix when time is advancing down the data columns or between successive data rows when time is advancing across the rows (the timing definition is described under the definition of **sen_mode** in the section describing the VIDF fields). It is used primarily with virtual instruments which contain vector sensors, especially those which require a finite dead time to reset themselves between successive sweeps. For many particle instruments this is equivalent to the flyback time. Note that if all of the sweeps within a sensor set are taken in parallel, the definition of this field and the **sen_reset** field described below become identical. Only one of the values should be set.

9.3.5 sen_reset

This is a 4-byte integer field that holds the number of microseconds of dead time between successive sensor sets of data. In many cases this field is redundant with the **swp_reset** field as described above. When this is the case only one of the fields should be set.

9.4 HEADER STATUS (MODE) Information

The header records contain all status (mode) information which is associated with the virtual instrument. Status information is commonly used to switch between sets of values in VIDF tables which are used to convert telemetry to various sets of units. The status information is held in two fields which are described below.

9.4.1 i_mode

This field is an unsigned 1-byte quantity that gives the number of status bytes contained within this header. This value is the length of the **mode_index** field and its value is identical with that given for the **status** field in the VIDF.

9.4.2 mode_index

This is a field of **i_mode** unsigned 1-byte elements, each of which contains the current state of one of the virtual instrument's defined status bytes. If there is no status (**i_mode** = 0) within the header record, this field is omitted from the header record.

9.5 HEADER SENSOR Information

The header records hold all of the sensor information for individual sensor sets. The sensor information is contained in three fields which are described below.

9.5.1 n_sen

This is a 2-byte integer field that gives the number of sensors containing data in the current sensor set. It also determines the number of columns within the sensor set matrix. The number of sensors in the header record (**n_sen**) cannot be greater than the number of sensors defined in the VIDF (see description of **sen**) for the virtual instrument.

9.5.2 sensor_index

This is a field of **n_sen** 2-byte integers which contains the sensor numbers in the order that they are written to the data records. The values are offsets into the **sen_name** field in the VIDF.

9.5.3 d_qual

This is a field of **n_sen** unsigned 1-byte quantities, where each holds the data quality flag associated with its corresponding sensor as listed in the **sensor_index** field. The values are offsets into the **qual_name** field in the VIDF.

9.6 HEADER SCAN Information

The header records hold all of the scan information for individual sensor sets. The scan information is contained in two fields which are described below.

9.6.1 n_sample

This is an unsigned 2-byte integer field that holds the number of data samples which are returned for each sensor within a sensor set of data. All sensors must return identical numbers of samples. This field determines the number of rows within the sensor set matrix. For vector sensors (**smp_id** = 1), **n_sample** is equivalent to the current scan length. For scalar sensors (**smp_id** = 2), **n_sample** is simply the number of successive measurements which are stacked in the sensor set.

9.6.2 scan_index

This is an array of 2-byte integers whose length is equivalent to the current scan length of the sensor. For scalar sensors (**smp_id** = 2), this is always 1. For vector sensors, the length is the value found in the header field **n_sample**. The values are offsets which are used to index into any VIDF tables which are a function of scan. In general, the value for a scalar virtual instrument has no meaning and must be set to zero.

10. DATA FILE

The data file is a series of fixed-length data records. Each data record consists of timing information (**dr_time**), spin information (**spin** and **sun_sen**), references to header records (**hdr_off** and **nss**) followed by the sensor and calibration data (**data_array**).

Data Record 1					Data Record 2					…
Data Info	Data Array				Data Info	Data Array				…
	Sen Set 1	Sen Set 2	…	Sen Set N		Sen Set 1	Sen Set 2	…	Sen Set N	…

Data storage in the IDFS data record is organized along the concept of sensor (primary) data, calibration (secondary) data and sensor sets. Sensor data is the basic, primary measurement identifier (object being studied). This data is placed in the **data_array** field first and can be visualized as a 2-dimensional matrix that has **n_sen** columns and **n_sample** rows, where **n_sen** and **n_sample** are obtained from the header record that is associated with the data being processed. In actuality, the sensor data is simply written as a sequence of 1-dimensional arrays that are **n_sample** in length, sensor by sensor. Calibration data is ancillary data which is necessary to interpret the primary data (e.g., automatic gain correction values). Not all virtual instruments have calibration data defined. Calibration data is placed in the **data_array** field after all sensor data has been written. This data can also be visualized as a 2-dimensional matrix that has **n_sen** columns and **num_cal** rows, where **n_sen** is obtained from the header record that is associated with the data being processed and **num_cal** is computed using the **cal_use** field as defined in the VIDF. These two data matrices, taken collectively, are what is referred to as an IDFS sensor set.

Sensor Set 1		Sensor Set 2		…
Primary Matrix	Secondary Matrix	Primary Matrix	Secondary Matrix	…

Since each sensor set can be described by a different header record, it is possible to have sensor set matrices that vary in size within the same data record. However, data records do not vary in size, but are fixed length given in bytes by the **data_len** field in the VIDF. This allows for rapid positioning in the data file.

Each record within the data file has the format shown below. The format is shown as the C data structure used for writing and reading data records.

```
struct
{
        long            dr_time;                /* 4-byte integer */
        long            spin;                   /* 4-byte integer */
        long            sun_sen;                /* 4-byte integer */
        long            hdr_off[max_nss];       /* array of 4-byte integers */
        long            nss;                    /* 4-byte integer */
        unsigned char   data_array[data_size];  /* array of unsigned 1-byte quantities */
};
```

Note that the data field is generically assigned the data type of unsigned character (8 bits) even though the data may be stored within the field with a base length of 8, 16, or 32 bits. The storage boundary used for individual data within a particular data file is determined from the VIDF (see **tdw_len**) and is used by the IDFS data access software to correctly unpack the data.

The following sections describe each of the fields within the data record structure.

10.1 DATA RECORD TIME Information

Each data record contains a single time field which is described below. It should be noted that when the data time duration spans multiple days, it will be necessary to record the data as multiple IDFS data records, where the duration of each IDFS record is no more than one day. This approach is necessary because IDFS allows the definition of data having time tags with very fine time resolution (i.e. accuracies to the nanoseconds resolution) rather than large time duration data.

10.1.1 dr_time

This 4-byte integer field holds the relative beginning time of day in milliseconds for the first data element of the first sensor set in the data record data field (**data_array**). From this single value together with the **sen_mode** and **time_off** fields in the VIDF and the timing fields in the header record, the absolute beginning acquisition time of any element within the data field can be obtained. The absolute beginning time of day in milliseconds for the first data element of the first sensor set in the **data_array** is found through the relationship:

TM = **dr_time** + **time_off**[**sensor_index**[0]]

where **time_off** is a field in the VIDF and **sensor_index** is a field in the IDFS header record. Note that time of day for subsequent data elements also depends on the **sen_mode** setting in the VIDF and the timing fields in the header record.

The absolute time of day value for the first data element of the first sensor set can be adjusted to a nanosecond precision by storing the nanosecond time adjustment factor within the first four bytes in **data_array** and by setting the VIDF field **nano_defined** to 1 (refer to section 5.2). Section 10.4 explains the layout of **data_array**. Note that the computation for **data_size** in section 10.4 does not include the 4 bytes utilized by the nanosecond time adjustment factor. The nanosecond time adjustment factor is interpreted as a signed, 4-byte quantity and therefore, can reach a maximum of 2,147,483,647. However, since this value contains the resolution between nanosecond and millisecond precision, the value should be no larger than 999,999 and no less than 0. The IDFS data access software returns time tags in 2 parts, one which is expressed in milliseconds of the day and one which contains the remaining nanoseconds of the day.

10.2 DATA RECORD SPIN Information

Each data record contains two fields which enable the spin phase of the data to be reconstructed. The descriptions of these fields are found below.

10.2.1 spin

This 4-byte integer field contains the azimuthal rate of rotation of the virtual instrument in milliseconds per revolution. If the virtual instrument is non-spinning then this field is set to zero. The spin rate can be either positive or negative. A positive rate of revolution is used when the phase of the instrument increases during the time covered by the data record and a negative rate of revolution is used when the phase of the instrument decreases during the time covered by the data record.

10.2.2 sun_sen

This field is a 4-byte integer that contains the time in milliseconds of day at which a predefined location crosses the azimuthal zero degree position. This location should be described in the comment field of the VIDF file pertaining to the virtual instrument.

If the instrument is non-spinning (**spin** = 0), then this field, if greater than or equal to zero, contains the azimuthal angular separation of the zero degree indicator from the zero degree position. The separation is given as degrees times 100. If the **sun_sen** value is negative then there is no angular information present in the data record. In the latter case, both this and the **spin** fields can be ignored.

On a spinning virtual instrument (**spin** != 0), the angular offset from the azimuthal zero degree position of any sensor J, at a time T, is found through the equation:

ANG = (T - **sun_sen**) / **spin** * 360 + ang_offset[J]

where ang_offset is the scaled angular offset value obtained from the VIDF (data if available is contained in a VIDF constant field with a **const_id** value of 2 and if absent the value is set to 0), and the **sun_sen** and **spin** are the fields described in this section.

10.3 DATA RECORD HEADER OFFSETS

Each data record has two fields which allow all sensor sets defined within the data record to be connected with one of the header records in the header file. The descriptions of these fields are found below.

10.3.1 nss

This is a 4-byte integer where the absolute value gives the number of sensor sets within the IDFS data record. If **nss** is positive then there is one header offset element in the **hdr_off** field per sensor set, and if **nss** is negative then only the first header offset element in the **hdr_off** array applies for all of the sensor sets (# sensor sets = |**nss**|) within the data record.

10.3.2 hdr_off

This field is an array of **max_nss** 4-byte integer elements where **max_nss** is a field defined in the VIDF file. Since the data records are fixed-size data records, this field must be allocated to accommodate the maximum number of sensor sets that can be returned in a single data record. The number of elements which are used in this array is given by the data field **nss**,

and each used element contains the byte offset into the IDFS header file for one of the data record sensor sets. When **nss** is negative, only the first **hdr_off** element contains valid data and all sensor sets utilize the same header record.

Negative **hdr_off** values are used to indicate end of file. This is handled by two separate values which have different meanings depending on whether the acquired data is being obtained under a real-time scenario or a post real-time (delayed time) scenario. The definitions are given in the table below.

NEGATIVE HEADER OFFSET DEFINITIONS		
HEADER OFFSET VALUE	REAL-TIME DEFINITION	POST REAL-TIME DEFINITION
-1	End Of Transmission	End Of File
-2	File Closed	End Of File

In the real-time scenario, an end of transmission (-1) means that the data reception is concluded (at least for the current time), the file is closed, and no further data files are being opened. A file closed (-2) means that the current file is being closed with data still being acquired and a new data file is being opened. A file closure may occur due to mode changes within the instrument, restrictions on file length, need to free up disk space, etc.

A data file is closed by writing a separate end-of-file record. When processing data, a partial record is sometimes generated at the end-of-file. This data must be written before the end-of-file record by indicating the actual number of sensor sets in that partial record in the **nss** field, placing the proper header offset values in the **hdr_off** array, and placing the proper sensor values in the **data_array**. The remainder of the **hdr_off** array and **data_array** should be filled with zero values. The final data record must be the end-of-file record which has the end-of-file character in the first location of the header offset array, **hdr_off**[0] = -2, and **nss** = 1. The remainder of the **hdr_off** array and the **data_array** are filled with zero values.

Under real-time operations, the end-of-file record is written to close out complete data files, **hdr_off**[0] = -2 as described above. However during signal processing, the data stream can be terminated indicating there is no more data to process. This case is indicated by writing a data record with **hdr_off**[0] = -1 and **nss** = 1 to indicate there is no more data from the telemetry stream.

10.4 DATA RECORD DATA ARRAY

All of the data contained within the data record is stored in a 1-dimensional array. The format of this array is described below. Note that the computation for **data_size** does not include the 4 bytes utilized by the nanosecond time adjustment factor (see section 10.1.1), if one is defined. The nanosecond time adjustment factor is written only once at the beginning of the data array since it applies to the first sensor set only (independent of the number of sensor sets defined).

10.4.1 data_array

This is an array of **data_size** 1-byte quantities that contains all of the data which is stored within the data record. The data is written as a series of sensor sets with each sensor set being a 2-dimensional matrix of primary sensor data followed by a 2-dimensional matrix of sensor calibration (secondary) data.

Sensor Set 1		Sensor Set 2		…
Primary Matrix	Secondary Matrix	Primary Matrix	Secondary Matrix	…

Data for a given sensor are stored down a column in the data matrices. The header record corresponding to a given sensor set identifies which sensor's data is stored in which column through the **sensor_index** field. In the linear data array, matrices are written column by column (or sensor by sensor).

Primary Matrix				Secondary Matrix				…
Sensor 1 Data	Sensor 2 Data	…	Sensor N Data	Sensor 1 Cal Sets	Sensor 2 Cal Sets	…	Sensor N Cal Sets	…

Also, the description for **sen_mode** in the VIDF section has examples of sensor sets which help visualize a sensor set as a 2-dimensional array.

For a vector virtual instrument, the number of rows for the sensor (primary) data matrix is equivalent to the number of sensor scan steps while for a scalar virtual instrument, the number of matrix rows is the number of successive measurements stored within the sensor set. The number of rows in the calibration (secondary) matrix is equivalent to the number of calibration elements stored for each of the calibration sets defined in the VIDF. The number of elements can be computed from the VIDF **cal_use** field. The calibration data is stored in the order in which they are defined in the VIDF (cal set 0 data first followed by cal set 1 data, etc.) per sensor returned.

The IDFS data access software require data alignment in the **data_array** based upon the word length of the sensor data AND the calibration data, if any is defined. That is, the VIDF fields **tdw_len** and **cal_wlen** are examined by the IDFS read routines and the *largest value of all entries* is used as the base size for all sensor and calibration values. For example, if you have four sensors and their corresponding word lengths (**tdw_len**) are 2, 8, 2, and 8, respectively, then sensors 0 and 2 are 2 bits, and sensors 1 and 3 are 8 bits. Thus, in storing the data into the **data_array**, a base size of 8 bits MUST be used for all values. Therefore, 1 byte is written per sensor data value. Each 2-bit value is placed in a separate 1-byte quantity. When the data is actually extracted from the byte, the **tdw_len** value for the individual sensor is used, and therefore, bits 0-1 are returned as the data value and bits 2-7 are ignored for sensors 0 and 2 in the example.

Now, if there are sensors that are ALL less than or equal to 4 bits, multiple quantities can be "packed" into a single byte. Bit lengths less than 8 bits are closest packed into an 8-bit word (or byte). Data which is 1 bit in length is packed 8 per byte, data which is 2 bits is packed 4 per byte, while 3 and 4 bits are packed 2 per byte. Any larger bit lengths are packed 1 per byte.

Bit values are packed starting at the least significant bit of the bit "chunk". For example, if the maximum bit length is 3 (**tdw_len** = 3) and the IDFS has 1, 2, and 3-bit quantities, then the 3-bit quantities are placed in bits 0-2 and 4-6 (4-bit alignment) of a single byte, the 2-bit quantities are placed in bits 0-1 and 4-5 (again 4-bit alignment), and the 1-bit quantities are placed in bits 0 and 4 (4-bit alignment again). The bit alignments are powers of 2 -- 1, 2, 4, ..., 32 (see definition for **tdw_len** in the VIDF section).

In combining data quantities that are more than 8 bits in length, keep in mind that the IDFS system works on 1, 2 and 4 byte alignments.

Given the information above, the **data_size** of the **data_array** is determined by the following steps:

1. Determine the bit alignment, **bit_align**, by rounding the *largest of the* **tdw_len** *AND* **cal_wlen** *values* defined in the VIDF to one of the above-mentioned bit alignment values (must be power of 2). From **bit_align**, the number of quantities per byte is determined as:

$$Vals\_per\_byte = \frac{8}{bit\_align}$$

   Note that valid **Vals_per_byte** values are 8 (all 1-bit quantities), 4 (no greater than 2-bit quantities), 2 (4-bit quantities or less), 1 (8-bit quantities), 0.5 (16-bit or 2-byte quantities), and 0.25 (32-bit or 4-byte quantities). Likewise, the number of bytes per value is:

$$Bytes\_per\_val = \frac{bit\_align}{8}$$

   Note that valid **Bytes_per_val** values are 0.125 (all 1-bit quantities), 0.25 (2-bit quantities), 0.5 (4-bit quantities), 1 (8-bit quantities), 2, and 4.

2. Determine the number of sensor (primary) values in the sensor set by:

$$Num\_Sen\_Vals = \textbf{n\_sample} * \textbf{sen}$$

   where **n_sample** and **sen** are defined in the header record.

3. The total number of bytes in a primary sensor set is then:

$$Num\_Sen\_Bytes = Num\_Sen\_Vals * Bytes\_per\_val$$

   OR

$$Num\_Sen\_Bytes = \frac{Num\_Sen\_Vals}{Vals\_per\_byte}$$

where **Num_Sen_Bytes** is rounded UP to the nearest integer value. For example, if there are 30 values (**Num_Sen_Vals** = 30) and the values are 2-bit quantities (**Bytes_per_val** = 0.25 and **Vals_per_byte** = 4), then **Num_Sen_Bytes** is computed to be 7.5 which should be rounded up to give 8 bytes needed for one sensor set.

4. The number of bytes within the secondary or calibration data is computed in two steps. First, the number of elements within each calibration set is determined. For a calibration set J this is determined from one of the following three equations:

   if (**cal_use[J]** == 0)

   $$Num\_Cal\_Vals[J] = 1 * \mathbf{sen}$$

   if (**n_sample** % **cal_use[J]** == 0)

   $$Num\_Cal\_Vals[J] = \frac{\mathbf{n\_sample}}{\mathbf{cal\_use[J]}} * \mathbf{sen}$$

   if (**n_sample** % **cal_use[J]** != 0)

   $$Num\_Cal\_Vals[J] = \left(\frac{\mathbf{n\_sample}}{\mathbf{cal\_use[J]}} + 1\right) * \mathbf{sen}$$

   where **sen** and **n_sample** are defined in the header record and **cal_use** is defined in the VIDF file.

5. The total number of calibration elements within the sensor set is then given by:

   $$Tot\_Cal\_Vals = \sum_{i=0}^{i=\text{cal\_set}-1} Num\_Cal\_Vals[i]$$

   where **cal_sets** is the number of calibration sets as defined in the VIDF.

6. The total number of bytes in a secondary (calibration) sensor set is then:

   Num_Cal_Bytes = Tot_Cal_Vals * Bytes_per_val

   OR

   $$Num\_Cal\_Bytes = \frac{Tot\_Cal\_Vals}{Vals\_per\_byte}$$

   where **Num_Cal_Bytes** is rounded UP to the nearest integer value as is done for **Num_Sen_Bytes** as described above.

7. Finally, the number of bytes within a sensor set is:

$$Tot\_SenSet\_Bytes = Num\_Sen\_Bytes + Num\_Cal\_Bytes$$

AND thus, the **data_size** is computed by:

$$\textbf{data\_size} = Tot\_SenSet\_Bytes * \textbf{max\_nss}$$

where **max_nss** is the maximum number of sensor sets allowed in a data record as defined in the VIDF.

CONTENTS

LIST OF FIGURES